

Tru64 UNIX

ネットワーク・プログラミング・ガイド

Part Number: AA-RK3QE-TE

2002 年 11 月

ソフトウェア・バージョン: Tru64 UNIX V5.1B 以降

本書では、HP Tru64 UNIX のネットワーク・プログラミング環境について説明します。システム・コール、ヘッダ・ファイル、ライブラリ、ソフトウェア・ブリッジに関する情報を含め、ソケット・フレームワークおよび STREAMS フレームワークについて説明します。ソフトウェア・ブリッジは、ソケット・プログラムが STREAMS ドライバを使用でき、STREAMS プログラムが BSD ベースのドライバを使用できるようにします。また、X/Open トランスポート・インタフェース (XTI) を使用するプログラムの作成方法、およびソケット・ベースのアプリケーションを、XTI を使用するアプリケーションにポータリングする方法についても説明します。

© 2002 日本ヒューレット・パッカード株式会社

本書の著作権は日本ヒューレット・パッカード株式会社が保有しており、本書中の解説および図、表は日本ヒューレット・パッカードの文書による許可なしに、その全体または一部を、いかなる場合にも再版あるいは複製することを禁じます。

日本ヒューレット・パッカードは、弊社または弊社の指定する会社から納入された機器以外の機器で対象ソフトウェアを使用した場合、その性能あるいは信頼性について一切責任を負いかねます。

本書に記載されている事項は、予告なく変更されることがありますので、あらかじめご承知おきください。万一、本書の記述に誤りがあった場合でも、弊社は一切その責任を負いかねます。

本書で解説するソフトウェア(対象ソフトウェア)は、所定のライセンス契約が締結された場合に限り、その使用あるいは複製が許可されます。

COMPAQ, Compaq ロゴ, Digital ロゴは U.S. Patent and Trademark Office に登録されています。Alpha, AlphaServer, NonStop, TruCluster, および Tru64 は米国 Compaq Computer Corporation の商標です。

Microsoft, Windows および Windows NT は米国 Microsoft 社の登録商標です。Intel は米国 Intel 社の登録商標です。Motif, OSF/1, UNIX, The Open Group および X/Open は、The Open Group の米国ならびに他の国における商標です。

このドキュメントに記載されているその他の会社名および製品名は、各社の商標または登録商標です。

Portions of this document are adapted from 『A STREAMS-based Data Link Provider Interface – Version 2』 © 1991 UNIX International, Inc. Permission to use, copy, modify, and distribute this documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name UNIX International not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. UNIX International makes no representations about the suitability of this documentation for any purpose. It is provided “as is” without express or implied warranty.

原典: Network Programmer's Guide (AA-RH9UE-TE)
© 2002 Hewlett-Packard Company

目次

まえがき

1 ネットワーク・プログラミング環境の概要

1.1	データ・リンク・インタフェース	1-3
1.2	ソケット・フレームワークおよび STREAMS フレームワーク	1-4
1.3	X/Open トランスポート・インタフェース	1-6
1.4	eSNMP	1-8
1.5	RSVP アプリケーション・プログラミング・インタフェース	1-8
1.6	ソケットと STREAMS の間のやりとり	1-8
1.7	統合	1-11

2 データ・リンク・プロバイダ・インタフェース

2.1	通信モード	2-3
2.2	サービスの種類	2-4
2.2.1	ローカル管理サービス	2-5
2.2.2	コネクション・モード・サービス	2-5
2.2.3	コネクションレス・モード・サービス	2-6
2.2.4	肯定応答コネクションレス・モード・データ転送	2-7
2.3	DLPI アドレッシング	2-7
2.4	DLPI プリミティブ	2-9
2.5	利用できる PPA の識別	2-11

3 X/Open トランスポート・インタフェース

3.1	XTI の概要	3-3
3.2	XTI の機能	3-5
3.2.1	サービス・モードと実行モード	3-5

3.2.1.1	コネクション指向型サービスおよびコネクションレス型サービス	3-5
3.2.1.2	非同期実行および同期実行	3-6
3.2.2	XTI ライブラリ, TLI ライブラリ, およびヘッダ・ファイル	3-7
3.2.2.1	XTI および TLI ヘッダ・ファイル	3-8
3.2.2.2	XTI ライブラリ・コール	3-9
3.2.3	イベントおよび状態	3-12
3.2.3.1	XTI イベント	3-12
3.2.3.2	XTI の状態	3-15
3.2.4	XTI イベントのトラッキング	3-17
3.2.4.1	発信イベント	3-17
3.2.4.2	着信イベント	3-18
3.2.5	XTI 関数, イベント, および状態のマップ	3-19
3.2.6	複数のプロセスおよび終端の同期	3-22
3.3	XTI の使用	3-23
3.3.1	関数の呼び出し順序のガイドライン	3-23
3.3.2	トランスポート・プロバイダによる状態の管理	3-25
3.3.3	コネクション指向型アプリケーションの作成	3-26
3.3.3.1	終端の初期化	3-26
3.3.3.2	XTI のオプションの使用	3-28
3.3.3.3	接続の確立	3-29
3.3.3.4	データの転送	3-31
3.3.3.5	接続の解放	3-32
3.3.3.6	終端の初期化解除	3-34
3.3.4	コネクションレス型アプリケーションの作成	3-35
3.3.4.1	終端の初期化	3-35
3.3.4.2	データの転送	3-35
3.3.4.3	終端の初期化解除	3-37

3.4	フェーズに依存しない関数	3-37
3.5	XTI へのポート	3-39
3.5.1	プロトコル独立およびポータビリティ	3-39
3.5.2	XTI と TLI の互換性	3-41
3.5.3	ソケット・アプリケーションから XTI 使用アプリケーションへの書き換え	3-43
3.6	XPG3 , XNS4.0 , XNS5.0 の相違点	3-47
3.6.1	XPG3 と XNS4.0 の主な相違点	3-47
3.6.2	XNS4.0 と XNS5.0 の主な相違点	3-48
3.6.3	ソース・コードの移行	3-48
3.6.3.1	アプリケーションの古いバイナリ・ファイルを使用する場合	3-49
3.6.3.2	ソースを変更しないで再コンパイルする場合	3-49
3.6.3.3	XNS4.0 に準拠させる場合	3-49
3.6.3.4	XNS5.0 に準拠させる場合	3-49
3.6.4	バイナリの互換性	3-50
3.6.5	パッケージ	3-50
3.6.6	相互運用性	3-51
3.6.7	XTI のオプションの使用	3-51
3.6.7.1	XNS4.0 および XNS5.0 の XTI のオプションの使用 ..	3-51
3.6.7.2	XPG3 でのプロトコル・オプションの折衝	3-67
3.7	XTI エラー	3-67
3.8	XTI トランスポート・プロバイダの構成	3-68

4 ソケット

4.1	ソケット・フレームワークの概要	4-2
4.1.1	ソケットの通信プロパティ	4-3
4.1.1.1	ソケット抽象表現	4-3
4.1.1.2	通信ドメイン	4-4

4.1.1.3	ソケット・タイプ	4-5
4.1.1.4	ソケット名	4-7
4.2	アプリケーションとソケットとのインタフェース	4-7
4.2.1	通信モード	4-8
4.2.1.1	コネクション指向型通信	4-9
4.2.1.2	コネクションレス型通信	4-9
4.2.2	クライアント/サーバ・モデル	4-10
4.2.3	システム・コール, ライブラリ・コール, ヘッダ・ファイル, およびデータ構造体	4-10
4.2.3.1	ソケット・システム・コール	4-11
4.2.3.2	ソケット・ライブラリ・コール	4-12
4.2.3.3	ヘッダ・ファイル	4-19
4.2.3.4	ソケットに関連するデータ構造体	4-20
4.3	ソケットの使用方法	4-24
4.3.1	ソケットの作成	4-25
4.3.1.1	実行モードの設定	4-26
4.3.2	名前とアドレスのバインド	4-28
4.3.3	接続の確立	4-28
4.3.4	接続の受け入れ	4-30
4.3.5	ソケット・オプションの設定と取得	4-32
4.3.6	データの転送	4-33
4.3.6.1	read システム・コールの使用	4-34
4.3.6.2	write システム・コールの使用	4-34
4.3.6.3	send, sendto, recv および recvfrom システム・コール の使用	4-34
4.3.6.4	sendmsg および recvmsg システム・コールの使用 ...	4-36
4.3.7	ソケットのシャットダウン	4-38
4.3.8	ソケットのクローズ	4-38
4.4	インターネット・アプリケーションの作成	4-39

4.4.1	IPv4 アプリケーションの作成	4-39
4.4.2	IPv6 アプリケーションの作成	4-41
4.4.3	アドレス・テスト・マクロ	4-48
4.5	BSD ソケット・インタフェース	4-49
4.5.1	可変長ネットワーク・アドレス	4-50
4.5.2	プロトコル・データとユーザ・データの受信	4-51
4.6	一般的なソケット・エラー	4-53
4.7	高度なソケット・プログラミング情報	4-54
4.7.1	AF_INET6 ソケットを使用するための, アプリケーション の移植	4-54
4.7.1.1	名前の変更	4-56
4.7.1.2	構造体の変更	4-56
4.7.1.3	関数呼び出しの変更	4-59
4.7.1.4	アプリケーションの他の変更	4-62
4.7.1.5	ipv6_sniff ユーティリティの使用	4-64
4.7.2	IPv6 raw ソケットの使用	4-65
4.7.2.1	ICMPv6 メッセージへのアクセス	4-66
4.7.2.2	IPv6 ヘッダへのアクセス	4-67
4.7.2.3	IPv6 ルーティング・ヘッダへのアクセス	4-69
4.7.2.4	IPv6 オプション・ヘッダへのアクセス	4-71
4.7.3	特定のプロトコルの選択	4-74
4.7.4	名前とアドレスのバインド	4-75
4.7.4.1	ワイルドカード・アドレスへのバインド	4-75
4.7.4.2	UNIX ドメインにおけるバインド	4-76
4.7.5	帯域外データ	4-77
4.7.6	インターネット・プロトコル・マルチキャスト	4-80
4.7.6.1	IPv4 マルチキャスト・データグラムの送信	4-81
4.7.6.2	IPv4 マルチキャスト・データグラムの受信	4-84
4.7.6.3	IPv6 マルチキャスト・データグラムの送信	4-85

4.7.6.4	IPv6 マルチキャスト・データグラムの受信	4-88
4.7.7	ブロードキャストとネットワーク構成の調査	4-89
4.7.8	inetd デーモン	4-93
4.7.9	入出力の多重化	4-94
4.7.10	割り込み駆動のソケット I/O	4-97
4.7.11	シグナルおよびプロセス・グループ	4-98
4.7.12	擬似端末	4-100

5 Tru64 UNIX STREAMS

5.1	STREAMS フレームワークの概要	5-2
5.1.1	STREAMS の構成要素の概説	5-3
5.1.2	ioctl プロセス	5-7
5.2	STREAMS に対するアプリケーション・インタフェース	5-8
5.2.1	ヘッダ・ファイルおよびデータ型	5-8
5.2.2	STREAMS 関数	5-9
5.2.2.1	open 関数	5-9
5.2.2.2	close 関数	5-9
5.2.2.3	read 関数	5-10
5.2.2.4	write 関数	5-10
5.2.2.5	ioctl 関数	5-10
5.2.2.6	mkfifo 関数	5-10
5.2.2.7	pipe 関数	5-11
5.2.2.8	putmsg および putpmsg 関数	5-11
5.2.2.9	getmsg および getpmsg 関数	5-12
5.2.2.10	poll 関数	5-12
5.2.2.11	isastream 関数	5-12
5.2.2.12	fattach 関数	5-13
5.2.2.13	fdetach 関数	5-13
5.3	カーネル・レベル関数	5-16

5.3.1	モジュール・データ構造体	5-16
5.3.2	メッセージ・データ構造体	5-17
5.3.3	ドライバおよびモジュールの STREAMS 処理ルーチン ...	5-19
5.3.3.1	オープンおよびクローズの処理	5-19
5.3.3.2	構成処理	5-21
5.3.3.3	読み取り側ブットおよび書き込み側ブットの処理	5-21
5.3.3.4	読み取り側サービスおよび書き込み側サービスの処理	5-22
5.3.4	Tru64 UNIX STREAMS の概念	5-23
5.3.4.1	同期	5-23
5.3.4.2	タイムアウト	5-25
5.4	Tru64 UNIX カーネル内へのユーザ作成の STREAMS ベース のモジュールまたはドライバの組み込み	5-25
5.5	デバイス特殊ファイル	5-30
5.6	エラーとイベントのロギング	5-32

6 拡張 SNMP アプリケーション・プログラミング・インタフェース

6.1	eSNMP の概要	6-2
6.1.1	eSNMP の構成要素	6-2
6.1.2	アーキテクチャ	6-3
6.1.3	SNMP バージョン	6-4
6.1.4	AgentX	6-5
6.2	拡張 SNMP アプリケーション・プログラミング・インタ フェースの概要	6-5
6.2.1	MIB サブツリー	6-8
6.2.2	オブジェクト・テーブル	6-10
6.2.2.1	subtree_tbl.h ファイル	6-10
6.2.2.2	subtree_tbl.c ファイル	6-13
6.2.3	サブエージェントの実現	6-15
6.2.4	サブエージェント・プロトコルの操作	6-20

6.2.4.1	操作の順序	6-20
6.2.4.2	関数の戻り値	6-21
6.3	拡張 SNMP アプリケーション・プログラミング・インタ フェース	6-24
6.3.1	呼び出しインタフェース	6-24
6.3.1.1	esnmp_init ルーチン	6-25
6.3.1.2	esnmp_allocate ルーチン	6-26
6.3.1.3	esnmp_deallocate ルーチン	6-33
6.3.1.4	esnmp_register ルーチン	6-36
6.3.1.5	esnmp_unregister ルーチン	6-39
6.3.1.6	esnmp_register2 ルーチン	6-40
6.3.1.7	esnmp_unregister2 ルーチン	6-45
6.3.1.8	esnmp_capabilities ルーチン	6-47
6.3.1.9	esnmp_uncapabilities ルーチン	6-47
6.3.1.10	esnmp_poll ルーチン	6-48
6.3.1.11	esnmp_are_you_there ルーチン	6-49
6.3.1.12	esnmp_trap ルーチン	6-50
6.3.1.13	esnmp_term ルーチン	6-51
6.3.1.14	esnmp_systime ルーチン	6-51
6.3.2	メソッド・ルーチン呼び出しインタフェース	6-52
6.3.2.1	*_get ルーチン	6-53
6.3.2.2	*_set メソッド・ルーチン	6-55
6.3.2.3	メソッド・ルーチンのアプリケーションでのプログラ ミング	6-61
6.3.3	libsnmp サポート・ルーチン	6-65
6.3.3.1	o_integer ルーチン	6-66
6.3.3.2	o_octet ルーチン	6-68
6.3.3.3	o_oid ルーチン	6-69
6.3.3.4	o_string ルーチン	6-71

6.3.3.5	str2oid ルーチン	6-72
6.3.3.6	sprintoid ルーチン	6-73
6.3.3.7	instance2oid ルーチン	6-73
6.3.3.8	oid2instance ルーチン	6-75
6.3.3.9	inst2ip ルーチン	6-76
6.3.3.10	cmp_oid ルーチン	6-79
6.3.3.11	cmp_oid_prefix ルーチン	6-80
6.3.3.12	clone_oid ルーチン	6-80
6.3.3.13	free_oid ルーチン	6-81
6.3.3.14	clone_buf ルーチン	6-82
6.3.3.15	mem2oct ルーチン	6-83
6.3.3.16	cmp_oct ルーチン	6-84
6.3.3.17	clone_oct ルーチン	6-84
6.3.3.18	free_oct ルーチン	6-85
6.3.3.19	free_varbind_data ルーチン	6-86
6.3.3.20	set_debug_level ルーチン	6-86
6.3.3.21	is_debug_level ルーチン	6-88
6.3.3.22	ESNMP_LOG ルーチン	6-89

7 RSVP アプリケーション・プログラミング・インタフェース

7.1	ネットワークのサービス品質	7-1
7.2	ネットワークのサービス品質の構成要素	7-2
7.2.1	Traffic Control	7-2
7.2.2	RSVP	7-3
7.2.3	RAPI	7-3
7.2.4	構成要素の相互運用	7-3
7.3	Traffic Control	7-5
7.4	RSVP	7-6
7.4.1	RSVP の構成要素	7-7
7.4.2	rsvdpd デーモン	7-7

7.5	RSVP アプリケーション・プログラミング・インタフェース .	7-8
7.5.1	サポートされているルーチン	7-8
7.5.2	RAPI 対応アプリケーションの作成	7-9
7.5.2.1	アプリケーションのリンク	7-10
7.5.3	RAPI アプリケーションのデバッグ	7-10
8	Tru64 UNIX における STREAMS とソケットの共存	
8.1	STREAMS ドライバからソケット・プロトコル・スタックへのブリッジ	8-2
8.1.1	STREAMS ドライバ	8-3
8.1.1.1	ifnet STREAMS モジュールの使用	8-4
8.1.1.2	データ・リンク・プロバイダ・インタフェース・プリミティブ	8-10
8.2	BSD ドライバから STREAMS プロトコル・スタックへのブリッジ	8-11
8.2.1	サポートする DLPI プリミティブおよびメディア・タイプ	8-12
8.2.2	STREAMS 擬似ドライバの使用	8-13
A	STREAMS モジュール例	
B	ソケットおよび XTI プログラム例	
B.1	コネクション指向型プログラム	B-2
B.1.1	ソケット・サーバ・プログラム	B-2
B.1.2	ソケット・クライアント・プログラム	B-7
B.1.3	XTI サーバ・プログラム	B-10
B.1.4	XTI クライアント・プログラム	B-16
B.2	コネクションレス型プログラム	B-20
B.2.1	ソケット・サーバ・プログラム	B-20
B.2.2	ソケット・クライアント・プログラム	B-23
B.2.3	XTI サーバ・プログラム	B-27

B.2.4	XTI クライアント・プログラム	B-32
B.3	共通コード	B-36
B.3.1	common.h ヘッダ・ファイル	B-36
B.3.2	server.h ヘッダ・ファイル	B-38
B.3.3	serverauth.c ファイル	B-39
B.3.4	serverdb.c ファイル	B-43
B.3.5	xtierror.c ファイル	B-45
B.3.6	client.h ヘッダ・ファイル	B-46
B.3.7	clientauth.c ファイル	B-46
B.3.8	clientdb.c ファイル	B-48
 C IPv4 および IPv6 のソケット・プログラムの例		
C.1	AF_INET ソケットを使用するプログラム	C-1
C.1.1	AF_INET ソケットを使用するクライアント・プログラム	C-2
C.1.2	AF_INET ソケットを使用するサーバ・プログラム	C-5
C.2	AF_INET6 ソケットを使用するプログラム	C-7
C.2.1	AF_INET6 ソケットを使用するクライアント・プログラム	C-7
C.2.2	AF_INET6 ソケットを使用するサーバ・プログラム	C-11
C.3	プログラムの出力例	C-15
 D TCP 固有のプログラミング情報		
D.1	TCP のスループットとウィンドウ・サイズ	D-1
D.1.1	TCP ソケット・バッファ・サイズのプログラミング	D-2
D.1.2	TCP ウィンドウ・スケール・オプション	D-2
D.1.2.1	システム・ソケット・バッファ・サイズ制限の増加 ..	D-3
D.2	TCP の性能とエラー回復	D-3
D.3	TCP の性能と往復時間の計測	D-4
D.4	TCP の信頼性とシーケンス番号	D-4

E トークン・リング・ドライバの開発に関連する情報

E.1	ソース・ルーティング	E-1
E.2	キャノニカル・アドレス	E-3
E.3	データの境界合わせ	E-3
E.4	ドライバの softc 構造体のフィールドの設定	E-4

F データ・リンク・インタフェース

F.1	DLI プログラミングの前提条件	F-1
F.2	DLI 概略	F-2
F.2.1	DLI サービス	F-3
F.2.2	ハードウェア・サポート	F-3
F.2.3	DLI を使用してのローカル・エリア・ネットワークへのアクセス	F-4
F.2.4	高次レベル・サービスのインクルード	F-4
F.3	DLI ソケット・アドレスのデータ構造体	F-5
F.3.1	標準フレーム・フォーマット	F-5
F.3.2	sockaddr_dl 構造体の機能	F-7
F.3.3	イーサネット・サブ構造体	F-8
F.3.3.1	イーサネット・フレーム	F-9
F.3.3.2	イーサネット・サブ構造体の値の定義	F-9
F.3.4	802.2 サブ構造体	F-12
F.3.4.1	802 サブ構造体値の定義	F-13
F.4	DLI プログラムの作成	F-18
F.4.1	データ・リンク・サービスの提供	F-18
F.4.2	Tru64 UNIX システム・コール	F-18
F.4.3	ソケットの作成	F-19
F.4.4	ソケット・オプションの設定	F-20
F.4.5	ソケットのバインド	F-21
F.4.6	sockaddr_dl 構造体	F-21

F.4.6.1	アドレス・ファミリの指定	F-22
F.4.6.2	I/O デバイス ID の指定	F-22
F.4.6.3	サブ構造体タイプの指定	F-22
F.4.7	バッファ・サイズの計算	F-24
F.4.8	データ転送	F-24
F.4.9	ソケットの非アクティブ化	F-25
F.5	DLI プログラミング例	F-25
F.5.1	イーサネット・フォーマット・パケットを使用した DLI クライアント・プログラム例	F-26
F.5.2	イーサネット・フォーマット・パケットを使用した DLI サーバ・プログラム例	F-29
F.5.3	802.3 フォーマット・パケットを使用した DLI クライアント・プログラム例	F-32
F.5.4	802.3 フォーマット・パケットを使用した DLI サーバ・プログラム例	F-38
F.5.5	getsockopt および setsockopt を使用した DLI プログラム例	F-42

用語集

索引

例

5-1	サンプル・モジュール	5-26
B-1	コネクション指向型ソケット・サーバ・プログラム	B-3
B-2	コネクション指向型ソケット・クライアント・プログラム ...	B-7
B-3	コネクション指向型 XTI サーバ・プログラム	B-10
B-4	コネクション指向型 XTI クライアント・プログラム	B-16
B-5	コネクションレス型ソケット・サーバ・プログラム	B-20
B-6	コネクションレス型ソケット・クライアント・プログラム ...	B-24
B-7	コネクションレス型 XTI サーバ・プログラム	B-27

B-8	コネクションレス型 XTI クライアント・プログラム	B-32
B-9	common.h ヘッダ・ファイル	B-37
B-10	server.h ヘッダ・ファイル	B-38
B-11	serverauth.c ファイル	B-39
B-12	serverdb.c ファイル	B-43
B-13	xtierror.c ファイル	B-45
B-14	client.h ファイル	B-46
B-15	clientauth.c ファイル	B-47
B-16	clientdb.c ファイル	B-49
C-1	クライアント・スタブ・ルーチン	C-2
C-2	サーバ・スタブ・ルーチン	C-5
C-3	クライアント・スタブ・ルーチン	C-8
C-4	サーバ・スタブ・ルーチン	C-11
F-1	イーサネット用 sockaddr_dl 構造体の代入	F-23
F-2	802.2 用 sockaddr_dl 構造体の代入	F-23

図

1-1	ソケット・フレームワークおよび STREAMS フレームワーク	1-5
1-2	XTI, STREAMS, およびソケット間のやりとり	1-7
1-3	STREAMS ドライバとソケット・プロトコル・スタックのブリッジ	1-9
1-4	BSD ドライバと STREAMS プロトコル・スタックのブリッジ	1-10
1-5	ネットワーク・プログラミング環境	1-11
2-1	DLPI インタフェース	2-2
2-2	DLPI サービス・インタフェース	2-3
2-3	DLPI アドレスの識別構成要素	2-8
3-1	X/Open トランスポート・インタフェース	3-3
3-2	トランスポート終端	3-4
3-3	コネクション指向型トランスポート・サービスの状態遷移 ...	3-24

3-4	コネクションレス型トランスポート・サービスの状態遷移 ...	3-25
4-1	ソケット・フレームワーク	4-2
4-2	IPv4 通信での AF_INET ソケットの使用	4-40
4-3	IPv6 通信での AF_INET6 ソケットの使用	4-42
4-4	IPv4 通信での AF_INET6 ソケットの使用 (送信)	4-44
4-5	IPv4 通信での AF_INET6 ソケットの使用 (受信)	4-47
4-6	4.3BSD と 4.4BSD の sockaddr 構造体	4-51
4-7	4.3BSD , 4.4BSD , XNS4.0 , および POSIX 1003.1g の msgHdr 構造体	4-52
5-1	STREAMS フレームワーク	5-3
5-2	ストリームの例	5-4
8-1	ifnet STREAMS モジュール	8-3
8-2	DLPI STREAMS 擬似ドライバ	8-12
E-1	典型的なフレーム	E-4
F-1	DLI とネットワーク・プログラミング環境	F-3
F-2	イーサネット・フレーム・フォーマット	F-5
F-3	802.3 フレーム・フォーマット	F-5
F-4	FDDI フレーム・フォーマット	F-6
F-5	802.2 構造体	F-6

表

1-1	ネットワーク・プログラミング環境の構成要素	1-1
2-1	サポートされている DLPI プリミティブ	2-9
3-1	XTI および TLI のヘッダ・ファイル	3-8
3-2	XTI ライブラリ・コール	3-9
3-3	非同期 XTI イベント	3-13
3-4	非同期イベントおよび消費関数	3-14
3-5	TLOOK を返す XTI 関数	3-15
3-6	XTI の状態	3-16

3-7	発信 XTI イベント	3-17
3-8	着信 XTI イベント	3-19
3-9	コネクション指向型またはコネクションレス型トランスポート・サービスの初期化における状態遷移	3-20
3-10	コネクションレス型トランスポート・サービスにおける状態遷移	3-20
3-11	コネクション指向型トランスポート・サービスにおける状態遷移:パート 1	3-21
3-12	コネクション指向型トランスポート・サービスにおける状態遷移:パート 2	3-22
3-13	フェーズ独立関数	3-38
3-14	XTI 関数とソケット関数の比較	3-44
3-15	ソケットと XTI のメッセージの対応	3-46
4-1	UNIX 通信ドメインおよびインターネット通信ドメインの特性	4-5
4-2	ソケット・システム・コール	4-11
4-3	ソケット・ライブラリ・コール	4-17
4-4	ソケット・インタフェースのヘッダ・ファイル	4-19
4-5	アドレス・テスト・マクロの要約	4-49
4-6	一般的なエラーと診断	4-53
4-7	BSD ソケット API に対する IPv6 拡張の要約	4-55
4-8	名前の変更	4-56
4-9	IPv4 raw ソケットと IPv6 raw ソケットの相違点	4-65
4-10	ICMPv6 フィルタリング・マクロの要約	4-67
4-11	オプション情報とソケット・オプション	4-68
4-12	ルーティング・ヘッダ用のソケット呼び出し	4-69
4-13	オプション・ヘッダ用のソケット呼び出し	4-72
5-1	ioctl 処理の I_STR メソッドと透過メソッドの比較	5-7
5-2	STREAMS リファレンス・ページ	5-14
7-1	クライアントのライブラリ・サービス・ルーチン	7-8
7-2	RAPI フォーマット・ルーチン	7-9

F-1	DLI プログラムの呼び出し順序	F-19
F-2	DLI で使用されるデータ転送システム・コール	F-25



まえがき

本書では、X/Openトランスポート・インタフェース (XTI) 呼び出し、STREAMS I/O 呼び出し、および Berkeley Software Distribution (BSD) ソケット呼び出しを使用するプログラムの作成方法について説明します。XTI とソケットに関しては、概念およびプログラミングについて記述します。また、トランスポート層インタフェース (TLI) から XTI、およびソケットから XTI へのアプリケーションのポート方法についても説明します。STREAMS に関しては、Tru64 UNIX のインプリメンテーションと、AT&T System V Release 4 のインプリメンテーションとの相違点について説明します。また、eSNMP (Extensible System Network Management Protocol) アプリケーション・プログラミング・インタフェース、RSVP (Resource ReSerVation Protocol) アプリケーション・プログラミング・インタフェースおよび AF_INET6 ソケットについても説明します。

本書は次のことを目標としています。

- ネットワーク用に Tru64 UNIX で提供されているプログラミング・サポートを理解する。
- コネクション指向型サービス、またはコネクションレス型サービスのいずれかを使用して、XTI アプリケーションを作成する。
- STREAMS の Tru64 UNIX でのインプリメンテーションを理解する。
- ソケット・アプリケーションを作成する。
- TLI と XTI 間、およびソケットと XTI 間の相違点を理解する。
- eSNMP アプリケーションを作成する。

本書の対象読者

本書は、UNIX を使用してプログラミング経験のあるプログラマを対象としています。対象読者は次のことに習熟していることが前提となっています。

- C 言語
- UNIX オペレーティング・システムのプログラミング・インタフェース

- 開放型システム間相互接続 (OSI) の 7 層モデルを含む、基本的なネットワークの概念
- ネットワーク・アプリケーションの作成に必要な労力

追加および変更された機能

『ネットワーク・プログラミング・ガイド』の今回の改訂では、次の変更が行われました。

- 第 4 章が改訂されました。この章には、AF_INET6 ソケットをサポートするようにアプリケーションを変更する場合に、プログラマがソース・ファイルを走査して編集できるようにする新しいツールについての情報が追加されました。この章には、IPv6 raw ソケットと、そのソケット上で受信した情報にアクセスする方法について説明している新しい項も追加されました。

本書の構成

本書の構成は次のとおりです。

- | | |
|-------|--|
| 第 1 章 | XTI, STREAMS, ソケット, およびネットワーク・アプリケーションに必要なプログラミング・タスクの概要について説明します。 |
| 第 2 章 | データ・リンク・プロバイダ・インタフェース (DLPI) のサブセットをインプリメントする dlb 擬似ドライバについて説明します。 |
| 第 3 章 | XTI に関連する基本的な概念、コネクション指向型アプリケーションおよびコネクションレス型アプリケーションの作成方法、TLI に関する互換性の問題、および XTI へのアプリケーションのポート方法について説明します。XTI のエラーについても、この章で説明します。 |
| 第 4 章 | ソケット・インタフェースに関連する概念、およびソケット・アプリケーションの作成方法について説明します。 |
| 第 5 章 | STREAMS の Tru64 UNIX でのインプリメンテーションについて説明します。 |
| 第 6 章 | 拡張 SNMP アプリケーション・プログラミング・インタフェースについて説明します。 |
| 第 7 章 | Resource ReSerVation Protocol アプリケーション・プログラミング・インタフェースについて説明します。 |
| 第 8 章 | ifnet STREAMS モジュールと dlb STREAMS 擬似ドライバ通信ブリッジについて説明します。 |
| 付録 A | STREAMS モジュールの例です。 |

付録 B	ソケットおよび XTI のプログラムの例です。
付録 C	AF_INET および AF_INET6 ソケットのプログラムの例です。
付録 D	伝送制御プロトコル (TCP) 固有のプログラミング情報について説明します。
付録 E	トークン・リング・ドライバの開発に関連する情報について説明します。
付録 F	データ・リンク・インタフェース (DLI) についての説明とプログラムの例です。
用語集	ネットワーク機能に関する用語集です。

関連資料

Tru64 UNIX でのプログラミングについての一般的な情報は、『プログラミング・ガイド』を参照してください。

ネットワーキングの API についての詳細は、次のマニュアルを参照してください。

- 『*UNIX Network Programming, Networking APIs: Sockets and XTI*』 (ISBN 0-13-490012-X)
W. Richard Stevens:Prentice-Hall

XTI についての詳細は、次のマニュアルを参照してください。

- 『*X/Open Portability Guide Volume 7: Networking Services*』 (XPG3)
(ISBN 0-13-685892-9)
- 『*Application Environment Specification (AES) Operating System Programming Interfaces Volume*』 (ISBN 0-13-043522-8)
このマニュアルは Prentice-Hall から出版されており、必須の XTI 呼び出しがすべて記載されています。
- 『*X/Open CAE Specification: Networking Services (XNS), Issue 5*』
(XNS5.0) (ISBN 1-85912-165-9)

STREAMS I/O フレームワークについての詳細は、次のマニュアルを参照してください。

- 『*Programmer's Guide: STREAMS*』
Englewood Cliffs:Prentice-Hall, Inc., 1990.

このマニュアルでは、STREAMS を使用するアプリケーション、モジュール、およびデバイス・ドライバの作成方法について説明しています。

- 『*AT&T System V Release 4 Programmer's Reference Manual*』 Englewood Cliffs:Prentice-Hall, Inc., 1989.

このマニュアルには、STREAMS 用のプログラミング・インタフェースを含む、すべてのプログラミング・インタフェースについてのリファレンス・ページが記載されています。

- 『*AT&T System V Release 4 System Administrator's Reference Manual*』 Englewood Cliffs:Prentice-Hall, Inc., 1989.

このマニュアルには、STREAMS `ioctl` コマンドについてのリファレンス・ページが記載されています。

- 『*Transport Provider Interface (TPI) Specification*』 UNIX International
ソケット・インタフェースについての詳細は、次の書籍を参照してください。

- 『*Internetworking with TCP/IP: Principles, Protocols, and Architecture*』 Englewood Cliffs:Prentice-Hall, Inc., 1988 (Douglas Comer 著)

本書には、ソケット・インタフェースについて説明している章があります。

- 『*X/Open CAE Specification: Networking Services, Issue 5*』 (XNS5.0) (ISBN 1-85912-165-9)

- 『*Protocol Independent Interfaces P1003.1g Draft 6.6*』 IEEE ドラフト, 1997

- 『*Design and Implementation of the 4.3BSD UNIX Operating System*』 Reading:Addison-Wesley Publishing Company, 1989 (Leffler, McKusick, Karels, Quarterman 著)

本書には、ソケットの目的と使用方法についての記述があります。

ネットワーキング・インタフェースの管理についての詳細は、『システム管理ガイド』および『ネットワーク管理ガイド：接続編』を参照してください。

本書で使用する表記法

本書では次の表記法を使用しています。

% \$	パーセント記号は、C シェルのシステム・プロンプトを表します。ドル記号は、Bourne シェル、Korn シェル、および POSIX シェルの場合のシステム・プロンプトを表します。
#	番号記号は root としてログインした場合のシステム・プロンプトを表します。
% cat	対話式の例における太字(ボールド体)は、ユーザが入力する文字を示します。
<i>file</i>	イタリック体(斜体)は、変数値、プレースホルダ、および関数の引数名を示します。
[] { }	構文定義では、大カッコはオプションの項目を示し、中カッコは必須項目を示します。大カッコまたは中カッコの中の項目を縦線で区切っている場合は、そこに併記されている項目の中から1つの項目を選択することを示します。
...	構文定義では、水平の反復記号は、前の項目を1回以上繰り返して使用できることを示します。
cat(1)	リファレンス・ページの参照には、該当するセクション番号をカッコ内に示します。たとえば、cat(1) は、cat コマンドについての情報が、リファレンス・ページのセクション1に記載されていることを示します。
Return	四角で囲まれたキー名はユーザがそのキーを押すことを示します。
Ctrl/x	この記号は、スラッシュの前に指定されているキーを押しながら、スラッシュの後のキーまたはマウ

ス・ボタンを押すことを示します。例中では、このようなキーの組み合わせは、四角あるいは大カッコで囲まれて示されます(たとえば、`Ctrl/C`)。

ネットワーク・プログラミング環境の概要

ネットワーク・プログラミング環境には、アプリケーション、カーネル、およびドライバの開発者が、ネットワーク・アプリケーションを作成したり、ネットワーク・プロトコルをインプリメントするためのプログラミング・インタフェースが含まれます。さらに、アプリケーションがデータの処理および伝送に必要とする、カーネル・レベルのリソースも含まれます。これには、ライブラリ、データ構造体、ヘッダ・ファイル、トランスポート・プロトコルなどがあります。

この章では、ネットワーク・プログラミング環境の概要を説明します。特に、データ・リンク・インタフェースおよびアプリケーション・プログラミング・インタフェースが、ユーザ空間のアプリケーションから、カーネル空間のネットワーク層を介して、ネットワークとデータのやりとりをする方法について詳しく説明します。

インタフェースをサポートするカーネルのリソースについては、後の章で説明します。各章では、それぞれのインタフェースについて、特定のシステム・コール、ライブラリ・コール、データ構造体、およびその他のプログラミング上の留意点を説明します。

表 1-1 に、ネットワーク・プログラミング環境の主な構成要素をまとめます。

表 1-1: ネットワーク・プログラミング環境の構成要素

構成要素	インタフェース	説明
データ・リンク・インタフェース	データ・リンク・インタフェース (DLI)	プログラムがデータ・リンク層にアクセスして、他のシステム上の DLI プログラムと通信できるようにする。DLI は、ULTRIX との互換性のために、Tru64 UNIX で提供されている。付録 F を参照。

表 1-1: ネットワーク・プログラミング環境の構成要素 (続き)

構成要素	インタフェース	説明
	dlb インタフェース	STREAMS プロトコル・モジュールをターゲットとするカーネル・レベルのインタフェース。データ・リンク・サービスを使用または提供する。dlb STREAMS 擬似ドライバはデータ・リンク・プロバイダ・インタフェース (DLPI) のサブセットをインプリメントする。第 2 章, および /usr/share/doc/lib/dlpi ディレクトリにあるデータ・リンク・プロバイダ仕様 (dlpi.ps) を参照。DLPI 仕様にオンラインでアクセスするには, OSFPGMR ⁿⁿⁿ サブセットがインストールされていなければならない。
アプリケーション・プログラミング・インタフェース	ソケット	事実上の業界標準のプログラミグ・インタフェース。Tru64 UNIX では, 4.3BSD, 4.4BSD, XNS4.0, および POSIX 1003.1g Draft 6.6 ソケット・インタフェースをインプリメントする。TCP, UDP, IP, ARP, ICMP, および SLIP からなるインターネット・プロトコル群がソケット上にインプリメントされる。RFC 1200: 『IAB Protocol Standards』 および第 4 章を参照。
	STREAMS	デバイス・ドライバおよびネットワーク・プロトコル・スタックのインプリメンテーションをサポートするカーネル機能。STREAMS フレームワークは, カーネル・レベルとユーザ・レベル間, およびカーネル内の文字入出力用のインタフェース標準を定義する。Tru64 UNIX オペレーティング・システムは, AT&T System V Release 4.0 の STREAMS 互換バージョンを提供する。第 5 章を参照。
	XTI/TLI	プロトコルに依存しないトランスポート層アプリケーション・インタフェース。関数の集まりから構成される。XTI は, トランスポート層インタフェース (TLI), および開放型システム間相互接続 (OSI) モデルのトランスポート・サービス定義に基づく。第 3 章を参照。

表 1-1: ネットワーク・プログラミング環境の構成要素 (続き)

構成要素	インタフェース	説明
STREAMS とソケット間の通信ブリッジ	eSNMP	MIB (Management Information Bases) を作成することによって SNMP エージェント・プロセスの拡張を可能にするルーチン群。第 6 章を参照
	RSVP	拡張サービス品質 (QoS) を要求するアプリケーションを利用可能にする、業界標準のプロトコルおよびルーチン群。第 7 章を参照。
	ifnet STREAMS モジュール	STREAMS ベースのネットワーク・デバイス・ドライバが、Tru64 UNIX が提供するソケット・ベースの TCP/IP プロトコル・スタックにアクセスできるようにする。第 8 章を参照。
	d1b 擬似ドライバ	STREAMS ベースのプロトコル・スタックを使用するアプリケーションが、BSD ベースのドライバにアクセスできるようにする。d1b 擬似ドライバは DLPI 仕様のサブセットをインプリメントする。第 8 章を参照。

ネットワーク・プログラミング環境を理解するには、各構成要素を調べるのが最もわかりやすい方法です。以降の各節で、ネットワーク・プログラミング環境の構成要素について 1 つずつ紹介します。ネットワークに最も関連のある構成要素から始めて、次第にレベルを上げていきます。

1.1 データ・リンク・インタフェース

ネットワーク・プログラミング環境では、データ・リンク・インタフェース (DLI) およびデータ・リンク・プロバイダ・インタフェース (DLPI) の両方をサポートします。DLI を使用すると、ULTRIX システムで実行されるプログラムを Tru64 UNIX システムにポートできます。DLI についての詳細は、付録 F を参照してください。

DLPI は、OSI 参照モデルのデータ・リンク層にマップするカーネル・レベルのインタフェースです。DLPI を使用すると、ユーザはデータ・リンク・プロバイダの特性について特別な知識を持つ必要がなくなるため、特定の通信媒体に依存することなくこれらの特性をインプリメントできます。第 2 章では、DLPI、Tru64 UNIX の d1b 擬似ドライバ、およびサポートするプリミティブについてさらに詳しく説明します。

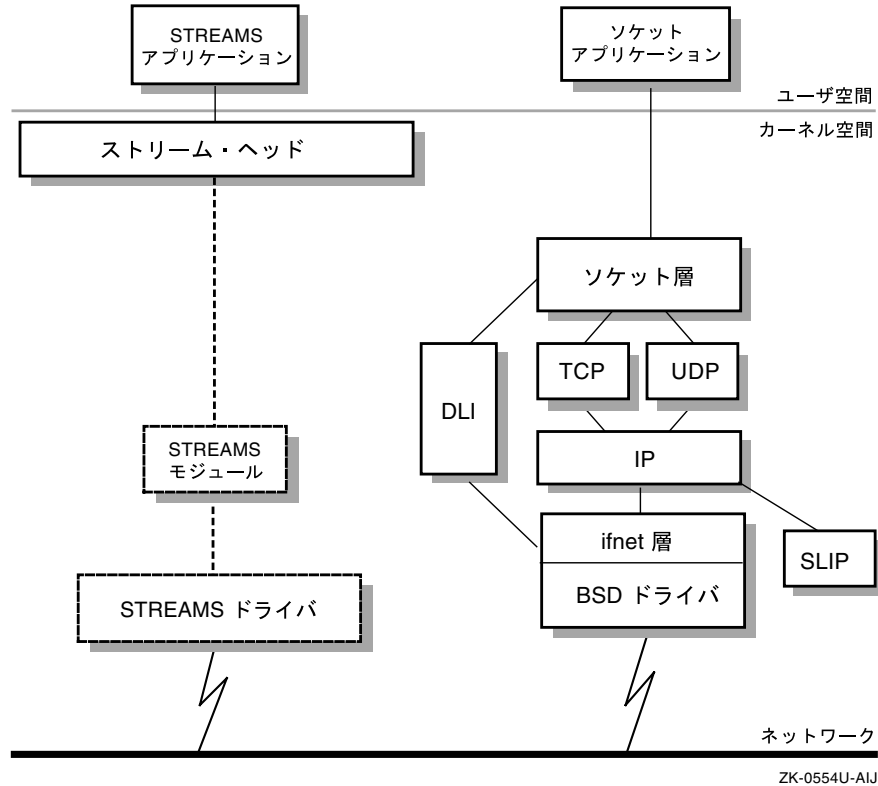
1.2 ソケット・フレームワークおよび **STREAMS** フレームワーク

Tru64 UNIX オペレーティング・システムは、ネットワーキング・アプリケーションの作成とカーネル・レベルのネットワーク入出力 (I/O) のために、AT&T System V Release 4 STREAMS フレームワークおよび BSD ソケット・フレームワークをサポートします。フレームワークは、特定のプログラミング・インタフェース、およびシステムがデータを送受信するために必要なカーネル・レベルのリソースから構成されます。

ソケットは、ネットワーキング・アプリケーションを作成するための、事実上の業界標準インタフェースです。ソケット・フレームワークは、BSD に基づいており、一連のシステム・コールとライブラリ・コール、ヘッダ・ファイルおよびデータ構造体からなります。アプリケーションは、ソケット・システム・コールを通じて、インターネット・プロトコル群などの、カーネル常駐ネットワーキング・プロトコルにアクセスできます。アプリケーションはまた、ソケット・ライブラリ・コールを使用して、ネットワーク情報を操作できます。たとえば、サービス名をサービス番号にマップしたり、着信データのバイト順を、ローカル・システムのアーキテクチャに適したものに変換することができます。

STREAMS フレームワークは、ソケットの代替機能を提供します。STREAMS インタフェースは、AT&T によって開発されました。システム・コール、カーネル・ルーチン、およびカーネル・ユーティリティから構成され、これらは、ネットワーキング・プロトコル群からデバイス・ドライバまで、あらゆるものをインプリメントするために使用されます。ユーザ空間のアプリケーションは、`open`、`close`、`putmsg`、`getmsg`、および `ioctl` などのシステム・コールを使用して、STREAMS フレームワークのカーネル部分にアクセスします。図 1-1 は、STREAMS フレームワークおよびソケット・フレームワークを示しています。

図 1-1: ソケット・フレームワークおよび STREAMS フレームワーク



注意

Tru64 UNIX では、STREAMS ベースのトランスポート・プロバイダをサポートはしていますが、提供していません (図 1-1 の点線の部分)。

ソケット・フレームワークでは、ユーザ空間のアプリケーションは、データを適切なソケット・システム・コールに引き渡します。次に、ソケット・システム・コールが、データをネットワーク層に引き渡します。最後に、ネットワーク層が ifnet 層を通して、BSD ドライバにデータを引き渡します。BSD ドライバはネットワークとデータのやりとりをします。

STREAMS フレームワークでは、ユーザ空間のアプリケーションは、データをストリーム・ヘッドに引き渡します。次に、ストリーム・ヘッドは、デー

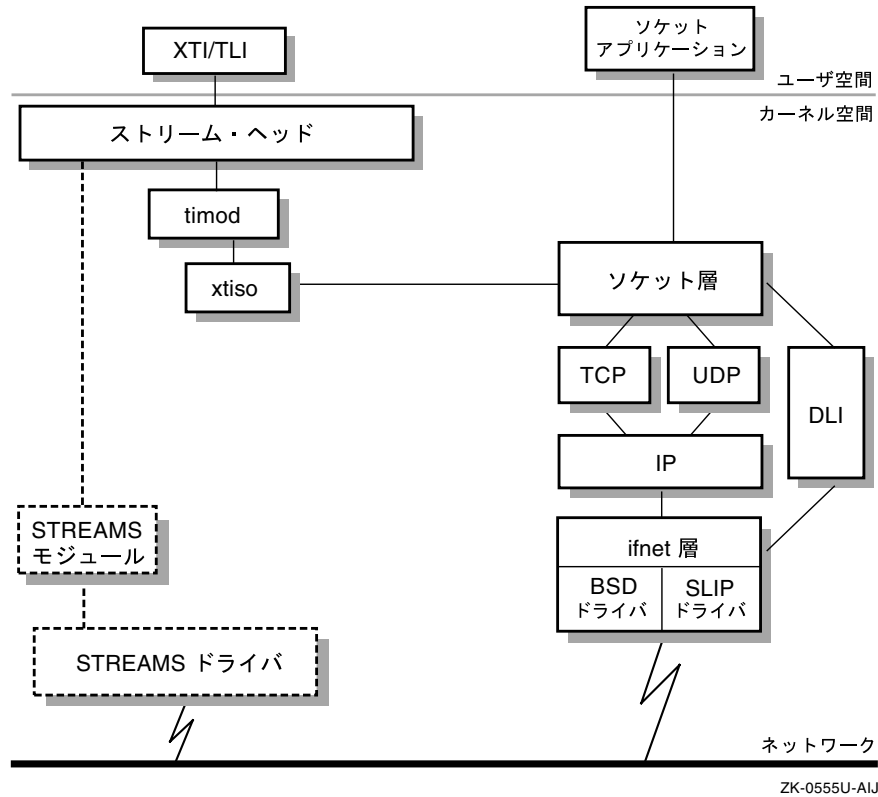
タを処理するために、ストリームにプッシュされている任意の STREAMS モジュールにデータを引き渡します。各モジュールは、STREAMS ドライバに到達するまで、次のモジュールにデータを引き渡します。STREAMS ドライバはネットワークとデータのやりとりをします。

1.3 X/Open トランスポート・インタフェース

X/Open トランスポート・インタフェース (XTI) は、トランスポート・プロバイダに依存しない、トランスポート層アプリケーション・インタフェースを定義します。つまり、XTI に合わせて作成されたプログラムは、伝送制御プロトコル (TCP) やユーザ・データグラム・プロトコル (UDP) などの、さまざまなトランスポート・プロバイダで実行できます。アプリケーションが、使用するトランスポート・プロバイダを指定します。

図 1-2 は、XTI と、STREAMS フレームワークおよびソケット・フレームワークとのやりとりを示しています。

図 1-2: XTI, STREAMS, およびソケット間のやりとり



アプリケーションによって指定されたトランスポート・プロバイダに応じて、データは、次の2つのパスのうち、どちらか1つを通ることができます。

1. STREAMS ベースのトランスポート・プロバイダが指定された場合

データは、STREAMS 上で実行するように作成されたアプリケーションの場合と同じ経路をたどります。データは、最初にストリーム・ヘッドをとり、次にアプリケーションがストリーム上にプッシュした任意のモジュールを通して、最後に STREAMS ドライバに引き渡されます。STREAMS ドライバはネットワークとデータのやりとりをします。

注意

Tru64 UNIX では、STREAMS ベースのトランスポート・プロバイダを提供していません。

2. ソケット・ベースのトランスポート・プロバイダ (TCP または UDP) が指定された場合

データは `timod` および `xtiso` を通じて引き渡されます。該当するソケット層ルーチンが呼び出され、データはインターネット・プロトコルおよび `ifnet` 層を通じて、BSD ベースのドライバに引き渡されます。BSD ベースのドライバはネットワークとデータのやりとりをします。

1.4 eSNMP

Tru64 UNIX SNMP エージェントは、eSNMP と呼ばれる拡張のためのフレームワークを提供します。SNMP デーモンは、拡張マスタ・エージェントとして機能し、eSNMP プロトコルによってさまざまなサブエージェントと通信を行います。マスタ・エージェントはシステム全体の代表として SNMP をインプリメントし、サブエージェントは、実際の MIB インストルメンテーションを提供します。eSNMP サブエージェント開発ツールおよび API は、マスタ・エージェントと通信し MIB を拡張するようなサブエージェントを開発するためのメカニズムをユーザに提供します。

1.5 RSVP アプリケーション・プログラミング・インタフェース

Tru64 UNIX での RSVP (Resource ReSerVation Protocol) の実装により、RSVP アプリケーション・プログラミング・インタフェース (RAPI) を使用して、特定のアプリケーションのデータ・ストリームやフローに対して拡張 QoS (サービス品質) を要求することができます。アプリケーションはこの要求を `rsvpd` デーモンに対して行います。このデーモンはその後、RSVP を使用して、ネットワークを通して RSVP メッセージを送受信します。`rsvpd` デーモンは、Traffic Control サブシステムとも通信し、指定されたネットワーク・インタフェース上のフローおよびフィルタのインストールや変更を行います。

1.6 ソケットと STREAMS の間のやりとり

`ifnet` STREAMS モジュールは、Tru64 UNIX の BSD ベースの TCP/IP を使用するプログラムが、STREAMS ベースのドライバにアクセスできるようにします。また、`d1b` 擬似ドライバを提供することによって、STREAMS ベースのプロトコル・スタックを使用するプログラムが、Tru64 UNIX で提供される BSD ベースのドライバにアクセスできるようにします。

図 1-3 は、オペレーティング・システムで提供される BSD ベースの TCP/IP を使用して、STREAMS ベースのドライバにアクセスするアプリケーションを示しています。

図 1-3: STREAMS ドライバとソケット・プロトコル・スタックのブリッジ

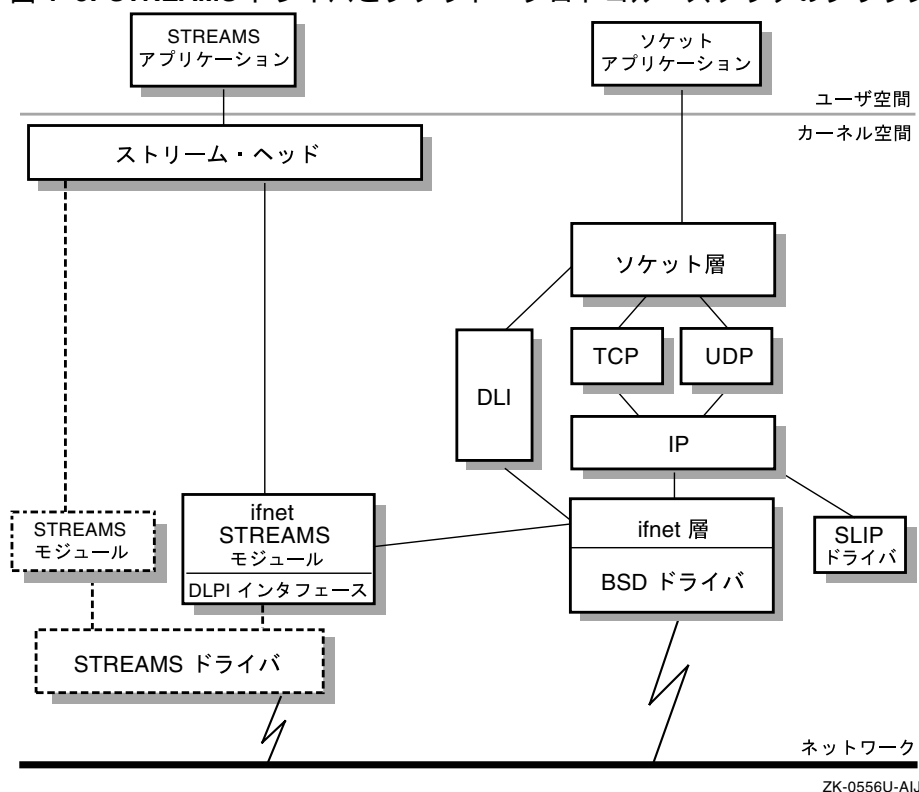


図 1-3 では、データは、ソケット・ベースのアプリケーションから適切なソケット・システム・コールを通じて流れ、インターネット・プロトコルによって処理されます。次に、BSD ifnet メッセージを DLPI にマップする機能を持つネットワーキング・サブシステムの BSD ifnet 層が、データを ifnet STREAMS モジュールに引き渡します。ifnet STREAMS モジュールは、データを処理して、STREAMS ドライバがネットワークとデータのやりとりをできるようにします。ソケット・ベースのアプリケーションに関する情報が返されると、STREAMS ドライバは、この情報をネットワークからピックアップして、ifnet STREAMS モジュールの DLPI インタフェースに引き渡します。ifnet STREAMS モジュールの DLPI インタフェースは、DLPI メッセージを BSD ifnet に変換して、BSD ifnet 層に

戻します。次に、データはインターネット・プロトコルによって処理され、アプリケーションに戻されます。

図 1-4 は、STREAMS ベースのプロトコル・スタックを使用して、BSD ベースのドライバにアクセスするアプリケーションを示しています。

図 1-4: BSD ドライバと STREAMS プロトコル・スタックのブリッジ

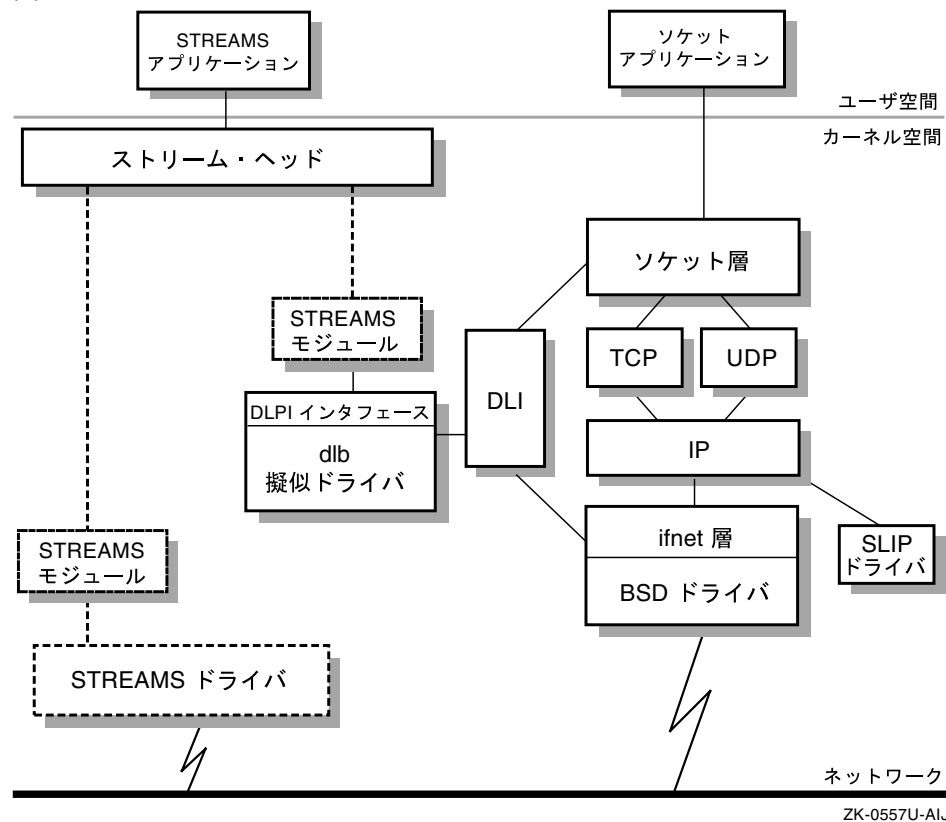
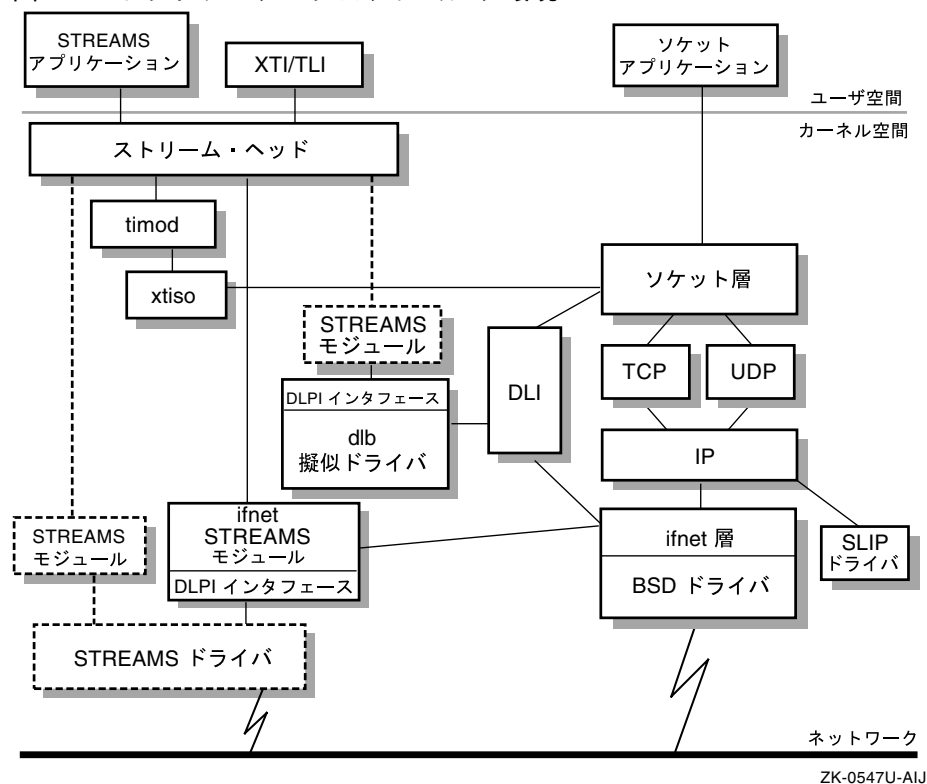


図 1-4 では、データは、STREAMS ベースのアプリケーションからストリーム・ヘッドを通じて流れ、スタックにプッシュされている STREAMS モジュールによって処理されます。最後に、データは STREAMS ドライバに引き渡されるのではなく、dlb STREAMS 擬似ドライバに引き渡されて、ソケット・フレームワークの ifnet 層に転送されます。そこから、さらに BSD ドライバによって処理され、ネットワークとやりとりされます。

1.7 統合

図 1-5 は、ネットワーク・プログラミング環境全体を示しています。この図のバリエーションを各章で使用して、その章で提示する情報の全体における位置付けを示します。

図 1-5: ネットワーク・プログラミング環境





2

データ・リンク・プロバイダ・インタフェース

Tru64 UNIX では、データ・リンク・プロバイダ・インタフェース (DLPI) の部分的なインプリメンテーションである d1b STREAMS 擬似ドライバを提供します。

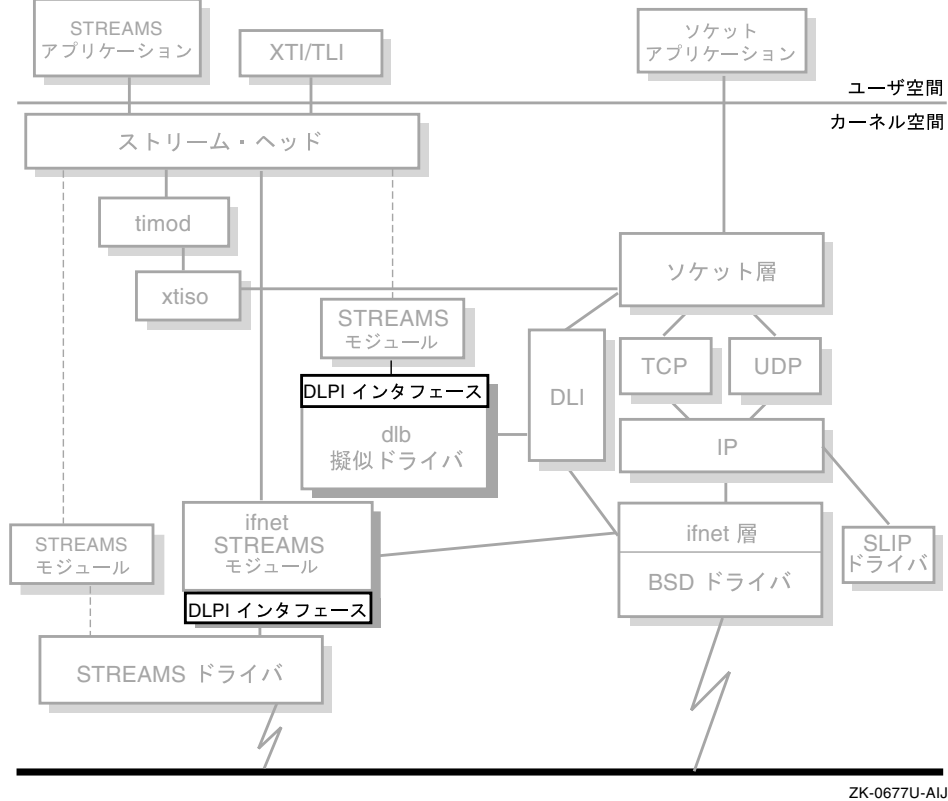
この章では、d1b STREAMS 擬似ドライバおよび DLPI の基礎について説明します。DLPI 仕様の PostScript ファイル (dlpi.ps) が、`/usr/share/doclib/dlpi` ディレクトリにあります。

注意

オンラインの DLPI 仕様にアクセスするには、OSFPGMR_{nnn} サブセットがインストールされていなければなりません。

図 2-1 は、データ・リンク・インタフェースを強調表示して、ネットワーク・プログラミング環境の他の部分との関係を示しています。

図 2-1: DLPI インタフェース



注意

dlb STREAMS 擬似ドライバは DLPI プリミティブのサブセットをサポートしています。サポートしているプリミティブの一覧については、2.4 節を参照してください。

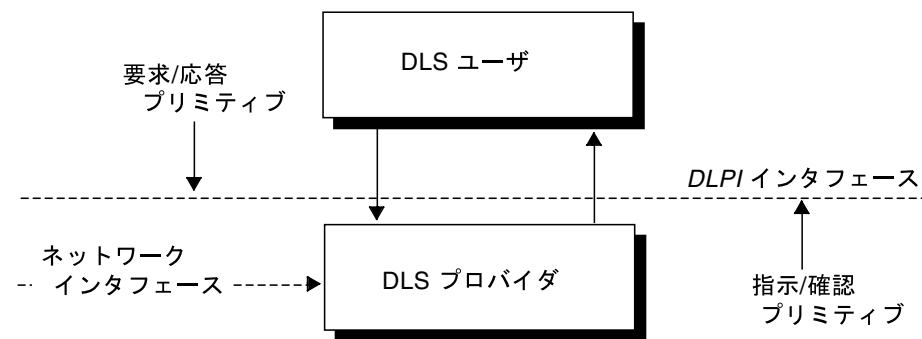
データ・リンク・インタフェースは、OSI 参照モデルのネットワーク層とデータ・リンク層の境界です。ネットワーク・アプリケーション、つまりデータ・リンク・サービス・ユーザ (DLSユーザ) は、データ・リンク・インタフェースのサービスを使用します。ドライバ、擬似ドライバ、つまりデータ・リンク・サービス・プロバイダ (DLSプロバイダ) は、データ・リンク層に対するサービスを提供します。

2-2 データ・リンク・プロバイダ・インタフェース

DLPI は、OSI 参照モデルへマップする、STREAMS カーネル・レベルのサービス・インタフェースを指定します。これは、データ・リンク層のサービスに対するインタフェースを定義し、そのインタフェースを構成するサービス・プリミティブを定義します。

図 2-2 は、DLPI の構成要素を示しています。DLS ユーザは、要求/応答プリミティブを使用して、DLS プロバイダと通信します。一方、DLS プロバイダは、指示/確認プリミティブを使用して、DLS ユーザと通信します。

図 2-2: DLPI サービス・インタフェース



ZK0731UJR

2.4 節にサポートされているプリミティブの一覧を記載しています。

2.1 通信モード

DLPI は、次の 3 つの通信モードをサポートします。

- コネクション・モード

DLS ユーザがデータ・リンク接続を確立し、その接続を通してデータを転送したり、リンクをリセットしたり、会話の終了時に接続を解放できるようにします。

接続確立サービスは、ローカル DLS ユーザとリモート DLS ユーザの間に、データの送信を目的としてデータ・リンク接続を確立します。1 つのストリームには、1 つのデータ・リンク接続しか許可されません。

- コネクションレス・モード

DLS ユーザが、接続の確立および解放のオーバーヘッドを発生することなく、データ・ユニットを対等 DLS ユーザに転送できるようにします。ただし、コネクションレス型サービスは、対等 DLS ユーザ間の

データ・ユニットの引き渡しにおける信頼性が高くありません。たとえば、フロー制御がないため、バッファ・リソースが不足し、データが破棄されることがあります。

ストリームは、ローカル管理サービスを使用して一度初期化されると、コネクションレス型データ・ユニットの送受信に使用できます。

注意

Tru64 UNIX は、コネクションレス型の通信モードだけをサポートします。

- 肯定応答コネクションレス・モード

一般に、対等 DLS ユーザ間で、情報を高い信頼性で転送する場合に使用します。このサービスは、LAN 間のデータ・ユニット転送に肯定応答が必要であるが、コネクション・モード・サービスに関連する煩雑さを避けたいアプリケーション用に設計されています。交換サービスはコネクションレス型ですが、開始ステーションによって送信されたデータは、確実に正しい順序で引き渡されます。

2.2 サービスの種類

この節では、サービスの種類、つまり DLPI がサポートする通信フェーズについて説明します。利用できるサービスの種類は、DLS プロバイダと DLS ユーザ間の通信モード (コネクション、コネクションレス、肯定応答コネクションレス) によって異なることに注意してください。

DLPI がサポートするサービスの種類を次に示します。

- ローカル管理サービス
 - 情報報告サービス
 - アタッチ・サービス
 - バインド・サービス
- コネクション・モード・サービス
 - 接続確立
 - データ転送
 - 接続解放

- リセット・サービス
- コネクションレス・モード・サービス
 - コネクションレス型データ転送
 - サービス品質 (QOS) 管理
 - エラー報告
- 肯定応答コネクションレス・モード・サービス
 - 肯定応答コネクションレス・モード・データ転送
 - サービス品質 (QOS) 管理
 - エラー報告

2.2.1 ローカル管理サービス

ローカル管理サービスは、DLPI がサポートする 3 つの通信モードすべてに適用されます。このサービスを使用すると、DLS ユーザは、DLS プロバイダに接続されているストリームを初期化し、そのプロバイダに対する ID を設定できます。ローカル管理サービスは、次のサービスをサポートします。

- 情報報告サービス

DLPI ストリームに関する情報を DLS ユーザに提供します。
- アタッチ・サービス

アタッチメントの物理ポイント (PPA : physical point of attachment) をストリームに割り当てます。詳細は、2.3 節を参照してください。
- バインド・サービス

データ・リンク・サービス・アクセス・ポイント (DLSAP) をストリームに関連付けます。

2.2.2 コネクション・モード・サービス

コネクション・モード・サービスを使用すると、2 人の DLS ユーザが、データ・リンク接続を確立してデータを交換できます。また、会話の終了時に、リンクをリセットして、接続を解放することも可能です。コネクション・モード・サービスでは、次のサービスをサポートします。

- 接続確立サービス

ローカル DLS ユーザとリモート DLS ユーザの間に、データの送信を目的として、データ・リンク接続を確立します。

- データ転送サービス

ユーザ・データを片方向または双方向で同時に交換します。データは、データ・リンク・サービス・データ・ユニット (DLSDU) という論理グループで送信され、送信された順序で確実に引き渡されます。

- 接続解放サービス

DLS ユーザまたは DLS プロバイダのどちらかが、確立された接続を切断できるようにします。

- リセット・サービス

DLS ユーザが、データ・リンク接続の使用を再同期できるか、または DLS プロバイダが、データ・リンク・サービス内で回復不可能なデータ損失の検出を報告できるようにします。

2.2.3 コネクションレス・モード・サービス

コネクションレス・モード・サービスを使用すると、DLS ユーザは、接続の確立および解放のオーバーヘッドを発生することなく、データを交換できます。コネクションレス・モード・サービスでは、次のサービスをサポートします。

- コネクションレス型データ転送サービス

ユーザ・データ (DLSDU) を、片方向または双方向で同時に交換します。

- サービス品質 (QOS) 管理サービス

DLS ユーザが、コネクションレス型データ転送サービスの各呼び出しに対して、予測可能なサービス品質を指定できるようにします。

- エラー報告サービス

以前に送信されたデータ・ユニットでエラーが発生したこと、またはそのデータ・ユニットが引き渡しできなかったことを、DLS ユーザに通知する機能を提供します。ただし、エラー報告サービスは必ずしも、データ・ユニットの引き渡しができないたびに、エラーを報告するわけではありません。

2.2.4 肯定応答コネクションレス・モード・データ転送

肯定応答コネクションレス・モード・データ転送サービスは、対等 DLS ユーザ間で信頼性の高いデータ転送を行うために、一般に使用されるものです。これらのサービスは、ローカル・エリア・ネットワーク間のデータ転送に肯定応答が必要であるが、コネクション・モード・サービスの使用を避けたいアプリケーションに適しています。開始ステーションによって送信されたデータは、確実に正しい順序で引き渡されます。次のサービスがサポートされます。

- 肯定応答コネクションレス・モード・データ転送サービス
LLC 下位層で肯定応答された DLSDU を交換できるようにします。
- サービス品質 (QOS) 管理サービス
DLS ユーザが、コネクションレス型データ転送サービスの各呼び出しに対して、予測可能なサービス品質を指定できるようにします。
- エラー報告サービス
以前に送信されたデータ・ユニットでエラーが発生したこと、またはそのデータ・ユニットが引き渡しできなかったことを、DLS ユーザに通知する機能を提供します。ただし、エラー報告サービスは必ずしも、データ・ユニットの引き渡しができないたびに、エラーを報告するわけではありません。

2.3 DLPI アドレッシング

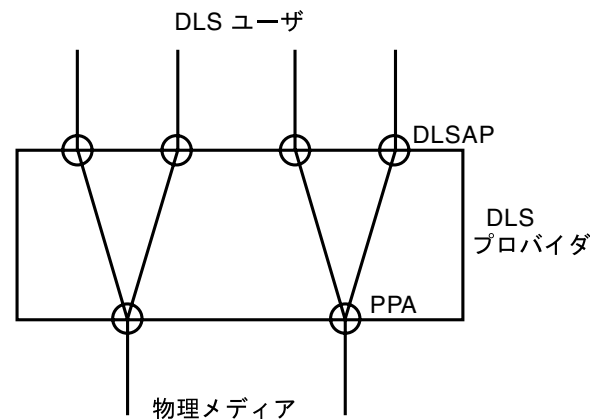
各 DLPI ユーザは、他のデータ・リンク・ユーザと通信するために、ID を設定しなければなりません。この ID は、次の情報から構成されます。

- 物理アタッチメント識別
これは、DLS ユーザが通信に使用する物理メディアを識別します。物理メディアの識別は、複数の物理メディアにアタッチされたシステム上で特に重要です。システムで利用できるアタッチメントの物理ポイント (PPA) の識別については、2.5 節を参照してください。
- データ・リンク・ユーザ識別
DLS ユーザは DLS プロバイダに登録して、プロバイダがそのユーザ宛のプロトコル・データ・ユニットを引き渡せるようにしなければなりません。

DLSAP アドレスのフォーマットには、MAC アドレスの後にバインドされるサービス・アクセス・ポイント (SAP) が続く、符号なしの文字配列を使用します。通常 SAP はイーサネットの場合 2 バイト、ISO 8802-2 (IEEE 802.2) の場合 1 バイトです。ただし、例外として HIERACHICAL DL_SUBS_BIND_REQ が処理される場合、DLSAP アドレスは、MAC アドレス、SNAP SAP (0xAA) および 5 バイトの SNAP で構成されます。

図 2-3は、この識別方式の構成要素を示しています。

図 2-3: DLPI アドレスの識別構成要素



PPA は、システムが自分自身を物理通信メディアにアタッチするポイントです。この物理メディアを通るすべての通信は、PPA をファネルします。DLS プロバイダが複数の物理メディアをサポートするシステムでは、DLS ユーザは、通信に使用するメディアを識別しなければなりません。PPA は、一意の PPA 識別子によって識別されます。

DLPI は、次の 2 つのスタイルの DLS プロバイダを定義します。この 2 つのスタイルは、DLS ユーザが特定の PPA を選択できるようにする方法が異なります。

- スタイル 1

スタイル 1 のプロバイダは、DLS ユーザがオープンした主デバイスまたは副デバイスに基づいて、PPA を割り当てます。スタイル 1 のドライバをインプリメントして、データ・リンク・ドライバがサポートする各 PPA 用に主デバイスを予約できるようにすることができます。

スタイル 1 のドライバをインプリメントすると、構成された各 PPA に対して、STREAMS の clone open 機能を使用できるようになります。スタイル 1 のプロバイダは、サポートする PPA の数が少ない場合に適しています。

- スタイル 2
スタイル 2 のプロバイダは、DLS ユーザが特殊な attach サービス・プリミティブを使用して、PPA を明示的に識別することを要求します。スタイル 2 のドライバでは、open システム・コールが、DLS ユーザと DLS プロバイダ間にストリームを作成します。次に、attach プリミティブが、特定の PPA をそのストリームに関連付けます。PPA 識別子のフォーマットは、DLS プロバイダによって異なります。

Tru64 UNIX は、スタイル 2 のプロバイダだけをサポートします。これは、スタイル 2 のプロバイダの方が多数の PPA をサポートするのに適しているからです。

2.4 DLPI プリミティブ

表 2-1 は、dlb STREAMS 擬似ドライバでサポートされている DLPI プリミティブの一覧とその説明です。DLPI プリミティブの完全な一覧については、/usr/share/doc/lib/dlpi/dlpi.ps ファイルの DLPI 仕様を参照してください。

表 2-1: サポートされている DLPI プリミティブ

プリミティブ	説明
DL_ATTACH_REQ	DLS プロバイダに、PPA をストリームに関連付けるように要求する。スタイル 2 のプロバイダに対してのみ使用する。
DL_BIND_REQ	DLS プロバイダに、DLSAP をストリームにバインドするように要求する。DLS ユーザは、ストリームにバインドされる DLSAP のアドレスを識別しなければならない。
DL_BIND_ACK	DLSAP のストリームへのバインドが正常に行われたことを報告し、バインドされた DLSAP アドレスを DLS ユーザに返す。DL_BIND_REQ への応答として生成される。
DL_DETACH_REQ	DLS プロバイダに、PPA のストリームへの関連付けを解除するように要求する。

表 2-1: サポートされている DLPI プリミティブ (続き)

プリミティブ	説明
DL_DISABMULTI_REQ	DLS プロバイダにマルチキャスト・アドレスを使用不可にするように要求する。
DL_ENABMULTI_REQ	DLS プロバイダに特定のマルチキャスト・アドレスを使用可能にするように要求する (DLB ドライバの現在のインプリメンテーションでは、DL_IDLE の状態であることが要求される)。
DL_ERROR_ACK	直前に発行された誤った要求を DLS ユーザに報告する。
DL_INFO_ACK	DL_INFO_REQ プリミティブに応答し、DLPI ストリームについての情報を伝送する。
DL_INFO_REQ	DLPI ストリームについての情報を返すように DLS プロバイダに要求する。
DL_OK_ACK	DLS ユーザに、以前に発行された要求プリミティブが正常に受信されたことを肯定応答する。
DL_PHYS_ADDR_REQ	要求で選択されたアドレス型の値に応じて、ストリームに関連付けられた物理アドレスの省略時の値 (工場出荷時の値)、または現在の値のいずれかを返すように、DLS プロバイダに要求する。
DL_PHYS_ADDR_ACK	DL_PHYS_ADDR_REQ への応答として、DLS ユーザに物理アドレスの値を返す。
DL_SUBS_BIND_ACK	DLS プロバイダからの DL_SUBS_BIND_REQ への肯定応答する。
DL_SUBS_BIND_REQ	DLS プロバイダに、順次 DLSAP をストリームにバインドするように要求する。順次バインド要求には HIERACHICAL と PEER の 2 つのクラスがある。HIERACHICAL 要求は SNAP (詳細は IEEE 802.1 仕様を参照) についてのみに有効で、SNAP に DL_SUBS_BIND_REQ を発行する前に、SNAP SAP (0xAA) が DL_BINDS_REQ とバインドされている必要がある。PEER 要求は追加 SAP とバインドするが、ストリームの DLSAP アドレスは変更しない。
DL_SUBS_UNBIND_REQ	DLS プロバイダに、既に DL_SUBS_BIND_REQ によってバインドされた SAP のバインドを解除するように要求する。
DL_TEST_CON	DL_TEST_REQ への応答として、DLSDU TEST 応答が受信されたことを伝達する。

表 2-1: サポートされている DLPI プリミティブ (続き)

プリミティブ	説明
DL_TEST_IND	DLS ユーザに TEST コマンド DLSDU が受信されたことを伝達する。
DL_TEST_REQ	DLS プロバイダに、TEST コマンド DLSDU を DLS ユーザに代わって伝送することを要求する。
DL_TEST_RES	DLS プロバイダに TEST 応答コマンドを DLS ユーザに代わって送信するように要求する。
DL_UDERROR_IND	DLS ユーザに直前に送信した DL_UNITDATA_REQ が失敗したことを報告する。
DL_UNBIND_REQ	DLS プロバイダに、直前の DL_BIND_REQ によってバインドされた DLSAP を、このストリームからバインド解除するように要求する。
DL_UNITDATA_REQ	対等 DLS ユーザへ伝送するために、1 つの DLSDU を DLS ユーザから DLS プロバイダへ引き渡す。
DL_UNITDATA_IND	DLS プロバイダから DLS ユーザに、1 つの DLSDU を引き渡す。
DL_XID_CON	DL_XID_REQ への応答として、XID DLSDU が受信されたことを伝達する。
DL_XID_IND	DLS ユーザに XID DLSDU が受信されたことを伝達する。
DL_XID_REQ	DLS プロバイダに、XID DLSDU をユーザの代わりに伝送するように要求する。
DL_XID_RES	DL_XID_REQ への応答として、XID DLSDU を DLS ユーザの代わりに送信するように要求する。

2.5 利用できる PPA の識別

次のプログラムは、ルートとしてコンパイルおよび実行すると、STREAMS デバイス /dev/streams/dlb をオープンし、システムで利用できる PPA をスクリーンに表示します。PPA 番号は、DL_ATTACH_REQ DLPI プリミティブの dl_ppa フィールドを使用して渡します。

```
#include <sys/ioctl.h>
#include <stropts.h>
#include <errno.h>
#include <fcntl.h>

#define ND_GET ('N' << 8 + 0)
#define BUFSIZE 256
```

```

main()
{
    int i;
    int fd;
    char buf [BUFSIZE];
    struct strioctl stri;

    fd = open("/dev/streams/dlb", O_RDWR, 0);
    if (fd < 0) {
        perror("open");
        exit(1);
    }

    sprintf(buf, "dl_ifnames");
    stri.ic_cmd = ND_GET;
    stri.ic_timeout = -1;
    stri.ic_len = BUFSIZE;
    stri.ic_dp = buf;

    if (ioctl(fd, I_STR, &stri) < 0) {
        perror("ioctl");
        exit(1);
    }

    printf("Valid PPA names on this system are:\n");
    for (i=0; i<stri.ic_len; i++) {
        if (buf[i] == 0)
            printf(" ");
        else
            printf("%c",buf[i]);
    }
    printf("\n");
}

# a.out
Valid PPA names on this system are:
sscc0 (PPA 1) ln0 (PPA 2) dsy0 (PPA 3) dsy1 (PPA 4) \
sl0 (PPA 5) sl1 (PPA 6) lo0
#

```

X/Open トランスポート・インタフェース

X/Open トランスポート・インタフェース (XTI) は、一連の関数からなるトランスポート層アプリケーション・インタフェースであり、使用する特定のトランスポート・プロバイダから独立するために開発されました。このオペレーティング・システムでは、XPG3、XNS4.0 および XNS5.0 仕様に従って XTI をインプリメントしています。XNS4.0 が省略時の設定です。XPG3 は下位互換性のために提供されており、コンパイラ・スイッチを使用して有効にできますが、将来的にはサポートされなくなります。XNS5.0 も、コンパイラ・スイッチを使用して有効にできます。XPG3、XNS4.0 および XNS5.0 についての詳細は、それぞれ『*X/Open Portability Guide Volume 7: Networking Services*』、『*X/Open CAE Specification: Networking Services, Issue 4*』および『*X/Open CAE Specification: Networking Services (XNS), Issue 5*』を参照してください。このオペレーティング・システムでの XTI のインプリメンテーションもスレッド・セーフです。

概念はバークレイ・ソケット・インタフェースと似ていますが、XTI は AT&T トランスポート層インタフェース (TLI) に基づいています。一方、TLI は、開放型システム間相互接続 (OSI) モデルのトランスポート・サービス定義に基づいています。

注意

このオペレーティング・システムには、伝送制御プロトコル (TCP) およびユーザ・データグラム・プロトコル (UDP) のトランスポート・プロバイダが含まれています。この章で説明する情報は、DECnet/OSI などこのオペレーティング・システムの XTI がサポートするすべてトランスポート・プロバイダに当てはまりませんが、例は TCP および UDP に固有のものです。XTI を TCP または UDP で使用する場合についての詳細は、`xti_internet(7)` を参照してください。他のトランスポート・プロバイダ固有の

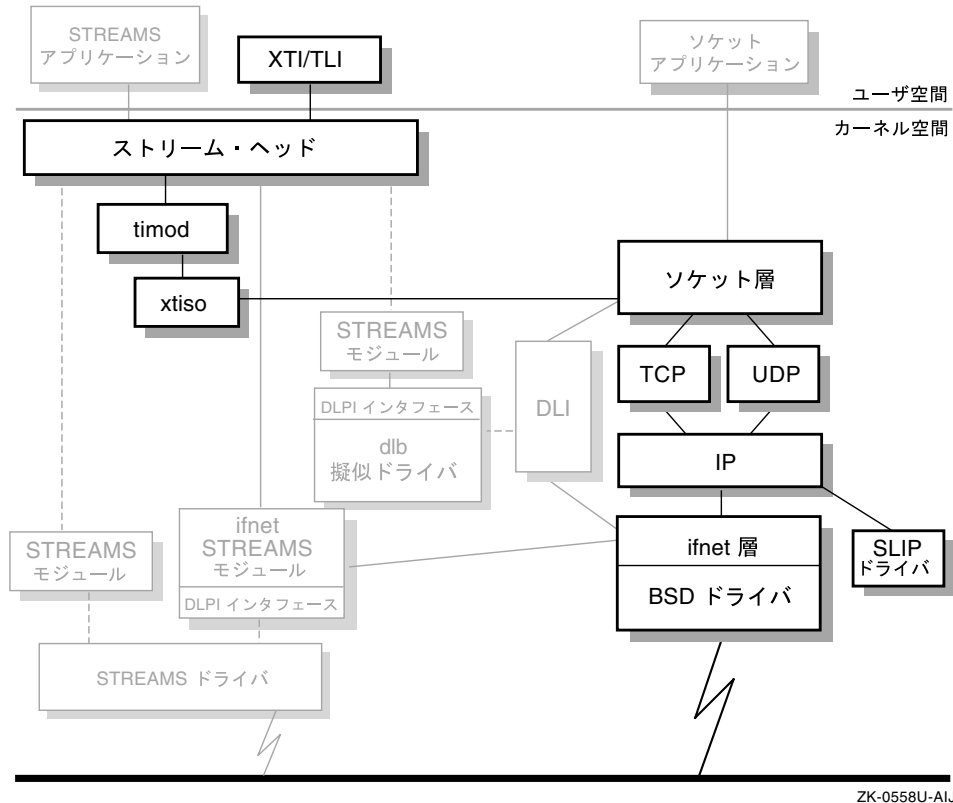
例および情報については、それらのソフトウェアに添付されているドキュメントを参照してください。

この章では、次の事項について説明します。

- XTI の概要
- XTI の機能
- XTI の使用方法
- アプリケーションの XTI へのポート方法
- XPG3 , XNS4.0 , XNS5.0 の相違点
- XTI エラーおよびエラー・メッセージ
- トランポート・プロバイダを組み込んだ構成の方法

図 3-1 は、XTI , および XTI とオペレーティング・システムのインターネット・プロトコル群のインプリメンテーションの関係を強調表示しています。また、XTI とインターネット・プロトコル群が、ネットワーク・プログラミング環境の他の部分に対し、どのように位置付けられているかも示しています。

図 3-1: X/Open トランスポート・インタフェース



ZK-0558U-AIJ

3.1 XTI の概要

XTI には、次の各エンティティ間のやりとりが含まれています。

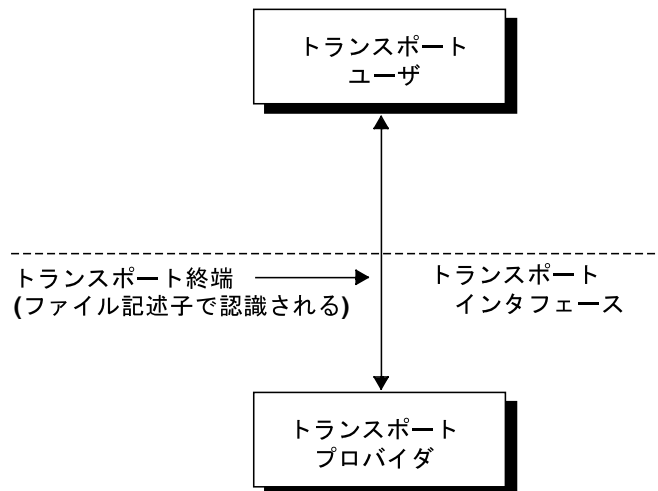
- **トランスポート・プロバイダ**
トランスポート・プロバイダは、トランスポート層サービスを提供する、TCP や UDP などのトランスポート・プロトコルです。
- **トランスポート・ユーザ**
トランスポート・ユーザは、別のプログラムとの間でデータを送受信するときに、トランスポート・プロバイダのサービスを必要とするアプリケーション・プログラムです。トランスポート・ユーザは、トランスポート終端によって識別される通信パスを通して、トランスポート・プロバイダと通信します。
- **トランスポート終端**

アプリケーションが `t_open` ライブラリ・コールを発行すると、トランスポート・ユーザは作成されます。トランスポート・ユーザからトランスポート・プロバイダへの要求はすべて、そのプロバイダに関連付けられた終端を経由します。

トランスポート・ユーザは、トランスポート・ユーザをトランスポート・アドレスにバインドすることによって、トランスポート・ユーザをアクティブにします。一度、トランスポート・ユーザがアクティブになると、トランスポート・ユーザは、この終端を通してデータを送信できます。トランスポート・プロバイダは、該当する対等ユーザ、または他のデスティネーションにデータの経路を指定します。

TCP などのコネクション指向型トランスポート・サービスを使用する場合には、トランスポート・ユーザはデータを送信する前に、アクティブな終端を指定して、`t_connect` 関数を呼び出すことにより、自分自身と対等トランスポート・ユーザの間に接続を確立しなければなりません。トランスポート・接続において、接続を開始するトランスポート・ユーザは、アクティブ・ユーザ、つまりクライアントです。一方、接続要求に応答する対等トランスポート・ユーザは、パッシブ・ユーザ、つまりサーバです。図 3-2 は、トランスポート・プロバイダ、トランスポート・ユーザ、およびトランスポート・終端の関係を示しています。

図 3-2: トランスポート・終端



ZK0522UJR

3.2 XTI の機能

XTI には、ライブラリ・コール、ヘッダ・ファイル、および、XTI プロセスの動作と対話処理の方法を記述する規則と制限があります。この節では、ライブラリ・コールおよびヘッダ・ファイルについて説明するとともに、通信プロセス間のやりとりを制御する規則についても記述します。

3.2.1 サービス・モードと実行モード

トランスポート・ユーザは、さまざまなサービス・モードと実行モードを使用して、トランスポート・プロバイダとの間でデータを交換する方法を決定します。次の 2 つの項で、XTI で利用できるサービス・モードと実行モードについて説明します。

3.2.1.1 コネクション指向型サービスおよびコネクションレス型サービス

XTI において、終端は、次のサービス・モードのうちの 1 つをサポートできます。

- コネクション指向型トランスポート・サービス

回線指向型サービス。確立された接続を通して、信頼性が高い方法でデータの順次転送を行います。

コネクション指向型トランスポートは、時間が長く、順序に依存し、信頼性が高いストリーム指向型のやりとりを必要とするアプリケーションに適しています。コネクション指向型トランスポートを使用すると、トランスポート・ユーザおよびプロバイダは、データ転送を制御するパラメータおよびオプションを折衝できます。さらに、接続により両者は識別可能になるので、トランスポート・ユーザは、データ転送中のアドレスの伝送と解析によるオーバーヘッドを回避できます。また、連続するデータ・ユニットを論理的に関連付けるコンテキストも提供されます。

- コネクションレス型トランスポート・サービス

メッセージ指向型サービス。データを自己完結型ユニットまたはデータグラムとして転送します。データ間の論理的連続はありません。

コネクションレス型トランスポートは、次のような特徴を持つアプリケーションに最適です。

- 短時間の要求と応答のやりとり
- 複数の終端に対する接続の動的な再構成

- データの順次引き渡しの保証が不必要

各データ・ユニットは自己完結型であり、直前または後続のデータ・ユニットとの関係がないので、トランスポート・プロバイダは独立して経路を指定できます。

3.2.1.2 非同期実行および同期実行

実行モードにより、トランスポート・ユーザは、関数の完了およびイベントの受信を処理できます。イベントとは、トランスポート・ユーザにとって意味のある出現事象または偶発事象です。XTI は、次の 2 つの実行モードをサポートします。

- 同期モード

トランスポート・プリミティブが完了してから、制御がトランスポート・ユーザに戻ります。ブロッキング・モードともいいます。

同期モードは、関数の完了までの待機、または単一のトランスポート接続だけの維持を必要とするアプリケーションに適しています。同期モードでは、関数が完了するのを待つ間、トランスポート・ユーザは他のタスクを実行することはできません。たとえば、トランスポート・ユーザが同期モードで `t_rcv` 関数を発行した場合、`t_rcv` はデータを受信してから、制御をトランスポート・ユーザに戻します。

同期モードの使用中でも、イベント通知を受け取ることができます。これは、通常ではトランスポート・ユーザは受け取らないものです。このような非同期イベントは、特殊エラーの TLOOK を通してユーザに返されます。

関数の実行中に非同期イベントが発生した場合、その関数は TLOOK エラーを返します。この場合、トランスポート・ユーザは、`t_look` 関数を発行してイベントを検出できます。

- 非同期モード

トランスポート・プリミティブが完了する前に、制御をトランスポート・ユーザに戻します。非ブロッキング・モードともいいます。

非同期モードは、関数が完了してから他のタスクを実行するまでの間に、長い遅延があるアプリケーションに適しています。また、このモードは、複数の接続を同時に処理するアプリケーションにも適しています。非同期モードでは、特定のネットワーキング関数が完了するのを待っている間に有用な作業を実行できるので、多くのアプリケーション

ンは、非同期モードでネットワーキング関数进行处理します。たとえば、トランスポート・ユーザが、非同期モードで `t_rcv` 関数呼び出しを発行すると、この関数は、データを手に入でなければ、制御をただちにユーザに戻します。ユーザは、データが到着するまで、定期的にデータをポーリングします。

省略時の値では、着信イベント进行处理するすべての関数は同期モードで動作し、そのタスクが完了するまでブロックします。非同期モードを選択する場合には、トランスポート・ユーザは、終端の作成時、または `fcntl` オペレーティング・システム・コールを使用して関数または関数のグループを実行する前に、`t_open` 関数で `O_NONBLOCK` フラグを指定します。

XTI がサポートする特定のイベントについての詳細は、3.2.3 項を参照してください。

3.2.2 XTI ライブラリ、TLI ライブラリ、およびヘッダ・ファイル

XTI 関数は、XTI ライブラリ `libxti.a` の一部としてインプリメントされています。TLI 関数は、別の TLI ライブラリ `libtli.a` にインプリメントされています。これらのライブラリのシェアード・バージョン `libxti.so` および `libtli.so` も提供されます。

XTI または TLI アプリケーションを XTI または TLI ライブラリにリンクするとき、省略時の設定によりシェアード・ライブラリ・サポートが提供されます。

次に 2 つの例を示します。最初の例は、XTI アプリケーションのオブジェクト・ファイルを XTI シェアード・ライブラリに再リンクする方法を示しています。2 番目の例は、TLI アプリケーションのオブジェクト・ファイルを TLI シェアード・ライブラリに再リンクする方法を示しています。

```
% cc -o XTIapp XTIappmain.o XTIapputil.o -lxti
```

```
% cc -o TLIapp TLIappmain.o TLIapputil.o -ltli
```

プログラムを XTI または TLI ライブラリに静的にリンクするには、`cc` コマンドで `non_shared` オプションを使用します。次の例は、XTI アプリケーションのオブジェクト・ファイルを XTI ライブラリに静的にリンクする方法を示しています。

```
% cc -non_shared -o XTIapp XTIappmain.o XTIapputil.o -lxti
```

詳細は `cc(1)` を参照してください。

プログラムをスレッド・セーフにするためには、POSIX スレッド `pthread` ルーチンでプログラムを構築します。詳細については、『*Guide to the POSIX Threads Library*』を参照してください。

XTI と TLI にはほとんど違いがありません。両者の相違点については 3.5.2 項で説明します。また 3.5.2 項では、コンパイル時にプログラムを正しいライブラリとリンクする方法についても説明しています。

3.2.2.1 XTI および TLI ヘッダ・ファイル

XTI および TLI ヘッダ・ファイルには、XTI および TLI ライブラリ・コールが使用する、データ定義、構造体、定数、マクロ、およびオプションが入っています。アプリケーション・プログラムでは、特定の XTI または TLI ライブラリ・コールが必要とする構造体や他の情報を使用するために、適切なヘッダ・ファイルをインクルードしなければなりません。表 3-1 は、XTI および TLI ヘッダ・ファイルの一覧です。

表 3-1: XTI および TLI のヘッダ・ファイル

ファイル名	説明
<tiuser.h>	TLI アプリケーション用のデータ定義および構造体が入っている。TLI アプリケーションにはすべて、このファイルをインクルードしなければならない。
<xti.h>	XTI アプリケーション用のデータ定義および構造体が入っている。XTI アプリケーションにはすべて、このファイルをインクルードしなければならない。
<xti_inet.h>	XTI アプリケーション用のインターネット・データ定義および構造体が入っている。XTI アプリケーションにはすべて、このファイルをインクルードしなければならない。
<xti_osi.h>	XTI アプリケーション用の ISO OSI データ定義および構造体が入っている。XTI アプリケーションにはすべて、このファイルをインクルードしなければならない。
<fcntl.h>	<code>t_open</code> 関数の実行モード用フラグを定義する。すべての XTI アプリケーションおよび TLI アプリケーションには、このファイルをインクルードしなければならない。

注意

一般的に、ヘッダ・ファイル名は山カッコ (< >) で囲まれます。ヘッダ・ファイルの絶対パスは、山カッコ内の情報の前に

/usr/include/ を付けたものです。たとえば、tiuser.h ヘッダ・ファイルの絶対パスは /usr/include/tiuser.h です。

3.2.2.2 XTI ライブラリ・コール

XTI ライブラリ・コールの中には、コネクション指向型トランスポート (COTS) に適用されるもの、コネクションレス型トランスポート (CLTS) に適用されるもの、正常解放機能とともに使用される場合にコネクション指向型トランスポートに適用されるもの (COTS_ORD)、およびすべてのサービス・モードに適用されるものがあります。一部の XTI ライブラリ・コールはユーティリティ関数であり、特定のサービス・モードには適用されません。表 3-2 に、各 XTI ライブラリ・コールの名前、説明、およびサービス・モードの一覧を示します。各ライブラリ・コールについて、同じ名前のリファレンス・ページがあります。

オペレーティング・システムでは、XTI のリファレンス・ページだけを提供し、TLI のリファレンス・ページは提供していません。TLI および TLI のリファレンス・ページについての詳細は、『*UNIX System V Programmer's Guide: Networking Interfaces*』を参照してください。これは、UNIX システムラボラトリーズ社から発行されています。オペレーティング・システムでは、それぞれの関数のリファレンス・ページを提供してします。詳細は、『*X/Open CAE Specification: Networking Services*』を参照してください。

表 3-2: XTI ライブラリ・コール

呼び出し名	説明	サービス・モード
t_accept	接続要求を受け入れる。	COTS, COTS_ORD
t_alloc	ライブラリ構造体のメモリを割り当てる。	すべて
t_bind	アドレスをトランスポート終端にバインドする。	すべて
t_close	トランスポート終端をクローズする。	すべて
t_connect	別のトランスポート・ユーザとの接続を確立する。	COTS, COTS_ORD
t_error	エラー・メッセージを出す。	すべて

表 3-2: XTI ライブラリ・コール (続き)

呼び出し名	説明	サービス・モード
t_free	以前にライブラリ構造体に割り当てられたメモリを解放する。	すべて
t_getinfo	プロトコル固有の情報を返す。	すべて
t_getprotaddr ^a	プロトコル・アドレスを返す。	すべて
t_getstate	トランスポート終端に現在の状態を返す。	すべて
t_listen	接続要求をリッスンする。	COTS, COTS_ORD
t_look	トランスポート終端上の現在のイベントを返す。	すべて
t_open	トランスポート終端を設定する。	すべて
t_optmgmt	プロトコル・オプションの検出, 確認, 折衝のいずれかを行う。	すべて
t_rcv	接続を通して, データまたは優先データを受信する。	COTS, COTS_ORD
t_rcvconnect	接続要求から確認を受信する。	COTS, COTS_ORD
t_rcvdis	切断の原因を識別し, 切断にともなって送信された情報を検出する。	COTS, COTS_ORD
t_rcvrel ^b	正常解放指示の受信に肯定応答する。	COTS_ORD
t_rcvreldata ^c	ユーザ・データがある正常解放指示または正常解放確認を受信する。	COTS_ORD
t_rcvv ^c	接続を通してデータまたは優先データを受信し, バッファに格納する。	COTS, COTS_ORD
t_rcvvudata ^c	データ・ユニットをバッファに受信する。	CLTS
t_rcvudata	データ・ユニットを受信する。	CLTS
t_rcvuderr	データ・ユニットに関連するエラーについての情報を受信する。	CLTS

表 3-2: XTI ライブラリ・コール (続き)

呼び出し名	説明	サービス・モード
<code>t_snd</code>	接続を通して、データまたは優先データを送信する。	COTS, COTS_ORD
<code>t_snddis</code>	確立された接続で解放を開始するか、または接続要求をリジェクトする。	COTS, COTS_ORD
<code>t_sndrel^b</code>	正常解放を開始する。	COTS_ORD
<code>t_sndreldata^c</code>	ユーザ・データがある正常解放の開始または応答を行う。	COTS_ORD
<code>t_sndudata</code>	データ・ユニットを送信する。	CLTS
<code>t_sndv^c</code>	接続を通してデータまたは優先データを送信する。	COTS, COTS_ORD
<code>t_sndvudata^c</code>	データ・ユニットを送信する。	CLTS
<code>t_strerror^a</code>	エラー・メッセージ文字列を作成する。	すべて
<code>t_sync</code>	トランスポート・ライブラリのデータ構造体の同期をとる。	すべて
<code>t_sysconf^c</code>	構成可能な XTI 変数を取り出す。	すべて
<code>t_unbind</code>	トランスポート終端を使用不能にする。	すべて

^aこの関数は XNS4.0 でだけサポートされている。

^bTru64 UNIX では、COTS_ORD の使用をサポートするトランスポート・プロバイダは備えていないため、この関数はエラーを返す。

^cこの関数は XNS5.0 でだけサポートされている。

XTI は、正常解放機能 (`t_sndrel` および `t_rcvrel` 関数) をサポートしています (詳細は表 3-2 を参照)。ただし、アプリケーションが、ISO トランスポート層に対して移植可能にしなければならない場合には、この機能は使用しないようにしてください。

最後に、XTI ヘッダ・ファイルは、サービス・モードを識別する次の定数を定義します。

- `T_COTS` – コネクション指向型トランスポート・サービス (たとえば、OSI トランスポート)

- T_CLTS – コネクションレス型トランスポート・サービス (たとえば, UDP)
- T_COTS_ORD – 正常解放機能がインプリメントされているコネクション指向型トランスポート・サービス (たとえば, TCP)

これらのサービス・モードは, ユーザが `t_open` 関数を使用して終端を作成したときに, トランスポート・プロバイダによって, `info` 構造体の `servtype` フィールドに返されます。

3.2.3 イベントおよび状態

各トランスポート・プロバイダには, トランスポート・ユーザから見た場合, 関連付けられた特定の状態があります。トランスポート・プロバイダの状態, および次に起こり得る状態への遷移は, 発信および着信のイベントによって制御され, 指定されたユーザ・レベルのトランスポート関数の正常復帰に対応します。発信イベントは, トランスポート・プロバイダに要求または応答を送信する関数に対応します。一方, 着信イベントは, トランスポート・プロバイダからデータまたはイベント情報を検出する関数に対応しています。

この項では, トランスポート・プロバイダが取り得る状態, 生じる可能性のある発信および着信のイベント, および関数の呼び出し可能な順序について説明します。

3.2.3.1 XTI イベント

XTI アプリケーションは, 非同期イベントを管理しなければなりません。非同期イベントは, XTI ヘッダ・ファイルに定数として定義されているニーモニックによって識別されます。表 3-3 は, XTI における非同期イベントのイベント名, 説明, およびサービス・モードの一覧です。

表 3-3: 非同期 XTI イベント

イベント名	説明	サービス・モード
T_CONNECT	トランスポート・プロバイダが接続応答を受信した。このイベントは、通常、トランスポート・ユーザが <code>t_connect</code> 関数を発行した後に発生する。	COTS, COTS_ORD
T_DATA	トランスポート・プロバイダが通常データを受信した。通常データとは、トランスポート・サービス・データ・ユニット (TSDU) のすべてまたは一部である。	COTS, CLTS, COTS_ORD
T_DISCONNECT	トランスポート・プロバイダが切断要求を受信した。このイベントは、通常、トランスポート・ユーザが、データ転送関数、 <code>t_accept</code> 関数、 <code>t_snddis</code> 関数のいずれかを発行した後に発生する。	COTS, COTS_ORD
T_EXDATA	トランスポート・プロバイダが優先データを受信した。	COTS, COTS_ORD
T_GODATA	通常データのフローに関するフロー制御制約が解除された。トランスポート・ユーザは、通常データを再度送信できる。	COTS, CLTS, COTS_ORD
T_GOEXDATA	優先データのフローに関するフロー制御制約が解除された。トランスポート・ユーザは、優先データを再度送信できる。	COTS, COTS_ORD
T_LISTEN	トランスポート・プロバイダがリモート・ユーザから接続要求を受信した。このイベントは、ファイル記述子が有効なアドレスにバインドされており、トランスポート接続が確立されていない場合にだけ発生する。	COTS, COTS_ORD
T_ORDREL	トランスポート・プロバイダが正常解放の要求を受信した。	COTS_ORD
T_UDERR	以前に送信されたデータグラムでエラーが検出された。このイベントは、通常、トランスポート・ユーザが <code>t_rcvudata</code> 関数または <code>t_unbind</code> 関数を発行した後に発生する。	CLTS

XTI は、トランスポート終端で発生するすべてのイベントを格納します。

同期実行モードを使用している場合、トランスポート・ユーザは、実行していた関数から `--1` の値を指定して戻ると、`t_errno` の TLOOK 値をチェックし、`t_look` 関数を使用してイベントを検出します。非同期実行モードでは、トランスポート・ユーザは作業を継続し、定期的に新しいイベントをチェックします。

トランスポート終端で発生する各イベントは、特定の XTI 関数によって消費されるか、または未処理のまま残ります。例外は `T_GODATA` イベントおよび `T_GOEXDATA` イベントであり、これは、`t_look` を使用して検出されるとクリアされます。このように、一度トランスポート・ユーザが、関数から TLOOK エラーを受け取ると、そのトランスポート・ユーザがそのイベントを消費するまで、同じ関数または別の関数への後続の呼び出しで TLOOK エラーが返されます。表 3-4 に、各非同期イベントの消費関数の一覧を示します。

表 3-4: 非同期イベントおよび消費関数

イベント	t_look によるクリア	消費関数
T_CONNECT	No	t_connect, t_rcvconnect
T_DATA	No	t_rcv, t_rcvudata
T_DISCONNECT	No	t_rcvdis
T_EXDATA	No	t_rcv
T_GODATA	Yes	t_snd, t_sndudata
T_GOEXDATA	Yes	t_snd
T_LISTEN	No	t_listen
T_ORDREL	No	t_rcvrel
T_ORDRELDATA ^a	No	t_rcvreldata
T_UDERR	No	t_rcvuderr

^aTru64 UNIX の TCP トランスポート・プロバイダは、正常解放のデータをサポートしていません。

表 3-5 は、特定の XTI 関数に TLOOK エラーを返させるイベントの一覧です。これは、XTI アプリケーションにイベント・チェック機能を構築する場合に役立ちます。

表 3-5: TLOOK を返す XTI 関数

関数	TLOOK の原因となるイベント
t_accept	T_DISCONNECT, T_LISTEN
t_connect	T_DISCONNECT, T_LISTEN ^a
t_listen	T_DISCONNECT ^b
t_rcv	T_DISCONNECT, T_ORDREL ^c
t_rcvconnect	T_DISCONNECT
t_rcvrel	T_DISCONNECT
t_rcvreldata	T_DISCONNECT
t_rcvudata	T_UDERR
t_rcvv	T_DISCONNECT, T_ORDREL
t_rcvvudata	T_UDERR
t_snd	T_DISCONNECT, T_ORDREL
t_sndv	T_DISCONNECT, T_ORDREL
t_snddis	T_DISCONNECT
t_sndrel	T_DISCONNECT
t_sndreldata	T_DISCONNECT
t_sndudata	T_UDERR
t_sndvudata	T_UDERR
t_unbind	T_LISTEN, T_DATA ^d

^aこのイベントは、*qlen* > 0 にバインドされ、保留の接続指示を持っている終端に対して *t_connect* が発行された場合だけ発生する。

^bこのイベントは、未処理の接続指示における切断を示す。

^cこのイベントは、未処理のデータがすべて読み取られた場合のみ発生する。

^dT_DATA はコネクションレス・モードでだけ発生する。

各 XTI 関数は、一度に 1 つのトランスポート終端を管理します。一度に複数のソース、特に複数のトランスポート接続から、複数のイベントを待つことはできません。XTI のこのインプリメンテーションでは、トランスポート・ユーザは *poll* 関数を使用して、複数のファイル記述子における着信と発信を監視することができます。詳細は、*poll*(2) を参照してください。

3.2.3.2 XTI の状態

XTI は、任意の時点でプログラムが実行する呼び出しの正当性を制御します。XTI は、8 つの状態を使用して、トランスポート終端を通した通信を管

理します。アクティブ・ユーザおよびパッシブ・ユーザはどちらも、処理中の関数を反映する固有の状態を持っています。

表 3-6 で、各 XTI の状態の意味について説明します。サービス・モードの COTS は、正常サービスがインプリメントされているかどうかに関係なく、その状態が生じることを示します。サービス・モードの COTS_ORD は、正常サービスがインプリメントされている場合にだけ、その状態が生じることを示します。

表 3-6: XTI の状態

状態	説明	サービス・モード
T_UNINIT	初期化されていない。インタフェースの最初と最後の状態。トランスポート終端を設定するには、ユーザは <code>t_open</code> を発行しなければならない。	COTS, CLTS, COTS_ORD
T_UNBIND	バインドされていない。ユーザはアドレスをトランスポート終端にバインドしたり、トランスポート終端をクローズしたりできる。	COTS, CLTS, COTS_ORD
T_IDLE	アイドル。アクティブ・ユーザは、パッシブ・ユーザとの間に接続を確立したり (COTS)、トランスポート終端を使用不能にしたり (COTS, CLTS)、データ・ユニットを送受信したり (CLTS) できる。パッシブ・ユーザは、接続要求をリスンできる (COTS)。	COTS, CLTS, COTS_ORD
T_OUTCON	発信接続の保留。アクティブ・ユーザは、接続要求の確認を受信できる。	COTS, COTS_ORD
T_INCON	着信接続の保留。パッシブ・ユーザは、接続要求を受け入れることができる。	COTS, COTS_ORD
T_DATAXFER	データ転送。アクティブ・ユーザは、パッシブ・ユーザとデータの送受信ができる。パッシブ・ユーザは、アクティブ・ユーザとデータの送受信ができる。	COTS, COTS_ORD
T_OUTREL	正常解放の発信。ユーザは、正常解放指示に応答できる。	COTS_ORD
T_INREL	正常解放の着信。ユーザは、正常解放指示を送信できる。	COTS_ORD

コネクション指向型アプリケーションを作成する場合，接続確立状態またはデータ転送状態の間は，いつでもプログラムで接続を解放できます。

3.2.4 XTI イベントのトラッキング

XTI ライブラリでは，発信イベントと着信イベントを追跡できるので，トランスポート終端の正当な状態を管理できます。これ以降の項では，この発信イベントおよび着信イベントについて説明します。

3.2.4.1 発信イベント

発信イベントは，要求または応答をトランスポート・プロバイダに送信する XTI 関数によって引き起こされます。発信イベントは，その関数が正常に戻ると発生します。このような関数の中には，次の値に応じて，異なるイベントを生じるものがあります。

<i>ocnt</i>	未処理の接続指示の数 (トランスポート・ユーザに引き渡されたが，受け入れもリジェクトもされていないもの)。この数は，現在のトランスポート終端 (<i>fd</i>) のみで意味を持つ。
<i>fd</i>	現在のトランスポート終端のファイル記述子。
<i>resfd</i>	接続が受け入れられる終端のファイル記述子。

表 3-7 は，XTI で使用できる発信イベントの一覧です。サービス・モードの COTS は，正常サービスがインプリメントされているかどうかに関係なく，そのイベントがコネクション指向型サービスに対して発生することを示します。サービス・モードの COTS_ORD は，正常サービスがインプリメントされている場合にだけ，そのイベントが発生することを示します。

表 3-7: 発信 XTI イベント

イベント	説明	サービス・モード
opened	t_open 関数の正常な戻り。	COTS, CLTS, COTS_ORD
bind	t_bind 関数の正常な戻り。	COTS, CLTS, COTS_ORD
optmgmt	t_optmgmt 関数の正常な戻り。	COTS, CLTS, COTS_ORD
unbind	t_unbind 関数の正常な戻り。	COTS, CLTS, COTS_ORD
closed	t_close 関数の正常な戻り。	COTS, CLTS, COTS_ORD

表 3-7: 発信 XTI イベント (続き)

イベント	説明	サービス・モード
connect1	同期実行モードでの <code>t_connect</code> 関数の正常な戻り。	COTS, COTS_ORD
connect2	<code>t_connect</code> 関数が非同期モードにおいて TNODATA エラーを返したか、またはトランスポート終端に切断指示が到着したために TLOOK エラーを返した。	COTS, COTS_ORD
accept1	<code>t_accept</code> 関数の正常な戻り。 (<code>ocnt == 1</code> かつ <code>fd == resfd</code> の場合)	COTS, COTS_ORD
accept2	<code>t_accept</code> 関数の正常な戻り。 (<code>ocnt == 1</code> かつ <code>fd != resfd</code> の場合)	COTS, COTS_ORD
accept3	<code>t_accept</code> 関数の正常な戻り。 (<code>ocnt > 1</code> の場合)	COTS
snd	<code>t_snd</code> 関数の正常な戻り。	COTS
snddis1	<code>t_snddis</code> 関数の正常な戻り。 (<code>ocnt <= 1</code> の場合)	COTS, COTS_ORD
snddis2	<code>t_snddis</code> 関数の正常な戻り。 (<code>ocnt > 1</code> の場合)	COTS, COTS_ORD
sndrel	<code>t_sndrel</code> 関数の正常な戻り。	COTS_ORD
sndudata	<code>t_sndudata</code> 関数の正常な戻り。	CLTS

3.2.4.2 着信イベント

着信イベントは、トランスポート・プロバイダからデータまたはイベントを検出する XTI 関数によって引き起こされます。着信イベントは、その関数が正常に戻ると発生します。このような関数の中には、`ocnt` 変数の値に応じて、異なるイベントを生じるものがあります。この変数は、未処理の接続指示 (トランスポート・ユーザに引き渡されたが、受け入れもリジェクトもされていないもの) の数です。この数は、現在のトランスポート終端 (`fd`) のみで意味を持ちます。

`pass_conn` 着信イベントは、指定した終端で関数が正常に戻ったこととは、直接は関係しません。`pass_conn` イベントは、現在の終端から接続が引き渡されている終端においてのみ発生します。`pass_conn` イベントが発生する終端では関数は発生しません。

表 3-8 は、XTI で使用できる着信イベントの一覧です。サービス・モードの COTS は、正常サービスがインプリメントされているかどうかに関係なく、そのイベントが発生することを示します。サービス・モードの COTS_ORD は、正常サービスがインプリメントされている場合にだけ、そのイベントが発生することを示します。

表 3-8: 着信 XTI イベント

イベント	説明	サービス・モード
listen	t_listen 関数の正常な戻り。	COTS, COTS_ORD
rcvconnect	t_rcvconnect 関数の正常な戻り。	COTS, COTS_ORD
rcv	t_rcv 関数の正常な戻り。	COTS, COTS_ORD
rcvdis1	t_rcvdis 関数の正常な戻り。 (ocnt == 0 の場合)	COTS, COTS_ORD
rcvdis2	t_rcvdis 関数の正常な戻り。 (ocnt == 1 の場合)	COTS, COTS_ORD
rcvdis3	t_rcvdis 関数の正常な戻り。 (ocnt > 1 の場合)	COTS, COTS_ORD
rcvrel	t_rcvrel 関数の正常な戻り。	COTS_ORD
rcvudata	t_rcvudata 関数の正常な戻り。	CLTS
rcvuderr	t_rcvuderr 関数の正常な戻り。	CLTS
pass_conn	別のトランスポート終端から引き渡された接続を正常に受け取った。	COTS, COTS_ORD

3.2.5 XTI 関数，イベント，および状態のマッピング

この項では、XTI 関数、発信イベントと着信イベント、および状態の関係について説明します。XTI では、状態遷移に関する規則が厳密に定義されているので、現在の状態および最新の受信イベントによって次に起こり得る状態を知ることができます。この項では、現在のイベントおよび状態を、次に起こり得る状態にマッピングする詳細な表を示します。

この項の状態遷移の説明では、t_getstate 関数、t_getinfo 関数、t_alloc 関数、t_free 関数、t_look 関数、t_sync 関数、および t_error 関数については触れません。これらのユーティリティ関数は、トランスポート

ト・インタフェースの状態には影響しないため、未初期化 (T_UNINIT) 状態以外のすべての状態から発行できます。

表 3-9、表 3-10、および表 3-11 を使用して、現在の着信イベントまたは発信イベントに一致する行と、現在の状態に一致する列を探してください。その行と列が交わる欄に示されている状態が、次に起こり得る状態です。交わる欄にダッシュ (--) が記述されている場合は、イベントと状態の組み合わせが無効であることを示しています。状態遷移の中には、右肩に英字が付いているものもあります。この英字は、トランスポート・ユーザが行わなければならないアクションを示しています。英字とその意味は、該当する表の脚注に記載されています。

表 3-9 は、初期化関数および初期化解除関数、つまりコネクション指向型とコネクションレス型の両方のサービス・モードに共通した関数の状態遷移を示しています。たとえば、現在のイベントと状態がそれぞれ bind と T_UNBND の場合、次に起こり得る状態は T_IDLE です。また、英字 a が示されているので、トランスポート・ユーザは、未処理の接続指示の数にゼロを設定しなければなりません。

表 3-9: コネクション指向型またはコネクションレス型トランスポート・サービスの初期化における状態遷移

イベント	T_UNINIT 状態	T_UNBND 状態	T_IDLE 状態
opened	T_UNBND	—	—
bind	—	T_IDLE ^a	—
unbind	—	—	T_UNBND
closed	—	T_UNINIT	T_UNINIT

^a未処理の接続指示の数 *ocnt* に 0 を設定する。

表 3-10 は、コネクションレス型トランスポート・サービスにおけるデータ転送関数の状態遷移を示しています。

表 3-10: コネクションレス型トランスポート・サービスにおける状態遷移

イベント	T_IDLE 状態
sndudata	T_IDLE
rcvudata	T_IDLE
rcvuderr	T_IDLE

表 3-11，表 3-12 は，着信イベントおよび発信イベントについて，コネクション指向型トランスポート・サービスにおける接続，解放，およびデータ転送の関数の遷移を示しています。たとえば，現在のイベントと状態がそれぞれ `accept2` と `T_INCON` の場合，次に起こり得る状態は `T_IDLE` です。このとき，トランスポート・ユーザは，未処理の接続指示の数を減少させて，接続を別のトランスポート終端に引き渡します。

表 3-11: コネクション指向型トランスポート・サービスにおける状態遷移:パート 1

イベント	T_IDLE 状態	T_OUTCON 状態	T_INCON 状態	T_DATAXFER 状態
<code>connect1</code>	<code>T_DATAXFER</code>	—	—	—
<code>connect2</code>	<code>T_OUTCON</code>	—	—	—
<code>rcvconnect</code>	—	<code>T_DATAXFER</code>	—	—
<code>listen</code>	<code>T_INCON</code> ^a	—	<code>T_INCON</code> ^a	—
<code>accept1</code>	—	—	<code>T_DATAXFER</code> _{a b}	—
<code>accept2</code>	—	—	<code>T_IDLE</code> ^{b c}	—
<code>accept3</code>	—	—	<code>T_INCON</code>	—
<code>snd</code>	—	—	—	<code>T_DATAXFER</code>
<code>rcv</code>	—	—	—	<code>T_DATAXFER</code>
<code>snddis1</code>	—	<code>T_IDLE</code>	<code>T_IDLE</code> ^b	<code>T_IDLE</code>
<code>snddis2</code>	—	—	<code>T_INCON</code> ^b	—
<code>rcvdis1</code>	—	<code>T_IDLE</code>	—	<code>T_IDLE</code>
<code>rcvdis2</code>	—	—	<code>T_IDLE</code> ^b	—
<code>rcvdis3</code>	—	—	<code>T_INCON</code> ^b	—
<code>sndrel</code>	—	—	—	<code>T_OUTREL</code>
<code>rcvrel</code>	—	—	—	<code>T_INREL</code>
<code>pass_conn</code>	<code>T_DATAXFER</code>	—	—	—
<code>optmgmt</code>	<code>T_IDLE</code>	<code>T_OUTCON</code>	<code>T_INCON</code>	<code>T_DATAXFER</code>
<code>closed</code>	<code>T_UNINIT</code>	<code>T_UNINIT</code>	<code>T_UNINIT</code>	<code>T_UNINIT</code>

^a未処理の接続指示の数を増加させる。

^b未処理の接続指示の数を減少させる。

^c`t_accept` 関数の指示に従って，別のトランスポート終端に接続を引き渡す。

表 3-12: コネクション指向型トランスポート・サービスにおける状態遷移:パート 2

イベント	T_OUTREL 状態	T_INREL 状態	T_UNBND 状態
connect1	—	—	—
connect2	—	—	—
rcvconnect	—	—	—
listen	—	—	—
accept1	—	—	—
accept2	—	—	—
accept3	—	—	—
snd	—	T_INREL	—
rcv	T_OUTREL	—	—
snddis1	T_IDLE	T_IDLE	—
snddis2	—	—	—
rcvdis1	T_IDLE	T_IDLE	—
rcvdis2	—	—	—
rcvdis3	—	—	—
sndrel	—	T_IDLE	—
rcvrel	T_IDLE	—	—
pass_conn	—	—	T_DATAXFER
optmgmt	T_OUTREL	T_INREL	T_UNBND
closed	T_UNINIT	T_UNINIT	—

3.2.6 複数のプロセスおよび終端の同期

一般に、複数のプロセスを使用する場合には、注意深くプロセスの同期をとって、インタフェースの状態違反を避ける必要があります。

トランスポート・プロバイダは、1つのトランスポート終端のすべてのトランスポート・ユーザを単一ユーザとして処理しますが、次の状況が可能になります。

- 1つのプロセスで、複数のトランスポート終端を同時に作成できる。
- 複数のプロセスが、単一の終端を同時に共用できる。

単一プロセスが同期実行モードにおいて複数の終端を管理する場合、そのプロセスは、各終端におけるアクションを並列にではなく直列に管理しなければなりません。またオプションで、サーバを作成して、複数の終端を一度に管理することができます。たとえば、プロセスは、1つの終端で着信接続指示をリッスンし、別の終端で接続を受け入れることができるので、着信接続はブロックされません。このため、アプリケーションは、子プロセスをフォークして、新しい接続からの要求に対してサービスを行うことができます。

複数のプロセスが単一の終端を共用する場合には、アクションを調整して、インタフェースの状態違反を避けなければなりません。これを行うため、各プロセスは、他の関数を呼び出す前に、`t_sync` 関数を呼び出すことによって、トランスポート・プロバイダの現在の状態を検出します。すべてのプロセスをこのように調整しておかなければ、別のプロセスまたは着信イベントが、インタフェースの状態を変更する可能性があります。

同様に、複数の終端で同じプロトコル・アドレスを共用できますが、着信接続をリッスンできるのは1つの終端だけです。プロトコル・アドレスを共用している他の終端は、競合することなくデータ転送状態や接続確立状態になることができます。つまり、1つのアドレスはサーバを1つしか持てませんが、複数の終端は同じアドレスを同時に呼び出すことができます。

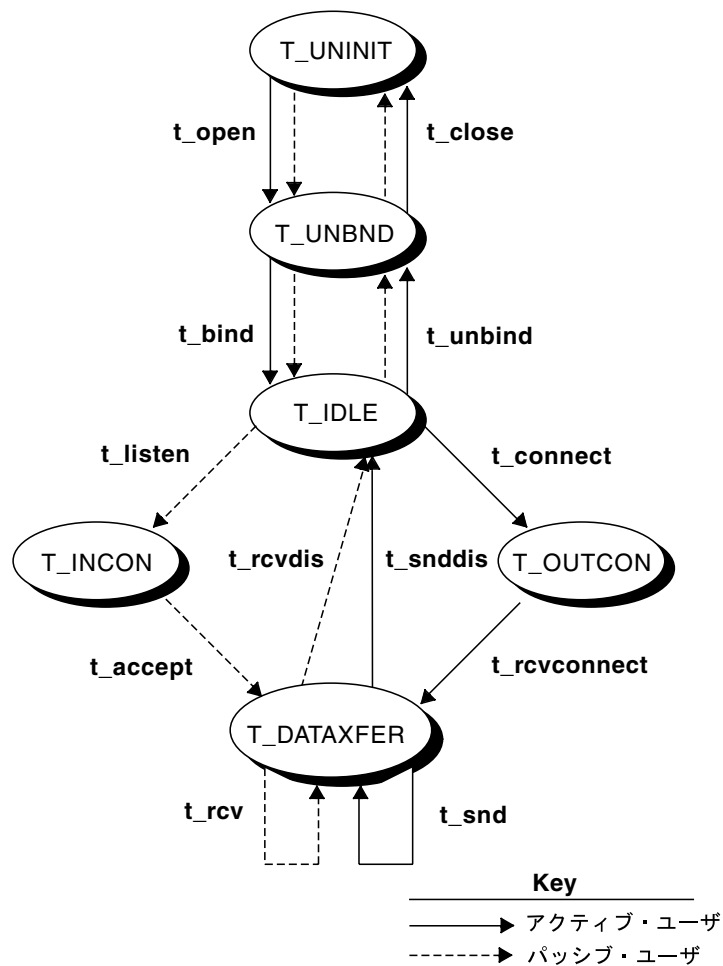
3.3 XTI の使用

この節では、まず、関数の呼び出し順序、状態管理、およびXTIのオプションの使用についてのガイドラインを示します。次に、XTIに合わせたコネクション指向型プログラムおよびコネクションレス型プログラムの作成に必要な手順について説明します。

3.3.1 関数の呼び出し順序のガイドライン

図 3-3 は、非ブロッキング・モードのコネクション指向型トランスポート・サービスを使用して通信しているアクティブ・ユーザおよびパッシブ・ユーザについて、一般的な関数の呼び出し順序および状態遷移を示しています。図中の実線は、アクティブ・ユーザの状態遷移を示し、破線は、パッシブ・ユーザの状態遷移を示しています。各線は、関数の呼び出しを表し、各楕円は、結果の状態を表します。この例には、オプションの正常解放機能は含まれていません。

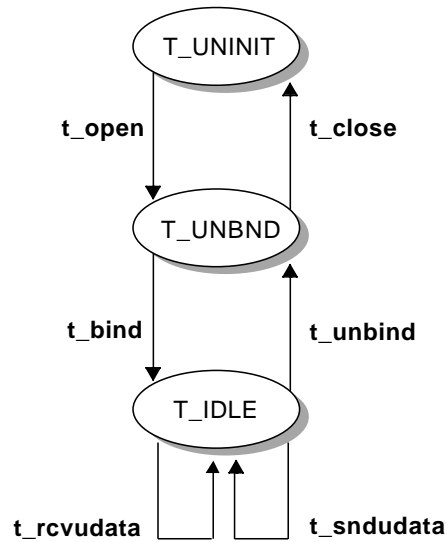
図 3-3: コネクション指向型トランスポート・サービスの状態遷移



ZK0524UJR

図 3-4 は、コネクションレス型トランスポート・サービスを使用して通信している2人のユーザについて、一般的な関数の呼び出し順序および状態遷移を示しています。図中の各線は、関数の呼び出しを表し、各楕円は、結果の状態を表します。両方のユーザは、実線で表されています。

図 3-4: コネクションレス型トランスポート・サービスの状態遷移



ZK-0525U-AI

3.3.2 トランスポート・プロバイダによる状態の管理

すべてのトランスポート・プロバイダは、状態に関して次のアクションを行わなければなりません。

- トランスポート・ユーザから見たインタフェースの状態を記録する。
- インタフェースが状態の範囲外になるような要求や応答をリジェクトして、エラーを返す。

この場合、状態は変更されません。たとえば、ユーザが関数を使用してデータを引き渡したとき、インタフェースが T_DATAXFER 状態でない場合、トランスポート・プロバイダは、データの受け入れまたは転送を行いません。

未初期化状態 (T_UNINIT) には、次の 2 つの意味があります。

- トランスポート終端の最初の状態
トランスポート・プロバイダがトランスポート終端をアクティブと見なすためには、その前に、トランスポート・ユーザがその終端を初期化してバインドしておかなければなりません。
- トランスポート終端の最後の状態

トランスポート・プロバイダは、この状態の終端を未使用と見なさなければなりません。トランスポート・ユーザが `t_close` 関数を発行すると、トランスポート・プロバイダはクローズされ、トランスポート・ライブラリに関連するリソースは、別の終端が使用できるように解放されます。

3.3.3 コネクション指向型アプリケーションの作成

この項では、コネクション・モードのアプリケーションの作成に必要な手順について説明します。

1. 終端の初期化
2. 接続の確立
3. データの転送
4. 接続の解放
5. 終端の初期化解除

3.3.3.1 終端の初期化

終端を初期化するには、次の手順に従ってください。

1. 終端をオープンする。
2. アドレスを終端にバインドする。
3. プロトコル・オプションを折衝する。

ここでは、コネクション指向型サービス用に終端を初期化する手順について説明します。この手順は、コネクションレス型サービスの場合も同じです。

トランスポート終端のオープン

コネクション指向型とコネクションレス型のどちらのアプリケーションも、`t_open` 関数を使用して、トランスポート終端をオープンしなければなりません。

関数の構文、パラメータ、エラーについては、`t_open(3)` を参照してください。XTIのエラーの概要の説明については、3.7 節を参照してください。

このXTIのインプリメンテーションは、デバイス・スペシャル・ファイルへのパス名を使用して、トランスポート・プロバイダを識別します。これ

は AT&T の TLI と同じ方法です。TCP または UDP のトランスポート・プロバイダとデータのやりとりをするデバイス・スペシャル・ファイルは、`/dev/streams/xtiso` ディレクトリにあります。異なるトランスポート・プロバイダを使用する場合には、そのマニュアルを参照して、適切なデバイス名を指定してください。

注意

XTI/TLI 以外の方式での特殊デバイスの使用 (たとえば、`open`、`read`、`write` などの直接呼び出しなど) は違法であり、定義されていない結果が生じます。

XTI の仕様では、`O_RDONLY` または `O_WRONLY` を使用して、終端を読み取り専用または書き込み専用にすることを禁止しています。

プロトコルに依存しないプログラムを作成する場合は、`t_open` 関数が返す `t_info` 構造体の情報にアクセスすることによって、データ・バッファ・サイズを決定できます。トランスポート・ユーザが、許容されているデータ・サイズを超えた場合には、エラーが返されます。代わりに、`t_alloc` 関数を使用して、データ・バッファを割り当てすることもできます。

次は、TCP トランスポート・プロバイダ (XNS4.0) 用の `t_open` 関数の例です。

```
if ( (newfd = t_open( "/dev/streams/xtiso/tcp+", O_RDWR, NULL) ) == -1 )
{
    (void) t_error("could not open tcp transport");
    exit (1);
}
```

終端へのアドレスのバインド

一度、終端をオープンすると、`t_bind` 関数を使用してプロトコル・アドレスをその終端にバインドする必要があります。アドレスをバインドすることによって、終端をアクティブにします。コネクション・モードでは、トランスポート・プロバイダに、接続指示の受け入れ、またはトランスポート終端上での接続要求のサービスを開始するように指示することもできます。`t_listen` 関数を発行して、トランスポート・プロバイダが接続指示を受け入れたかどうかを確認することができます。コネクションレス・モードでは、一度、アドレスをバインドすると、トランスポート終端を通してデータ・ユニットの送受信ができます。

関数の構文、パラメータ、エラーについては、`t_bind(3)` を参照してください。XTI のエラーの概要の説明については、3.7 節 を参照してください。

トランスポート・プロバイダがアドレスを生成するかどうかを判断するには、`t_bind` 関数でアドレスを指定しません (`req` に空ポインタを設定します)。トランスポート・プロバイダがアドレスを生成する場合、この関数は、割り当てられたアドレスを `ret` フィールドに返します。トランスポート・プロバイダがアドレスを生成しない場合には、`TNOADDR` エラーを返します。

接続指示のリッスンに使用する終端上で接続を受け入れる場合、バインドされたアドレスは、接続中はビジーになります。最初のリッスンに使用した終端が、データをアクティブに送信しているか、または `T_IDLE` 状態のときは、リッスン用に他の終端を同じアドレスにバインドすることはできません。

TCP または UDP のいずれかが基礎となるトランスポート・プロバイダの場合には、4.2.3.2 項で説明する `gethostbyname` ルーチンを使用して、ホスト情報を取得することができます。

`gethostbyname` ルーチン以外の方法を使用してホスト情報を検索する場合には、次の点に考慮してください。

- アプリケーションは、トランスポート・プロバイダが指定するフォーマットでソケット・アドレスを XTI 関数に引き渡さなければならない。
TCP/IP 上の XTI の場合には、指定のアドレス・フォーマットは `sockaddr_in` 構造体です。
- アプリケーションは、XTI 関数にトランスポート・プロバイダ識別子も引き渡さなければならない。
オペレーティング・システムは、この識別子がトランスポート・プロバイダ用のデバイス特殊ファイルへのパス名のフォーマットになっていると考えます。

3.3.3.2 XTI のオプションの使用

XPG3、XNS4.0 および XNS5.0 では、オプション管理のインプリメント方法が異なります。

XPG3 では、オプション管理は `t_optmgmt` 関数だけが取り扱えます。XNS4.0 および XNS5.0 では、いくつかの関数が引数 `opt` を持ち、この引数を使用してトランスポート・ユーザとトランスポート・プロバイダとの間でオプションをやりとりできます。

詳細は、3.6.7 項を参照してください。

3.3.3.3 接続の確立

一般には、次の手順で接続を確立します。

1. パッシブ・ユーザ、つまりサーバが接続要求をリッスンする。
2. アクティブ・ユーザ、つまりクライアントが接続を開始する。
3. パッシブ・ユーザ、つまりサーバが、接続要求を受け入れて、接続指示が受信される。

これらの手順について、次の各項で説明します。

接続指示のリッスン

パッシブ・ユーザは、`t_listen` 関数を発行して、キューに入っている接続指示を検索します。`t_listen` 関数が、キューの先頭で接続指示を見つけた場合は、接続指示に関する詳細な情報、およびその指示を識別するローカル・シーケンス番号を返します。キューにいれることが可能な未処理の接続指示の数は、`t_bind` 関数の発行時にトランスポート・プロバイダが受け入れた、`qlen` パラメータの値によって制限されます。

省略時には、`t_listen` 関数は同期モードで実行され、接続指示が到着するのを待ってから制御をユーザに戻します。`t_open` 関数または `fcntl` 関数の `O_NONBLOCK` フラグを設定して非同期モードを指定している場合には、`t_listen` 関数は、接続指示があるかどうかをチェックして、接続指示がなければ `TNODATA` エラーを返します。

関数の構文、パラメータ、エラーについては、`t_listen(3)` を参照してください。XTIのエラーの概要の説明については、3.7 節を参照してください。

接続の開始

接続は、同期または非同期のいずれかのモードで開始されます。同期モードでは、アクティブ・ユーザが `t_connect` 関数を発行すると、この呼び出しは、パッシブ・ユーザの応答を待ってから、アクティブ・ユーザに制御を戻します。非同期モードでは、`t_connect` は接続を開始しますが、接続に対する応答が到着する前に、アクティブ・ユーザに制御を戻します。このとき、アクティブ・ユーザは、`t_rcvconnect` 関数を発行して、接続要求の状態を判別することができます。パッシブ・ユーザが要求を受け入れた場合、

`t_rcvconnect` 関数は正常に戻り、接続確立フェーズが完了します。応答をまだ受信していない場合は、`TNODATA` エラーを返します。アクティブ・ユーザは、後で `t_rcvconnect` 関数を再度発行する必要があります。

関数の構文、パラメータ、エラーについては、`t_connect(3)` を参照してください。XTI のエラーの概要の説明については、3.7 節を参照してください。

`t_connect` 関数は、正常終了すると 0 を返します。それ以外の場合は -1 を返し、変数 `t_errno` に、3.7 節に示されている値の 1 つが設定されます (マルチスレッド・アプリケーションの場合、`t_errno` はスレッド固有です)。

接続の受け入れ

パッシブ・ユーザが接続指示を受け入れると、同じ終端 (`t_listen` を使用してリッスンしていた終端)、または別の終端で `t_accept` 関数を発行できます。

パッシブ・ユーザが同じ終端で受け入れると、その終端はそれ以上は着信接続指示を受信することも、キューにいれることもできなくなります。その終端にバインドされたプロトコル・アドレスは、終端がアクティブな間はビジーのままです。リッスンに使用している終端と同じプロトコル・アドレスに、トランスポート終端をバインドすることはできません。つまり、パッシブ・ユーザが `t_unbind` 関数を発行するまで、他の終端はバインドできません。さらに、同じ終端で接続を受け入れるためには、パッシブ・ユーザは、(`t_accept` 関数または `t_snddis` 関数のいずれかを使用して) 以前に受信したすべての接続指示に応答しなければなりません。それ以外の場合、`t_accept` は、`TBADF` エラーを返します。

パッシブ・ユーザが別の終端で接続を受け入れた場合には、リッスンに使用している終端は、引き続き着信接続要求を受信してキューにいれることができます。別の終端は、前もってプロトコル・アドレスにバインドして、`T_IDLE` 状態になっていなければなりません。プロトコル・アドレスが、接続指示を受信した終端と同じ場合には、`qlen` パラメータにゼロ (0) を設定しなければなりません。

どちらの終端を使用する場合にも、受信を待っている接続指示または切断指示があるときには、`t_accept` 関数は失敗し、`TLOOK` エラーを返します。

関数の構文、パラメータ、エラーについては、`t_accept(3)` を参照してください。XTI のエラーの概要の説明については、3.7 節を参照してください。

3.3.3.4 データの転送

2つの終端間に一度接続が確立されると、アクティブ・ユーザとパッシブ・ユーザは、その接続を通して全二重方式でデータを転送できます。コネクション指向型サービスのこのフェーズは、データ転送フェーズといいます。これ以降の項では、データ転送フェーズでデータの送受信を行う方法について説明します。

データの送信

トランスポート・ユーザは、`t_snd` 関数を使用して、通常データまた優先データのいずれかを接続を通して送信できます。通常では、トランスポート・プロバイダがすべてのデータをただちに受け入れることができる場合、`t_snd` は正常に送信して、受け入れられたバイト数を返します。データがただちに受け入れられない場合、`t_snd` の結果は、同期モードまたは非同期モードのどちらで実行していたかによって異なります。

省略時には、`t_snd` 関数は同期モードで実行され、フロー制御条件によってトランスポート・プロバイダがデータを受け入れることができない場合は待ちます。この関数は、次の条件の1つが真になるまでブロックします。

- フロー制御条件がクリアであり、トランスポート・プロバイダは新しいデータ・ユニットを受け入れることができる。

この場合、`t_snd` 関数は正常に戻ります。

- 切断指示が受信された。

`t_snd` 関数は、`TLOOK` エラーを指定して戻ります。`t_look` 関数を呼び出すと、`T_DISCONNECT` イベントが返されます。転送中のデータはすべて失われます。

- 内部問題が発生した。

`t_snd` 関数は、`TSYSERR` エラーを指定して戻ります。転送中のデータはすべて失われます。

終端の作成時に `O_NONBLOCK` フラグがセットされていれば、`t_snd` は非同期モードで実行され、フロー制御制限がある場合はすぐに失敗します。場合によっては、データの一部だけがトランスポート・プロバイダに受け入れられ、`t_snd` は、送信要求したバイト数より少ない値を返すことがあります。このときには、ユーザは次のうちの1つを行います。

- 残りのデータを指定して、`t_snd` をもう一度発行する。

- `t_look` 関数を使用して、フロー制御制限が解除されたかどうかをチェックしてから、データを再送信する。

`t_look` 関数については、この章の最後で説明します。

関数の構文、パラメータ、エラーについては、`t_snd(3)` を参照してください。XTI のエラーの概要の説明については、3.7 節 を参照してください。

データの受信

トランスポート・ユーザは、`t_rcv` 関数を使用して、通常データまたは優先データのいずれかを接続を通して受信できます。通常では、データが受信できる場合、`t_rcv` はデータを返します。接続が切断されている場合は、`t_rcv` は、ただちにエラーを指定して戻ります。データは受信できないが、接続が存在している場合には、`t_rcv` の動作は、実行モードに応じて異なります。

- 省略時には、`t_rcv` は同期モードで実行され次のうちの 1 つが到着するのを待つ。
 - データ
 - 切断指示
 - シグナル

`t_rcv` を発行して待つ代わりに、`t_look` 関数を発行して、`T_DATA` または `T_EXDATA` イベントをチェックすることができます。

- `O_NONBLOCK` フラグをセットした場合、`t_rcv` は非同期モードで実行される。

データが受信できない場合は、`TNODATA` エラーを返して失敗します。その場合には、`t_rcv` 関数または `t_look` 関数を発行することによって、データのポールを継続する必要があります。

関数の構文、パラメータ、エラーについては、`t_rcv(3)` を参照してください。XTI のエラーの概要の説明については、3.7 節 を参照してください。

3.3.3.5 接続の解放

XTI では、2 つの接続の解放方法をサポートしています。つまり、打ち切りによる解放と正常解放です。すべてのトランスポート・プロバイダは、打ち切りによる解放をサポートします。正常解放は、XTI ではオプション機能であるため、トランスポート・プロバイダの中には、これを提供しないものがあります。たとえば、OSI トランスポートは打ち切りによる解放

だけをサポートしますが、TCP は打ち切りによる解放と、オプションとして正常解放をサポートします。

打ち切りによる解放

打ち切りによる解放は、トランスポート・ユーザまたはトランスポート・プロバイダから要求でき、接続をただちに打ち切ります。打ち切りによる解放は折衝できないため、一度打ち切りによる解放を要求すると、ユーザ・データが引き渡される保証はありません。

トランスポート・ユーザは、接続確立フェーズまたはデータ転送フェーズのいずれかで、打ち切りによる解放を要求できます。接続確立フェーズでは、トランスポート・ユーザは、打ち切りによる解放を使用して接続要求をリジェクトすることができます。データ転送フェーズでは、どちらかのユーザがいつでも接続を解放できます。トランスポート・プロバイダが打ち切りによる解放を要求すると、接続がもう存在しないことが両方のユーザに通知されます。

関数の構文、パラメータ、エラーについては、`t_snddis(3)` を参照してください。XTI のエラーの概要の説明については、3.7 節を参照してください。

トランスポート・ユーザは、`T_DISCONNECT` イベントを通じて、打ち切りによる解放について通知されます。プログラムが `T_DISCONNECT` イベントを受信した場合には、`t_rcvdis` 関数を発行して、切断に関する情報を検索し、`T_DISCONNECT` イベントを消費しなければなりません。

関数の構文、パラメータ、エラーについては、`t_rcvdis(3)` を参照してください。XTI のエラーの概要の説明については、3.7 節を参照してください。

正常解放

正常解放を使用すると、データを損失することなく接続を解放することができます。正常解放はすべてのトランスポート・プロバイダが提供している機能ではありません。トランスポート・プロバイダが、`t_open` 関数または `t_getinfo` 関数で `T_COTS_ORD` のサービス・タイプを返す場合には、正常解放がサポートされます。トランスポート・ユーザは、データ転送フェーズで正常解放を要求できます。正常解放の一般的な手順は次のとおりです。

1. アクティブ・ユーザが `t_sndrel` 関数を発行して、接続の正常解放を要求する。

2. パッシブ・ユーザは、アクティブ・ユーザからの正常解放要求を示す T_ORDREL イベントを受信すると、t_rcvrel 関数を発行して、要求を受信したことを示し、T_ORDREL イベントを消費する。
3. 切断の準備ができたなら、パッシブ・ユーザは、t_sndrel 関数を発行する。
4. アクティブ・ユーザは、t_rcvrel 関数を発行して応答する。

トランスポート・ユーザは t_sndrel 関数を発行した後に、その接続を通してそれ以上データを送信することはできません。ただし、正常解放指示 (T_ORDREL イベント) を受信するまで、データの受信を継続することはできます。

関数の構文、パラメータ、エラーについては、t_sndrel(3) を参照してください。XTI のエラーの概要の説明については、3.7 節を参照してください。

正常解放指示の受信に肯定応答するには、t_rcvrel 関数を発行します。

トランスポート・ユーザは、正常解放指示 (T_ORDREL) を受信した後に、それ以上データを受信できません。トランスポート・ユーザがデータを受信しようとする、その関数は無期限にブロックします。ただし、トランスポート・ユーザは、t_sndrel 関数を発行するまで、データの送信を継続することができます。

関数の構文、パラメータ、エラーについては、t_rcvrel(3) を参照してください。XTI のエラーの概要の説明については、3.7 節を参照してください。

3.3.3.6 終端の初期化解除

終端の使用が終了したら、t_unbind および t_close 関数を使用して、終端をアンバインドし、クローズすることにより、その終端の初期化を解除します。ここでは、コネクション指向型サービスを使用して終端の初期化を解除する手順について説明しますが、コネクションレス型サービスの場合も手順は同じです。

終端をアンバインドしたら、終端は使用不能になり、トランスポート・プロバイダはそれ以上要求を受け入れません。

関数の構文、パラメータ、エラーについては、t_unbind(3) を参照してください。XTI のエラーの概要の説明については、3.7 節を参照してください。

終端をクローズすることによって、トランスポート・プロバイダにその終端の使用が終了したことを通知し、その終端に関連するライブラリ・リソースをすべて解放します。

終端が T_UNBND 状態の場合は、`t_close` を呼び出さなければなりません。ただし、この関数は状態情報をチェックしないので、どの状態からでも呼び出して、トランスポート終端をクローズすることができます。

T_UNBND 状態ではない終端をクローズすると、その終端に関連するライブラリ・リソースが自動的に解放されて、その終端に関連するファイルがクローズされます。そのプロセスまたはその終端を参照するプロセスに他の記述子がない場合、トランスポート接続は切断されます。

関数の構文、パラメータ、エラーについては、`t_close(3)` を参照してください。XTI のエラーの概要の説明については、3.7 節を参照してください。

3.3.4 コネクションレス型アプリケーションの作成

この項では、コネクションレス・モードのアプリケーションを作成するために必要な手順について説明します。

1. 終端の初期化
2. データの転送
3. 終端の初期化解除

3.3.4.1 終端の初期化

コネクションレス型アプリケーションのために終端を初期化する手順は、コネクション指向型アプリケーションの場合と同じです。コネクションレス型アプリケーションのために終端を初期化する方法についての詳細は、3.3.3.1 項を参照してください。

3.3.4.2 データの転送

コネクションレス型サービスのデータ転送フェーズは、次の操作から構成されています。

- 他のユーザへのデータの送信
- 他のユーザからのデータの受信
- 以前に送信したデータに関するエラー情報の検索

コネクションレス型サービスについては、次のことに注意してください。

- このサービスでは、優先データをサポートしていない。
- このサービスでは、T_UDERR、T_DATA、およびT_GODATA のイベントだけを報告する。

データの送信

t_sndudata 関数は、同期モードまたは非同期モードで実行できます。同期モードで実行している場合、t_sndudata は、トランスポート・プロバイダが別のデータグラムを受け入れることができるときに、制御をユーザに戻します。場合によっては、この状態になるまで少しの間、この関数はブロックします。非同期モードでは、フロー制御制限が存在する場合、トランスポート・プロバイダは、新しいデータグラムの送信を拒否します。この場合、t_sndudata 関数は TFLOW エラーを返します。したがって、ユーザは、後でこの呼び出しを再試行するか、または t_look 関数を発行して、フロー制御制限が解除されたかどうかを確かめなければなりません。フロー制御制限の解除は、T_GODATA または T_GOEXDATA のイベントによって示されます。

t_bind 関数を使用して終端をアクティブにする前に、データ・ユニットを送信しようとする、トランスポート・プロバイダはそのデータを破棄します。

関数の構文、パラメータエラーについては、t_sndudata(3) を参照してください。XTIのエラーの概要の説明は、3.7 節

データの受信

t_rcvudata 関数を呼び出しデータが受信できる場合、t_rcvudata は、受信オクテット数を示して、すぐに戻ります。データを受信できない場合の t_rcvudata の動作は、実行モードに応じて次のように異なります。

- 同期モード

t_rcvudata 関数は、データグラム、エラー、またはシグナルのいずれか 1 つを受信するまでブロックします。t_rcvudata が戻るのを待つ代わりに、t_look 関数を定期的に発行して T_GODATA イベントを確かめた後、t_rcvudata を発行してデータを受信することもできます。

- 非同期モード

`t_rcvudata` は、エラーを指定してただちに戻ります。この場合、ユーザはこの関数を定期的に再試行するか、または `t_look` 関数を使用して着信データをポーリングしなければなりません。

関数の構文、パラメータ、エラーについては、`t_rcvudata(3)` を参照してください。XTL のエラーの概要の説明は、3.7 節を参照してください。

エラー情報の検索

`t_look` を発行して `T_UDERR` イベントを受信した場合は、以前に送信したデータでエラーが生じています。エラーをクリアして `T_UDERR` イベントを消費するには、`t_rcvuderr` 関数を発行する必要があります。この関数はまた、エラーを生じたデータとエラーの性質に関する情報も返します（ただしこれは指定した場合のみです）。

データ・ユニットに関するエラー情報を受信するには、`t_rcvuderr` 関数を発行します。

関数の構文、パラメータ、エラーについては、`t_rcvuderr(3)` を参照してください。XTI のエラーの概要の説明については、3.7 節を参照してください。

3.3.4.3 終端の初期化解除

コネクションレス型アプリケーションのために終端の初期化を解除する手順は、コネクション指向型アプリケーションの場合と同じです。コネクションレス型アプリケーションのために終端の初期化を解除する方法についての詳細は、3.3.3.6 項を参照してください。

3.4 フェーズに依存しない関数

XTI は、コネクション指向型サービスまたはコネクションレス型サービスの任意のフェーズ（初期化されていない状態を除く）で発行できる関数を、多数提供しています。これらの関数は、インタフェースの状態には影響を与えません。このような関数の一覧と簡単な説明を、表 3-13 に示します。

表 3-13: フェーズ独立関数

関数	説明
<code>t_getinfo</code>	終端に関連するトランスポート・プロバイダの特性に関する情報を返す。
<code>t_getprotaddr</code> ^a	プロトコル・アドレスを返す。
<code>t_getstate</code>	終端の現在の状態を返す。
<code>t_strerror</code> ^a	エラー・メッセージ文字列を作成する。
<code>t_sync</code>	トランスポート・プロバイダからの情報と、トランスポート・ライブラリによって管理されるデータ構造体の同期をとる。
<code>t_sysconf</code> ^b	構成可能な XTI 限界値またはオプションの現在値を返す。
<code>t_alloc</code>	指定されたデータ構造体用にストレージを割り当てる。
<code>t_free</code>	以前に <code>t_alloc</code> によって割り当てられたデータ構造体用のストレージを解放する。
<code>t_error</code>	XTI 関数によって返された最新のエラーを記述するメッセージを出力する。(オプション)
<code>t_look</code>	終端に関連する現在のイベントを返す。

^aこの関数は XNS4.0 でだけサポートされている。

^bこの関数は XNS5.0 でだけサポートされている。

`t_getinfo` 関数および `t_getstate` 関数は、重要な情報の検索に役立ちます。 `t_getinfo` 関数は、`t_open` と同様にトランスポート・プロバイダに関する情報を返します。 `t_open` が初期化フェーズだけで呼び出すことができるのに対して、`t_getinfo` は、通信のどのフェーズでも呼び出すことができます。 関数が TOUTSTATE エラーを返して、終端が適切な状態にないことを示している場合には、`t_getstate` を発行して現在の状態を検出し、その状態に対して適切なアクションをとることができます。

`t_sync` 関数は、次のことを実行できます。

- 基礎となるトランスポート・プロバイダからの情報とトランスポート・ライブラリによって管理されるデータ構造体の同期をとる。
- 共同して動作している 2 つのプロセスが、トランスポート・プロバイダとのやりとりの同期をとることを許可する。

`t_alloc` 関数および `t_free` 関数は、メモリの割り当ておよび解放に便利です。 これらの関数では、サイズに関する情報ではなく XTI 構造体の名前を指定できます。 `t_alloc` および `t_free` を使用して XTI 構造体のメモリを

管理していれば、XTI 構造体が将来のリリースにおいて変更された場合でも、プログラムを変更する必要がありません。

`t_error` を使用すると、`t_errno` の内容に加えて、ユーザ指定のメッセージ (説明) を標準出力にプリントできます。

`t_look` は、終端に関連する現在の未処理のイベントを検出する、重要な関数です。一般に、XTI 関数がエラーとして `TLOOK` を返して、重要な非同期イベントが発生したことを示すと、トランスポート・ユーザは、`t_look` 関数を発行してそのイベントを検出します。イベントについての詳細は、3.2.3 項を参照してください。

3.5 XTI へのポート

この節では、次の事柄について説明します。

- XTI に合わせたプログラムを作成するときのガイドライン
- XTI と TLI の互換性に関する情報
- ソケット・アプリケーションから XTI 使用アプリケーションへの書き換えに関する情報

3.5.1 プロトコル独立およびポータビリティ

XTI は、使用する特定のトランスポート・プロトコルに依存しないインタフェースを提供するために設計されました。基礎となるトランスポート・プロバイダがサポートする XTI の関数および機能の任意のサブセットに応じて、動作を変更できるアプリケーションを作成できます。

プロバイダは、XTI の機能をすべて備えていなくてもかまいません。したがって、アプリケーション・プログラマは、次のガイドラインに従って、XTI アプリケーションを作成する必要があります。

- XTI の必須機能の関数だけを使用する。

すべてのトランスポート・プロバイダで備えていない機能をアプリケーションで使用すると、トランスポート・プロバイダや XTI のインプリメンテーションによっては、その機能を使用できない場合があります。

たとえば、正常解放機能 (`t_sndrel` および `t_rcvrel` 関数) は、すべてのコネクション指向型のトランスポート・プロトコルがサポートしているとは限りません。特に、ISO プロトコルはサポートしていません。複

数のプロトコルを使用する環境でアプリケーションを実行する場合には、正常解放機能は使用しないでください。

一般的にサポートされていない機能も使用する場合には、各トランスポート・プロバイダがサポートする XTI 関数のサブセットに応じて動作を変更するように、アプリケーションを作成してください。

- 論理データ境界が接続を通して保存されていると想定しない。
TCP など、トランスポート・プロバイダによっては、TSDU の概念をサポートしていないものがあります。そのためそのトランスポート・プロバイダで関数 `t_snd`、`t_sndudata`、`t_rcv`、および `t_rcvudata` を使用する場合、`T_MORE` フラグは無視されます。
- `t_open` 関数および `t_getinfo` 関数によって返される、プロトコル固有のサービス制限を超えないようにする。
アプリケーションで、データを転送する前にこれらの制限を検索し、通信プロセス全体を通してこれらの制限を厳守していることを確認してください。
- プロトコル固有のオプションに依存しない。
`t_optmgmt` 関数を使用すると、アプリケーションはトランスポート・プロバイダから省略時のプロトコル・オプションにアクセスし、これらのオプションを引数として接続確立関数に引き渡すことができます。ただしアプリケーションでそのオプションを調べたり、特定のオプションに依存したりしないようにしてください。
- `t_rcvdis` 関数に関連する理由コードや `t_rcvuderr` 関数に関連するエラー・コードを解釈しない。
これらのコードは基礎となるプロトコルに依存しているので、プロトコル独立を実現するためには、アプリケーションでこれらのコードを解釈しないようにしてください。
- `t_open` 関数によって返されるファイル記述子に対しては、XTI オペレーションだけを行う。
これ以外のオペレーションを行うと、システムごとに異なった結果を生じる可能性があります。

以降の項で、異なるトランスポート・レベルのプログラミング・インタフェースから XTI にアプリケーションをポートする方法について説明します。特に、最も一般に使用される 2 つのトランスポート・レベルのプログラミン

グ・インタフェースからポートする方法について説明します。この2つのインタフェースとは、多数の UNIX System V アプリケーションで使用されているトランスポート層インタフェース (TLI)，および多数のバークレイ版 UNIX アプリケーションで使用されている 4.3BSD ソケット・インタフェースです。

以降の項で示される情報は、ユーザが、TLI またはソケットを使用したプログラミングに習熟していること、および基本的な XTI の概念および構文を理解していることを前提としています。

3.5.2 XTI と TLI の互換性

この項では、TLI プログラムを再コンパイルする前に考慮すべき事項について説明するとともに、再コンパイルの方法について説明します。長期的な観点から考えた場合は、TLI インタフェースではなく XTI インタフェースを使用することをお勧めします。XTI を使用するアプリケーションおよびトランスポート・プロバイダが多くなるにつれ、その利点が明らかになります。

XTI と TLI は同じ関数、状態、およびサービス・モードをサポートします。XTI または TLI アプリケーションを XTI または TLI ライブラリとリンクする場合には、シェアード・ライブラリ・サポートが省略時の設定です。シェアード・ライブラリ・サポートの詳細については、3.2.2 項を参照してください。

TLI プログラムを再コンパイルする前に、次のようなイベント管理について、プログラムの現在のインプリメンテーションを検討する必要があります。つまり、System V UNIX オペレーティング・システムは、イベント管理用のツールとして、`poll` 関数を提供します。Tru64 UNIX の XTI インプリメンテーションは、`poll` 関数をサポートしているので、アプリケーションでこの関数を使用している場合は、再コンパイルできます。プログラムが、独自のイベント管理機構を使用している場合は、そのイベント管理機構を Tru64 UNIX にポートするか、または Tru64 UNIX が提供するポーリング機構に変更する必要があります。

Tru64 UNIX の TLI のインプリメンテーションは、AT&T の TLI とソース・レベルで互換性があるため、Tru64 UNIX の TLI ライブラリを使用して TLI プログラムを再コンパイルすることができます。次の手順に従ってください。

1. TLI ヘッダ・ファイルがソース・コードにインクルードされていることを確認する。

```
#include <tli/tiuser.h>
```

2. 次のコマンド構文を使用して、アプリケーションを再コンパイルする。

```
cc -o name name.c -ltli
```

TLI アプリケーションを XTI アプリケーションに変更する場合には、次に示す TLI と XTI のわずかな相違を認識しておく必要があります。

- `t_error` は、XTI では整数値 (正常終了時は 0, 異常時は -1) を返す `int` 型の関数であり、TLI では `void` 型のプロシージャである。
- XTI では、`t_look` は、(TLI とは異なり) `T_ERROR` イベントをサポートしない。`T_ERROR` イベントの代わりに、-1 および `t_errno` を返す。
- `t_open` 関数の `oflag` パラメータで、TLI の `O_NDELAY` 値は、XTI では `O_NONBLOCK` 値として認識される。

- XTI は、読み取り書き込みアクセス許可を持つ終端をオープンする。

これは、ほとんどの関数がトランスポート・プロバイダに対して読み取り書き込みアクセスを必要とするからです。TLI は、読み取り専用、書き込み専用、読み取り書き込みのいずれかのアクセス許可でオープンします。特に、`t_open` 関数においては、XTI は、`oflag` パラメータの値として、`O_RDWR` および `O_NONBLOCK` のビット単位の論理和を使用します。一方、TLI は、`O_NDELAY` と、`O_RDONLY`、`O_WRONLY`、`O_RDWR` のいずれか 1 つとのビット単位の論理和を使用します。`O_RDONLY` および `O_WRONLY` の値は、XTI では使用できません。XTI では、`O_RDWR` が、終端のアクセス用として唯一の有効な値です。

- TLI では、トランスポート・プロバイダに自動アドレス生成機能があると想定されているが、XTI では想定されていない。

トランスポート・プロバイダに自動アドレス生成機能がない場合には、競合する要求が発行されると、XTI は適切なエラー・メッセージを返すことができます。

- XTI は、TCP/IP プロトコルおよび OSI プロトコルに対して、プロトコル固有の情報を定義する。

Tru64 UNIX の XTI インプリメンテーションでは、STREAMS ベースのプロトコルに対して、プロトコル固有のオプションもサポートしていますが、TLI は、そのような情報は提供しません。

- XTI は、フロー制御を管理するために、`T_GODATA` や `T_GOEXDATA` などの追加のイベントを提供するが、TLI では、正常終了するまで送信を続行する。

- XTI は、より正確なエラー情報をアプリケーションに伝えるために、追加のエラー・メッセージを提供する。

終端の状態を変更するすべての関数は、TOUTSTATE エラーを使用して、終端の状態が正しくないときに関数が呼び出されたことを示します。XTI 関数の中には、TLOOK エラーを返して、緊急の非同期イベントが発生したことを示すものがあります。TLI では、多少面倒ですが、呼び出しの前に明示的に `t_look` 関数を呼び出すか、または TLOOK イベント用にシグナルをセットしなければなりません。キューに入っている接続要求がない場合には、TBADQLEN エラーが返されて、アプリケーションが `t_listen` 関数を発行した後に永遠に待ち続けることを防ぎます。エラー・メッセージについては、XTI のリファレンス・ページを参照してください。

TLI アプリケーションを真の XTI アプリケーションにするには、次のことを行います。

1. TLI ヘッダ・ファイルの代わりに XTI ヘッダ・ファイルをソース・コードにインクルードする。

```
#include <xti.h>
```

2. TLI と XTI の相違から必要となる、すべての変更または拡張をプログラムに加える。

3. 次のコマンドを使用して、アプリケーションを再コンパイルする。

```
cc -o name name.c -lxti
```

3.5.3 ソケット・アプリケーションから XTI 使用アプリケーションへの書き換え

この項では、ソケット・インタフェースと XTI との相違について説明します。ここでは、アプリケーションが標準 4.3 BSD ソケット・インタフェースを使用していることを前提としており、ソケット・インタフェースに加えた拡張や変更は考慮していません。ソケットと XTI のサーバおよびクライアントの例については、付録 B を参照してください。

XTI には、ソケット・インタフェースと共通の関数が多数あります。ただし、XTI を使用できるようにアプリケーションを書き換える場合には、XTI と現在のソケット・インタフェースとのあらゆる相違を認識しておく必要があります。

XTI では、30 個の関数を提供しています。ソケット関数に対応する XTI 関数は 14 個ありますが、そのうち 6 個の関数にはわずかな相違があります。表 3-14 は、各 XTI 関数と、それに対応するソケット関数 (存在する場合)、およびその 2 つの関数が同じ意味を持つかどうかを示した一覧です。一般に、ソケット関数はパラメータを値で渡しますが、ほとんどの XTI 関数は、入出力パラメータの構造体を指し示すポインタを渡します。

表 3-14: XTI 関数とソケット関数の比較

XTI 関数	ソケット関数	意味
t_accept	accept	異なる
t_alloc	—	—
t_bind	bind	異なる
t_close	close	同じ
t_connect	connect	同じ
t_error	—	—
t_free	—	—
t_getinfo	—	—
t_getstate	—	—
t_listen	listen, accept	同じ ^a
t_look	select	異なる
t_open	socket	同じ
t_optmgmt	setsockopt, getsockopt	異なる
t_rcv	recv	同じ
t_rcvconnect	—	—
t_rcvdis	—	—
t_rcvrel	—	—
t_rcvreldata	—	—
t_rcvudata	recvfrom	同じ
t_rcvuderr	—	—
t_rcvv	recv	同じ
t_rcvvudata	recv	同じ
t_snd	send	同じ

表 3-14: XTI 関数とソケット関数の比較 (続き)

XTI 関数	ソケット関数	意味
t_snddis	shutdown	異なる
t_sndrel	—	—
t_sndreldata	—	—
t_sndudata	sendto	同じ
t_sndv	send	同じ
t_sndvudata	send	同じ
t_sync	—	—
t_sysconf	—	—
t_unbind	—	—

^aXTI では、t_listen 関数は、キューの長さを示すパラメータを指定するとともに着信接続を待つ。ソケットでは、listen 関数はキューの長さを示すパラメータを指定するだけである。

対応するソケット関数と意味が異なる XTI 関数は、次の点で異なります。

t_accept

t_accept 関数は、ユーザ指定の *resfd* 引数を取り、リモート終端との接続を確立する。一方、ソケットからの accept 呼び出しは、接続が確立されるファイル記述子の選択をシステムに命じる。また t_accept 関数は、接続指示が受信されている場合に発行される。したがって、この関数はブロックしない。逆に、accept 呼び出しは接続要求を見越して発行されるため、接続要求が発生するまでブロックすることがある。

t_bind

XTI は、1 つのプロトコル・アドレスを複数の終端にバインドできるが、ソケット・インタフェースでは、1 つのアドレスは 1 つのソケットにしかバインドできない。

t_look

t_look 関数は、現在のイベントを返す。現在のイベントは、T_LISTEN、T_CONNECT、T_DATA、T_EXDATA、T_DISCONNECT、T_UDERR、T_ORDREL、T_GODATA、および T_GOEXDATA のいずれかである。poll 関数は、トランスポート終端で着信イベントを監視するときに使用できる。select 呼び出しは、単

一の記述子が読み取り用または書き込み用に準備できているかどうか、または例外条件が保留中かどうかを調べる場合に使用できる。

`t_snddis`

`t_snddis` 関数は、確立された接続での打ち切りによる解放を開始するか、または接続要求をリジェクトする。XTI プログラムは `t_snddis` 関数を発行した後も、`t_listen` 関数を使用して要求のリッスンを継続したり、`t_connect` 関数を使用して接続を再確立することができる。ソケットでは、`shutdown` 呼び出しおよび `close` 呼び出しを使用して接続を一度シャットダウンすると、この接続に割り当てられたすべてのローカル・リソースが自動的に解放される。したがって、接続のリッスンを継続したり、接続を確立するために、プログラムは、`socket` 呼び出しと `bind` 呼び出しを再発行する必要がある。

XTI およびソケットはどちらも一連の状態を使用して、適切な呼び出しの順序を制御しますが、各々が使用する状態は異なります。XTI の状態とソケットの状態が持つ意味は同じではありません。たとえば、XTI の状態は相互に排他的ですが、ソケットの状態は排他的ではありません。

ほとんどのエラー・メッセージは、ソケットと XTI で異なります。表 3-15 に、XTI エラー・メッセージに対応するソケット・エラー・メッセージの一覧を示します。

表 3-15: ソケットと XTI のメッセージの対応

ソケット・エラー	XTI エラー	説明
EBADF	TBADF	無効なファイル記述子を指定した。
EOPNOTSUPP	TNOTSUPPORT	基礎となるトランスポート・プロバイダがサポートしていない関数を発行した。
EADDRINUSE	TADDRBUSY	すでに使用されているアドレスを指定した。
EACCES	TACCES	指定したアドレスを使用する許可がない。

注意

XTI および TLI は、STREAMS を使用してインプリメントされます。STREAMS ファイル記述子に関しては、`select` 呼び出しではなく、`poll` 関数を使用してください。

3.6 XPG3 , XNS4.0 , XNS5.0 の相違点

この節では、XPG3 , XNS4.0 , XNS5.0 の XTI のインプリメントの相違点について説明します。

これまでのバージョンの Tru64 UNIX では、XTI のインプリメントは X/Open の XPG3 仕様に準拠していました。現在のインプリメントは、X/Open の XNS4.0 および XNS5.0 の XTI 仕様に準拠しています。XNS4.0 が省略時の設定です。

これらの仕様には、プログラマが知っておかなければならない変更がいくつかあります。この節では、そのような相違点と、関連するプログラミング上の問題について概説します。

Tru64 UNIX は、XPG3 , XNS4.0 , および XNS5.0 の XTI を 1 つのサブセットに集約してインプリメントしていることに注意してください。この節では、該当するレベルの機能の使用についての詳細も記述します。

本書では、XNS4.0 または XNS4.0 XTI という用語は、Tru64 UNIX の本バージョンでの XTI の省略時のインプリメントを指して使用します。XNS5.0 または XNS5.0 XTI という用語は、X/Open の XNS5.0 仕様に準拠する XTI のインプリメントを指して使用します。XPG3 の XTI という用語は、X/Open の XPG3 仕様に準拠する XTI のインプリメントを指して使用します。XPG3 の XTI は、Tru64 UNIX の現在のバージョンでも、バイナリの互換性やソース移行の機能のために使用できます。

3.6.1 XPG3 と XNS4.0 の主な相違点

2 つの仕様の間の変更点のほとんどには、上位互換性があります。しかし `t_optmgmt` 関数はこれに当てはまりません。

次に、XTI の XPG3 から XNS4.0 への変更についての基本事項を簡単にまとめます。

- オプションだった関数が必須になった。Tru64 UNIX は XPG3 の XTI でオプションの関数をすべてインプリメントしているので、Tru64 UNIX の XTI のインプリメントでは何の影響もない。
- XPG3 仕様の多数の点が明瞭になり、XTI のアプリケーションの移植が容易になった。
- 新しくエラー・コードがいくつか追加され、プログラムの性能が向上した。
- オプションおよびその管理構造が見直され、通信の各種の面に対する制御性が高まった。

`t_optmngmt` 関数の変更は非常に大きく、XPG3 仕様とは互換性がありません。一般に、XPG3 のインプリメントの `t_optmngmt` 関数を使用しているアプリケーションは、XNS4.0 または XNS5.0 仕様が行われているシステムの `t_optmngmt` 関数を使用できません。ソースにいくつかの修正が必要です。

3.6.2 XNS4.0 と XNS5.0 の主な相違点

2 つの仕様間の変更点のほとんどには、上位互換性があります。XNS4.0 の XTI に対する XNS5.0 の XTI の基本的な変更点の概要を、次に示します。

- 7 つの新しい関数 (`t_rcvreldata`, `t_rcvv`, `t_rcvvudata`, `t_sndreldata`, `t_sndv`, `t_sndvudata`, `t_sysconf`) が追加された。

3.6.3 ソース・コードの移行

XPG3 の XTI 用に開発したアプリケーションがある場合、オペレーティング・システムの現在のバージョンでそのアプリケーションをサポートするときには、次のいずれかを選択します。

- アプリケーションの古いバイナリ・ファイルを使用する。 3.6.4 項を参照。
- ソースを変更しないで再コンパイルする。
- 必要であれば、ソースを変更して XNS4.0 の XTI に準拠させる。
- 必要であれば、ソースを変更して XNS5.0 の XTI に準拠させる。

どの方法を選択するかは、ユーザの状況によって異なります。次の各節では、そのような状況について詳細に説明します。

3.6.3.1 アプリケーションの古いバイナリ・ファイルを使用する場合

アプリケーションのソースと機能を変化を変更しない場合は、この方法を選択します。以前のリリースのまま製品を継続して機能させることができる点が便利です。

3.6.3.2 ソースを変更しないで再コンパイルする場合

小さな問題点を修正するためにわずかな変更を行う場合は、この方法を選択します。したがって、アプリケーションの構成や機能は変更しません。XPG3の開発環境と同じ方法でソースをコンパイルしたい場合は、`-DXPG3` コンパイラ・スイッチでコンパイルします。そうすれば、ヘッダが自動的に以前の機能を定義します。

3.6.3.3 XNS4.0 に準拠させる場合

XNS4.0のXTIがサポートする新しい機能を使用する必要がある場合は、ソース・コードに変更を加えなければなりません。XPG3とXNS4.0のXTIの機能を組み合わせることはできません。したがって、複数のファイルからなる大きいアプリケーションの場合は、変更を行った一部のファイルだけではなく、すべてのファイルを新しい機能とともに再コンパイルする必要があります。

ソース・コードをコンパイルするには、`-DXOPEN_SOURCE` コンパイラ・スイッチを付ける必要があります。さらに、トランスポート・プロトコルの名前 (`/dev/streams/xtiso/tcp` といったストリームのデバイス特殊ファイルとして与えられるもの) は、XNS4.0のXTIで使用されている命名規則に合わせて更新しなければなりません。たとえば、TCPとUDPの名前は、`/dev/streams/xtiso/tcp+` と `/dev/streams/xtiso/udp+` になります。その他のプロトコルの名前については、それぞれのリファレンス・マニュアルで確認してください。

3.6.3.4 XNS5.0 に準拠させる場合

XNS5.0のXTIがサポートする新しい機能を使用する必要がある場合は、ソース・コードに変更を加えなければなりません。XPG3とXNS5.0のXTIの機能を組み合わせることはできません。したがって、複数のファイルからなる大きいアプリケーションの場合は、変更を行った一部のファイルだけではなく、すべてのファイルを新しい機能とともに再コンパイルする必要があります。

ソース・コードをコンパイルするには、`-DXOPEN_SOURCE=500` コンパイラ・スイッチを付ける必要があります。さらに、トランスポート・プロトコルの名前 (`/dev/streams/xtiso/tcp` といったストリームのデバイス特殊ファイルとして与えられるもの) は、XNS5.0 の XTI で使用されている命名規則に合わせて更新しなければなりません。たとえば、TCP と UDP の名前は、`/dev/streams/xtiso/tcp5` と `/dev/streams/xtiso/udp5` になります。その他のプロトコルの名前については、それぞれのリファレンス・マニュアルで確認してください。

3.6.4 バイナリの互換性

XPG3 の XTI を使用して開発されたアプリケーションのバイナリ・ファイルを実行する際には、いくつかの注意が必要です。

正常でない状況では、XPG3 を使用するプログラムの中のエラーが、プログラムまたはライブラリをコンパイルおよびリンクする方法によってマスクされてしまうことがあります。こうした場合に、新しいインプリメントによってエラーのフラグがセットされることがあります。明らかになるエラーはアプリケーションのプログラム上のエラーなので、プログラマが修正します。通常、このようなエラーが発生するのは、プログラム上のポインタのオーバーランや変数の初期化忘れです。

もう 1 つ考慮が必要な点は、STREAMS 特殊ファイルを介して XNS4.0 の機能が使用できることです。これが重要になるのは、アプリケーションがコマンド行からのトランスポート・プロトコルの入力や、何らかの構成ファイルからのプロトコル名の取り込みを受け付ける場合です。XTI を組み込んで構成したシステムは、XNS4.0 に準拠したプロトコルのためのファイル名を持ちます。したがって、バイナリの互換モードで実行しているアプリケーションでは、こうした特別な名前を使用しないようにユーザやシステム管理者に警告しておくことが大切です。使用した場合の結果は、定義されていません。

古いアプリケーションを再コンパイルせずに実行する場合は、バイナリの互換性をチェックして、こうした問題が起こらないようにしてください。

3.6.5 パッケージ

オペレーティング・システムの現在のバージョンが実行されていて、XTI を組み込んで構成されているシステムは、XPG3 準拠、XNS4.0 準拠、および XNS5.0 準拠の機能をサポートしています。XPG3、XNS4.0、および

XNS5.0 の機能を分けて実行することはできません。したがって、XTI のサブシステムが構成に組み込まれていることだけを確認する必要があります。

3.6.6 相互運用性

XPG3, XNS4.0, および XNS5.0 の XTI を 1 つのネットワーク上で使用できます。アプリケーションの、互換性のあるバージョンを使用する場合は、動作の違いはユーザからはわかりません。

アプリケーションを簡単な手順で変換して、ある部分は XPG3 の XTI に互換で、ある部分は XNS4.0 の XTI に互換で、ある部分は XNS5.0 の XTI に互換にすることができます。必ず守らなければならないのは、アプリケーション・レベルのプロトコルを同じに保つことです。その他には、それぞれの構成要素の相互運用性について問題はありません。したがって、アプリケーションにクライアントとサーバがあれば、サーバを XNS4.0 または XNS5.0 準拠にアップグレードして、クライアントはそのままバイナリ互換のモードで動作させておくことができます。後で、サーバの機能のアップグレードが完了したら、クライアントをアップグレードすることもできます。

3.6.7 XTI のオプションの使用

この節では、XNS4.0, XNS5.0, および XPG3 の XTI のオプションの使用について説明します。

3.6.7.1 XNS4.0 および XNS5.0 の XTI のオプションの使用

この節では、XTI のオプションを使用する上での、次の事柄について説明します。

- オプションの使用についての全般的な説明
- オプションの形式
- 折衝の各要素
- トランスポート終端のオプション管理

全般的な説明

次の各関数は、入出力パラメータとして `struct netbuf` 型の引数 `opt` を持っています。この引数は、トランスポート・ユーザとトランスポート・プロバイダの間でオプションをやりとりするために使用します。

- `t_accept`
- `t_connect`
- `t_listen`
- `t_optmgmt`
- `t_rcvconnect`
- `t_rcvudata`
- `t_rcvuderr`
- `t_sndudata`

オプションの内容についての一般的な規定はありません。オプションには、XTI 全般で使用するものと、それぞれのトランスポート・プロバイダに固有のものがあります。オプションのいくつかは、ユーザの通信上の必要に応えるためのものです。たとえば、高いスループットや短い遅延を要求するようなものです。その他のオプションには、プロトコルの動作を微調整して、特殊な特性の通信を、より効率的に処理できるようにするためのものもあります。また、デバッグを目的としたオプションもあります。

すべてのオプションには、省略時の値があります。値は、それぞれが適用されるプロトコル・レベルで意味を持ち、定義されています。しかし、トランスポート・ユーザが値を折衝できます。これには、トランスポート・ユーザが値の使用を強制できる場合などの単純なケースも含まれます。多くの場合は、トランスポート・プロバイダ、またはリモートのトランスポート・ユーザまでもが、値を折衝して、提案されたものよりも低い品質の値にする権限を持っています。つまり、遅延が長くなったり、スループットが低くなったりすることがあります。

オプションを、関連付けに関するものとそうでないものとに区別すると役に立ちます。関連付けに関するとは、通信するトランスポート・ユーザのペアに関することを意味します。関連付けに関するオプションは、特定のトランスポート接続やデータグラム転送に密接に関係しています。呼び出し側ユーザがこのようなオプションを指定すると、ほとんどの場合はネットワークを介して何らかの補助的な情報がやりとりされます。この情報の解釈およびその後の処理は、プロトコルによって変わります。たとえば、ISO のコネクション指向型の通信では、呼び出し側ユーザは、接続の確立でサービス品質パラメータを指定できます。このパラメータは、最初にローカルのトランスポート・プロバイダで処理されて、おそらくは低下され、そしてリ

モートのトランスポート・プロバイダに送られ、さらに低下されます。最後に、呼び出された側のユーザに渡されて最終の選択が行われ、選択された値が呼び出し側に返されます。

関連付けに関したものでないオプションは、リモートのトランスポート・ユーザに送る情報を持ちません。デバッグを行うためのオプションのように、まったくローカルでだけ意味を持つものがあれば、IP の *time-to-live* フィールドや、TCP_NODELAY を設定するオプションのように、転送に影響を与えるものもあります (xti_internet(7) を参照)。ローカルのオプションの折衝は、トランスポート・ユーザとトランスポート・プロバイダの間でだけ行われます。こうした関連付けについての 2 つの分類のオプションの区別は、XTI の次の各点において見られます。出力では、t_listen および t_rcvudata 関数は関連付けに関するオプションだけを返します。t_rcvconnect および t_rcvuderr 関数はどちらの分類のオプションも返します。入力では、t_accept および t_sndudata 関数はどちらの分類のオプションも指定できます。t_connect および t_optmgmt 関数はどちらの分類のオプションも処理して返します。

トランスポート・プロバイダは、サポートするそれぞれのオプションに対して、省略時の値を持っています。大部分の通信は省略時の値で十分です。したがって、トランスポート・ユーザは、作業を行うのに実際に必要なオプションだけを要求して、その他はすべて省略時の値にするようにしてください。

この項では、オプションの使用についての全般的なフレームワークを説明しています。このフレームワークは、トランスポート・プロバイダに必須なものです。XTI の一般的なオプションについては、t_optmgmt リファレンス・ページで説明しています。TCP および UDP のトランスポート・プロバイダで有効な固有のオプションについては、xti_internet リファレンス・ページで説明しています。

オプションの形式

オプションは、struct netbuf の引数 opt で受け渡します。指定されたバッファ内のそれぞれのオプションは、struct t_opthdr 型の形式をしていて、通常はオプションの値が後に続きます。

トランスポート・プロバイダはプロトコル・スタックを形成しています。struct t_opthdr の level フィールドは、XTI レベル、または TCP や ISO 8073:1986 といったトランスポート・プロバイダのプロトコルを指定

します。 *name* フィールドは、そのレベル内のオプションを指定し、*len* フィールドは全体の長さ、つまりオプションのヘッダ `t_opthdr` にオプションの値部分を合わせた長さを収めています。 *status* フィールドは、XTI レベルまたはトランスポート・プロバイダが使用して、折衝の成功あるいは失敗を示します。

複数のオプションを連結することもできます。ただし、それぞれのオプションは必ずロング・ワード単位の境界で始まらなければなりません。XNS4.0 では `OPT_NEXTHDR(pbuf, buflen, poptions)` マクロ、XNS5.0 では `T_OPT_NEXTHDR(pbuf, buflen, poptions)` マクロを使用すると、ロング・ワード単位の境界で始めることができます。パラメータ *pbuf* はオプション・バッファ *opt.buf* へのポインタを示し、*buflen* はバッファの長さを示します。パラメータ *poption* は、オプション・バッファ内の現在のオプションを指します。 `OPT_NEXTHDR` および `T_OPT_NEXTHDR` は、次のオプションの位置へのポインタを返すか、またはオプション・バッファを使い切った場合に `NULL` ポインタを返します。これらのマクロは、オプションのリストを読み書きするときに便利です。

折衝の各要素

この項では、オプションの受け渡しと取り出しに対する一般的な規則、および起こり得るエラーについて説明します。明示的な制約がない限り、これらの規則はオプションの交換が可能なすべての関数に適用されます。

複数のオプションの指定とオプション・レベル

入力のオプション・バッファで複数のオプションを指定した場合は、呼び出す関数によって、指定するレベルに対して違った規則が適用されます。 `t_optmgmt` に入力で指定する複数のオプションは、同じオプション・レベルに向けたものでなければなりません。 `t_connect`、`t_accept`、および `t_sndudata` に入力で指定するオプションは、違ったレベルに向けたものでも構いません。

違反オプション

折衝できるのは、規則に従っているオプションだけです。違反オプションは障害を起こす可能性があります。オプションが違反になるのは、次のいずれかの場合です。

- `t_opthdr.len` パラメータに指定した長さが、(オプションの先頭から数えて) オプション・バッファの末尾までのサイズを超えている。
- オプションの値が違反している。それぞれのオプションには、規則に従って値が定義されている。詳細は `t_optmgmt(3)` および `xti_internet(7)` を参照。

違反オプションを XTI に渡した場合は、次のようになります。

- `t_optmgmt` 関数の呼び出しは、TBADOPT エラーで失敗する。
- `t_accept` または `t_connect` 関数は、TBADOPT エラーで失敗するか、または接続の確立が打ち切られる。どちらになるかは、インプリメント、および違反オプションが検出されるタイミングにより異なる。

接続が打ち切られる場合は、`T_DISCONNECT` イベントが発生し、`t_connect` の同期呼び出しは `TLOOK` エラーで失敗します。`t_accept` 関数の場合は、接続が打ち切られると、それでも正常に終了するか、または `TLOOK` エラーで失敗します。どちらになるかは、インプリメントおよびタイミングにより異なります。

- `t_sndudata` 関数の呼び出しは、TBADOPT エラーで失敗するか、または正常に戻る。

ただし、`T_UDERR` イベントが発生して、データグラムが送信されなかったことを示します。

トランスポート・ユーザが 1 つの呼び出しで複数のオプションを渡した場合に、オプションの 1 つが違反していると、呼び出しは前述のように失敗します。ただし、渡した違反していないオプションの一部、または場合によってはすべてが正常に折衝されることもあります。トランスポート・ユーザは、`t_optmgmt` 関数を `T_CURRENT` フラグをセットして呼び出すことで、現在の状態を確認できます。`t_optmgmt(3)` および `xti_internet(7)` を参照してください。

オプション・レベルが選択したプロトコルに対して未知、またはそのプロトコルがサポートしていないオプション・レベルを指定した場合は、失敗になりません。このオプションは、`t_connect`、`t_accept`、または `t_sndudata` 関数の呼び出しでは破棄されます。`t_optmgmt` 関数では、オプションの `level` フィールドで `T_NOTSUPPORT` を返します。

オプション折衝の開始

T_NEGOTIATE フラグ付きの `t_optmgt`, `t_connect`, または `t_sndudata` 関数を呼び出すとき, トランスポート・ユーザがオプション折衝を開始します。

これらの関数の折衝についての規則は, オプション要求が絶対必要条件かどうかによって変わります。絶対必要条件か否は, それぞれのオプションで明示的に定義されています。 `t_optmgt(3)` および `xti_internet(7)` を参照してください。たとえば, ISO トランスポート・プロバイダの場合は, 優先データの使用を要求するオプションは絶対必要条件ではありません。一方, 保護を要求するオプションは, 絶対必要条件である場合があります。

注意

絶対必要条件という用語は, 本来は ISO 8072:1986 仕様のサービス品質パラメータについてのものです。ここでは, すべてのオプションに対して使用しています。

提案したオプション値が絶対必要条件の場合は, 次の3つの結果が考えられます。

- 折衝された値が提案した値と同一になる。折衝の結果を取り出すと, `t_opthdr` の `status` フィールドが T_SUCCESS に設定されている。
- オプションがサポートされていても提案した値が折衝できない場合は, 折衝が拒否される。この結果, 次のようになる。
 - `t_optmgt` 関数は, 正常に戻る。しかし, 返される値の `status` フィールドは T_FAILURE に設定されている。
 - 接続を確立する試みは, すべて打ち切られる。T_DISCONNECT イベントが発生し, `t_connect` 関数の同期呼び出しは TLOOK エラーで失敗する。
 - `t_sndudata` 関数は, TLOOK エラーで失敗するか, または正常に戻る。ただし, T_UDERR イベントが発生して, データグラムが送信されなかったことを示す。

1つの呼び出しで複数のオプションを送り, その内の1つがリジェクトされた場合は, XTI は上述のように動作します。接続の確立またはデータグラムの転送は失敗しますが, いずれかのオプションがリジェクトさ

れる前に正常に折衝されたオプションは、折衝された値を保持します。ロールバックの機構はありません。詳細は、「トランスポート終端のオプション管理」の項を参照してください。

`t_optmgmt` 関数は、それぞれのオプションを折衝しようと試みます。返されるオプションの `status` フィールドは、成功 (`T_SUCCESS`) または失敗 (`T_FAILURE`) を示します。

- ローカルのトランスポート・プロバイダが、そのオプションをサポートしていない場合は、`t_optmgmt` 関数は `status` フィールドで `T_NOTSUPPORT` を報告します。`t_connect` および `t_sndudata` 関数は、このオプションを無視します。

提案したオプション値が絶対必要条件でない場合は、次の結果が考えられます。

- 折衝された値が、提案した値以下の質になる (たとえば、遅延が長くなるなど)。

折衝の結果を取り出すと、折衝された値が提案した値に等しい場合は、`t_opthdr` の `status` フィールドが `T_SUCCESS` に設定されます。等しくない場合は、`T_PARTSUCCESS` に設定されます。

- ローカルのトランスポート・プロバイダが、そのオプションをサポートしていない場合は、`t_optmgmt` は `status` フィールドで `T_NOTSUPPORT` を報告する。`t_connect` および `t_sndudata` 関数はこのオプションを無視します。

オプションがサポートされていないために、関数が失敗したり、接続が打ち切られることはありません。これは、ベンダによって、違ったサブセットのオプションをインプリメントしているためです。さらに、XTI の将来の拡張で、トランスポート・プロバイダのこれまでのインプリメントでは未知のオプションも含められる可能性があります。オプションがサポートされていない場合の通信を受け入れるかどうかの判断は、トランスポート・ユーザに任されています。

トランスポート・プロバイダは、同じオプションの (おそらくは違った値を持つ) 重複した発生をチェックしません。単にオプションをオプション・バッファから順番に処理します。ただし、ユーザは処理の順番について何も知ることができません。

オプションは、互いに独立していないものもあります。要求したオプション値が、同じ呼び出しで指定している別のオプション値や、現在有効な別のオプション値と矛盾することがあります。詳細は、「トランスポート終端のオプション管理」の項を参照してください。こうした矛盾はすぐには検出されなくても、後になって予測しない結果につながる場合があります。折衝時に検出された場合は、これらの矛盾は前述の規則で解決されます。したがって、結果は、矛盾にかかわる要求が絶対的な要求かどうかによって大きく変わります。

通常接続が確立されるか、またはデータグラムが送信されるときに、矛盾は検出されます。オプションが `t_optmgmt` 関数で折衝される場合は通常、矛盾はこのときには検出されません。これは、要求されたオプションを独立して処理するので、一時的な不一致が許されるためです。

`t_connect` および `t_sndudata` 関数を呼び出した場合は、関連付けに関するすべてのオプションに対して、この項の規則に従って折衝が開始されます。関数呼び出し自体では明示的に指定していないオプションは、以前の折衝の値を収めた内部のオプション・バッファから取り出されます。詳細は、「トランスポート終端のオプション管理」の項を参照してください。

折衝の提案への応答

コネクション指向型の通信では、対等トランスポート・ユーザが、確立するトランスポート接続の特性を折衝できます。これは、関連付けに関するオプションの場合です。接続指示に対して、呼び出された側のユーザは (`t_listen` 関数を介して)、その接続で有効にするオプションの値についての提案を受け取ります。呼び出された側のユーザは、この提案を受け入れるか、またはこれよりも低い質の値を選んで低くする (たとえば、提案よりも遅延を長くする) ことができます。当然、呼び出された側のユーザが接続の確立を拒否することもできます。

呼び出された側のユーザは、`t_accept` 関数を使用して折衝の提案に応答します。呼び出された側のトランスポート・ユーザが、提案よりも高い質のオプションを折衝しようとした場合の結果は、オプションが適用されるプロトコルによって変わります。プロトコルはオプションを拒否したり、プロトコル固有のリファレンス・ページに記述されているような他の適当な動作を行ったりします。オプションがリジェクトされた場合は、接続が失敗して `T_DISCONNECT` イベントが発生します。この場合、`t_accept` 関数がそれ

でも正常終了するか、または TLOOK エラーで失敗するかは、タイミングとインプリメントの条件によって決まります。

複数のオプションを `t_accept` 関数に渡して、その内の 1 つがリジェクトされた場合は、接続は前述のようにして失敗します。エラーになるオプションの処理の前に正常に折衝されたオプションは、折衝された値を保ちます。ロールバックの機構はありません。詳細は、「トランスポート終端のオプション管理」の項を参照してください。

応答するオプションは、`t_accept` 呼び出しで指定するか、または `t_accept` 呼び出しの前に `t_optmgmt` 呼び出し (T_NEGOTIATE アクション) の中の応答する終端 (リッスンする終端ではありません) `resfd` に渡すかのどちらかができます。詳細は、「トランスポート終端のオプション管理」の項を参照してください。なお、折衝の提案への応答が起動されるのは、`t_accept` 関数が呼び出されたときです。`t_optmgmt` 関数の呼び出しが前述のようなエラーになるオプション値を持つ場合は、正常終了します。接続が打ち切られるのは、`t_accept` 関数が呼び出されたときです。

選択されたオプションが矛盾につながる場合も、接続は失敗します。

`t_accept` 関数は、オプションの指定の重複をチェックしません。「オプション折衝の開始」の項を参照してください。サポートしていないオプションは無視されます。

オプション情報の取り出し

この項では、トランスポート・ユーザがオプションに関する情報を取り出す方法について説明します。

トランスポート・ユーザは、次のことができればなりません。

- 折衝の結果を知る。たとえば、接続確立の後の時点などで。
- 接続確立の中で折衝に提案されたオプションの値を知る。
- リモートのトランスポート・ユーザが、通知のためだけに送ってきたオプションの値を取り出す。たとえば、IP オプション。
- トランスポート終端の、現在有効なオプションの値を確認する。

このために、次の各関数は `struct netbuf` 型の引数 `opt` を持っています。

- `t_connect`

- `t_listen`
- `t_optmgmt`
- `t_rcvconnect`
- `t_rcvudata`
- `t_rcvuderr`

トランスポート・ユーザは、オプションを書き込むためのバッファを用意しなければなりません。 `opt.buf` パラメータでこのバッファを指し示し、 `opt.maxlen` パラメータにバッファのサイズを収めます。 トランスポート・ユーザは、 `opt.maxlen` パラメータをゼロに設定して、取り出すオプションがないことを示すことができます。

どのオプションが返されるかは、次のように各関数によって違います。

- 同期モードの `t_connect` および `t_rcvconnect`

関数が返すのは、接続の応答で受け取った、関連付けに関するオプションの値、および入力で指定した、関連付けに関しないオプションの折衝された値です。ただし、 `t_connect` 呼び出しの入力で指定したオプションがサポートされていなかったり、未知のオプション・レベルを参照していたりする場合は、このオプションは破棄されて出力に返されません。

`t_connect` および `t_rcvconnect` 関数のそれぞれのオプションの `status` フィールドは、折衝された値が、提案した値 (T_SUCCESS) か、低下された値 (T_PARTSUCCESS) かを示します。折衝の対象にならない、受け取った補助的な情報 (たとえば IP オプション) の `status` フィールドは、常に T_SUCCESS に設定されます。

- `t_listen`

関連付けに関する受け取るオプションは、着信接続 (シーケンス番号で識別されます) についてのもので、リッスンしている終端についてのものではありません。ただし、リッスンしている終端で現在有効なオプション値が、 `t_listen` 関数で取り出す値に影響することはあります。これは、トランスポート・プロバイダも折衝処理に関わることがあるためです。したがって、T_CURRENT アクションの `t_optmgmt` 関数の呼び出しで同じオプションを指定すると、通常同じ値が返されます。

受け取るオプションの数は、後に続く接続指示の数によって変わります。関連付けに関するオプションの多くは、(たとえば、IP オプショ

ンや、ISO 8072:1986 のスループットのように) 呼び出し側ユーザの明示的な要求によってだけ転送されるためです。オプションがまったく返されない場合もあります。

status フィールドは無関係です。

- `t_rcvudata`

関連付けに関する受け取るオプションは、着信データグラムについてのもので、トランスポート終端 *fd* についてものではありません。したがって、`T_CURRENT` アクションの `t_optmgmt` 関数の呼び出しで同じオプションを指定すると、通常同じ値は返されません。

受け取るオプションの数は、呼び出しによって変わります。

status フィールドは無関係です。

- `t_rcvuderr`

返されるオプションは、前の、エラーを発生した `t_sndudata` 呼び出しの入力オプションについてのもので、どのオプションが返されてどのような値を持っているかは、それぞれのエラー状況によって変わります。 *status* フィールドは無関係です。

- `t_optmgmt`

この呼び出しは、関連付けについての両方の分類のオプションを処理して返すことができます。扱うオプションは、指定したトランスポート終端についてのもので、接続指示や着信データグラムについてものではありません。詳細は、`t_optmagmt(3)` を参照してください。

特権オプションおよび読み取り専用オプション

特権オプションとその値は、特権ユーザだけが要求できます。ここでの特権の意味はインプリメントで定義されます。

読み取り専用オプションは、参照のためだけに使用します。トランスポート・ユーザはオプションの値を読み出すことはできますが、変更はできません。たとえばプロトコル・タイマや、プロトコル・データ・ユニットの最大長の選択は非常に微妙なので、トランスポート・ユーザには任せられないことがあります、その値を知る必要はあります。オプションは、すべてのユーザまたは特権のないユーザに対して読み出し専用にできます。特権オプションは、特権のないユーザに対してアクセス不可または読み出し専用にできます。

オプションは、XTI の状態によって折衝可能であったり、読み出し専用であったりすることがあります。たとえば、ISO のサービス品質オプションは、T_IDLE および T_INCON 状態では折衝可能で、その他のすべての状態 (T_UNINIT を除きます) では読み出し専用です。

トランスポート・ユーザが読み出し専用オプションの折衝を要求するか、または特権のないユーザが特権オプションに不法にアクセスした場合は、次のような結果になります。

- `t_optmgmt` 関数は正常に戻るが、返されたオプションの *status* フィールドは、特権オプションを不法にアクセスした場合は T_NOTSULPORT に、読み出し専用オプションの変更を要求した場合は T_READONLY に設定される。
- 読み出し専用オプションの折衝を要求すると、`t_accept` または `t_connect` 関数は TACCES で失敗するか、または接続の確立が打ち切られて T_DISCONNECT イベントが発生する。接続が打ち切られた場合は、`t_connect` の同期呼び出しは TLOOK で失敗する。特権オプションを不法に要求した場合は、オプションが単に無視される。特権のないユーザは、特権オプションやサポートされていないオプションは選択できない。`t_accept` 呼び出しが正常に終了するか、または TLOOK で失敗するかは、タイミングとインプリメントの条件で決まる。
- 読み出し専用オプションの折衝を要求すると、`t_sndudata` 関数は TLOOK を返すか、または正常に返す。しかし、T_UDERR イベントが発生して、データグラムが送信されなかったことを示す。特権オプションを不法に要求した場合は、単に無視される。特権のないユーザは、特権オプションやサポートされていないオプションは選択できない。

複数のオプションを `t_connect`、`t_accept`、または `t_sndudata` 関数に渡して、読み出し専用オプションが拒否された場合は、接続またはデータグラム転送は前述のようにして失敗します。エラーになるオプションが処理される前に正常に折衝されたオプションは、折衝された値を保ちます。ロールバックの機構はありません。詳細は、「トランスポート終端のオプション管理」の項を参照してください。

トランスポート終端のオプション管理

この項では、トランスポート終端の存在期間中にオプション管理がどのように行われるかについて説明します。

それぞれのトランスポート終端は、内部のオプション・バッファと（論理的に）結び付いています。トランスポート終端が作成されると、このバッファに、サポートされているそれぞれのオプションの省略時のシステム設定値が入ります。オプションによって、省略時の設定値は、OPTION ENABLED、OPTION DISABLED、またはタイム・スパンを示すものやその他の値になります。たいていの操作では、省略時のこれらの設定は適切です。オプションの折衝を通してオプションの値が変更されると、必ず修正後の値が以前の値を上書きしてこのバッファに書き込まれます。バッファは、常にすべてのオプションに対して、そのトランスポート終端で現在有効な値を収めています。

`t_optmgt` 関数を `T_CURRENT` フラグをセットして呼び出すことにより、オプションの現在の値はいつでも取り出せます。`t_optmgt` 関数を `T_DEFAULT` フラグをセットして呼び出すと、指定したオプションの省略時のシステム設定が返されます。

`t_optmgt` 関数を `T_NEGOTIATE` フラグをセットして呼び出すことで、トランスポート・ユーザはオプションの新しい値を折衝できます。折衝は、「折衝の各要素」の項で説明した規則に従います。

オプションによっては、特定の XTI 状態でだけ変更できたり、他の XTI 状態で読み出し専用になったりするものがあります。たとえば、関連付けに関するオプションの多くは、`T_DATAXFER` 状態では変更できず、変更しようとすると失敗します。「特権オプションおよび読み出し専用オプション」の項を参照してください。それぞれのオプションの状態についての規則は、その定義の中で指定されています。

関連付けに関するオプションは通常、接続の確立時またはデータグラムの転送の時に有効になります。これはオプションが、ネットワークを介して転送される情報を含んでいる場合や、特定の転送特性を決める場合です。このようなオプションを `t_optmgt` 関数の呼び出しで変更する場合には、トランスポート・プロバイダは、オプションをサポートしているかどうかを確認して、値を現在の情報に従って折衝します。折衝した値は、内部のオプション・バッファに書き込みます。

最終的な折衝は、接続が確立されるか、またはデータグラムが転送されると起こります。この結果、オプションの値が低下されたり、または折衝が失敗したりすることさえあります。折衝された値は、内部のオプション・バッファに書き込まれます。

T_DATAXFER 状態でも変更できるオプションがあります。たとえば、バッファ・サイズを指定するオプションです。このような変更は転送の特性に影響し、予測できない副作用を起こすことがあります。たとえば、バッファ・サイズを短くするとデータを失う可能性があります。

`t_connect`、`t_accept`、または `t_sndudata` 関数を呼び出す場合は、トランスポート・ユーザは関連付けについての両方の分類のオプションを、入力で明示的に指定できます。オプションは、最初にローカルで1つ1つ折衝され、結果の値が内部のオプション・バッファに書き込まれます。この後、ネットワークを介した折衝段階がさらに必要な場合は、変更されたオプション・バッファが使用されます。たとえば、コネクション指向型の ISO の通信がそうです。その後、新たに折衝された値が、内部のオプション・バッファに書き込まれます。

どの段階でも、折衝が失敗すると転送が打ち切られます。転送が打ち切られると、オプション・バッファは、失敗が起こった時点の内容を保ちます。XTI 呼び出しが失敗しても成功しても、エラーが発生する前に折衝されたオプションはオプション・バッファに書き込まれます。

`t_connect`、`t_accept`、または `t_sndudata` 関数を呼び出すときに、どのオプションを明示的に指定するかを決めるのは、トランスポート・ユーザです。トランスポート・ユーザは、関数の入力引数 `opt` の `len` フィールドをゼロ (0) に設定すれば、オプションをまったく渡さなくてもかまいません。この場合、内部のオプション・バッファの現在の内容が、変更なしに折衝に使用されます。

`t_connect`、`t_accept`、または `t_sndudata` 呼び出しのときのオプションの折衝プロシージャは、常に「オプション折衝の開始」の項の規則に従います。オプションが呼び出しで明示的に指定されたか、内部のオプション・バッファから暗黙に取られたかには関係ありません。

折衝でオプションが処理される順番については、トランスポート・ユーザは何も知ることができません。

オプション・バッファの値が変更されるのは、オプションが正常に折衝された結果による場合だけです。特に、接続の解放では変更されません。ヒストリの機構はなく、データグラム転送や接続の確立以前に存在していたバッファの状態はリストアできません。トランスポート・ユーザは、(オプションが、最初に省略時の値に初期化されていた場合でも) 接続の確立またはデー

タグラム転送が内部のオプション・バッファを変更することがあることを知っておかなければなりません。

オプション値 **T_UNSPEC**

オプションの中には、常に完全に決まった値を持っているとは限らないものがあります。たとえば、複数のプロトコル・クラスをサポートする ISO トランスポート・プロバイダは、接続の確立を開始するまでは、優先クラスを選択していない場合があります。接続要求の時点で、トランスポート・プロバイダが、宛先のアドレス、サービス品質パラメータ、およびその他のローカルの情報からどの優先クラスを使用するかを決めます。トランスポート・ユーザが、T_IDLE 状態で優先クラスのオプションの省略時の値を要求すると、T_UNSPEC の値を得ます。この値は、トランスポート・プロバイダがまだ値を選択していないことを示します。トランスポート・ユーザは、T_CLASS2 といった別の値を優先クラスの値として折衝することはできません。この場合は、トランスポート・プロバイダは、クラス 2 を優先クラスとして接続要求を開始するよう強制されます。

XTI のインプリメントによっては、オプションの値に現在アクセスできないときに T_UNSPEC 値を返すこともあります。これは、プロトコル・スタックがホストではなく別のコントローラ・カード上に位置するシステムで、T_UNBND 状態において起こることがあります。オプションがサポートされていない場合に、このようなインプリメントが T_UNSPEC を返すことはありません。

T_UNSPEC は、これが有効な値であるオプションでは、入力でも使用できます。適切な値の選択をプロバイダに任せることを指定するために使用します。ISO のスループットのように、オプション値が内部で構造体を持つような複雑なオプションで特に役に立ちます。この値を選択することで、トランスポート・ユーザはフィールドのいくつかを指定しないままにしておくことができます。ユーザが T_UNSPEC を指定すると、トランスポート・プロバイダが適切な値を自由に選択します。この結果は、省略時の値になることも、明示的にその他の値になることも、または T_UNSPEC になることもあります。

それぞれのオプションについて、折衝に対して T_UNSPEC が有効な値かどうか指定されています。

引数 **info**

`t_open` および `t_getinfo` 関数は、引数 `info` でトランスポート・プロバイダの特性を表す値を返します。`info->options` の値は、`t_alloc` 関数を使用します。`t_alloc` 関数は、XTI 呼び出しで使用するオプション・バッファの記憶割当てにこの値を使用します。この値は、すべての用途に十分な大きさです。

一般に、`info->options` には、特権オプションのサイズも含まれています。特権オプションが、特権のないユーザに対して読み出し専用でない場合でも同じです。あるいはインプリメントによっては、`info->options` で、特権ユーザと非特権ユーザに関して別の値を返すようにすることもあります。

`info->etsdu`、`info->connect`、および `info->discon` の値は、`T_DATAXFER` 状態に入ると通常小さくなります。`t_optmgt` 関数を呼び出してもこれらの値には影響ありません。詳細は、`t_optmgt(3)` を参照してください。

移植性の問題

XTI プログラムを作成するアプリケーション・プログラマは、次の各点にわたる移植性に関する問題に注意してください。

- 各プロトコル・プロファイル
- それぞれのシステムのプラットフォーム

オプションは、本質的に特定のプロトコルまたはプロトコル・プロファイルと結び付いています。したがって、オプションを明示的に使用すると、複数のプロトコル・プロファイルにわたる移植性が低下します。

異なるベンダは、異なるオプションをサポートするトランスポート・プロバイダを提供します。これは、インプリメントや製品のポリシがそれぞれ違うためです。`t_optmgt(3)` リファレンス・ページおよびプロトコル固有のリファレンス・ページにあるオプションのリストは最大限のものです、必ずしも一般的なインプリメントの慣例に従っているとは限りません。各ベンダは、それぞれの要件に合ったサブセットをインプリメントしています。したがって、オプションを不用意に使用すると、それぞれのシステムのプラットフォーム間の移植性が低くなります。

XTI でアクセス可能なプロトコル・プロファイルのどのインプリメントのオプションも、省略時の設定ではすべて使用できます。アプリケーションは、したがって、オプションをまったく気にしないで作成することもできます。

アプリケーション・プログラムが、XTI 関数で取り出したオプションを処理する場合は、未知のオプションを破棄するようにしてください。これにより、それぞれのシステム・プラットフォームからの依存性、およびサポートするオプションが増える予定の将来の XTI リリースからの依存性を軽減できます。

3.6.7.2 XPG3 でのプロトコル・オプションの折衝

Tru64 UNIX による XTI の XPG3 のインプリメントは、オプションの関数 `t_optmgmt` を提供します。この関数はトランスポート・プロバイダとの間でプロトコル・オプションの取り出し、確認、および折衝を行います。終端を `t_open` で作成してアドレスをバインドすると、トランスポート・プロバイダとの間でオプションを確認または折衝できます。

注意

他のトランスポート・プロバイダには `t_optmgmt` 関数をサポートしているものもありますが、このオペレーティング・システムで提供されている TCP トランスポート・プロバイダはサポートしていません。オプション管理については、トランスポート・プロバイダのドキュメントを参照してください。

関数の構文、パラメータ、エラーについては、`t_optmgmt(3)` を参照してください。XTI のエラーの概要の説明は、3.7 節を参照してください。

`t_optmgmt` 関数は、正常終了すると 0 を返します。それ以外の場合は -1 を返し、`t_errno` に、3.7 節に示されている値の 1 つが設定されます (マルチスレッド・アプリケーションの場合、`t_errno` はスレッド固有です)。

3.7 XTI エラー

XTI は、ライブラリ・エラーおよびシステム・エラーを返します。XTI 関数はエラーを検出すると -1 を返し、次のいずれかの処理を行うことができます。

- 外部変数 `t_errno` をチェックして、特定のエラーを取得する。マルチスレッド・アプリケーションに対しては、`t_errno` はスレッドに固有。

- `t_error` 関数を呼び出して、`t_errno` に格納されたエラーに関連するメッセージ・テキストをプリントする。
- `t_getstate` 関数を使用して、トランスポート終端の状態をチェックする。エラーによっては、終端の状態が変更される。

注意

XTI 関数の呼び出しが正常終了しても、`t_errno` の内容はクリアされないので、エラーが発生した後だけ `t_errno` をチェックしてください。

<xti.h> ヘッダ・ファイルは、次のように `t_errno` 変数をマクロとして定義します。

```
#define t_errno (*(_terrno()))
```

エラーについての詳細は、それぞれの XTI リファレンス・ページを参照してください。

3.8 XTI トランスポート・プロバイダの構成

XTI トランスポート・プロバイダを構成するためには、`xtiso` カーネル構成オプションを使用します。`xtiso` オプションは、インストール時にシステムに構成することも、また、`doconfig` コマンドを使用してシステムに追加することもできます。詳細は『インストール・ガイド』を参照してください。

`doconfig` コマンドは、次のいずれかの方法で使用できます。

- カーネルをカスタマイズしていない場合は、オプションを指定せずに `doconfig` コマンドを使用する。オプションを指定しない場合、`doconfig` コマンドは、システム用に新しいカーネル構成ファイルを作成する。
- カーネルをカスタマイズしてあり、再度カスタマイズをしたくない場合は、`doconfig -c` コマンドを使用する。`doconfig -c` コマンドを使用すると、既存のカーネル構成ファイルに、情報を追加できる。

オプションを指定せずに doconfig コマンドを使用する場合は、次の手順に従ってください。

1. スーパユーザ・プロンプト (#) から /usr/sbin/doconfig コマンドを入力する。

2. カーネル構成ファイルの名前を入力する。

ファイル名は大文字のシステム名であり、通常は省略時の設定として、大カッコ ([]) の中に表示されます。次に例を示します。

```
Enter a name for the kernel configuration file. [HOST1]: RETURN
```

3. システム構成ファイルを置き換えるかどうかを尋ねられたら、y を入力する。

次に例を示します。

```
A configuration file with the name 'HOST1' already exists.  
Do you want to replace it? (y/n) [n]: y
```

```
Saving /sys/conf/HOST1 as /sys/conf/HOST1.bck
```

```
*** KERNEL CONFIGURATION AND BUILD PROCEDURE ***
```

4. カーネル・オプション選択メニューから X/Open Transport Interface (XTISO, TIMOD, TIRDWR) を選択する。システムが確認を求めるので、選択の確認を行う。

次に例を示します。

```
*** KERNEL OPTION SELECTION ***
```

Selection	Kernel Option

1	System V Devices
2	NTP V3 Kernel Phase Lock Loop (NTP_TIME)
3	Kernel Breakpoint Debugger (KDEBUG)
4	Packetfilter driver (PACKETFILTER)
5	Point-to-Point Protocol (PPP)
6	STREAMS pckt module (PCKT)
7	X/Open Transport Interface (XTISO, TIMOD, TIRDWR)
8	File on File File System (FFM)
9	ISO 9660 Compact Disc File System (CDFS)
10	Audit Subsystem
11	ACL Subsystem
12	Logical Storage Manager (LSM)
13	Advanced File System (ADVFS)
14	All of the above
15	None of the above
16	Help

```
Enter the selection number for each kernel option you want.  
For example, 1 3 [15]: 7
```

```
Enter the selection number for each kernel option you want.  
For example, 1 3 : 7
```

```
You selected the following kernel options:
```

```
      X/Open Transport Interface (XTISO, TIMOD, TIRDWR)  
Is that correct? (y/n) [y]: y
```

```
Configuration file complete.
```

5. 構成ファイルを編集するかどうかを尋ねられるので、nを入力する。

doconfig コマンドは、デバイス特殊ファイルを作成し、作成したファイルのログをいれる場所を示して、新しいカーネルを構築します。新しいカーネルが構築されたら、カーネルを doconfig が置いたディレクトリからルート・ディレクトリ (/) に移動して、システムをリブートしなければなりません。

リブートすると、strsetup -i コマンドが自動的に実行され、新しい STREAMS モジュールに対してデバイス特殊ファイルが作成されます。

6. strsetup -c コマンドを実行して、デバイスが正しく構成されたことを確認する。

strsetup -c コマンドからの出力例を次に示します。

```
# /usr/sbin/strsetup -c
```

```
STREAMS Configuration Information...Fri Nov  3 14:23:36 1995
```

Name	Type	Major	Module ID
----	----	-----	-----
clone		32	0
dlb	device	52	5010
kinfo	device	53	5020
log	device	54	44
nuls	device	55	5001
echo	device	56	5000
sad	device	57	45
pipe	device	58	5304
xtisoUDP	device	59	5010
xtisoTCP	device	60	5010
xtisoUDP+	device	61	5010
xtisoTCP+	device	62	5010
ptm	device	63	7609
pts	device	6	7608
bba	device	64	24880
lat	device	5	5
pppif	module		6002
pppasync	module		6000
pppcomp	module		6001
bufcall	module		0
null	module		5002
pass	module		5003
errm	module		5003
ptem	module		5003
spass	module		5007
rpass	module		5008
pipemod	module		5303


```

timod      module      5006
tirdwr     module      0
ldtty      module      7701

```

```
Configured devices = 15, modules = 14
```

doconfig -c コマンドを使用して XTISO オプションをカーネル構成ファイルに追加する場合は、次の手順に従ってください。

1. スーパユーザ・プロンプト (#) から doconfig -c *HOSTNAME* コマンドを入力する。

HOSTNAME は大文字のシステム名です。たとえば、host1 というシステムの場合には、次のように入力します。

```
# doconfig -c HOST1
```

2. XTISO をカーネル構成ファイルのオプション・セクションに追加する。カーネル構成ファイルを編集するかどうかを尋ねられるので、y を入力する。

doconfig コマンドを実行すると、ed エディタを使用して構成ファイルを編集できます。ed エディタの使用方法については、ed(1) を参照してください。

次の例は、ed 編集セッションで、host1 用のカーネル構成ファイルに XTISO オプションを追加する方法を示しています。新しい行を追加する行番号は、カーネル構成ファイルによって異なる場合があります。

```

*** KERNEL CONFIGURATION AND BUILD PROCEDURE ***

Saving /sys/conf/HOST1 as /sys/conf/HOST1.bck

Do you want to edit the configuration file? (y/n) [n]: y

Using ed to edit the configuration file. Press return when ready,
or type 'quit' to skip the editing session:
2153

48a
options      XTISO
.
1,$w
2185
q

*** PERFORMING KERNEL BUILD ***

```

3. 新しいカーネルが構成されたら、doconfig が置いたディレクトリからルート・ディレクトリ (/) へそのカーネルを移動して、システムをリブートする。

リブートすると、`strsetup -i` コマンドが自動的に実行され、新しい STREAMS モジュールに対してデバイス特殊ファイルが作成されます。

4. `strsetup -c` コマンドを実行して、デバイスが正しく構成されていることを確認する。

次は、`strsetup -c` コマンドからの出力例です。

```
# /usr/sbin/strsetup -c

STREAMS Configuration Information...Fri Nov  3 14:23:36 1995

      Name      Type  Major  Module ID
      ----      -
clone          32      0
dlb            device  52      5010
kinfo          device  53      5020
log            device  54      44
nuls           device  55      5001
echo           device  56      5000
sad            device  57      45
pipe           device  58      5304
xtisoUDP        device  59      5010
xtisoTCP        device  60      5010
xtisoUDP+       device  61      5010
xtisoTCP+       device  62      5010
ptm            device  63      7609
pts            device   6      7608
bba            device  64      24880
lat            device   5         5
pppif          module      6002
pppasync       module      6000
pppcomp        module      6001
bufcall        module      0
null           module      5002
pass           module      5003
errm           module      5003
ptem           module      5003
spass          module      5007
rspass         module      5008
pipemod        module      5303
timod          module      5006
tirdwr         module      0
ldtty          module      7701

Configured devices = 15, modules = 14
```

カーネルの再構成および `doconfig` コマンドについての詳細は、『システム管理ガイド』を参照してください。

ソケット

オペレーティング・システムのソケット・プログラミング・インタフェースは、XNS5.0 標準，XNS4.0 標準，POSIX 1003.1g Draft 6.6，および Berkeley Software Distribution (BSD) のソケット・プログラミング・インタフェースをサポートしています。さらに、オペレーティング・システムは、RFC 2553 で定義されている IPv6 (Internet Protocol Version 6) 用の基本ソケット・インタフェース拡張ソケットに対応しています。ソケット機能の基本の構文はそのままです。既存の IPv4 のアプリケーションは以前同様に動作しますし、IPv6 のアプリケーションは IPv4 アプリケーションと相互運用することができます。

このオペレーティング・システムでは、ソケットは、インターネット・プロトコル群 (TCP/IP) および同じシステム上でのプロセス間通信の UNIX ドメインとのインタフェースを提供します。ただし、ソケットを使用すると、基礎となるネットワーキング・プロトコルおよびハードウェアに依存しない、ネットワーク・ベースのアプリケーションを作成することができます。

プログラム中で XNS4.0 標準のインプリメンテーションを使用するためには、`c89` コンパイラ・コマンドでプログラムをコンパイルしなければなりません。詳細については、`standards(5)` を参照してください。この章で使用する例は、XNS4.0 標準に基づいています。XNS4.0，XNS5.0，POSIX 1003.1g Draft 6.6，および BSD の各インタフェースの相違点についての詳細は、4.5 節を参照してください。

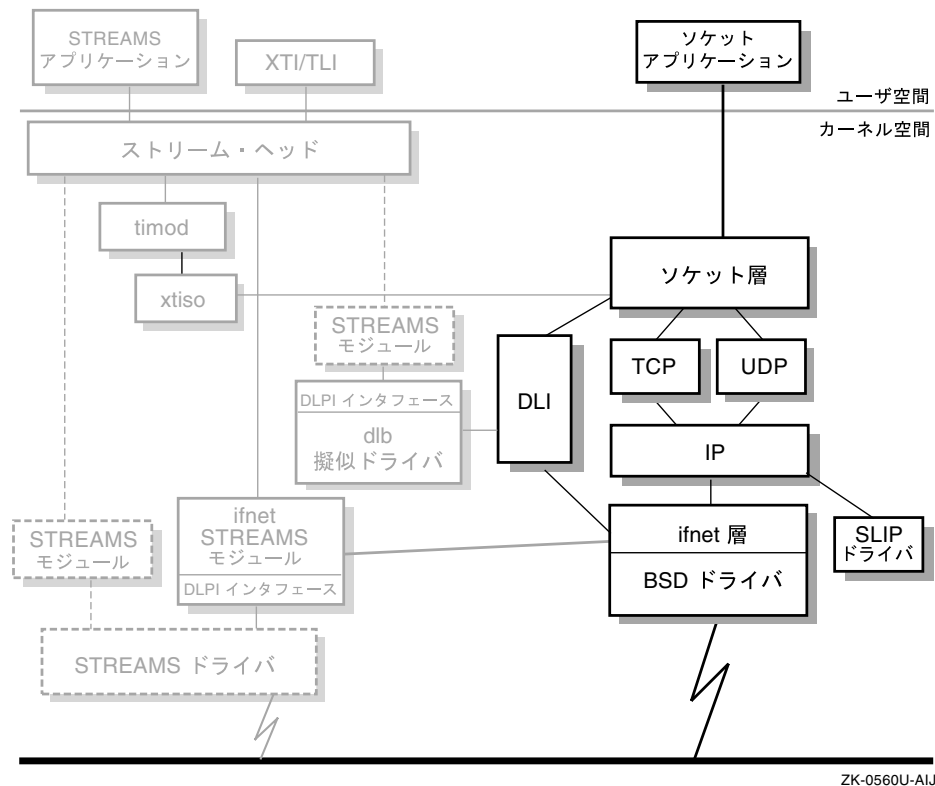
この章では、次の事項について説明します。

- ソケット・フレームワークの概要
- アプリケーションとソケットとのインタフェース
- ソケットの使用方法
- ネットワーク・アプリケーションの作成
- BSD ソケット・インタフェースについての情報

- 一般的なソケット・エラー・メッセージの説明
- 高度なソケット・プログラミング情報

図 4-1は、ソケット・フレームワークを強調表示して、ネットワーク・プログラミング環境における他の部分との関係を示しています。

図 4-1: ソケット・フレームワーク



4.1 ソケット・フレームワークの概要

ソケット・フレームワークは、次のものから構成されます。

- ソケット通信プロパティを定義する、通信ドメインやソケット・タイプなどの抽象的な表現のセット。
- アプリケーション・プログラムがソケット・フレームワークにアクセスするときに使用する、プログラミング・インタフェース。システム・コールおよびライブラリ・コールのセットからなる。

4-2 ソケット

- アプリケーション・プログラムがシステム・コールおよびライブラリ・コールを使用してアクセスする，ネットワーキング・プロトコルを含むカーネル・リソース。

オペレーティング・システムは，プロセス間通信の実現のために，ソケットを使用してインターネット・プロトコル群と UNIX ドメインをインプリメントしています。また，ソケット・システム・コールを使用してアクセスする，BSD ベースのデバイス・ドライバもインプリメントしています。

4.1.1 ソケットの通信プロパティ

この項では，ソケット通信プロパティの基礎となる，抽象表現およびその定義について説明します。

4.1.1.1 ソケット抽象表現

ソケットは，通信の終端として機能します。単一のソケットは，1 つの終端です。1 組のソケットで，双方向の通信チャネルを構成して，関連のないプロセス間でローカルに，またはネットワークを通して，データ交換を行うことができます。

アプリケーション・プログラムは，必要に応じて，ソケットの作成をオペレーティング・システムに要求します。オペレーティング・システムはソケット記述子を返すので，プログラムはこの記述子を使用して，新しく作成されたソケットを参照し，さまざまなオペレーションを行います。

ソケットには次の特性があります。

- ソケットを参照する記述子を保持しているプロセスがある間だけ存在する。
- 記述子によって参照され，文字型特殊デバイスと同じ特質を持つ。適切なシステム・コールを使用することによって，ソケット上で読み取り，書き込み，および選択オペレーションが実行される。
- ペアで作成できる。また，名前を指定し，同じ通信ドメインの他のソケットとのランデブに使用して，それらのソケットからの接続を受け入れたり，メッセージを交換したりすることができる。

ソケットは，通信プロパティに応じてタイプが決まります。使用できるソケット・タイプについては，4.1.1.3 項を参照してください。

4.1.1.2 通信ドメイン

通信ドメインによって、ハードウェアおよびソフトウェアが異なるシステム間の通信の意味が定義されます。通信ドメインでは次のことを指定します。

- プロトコル・ファミリと呼ばれるプロトコルのセット。
- 名前を操作および解釈するための規則のセット。
- 1つのアドレス・ファミリを構成する、関連のあるソケット・アドレス・フォーマットの集まり。

インターネット通信ドメインのソケット・アドレスには、インターネット・アドレスとポート番号が入っています。UNIX 通信ドメインのソケット・アドレスには、ローカル・パス名が入っています。

ソケットに関連するデータ構造体についての詳細は、4.2.3.4 項を参照してください。

オペレーティング・システムは、省略時に次のソケット・ドメインをサポートします。¹

- UNIX ドメイン

オペレーティング・システムは、AF_UNIX のドメインが指定された場合、同一システム上で実行しているプロセス間にソケット通信を提供します。UNIX 通信ドメインでは、ソケットは、/dev/printer のように、UNIX パス名で指定されます。

- インターネット・ドメイン

オペレーティング・システムは、AF_INET か AF_INET6 のドメインが指定された場合、ローカルで実行しているプロセスとリモート・ホスト上で実行しているプロセスとの間にソケット通信を提供します。このドメインには、使用しているシステムで TCP/IP が構成されて動作している必要があります。

表 4-1 は、UNIX ドメインおよびインターネット・ドメインの特性をまとめたものです。

¹ オペレーティング・システムは、AF_DLI ドメインをサポートするようにも構成できません。データ・リンク・インタフェース、および AF_DLI ドメインの使用方法についての詳細は、付録 F を参照してください。

表 4-1: UNIX 通信ドメインおよびインターネット通信ドメインの特性

	UNIX	インターネット
ソケット・タイプ	SOCK_STREAM, SOCK_DGRAM	SOCK_STREAM, SOCK_DGRAM, SOCK_RAW。
名前	ASCII 文字列。たとえば, /dev/printer。	32 ビットの IP Version 4 アドレスと, 16 ビットのポート番号 (AF_INET)。128 ビットの IP Version 6 アドレスと, 16 ビットのポート番号 (AF_INET6)。
セキュリティ	パス名と接続しているプロセスは, そのパスへの書き込みアクセスを持っていなければならない。	利用できない。
rawアクセス	利用できない。	特権プロセスは, IP の raw ファシリティにアクセスできる。raw ソケットは, 1 つの IP プロトコル番号に関連付けられ, そのプロトコル用に受信したすべてのトラフィックを受信する。

4.1.1.3 ソケット・タイプ

各ソケットには, 抽象表現のタイプが関連付けられており, そのソケット・タイプを使用する通信の意味を記述します。メッセージの信頼性, 順序付け, 重複の防止などのプロパティは, このソケット・タイプによって決定されます。ソケット・タイプの基本的なセットは, <sys/socket.h> ヘッダ・ファイルに定義されています。

注意

通常, ヘッダ・ファイル名は山カッコ (< >) で囲まれています。ヘッダ・ファイルへの絶対パスは, 山カッコ内のヘッダ・ファイル名の前に /usr/include/ を付けたものです。
<sys/socket.h> の場合, socket.h は /usr/include/sys ディレクトリにあります。

UNIX ドメインおよびインターネット・ドメインでは, 次のソケット・タイプを使用できます。

SOCK_DGRAM

固定最大長のコネクションレス型メッセージであるデータグラムを提供する。

各メッセージは、個別にアドレスを指定できます。このソケット・タイプでは、メッセージの引き渡し順序および信頼性が保証されないので、一般に短いメッセージに使用されます。データグラム・ソケットの重要な特性は、データのレコード境界が保存されるので、個々のデータグラムを別々に読み取ることにあります。

データグラムは、`finger` プログラムのような受信側からの 1 つ以上の応答を必要とする要求に対してよく使用されます。指定された時間内に受信側が応答しない場合、送信側アプリケーションは要求を繰り返すことができます。この時間は、通信ドメインによって異なります。

UNIX ドメインでは、SOCK_DGRAM はメッセージ・キューに似ています。インターネット・ドメインでは、SOCK_DGRAM はユーザ・データグラム・プロトコル (UDP) を使用してインプリメントされます。

SOCK_STREAM

帯域外データ用に、接続を通し、伝送機能を伴う連続した双方向バイト・ストリームを提供する。

データの伝送は、高い信頼性で、順番に行われます。

UNIX ドメインでは、SOCK_STREAM は、全二重タイプに似ています。インターネット・ドメインでは、SOCK_STREAM は、伝送制御プロトコル (TCP) を使用してインプリメントされます。

SOCK_RAW

ネットワーク・プロトコルおよびインタフェースへのアクセスを提供する。

raw ソケットは、特権プロセスだけが利用できます。raw ソケットを使用すると、アプリケーションは、下位レベルの通信プロトコルに直接アクセスできます。raw ソケットは、通常のインタフェースを

通しては直接アクセスできないプロトコル機能を使用したい上級ユーザ，または既存の下位レベルのプロトコルを使用して，新しいプロトコルを構築したい上級ユーザが使用することを目的としています。また，SOCK_RAW は，ハードウェア・インタフェースとの通信にも使用できます。

raw ソケットは，通常，データグラム指向です。ただし，厳密な特性は，プロトコルが提供するインタフェースに依存します。raw ソケットは，インターネット・ドメインだけで利用できます。

4.1.1.4 ソケット名

ソケットに名前を付けることによって，システムまたはネットワーク上の関連のないプロセスが，特定のソケットを指定して，そのソケットとデータを交換することができます。バインドされた名前は，可変長のバイト列であり，サポートしている 1 つまたは複数のプロトコルによって解釈されます。名前の解釈は，通信ドメインによって異なります。インターネット・ドメインでは，名前はインターネット・アドレスとポート番号から構成され，ファミリーは AF_INET か AF_INET6 のいずれかです。AF_INET ソケットは IPv4 通信をサポートし，AF_INET6 ソケットは IPv4 通信と IPv6 通信の両方をサポートします。UNIX ドメインでは，名前はパス名であり，ファミリーは AF_UNIX です。

互いに通信しているプロセスは，関連付け (association) によってバインドされます。インターネット・ドメインでは，関連付けは，プロトコル，ローカル・アドレスと外部アドレス，およびローカル・ポートと外部ポートから構成されます。インターネット・ドメインのソケットに名前をバインドすると，ローカル・アドレスおよびローカル・ポートが指定されます。

UNIX ドメインでは，関連付けは，ローカル・パス名から構成されます。UNIX ドメインのソケットに名前をバインドすると，パス名が指定されます。

ほとんどのドメインにおいて，関連付けは一意でなければなりません。

4.2 アプリケーションとソケットとのインタフェース

カーネルにソケットをインプリメントすると，ネットワーキング・サブシステムは，次の 3 つの対話処理層に分かれます。

- ソケット層

アプリケーション・プログラムと、伝送制御プロトコル (TCP) またはユーザ・データグラム・プロトコル (UDP) と IP などの下位層との間のインタフェースを提供します。

- プロトコル層

トランスポート層プロトコル (TCPおよびUDP) およびネットワーク層プロトコル (IP) から構成されます。

- デバイス層

ifnet 層およびデバイス・ドライバから構成されます。

4.1.1 項で説明した抽象表現の他に、ソケット・インタフェースには、システム・コールとライブラリ・コール、ライブラリ関数、およびデータ構造体があります。これらにより、ユーザは、ソケットを操作したり、データの送受信を行うことができます。

さらに、カーネルはソケット・フレームワークに対して補助サービスを提供します。これには、バッファ管理、メッセージのルーティング、プロトコルへの標準インタフェース、およびさまざまなネットワーク・プロトコルを使用するためのネットワーク・インタフェース・ドライバへのインタフェースなどが含まれます。

4.2.1 通信モード

ソケット・フレームワークは、コネクション指向型およびコネクションレス型の両通信モードをサポートします。コネクション指向型通信では、アプリケーションは、コネクション指向型プロトコルをサポートするソケット・タイプを通信ドメインに指定します。たとえば、アプリケーションは、AF_INET ドメインで SOCK_STREAM ソケットをオープンできます。AF_INET ドメインおよび AF_INET6 ドメインの SOCK_STREAM ソケットは TCP プロトコルによってサポートされ、これは、コネクション指向型プロトコルです。

コネクションレス型通信では、アプリケーションは、コネクションレス型プロトコルをサポートするソケット・タイプを通信ドメインに指定します。たとえば、AF_INET 通信ドメインの SOCK_DGRAM ソケットは、UDP プロトコルによってサポートされ、これは、コネクションレス型プロトコルです。

4.2.1.1 コネクション指向型通信

TCP は、オペレーティング・システムでインプリメントされているコネクション指向型プロトコルです。TCP は、信頼性の高い端末相互トランスポート・プロトコルであり、消失データ、伝送エラー、および介入するゲートウェイにおける障害からの回復機能を提供します。TCP では、2 つのプロセスが通信前に接続されていることを要求するため、データが正確に引き渡されることが保証されます。TCP/IP 接続は、電話回線接続とよく比較されます。AF_INET もしくは AF_INET6 ドメインの SOCK_STREAM ソケットを介して送信されるデータは、セグメントに分けられ、連番によって識別されます。リモート・プロセスは、肯定応答に連番を含めることによって、データの受信を確認します。データが途中で失われた場合には、再送信されるため、データは確実に正しい順序でアプリケーションに届きます。

大量のデータを交換し、データの到着順序が重要なアプリケーションでは、コネクション指向型通信が適しています。ファイル転送プログラムは、TCP によって提供されるコネクション指向モードの通信を利用するアプリケーションの良い例です。

4.2.1.2 コネクションレス型通信

UDP は、オペレーティング・システムでインプリメントされているコネクションレス型プロトコルです。UDP の機能を次に示します。

- メッセージのアドレス情報に基づいてメッセージを引き渡す。
- 通信プロセス間に接続を必要としない。
- データの到着を確実にする肯定応答を使用しない。
- 着信メッセージの順序付けを行わない。
- ホスト間のデータ交換速度を制御するためのフィードバックを行わない。

UDP メッセージは、失われたり、重複したり、または順序どおりに到着しないことがあります。

交換するデータ量が少なく、到着順序が重要でない場合は、コネクションレス型通信が適しています。コネクションレス・モードの通信を使用した例の 1 つに、rwhod デーモンがあります。この rwhod デーモンは、システム情報を含む UDP パケットを定期的にネットワークにブロードキャストします。rwhod デーモンでは、パケットが送信されたかどうかや、送信順序はあまり重要ではありません。

UDP は、IP マルチキャストを使用して、データグラムをローカル・エリア・ネットワーク上のホストのサブセットに引き渡すアプリケーションにも適しています。

4.2.2 クライアント/サーバ・モデル

分散型アプリケーションの作成において最も一般に使用されるモデルは、クライアント/サーバ・モデルです。サーバ・プロセスは、ネットワークに対してサービスを提供し、クライアント・プロセスは、これらのサービスを使用します。クライアントとサーバが、サービスを提供して、受け入れるためには、周知の規約を定めておく必要があります。この規約とは、プロトコルを接続の両端にインプリメントしなければならないということです。状況に応じて、このプロトコルは、コネクション指向型 (非対称)、またはコネクションレス型 (対称) のいずれかにすることができます。

TCP などのコネクション指向型プロトコルにおいては、通信の片側は常にサーバとして認識され、もう片側はクライアントとして認識されます。サーバは、サービスに関連付けられた周知のアドレスにソケットをバインドし、次に、そのソケット上でパッシブにリッスンします。クライアントは、サーバのソケットに対する接続を開始することによって、サーバからのサービスを要求します。サーバが接続を受け入れると、サーバとクライアントはデータを交換できます。コネクション指向型プロトコルの例として、Telnet があります。

UDP などのコネクションレス型プロトコルにおいては、通信のどちら側も、サーバまたはクライアントの役割を果たすことができます。クライアントは、サーバとの接続を確立しません。代わりに、データグラムをサーバのアドレスに送信するだけです。同様に、サーバも、クライアントからの接続を受け入れません。代わりに、`recvfrom` システム・コールを発行して、クライアントからのデータが到着するまで待ちます (4.3.6 項参照)。

4.2.3 システム・コール、ライブラリ・コール、ヘッダ・ファイル、およびデータ構造体

この項では、ソケット層に属するシステム・コールおよびライブラリ・コールの一覧を示します。また、ソケットに関連する定数と構造体を定義するヘッダ・ファイルの一覧も示して、これらのヘッダ・ファイルに記述されている最も重要なデータ構造体について説明します。

4.2.3.1 ソケット・システム・コール

表 4-2 は、ソケット・システム・コールと、その機能の簡単な説明の一覧です。各システム・コールには、同じ名前の関連するリファレンス・ページがあります。

表 4-2: ソケット・システム・コール

システム・コール	説明
<code>accept</code>	新しいソケットを作成するために、ソケット上で接続を受け入れる。
<code>bind</code>	名前をソケットにバインドする。
<code>connect</code>	ソケットで接続を開始する。
<code>getpeername</code>	接続された対等プロセスの名前を取得する。
<code>getsockname</code>	ソケット名を取得する。
<code>getsockopt</code>	ソケットのオプションを取得する。
<code>listen</code>	ソケットの接続をリッスンし、キューにいれる要求の最大数を指定する。
<code>recv</code>	メッセージを受信し、着信データの中身を見て、帯域外データを受信する。
<code>recvfrom</code>	メッセージを受信する。 <code>recv</code> システム・コールのすべての機能を行うほか、対等プロセスのアドレスを提供する。
<code>recvmsg</code>	メッセージを受信する。 <code>recv</code> および <code>recvfrom</code> システム・コールのすべての機能を行うほか、特殊に解釈されたデータ (アクセス権) を受信し、メッセージ・バッファに対して分散 I/O オペレーションを行う。
<code>send</code>	メッセージを送信する。ネットワーク・ルーティングを指定せずに、帯域外データおよび通常データも送信する。
<code>sendmsg</code>	メッセージを送信する。 <code>send</code> および <code>sendto</code> システム・コールのすべての機能を行うほか、特殊に解釈されたデータ (アクセス権) を伝送し、メッセージ・バッファに対して収集 I/O オペレーションを行う。
<code>sendto</code>	メッセージを送信する。 <code>send</code> システム・コールのすべての機能を行うほか、対等プロセスのアドレスを提供する。
<code>setsockopt</code>	ソケット・オプションを設定する。

表 4-2: ソケット・システム・コール (続き)

システム・コール	説明
shutdown	すべてのソケット送受信オペレーションをシャットダウンする。
socket	通信用の終端を作成し，記述子を返す。
socketpair	接続した 1 組のソケットを作成する。

4.2.3.2 ソケット・ライブラリ・コール

アプリケーション・プログラムは，分散環境におけるプロセス間通信機能を使用するために，ソケット・ライブラリ・コールを使用して，ネットワーク・アドレスを構築します。

ネットワーク・ライブラリ・サブルーチンは，次の項目をマップします。

- ホスト名をネットワーク・アドレスにマップする。
- ネットワーク名をネットワーク番号にマップする。
- プロトコル名をプロトコル番号にマップする。
- サービス名をポート番号にマップする。

名前およびアドレスの操作を簡単にするために，追加のソケット・ライブラリ・コールが用意されています。

ソケット・ライブラリ・コールを使用する場合は，アプリケーション・プログラムに，<netdb.h> ヘッダ・ファイルをインクルードしなければなりません。

ホスト名

アプリケーション・プログラムは，次のネットワーク・ライブラリ・ルーチンを使用して，インターネット・ホスト名をアドレスにマップします。

- gethostbyname (AF_INET のみ)
- gethostbyaddr (AF_INET のみ)
- getaddrinfo (AF_INET および AF_INET6)
- getnameinfo (AF_INET および AF_INET6)

gethostbyname ルーチンは、インターネット・ホスト名を受け取り、hostent 構造体を返します。一方、gethostbyaddr ルーチンは、インターネット・ホスト・アドレスを hostent 構造体にマップします。hostent 構造体は、次の構成要素からなります。

```
struct hostent {
    char *h_name;           /* official name of host */
    char **h_aliases;       /* alias list */
    int h_addrtype;         /* host address type (AF_INET or AF_INET6) */
    int h_length;           /* length of address */
    char **h_addr_list;     /* list of addresses, null terminated
                             first address, network byte order */
#define h_addr h_addr_list[0]
};
```

gethostbyaddr サブルーチンおよび gethostbyname サブルーチンは、ホストの公式名と公用の別名を返します。また、アドレス・ファミリおよび可変長アドレスのヌル終了リストも同時に返されます。このアドレスのリストが必要なのは、ホストが、同じ名前で多くのアドレスを持つ可能性があるからです。

これらのライブラリ・コールのデータベースは、/etc/hosts ファイルです。named ネーム・サーバが実行中の場合は、ホスト・データベースはネットワーク上の指定サーバ上で保持されます。データベース間やそのアクセス・プロトコル間の相違のために、返される情報が異なることがあります。/etc/hosts バージョンの gethostbyname を使用している場合は、1つのアドレスだけが返されますが、リストされている別名はすべて含まれます。named バージョンの場合には、代替アドレスを返すことができますが、パラメータ値として指定されたもの以外の別名は返しません。

getaddrinfo ルーチンは、インターネット・ノード名またはサービス名を受け取り、1つ以上の addrinfo 構造体を返します。一方、getnameinfo ルーチンは、sockaddr を受け取り、ホスト名 (要求された場合) と、そのポートのサービス名 (要求された場合) を返します。

これらのコールのデータベースは、/etc/ipnodes ファイルと /etc/hosts ファイルの両方です。named ネーム・サーバが実行中の場合は、ホスト・データベースはネットワーク上の指定サーバ上で保持されます。データベース間やそのアクセス・プロトコル間の相違のために、返される情報が異なることがあります。/etc/ipnodes および /etc/hosts バージョンの getaddrinfo を使用している場合は、要求に応じて、2つのアドレス (各ファイルから1つずつ) が返されることがあります。別名は返されません。

named バージョンの場合には、代替アドレスを返すことができますが、パラメータ値として指定されたもの以外の別名は返しません。

ネットワーク名

アプリケーション・プログラムは、次のネットワーク・ライブラリ・ルーチンを使用して、ネットワーク名をネットワーク番号に、ネットワーク番号をネットワーク名にマップします。

- `getnetbyaddr`
- `getnetbyname`
- `getnetent`

`getnetbyaddr`、`getnetbyname`、および `getnetent` の各ルーチンは、`/etc/networks` ファイルから情報を取り出し、次のような `netent` 構造体を返します。

```
struct netent {
    char      *n_name;      /* official name of net */
    char      **n_aliases; /* alias list */
    int       n_addrtype;   /* net address type */
    in_addr_t n_net;        /* network number, host byte order */
};
```

プロトコル名

アプリケーション・プログラムは、次のネットワーク・ライブラリ・ルーチンを使用して、プロトコル名をプロトコル番号にマップします。

- `getprotobynumber`
- `getprotobyname`
- `getprotoent`

`getprotobynumber`、`getprotobyname`、および `getprotoent` の各サブルーチンは、`/etc/protocols` ファイルから情報を取り出し、次のような `protoent` エントリを返します。

```
struct protoent {
    char *p_name;      /* official protocol name */
    char **p_aliases; /* alias list */
    int  p_proto;      /* protocol number */
};
```


サービス名

アプリケーション・プログラムは、次のネットワーク・ライブラリ・ルーチンを使用して、サービス名をポート番号にマップします。

- `getservbyname`
- `getservbyport`
- `getservent`

サービスは、特定のポートに常駐し、特定の通信プロトコルを使用するものとされています。この考え方は、インターネット・ドメインでは一貫していますが、他のネットワーク・アーキテクチャでは一貫していません。さらに、サービスは、複数のポートに常駐することがあります。この場合には、上位レベルのライブラリ・ルーチンを迂回するか、または拡張しなければなりません。利用できるサービスは、`/etc/services` ファイルに入っています。サービス・マッピングは、次のような `servent` 構造体によって記述されます。

```
struct servent {  
    char *s_name;      /* official service name */  
    char **s_aliases; /* alias list */  
    int  s_port;       /* port number, network byte order */  
    char *s_proto;     /* protocol to use */  
};
```

`getservbyname` ルーチンは、サービス名、および修飾プロトコル(オプション)を指定することにより、サービス名を `servent` 構造体にマップします。したがって、次のライブラリ・コールは、任意のプロトコルを使用する Telnet サーバのサービス仕様を返します。

```
sp = getservbyname("telnet", (char *) NULL);
```

一方、次のライブラリ・コールは、TCP プロトコルを使用する Telnet サーバだけを返します。

```
sp = getservbyname("telnet", "tcp");
```

`getservbyport` ルーチンと `getservent` ルーチンも提供されています。`getservbyport` ルーチンには、`getservbyname` が提供するインタフェースと類似したインタフェースがあり、必要に応じてプロトコル名を指定することにより、限定して検索することができます。

ネットワーク・バイト順の変換

プログラムでインターネット・プロトコル (IP) 群のデータを作成または解釈する必要がある場合には、標準の変換方法を使用します。IP 群では、特定のデータ・フォーマットを使用することによって、一貫性を保証します。オペレーティング・システムでは、プログラムがそのフォーマットとの間でデータ変換を行うことができるようにする関数を提供しています。また、インターネット・プロトコル群では、最上位バイトが最下位アドレスにあると想定します。このようなフォーマットを *big-endian* と呼びます。ネットワーク・バイト順とホスト・バイト順を相互に変換する関数が用意されています。

次の 4 つの関数によって、ネットワークが、ユーザのプログラムから渡されるデータを正しく解釈し、また、ユーザのプログラムが、ネットワークから渡されるデータを解釈することが保証されます。

- `htonl`
- `htons`
- `ntohl`
- `ntohs`

アプリケーション・プログラムは、次の互いに関連するネットワーク・ライブラリ・ルーチンを使用して、インターネット・アドレス文字列とバイナリ・アドレスの操作を行います。

- `inet_addr` (AF_INET のみ)
- `inet_lnaof` (AF_INET のみ)
- `inet_makeaddr` (AF_INET のみ)
- `inet_netof` (AF_INET のみ)
- `inet_network` (AF_INET のみ)
- `inet_ntoa` (AF_INET のみ)
- `getnameinfo` (AF_INET および AF_INET6)
- `getaddrinfo` (AF_INET および AF_INET6)

表 4-3 に、ソケット・ライブラリ・コールの一覧と、簡単な説明を示します。各ライブラリ・コールには、同じ名前の関連するリファレンス・ページがあります。ソケット・ライブラリ・コールは `libc` の一部であるため、特別なライブラリをリンクする必要はありません。

表 4-3: ソケット・ライブラリ・コール

名前	説明
endhostent	一連のホスト・エントリ検索を終了する。
endnetent	一連のネットワーク・エントリ検索を終了する。
endprotoent	一連のプロトコル・エントリ検索を終了する。
endservent	一連のサービス・エントリ検索を終了する。
freeaddrinfo	getaddrinfo から返された addrinfo 構造体と記憶域を解放する。
getaddrinfo	ホストの名前とオプションのアドレス・ファミリが指定されると、ネーム・サーバ (named)、 <code>/etc/ipnodes</code> ファイル、または <code>/etc/hosts</code> ファイルのいずれかから、そのホスト・エントリを検索する。 標準の数値文字列フォーマットのノード・アドレスを、sockaddr 構造体のインターネット・アドレスに変換する。
gethostbyaddr	ホストのアドレスが指定されると、ネーム・サーバ (named) または <code>/etc/hosts</code> ファイルのいずれかから、そのホスト・エントリを検索する。
gethostbyname	ホスト名が指定されると、ネーム・サーバ (named) または <code>/etc/hosts</code> ファイルのいずれかから、そのホスト・エントリを検索する。
gethostent	ネーム・サーバ (named)、または <code>/etc/hosts</code> ファイル (必要に応じてこのファイルをオープンする) から、次のホスト・エントリを検索する。
getnameinfo	ノードのアドレスを含む sockaddr 構造体が指定されると、ネーム・サーバ (named)、 <code>/etc/ipnodes</code> ファイル、または <code>/etc/hosts</code> ファイルのいずれかから、そのホスト・エントリを検索する。 sockaddr 構造体のインターネット・アドレスを、標準の数値文字列フォーマットに変換する。
getnetbyaddr	ネットワークのアドレスが指定されると、そのネットワーク・エントリを <code>/etc/networks</code> ファイルから検索する。
getnetbyname	ネットワーク名が指定されると、そのネットワーク・エントリを <code>/etc/networks</code> ファイルから検索する。

表 4-3: ソケット・ライブラリ・コール(続き)

名前	説明
getnetent	/etc/networks ファイル (必要に応じてこのファイルをオープンする) から, 次のネットワーク・エントリを検索する。
getprotobyname	プロトコル名が指定されると, そのプロトコル・エントリを /etc/protocols ファイルから検索する。
getprotobynumber	プロトコル番号が指定されると, そのプロトコル・エントリを /etc/protocols ファイルから検索する。
getprotoent	/etc/protocols ファイル (必要に応じてこのファイルをオープンする) から, 次のプロトコル・エントリを検索する。
getservbyname	サービスの名前が指定されると, そのサービス・エントリを /etc/services ファイルから検索する。
getservbyport	サービスのポート番号が指定されると, そのサービス・エントリを /etc/services ファイルから検索する。
getservent	/etc/services ファイル (必要に応じてこのファイルをオープンする) から, 次のサービス・エントリを検索する。
htonl	32 ビット整数を, ホストのバイト順から, インターネット・ネットワークのバイト順に変換する。
htons	符号なしの短整数を, ホストのバイト順から, インターネット・ネットワークのバイト順に変換する。
inet_addr	インターネットの標準ドット (.) 表記で表現された番号を表す文字列を分割して, インターネット・アドレスを返す。
inet_lnaof	インターネット・ホスト・アドレスを分割して, ローカル・ネットワーク・アドレスを返す。
inet_makeaddr	インターネット・ネットワーク番号およびローカル・ネットワーク・アドレスから, インターネット・アドレスを構築する。
inet_ntoa	インターネット・アドレスを文字列に変換する。
inet_netof	インターネット・ホスト・アドレスを分割して, ネットワーク番号を返す。
inet_network	インターネットの標準ドット (.) 表記で表現された番号を表す文字列を分割して, インターネット・ネットワーク番号を返す。

表 4-3: ソケット・ライブラリ・コール(続き)

名前	説明
ntohl	32 ビット整数を、インターネット・ネットワーク標準バイト順から、ホストのバイト順に変換する。
ntohs	符号なしの短整数を、インターネット・ネットワークのバイト順から、ホストのバイト順に変換する。
sethostent	一連のホスト・エントリ検索を開始する。
setnetent	一連のネットワーク・エントリ検索を開始する。
setprotoent	一連のプロトコル・エントリ検索を開始する。
setservent	一連のサービス・エントリ検索を開始する。

4.2.3.3 ヘッダ・ファイル

ソケット・ヘッダ・ファイルには、データ定義、構造体、定数、マクロ、およびオプションが記述されており、ソケット・システム・コールおよびサブルーチンがこれらを使用します。アプリケーション・プログラムでは、特定のシステム・コールまたはサブルーチンに必要な構造体、およびその他の情報を使用するために、適切なヘッダ・ファイルをインクルードしなければなりません。表 4-4 は、一般に使用されるソケット・ヘッダ・ファイルの一覧です。

表 4-4: ソケット・インタフェースのヘッダ・ファイル

ファイル名	説明
<sys/socket.h>	データ定義およびソケット構造体が記述されている。このファイルは、すべてのソケット・アプリケーションにインクルードする必要がある。
<sys/types.h>	データ型定義が記述されている。このファイルは、すべてのソケット・アプリケーションにインクルードする必要がある。このヘッダ・ファイルは <sys/socket.h> でインクルードされている。
<sys/un.h>	UNIX ドメイン用の構造体を定義する。UNIX ドメインのソケットを使用する場合には、アプリケーションにこのファイルをインクルードする必要がある。

表 4-4: ソケット・インタフェースのヘッダ・ファイル (続き)

ファイル名	説明
<netinet/in.h>	インターネット・ドメイン用の定数と構造体を定義する。インターネット・ドメインで TCP/IP を使用する場合には、アプリケーションにこのファイルをインクルードする必要がある。
<netdb.h>	ソケット・サブルーチン用のデータ定義が記述されている。TCP/IP を使用し、ホスト・エントリ、ネットワーク・エントリ、プロトコル・エントリ、サービス・エントリのいずれかを検索する必要がある場合は、アプリケーションにこのファイルをインクルードする必要がある。

4.2.3.4 ソケットに関連するデータ構造体

この項では、次のデータ構造体について説明します。

- sockaddr
- sockaddr_in
- sockaddr_in6
- sockaddr_storage
- sockaddr_un
- msghdr
- cmsghdr

sockaddr 構造体には、ソケットのアドレス・フォーマットに関する情報が入っています。アプリケーションがソケットを作成する通信ドメインは、そのソケットのアドレス・フォーマットを決定し、そのデータ構造体も決定します。

ソケット・アドレスのデータ構造体は、4.2.3.3 項で説明しているヘッダ・ファイルに定義されています。どのヘッダ・ファイルが適切であるかは、作成するソケットのタイプによって異なります。使用可能なソケット・アドレスのデータ構造体の型は、次のとおりです。

```
struct sockaddr
```

汎用的なソケット・アドレス構造体を定義する。

このソケットは、14 バイトの直接アドレッシングに制限されています。sockaddr 構造体は <sys/socket.h> ファイルに記述され、次の要素が含まれます。

```
unsigned char    sa_len;           /* total length */
sa_family_t      sa_family;        /* address family */
char             sa_data[14];      /* actually longer;
                                   address value */
```

sa_len パラメータは、合計長さを定義します。sa_family パラメータは、ソケット・アドレス・ファミリまたはドメインを定義します。UNIX ドメインには AF_UNIX、インターネット・ドメインでは AF_INET もしくは AF_INET6 です。sa_data の内容は、使用しているプロトコルによって異なりますが、一般に、ソケット名は、マシン名、およびポート名またはサービス名から構成されます。

struct sockaddr_storage

ネットワークを介したマシン間通信およびローカルなプロセス間通信に使用されるインターネット・ドメイン・ソケット (AF_INET および AF_INET6 アドレス・ファミリ) を定義する。この構造体を使用すると、アプリケーションは複数のアドレス・ファミリを 1 つの変数として扱うことができます。sockaddr_storage 構造体は、<sys/socket.h> ファイルに記述されています。sockaddr_storage 構造体には、次の要素が含まれます。

```
unsigned char    ss_len;           /* address length */
sa_family_t      ss_family;        /* address family */
char             __ss_pad1[_SS_PAD1SIZE]; /* pad to alignment field */
ulong_t          __ss_align;       /* force structure alignment */
char             __ss_pad2[_SS_PAD2SIZE]; /* pad to desired size */
```

_SS_PAD1SIZE 変数と _SS_PAD2SIZE 変数も、<sys/socket.h> で定義されています。

struct sockaddr_un

同一マシン上におけるプロセス間通信に使用する、UNIX ドメインのソケットを定義する。

このソケットには、絶対パス名の指定が必要です。sockaddr_un 構造体は <sys/un.h> ヘッダ・ファイルに記述されています。sockaddr_un 構造体には、次の要素が含まれます。

```
unsigned char    sun_len;          /* sockaddr len including null*/
sa_family_t      sun_family;       /* AF_UNIX, address family*/
char             sun_path[];       /* path name */
```

UNIX ドメインのプロトコル (AF_UNIX) は、最大で PATH_MAX に 2 バイトを加えた長さのソケット・アドレスを持ちます。PATH_MAX パラメータは、パス名の最大バイト数を定義します。

```
struct sockaddr_in
```

ネットワークを通したマシン間通信，およびローカルなプロセス間通信に使用する，インターネット・ドメインのソケット (AF_INET アドレス・ファミリ) を定義する。

sockaddr_in 構造体は <netinet/in.h> ファイルに記述されています。sockaddr_in 構造体には，次の要素が含まれます。

```
unsigned char    sin_len;  
sa_family_t     sin_family;  
in_port_t       sin_port;  
struct in_addr  sin_addr;
```

```
struct sockaddr_in6
```

ネットワークを通したマシン間通信，およびローカルなプロセス間通信に使用する，インターネット・ドメイン (AF_INET6 アドレス・ファミリ) のソケットを定義する。

sockaddr_in6 構造体は <netinet/in.h> ファイルに記述されています。sockaddr_in6 構造体には，次の要素が含まれます。

```
uint8_t         sin6_len;  
sa_family_t     sin6_family;  
in_port_t       sin6_port;  
uint32_t        sin6_flowinfo  
struct in6_addr sin6_addr;  
uint32_t        sin6_scope_id
```

in6_addr 構造体には，ネットワーク・バイト順のアドレスを 16 個の 8 ビットの要素の配列として格納されます。

次のデータ構造体を使うと，アプリケーションは sendmsg システム・コールと recvmsg システム・コールを使って，補助データの送信や受信を行うことができます。

```
struct msghdr
```

アプリケーションは，sendmsg および recvmsg の各システム・コールを使用して，システムが保持するオブジェクト (ファイル，デバイス，ソケットなど) に対するアクセス権を引き渡すことができます。sendmsg および recvmsg の各システム・コールについての詳細は，

4.3.6 項を参照してください。データを伝送するプロセスは、UNIX ドメインのソケットに接続しなければなりません。

このデータ構造体 (<sys/socket.h> ヘッダ・ファイルに定義されています) により、AF_INET ソケットと raw AF_INET6 ソケットは、ある種のデータを受け取ることもできます。IP_RECVDSTADDR オプションと IP_RECVOPTS オプション (IPv4 の場合)、IPV6_RECVHOPOPTS オプション、IPV6_RECVDSTOPTS オプション、および IPV6_RECVRTHDR オプション (IPv6 の場合) の説明については、ip(7) を参照してください。

msghdr データ構造体には、次の要素が含まれます。

```
struct msghdr {
    void          *msg_name;          /* optional address */
    size_t        msg_namelen;        /* size of address */
    struct iovec   *msg_iov;           /* scatter/gather array */
    int            msg_iovlen;         /* # elements in msg_iov */
    void          *msg_control;        /* ancillary data, see below */
    size_t        msg_controllen;     /* ancillary data buffer len */
    int            msg_flags;          /* flags on received message */
};
```

XNS4.0 の msghdr データ構造体に加えて、オペレーティング・システムは、4.3BSD、4.4BSD、および POSIX 1003.1g Draft 6.6 の msghdr データ構造体もサポートしています。BSD の msghdr データ構造体については、4.5 節で詳細に説明しています。

struct cmsghdr

sendmsg システム・コールと recvmsg システム・コールで転送される補助データ・オブジェクトの構造体です。msghdr データ構造体の msg_control メンバは、cmsghdr 構造体に含まれる補助データを指しています。

一般的に、データ・オブジェクトは 1 個だけ cmsghdr 構造体で渡されます。ただし、IPv6 Advanced Sockets API を使うと、sendmsg システム・コールと recvmsg システム・コールで複数のオブジェクトを渡せるようになります。raw IPv6 ソケットについては、4.7.2 項を参照してください。このデータ構造体は、<sys/socket.h> ヘッダ・ファイルに定義されています。

cmsghdr データ構造体は、次の構成要素からなっています。

```
struct cmsghdr {
    socklen_t      cmsg_len;           /* #bytes, including this header */
    int            cmsg_level;         /* originating protocol */
    int            cmsg_type;          /* protocol-specific type */
    /* followed by unsigned char cmsg_data[]; */
};
```

```
};
```

4.3 ソケットの使用方法

この節では、ソケットの作成と使用に必要な手順の概略を説明します。次の手順は、コネクション指向型およびコネクションレス型の通信モードについて説明しています。

- ソケットの作成

`socket` および `socketpair` の各システム・コールを使用して、ソケットを作成する方法を説明します。

- 名前とアドレスのバインド

`bind` システム・コールを使用して、名前とアドレスをソケットにバインドする方法を説明します。

- 接続の確立 (クライアント)

クライアントで `connect` システム・コールを使用して、サーバに接続する方法を説明します。

- 接続の受け入れ (サーバ)

`listen` および `accept` の各システム・コールを使用して、サーバをクライアントに接続する方法を説明します。

- ソケット・オプションの設定と取得

`setsockopt` および `getsockopt` の各システム・コールを使用して、ソケット特性の値の設定および取り出しを行う方法を説明します。

- データの転送

`read` と `write` のシステム・コール、および `send` と `recv` 関連のシステム・コールを使用して、データの伝送を行う方法を説明します。

- ソケットのシャットダウン

`shutdown` システム・コールを使用して、ソケットをシャットダウンする方法を説明します。

- ソケットのクローズ

`close` システム・コールを使用して、ソケットをクローズする方法を説明します。

4.3.1 ソケットの作成

ソケットを使用する最初の手順は、ソケットの作成です。ソケットは、`socket` または `socketpair` のシステム・コールを使用して、オープン、つまり作成します。

`socket` システム・コールは、ソケット記述子 `s` を返します。これは、アプリケーション・プログラムが、新しく作成されたソケットを後続のシステム・コールで参照するときに使用する、非負の整数です。返されるソケット記述子は、呼び出したプロセスでそのような記述子に使用できる値のうち、未使用の最小値であり、カーネル記述子テーブルへのインデックスになります。

関数の構文、パラメータ、エラーについては `socket(2)` を参照してください。

たとえば、`AF_INET6` アドレス・ファミリーを使用してインターネット・ドメインでストリーム・ソケットを作成するには、次の呼び出しを使用できます。

```
if ((s = socket(AF_INET, SOCK_STREAM, 0)) == -1) {  
    fprintf(file1, "socket() failed\n");  
    local_flag = FAILED;  
}
```

このように呼び出すと、基礎となる通信が TCP プロトコルをサポートしている場合には、そのプロトコルを使用してストリーム・ソケットが作成されます。UNIX ドメインでデータグラム・ソケットを作成するには、次の呼び出しを使用できます。

```
if ((s = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1) {  
    fprintf(file1, "socket() failed\n");  
    local_flag = FAILED;  
}
```

このように呼び出すと、基礎となる通信が UNIX ドメインのプロトコルをサポートしている場合には、そのプロトコルを使用してデータグラム・ソケットが作成されます。

また、`socketpair` システム・コールも、ソケットの作成に使用できます。`socketpair` システム・コールは、すでに接続されている、名前のない 1 組のソケットを作成します。

`socketpair` システム・コールは、1 組のソケット記述子を返します。これは、アプリケーションが、新しく作成された 1 組のソケットを後続のシステム・コールで参照するときに使用する、非負の整数です。

関数の構文，パラメータ，エラーについては，`socketpair(2)` を参照してください。

次の例は，1 組のソケットを作成する方法を示しています。

```
{
:

    int sv[2];
:

    if ((s = socketpair (AF_UNIX, SOCK_STREAM, 0, sv)) < 0) {
        local_flag=FAILED;
        fprintf(file1, "socketpair() failed\n");
    }
:
}
```

4.3.1.1 実行モードの設定

ソケットは，ブロッキングまたは非ブロッキングのいずれかの I/O モードに設定できます。fcntl オペレーションの `O_NONBLOCK` が，このモードの決定に使用されます。省略時には，`O_NONBLOCK` はクリアされており（セットされない），ソケットはブロッキング・モードになります。ブロッキング・モードでは，ソケットは，`read` を実行しようとしてデータが利用できなかった場合，データが利用できるようになるまで待ちます。

`O_NONBLOCK` がセットされている場合，ソケットは非ブロッキング・モードです。非ブロッキング・モードでは，呼び出したプロセスが `read` を実行しようとしてデータが利用できなかった場合，そのソケットは即座に `EWOULDBLOCK` エラー・コードを指定して戻ります。データが利用できるようになるのを待ちません。同様に，書き込み処理においては，ソケットに `O_NONBLOCK` がセットされていると，出力キューが一杯の場合，ソケットが `write` を実行しようとする時，プロセスは `EWOULDBLOCK` エラー・コードを指定して即座に戻ります。

次の例は，ソケットを非ブロッキングにマークする方法を示しています。

```
#include <fcntl.h>
:
:
int s;
```

```

:
:
if (fcntl(s, F_SETFL, O_NONBLOCK) < 0)
    perror("fcntl F_SETFL, O_NONBLOCK");
    exit(1);
}
:
:

```

ソケットで非ブロッキング I/O を行う場合、プログラムは EWOULDBLOCK エラーをチェックしなければなりません。このエラーはグローバル変数 `errno` に格納されます。オペレーションが通常にブロックしても、実行していたソケットが非ブロッキングと設定されている場合には、EWOULDBLOCK エラーが生じます。次のソケット・システム・コールはすべて、EWOULDBLOCK エラー・コードを返します。

- `accept`
- `connect`
- `send`
- `sendto`
- `sendmsg`
- `recv`
- `recvfrom`
- `recvmsg`
- `read`
- `write`

これらのシステム・コールを非ブロッキングのソケットで使用するプロセスは、EWOULDBLOCK リターン・コードを処理する準備をしておかなければなりません。

`send` などのオペレーションが完了できなくても、一部の書き込みが可能な場合（たとえば、SOCK_STREAM ソケットを使用している場合）には、ただちに送信できるデータが処理され、リターン値は、実際に送信されたデータの量を示します。

4.3.2 名前とアドレスのバインド

`bind` システム・コールはアドレスをソケットに関連付けます。ソケットのドメインは `socket` システム・コールを使用して設定します。`bind` システム・コールを使用すると、それが使用されるドメインに関係なく、ローカル・プロセスは自身の情報、たとえばローカル・ポートやローカル・パス名などを書き込むことができます。この情報によって、サーバ・アプリケーションをクライアント・アプリケーションから呼び出すことができます。

次の例は、`AF_INET` アドレス・ファミリのインターネット・ドメインで作成された `SOCK_STREAM` ソケットで、`bind` システム・コールを使用する方法を示しています。

```
#define PORT 3000

int      retval;          /* General return value */
int      s1_descr;        /* Socket 1 descriptor */
:
:

struct sockaddr_in sockladdr; /* Address struct for socket1.*/
:
:

s1_descr = socket (AF_INET, SOCK_STREAM, 0);
if (s1_descr < 0)          /* Call failed */
:
:

bzero(&sockladdr, sizeof(sockladdr));
sockladdr.sin_family      = AF_INET;
sockladdr.sin_addr.s_addr = INADDR_ANY;
sockladdr.sin_port        = htons(PORT);
retval = bind (s1_descr, &sockladdr, sizeof(sockladdr));
if (retval < 0)           /* Call failed */
:
:
```

関数の構文、パラメータ、エラーについては、`bind(2)` を参照してください。名前とアドレスのバインドの詳細については、4.7.4 項を参照してください。

4.3.3 接続の確立

ソケットは、接続されていない状態で作成されます。クライアント・プロセスは、`connect` システム・コールを使用して、サーバ・プロセスに接続するか、またはサーバのアドレスをローカルに格納します。これは、通信が、コネクション指向型かコネクションレス型かによって異なります。インター

ネット・ドメインでは、一般に、`connect` システム・コールによって、関連付けのローカル・アドレス、ローカル・ポート、外部アドレス、および外部ポートが割り当てられます。

`connect` システム・コールの構文は、通信ドメインによって異なります。

接続に失敗した場合には、エラーが返されます。ただし、システムによって自動的にバインドされた名前はすべて残ります。アプリケーションは、`close` システム・コールを使用して、ソケットおよび記述子の割り当てを解除します。4.6 節の表 4-6 に、ソケットに関連する一般的なエラーをリストします。接続が正常に行われた場合、ソケットはサーバに関連付けられ、データ転送が開始されます。

関数の構文、パラメータ、エラーについては、`connect(2)` を参照してください。

インターネット・ドメインでコネクション指向型プロトコルを選択すると、TCP が選択されます。この場合、`connect` システム・コールは、デスティネーションとの TCP 接続を確立するか、または確立できない場合はエラーを返します。TCP を使用するクライアント・プロセスは、`connect` システム・コールを呼び出して接続を確立しておかなければなりません。これはストリーム・ソケット (SOCK_STREAM) を通してデータ転送の信頼性を高めるためです。

インターネット・ドメインでコネクションレス型プロトコルを選択すると、UDP が選択されます。コネクションレス型プロトコルを使用するクライアント・プロセスは、使用前に接続する必要はありません。このような環境で `connect` を使用すると、デスティネーション (つまりサーバ) のアドレスがローカルに格納されるので、クライアント・プロセスは、メッセージを送信するたびにサーバのアドレスを指定する必要がありません。このソケットで送信されるすべてのデータは、接続されたサーバ・プロセスに自動的にアドレス指定され、そのサーバ・プロセスから受信したデータだけが引き渡されます。

接続するアドレスは、各ソケットに対して常に 1 つしか指定できません。2 回目の `connect` システム・コールでは、デスティネーション・アドレスが変更され、空アドレス (たとえば、AF_INET アドレス INADDR_ANY) への `connect` システム・コールを呼び出すと、切断されます。コネクションレス型プロトコルにおける `connect` システム・コールは、ただちに戻ります。これは、オペレーティング・システムがサーバのソケットのアド

レスを記録するだけだからです。コネクション指向型プロトコルの場合、接続要求は端末相互接続の確立を開始するので、コネクションレス型プロトコルはこの点で異なります。

コネクションレス型プロトコルを使用するソケットが接続されると、最後に呼び出した `send` システム・コールからのエラーが、同期をはずれて戻ることがあります。このようなエラーは、後でそのソケットに対して続いて行うオペレーションで報告されます。特殊ソケット・オプション `SO_ERROR` (`getsockopt` システム・コールで使用する) を使用して、エラー状態を照会することもできます。次のデータをいつ送受信できるかを判断するために、`select` システム・コールを発行すると、このシステム・コールは、プロセスがエラー表示を受信した場合、真の値を返します。

どんな場合でも、次のオペレーションはエラーを返し、エラー状態をクリアします。

関数の構文、パラメータ、エラーについては、`select(2)` を参照してください。

次は、`select` システム・コールの例です。

```
if ( (ret_val = select(20,&read_mask,NULL,NULL,&tp)) != i )
```

4.3.4 接続の受け入れ

コネクション指向型のサーバ・プロセスは、通常、周知のアドレスでサービス要求をリッスンします。つまり、サーバ・プロセスは、クライアントをサーバ・アドレスに接続することによって接続が要求されるまで、休止状態のままです。接続が要求されると、サーバ・プロセスはウェイクアップし、クライアントが要求するアクションを実行してクライアントに対してサービスを行います。

コネクション指向型サーバは、`listen` および `accept` のシステム・コールを使用して、クライアント・プロセスとの接続に対して準備し、接続を受け入れます。

`listen` システム・コールは、通常、`socket` および `bind` のシステム・コールの後で呼び出されます。このシステム・コールは、サーバがクライアントから接続要求を受信する準備ができていることを示します。

関数の構文、パラメータ、エラーについては、`listen(2)` を参照してください。

サーバは、`accept` システム・コールを使用することによって、クライアントとの接続を受け入れます。

`accept` システム・コールは、クライアントがサービスを要求するまで、サーバをブロックします。このシステム・コールは、呼び出し中に `SIGCHLD` などのシグナルによって割り込まれると、障害状態を返します。したがって、`accept` からのリターン値は、接続が確立されたことを保証するためにチェックされます。

関数の構文、パラメータ、エラーについては、`accept(2)` を参照してください。

接続が確立されると、サーバは通常、子プロセスをフォークします。これにより、リッスンしているソケットと同じプロパティを持つ、別のソケットが作成されます。次の例は、親プロセスが接続要求をキューにいれるために使用するソケット `s` が子プロセスでクローズされ、`accept` システム・コールで作成されたソケット `g` が親プロセスでクローズされる方法を示しています。クライアントのアドレスも、`doit` ルーチンに渡されます。これは、クライアントを認証するために必要だからです。`accept` システム・コールが新しいソケットを作成した後は、元のソケット上でリッスンを継続しながら、新しいソケットで、クライアントの接続要求にサービスできます。次に例を示します。

```
for (;;) {
    int g, len = sizeof (from);

    g = accept(s, (struct sockaddr *)&from, &len);
    if (g < 0) {
        if (errno != EINTR)
            syslog(LOG_ERR, "rlogind: accept: %m");
        continue;
    }
    if (fork() == 0) {    /* Child */
        close(s);
        doit(g, &from);
    }
    close(g);            /* Parent */
}
```

コネクションレス型サーバは、`bind` システム・コールを使用しますが、このサーバは、`accept` システム・コールではなく、`recvfrom` システム・コールを使用するため、クライアントの要求を待つことになります。デー

タ交換中には、コネクションレス型サーバとクライアントの間に接続は確立されません。

4.3.5 ソケット・オプションの設定と取得

ローカル・アドレスへのソケットのバインドやデスティネーション・アドレスへのソケットの接続の他に、アプリケーション・プログラムは、ソケットの制御ができなければなりません。たとえば、タイム・アウトおよび再伝送を使用するプロトコルの場合には、アプリケーション・プログラムで、タイム・アウト・パラメータを取得または設定しなければならないことがあります。また、バッファ・スペースの割り当ての制御、ソケットでのブロードキャストの伝送を可能にするかどうかの決定、あるいは帯域外データの処理の制御が必要な場合もあります。

`getsockopt` および `setsockopt` のシステム・コールによって、アプリケーション・プログラムでソケットのオペレーションを制御することができます。 `setsockopt` システム・コールを使用すると、アプリケーション・プログラムは、ソケット・オプションを設定することができます。このとき使用する値のセットは、`getsockopt` システム・コールで取得する値のセットと同じです。

関数の構文、パラメータ、エラーについては、`setsockopt(2)` を参照してください。

次の例は、インターネット通信ドメインにおいて、ソケットに `SO_SNDBUF` オプションを設定する方法を示しています。

```
# include      <sys/socket.h>
:

int      retval;          /* General return value. */
int      s1_descr;        /* Socket 1 descriptor  */
int      sockbufsize=16384;
:

retval = setsockopt (s1_descr, SOL_SOCKET, SO_SNDBUF, (void *)
                    &sockbufsize, sizeof(sockbufsize));
```

`getsockopt` システム・コールを使用すると、アプリケーション・プログラムは、`setsockopt` システム・コールで設定されたソケット・オプションに関する情報を要求できます。

関数の構文，パラメータ，エラーについては，`getsockopt(2)` を参照してください。

次の例は，`getsockopt` システム・コールを使用して，既存のソケット上の `SO_SNDBUF` のサイズを判断する方法を示しています。

```
#include <sys/socket.h>
:
:
int      retval;                /* General return value. */
int      s1_descr;              /* Socket 1 descriptor  */
int      sbufsize;
int      len = sizeof(sbufsize);
:
:
retval = getsockopt (s1_descr, SOL_SOCKET, SO_SNDBUF,
                    (void *)&sbufsize, &len);
```

`SOL_SOCKET` パラメータは，汎用ソケット・レベル・コードで `SO_SNDBUF` パラメータが解釈されることを示します。 `SO_SNDBUF` パラメータは，ソケットで使用されている送信ソケット・バッファのサイズを示します。

すべてのソケット・オプションがすべてのソケットに適用されるわけではありません。 設定できるオプションは，ソケットで使用しているアドレス・ファミリやプロトコルによって異なります。

4.3.6 データの転送

ソケット層で行われる作業の大半は，データの送受信です。 ソケット層自体は，ソケットを通して送受信されるデータに対して，いかなる構造体の使用も義務付けていません。 すべてのデータの解釈または構築は，通信ドメインのインプリメンテーションとは論理的に分離されています。

アプリケーションがデータの送受信に使用するシステム・コールは，次のとおりです。

- `read`
- `write`
- `send`
- `sendto`
- `recv`

- `recvfrom`
- `sendmsg`
- `recvmsg`

4.3.6.1 `read` システム・コールの使用

`read` システム・コールによって、送信側のアドレスを受信せずにプロセスがソケットでデータを受信できます。

関数の構文、パラメータ、エラーについては、`read(2)` を参照してください。

4.3.6.2 `write` システム・コールの使用

`write` システム・コールは、接続状態のソケットで使います。`write` システム・コールを使用して転送されるデータのデスティネーションは、接続によって暗黙的に指定されます。

関数の構文、パラメータ、エラーについては、`write(2)` を参照してください。

4.3.6.3 `send`, `sendto`, `recv` および `recvfrom` システム・コールの使用

`send`, `sendto`, `recv`, および `recvfrom` の各システム・コールは、`read` および `write` のシステム・コールとよく似ています。いずれも、最初の 3 つのパラメータは同じですが、`send`, `sendto`, `recv`, および `recvfrom` の各システム・コールには、追加のフラグが必要となります。これらのフラグは `<sys/socket.h>` ヘッダ・ファイルに定義されており、アプリケーション・プログラムで次のうちの 1 つ以上のことを行う必要がある場合に、非ゼロの値として定義できます。

フラグ	説明
<code>MSG_OOB</code>	帯域外データを送受信する。
<code>MSG_PEEK</code>	読み取りを行わずにデータの中身を見る。 <code>recv</code> および <code>recvfrom</code> の各システム・コールで有効。
<code>MSG_DONTROUTE</code>	パケットのルーティングを行わずにデータを送信する。 <code>send</code> および <code>sendto</code> の各システム・コールで有効。

`MSG_OOB` フラグは、帯域外データまたは緊急データを示し、ストリーム・ソケット (`SOCK_STREAM`) 固有のフラグです。帯域外データについての詳細は、4.7.5 項を参照してください。

MSG_PEEK フラグを使用すると、アプリケーションは、読み取りに利用できるデータをプレビューでき、`recv` または `recvfrom` システム・コールが戻った後も、システムによるデータ破棄は行われません。MSG_PEEK フラグは、`recv` システム・コールに指定されると、現在あるすべてのデータがユーザに返されますが、読み取られていないデータとみなされます。したがって、そのソケットに対する次の `read` または `recv` システム・コールは、以前にプレビューされたデータを返します。

MSG_DONTROUTE フラグを使用するのは、現在のところ、ルーティング・テーブル管理プロセスだけなので、詳しくは説明しません。

send

`send` システム・コールは、接続状態のソケットで使用されます。`send` システム・コールと `write` システム・コールの機能は、ほとんど同じです。唯一の相違点は、`send` では、この項の最初で説明したフラグがサポートされるという点です。

関数の構文、パラメータ、エラーについては、`send(2)` を参照してください。

sendto

`sendto` システム・コールは、接続されたソケット、または接続されていないソケットで使用します。これによって、プロセスは、メッセージのデスティネーションを明示的に指定できます。

関数の構文、パラメータ、エラーについては、`sendto(2)` を参照してください。

recv

`recv` システム・コールでは、プロセスは、送信側のアドレスを受信せずにソケットでデータを受信できます。`read` システム・コールと `recv` システム・コールの機能は、ほとんど同じです。唯一の相違点は、`recv` では、この項の最初で説明したフラグがサポートされるという点です。

関数の構文、パラメータ、エラーについては、`recv(2)` を参照してください。

recvfrom

`recvfrom` システム・コールは、接続されたソケット、または接続されていないソケットで使用できます。`recvfrom` システム・コールは、`recv`

システム・コールと機能が似ています。ただし、`recvfrom` システム・コールでは、アプリケーションは、通信している対等プロセスのアドレスを受信できます。

関数の構文、パラメータ、エラーについては、`recvfrom(2)` を参照してください。

4.3.6.4 `sendmsg` および `recvmsg` システム・コールの使用

`sendmsg` および `recvmsg` システム・コールは、ローカル・マシン上の関連のないプロセス間で、ファイル記述子を相互に引き渡すことができるようにします。この点で、他の送信または受信に関連するシステム・コールとは異なります。この 2 つのシステム・コールは、アクセス権の概念をサポートする唯一のシステム・コールです。つまり、プロセスは、システムが保持するオブジェクトにアクセスする権利を、システムによって与えられます。`sendmsg` および `recvmsg` のシステム・コールを使用すると、このアクセス権を別のプロセスに引き渡すことができます。

`sendmsg` および `recvmsg` のシステム・コールは、アクセス権の引き渡しに `msghdr` データ構造体を使用します。`msghdr` データ構造体は、`msg_control` および `msg_controllen` の 2 つのパラメータを定義します。この 2 つのパラメータが、プロセス間でのアクセス権の引き渡しおよび受け取りを処理します。`msghdr` データ構造体についての詳細は、4.2.3.4 項および 4.5.2 項を参照してください。

`sendmsg` および `recvmsg` のシステム・コールは、コネクション指向型またはコネクションレス型のプロトコルを使用し、インターネット・ドメインまたは UNIX ドメインのいずれかで利用できますが、プロセスが記述子を引き渡すためには、そのプロセスは、UNIX ドメイン・ソケットと接続していなければなりません。

sendmsg

`sendmsg` システム・コールは、接続されたソケット、または接続されていないソケットで使用されます。このシステム・コールは、`msghdr` データ構造体を使用して、データを転送します。`msghdr` データ構造体についての詳細は、4.2.3.4 項および 4.5.2 項を参照してください。

関数の構文、パラメータ、エラーについては、`sendmsg(2)` を参照してください。

次は，sendmsg システム・コールの例です。

```
struct msghdr send;
struct iovec saiov;
struct sockaddr destAddress;
char sendbuf[BUFSIZE];
.
.
.
send.msg_name = (void *)&destAddress;
send.msg_namelen = sizeof(destAddress);
send.msg_iov = &saiov;
send.msg_iovlen = 1;
saiov.iov_base = sendbuf;
saiov.iov_len = sizeof(sendbuf);
send.msg_control = NULL;
send.msg_controllen = 0;
send.msg_flags = 0;
if ((i = sendmsg(s, &send, 0)) < 0) {
    fprintf(file1, "sendmsg() failed\n");
    exit(1);
}
```

recvmsg

recvmsg システム・コールは，接続されたソケット，または接続されていないソケットで使用されます。このシステム・コールは，msghdr データ構造体を使用してデータを転送します。msghdr データ構造体についての詳細は，4.2.3.4 項および 4.5.2 項を参照してください。

関数の構文，パラメータ，エラーについては，recvmsg(2) を参照してください。

次は，recvmsg システム・コールの例です。

```
struct msghdr recv;
struct iovec rcviov;
struct sockaddr_in rcvaddress;
char rcvbuf[BUFSIZE];
.
.
.
recv.msg_name = (void *) &rcvaddress;
recv.msg_namelen = sizeof(rcvaddress);
recv.msg_iov = &rcviov;
recv.msg_iovlen = 1;
rcviov.iov_base = rcvbuf;
rcviov.iov_len = sizeof(rcvbuf);
recv.msg_control = NULL;
```

```

recv.msg_controllen = 0
recv.msg_flags = 0
if ((i = recvmsg(r, &recv, 0)) < 0) {
    fprintf(file1, "recvmsg() failed\n");
    exit(1);
}
.
.
.

```

4.3.7 ソケットのシャットダウン

アプリケーション・プログラムは、すべての保留中のデータが必要でない場合には、クローズする前に、ソケットに対して `shutdown` システム・コールを使用できます。関数の構文、パラメータ、エラーについては、`shutdown(2)` を参照してください。

4.3.8 ソケットのクローズ

`close` システム・コールは、ソケットのクローズに使用されます。関数の構文、パラメータ、エラーについては、`close(2)` を参照してください。

ソケットのクローズ、およびそのリソースの再生は、複雑になる場合があります。たとえば、`close` システム・コールは、プロセスが終了するときには、決して失敗しないと思われています。しかし、信頼性の高いデータの引き渡しを保証しているソケットをクローズするときに、伝送するためにキューにいれられているデータがまだある場合、または受信の肯定応答を待っているデータがある場合には、ソケットはデータの伝送を行わなければなりません。ソケットがキューに入っているデータを破棄して、`close` システム・コールが正常終了できるようにすると、信頼性の高いデータの引き渡しの保証に違反することになります。データを破棄すると、`close` システム・コールの暗黙的な意味に依存するプロセス動作の信頼性が、ネットワーク環境において低くなります。

ただし、すべてのデータを正常に伝送するまでソケットがブロックすると、`close` システム・コールが終了できなくなる通信ドメインもあります。

ソケット層は、問題の解決を図りながら、`close` システム・コールの意味を維持しようとします。通常の実行では、ソケットをクローズすると、キューにいれられたが受け入れられていない接続はすべて破棄されます。ソケットが接続状態にあるときは、切断が開始されます。ソケットは、記述子がもう参照していないことを示すようにマークされ、`close` オペレー

ションは正常に戻ります。切断要求が終了すると、ネットワーク・サポートはソケット層へ通知し、ソケット・リソースは再生されます。ネットワーク層は、ソケットの送信バッファに入っているすべてのデータの伝送を試みます。しかし、伝送が正常終了する保証はされません。

一方、ソケットを明示的にマークして、クローズ時に、保留中のデータがフラッシュされて接続がシャットダウンされるまで、アプリケーション・プログラムで強制的にリングする (残す) ことができます。このオプションは、`setsockopt` システム・コールを `SO_LINGER` オプションとともに使用することによって、ソケット・データ構造体内にマークされます。

注意

リング・オプションを使用する `setsockopt` システム・コールは、`<sys/socket.h>` ヘッダ・ファイルに定義されている `linger` 構造体を使用します。

アプリケーション・プログラムでソケットをリングすることを指定する場合には、リングする期間も指定します。切断が完了する前にリング期間が満了すると、ソケット層は、すべての保留中のデータを破棄して、強制的にソケットをシャットダウンします。

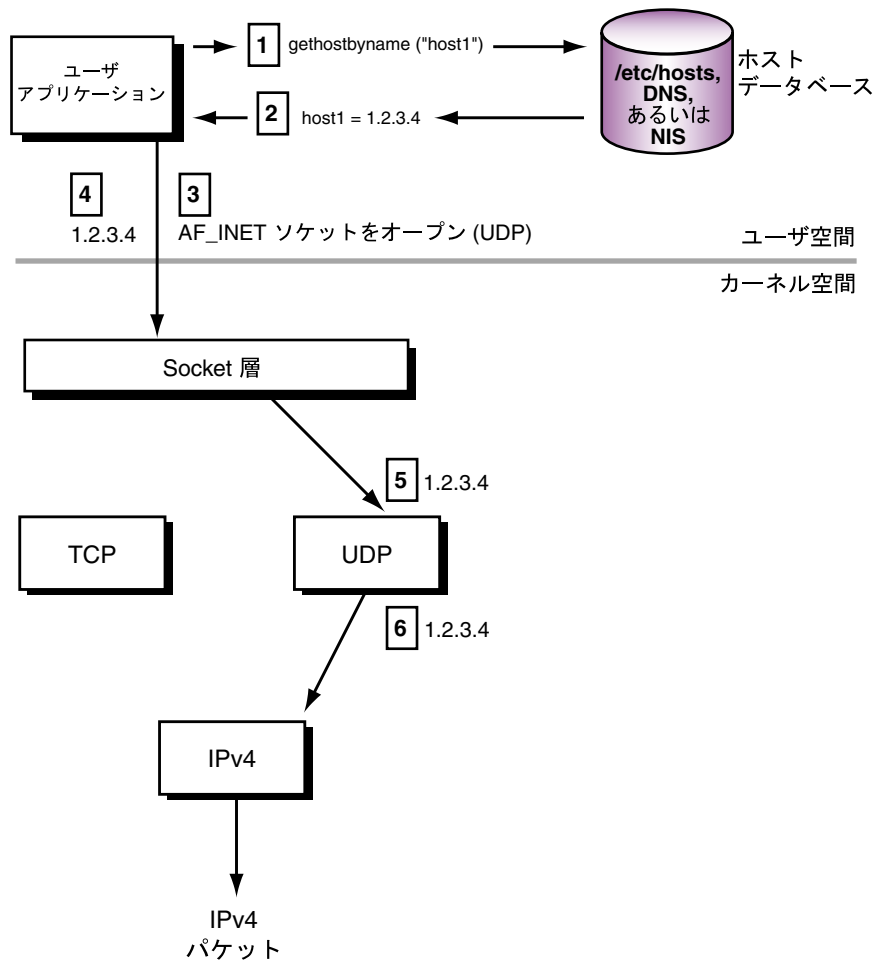
4.4 インターネット・アプリケーションの作成

この節では、4.3 節の情報を基にしてインターネット・ドメイン (IPv4 および IPv6) のアプリケーションを作成して使用するために必要な手順の概要について説明します。またこの節では、アプリケーションが使用する可能性のあるアドレス・テスト・マクロについて概要を示します。IPv6 アプリケーションを開発するには、ここで説明する情報と、4.7.1 項の移植ガイドラインを使用してください。

4.4.1 IPv4 アプリケーションの作成

現在のインターネット・アプリケーションは、IPv4 通信に `AF_INET` ソケットを使用します。図 4-2 では、`AF_INET` ソケットを使用して IPv4 パケットを送信するクライアント・アプリケーションのイベントのシーケンス例を示しています。

図 4-2: IPv4 通信での AF_INET ソケットの使用



ZK-1558U-AIJ

- ❶ アプリケーションは `gethostbyname` を呼び出し、ホスト名 `host1` を渡す。
- ❷ 検索によって `hosts` データベースから `host1` を発見し、`gethostbyname` は IPv4 アドレス `1.2.3.4` を、`hostent` 型の構造体に返す。
- ❸ アプリケーションは `socket` を呼び出し、`AF_INET` ソケットを作成する。この例ではこのソケットはデータグラム・ソケット (UDP) だが、ストリーム・ソケット (TCP) とすることもできる。
- ❹ `socket` 呼び出しが成功すると、アプリケーションは `sockaddr_in` 構造体を設定して `connect` を呼び出し、`host1` との接続を確認する。`connect` 呼び出しが成功すると、アプリケーションは `send` を呼び出して、情報をアドレス `1.2.3.4` に送信する。
- ❺ ソケット層は、情報とアドレスを UDP モジュールに渡す。
- ❻ UDP モジュールはアドレス `1.2.3.4` をパケット・ヘッダにいれ、情報を IPv4 に渡し伝送する。

これ以降、アプリケーションは次の処理を実行できます。

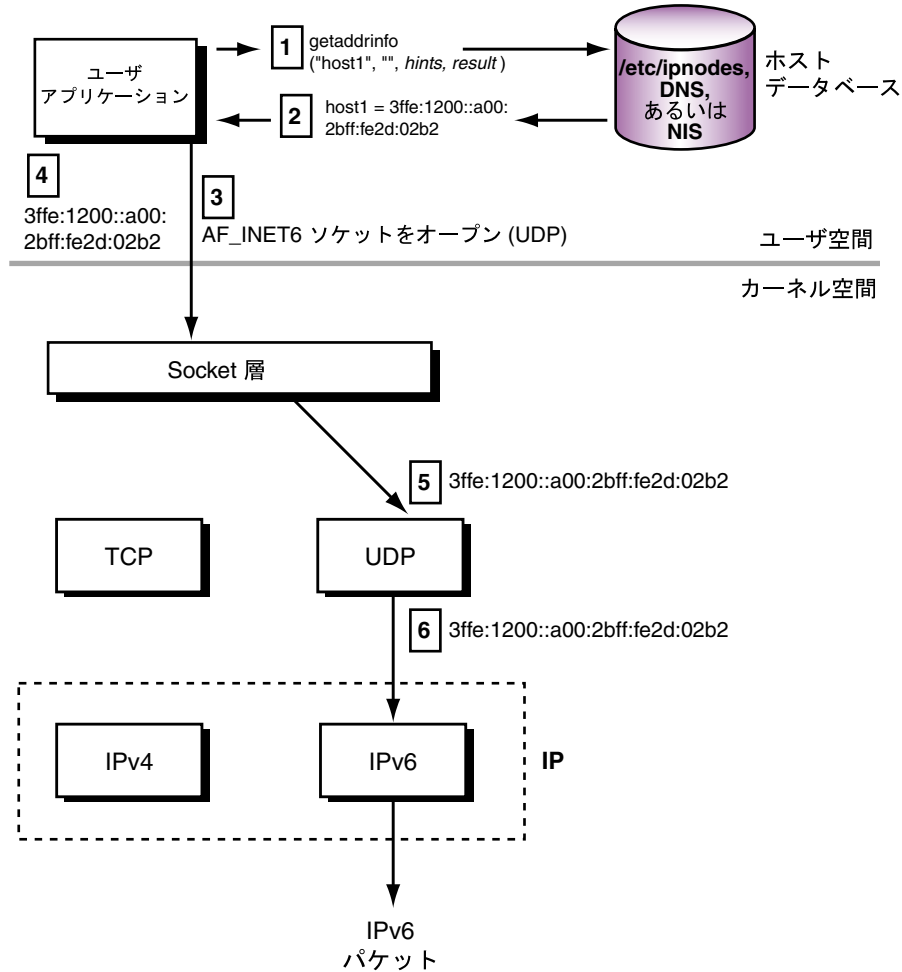
1. `recv` を呼び出し、サーバ・システムからの応答を待つ。
2. 応答を受信した後、`gethostbyaddr` を呼び出し、`sockaddr_in` 構造体でサーバのアドレスを渡す。検索によりそのアドレスがホスト・データベース内で見つかり、`gethostbyaddr` は、`hostent` 型の構造体でホスト名を返す。
3. `inet_ntoa` を呼び出して、サーバ・アドレスをテキスト文字列に変換する。

C.1.1 項には、これらの手順を示す、クライアント・プログラム・コード例があります。

4.4.2 IPv6 アプリケーションの作成

インターネット・アプリケーションは、IPv6 通信に `AF_INET6` ソケットを使用できます。さらに `AF_INT6` ソケットでは、IPv4 通信もできます。図 4-3 では、`AF_INET6` ソケットを使用して IPv6 パケットを送信するクライアント・アプリケーションのイベントのシーケンスを示しています。

図 4-3: IPv6 通信での AF_INET6 ソケットの使用



ZK-1556U-AIJ

- ① アプリケーションは `getaddrinfo` を呼び出し、ホスト名 (host1), AF_INET6 アドレス・ファミリ・ヒント, AI_ADDRCONFIG フラグ・ヒント, AI_V4MAPPED フラグ・ヒントを渡す。このフラグ・ヒントは、host1 の IPv6 アドレスが見つかった場合は、それを返すように関数に指示する。ヒントのフィールドと値についての詳細は、`getaddrinfo(3)` を参照してください。
- ② 検索によって hosts データベースで host1 の IPv6 アドレスを発見し、`getaddrinfo` は IPv6 アドレス `3ffe:1200::a00:2bff:fe2d:02b2` を、1 つ以上の `addrinfo` 型の構造体に返す。

- ③ アプリケーションは `socket` を呼び出し、`addrinfo` 構造体内のアドレス・ファミリおよびソケット・タイプを使用して、`AF_INET6` ソケットを作成する。この例ではこのソケットはデータグラム・ソケット (UDP) だが、ストリーム・ソケット (TCP) とすることもできる。
- ④ `socket` 呼び出しが成功すると、アプリケーションは `addrinfo` 構造体内のホスト・アドレスと長さを使用して `connect` を呼び出し、`host1` とのコネクションを確立する。`connect` 呼び出しが成功すると、アプリケーションは情報をアドレス `3ffe:1200::a00:2bff:fe2d:02b2` に送信する。

注意

`addrinfo` 構造体の情報を使用した後、アプリケーションは `freeaddrinfo` を呼び出して、この構造体に使用されていたシステム・リソースを解放します。

- ⑤ ソケット層は、情報とアドレスを UDP モジュールに渡す。
- ⑥ UDP モジュールは IPv6 アドレスを識別し、アドレス `3ffe:1200::a00:2bff:fe2d:02b2` をパケット・ヘッダにいれ、情報を IPv6 モジュールに渡し伝送する。

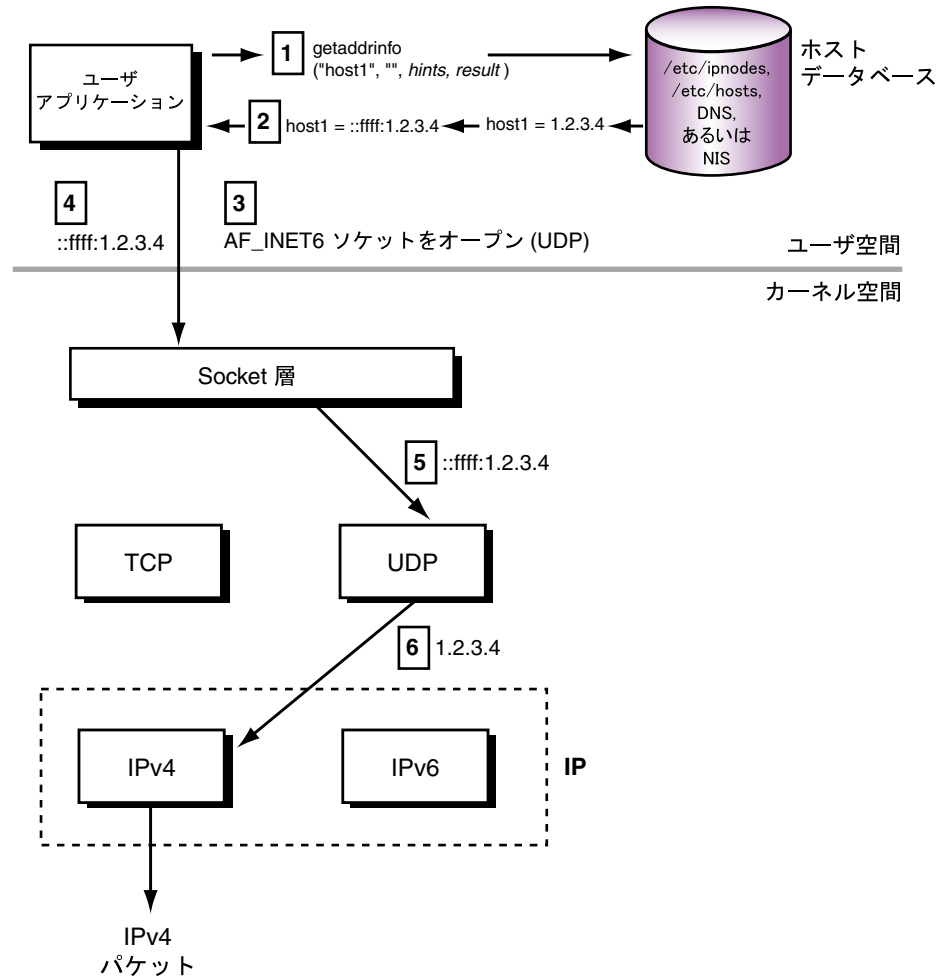
これ以降、アプリケーションは次の処理を実行できます。

1. `recv` を呼び出し、サーバ・システムからの応答を待つ。
2. 応答を受信した後、`getpeername` を呼び出し、接続されたソケットのアドレスを調べる。このアドレスは、`sockaddr_in6` 型の構造体で返される。アプリケーションがプロトコルに依存しないようにしたい場合は、`sockaddr_in6` 構造体の代わりに `sockaddr_storage` 構造体を使用することもできる。
3. `NI_NAMEREQD` フラグを指定して `getnameinfo` を呼び出し、サーバ名を取得する。
4. `NI_NUMERICHOST` フラグを指定して `getnameinfo` を呼び出し、サーバ・アドレスをテキスト文字列に変換する。

C.2.1 項 には、これらの手順を示す、クライアント・プログラム・コードの例があります。

AF_INET6 ソケットは、IPv4 通信に使用することもできます。図 4-4 では、AF_INET6 ソケットを使用して IPv4 パケットを送信するクライアント・アプリケーションのイベントのシーケンスを示しています。

図 4-4: IPv4 通信での AF_INET6 ソケットの使用 (送信)



ZK-1557U-AIJ

- ① アプリケーションは `getaddrinfo` を呼び出し、ホスト名 (`host1`)、`AF_INET6` アドレス・ファミリ・ヒント、`AI_ADDRCONFIG` フラグ・ヒント、`AI_V4MAPPED` フラグ・ヒントを渡す。このフラグ・ヒントは、`host1` の IPv4 アドレスが見つかった場合は、それを IPv4 にマップされた IPv6 アドレスとして返すように関数に指示する。ヒントのフィールドと値についての詳細は、`getaddrinfo(3)` を参照してください。
- ② 検索によって `hosts` データベースで `host1` の IPv4 アドレス `1.2.3.4` を発見し、`getaddrinfo` は IPv4 にマップされた IPv6 アドレス `::ffff:1.2.3.4` を、1 つ以上の `addrinfo` 型の構造体に返す。
- ③ アプリケーションは `socket` を呼び出し、`addrinfo` 構造体内のアドレス・ファミリおよびソケット・タイプを使用して、`AF_INET6` ソケットを作成する。この例ではこのソケットはデータグラム・ソケット (UDP) だが、ストリーム・ソケット (TCP) とすることもできる。
- ④ `socket` 呼び出しが成功すると、アプリケーションは `addrinfo` 構造体内のホスト・アドレスと長さを使用して `connect` を呼び出し、`host1` とのコネクションを確立する。`connect` 呼び出しが成功すると、アプリケーションは情報をアドレス `::ffff:1.2.3.4` に送信する。

注意

`addrinfo` 構造体の情報を使用した後、アプリケーションは `freeaddrinfo` を呼び出して、この構造体に使用されていたシステム・リソースを解放します。

- ⑤ ソケット層は、情報とアドレスを UDP モジュールに渡す。
- ⑥ UDP モジュールは IPv4 にマップされた IPv6 アドレスを識別し、アドレス `1.2.3.4` をパケット・ヘッダにいれ、情報を IPv4 モジュールに渡し伝送する。

これ以降、アプリケーションは次の処理を実行できます。

1. `recv` を呼び出し、サーバ・システムからの応答を待つ。
2. 応答を受信した後、`getpeername` を呼び出し、接続されたソケットのアドレスを調べる。このアドレスは、`sockaddr_in6` 型の構造体で返される。アプリケーションがプロトコルに依存しないようにしたい

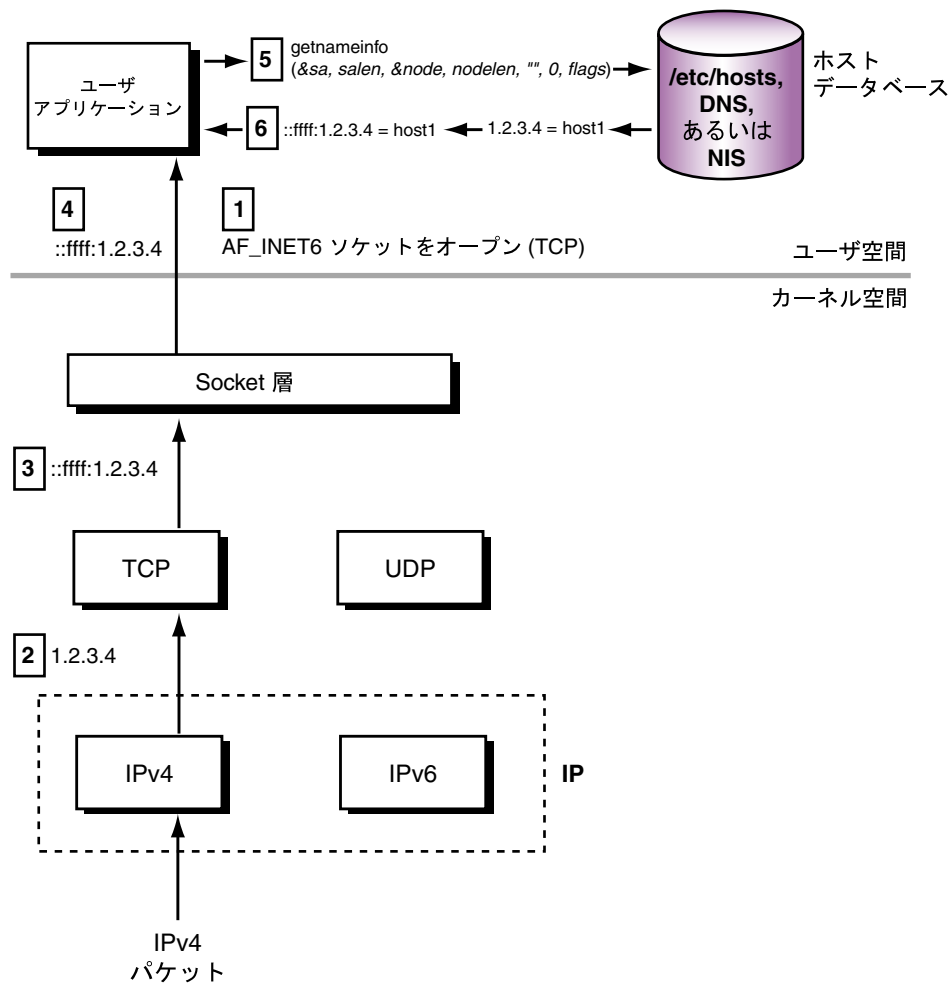
場合は、`sockaddr_in6` 構造体の代わりに `sockaddr_storage` 構造体を使用することもできる。

3. `NI_NAMEREQD` フラグを指定して `getnameinfo` を呼び出し、サーバ名を取得する。
4. `NI_NUMERICHOST` フラグを指定して `getnameinfo` を呼び出し、サーバ・アドレスをテキスト文字列に変換する。

C.2.1 項 には、これらの手順を示す、クライアント・プログラム・コードの例があります。

`AF_INET6` ソケットは、IPv4 アドレスまたは IPv6 アドレスのどちらに送られたメッセージも受信できます。`AF_INET6` ソケットは、IPv4 アドレスを表すために IPv4 にマップされた IPv6 アドレス・フォーマットを使用します。図 4-5 では、`AF_INET6` ソケットを使用して IPv4 パケットを受信するサーバ・アプリケーションのイベントのシーケンスを示しています。

図 4-5: IPv4 通信での AF_INET6 ソケットの使用 (受信)



ZK-1770U-AIJ

- ❶ アプリケーションは次の処理を行う。
 1. `socket` を呼び出して、`AF_INET6` ソケットを作成する。
 2. `sockaddr_in6` 構造体を初期化し、ファミリ、アドレス、ポートを設定する。
 3. `bind` を呼び出して、ソケットにアドレスを割り当てる。
 4. `listen` を呼び出して、ソケットをコネクション受け付け状態にする。
- ❷ IPv4 パケットが到着し、IPv4 モジュールを通過する。
- ❸ TCP レイヤでは、パケット・ヘッダを削除し、その情報と IPv4 にマップされた IPv6 アドレス (`::ffff:1.2.3.4`) をソケット・レイヤへ渡す。
- ❹ アプリケーションは `accept` を呼び出し、ソケットから情報を取り出す。ソケットからの情報は、`sockaddr_storage` 構造体でアプリケーションに渡される。これにより、アプリケーションがプロトコルに依存しなくなる。
- ❺ アプリケーションは `getnameinfo` を呼び出し、アドレス `::ffff:1.2.3.4` と `NI_NAMEREQD` フラグを渡す。このフラグは、アドレスに対応するホスト名を返すように関数に指示する。フラグ・ビットとその意味についての詳細は、`getnameinfo(3)` を参照してください。
- ❻ 検索の結果、アドレス `1.2.3.4` のホスト名が `hosts` データベースに見つかり、`getnameinfo` はホスト名を返す。

C.2.2 項 には、これらの手順を示す、サーバ・プログラム・コードの例があります。

4.4.3 アドレス・テスト・マクロ

`AF_INET6` ソケットを通信に使用するアプリケーションは、場合によっては、構造体に返されるアドレスのタイプを調べる必要があります。このような場合のために、API にはアドレスをテストするマクロが定義されています。表 4-5 に、現在定義されているアドレス・テスト・マクロと、有効なテストで返される値をリストします。これらのマクロを使用するには、アプリケーション内で次のファイルをインクルードします。

```
#include <netinet/in.h>
```

表 4-5: アドレス・テスト・マクロの要約

マクロ	返却値
IN6_IS_ADDR_UNSPECIFIED	指定されたタイプの場合, True。
IN6_IS_ADDR_LOOPBACK	指定されたタイプの場合, True。
IN6_IS_ADDR_MULTICAST	指定されたタイプの場合, True。
IN6_IS_ADDR_LINKLOCAL	指定されたタイプの場合, True。
IN6_IS_ADDR_SITELOCAL	指定されたタイプの場合, True。
IN6_IS_ADDR_V4MAPPED	指定されたタイプの場合, True。
IN6_IS_ADDR_V4COMPAT	指定されたタイプの場合, True。
IN6_IS_ADDR_MC_NODELOCAL	指定されたスコープの場合, True。
IN6_IS_ADDR_MC_LINKLOCAL	指定されたスコープの場合, True。
IN6_IS_ADDR_MC_SITELOCAL	指定されたスコープの場合, True。
IN6_IS_ADDR_MC_ORGLOCAL	指定されたスコープの場合, True。
IN6_IS_ADDR_MC_GLOBAL	指定されたスコープの場合, True。
IN6_ARE_ADDR_EQUAL	アドレスが等しい場合, True。

4.5 BSD ソケット・インタフェース

XNS4.0 ソケット・インタフェースに加えて, オペレーティング・システムは, 4.3BSD, 4.4BSD および POSIX 1003.1g Draft 6.6 のソケット・インタフェースもサポートしています。4.4BSD ソケット・インタフェースは, 4.3BSD ソケットに多数の変更を加えています。4.3BSD と 4.4BSD のソケット・インタフェースの間の変更のほとんどは, ソケット・フレームワークのもとで, 国際標準化機構 (ISO) プロトコル群のインプリメンテーションを容易にすることを目的としています。XNS4.0 ソケット・インタフェースは, 標準のソケット・インタフェースを提供しています

注意

4.4BSD が利用できても, そのサイトが ISO プロトコルをサポートするということではありません。それぞれのサイトの適切な担当者に問い合わせてください。

4.4BSD ソケット・インタフェースを使用するには、プログラムまたはメイクファイルに次の行を追加しなければなりません。

```
#define _SOCKADDR_LEN
```

4.4BSD ソケット・インタフェースには、アプリケーション・プログラム用に、4.3BSD インタフェースから次の変更が加えられています。

- 可変長 (長い) のネットワーク・アドレスをサポートするための `sockaddr` 構造体
- データとともにプロトコル情報および状態の受信を可能にする `msghdr` 構造体

以降の項で、これらの機能について説明します。

4.5.1 可変長ネットワーク・アドレス

4.4BSD の `sockaddr` 構造体は、可変長のネットワーク・アドレスをサポートしています。構造体に長さのフィールドが加わり、次のように定義されています。

```
/* 4.4BSD sockaddr Structure */

struct sockaddr {
    u_char sa_len;          /* total length */
    u_char sa_family;       /* address family */
    char    sa_data[14];    /* actually longer; address value */
};
```

4.3BSD の `sockaddr` 構造体には、次のフィールドがあります。

```
u_short  sa_family;
char      sa_data[14];
```

図 4-6 では、4.4BSD と 4.3BSD の `sockaddr` 構造体を比較しています。

図 4-6: 4.3BSD と 4.4BSD の `sockaddr` 構造体



ZK0526UJR

4.5.2 プロトコル・データとユーザ・データの受信

4.3BSD バージョンの `msghdr` 構造体 (`cc` コマンドを使用するときの省略時の設定) は, `sendmsg` および `recvmsg` の各システム・コールのオプション機能を使用するときに必要なパラメータを提供します。

4.3BSD の `msghdr` 構造体は, 次のとおりです。

```
/* 4.3BSD msghdr Structure */
struct msghdr {
    caddr_t msg_name;           /* optional address */
    int      msg_namelen;       /* size of address */
    struct iovec *msg_iov;      /* scatter/gather array */
    int      msg_iovlen;       /* # elements in msg_iov */
    caddr_t msg_accrights;      /* access rights sent/re-
                                /* ceived */
    int      msg_accrightslen;

};
```

パラメータ `msg_name` および `msg_namelen` は, ソケットが接続されていないときに使用します。パラメータ `msg_iov` および `msg_iovlen` は, 分散 (読み取り) および収集 (書き込み) オペレーションで使用します。前述したように, パラメータ `msg_accrights` および `msg_accrightslen` によって, 送信プロセスは, 受信プロセスにアクセス権を引き渡すことができます。

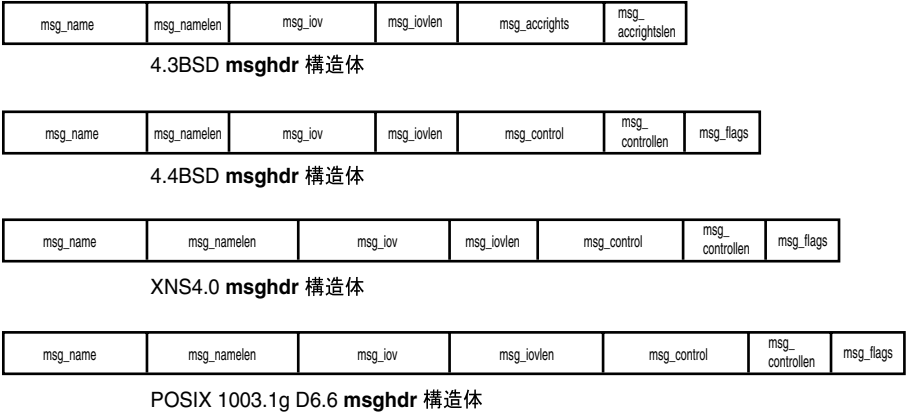
4.4BSD 構造体には追加フィールドがあり, これによって, アプリケーション・プログラムは, メッセージ内にユーザ・データとともにプロトコル情報を含めることができます。

ユーザ・データとともにプロトコル・データを受信できるようにするために、オペレーティング・システムは、4.4BSD ソケット・インタフェースからの msghdr 構造体を提供しています。構造体には、制御データへのポインタ、制御データの長さを表す長さフィールド、およびフラグ・フィールドが、次のように追加されています。

```
/* 4.4BSD msghdr Structure */
struct msghdr {
    caddr_t msg_name;           /* optional address */
    u_int  msg_namelen;        /* size of address */
    struct iovec *msg_iov;      /* scatter/gather array */
    u_int  msg_iovlen;         /* # elements in msg_iov */
    caddr_t msg_control;        /* ancillary data, see below */
    u_int  msg_controllen;     /* ancillary data buffer len */
    int    msg_flags;          /* flags on received message */
};
```

XNS4.0 および POSIX 1003.1g Draft 6.6 の msghdr データ構造体は、4.4BSD と同じフィールドを持っています。ただし、msg_namelen および msg_controllen フィールドのサイズは、4.4BSD の msghdr データ構造体では 4 バイトなのに対して、XNS4.0 および POSIX 1003.1g Draft 6.6 の msghdr データ構造体では 8 バイトです。さらに、msg_iovlen フィールドのサイズが、4.4BSD および XNS4.0 の msghdr データ構造体では 4 バイト長であるのに対し、POSIX 1003.1g Draft 6.6 の msghdr データ構造体では 8 バイトです。図 4-7 に、4.3BSD、4.4BSD、XPG4、および POSIX 1003.1g Draft 6.6 の msghdr 構造体を示します。

図 4-7: 4.3BSD、4.4BSD、XNS4.0、および POSIX 1003.1g の msghdr 構造体



ZK-1468U-AIJ

4.3BSD の `msg_hdr` 構造体では、送信側プロセスは、`msg_accrights` および `msg_accrightslen` フィールドで、システムが維持するオブジェクトへのアクセス権を受信側プロセスに渡すことができます。この場合、システムが維持するオブジェクトとはソケットです。4.4BSD、XNS4.0、および POSIX 1003.1g Draft 6.6 では、`msg_control` および `msg_controllen` フィールドで渡すことができます。

4.6 一般的なソケット・エラー

表 4-6 は、一般的なソケット・エラー・メッセージと、そのメッセージが示す問題の一覧です。

表 4-6: 一般的なエラーと診断

エラー	診断
[EAFNOSUPPORT]	指定したアドレス・ファミリのアドレスを、プロトコル・ファミリがサポートしていない。
[EBADF]	ソケット・パラメータが有効でない。
[ECONNREFUSED]	接続の試みがリジェクトされた。
[EFAULT]	ポインタが、ユーザ・アドレス空間の有効な部分を指し示していない。
[EHOSTDOWN]	ホストがダウンしている。
[EHOSTUNREACH]	ホストに到達できない。
[EINVAL]	誤った引数を使用している。
[EMFILE]	現在のプロセスにあるオープンしたファイル記述子が多すぎる。
[ENETDOWN]	ネットワークがダウンしている。
[ENETUNREACH]	ネットワークに到達できない。ネットワークへの経路がない。
[ENOMEM]	システムが、プロセス記述子テーブルを増やすためのカーネル・メモリを割り当てることができなかった。
[ENOTSOCK]	ソケット・パラメータが、ソケットではなくファイルを参照している。
[EOPNOTSUPP]	指定したプロトコルでは、ソケットのペアを作成できない。
[EOPNOTSUPP]	参照したソケットは、接続を受け入れることができない。

表 4-6: 一般的なエラーと診断 (続き)

エラー	診断
[EPROTONOSUPPORT]	指定したプロトコルを，このシステムがサポートしていない。
[EPROTOTYPE]	指定したプロトコルで，ソケット・タイプをサポートしていない。
[ETIMEDOUT]	リモート・アプリケーションからの応答がないまま，接続がタイム・アウトになった。
[EWOULDBLOCK]	ソケットが非ブロッキングとマークされており，作業を完了できなかった。

4.7 高度なソケット・プログラミング情報

この節では，次の項目について説明します。これは，ソケット用に複雑なアプリケーションを作成するプログラマにとって有効な情報です。

- AF_INET6 ソケットを使用するための，アプリケーションの移植
- IPv6 raw ソケットの使用
- 特定のプロトコルの選択
- 名前とアドレスのバインド
- 帯域外データ
- IP マルチキャスト
- ブロードキャストとネットワーク構成の調査
- inetd デーモン
- 入出力の多重化
- 割り込み駆動のソケット I/O
- シグナルとプロセス・グループ
- 擬似端末

4.7.1 AF_INET6 ソケットを使用するための，アプリケーションの移植

AF_INET6 ソケットにより，アプリケーションは IPv6 プロトコル，IPv4 プロトコル，またはその両方を使用して通信できるようになります。IPv6 通信については，RFC 2553 の『*Basic Socket Interface Extensions for IPv6*』で，

BSD ソケットのアプリケーション・プログラミング・インタフェース (API) への変更点が規定されています。この変更点の要約を、表 4-7 に示します。

表 4-7: BSD ソケット API に対する IPv6 拡張の要約

カテゴリ	変更点
コア関数の呼び出し	なし。ソケット関数の基本的な構文は同じです。ソケット関数を使用するとき、アプリケーションではプロトコル固有のアドレス構造体へのポインタを、汎用の <code>sockaddr</code> アドレス構造体へのポインタにキャストしなければなりません。インターネット・アプリケーションの作成については、4.4 節を参照してください。
ソケット・アドレス構造体	IPv6 通信用に新しい <code>sockaddr_in6</code> 構造体、プロトコル非依存の通信用に <code>sockaddr_storage</code> 構造体が規定されました。 <code>sockaddr_in</code> 構造体は、IPv4 通信用に残されています。詳細は、4.7.1.2 項を参照してください。
名前からアドレスへの変換	プロトコル非依存 (IPv4 および IPv6) の通信用に、 <code>getnameinfo</code> 関数と <code>getaddrinfo</code> 関数が規定されました。 <code>gethostbyaddr</code> 関数と <code>gethostbyname</code> 関数は、IPv4 通信用に残されています。詳細は、4.7.1.3 項を参照してください。
アドレス変換関数	プロトコル非依存 (IPv4 および IPv6) のアドレス変換用に、 <code>getnameinfo</code> 関数と <code>getaddrinfo</code> 関数が規定されました。 <code>inet_ntoa</code> 関数と <code>inet_addr</code> 関数は、IPv4 アドレス変換用に残されています。詳細は、4.7.1.3 項を参照してください。
ソケット・オプション	IPv6 マルチキャスト用の新しいソケット・オプションが規定されました。詳細は、4.7.6 項を参照してください。

この項では、IPv6 ネットワーク環境で動作させるために既存のアプリケーション・コードに対して行わなければならない、次の変更について説明します。また、インターネット・アプリケーションが `AF_INET6` ソケット上でどのように動作するかも知っていなければなりません。詳細は、4.4.2 項を参照してください。

- 名前の変更
- 構造体の変更
- 関数呼び出しの変更
- アプリケーションに対するその他の変更

この項では、既存の IPv4 アプリケーションと連携して動作させる必要のあるアプリケーションについても、コードを変更する際のガイドラインを説明します。これらの変更を行うと、移植後のアプリケーションは、IPv4 と IPv6 の両方で通信できるようになります。既存の IPv4 アプリケーションは以前と同じように動作し、IPv6 アプリケーションと連携して動作させることができます。

移植作業を容易にするために、オペレーティング・システムには `ipv6_sniff` ユーティリティが用意されています。このユーティリティを使うと、ソース・ファイルを走査して変更が必要なトークンを見つけ、必要であればソース・ファイルを編集して変更することができます。このユーティリティの使い方については、4.7.1.5 項を参照してください。

4.7.1.1 名前の変更

変更のほとんどは単純で機械的なものですが、コードの再構成が少し必要になることもあります(たとえば、IPv4 アドレスを保持する `int` データ型を返すルーチンでは、`in6_addr` へのポインタのパラメータを追加して、そこに IPv6 アドレスを書き込まなければならないことがあります)。表 4-8 は、アプリケーションのコードに対して行う変更の要約です。

表 4-8: 名前の変更

変更前	変更後
<code>AF_INET</code>	<code>AF_INET6</code>
<code>PF_INET</code>	<code>PF_INET6</code>
<code>INADDR_ANY</code>	<code>in6addr_any</code>

4.7.1.2 構造体の変更

次の構造体は、構造体名とフィールド名が変更されています。

- `in_addr`
- `sockaddr_in`
- `sockaddr`
- `hostent`

以降の項で、これらの変更について説明します。

in_addr 構造体

in_addr 構造体を使用するアプリケーションは、in6_addr 構造体を使用するように必要に応じて変更しなければなりません。

AF_INET の構造体

```
struct in_addr[1]  
    unsigned int s_addr[2]
```

AF_INET6 の構造体

```
struct in6_addr[1]  
    uint8_t s6_addr [16][2]
```

必要に応じて、アプリケーションを次のように変更します。

- ① 構造体名 in_addr を in6_addr に変更する。
- ② データ型を unsigned int から uint8_t に、フィールド名を s_addr から s6_addr に変更する。

in6_addr 構造体を使用する場合は、この他にもアプリケーション対して変更が必要となる場合がありますので、4.7.1.4 項を参照してください。

sockaddr_in 構造体

4.4 BSD の sockaddr_in 構造体を使用するアプリケーションは、次の例に示すように、sockaddr_in6 構造体を使用するように必要に応じて変更しなければなりません。

AF_INET の構造体

```
struct sockaddr_in[1]  
    unsigned char sin_len[2]  
    sa_family_t sin_family[3]  
    in_port_t sin_port[4]  
    struct in_addr sin_addr[5]
```

AF_INET6 の構造体

```
struct sockaddr_in6[1]  
    uint8_t sin6_len[2]  
    sa_family_t sin6_family[3]  
    int_port_t sin6_port[4]  
    struct in6_addr sin6_addr[5]
```

必要に応じて、アプリケーションを次のように変更します。

- ① 構造体名 sockaddr_in を sockaddr_in6 に変更する。
- ② データ型を unsigned char から uint8_t に、フィールド名を sin_len から sin6_len に変更する。
- ③ フィールド名を sin_family から sin6_family に変更する。
- ④ フィールド名を sin_port から sin6_port に変更する。
- ⑤ フィールド名を sin_addr から sin6_addr に変更する。

4.3 BSD の `sockaddr_in` 構造体を使用するアプリケーションは、次の例に示すように、`sockaddr_in6` 構造体を使用するように必要に応じて変更しなければなりません。

AF_INET の構造体

```
struct sockaddr_in1  
  u_short sin_family2  
  in_port_t sin_port3  
  struct in_addr sin_addr4
```

AF_INET6 の構造体

```
struct sockaddr_in61  
  u_short sin6_family2  
  in_port_t sin6_port3  
  struct in6_addr sin6_addr4
```

必要に応じて、アプリケーションを次のように変更します。

- ① 構造体名 `sockaddr_in` を `sockaddr_in6` に変更する。
- ② フィールド名 `sin_family` を `sin6_family` に変更する。
- ③ フィールド名 `sin_port` を `sin6_port` に変更する。
- ④ フィールド名 `sin_addr` を `sin6_addr` に変更する。

注意

どちらの場合も、構造体を宣言した後に、`sockaddr_in6` 構造体全体をゼロで初期化しなければなりません。

sockaddr 構造体

汎用のソケット・アドレス構造体 (`sockaddr`) を用いて `AF_INET` ソケット・アドレス (`sockaddr_in`) を保持しているアプリケーションは、`AF_INET6` の `sockaddr_in6` 構造体か `sockaddr_storage` 構造体を使用するように変更しなければなりません。

アプリケーションが IPv6 アドレスだけを扱う場合は、必要に応じて次の変更をアプリケーションに行います。

AF_INET の構造体

```
struct sockaddr1
```

AF_INET6 の構造体

```
struct sockaddr_in61
```

- ① `struct sockaddr_in` にすべきところは、構造体名 `sockaddr` を `sockaddr_in6` に変更する。たとえば、`sizeof(struct sockaddr)`。

アプリケーションが IPv4 と IPv6 の両方のノードから送られてくるアドレスを操作する場合は、必要に応じて次の変更をアプリケーションに行います。

AF_INET の構造体

```
struct sockaddr[1]
```

AF_INET6 の構造体

```
struct sockaddr_storage[1]
```

- [1] struct sockaddr_in にすべきところは、構造体名 sockaddr を sockaddr_storage に変更する。たとえば、sizeof(struct sockaddr)。

注意

sockaddr_in6 構造体と sockaddr_storage 構造体は、どちらもサイズが sockaddr 構造体よりも大きくなっています。

hostent 構造体

hostent 構造体を使用するアプリケーションは、次の例に示すように、addrinfo 構造体を使用するように必要に応じて変更しなければなりません。

AF_INET の構造体

```
struct hostent[1]
```

AF_INET6 の構造体

```
struct addrinfo[1]
```

必要に応じて、アプリケーションを次のように変更します。

- [1] 構造体名 hostent を addrinfo に変更する。

4.7.1.3 関数呼び出しの変更

次のライブラリ・ルーチンを使用しているアプリケーションは、必要に応じて変更しなければなりません。

- gethostbyaddr
- gethostbyname
- inet_ntoa
- inet_addr

以降の項で、これらの変更について説明します。

gethostbyaddr 関数呼び出し

gethostbyaddr 関数を呼び出しているアプリケーションは、次の例に示すように、getnameinfo 関数を呼び出すように変更しなければなりません。

AF_INET の呼び出し

```
gethostbyaddr (xxx, 4, AF_INET) [1]
```

AF_INET6 の呼び出し

```
err=getnameinfo(&sockaddr,sockaddr_len, node_name, name_len,  
service, service_len, flags);[1]
```

必要に応じて、アプリケーションを次のように変更します。

- 関数名を gethostbyaddr から getnameinfo に変更し、ソケット・アドレス構造体へのポインタ、返されるノード名用の文字列、返されるノード名の長さを示す整数値、返されるサービス名用の文字列、返されるサービス名の長さを示す整数値、実行されるアドレス処理のタイプを示す整数値を指定する。フラグ・ビットとその意味についての詳細は、getnameinfo(3) を参照してください。

gethostbyname 関数呼び出し

gethostbyname 関数を呼び出すアプリケーションは、次の例に示すように、getaddrinfo 関数を呼び出すように変更しなければなりません。

AF_INET の呼び出し

```
gethostbyname (name) [1]
```

AF_INET6 の呼び出し

```
err=getaddrinfo(node_name, service_name, &hints, &result);[1]  
:  
:  
freeaddrinfo(&result);[2]
```

必要に応じて、アプリケーションを次のように変更します。

- 関数名を gethostbyname から getaddrinfo に変更し、ノード名の文字列、使用するサービス名の文字列、処理オプションが格納されている hints 構造体へのポインタ、addrinfo 構造体 (アドレス情報が返される構造体) へのポインタを指定する。ヒントのフィールドと値についての詳細は、getaddrinfo(3) を参照してください。

- ❷ freeaddrinfo ルーチンの呼び出しを追加して、アプリケーションが使用を終えたときに、addrinfo 構造体を解放する。

アドレスが IPv4 アドレスと IPv6 アドレスのどちらかであることをアプリケーションで調べる必要があるが、アドレス・ファミリからは判断できない場合は、IN6_IS_ADDR_V4MAPPED マクロを使用してください。IPv6 アドレス・テスト・マクロの一覧については、4.4.3 項を参照してください。

inet_ntoa 関数呼び出し

inet_ntoa 関数を呼び出しているアプリケーションは、次の例に示すように、getnameinfo 関数を呼び出すように変更しなければなりません。

AF_INET の呼び出し

```
inet_ntoa(addr);❶
```

AF_INET6 の呼び出し

```
err=getnameinfo(&sockaddr,sockaddr_len,node_name,name_len,
service,service_len,NI_NUMERICHOST);❶
```

必要に応じて、アプリケーションを次のように変更します。

- ❶ 関数名を gethostbyaddr から getnameinfo に変更し、ソケット・アドレス構造体へのポインタ、返されるノード名用の文字列、返されるノード名の長さを示す整数値、返されるサービス名用の文字列、返されるサービス名の長さを示す整数値、NI_NUMERICHOST フラグを指定する。フラグ・ビットとその意味についての詳細は、getnameinfo(3) を参照してください。

inet_addr 関数呼び出し

inet_addr 関数を呼び出すアプリケーションは、次の例に示すように、getaddrinfo 関数を呼び出すように変更しなければなりません。

AF_INET の呼び出し

```
result=inet_addr(&string);❶
```

AF_INET6 の呼び出し

```
err=getaddrinfo(node_name,service_name,&hints,&result);❶
:
freeaddrinfo(&result);❷
```

必要に応じて、アプリケーションを次のように変更します。

- ❶ 関数名を `inet_addr` から `getaddrinfo` に変更し、ノード名の文字列、使用するサービス名の文字列、`AI_NUMERICHOST` オプションが格納されている `hints` 構造体へのポインタ、`addrinfo` 構造体 (アドレス情報が返される構造体) へのポインタを指定する。ヒントのフィールドと値についての詳細は、`getaddrinfo(3)` を参照してください。
- ❷ `freeaddrinfo` ルーチンの呼び出しを追加して、アプリケーションが使用を終えたときに、`addrinfo` 構造体を解放する。

アドレスが IPv4 アドレスと IPv6 アドレスのどちらであるかをアプリケーションで調べる必要があるが、アドレス・ファミリからは判断できない場合は、`IN6_IS_ADDR_V4MAPPED` マクロを使用してください。IPv6 アドレス・テスト・マクロの一覧については、4.4.3 項を参照してください。

4.7.1.4 アプリケーションの他の変更

名前の変更以外にも、コードを見直して IP アドレス情報や変数を使用しているところを個別に確認してください。

IP アドレスの比較

アプリケーションが IP アドレスを比較したり、IP アドレスが等しいかテストしている場合、4.7.1.2 項で行った `in6_addr` 構造体の変更により、`int` の数値の比較が構造体の比較に変わります。これによってコードが誤った状態になり、コンパイル時にエラーになります。

必要に応じて、アプリケーションを次のように変更します。

AF_INET のコード

```
(addr1->s_addr == addr2->s_addr)❶
```

AF_INET6 のコード

```
(memcmp(addr1, addr2, sizeof(struct in6_addr)) == 0)❶
```

- ❶ 等値比較式を、`memcmp` (メモリの比較) 関数を使用したものに変更する。

AF_INET のコード

```
(addr1->s_addr == addr2->s_addr) [1]
```

AF_INET6 のコード

```
IN6_ARE_ADDR_EQUAL(addr1, addr2) [1]
```

- [1]** 等値比較式を、IN6_ARE_ADDR_EQUAL マクロを使用したものに変更する。IPv6 アドレス・テスト・マクロの一覧については、4.4.3 項を参照してください。

IP アドレスをワイルドカード・アドレスと比較

アプリケーションが IP アドレスをワイルドカード・アドレスと比較している場合、4.7.1.2 項で行った in6_addr 構造体への変更により、int の数値の比較が構造体の比較に変わります。これによってコードが誤った状態になり、コンパイル時にエラーになります。

必要に応じて、アプリケーションを次のように変更します。

AF_INET のコード

```
(addr->s_addr == INADDR_ANY) [1]
```

AF_INET6 のコード

```
IN6_IS_ADDR_UNSPECIFIED(addr) [1]
```

- [1]** 等値比較式を、IN6_IS_ADDR_UNSPECIFIED マクロを使用したものに変更する。IPv6 アドレス・テスト・マクロの一覧については、4.4.3 項を参照してください。

AF_INET のコード

```
(addr->s_addr == INADDR_ANY) [1]
```

AF_INET6 のコード

```
(memcmp(addr, in6addr_any, sizeof(struct in6_addr)) == 0) [1]
```

- [1]** 等値比較式を、memcmp (メモリの比較) 関数を使用したものに変更する。

int データ型による IP アドレスの保持

アプリケーションが int データ型を用いて IP アドレスを保持している場合、4.7.1.2 項で行った in6_addr 構造体への変更によって、代入が変わります。これによってコードが誤った状態になり、コンパイル時にエラーになります。

必要に応じて、アプリケーションを次のように変更します。

AF_INET のコード

```
struct in_addr foo;
int bar; [1]
:
:
bar = foo.s_addr; [2]
```

AF_INET6 のコード

```
struct in6_addr foo;
struct in6_addr bar; [1]
:
:
bar = foo; [2]
```

[1] bar のデータ型を int から struct in6_addr に変更する。

[2] bar への代入文を変更して、s_addr フィールドの参照を削除する。

IP アドレスを返す関数の使用

IP アドレスを int データ型で返す関数をアプリケーションで使用している場合、4.7.1.2 項で行った in6_addr 構造体の変更により、戻り値の代入先が int から char の配列に変わります。これによってコードが誤った状態になり、コンパイル時にエラーになります。

必要に応じて、アプリケーションを次のように変更します。

AF_INET のコード

```
struct in_addr *addr;
addr->s_addr = foo(xxx); [1]
```

AF_INET6 のコード

```
struct in6_addr *addr;
foo(xxx, addr); [1]
```

[1] 関数を再構築して、呼び出し時に構造体のアドレスを渡せるようにする。さらに、関数を変更して addr が指す構造体の中に戻り値を書き込む。

ソケット・オプションの変更

アプリケーションが IPv4 IP レベルのソケット・オプションを使用している場合、これを対応する IPv6 オプションに変更します。詳細は、ip(7) を参照してください。

4.7.1.5 ipv6_sniff ユーティリティの使用

ipv6_sniff ユーティリティは、1 個以上のファイルを走査して IPv6 へ移植する上で問題となる可能性のある部分を検出します。特に指定しなければ、このユーティリティは IPv4 専用ソケットの使用およびオプションと、IPv4 の名前およびアドレス解決を探します。ファイルの走査が終わると、このユーティリティは結果をソートして報告します。このユーティリティ内からユーザの好みに合ったエディタを起動し、ファイルを参照したり、変更することもできます。

このユーティリティを実行するには、次のコマンドを入力します。

```
# /usr/sbin/ipv6sniff filename...
```

詳細については、`ipv6_sniff(8)` を参照してください。

4.7.2 IPv6 raw ソケットの使用

raw ソケットは、IPv4 と IPv6 の両方で使われ、TCP および UDP のトランスポート層をバイパスします。表 4-9 に、IPv4 raw ソケットと IPv6 raw ソケットの主な相違点を示します。

表 4-9: IPv4 raw ソケットと IPv6 raw ソケットの相違点

	IPv4	IPv6
用途	ICMPv4, IGMPv4 のアクセスと、カーネルが認識しないプロトコル・フィールドを含む IPv4 データグラムの読み書き。	ICMPv6 のアクセスと、カーネルが認識しない次ヘッダ・フィールドを含む IPv6 データグラムの読み書き。
バイト順	規定していない。	すべてのデータ送受信で、ネットワーク・バイト順。
完全なパケットの送受信	はい。	いいえ。補助データ・オブジェクトを使って、拡張ヘッダとホップ限界情報を転送します。

アプリケーションが完全な IPv6 パケットにアクセスする必要がある場合は、DLPI (Data Link Provider Interface) を使います。

出力の場合、補助データまたはソケット・オプションを使うと、すべてのフィールド (フロー・ラベル・フィールドを除く) を変更できます。入力の場合、アプリケーションはすべてのフィールド (フロー・ラベル、バージョン番号、次ヘッダ・フィールド、およびすべての拡張ヘッダを除く) を、補助データを使ってアクセスできます。

ICMPv6 raw ソケット以外の IPv6 raw ソケットでは、アプリケーションは `IPV6_CHECKSUM` ソケット・オプションを設定しなければなりません。たとえば、次のようにします。

```
int offset = 2;
setsockopt (fd, IPPROTO_IPV6, IPV6_CHECKSUM, &offset, sizeof(offset));
```

これによりカーネルは、出力時にはチェックサムを計算して格納し、入力時にはチェックサムを検査できるようになります。このためアプリケーション

は、すべての出力パケットでソース・アドレスを選択する必要がなくなります。このソケット・オプションは、省略時の設定では無効です。オフセット変数に -1 を設定しても、このオプションを無効にできます。

IPv6 raw ソケットを使うと、アプリケーションは次の情報にアクセスできます。

- ICMPv6 メッセージ
- IPv6 ヘッダ
- ルーティング・ヘッダ
- IPv6 オプション・ヘッダ (ホップ・バイ・ホップ・オプション・ヘッダとデスティネーション・オプション・ヘッダ)

この項では、これらの情報にアクセスする方法について説明します。

4.7.2.1 ICMPv6 メッセージへのアクセス

ICMPv6 raw ソケットは、`socket` 関数を `PF_INET6`, `SOCK_RAW`, および `IPPROTO_ICMPV6` 引数で呼び出したときに作成されるソケットです。カーネルは、すべての送信 ICMPv6 パケットに ICMPv6 チェックサムを計算して挿入し、すべての受信パケットのチェックサムを検査します。受信チェックサムが不正の場合、パケットは廃棄されます。

ICMPv6 は ICMPv4 のスーパーセットのため、ICMPv6 raw ソケットは ICMPv4 raw ソケットよりも多くのメッセージを受信できます。特に指定しなければ、ICMPv6 raw ソケットを作成すると、ソケットはすべてのタイプの ICMPv6 メッセージをアプリケーションに渡します。ただし、アプリケーションは、すべてのメッセージにアクセスする必要はありません。アプリケーションは、ICMPv6 フィルタを作成することで、渡される ICMPv6 メッセージのタイプを指定できます。

ICMPv6 フィルタのデータ型は、`struct icmp6_filter` です。`getsockopt` を使って現在のフィルタを取り出し、`setsockopt` を使ってフィルタを格納します。たとえば、ICMPv6 メッセージのフィルタリングを有効にするには、次のように `ICMP6_FILTER` オプションを使います。

```
struct icmp6_filter myfilter;  
setsockopt (fd, IPPROTO_ICMPV6, IPV6_FILTER, &myfilter, sizeof(myfilter));
```

`myfilter` の値は、ICMPv6 のメッセージ・タイプ (0 ~ 255) です。

表 4-10 に、すべての ICMPv6 フィルタ・マクロとその説明を示します。

表 4-10: ICMPv6 フィルタリング・マクロの要約

マクロ	説明
ICMP6_FILTER_SETPASSALL	すべての ICMPv6 メッセージをアプリケーションに渡します。
ICMP6_FILTER_SETBLOCKALL	すべての ICMPv6 メッセージをアプリケーションに渡さずブロックします。
ICMP6_FILTER_SETPASS	指定されたタイプの ICMPv6 メッセージをアプリケーションに渡します。
ICMP6_FILTER_SETBLOCK	指定されたタイプの ICMPv6 メッセージをアプリケーションに渡さずブロックします。
ICMP6_FILTER_WILLPASS	指定されたタイプのメッセージがアプリケーションに渡されるときに true を返します。
ICMP6_FILTER_WILLBLOCK	指定されたタイプのメッセージがアプリケーションに渡されずブロックされるときに true を返します。

設定済みのフィルタをクリアするには、ICMP_FILTER オプションに長さ 0 のフィルタを指定して `setsockopt` を呼び出します。詳細については、`icmp(7)` を参照してください。

カーネルは、メッセージ・タイプ、メッセージの内容、パケット構造の正当性チェックは行いません。このチェックは、アプリケーションが行う必要があります。

4.7.2.2 IPv6 ヘッダへのアクセス

IPv6 raw ソケットを使う場合、アプリケーションは IPv6 ヘッダの内容を受信できなければなりません。このオプション情報を受信するには、対応するソケット・オプションを指定して `setsockopt` システム・コールを呼び出します。表 4-11 に、オプション情報と、対応するソケット・オプションの一覧を示します。

表 4-11: オプション情報とソケット・オプション

オプション情報	ソケット・オプション	cmsg_type
デスティネーション IPv6 アドレスと受信インタフェース (受信の場合)。ソース IPv6 アドレスと送信インタフェース (送信の場合)	IPV6_RECVPKTINFO	IPV6_PKTINFO
ホップ限界値	IPV6_RECVHOPLIMIT	IPV6_HOPLIMIT
ルーティング・ヘッダ	IPV6_RECVRTHDR	IPV6_RTHDR
ホップ・バイ・ホップ・オプション	IPV6_RECVHOPOPTS	IPV6_HOPOPTS
デスティネーション・オプション	IPV6_RECVDSTOPTS	IPV6_DSTOPTS

これらのソケット・オプションについての詳細は、ip(7) を参照してください。

recvmsg システム・コールは、受信したデータを、cmsghdr データ構造体内の 1 個以上の補助データ・オブジェクトとして返します。この構造体についての詳細は、4.2.3.4 項を参照してください。

ソケット・オプションの値を調べるには、対応するオプションを指定して getsockopt システム・コールを呼び出します。IPV6_RECVPKTINFO オプションが設定されていない場合、この関数は、ipi6_addr に in6addr_any が設定され、ipi6_addr に 0 が設定された状態で in6_pktinfo データ構造体を返します。他のオプションについては、オプション値がない場合、この関数は option_len の値として 0 を返します。

アプリケーションは次の IPv6 ヘッダ情報を、入力パケットの補助データとして受信できます。

- デスティネーション IPv6 アドレス
- インタフェース・インデックス
- ホップ限界値

IPv6 アドレスとインタフェース・インデックスは、recvmsg システム・コールの補助データとして受信された in6_pktinfo データ構造体に含まれています。in6_pktinfo データ構造体は、netinet/in.h に定義されています。

IPv6 アドレスの受信

IPV6_RECVPKTINFO オプションが有効の場合、recvmsg システム・コールは in6_pktinfo データ構造体を補助データとして返します。 ipi6_addr メンバには、受信パケットから取り出したデスティネーション IPv6 アドレスが入っています。 TCP ソケットの場合、デスティネーション・アドレスはコネクションのローカル・アドレスです。

インタフェースの受信

IPV6_RECVPKTINFO オプションが有効の場合、recvmsg システム・コールは in6_pktinfo データ構造体を補助データとして返します。 ipi6_ifindex メンバには、パケットを受信したインタフェースのインタフェース・インデックスが入っています。

ホップ限界値の受信

IPV6_RECVHOPLIMIT オプションが有効の場合、recvmsg システム・コールは cmsghdr データ構造体を補助データとして返します。 cmsg_type メンバは IPV6_HOPLIMIT で、cmsg_data[] メンバには整数のホップ限界値の第 1 バイトが入っています。

4.7.2.3 IPv6 ルーティング・ヘッダへのアクセス

Advanced Sockets API を使うと、IPv6 ルーティング・ヘッダにアクセスできます。 ルーティング・ヘッダは IPv6 の拡張ヘッダであり、アプリケーションがソース・ルーティングを行えるようにします。 現在 RFC 2460 では、127 個までの中間ノード (128 ホップ) をサポートするタイプ 0 のルーティング・ヘッダが定義されています。

表 4-12 に、アプリケーションがルーティング・ヘッダの作成や調査に使うソケット呼び出しの一覧を示します。 以降この項では、ルーティング・ヘッダに関連する作業について説明します。

表 4-12: ルーティング・ヘッダ用のソケット呼び出し

名前	説明
inet6_rth_space	ルーティング・ヘッダに必要なバイト数を返します。
inet6_rth_init	ルーティング・ヘッダのバッファ・データを初期化します。

表 4-12: ルーティング・ヘッダ用のソケット呼び出し (続き)

名前	説明
<code>inet6_rth_add</code>	ルーティング・ヘッダにアドレスを 1 つ追加します。
<code>inet6_rth_reverse</code>	ルーティング・ヘッダ内のフィールドの順序を逆にします。
<code>inet6_rth_segments</code>	ルーティング・ヘッダ内のセグメント数 (アドレスの数) を返します。
<code>inet6_rth_getaddr</code>	ルーティング・ヘッダからアドレスを 1 つ取り出します。

ルーティング・ヘッダの受信

ルーティング・ヘッダを受信するには、アプリケーションは `IPV6_RECVRTHDR` オプションを有効にして `setsockopt` を呼び出します。このオプションの使用例は、`ip(7)` を参照してください。

各受信ルーティング・ヘッダに対し、カーネルは補助データ・オブジェクトを 1 つ、`cmsg_type` メンバに `IPV6_RTHDR` が設定された `cmsg_hdr` 構造体に格納して渡します。アプリケーションは、`inet6_rth_reverse`、`inet6_rth_segments`、`inet6_rth_getaddr` を呼び出して、ルーティング・ヘッダを処理します。詳細については、`inet6_rth_reverse(3)`、`inet6_rth_segments(3)`、および `inet6_rth_getaddr(3)` を参照してください。

ルーティング・ヘッダの送信

ルーティング・ヘッダを送信するには、アプリケーションはルーティング・ヘッダを `sendmsg` 呼び出しの補助データとして指定するか、`setsockopt` を呼び出して指定します。アプリケーションは、`IPV6_RTHDR` オプションの `setsockopt` を呼び出して、オプションの長さにゼロ (0) を指定することで、スティッキ・ルーティング・ヘッダを削除できます。

補助データを使うときには、アプリケーションは `cmsg_level` メンバに `IPPROTO_IPV6` を設定し、`cmsg_type` メンバに `IPV6_RTHDR` を設定します。`inet6_rth_space` 呼び出し、`inet6_rth_init` 呼び出し、および `inet6_rth_add` 呼び出しを使って、ルーティング・ヘッダを構築しま

す。詳細については、`inet6_rth_space(3)`、`inet6_rth_init(3)`、および `inet6_rth_append(3)` を参照してください。

アプリケーションがルーティング・ヘッダを指定するときには、`connect`、`sendto`、または `sendmsg` の呼び出しで指定されたデスティネーション・アドレスが、データグラムの最終デスティネーションになります。このため、ルーティング・ヘッダには、すべての中間ノードのアドレスが入っています。

RFC 2460 で規定されている拡張ヘッダの順序の都合上、アプリケーションは送信ルーティング・ヘッダを 1 つだけ指定できます。

4.7.2.4 IPv6 オプション・ヘッダへのアクセス

Advanced Sockets API を使うと、アプリケーションは次の IPv6 オプション・ヘッダにアクセスできます。

- ホップ・バイ・ホップ

1 つのホップ・バイ・ホップ・オプション・ヘッダに、複数のホップ・バイ・ホップ・オプションを入れることができます。各オプションは、タイプ、長さ、値 (TLV) がコード化されています。アプリケーションはスティッキ・オプションまたは補助データを使って、この情報をカーネルとやりとりします。

- デスティネーション

1 個以上のデスティネーション・オプション・ヘッダに、複数のデスティネーション・オプションを入れることができます。ルーティング・ヘッダより前にあるデスティネーション・オプション・ヘッダは、ルーティング・ヘッダ内に指定されている 1 番目以降のデスティネーションによって処理されます。ルーティング・ヘッダより後にあるデスティネーション・オプションは、最後のデスティネーションによってのみ処理されます。各オプションは、タイプ、長さ、値 (TLV) がコード化されています。アプリケーションはスティッキ・オプションまたは補助データを使って、この情報をカーネルとやりとりします。

ヘッダの境界合わせの要件や拡張ヘッダの順序についての詳細は、RFC 2460 を参照してください。

表 4-13 に、ホップ・バイ・ホップ・オプション・ヘッダおよびデスティネーション・オプション・ヘッダの構築や調査にアプリケーションが使う

ソケット呼び出しの一覧を示します。以降この項では、これらのオプション・ヘッダに関連する作業について説明します。

表 4-13: オプション・ヘッダ用のソケット呼び出し

名前	説明
<code>inet6_opt_init</code>	オプションのバッファ・データを初期化します。
<code>inet6_opt_append</code>	オプション・ヘッダにオプションを追加します。
<code>inet6_opt_finish</code>	オプション・ヘッダへのオプションの追加を終了します。
<code>inet6_opt_set_val</code>	オプション・ヘッダにオプション・コンテンツの構成要素を 1 個追加します。
<code>inet6_opt_next</code>	オプション・ヘッダから次のオプションを取り出します。
<code>inet6_opt_find</code>	オプション・ヘッダから、指定されたタイプのオプションを取り出します。
<code>inet6_opt_get_val</code>	オプション・ヘッダから、オプション・コンテンツの構成要素を 1 個取り出します。

ホップ・バイ・ホップ・オプションの受信

ホップ・バイ・ホップ・オプション・ヘッダを受信するには、アプリケーションは `IPV6_RECVHOPOPTS` オプションを有効にして `setsockopt` を呼び出します。このオプションの使用例は、`ip(7)` を参照してください。

補助データを使う場合、カーネルはホップ・バイ・ホップ・オプション・ヘッダをアプリケーションに渡し、`cmsg_level` メンバに `IPPROTO_IPV6` を設定し、`cmsg_type` メンバに `IPV6_HOPOPTS` を設定します。

アプリケーションは `inet6_opt_next`、`inet6_opt_find`、および `inet6_opt_get_val` を呼び出して、これらのオプションを取り出します。詳細については、`inet6_opt_next(3)`、`inet6_opt_find(3)`、および `inet6_opt_get_val(3)` を参照してください。

ホップ・バイ・ホップ・オプションの送信

ホップ・バイ・ホップ・オプション・ヘッダを送信するには、アプリケーションはヘッダを `sendmsg` 呼び出しの補助データとして指定するか、`setsockopt` を呼び出して指定します。アプリケーションは、`IPV6_HOPOPTS` オプションで `setsockopt` を呼び出して、オプションの長

さにゼロ (0) を指定することで、スティッキ・ホップ・バイ・ホップ・オプション・ヘッダを削除できます。

補助データを使う場合、すべてのホップ・バイ・ホップ・オプションは、1 個の補助データ・オブジェクトで指定されます。アプリケーションは、`cmsg_level` メンバに `IPPROTO_IPV6` を設定し、`cmsg_type` メンバに `IPV6_HOPOPTS` を設定します。 `inet6_opt_init` 呼び出し、`inet6_opt_append` 呼び出し、`inet6_opt_finish` 呼び出し、および `inet6_opt_set_val` 呼び出しを使って、オプション・ヘッダを構築します。 詳細については、`inet6_opt_init(3)`、`inet6_opt_append(3)`、`inet6_opt_finish(3)`、および `inet6_opt_set_val(3)` を参照してください。

デスティネーション・オプションの受信

デスティネーション・オプション・ヘッダを受信するには、アプリケーションは `IPV6_RECVDSTOPTS` オプションを有効にして `setsockopt` を呼び出します。 このオプションの使用例は、`ip(7)` を参照してください。 カーネルは、各デスティネーション・オプション・ヘッダを 1 個の補助データ・オブジェクトとしてアプリケーションに渡し、`cmsg_level` メンバに `IPPROTO_IPV6` を設定し、`cmsg_type` メンバに `IPV6_DSTOPTS` を設定します。

アプリケーションは、`inet6_opt_next`、`inet6_opt_find`、および `inet6_opt_get_val` を呼び出して、これらのオプションを処理します。 詳細については、`inet6_opt_next(3)`、`inet6_opt_find(3)`、および `inet6_opt_get_val(3)` を参照してください。

デスティネーション・オプションの送信

デスティネーション・オプション・ヘッダを送信するには、アプリケーションはヘッダを `sendmsg` 呼び出しの補助データとして指定するか、`setsockopt` を呼び出して指定します。 アプリケーションは、`IPV6_DSTOPTS` オプションで `setsockopt` を呼び出して、オプションの長さにゼロ (0) を指定することで、スティッキ・デスティネーション・オプション・ヘッダを削除できます。

RFC 2460 に従い、API は拡張ヘッダが順番に並んでいることを前提としています。 デスティネーション・オプションは、ルーティング・ヘッダの前に 1 セットだけと、ルーティング・ヘッダの後ろに 1 セットだけ置くことができます。 各セットには 1 個以上のオプションを含めることができますが、各セットは 1 個の拡張ヘッダと見なされます。

ルーティング・ヘッダの後ろのデスティネーション・オプションで補助データを使う場合や、ルーティング・ヘッダが指定されていない場合、アプリケーションは `cmsg_level` メンバに `IPPROTO_IPV6` を設定し、`cmsg_type` メンバに `IPV6_DSTOPTS` を設定します。

アプリケーションは `inet6_opt_init`、`inet6_opt_append`、`inet6_opt_finish`、および `inet6_opt_set_val` を呼び出して、これらのオプションを構築します。詳細については、`inet6_opt_init(3)`、`inet6_opt_append(3)`、`inet6_opt_finish(3)`、および `inet6_opt_set_val(3)` を参照してください。

4.7.3 特定のプロトコルの選択

`socket` システム・コールの構文は、4.3.1 項で説明しています。`socket` システム・コールの 3 番目の引数の `protocol` 引数がゼロ (0) である場合、`socket` システム・コールは、省略時のプロトコルを選択して、返されたソケット記述子と一緒に使用します。省略時のプロトコルは通常適切であるので、必ずしも代替りのプロトコルを選択できるとは限りません。ただし、`raw` ソケットを使用して、低レベルのプロトコルまたはハードウェア・インタフェースと直接通信するには、プロトコル引数を必ず使用して、多重分離をセットアップします。

たとえば、インターネット・ファミリの `raw` ソケットを使用して、IP の上に新しいプロトコルをインプリメントすることができます。この場合、そのソケットは、指定されたプロトコルのパケットだけを受信します。特定のプロトコルを使用するには、通信ドメイン内で定義されているプロトコル番号を確認しなければなりません。インターネット・ドメインでは、4.2.3.2 項で説明しているライブラリ・ルーチンの1つを使用することができます。

次のコードは、`getprotobyname` ライブラリ・コールを使用して、インターネット・ドメインでオープンする `SOCK_STREAM` ソケットのプロトコル `newtcp` を選択する方法を示しています。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
.
.
.
struct protoent *pp;
.
```

```

.
.
pp = getprotobyname("newtcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);

```

4.7.4 名前とアドレスのバインド

bind システム・コールは、アドレスをソケットに関連付けます。

4.7.4.1 ワイルドカード・アドレスへのバインド

ローカル・マシンのソケットのアドレスには、そのマシンの任意の有効なネットワーク・アドレスを指定することができます。1つのシステムは、有効なネットワーク・アドレスを複数持つことができるので、インターネット・ドメインでソケットにアドレスをバインドすると複雑になることがあります。ローカル・アドレスのバインドを簡単にするために、ワイルドカード・アドレスである、定数 INADDR_ANY (AF_INET) と in6addr_any (AF_INET6) が用意されています。アドレスが複数ある場合、ワイルドカード・アドレスは、このサーバ・プロセスが、どのインターネット・インタフェースの接続でも受け入れることをシステムに示します。

次の例は、ワイルドカード値 INADDR_ANY をローカル・ソケットにバインドする方法を示しています。

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

main()
{
    int s, length;
    struct sockaddr_in name;
    char buf[1024];
    .
    .
    .
    /* Create name with wildcards. */
    name.sin_family = AF_INET;
    name.sin_len = sizeof(name);
    name.sin_addr.s_addr = INADDR_ANY;
    name.sin_port = 0;
    if (bind(s, (struct sockaddr *)&name, sizeof(name)) == -1) {
        perror("binding datagram socket");
        exit(1);
    }
}

```

```
·  
·  
·  
}
```

ワイルドカードのローカル・アドレスを持つソケットは、指定されたポート番号宛のメッセージを受信でき、該当するホストに割り当てられた使用可能なアドレスのうちの任意のアドレスに対してメッセージを送信できます。ソケットは、ローカル・アドレスにワイルドカード値を使用しますが、指定のソケットにメッセージを送信するプロセスは、有効なネットワーク・アドレスを指定しなければならないことに注意してください。プロセスはどこからのメッセージでも受信できますが、どこへでも送信できるわけではありません。

AF_INET ソケットはシステム上の IPv4 アドレスを指定するメッセージだけを受信できます。しかし、AF_INET6 ソケットはシステム上の IPv4 アドレス、もしくは IPv6 アドレスを指定して送信されたメッセージのどちらでも受信できます。AF_INET6 ソケットは、IPv4 アドレス表現するのに IPv4 にマップされた IPv6 アドレス形式を使用しています。IPv6 アドレスについては、『ネットワーク管理ガイド：接続編』を参照してください。

複数のネットワーク・インタフェースを持つシステム上のサーバ・プロセスで、そのインタフェース・アドレスの 1 つだけとホストが接続できるようにする場合には、サーバ・プロセスは、該当するインタフェースのアドレスをバインドします。たとえば、システムに、130.180.123.45 と 131.185.67.89 の 2 つのアドレスがある場合、サーバ・プロセスは、アドレス 130.180.123.45 をバインドすることができます。このアドレスをバインドすると、130.180.123.45 への接続だけがサーバ・プロセスに接続できるようになります。

同様に、ローカル・ポートは、指定しないまま（ゼロを指定する）にしておくことができます。この場合には、システムによって、ポート番号が選択されます。

4.7.4.2 UNIX ドメインにおけるバインド

UNIX ドメイン (AF_UNIX) で通信しているプロセスは、ローカル・パス名および外部パス名を含んだ関連付けによってバインドされます。UNIX ドメインのソケットには、名前をバインドする必要はありません。ただし、名前をバインドすれば、プロトコル、ローカル・パス名、または外部パス名のバインドが重複することがなくなります。パス名は、システムに存在して

いるファイルを参照することはできません。名前をソケットにバインドするプロセスは、バインドされたソケットが入るディレクトリに対する書き込み許可がなければなりません。

次の例は、名前 `socket` を、UNIX ドメインで作成されたソケットにバインドする方法を示しています。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define NAME "socket"

main()
{
    int s, length;
    struct sockaddr_un name;
    char buf[1024];
    :

    /* Create name. */
    name.sun_len = sizeof(name.sun_len) +
        sizeof(name.sun_family) +
        strlen(NAME);
    name.sun_family = AF_UNIX;
    strcpy(name.sun_path, NAME);
    if (bind(s, (struct sockaddr *) &name, sizeof(name)) == -1) {
        perror("binding name to datagram socket");
        exit(1);
    }
    :
}
```

4.7.5 帯域外データ

帯域外データは、接続されたストリーム・ソケットの各ペアに関連付けられた、論理的に独立している伝送チャンネルです。帯域外データは、`setsockopt` システム・コールを使用して設定される `SO_OOBINLINE` オプションの状態に応じて、通常の受信キューとは無関係に、または受信キュー内で、ソケットに引き渡すことができます。

ストリーム・ソケットの抽象表現では、帯域外データ処理機能は、少なくとも一度に 1 つは高い信頼性で帯域外メッセージの引き渡しを行わなければならないことが指定されています。このメッセージには、1 バイト以上のデー

タが入っていなければなりません。また、どの時点でも、1 つ以上のメッセージのユーザへの引き渡しを保留できます。

ソケット層は、緊急データまたは帯域外処理の終わりを示す、データ・ストリーム内のマークをサポートします。ソケット・メカニズムでは、1 つのシステム・コールでマークの両側からのデータを返しません。

MSG_PEEK を使用すると、帯域外データの中身を見ることができます。ソケットにプロセス・グループがある場合は、プロトコルにその存在が通知されたときに、SIGURG シグナルが生成されます。プロセスは、プロセス・グループまたはプロセス ID を、適切な `fcntl` コールにより SIGURG シグナルを使用して通知されるように設定できます。SIGIO については、4.7.10 項で説明します。

複数のソケットに、引き渡しを待つ帯域外データがある場合、アプリケーション・プログラムは、`select` システム・コールを使用して例外的な条件を調べ、そのような保留中のデータを持つソケットを判別できます。SIGURG シグナルまたは `select` システム・コールは、アプリケーション・プログラムに、データが保留中であることを通知します。この場合、アプリケーションは、適切な呼び出しを発行して、実際にデータを受信しなければなりません。

引き渡された情報に加えて、データ・ストリームに論理マークが置かれ、帯域外データが送信されたポイントを示します。シグナルが保留中の出力をフラッシュすると、データ・ストリーム内の論理マークの前までのすべてのデータが破棄されます。

帯域外メッセージを送信するには、`send` または `sendto` のシステム・コールに MSG_OOB フラグをセットします。帯域外データを受信するには、`recvfrom` または `recv` のシステム・コールを実行するときに、アプリケーション・プログラムで MSG_OOB フラグをセットしなければなりません。

アプリケーション・プログラムは、次の `ioctl` の SIOCATMARK を使用することによって、読み取りポイントが、データ・ストリーム内のマークを現在参照しているかどうかを判別できます。

```
ioctl(s, SIOCATMARK, &yes);
```

この呼び出しが戻ったとき `yes` が 1 の場合は、帯域外データが到着していないことを示し、次の読み取りでは、マークより後のデータが返されます。帯域外データが到着している場合には、次の読み取りで、帯域外シグナルを伝送する前にクライアントから送信されたデータが返されます。次

のプログラムは、リモート・ログイン・プロセスで使用して、割り込みまたは終了シグナルの受信時に出力をフラッシュするルーチンです。このプログラムは、通常データをマークの前まで (破棄するために) 読み取ってから、帯域外バイトを読み取ります。

```
#include <sys/ioctl.h>
#include <sys/file.h>
:

oob()
{
    int out = FWRITE, mark;
    char waste[BUFSIZ];

    /* flush local terminal output */
    ioctl(1, TIOCFLUSH, (char *)&out);
    for (;;) {
        if (ioctl(rem, SIOCATMARK, &mark) < 0) {
            perror("ioctl");
            break;
        }
        if (mark)
            break;
        (void) read(rem, waste, sizeof (waste));
    }
    if (recv(rem, &mark, 1, MSG_OOB) < 0) {
        perror("recv");
    }
    :
}
```

プロセスでは、論理マークより前のデータを先に読み取らなくても、帯域外データを読み取ったり、中身を見ることができます。これは、基礎となるプロトコルが、緊急帯域内データを通常データとともに引き渡し、その存在を示す通知だけを前もって送信する場合には困難です。たとえば、TCP プロトコルがそうです。そのようなプロトコルでは、帯域外バイトが到着していない場合に、`recv` システム・コールが `MSG_OOB` フラグをセットして実行されると、このシステム・コールは、`EWOULDBLOCK` エラーを返します。入力バッファに帯域内データが十分入っていると、通常のフロー制御は、バッファがクリアされるまで、対等プロセスが緊急データを送信するのを妨げることがあります。このとき、プロセスは、キューに入ったデータを十分に読み取って、緊急データが引き渡せるようにしなければなりません。

注意

プログラムの中には、緊急データの複数バイトを使用して、複数の緊急シグナルを処理しなければならないものがあります。このようなプログラムでは、ストリーム内の緊急データの位置を保持する必要があります。ソケット・レベルの `SO_OOBINLINE` オプションがこの機能を提供しているので、できるだけこのオプションを使用してください。

ソケット・レベルの `SO_OOBINLINE` オプションは、緊急データの位置 (論理マーク) を保持します。緊急データは、`MSG_OOB` フラグなしで返された通常データ・ストリーム内で、マークの直後に続きます。複数の緊急指示を受信するとマークは移動しますが、帯域外データは失われません。

4.7.6 インターネット・プロトコル・マルチキャスト

インターネット・プロトコル (IP) ・マルチキャストは、アプリケーションがイーサネットおよび Fiber Distribution Data Interface (FDDI) ネットワークのマルチキャスト機能に IP 層アクセスを行うことができるようにします。IP マルチキャストはデータグラムを効率よく引き渡そうとするため、IP ブロードキャスト (4.7.7 項で説明) が関係のないホストに課すオーバーヘッドを避けます。IP マルチキャストはまた、アプリケーションによるネットワーク帯域幅の消費もありません。IP マルチキャストを使用しない場合、アプリケーションは同一データの入っている別々のパケットを複数のデスティネーションに伝送します。

IPv4 マルチキャストは、マルチキャスト・グループを使用することにより、複数ホストへの配信を効率よく行います。マルチキャスト・グループは、単一のクラス D IP デスティネーション・アドレス (IPv4) または単一のマルチキャスト・アドレス (IPv6) によって識別される 0 個以上のノードから構成されるグループです。IPv4 クラス D アドレスの上位 4 ビットは 1110 です。ドット表記 10 進数では、IP マルチキャスト・アドレスは 224.0.0.0 から 239.255.255.255 までの範囲ですが、224.0.0.0 は予約されています。IPv6 マルチキャスト・アドレスのフォーマットには、プレフィックス `FF00::/8` が付いています。

あるマルチキャスト・グループのメンバは、そのマルチキャスト・グループを表す IP アドレスに送信されたすべてのデータのコピーを受信します。マ

ルチキャスト・グループは永久的なものでも一時的なものでもかまいません。永久的なグループは、管理用に割り当てられた周知の IP アドレスを持ちます。永久的なグループで、永久的なものはアドレスであり、メンバではありません。グループのメンバ数は変化し、ゼロになることもあります。

IPv4 では、all hosts というグループは永久的なホスト・グループの例であり、割り当てられているアドレスは 224.0.0.1 です。Tru64 UNIX システムは、インターネット・グループ管理プロトコル (IGMP) に加わるために all hosts group に加わります。IGMP および IP マルチキャストについての詳細は、RFC 1112: 『*Host Extensions for IP Multicasting*』を参照してください。

IPv6 では、All Nodes マルチキャスト・アドレスは、アドレスが FF01::1 (ノード・ローカル、つまりスコープ 1) および FF02::1 (リンク・ローカル、つまりスコープ 2) の永久的なグループの一例です。IPv6 マルチキャスト・アドレスについての詳細は、RFC 1884: 『*IPv6 Addressing Architecture*』を参照してください。

永久的なマルチキャスト・グループのために予約されていない IP アドレスは、一時的なグループに動的に割り当てることができます。一時的なグループは 1 つ以上のメンバを持つときのみ存在します。

注意

IP マルチキャストは、TCP などのコネクション指向型トランスポートではサポートされていません。

IP マルチキャストは、次の項で説明するように、setsockopt システム・コールにオプションを使用してインプリメントされます。マルチキャストに関係するソケット・オプションに必要な定義は <netinet/in.h> にあります。IP マルチキャスト・データグラムを受信するアプリケーションには、このヘッダ・ファイルをインクルードしなければなりません。

4.7.6.1 IPv4 マルチキャスト・データグラムの送信

IPv4 マルチキャスト・データグラムを送信するには、sendto システム・コールにおいて、224.0.0.0 から 239.255.255.255 の範囲で IP デスティネーション・アドレスを指定することにより、送信先のホスト・グループを示します。システムはデータグラムを伝送する前に、指定された IP デス

ティネーション・アドレスを適切なイーサネットまたは FDDI マルチキャスト・アドレスにマップします。

アプリケーションは、`setsockopt` システム・コールに引数を指定することにより、マルチキャスト・オプションを明示的に制御できます。次のオプションは、`setsockopt` システム・コールを使用して、アプリケーションで設定できます。

- Time-to-live (TTL) フィールド (`IP_MULTICAST_TTL`)
- マルチキャスト・インタフェース (`IP_MULTICAST_IF`)
- ローカル引き渡しのループバックの使用不能化 (`IP_MULTICAST_LOOP`)

注意

`setsockopt` システム・コールの構文および引数については、4.3.5 項および `setsockopt(2)` に説明があります。この項および 4.7.6.2 項の例は、IPv4 マルチキャスト・データグラムだけに適用される `setsockopt` オプションの使用方法を示しています。

`setsockopt` システム・コールの `IP_MULTICAST_TTL` オプションを使用すると、アプリケーションは time-to-live (TTL) フィールドに対して 0 ~ 255 の値を指定することができます。TTL 値が 0 のマルチキャスト・データグラムは、ローカル・ホスト上で実行しているアプリケーションへのマルチキャスト・データグラムの配信を制限します。TTL 値が 1 のマルチキャスト・データグラムは、ローカル・サブネット上のホストのみに転送されます。マルチキャスト・データグラムの TTL 値が 1 より大きく、送信側のホストのネットワークにマルチキャスト・ルータが接続されている場合には、マルチキャスト・データグラムはローカル・サブネットを越えて転送されます。マルチキャスト・ルータは、指定されたマルチキャスト・グループに属するホストを持つ既知のネットワークにデータグラムを転送します。TTL 値はパスの各マルチキャスト・ルータでカウント・ダウンされます。TTL 値が 0 になると、データグラムはそれ以上転送されません。

次の例は、`setsockopt` システム・コールの `IP_MULTICAST_TTL` オプションの使用方法を示しています。

```
u_char ttl;  
ttl=2;
```

```

if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &tttl,
               sizeof(tttl)) == -1)
    perror("setsockopt");

```

IP マルチキャスト・デスティネーション宛のデータグラムは、アプリケーションで代替ネットワーク・インタフェースがソケットに関連付けられていることを指定していなければ、省略時のネットワーク・インタフェースから伝送されます。省略時のインタフェースは、カーネル・ルーティング・テーブルの省略時の経路に関連付けられているインタフェース、または、明示的な経路がある場合には、それに関連付けられているインタフェースによって決定されます。setsockopt システム・コールで IP_MULTICAST_IF オプションを使用すると、アプリケーションは、カーネル・ルーティング・テーブルの経路で指定されているネットワーク・インタフェース以外のものを指定することができます。

次の例は、setsockopt システム・コールの IP_MULTICAST_IF オプションを使用して、省略時のインタフェース以外のインタフェースを指定する方法を示しています。

```

int sock;
struct in_addr ifaddress;
char *if_to_use = "16.141.64.251";
:

ifaddress.s_addr = inet_addr(if_to_use);
if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, &ifaddress,
               sizeof(ifaddress)) == -1)
    perror ("error from setsockopt IP_MULTICAST_IF");
else
    printf ("new interface set for sending multicast datagrams\n");

```

送信側のホストが属しているグループへマルチキャスト・データグラムを送信する場合、省略時にはデータグラムのコピーが IP 層によりローカル引き渡しのためにループ・バックされます。setsockopt システム・コールの IP_MULTICAST_LOOP オプションを使用すると、アプリケーションはこのループバック引き渡しを使用不能にすることができます。

次の例は、setsockopt システム・コールの IP_MULTICAST_LOOP オプションの使用方法を示しています。

```

u_char loop=0;
if (setsockopt( sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop

```

```

        sizeof(loop)) == -1)
    perror("setsockopt");

```

`loop` の値が 0 の場合、ループバックは使用不能です。 `loop` の値が 1 の場合には使用可能です。性能上の理由により、同じホストのアプリケーションがデータグラムのコピーを受信する必要がある場合を除いて、省略時の動作 (使用可能) から使用不能に変更することをお勧めします。

4.7.6.2 IPv4 マルチキャスト・データグラムの受信

`all hosts group` 以外の特定のマルチキャスト・グループ宛の IP マルチキャスト・データグラムをホストが受信できるようにするために、アプリケーションはホストにそのマルチキャスト・グループのメンバになるように指示しなければなりません。この項では、アプリケーションがホストに指示して、マルチキャスト・グループに加わったり抜けたりする方法を説明します。

アプリケーションは、`setsockopt` システム・コールの `IP_ADD_MEMBERSHIP` オプションを使用すると、そのアプリケーションを実行しているホストにマルチキャスト・グループに加わるように指示することができます。

```

struct ip_mreq mreq;
if (setsockopt( sock, IPPROTO_IP, IP_ADD_MULTICAST, &mreq
               sizeof(mreq)) == -1)
    perror("setsockopt");

```

`mreq` 変数の構造体を次に示します。

```

struct ip_mreq{
    struct in_addr imr_multiaddr; /* IP multicast address of group */
    struct in_addr imr_interface; /* local IP address of interface */
};

```

各マルチキャスト・グループのメンバシップは、特定のインタフェースと関連付けられています。複数のインタフェース上にある同一のグループに加わることができます。 `imr_interface` 変数を `INADDR_ANY` と指定すると、アプリケーションは省略時のマルチキャスト・インタフェースを選択することができます。もう 1 つの方法として、ホストのローカル・アドレスのいずれかを指定すると、アプリケーションは特定のマルチキャスト可能なインタフェースを選択することができます。単一のソケットに加えることができるメンバシップの最大数は、`<netinet/in.h>` ヘッダ・ファイルに定義されている `IP_MAX_MEMBERSHIPS` の値によって決まります。

特定のマルチキャスト・グループのメンバシップを除くには、`setsockopt` システム・コールの `IP_DROP_MEMBERSHIP` オプションを使用します。

```

struct ip_mreq mreq;
if (setsockopt( sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq,
               sizeof(mreq)) == -1)
    perror("setsockopt");

```

`mreq` 変数には、メンバシップを加えるときと同じ構造体の値が入ります。

あるホストを特定のマルチキャスト・グループに加えることを複数のソケットが要求した場合には、それらのソケットのうちの最後のソケットがクローズされるまで、そのホストはマルチキャスト・グループのメンバのままになります。

特定の UDP ポートに送信されたマルチキャスト・データグラムを受信するには、受信側のソケットは `bind` システム・コールを使用してそのポートにバインドされていなければなりません。 `bind` システム・コール (4.3.2 項を参照) の前に `SO_REUSEPORT` オプションを指定する `setsockopt` システム・コールが発行されている場合には、同一のポート宛の UDP データグラムを複数のプロセスが受信できます。次の例は、`setsockopt` システム・コールで `SO_REUSEPORT` オプションを使用する方法を示しています。

```

int setreuse = 1;
if (setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &setreuse,
               sizeof(setreuse)) == -1)
    perror("setsockopt");

```

`SO_REUSEPORT` オプションが設定されている場合、共用ポート宛のすべての着信マルチキャストまたはブロードキャスト UDP データグラムは、そのポートにバインドされているすべてのソケットに引き渡されます。

IP マルチキャスト・データグラムの `SOCK_RAW` ソケットへの引き渡しは、デスティネーションのプロトコル・タイプによって決まります。

4.7.6.3 IPv6 マルチキャスト・データグラムの送信

IPv6 マルチキャスト・データグラムを送信するには、`sendto` システム・コールで IPv6 マルチキャスト・アドレスを指定することにより、送信先のマルチキャスト・グループを示します。システムはデータグラムを伝送する前に、指定された IPv6 デスティネーション・アドレスを適切なイーサネットまたは FDDI マルチキャスト・アドレスにマップします。

アプリケーションは、`setsockopt` システム・コールに引数を指定することにより、マルチキャスト・オプションを明示的に制御できます。次のオ

プションは、`setsockopt` システム・コールを使用して、アプリケーションで設定できます。

- ホップの限界値 (`IPV6_MULTICAST_HOPS`)
- マルチキャスト・インタフェース (`IPV6_MULTICAST_IF`)
- ローカル引き渡しのループバックの使用不能化 (`IPV6_MULTICAST_LOOP`)

注意

`setsockopt` システム・コールの構文および引数については、4.3.5 項および `setsockopt(2)` に説明があります。この項および 4.7.6.4 項の例は、IPv6 マルチキャスト・データグラムだけに適用される `setsockopt` オプションの使用方法を示しています。

`setsockopt` システム・コールの `IPV6_MULTICAST_HOPS` オプションを使用すると、アプリケーションはホップ限界値のフィールドに対して 0 ~ 255 の値を指定することができます。ホップ限界値が 0 のマルチキャスト・データグラムは、ローカル・ホスト上で実行しているアプリケーションへのマルチキャスト・データグラムの配信を制限します。ホップ限界値が 1 のマルチキャスト・データグラムは、ローカル・リンク上のホストのみに転送されます。マルチキャスト・データグラムのホップ限界値が 1 より大きく、送信側のホストのネットワークにマルチキャスト・ルータが接続されている場合には、マルチキャスト・データグラムをローカル・リンク外へ転送することができます。マルチキャスト・ルータは、指定されたマルチキャスト・グループに属するホストを持つ既知のネットワークにデータグラムを転送します。ホップ限界値はパスの各マルチキャスト・ルータでカウント・ダウンされます。ホップ限界値が 0 になると、データグラムはそれ以上転送されません。

次の例は、`setsockopt` システム・コールの `IPV6_MULTICAST_HOPS` オプションの使用方法を示しています。

```
int hops;
hops=2;

if (setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_HOPS, &hops,
               sizeof(hops)) < 0)
    perror("setsockopt: IPV6_MULTICAST_HOPS error");
```


IPv6 マルチキャスト・アドレス宛のデータグラムは、アプリケーションで代替ネットワーク・インタフェースがソケットに関連付けられていることを指定していなければ、省略時のネットワーク・インタフェースから伝送されます。省略時のインタフェースは、カーネル・ルーティング・テーブルの省略時の経路に関連付けられているインタフェース、または明示的な経路がある場合には、それに関連付けられているインタフェースによって決定されます。setsockopt システム・コールで IPV6_MULTICAST_IF オプションを使用すると、アプリケーションは、カーネル・ルーティング・テーブルの経路で指定されているネットワーク・インタフェース以外のものを指定することができます。

次の例は、setsockopt システム・コールの IPV6_MULTICAST_IF オプションを使用して、省略時のインタフェース以外のインタフェースを指定する方法を示しています。

```
u_int if_index = 1;
:
:

    if (setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_IF, &if_index,
                  sizeof(if_index)) < 0)
        perror ("setsockopt: IPV6_MULTICAST_IF error");
    else
        printf ("new interface set for sending multicast datagrams\n");
```

if_index パラメータは、必要なインタフェースのインタフェース・インデックスを指定するか、0 を指定して省略時のインタフェースを選択します。if_nametoindex(3) を使用すると、インタフェース・インデックスを見つけることができます。

送信側のノードが属しているグループへマルチキャスト・データグラムを送信する場合、省略時にはデータグラムのコピーが IP 層によりローカル引き渡しのためにループ・バックされます。setsockopt システム・コールの IPV6_MULTICAST_LOOP オプションを使用すると、アプリケーションはこのループバック引き渡しを使用不能にすることができます。

次の例は、setsockopt システム・コールの IPV6_MULTICAST_LOOP オプションの使用方法を示しています。

```
u_char loop=0;
if (setsockopt( sock, IPPROTO_IPV6, IPV6_MULTICAST_LOOP, &loop,
               sizeof(loop)) < 0)
    perror("setsockopt: IPV6_MULTICAST_LOOP error");
```

loop の値が 0 の場合、ループバックは使用不能です。loop の値が 1 の場合には使用可能です。性能上の理由により、同じホストのアプリケーションが

データグラムのコピーを受信する必要がある場合を除いて、省略時の動作 (使用可能) から使用不能に変更することをお勧めします。

4.7.6.4 IPv6 マルチキャスト・データグラムの受信

All Nodes グループ以外の特定のマルチキャスト・グループ宛の IPv6 マルチキャスト・データグラムをノードが受信できるようにするために、アプリケーションはホストにそのマルチキャスト・グループのメンバになるように指示しなければなりません。この項では、アプリケーションがノードに指示して、マルチキャスト・グループに加わったり抜けたりする方法を説明します。

アプリケーションは、`setsockopt` システム・コールの `IPV6_JOIN_GROUP` オプションを使用すると、そのアプリケーションを実行しているノードにマルチキャスト・グループに加わるように指示することができます。

```
struct ipv6_mreq imr6;
:

imr6.ipv6mr_interface = if_index;
if (setsockopt( sock, IPPROTO_IPV6, IPV6_JOIN_GROUP,
               (char *)&imr6, sizeof(imr6)) < 0)
    perror("setsockopt: IPV6_JOIN_GROUP error");
```

`imr6` 変数の構造体を次に示します。

```
struct ipv6_mreq {
    struct in6_addr ipv6mr_multiaddr; /*IP multicast address of
    group*/
    unsigned int ipv6mr_interface; /*local interface index*/
};
```

各マルチキャスト・グループのメンバシップは、特定のインタフェースと関連付けられています。複数のインタフェース上にある同一のグループに加わることができます。`ipv6mr_interface` 変数に値 0 を指定すると、アプリケーションは省略時のマルチキャスト・インタフェースを選択することができます。もう 1 つの方法として、ホストのローカル・インタフェースのいずれかを指定すると、アプリケーションは特定のマルチキャスト可能なインタフェースを選択することができます。`ipv6mr_multiaddr` 変数は、IPv6 マルチキャスト・アドレスまたは IPv4 射影 IPv6 アドレスとして指定できます。

単一のソケットに加えることができるメンバシップの最大数は、`<netinet/in.h>` ヘッダ・ファイルに定義されている `IPV6_MAX_MEMBERSHIPS` の値によって決まります。

特定のマルチキャスト・グループのメンバシップを除くには、`setsockopt` システム・コールの `IPV6_LEAVE_GROUP` オプションを使用します。

```
struct ipv6_mreq imr6;  
if (setsockopt( sock, IPPROTO_IPV6, IPV6_LEAVE_GROUP, &imr6,  
    sizeof(imr6)) < 0)  
    perror("setsockopt: IPV6_LEAVE_GROUP error");
```

`imr6` パラメータには、メンバシップを加えるときと同じ構造体の値が入ります。このアドレスは、IPv6 マルチキャスト・アドレス、またはIPv4 射影 IPv6 アドレスとすることができます。

あるノードを特定のマルチキャスト・グループに加えることを複数のソケットが要求した場合には、それらのソケットのうちの最後のソケットがクローズされるまで、そのノードはマルチキャスト・グループのメンバのままになります。

特定の UDP ポートに送信されたマルチキャスト・データグラムを受信するには、受信側のソケットは `bind` システム・コールを使用してそのポートにバインドされていなければなりません。 `bind` システム・コール (4.3.2 項を参照) の前に `SO_REUSEPORT` オプションを指定する `setsockopt` システム・コールが発行されている場合には、同一のポート宛の UDP データグラムを複数のプロセスが受信できます。このオプションの使用例については、4.7.6.2 項を参照してください。

IP マルチキャスト・データグラムの `SOCK_RAW` ソケットへの引き渡しは、デスティネーションのプロトコル・タイプによって決まります。

4.7.7 ブロードキャストとネットワーク構成の調査

データグラム・ソケットを使用すると、システムがサポートするさまざまなネットワーク上で、ブロードキャスト・パケットを送信することができます。ネットワーク自身はブロードキャストをサポートしなければなりませんが、システムでは、ソフトウェアでのブロードキャストのシミュレーションを提供していません。

ブロードキャスト・メッセージは高い負荷をネットワークに与えることになります。これは、ネットワーク上のすべてのホストがブロードキャストを処理しなければならないためです。そのため、ブロードキャスト・パケットを送信する能力は、明示的に許可されているソケットに限定されます。

ブロードキャストは、通常、次の 2 つのうちいずれかの理由で使用されます。つまり、リソースのアドレスが分からない状態でローカル・ネットワークのリソースを探すため、またはすべてのアクセス可能な他のホストに情報を送信するためです。

注意

ブロードキャストは、TCP のようなコネクション指向型トランスポートではサポートされていません。

ブロードキャスト・メッセージを送信するには、次の手順に従います。

1. データグラム・ソケットを作成する。

たとえば、次のように作成します。

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

2. ソケットをブロードキャスト用に指定する。

たとえば、次のようになります。

```
int    on = 1;

if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on,
               sizeof(on)) == -1)
    perror("setsockopt");
```

3. 少なくとも 1 つのポート番号を、必ずソケットにバインドする。

```
sin.sin_len = sizeof(sin);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
if (bind(s, (struct sockaddr *) &sin, sizeof (sin)) == -1)
    perror("setsockopt");
```

メッセージのデスティネーション・アドレスは、ブロードキャストされる先のネットワークによって異なります。インターネット・ドメインでは、ブロードキャストの簡略表記法をローカル・ネットワークでサポートします。そのアドレスは、`netinet/in.h` に定義されているように、`INADDR_BROADCAST` です。

すべてのホストに送信できるようにアドレスのリストを決定するには、ホストが接続するネットワークの情報がが必要です。オペレーティング・システムでは、システム・データ構造からのこの情報を検索する方法を提供していま

す。ioctl 呼び出しの SIOCGIFCONF は、ホスト側のインタフェースの構成を単一の ifconf 構造体の形で返します。この構造体は、ifreq 構造体の配列を収めたデータ領域を持ちます。ifreq 構造体は、ホストが接続する各ネットワーク・インタフェースに対して 1 つです。この構造体は、<net/if.h> ヘッダ・ファイルで次のように定義されています。

```
struct ifconf {
    int    ifc_len;                /* size of associated buffer */
    union {
        caddr_t ifcu_buf;
        struct ifreq *ifcu_req;
    } ifc_ifcu;
#define ifc_buf ifc_ifcu.ifcu_buf /* buffer address */
#define ifc_req ifc_ifcu.ifcu_req /* array of structures returned */
};

struct ifreq {
#define IFNAMSIZ 16
    char    ifr_name[IFNAMSIZ];    /* if name, e.g. "ee0" */
    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        struct sockaddr ifru_broadaddr;
        short    ifru_flags;
        int    ifru_metric;
        caddr_t ifru_data;
    } ifr_ifru;
#define ifr_addr ifr_ifru.ifru_addr /* address */
#define ifr_dstaddr ifr_ifru.ifru_dstaddr /* other end of */
/* p-to-p link */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr /* broadcast address */
#define ifr_flags ifr_ifru.ifru_flags /* flags */
#define ifr_metric ifr_ifru.ifru_metric /* metric */
#define ifr_data ifr_ifru.ifru_data /* for use by */
/* interface */
};
```

インタフェース構成を取得する実際の呼び出しは、次のようになります。

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof (buf);
ifc.ifc_buf = buf;
if (ioctl(s, SIOCGIFCONF, (char *) &ifc) < 0) {
    :
}
}
```

この呼び出しの後、ホストが接続する各ネットワークに対して 1 つの ifreq 構造体が、buf に設定されます。そして、ifc.ifc_len が、ifreq 構造体で使用するバイト数を反映するように変更されます。

各構造体には、インタフェース・フラグのセットがあり、そのフラグに対応するネットワークが起動しているか停止しているか、二地点間またはブ

ロードキャストか、などを示します。 `ioctl` の `SIOCGIFFLAGS` は、フラグを検索して、`ifreq` 構造体で指定されるインタフェースがないかどうかを、次のように調べます。

```
struct ifreq *ifr;

ifr = ifc.ifc_req;

for (n = ifc.ifc_len / sizeof (struct ifreq); --n >= 0; ifr++) {
    /*
     * We must be careful that we don't use an interface
     * devoted to an address family other than those intended.
     */
    if (ifr->ifr_addr.sa_family != AF_INET)
        continue;
    if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
        :
    }
    /*
     * Skip irrelevant cases.
     */
    if ((ifr->ifr_flags & IFF_UP) == 0 ||
        (ifr->ifr_flags & IFF_LOOPBACK) ||
        (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTOPOINT)) == 0)
        continue;
}
```

フラグを取得すると、ブロードキャスト・アドレスも取得しなければなりません。ブロードキャスト・ネットワークの場合、これは `ioctl` の `SIOCGIFBRDADDR` を介して行われます。二地点間ネットワークでは、デスティネーション・ホストのアドレスは、`SIOCGIFBRDADDR` を使用して取得します。たとえば、次のように使用します。

```
struct sockaddr dst;

if (ifr->ifr_flags & IFF_POINTOPOINT) {
    if (ioctl(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
        ...
    }
    bcopy((char *) ifr->ifr_dstaddr, (char *) &dst,
          sizeof (ifr->ifr_dstaddr));
} else if (ifr->ifr_flags & IFF_BROADCAST) {
    if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
        ...
    }
    bcopy((char *) ifr->ifr_broadaddr, (char *) &dst,
          sizeof (ifr->ifr_broadaddr));
}
```

適切な `ioctl` システム・コールにより、ブロードキャスト・アドレスまたはデスティネーション・アドレス(`dst` に格納される)を取得した後に、`sendto` システム・コールを使用します。たとえば、次のように使用します。

```
if (sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof (dst)) < 0)
    perror("sendto");
```

前述のループでは、ホストが接続し、ブロードキャストおよび二地点間のアドレス指定をサポートするすべてのインタフェースに対して、`sendto` システム・コールが一度実行されます。ブロードキャスト・メッセージを所定のネットワークに転送するだけのプロセスの場合は、前述の例で示したようにコーディングします。ただし、適切なデスティネーション・アドレスを見つけるループ処理が必要になります。

4.7.8 inetd デーモン

オペレーティング・システムでは `inetd` インターネット・スーパーバ・デーモンをサポートしています。この `inetd` デーモンは、ブート時に呼び出され、`/etc/inetd.conf` ファイルを読み込んでリッスンするサーバを決定します。

注意

Tru64 UNIX では、ソケット上で実行するように作成されているサーバ・アプリケーションのみが `inted` デーモンを使用できます。Tru64 UNIX の `inetd` デーモンは、STREAMS、XTI、または TLI で実行するサーバ・アプリケーションをサポートしていません。

`/etc/inetd.conf` ファイルにリストされている各サーバに対して、`inetd` デーモンは次の処理を行います。

1. ソケットを作成して、適切なポート番号をバインドする。
2. `select` システム・コールを発行して読み取りを可能にし、そのソケットに対応するサービスに接続を要求するプロセスを待つ。
3. `accept` システム・コールを発行してフォークし、`dup` 呼び出しで新しいソケットをファイル記述子 0 および 1 (`stdin` と `stdout`) に複写する。さらに、他のオープンしているファイル記述子をクローズし、`exec` 呼び出しで適切なサーバを実行する。

`inetd` を使用するサーバでは、処理が簡略化されています。これは、`inetd` が接続の確立に必要なプロセス間通信処理のほとんどを行っているためで

す。inetd によって呼び出されたサーバは、ソケットがファイル記述子 0 と 1 でクライアントに接続されているものと解釈して、ただちに read , write , send , recv などのオペレーションを実行します。

inetd デーモンによって呼び出されたサーバは、<stdio.h> ヘッダ・ファイルにある規約に従い、必要なときに fflush システム・コールを呼び出すという条件付きで、バッファ利用入出力を使用します。詳細は fflush(3) を参照してください。

getpeername システム・コールは、ソケットのもう一方に接続された対等プロセスのアドレスを返します。このシステム・コールは、inetd を使用するサーバ・アプリケーションを作成するプログラマにとって役に立ちます。次のコード例は、inetd を使用してサーバに接続するクライアントのインターネット・アドレス(ドット表記法)のログを取る方法を示しています。

```
struct sockaddr_storage name;
size_t namelen = sizeof (name);
:
:

if (getpeername(0, (struct sockaddr *)&name, &namelen) < 0) {
    syslog(LOG_ERR, "getpeername: %m");
    exit(1);
} else
    syslog(LOG_INFO, "Connection from %s", inet_ntoa(name.sin_addr));
:
:
```

getpeername システム・コールは、inetd を使用するプログラムを作成する場合には特に有効ですが、それ以外の場合にも使用できます。

4.7.9 入出力の多重化

多重化とは、複数のソケット間で入出力要求を伝送したり受信したりするアプリケーションに使用される機能です。これは、次のように select システム・コールを使用して行うことができます。

```
#include <sys/time.h>
#include <sys/types.h>
:
:
:
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
:
:
:
if (select(nfds, &readmask, &writemask, &exceptmask, &timeout) < 0)
    perror("select");
```


`select` システム・コールでは、引数として次の 3 セットのポインタをとります。

1. 呼び出しているアプリケーションが読み取るデータのためのソケット記述子のセット
2. データを書き込むソケット記述子
3. 保留されている例外的な条件

アプリケーションが読み取り、書き込み、例外など、特定の条件にとらわれない場合、`select` システム・コールの対応する引数は空ポインタにしなければなりません。

注意

XTI と TLI は STREAMS を使用してインプリメントされているので、STREAMS ファイル記述子についてはすべて、`select` システム・コールではなく `poll` システム・コールを使用してください。

各セットは、実際には、整数のビット・マスクの配列を含む構造体です。配列のサイズは `FD_SETSIZE` の定義によって設定されます。配列は、`FD_SETSIZE` の各ファイル記述子に対して 1 ビットずつの長さが必要です。

`mask` セットの `fd` ファイル記述子を追加したり削除したりするために、`FD_SET(fd, &mask)` および `FD_CLR(fd, &mask)` マクロが提供されています。セットは使用する前にゼロに初期化する必要があります。`mask` セットをクリアにするために、`FD_ZERO(&mask)` マクロが提供されています。

`select` システム・コールの `nfds` パラメータを使用して、1 セット中でチェックするファイル記述子の範囲、たとえば最大の記述子に 1 をプラスした値などを指定します。

タイムアウト値は、`select` システム・コールが事前に設定した時間を超えて作動しないように指定するためのものです。`timeout` のフィールドがゼロ (0) に設定されている場合、`select` システム・コールはポーリングと同様に動作して、ただちに帰ってきます。最後のパラメータが空ポインタの場合、`select` システム・コールは、無期限にブロックします。特に、記述子が選

択可能な場合、またはシグナルがシステム・コールに割り込んで呼び出し側に受信された場合に限り、`select` が戻ります。

通常、`select` システム・コールは選択した記述子ファイルの数を返します。タイムアウト時間が満了になったため、`select` システム・コールが戻る場合には、0 が返されます。エラーや割り込みのために `select` システム・コールが終了した場合には、`errno` にエラー番号が設定され、ファイル記述子のマスクが変更されないまま、-1 が返されます。

`select` システム・コールが正常に戻った場合、前述の 3 セットは、書き込みや読み取りに使用できるファイル記述子、または保留中の例外条件を持っているファイル記述子を示しています。`select` システム・コールのマスクにあるファイル記述子の状態は、`FD_ISSET(fd, &mask)` マクロを使用して確認することができます。このマクロは、*fd* が *mask* セットのメンバである場合は非ゼロを、メンバでない場合は 0 を返します。

`accept` システム・コールで使用されているソケットに待機している接続があるかどうかを確認するには、`FD_ISSET(fd, &mask)` マクロを使用して、適当なソケット上で読み取りの準備ができていることを確認した後、`select` システム・コールを使用します。`FD_ISSET` が非ゼロを返して、読み取るデータがあることを示した場合、接続はそのソケット上で待機していることになります。

注意

4.2BSD では、`select` システム・コールの引数は、`fd_set` へのポインタではなく、整数へのポインタでした。この種のシステム・コールは、`select` システム・コールでチェックされるファイル記述子の数が整数値のビットより少ない場合に限り動作しますが、次のコードで示す方法をお勧めします。

次の例は、1 秒間のタイムアウトを指定して、ソケット `s1` および `s2` のデータが利用可能になったとき、アプリケーションでデータを読み取る方法を示しています。

```
#include <sys/time.h>
#include <sys/types.h>
:
:
fd_set read_template;
```

```

struct timeval wait;
:

for (;;) {
    wait.tv_sec = 1;    /* one second */
    wait.tv_usec = 0;

    FD_ZERO(&read_template);

    FD_SET(s1, &read_template);
    FD_SET(s2, &read_template);

    nb = select(FD_SETSIZE, &read_template, (fd_set *) 0,
                (fd_set *) 0, &wait);
    if (nb <= 0) {
        An error occurred during the select, or
        the select timed out    }

    if (FD_ISSET(s1, &read_template)) {
        Socket #1 is ready to be read from.
    }

    if (FD_ISSET(s2, &read_template)) {
        Socket #2 is ready to be read from.
    }
}

```

select システム・コールは、同期多重化機構を提供します。4.7.11 項で説明している SIGIO および SIGURG シグナルを使用すると、出力の終了、入力の可能、および例外条件を非同期に通知することができます。

4.7.10 割り込み駆動のソケット I/O

ソケット (通常は、ファイル記述子) に読み取るデータがあるとき、SIGIO シグナルを使用すれば、それをプロセスにシグナル通知することができます。SIGIO 機能を使用するには、次の 3 つの手順を実行しなければなりません。

1. signal または sigvec 呼び出しを使用して、プロセスで SIGIO シグナル・ハンドラを起動する。
2. 保留状態の入力の通知を受信するプロセス ID、またはプロセス・グループ ID を、そのプロセス自身のプロセス ID またはプロセス・グループのプロセス・グループ ID に設定する。

ソケットの省略時のプロセス・グループはグループ0であることに注意してください。これらの設定には、`fcntl` システム・コールを使用します。

3. 別の `fcntl` システム・コールで、保留中の入出力要求を非同期に通知できるようにする。

次のコードは、ソケット `s` で入出力処理が保留になったときに、特定のプロセスでその情報を受信できるようにする例です。SIGURG ハンドラを追加すれば、次のようなコードを使用して、SIGURG シグナルを受信することができます。

```
#include <fcntl.h>
:
:
int    io_handler();
:

signal(SIGIO, io_handler);

/* Set the process receiving SIGIO/SIGURG signals to us */

if (fcntl(s, F_SETOWN, getpid()) < 0) {
    perror("fcntl F_SETOWN");
    exit(1);
}

/* Allow receipt of asynchronous I/O signals */

if (fcntl(s, F_SETFL, FASYNC) < 0) {
    perror("fcntl F_SETFL, FASYNC");
    exit(1);
}
```

4.7.11 シグナルおよびプロセス・グループ

各ソケットには対応するプロセス番号があり、この番号の値は、ゼロ (0) に初期化されます。この番号は、4.7.10 項で行ったように、`fcntl` システム・コールの `F_SETOWN` パラメータを使用して再定義し、SIGURG および SIGIO シグナルが受信できるようにしなければなりません。シグナル用にソケットのプロセスIDを設定するには、`fcntl` システム・コールに正の引数を指定しなければなりません。シグナル用にソケットのプロセス・グループを設定するには、`fcntl` システム・コールに負の引数を渡さなければなりません。

ん。プロセス番号は、対応するプロセス ID または対応するプロセス・グループを示しています。両方を同時に指定することはできません。

fcntl システム・コールの F_GETOWN パラメータを使用すると、ソケットの現在プロセス番号を取得することができます。

サーバ・プロセスの作成時には、SIGCHLD シグナルも役立ちます。このシグナルは、いずれかの子プロセスの状態が変更されると、プロセスに送信されます。通常、サーバは、明示的に終了を待ったり、終了状態の定期的なポーリングを行わずに、SIGCHLD シグナルを使用して終了した子プロセスを呼び出します。親サーバ・プロセスが子プロセスを呼び出せない場合には、多数のゾンビ・プロセスが作成されます。次のコードは、SIGCHLD シグナルの使用方法を示しています。

```
int reaper();
:

signal(SIGCHLD, reaper);
listen(f, 5);
for (;;) {
    int g;
    size_t len = sizeof (from);

    g = accept(f, (struct sockaddr *)&from, &len);
    if (g < 0) {
        if (errno != EINTR)
            syslog(LOG_ERR, "rlogind: accept: %m");
        continue;
    }
    :
}
:

#include <wait.h>
reaper()
{
    union wait status;

    while (wait3(&status, WNOHANG, 0) > 0)
        ;
}
```

4.7.12 擬似端末

ほとんどのプログラムは、標準入出力の端末がなければ正しく機能することができません。ソケットが端末の意味を提供しないため、ネットワーク上で通信しているプロセスは、擬似端末 (pty) を介して端末の意味を提供する必要があります。擬似端末とは、マスタとスレーブの 1 対のデバイスであり、アプリケーションとユーザ間の通信において、プロセスをアクティブ・エージェントとして使用することができます。

擬似端末のスレーブ側に書き込まれたデータは、マスタ側から読み込まれたプロセスへの入力として使用され、マスタ側に書き込まれたデータは、スレーブの端末入力として処理されます。このように、実際の端末でキーボードを操作したりスクリーンを読み取ったりするかのようになり、擬似端末のマスタ側を操作するプロセスは、スレーブ側で読み取ったり書き込んだ情報を制御します。擬似端末抽象表現の目的は、ネットワーク接続を通して端末の意味を保持することです。つまり、スレーブに対して読み込みや書き込みを行うプロセスで、スレーブ側が通常の端末に見えるようにすることです。

たとえば、`rlogind` リモート・ログイン・サーバは、擬似端末をリモート・ログイン・セッションに使用します。ネットワーク上でマシンにログインしているユーザには、標準入力、標準出力、および標準エラーとしてシェルにスレーブ擬似端末が提供されます。すると、サーバ・プロセスは、リモート・シェルから呼び出されたプログラムとユーザのローカル・クライアント・プロセスとの間の通信を処理します。ユーザが、端末出力をフラッシュするリモート・マシン上に割り込みを生成する文字を送信する場合には、擬似端末はサーバ・プロセスの制御メッセージを生成します。この場合、サーバは帯域外メッセージをクライアント・プロセスに送信して、実際の端末およびネットワークのバッファに格納された介入データ上でのデータのフラッシュをシグナル通知します。

オペレーティング・システムでは、擬似端末のスレーブ側には `/dev/ttyxy` 形式の名前があります。 `x` は `d` 以外の任意の 1 文字で、大文字または小文字も使えます。 `y` は 16 進数で、`0~9` または `a~f` の 1 文字です。擬似端末のマスタ側には、`/dev/ptyxy` 形式の名前があります。 `x` と `y` は、擬似端末のスレーブ側の `x` と `y` に対応します。

`openpty` 関数および `forkpty` 関数が `libc.a` ライブラリに追加されて、擬似端末の割り当てが容易になりました。これらの関数は、`clone open` 呼び出しを使用して、複数の `open` 呼び出しが実行されないようにします。

`forkpty` 関数は擬似端末を割り当てます。また、子プロセスをフォークして、スレーブ擬似端末を子プロセスの制御端末にします。スレーブのファイル記述子は呼び出しプロセスに渡されないため、`forkpty` 関数は 5 つではなく 4 つの引数をとります。代わりに、スレーブのファイル記述子は、新規に作成された子プロセスで、`stdin`、`stdout`、および `stderr` として複写されます。その他の 4 つの引数は、`openpty` 関数の引数と同じです。

`openpty` 関数および `forkpty` 関数は、`--1` を返してエラー状態を示します。`openpty` 関数は、正常終了するとゼロ (0) を返しますが、`forkpty` 関数は、子プロセスの `pid` (プロセス ID) を返します。関数の構文、パラメータ、エラーについては、`openpty(3)` を参照してください。

`openpty` 関数は、次のように動作します。

1. 正常終了すると、擬似端末のスレーブ側が適切な端末モードに設定される。
擬似端末のマスタ側とスレーブ側がオープンされるとき、オペレーティング・システムは必要なセキュリティ検査を実行します。
2. プロセスがフォークする。
子プロセスは擬似端末のマスタ側をクローズして、適切なプログラムを (`exec` 呼び出しを使用して) 実行します。
3. 親プロセスは擬似端末のスレーブ側をクローズして、マスタ側からの読み込みおよび書き込みを開始する。

次の例では、擬似端末を使用しています。このコード例では、次のように仮定しています。

- ソケットの接続が存在する。
- ソケットは、特定のサービスを要求する対等プロセスに接続されている。
- プロセスは、以前の制御端末からの関連をすべて解除されている。

```
if (openpty(&mast,&slave,NULL,NULL,NULL) {
    syslog(LOG_ERR, "All network ports in use");
    exit(1);
}
ioctl(slave, TIOCGETA, &term); /* get default slave termios struct */
term.c_iflag |= ICRNL;
term.c_oflag |= OCRNL;
ioctl(slave, TIOCSSETA, &term); /* set slave characteristics */
i = fork();
if (i < 0) {
    syslog(LOG_ERR, "fork: %m");
}
```

```
        exit(1);
    } else if (i) {          /* Parent */
        close(slave);
        :
    } else {                 /* Child */
        (void) close(s);
        (void) close(master);
        dup2(slave, 0);
        dup2(slave, 1);
        dup2(slave, 2);
        if (slave > 2)
            (void) close(slave);
        :
    }
}
```

ソケットの使用方法についての詳細は、4.3 節を参照してください。

Tru64 UNIX STREAMS

オペレーティング・システムは、AT&T の System V によって指定された STREAMS フレームワーク、STREAMS のバージョン 4.0 リリースを提供します。このフレームワークでは、従来の UNIX の文字入出力 (I/O) の代わりに、ユーザは、I/O 関数をモジュール方式でインプリメントできます。モジュール方式で開発された I/O 関数を使用すると、アプリケーションは、通信サービスを容易に構築したり再構成したりできます。

「STREAMS」はフレームワーク全体を指し、「ストリーム」は、アプリケーション・プログラムが open システム・コールを使用して作成したエンティティを指すことに注意してください。

この章では、次の項目について説明します。

- STREAMS フレームワークの概要
- STREAMS に対するアプリケーション・インタフェース
- カーネル・レベル関数
- モジュールまたはドライバの構成方法
- Tru64 UNIX 同期機構
- デバイス特殊ファイルの作成方法
- エラーおよびイベントのロギング
- STREAMS リファレンス・ページに関する情報

この章では、Tru64 UNIX の STREAMS のインプリメンテーションと、AT&T System V バージョン 4.0 のインプリメンテーションの相違について、詳細に説明します。Tru64 UNIX のインプリメンテーションが、AT&T のインプリメンテーションと大きく変わらないところについては、AT&T の該当するマニュアルに関する情報を提示します。

この章では、STREAMS フレームワークを使用してプログラムを作成する方法については説明しません。プログラミング情報についての詳細は、『*Programmer's Guide: STREAMS*』を参照してください。

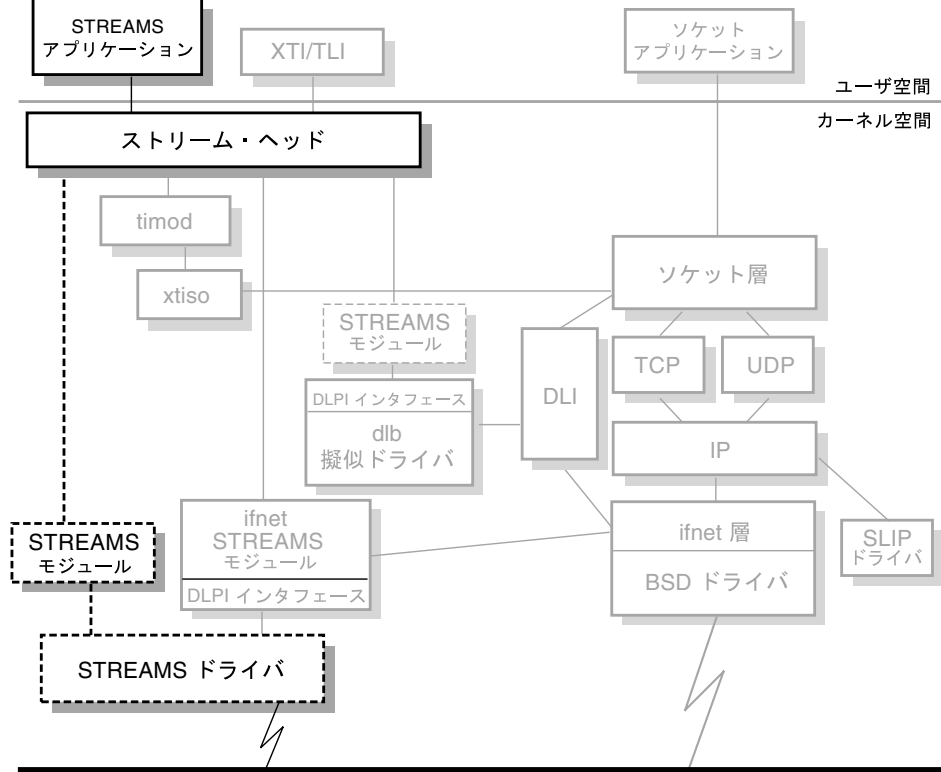
5.1 STREAMS フレームワークの概要

STREAMS フレームワークは、次のものから構成されます。

- アプリケーション・プログラムが STREAMS フレームワークへのアクセスに使用する、プログラミング・インタフェース、またはシステム・コールのセット。
- ストリーム・ヘッドなどのカーネル・リソース、およびストリームが使用するキューのデータ構造体。
- ストリームのキュー・スケジューリングやフロー制御、メモリ割り当て、およびエラー・ロギングなどのタスクを処理する、カーネル・ユーティリティ。

図 5-1 では、STREAMS フレームワークを強調表示して、ネットワーク・プログラミング環境における位置付けを示しています。

図 5-1: STREAMS フレームワーク



ZK-0559U-AIJ

5.1.1 STREAMS の構成要素の概説

STREAMS を使用して通信するには、アプリケーションでストリームを作成します。ストリームは、ユーザ・プロセスとデバイス・ドライバの間の全二重の通信パスです。ストリーム自体はカーネル・デバイスであり、アプリケーションに対しては文字型特殊ファイルとして表されます。他の文字型特殊ファイルと同様に、ストリームはオープンしなければなりません。それ以外の場合は、システム・コールによって操作します。

各ストリームの先頭と末尾には、それぞれ1つ以上のストリーム・ヘッドとストリーム・エンドがあります。ストリームを通して引き渡されるデータの処理に必要な場合には、リンクしたキューのペアから構成される追加モジュールを、ストリーム・ヘッドとストリーム・エンドの間に挿入できます。データは、モジュール間でメッセージを使用して引き渡されます。

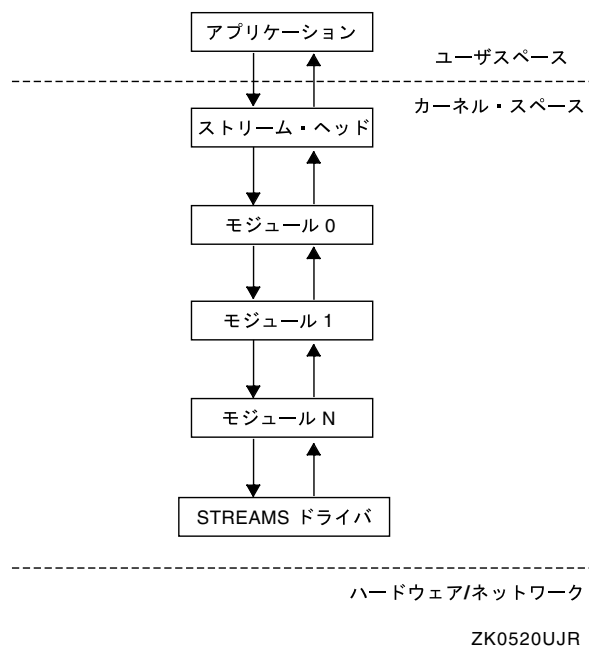
この項では、次の STREAMS の構成要素について簡単に説明します。

- ストリーム・ヘッド
- ストリーム・エンド
- モジュール

また、メッセージについて説明するとともに、STREAMS フレームワーク内におけるメッセージの役割についても説明します。

図 5-2 は、一般的なストリームを示しています。ストリーム・ヘッドからストリーム・エンド (図 5-2 では STREAMS ドライバ) に流れるデータは、ダウンストリーム、または書き込み方向に流れるといいます。ストリーム・エンドからストリーム・ヘッドに流れるデータは、アップストリーム、または読み取り方向に流れるといいます。

図 5-2: ストリームの例



ストリーム・ヘッドは、カーネル内においてユーザ・プロセスとストリームの間にインタフェースを提供する、ルーチンとデータ構造体のセットです。アプリケーションが `open` システム・コールを発行すると、ストリームが作成されます。ストリーム・ヘッドが実行する主なタスクは、次のとおりです。

1. STREAMS システム・コールの `write` や `putmsg` といった標準サブセットを解釈する。

2. ユーザ空間からの STREAMS システム・コールを、STREAMS メッセージ (M_PROTO, M_DATA など) の標準の範囲に変換する。

STREAMS メッセージは、データおよび制御情報から構成されます。

3. メッセージを次のモジュールヘダウンストリームに送信する。

メッセージは、最終的にストリーム・エンド、つまりドライバに到達します。

4. ドライバからアップストリームに送信されたメッセージを受信して、カーネル空間からの STREAMS メッセージを、アプリケーションによって呼び出されるシステム・コール (getmsg, read など) に適したフォーマットに変換する。

フォーマットは、システム・コールによって異なります。

ストリーム・エンドは、STREAMS モジュールの特殊な形式であり、ハードウェア・デバイス・ドライバまたは擬似デバイス・ドライバのいずれかです。ハードウェア・デバイス・ドライバの場合、ストリーム・エンドは、カーネルと外部通信デバイスとの間の通信を提供します。擬似デバイス・ドライバの場合には、ストリーム・エンドはソフトウェアでインプリメントされ、外部デバイスには関連しません。ハードウェア・デバイス・ドライバ、または擬似デバイス・ドライバのいずれであるかにかかわらず、ストリーム・エンドは、その上のモジュールから送信されたメッセージを受信し、解釈して、要求されたオペレーションを実行します。次に、ストリーム・ヘッドに向かってアップストリームに送信する適切なメッセージ・タイプのメッセージを作成して、アプリケーションにデータおよび制御情報を返します。

ドライバは、次の点を除いて、他の STREAMS モジュールと同じです。

- (必要がなくても) 割り込みを処理できる。

デバイス・ドライバには、1 つ以上の割り込みルーチンがあります。割り込みルーチンは、後で処理を行うために、読み取り側のサービス・ルーチンに関するデータをキューにイれる必要があります。

- 複数のストリームに接続できる。

ドライバは、多重化デバイスとしてインプリメントできます。つまり、アップストリームまたはダウンストリームのいずれかの方向で、複数のストリームに接続できます。詳細は、『*Programmer's Guide: STREAMS*』を参照してください。

- `open` および `close` の各システム・コールによって、初期化および初期化解除される。

他のモジュールは、`ioctl` システム・コールの `I_PUSH` コマンドおよび `I_POP` コマンドを使用します。

デバイス・ドライバおよびデバイス・ドライバ・ルーチンについての詳細は、『*Writing Device Drivers*』および『*Programmer's Guide: STREAMS*』を参照してください。

モジュールは、データを、ストリーム・ヘッドからストリーム・エンドに引き渡す際と、戻す際に処理します。ストリームは、データが要求する処理の量と種類に応じて、ゼロ個以上のモジュールを持つことができます。ドライバが、データに対して必要なすべての処理を実行できる場合には、追加のモジュールは必要ありません。

モジュールは、データをいれる 1 組のキュー、および各モジュールの役割を定義する他の構造体へのポインタで構成されます。1 つのキューは、ドライバに向かってダウンストリームに流れるデータを処理し、もう 1 つのキューは、ストリーム・ヘッドおよびアプリケーションに向かってアップストリームに流れるデータを処理します。ポインタは、各モジュールのダウンストリーム・キューおよびアップストリーム・キューを、次のモジュールのダウンストリーム・キューおよびアップストリーム・キューにリンクします。

処理の必要条件に応じて、アプリケーションは、特定のモジュールをストリームにプッシュするように要求します。ストリーム・ヘッドは、アプリケーションが要求したモジュールをアセンブルした後、モジュールのパイプラインを通してメッセージの経路指定を行います。

情報は、メッセージを使用して、モジュールからモジュールに引き渡されます。STREAMS 環境において、さまざまなタイプのメッセージが定義されますが、すべてのメッセージのタイプは、次のカテゴリに分類できます。

- 通常メッセージ
- 優先メッセージ

`M_DATA` および `M_IOCTL` などの通常メッセージは、受信した順に処理されて、STREAMS フロー制御およびキュー登録機構に従います。優先メッセージは、ストリームを通して優先的に引き渡されます。

メッセージおよびメッセージ・データ構造体についての詳細は、5.3.2 項を参照してください。

5.1.2 ioctl プロセス

STREAMS では、ユーザ・プロセスは、ストリーム内で `ioctl` を呼び出して、特定のモジュールとドライバを制御することができます。ユーザ・プロセスが `ioctl` コマンドを実行する際に、STREAMS はプロセスをブロックし、強制的にストリーム・ヘッドがコマンドを処理するようにします。また、必要に応じてメッセージ (M_IOCTL) を書き込み方向に送信して、特定のモジュールまたはドライバが受信して処理するようにします。ユーザ・プロセスは、次のいずれかが行われるまでブロックされます。

- モジュールまたはドライバが肯定応答 (M_IOCACK) または否定応答 (M_IOCNAK) で応答した場合
- 要求がタイムアウトした場合 (受信メッセージなし)
- ユーザ・プロセスが `ioctl` に割り込んだ場合
- エラー条件が発生した場合

STREAMS には、`ioctl` を処理するメソッドとして、`I_STR` と透過の 2 つがあります。表 5-1 で、その 2 つのメソッドの比較をしています。

表 5-1: `ioctl` 処理の `I_STR` メソッドと透過メソッドの比較

I_STR 処理	透過処理
STREAMS ファイルに対して作成されたアプリケーションのみをサポートする。	STREAMS ファイルか、非 STREAMS ファイルのいずれに対してでも作成されたアプリケーションをサポートする。
<code>streamio(7)</code> に記述されているコマンド群が利用できる。	<code>ioctl(2)</code> に記述されているコマンド群が利用できる。
データ・フォーマットおよびアドレッシングに制限がある。	データ・フォーマットおよびアドレッシングの制限は <code>ioctl</code> に依存する。
<code>ioctl</code> の処理を完了するには、1 対のメッセージが必要である。	<code>ioctl</code> の処理を完了するために、複数対のメッセージが必要になることがある。
タイムアウトの値はユーザ定義または省略時の値。	タイムアウトはしない。
STREAMS 処理を前提とする。	もっと汎用的。

`ioctl` 処理に対する両方のメソッドについての詳細は、『*Programmer's Guide: STREAMS*』を参照してください。

5.2 STREAMS に対するアプリケーション・インタフェース

STREAMS フレームワークに対するアプリケーション・インタフェースによって、STREAMS メッセージは、アプリケーションで送受信できます。以降の項で、STREAMS ヘッダ・ファイルおよびデータ型へのポインタを含むアプリケーション・インタフェースについて説明するとともに、STREAMS および STREAMS 関連のシステム・コールについても説明します。

5.2.1 ヘッダ・ファイルおよびデータ型

基本的な STREAMS のデータ型の定義は、次のヘッダ・ファイルに含まれています。

- `<sys/stream.h>`

このヘッダ・ファイルは、すべてのモジュールおよびストリーム・アプリケーションにインクルードしなければなりません。

- `<stropts.h>`

このヘッダ・ファイルは、アプリケーションで `ioctl` システム・コールを使用する場合に、インクルードしなければなりません。

- `<strlog.h>`

このヘッダ・ファイルは、アプリケーションで、STREAMS のエラー・ロガーおよびトレース機能を使用する場合に、インクルードしなければなりません。

注意

一般に、ヘッダ・ファイル名は山カッコ (< >) で囲まれています。ヘッダ・ファイルの絶対パスは、山カッコ内の情報の前に `/usr/include/` を付けたものです。`<sys/stream.h>` の場合、`stream.h` は `/usr/include/sys` ディレクトリに存在します。

5.2.2 STREAMS 関数

アプリケーションは、次の関数を使用して、STREAMS のカーネル・リソースにアクセスして、操作します。

- `open`
- `close`
- `read`
- `write`
- `ioctl`
- `mkfifo`
- `pipe`
- `putmsg` および `putpmsg`
- `getmsg` および `getpmsg`
- `poll`
- `isastream`
- `fattach`
- `fdetach`

この項では、これらの関数について簡単に説明します。これらの関数についての詳細は、リファレンス・ページおよび『*Programmer's Guide: STREAMS*』を参照してください。

5.2.2.1 `open` 関数

`open` 関数は、ストリームをオープンするときに使用します。

関数の構文、パラメータ、エラーについては、`open(2)` を参照してください。

次の例は、`open` 関数の使用方法を示しています。

```
int fd;  
fd = open("/dev/streams/echo", O_RDWR);
```

5.2.2.2 `close` 関数

`close` 関数は、ストリームをクローズするときに使用します。

関数の構文，パラメータ，エラーについては，`close(2)` を参照してください。

ストリームに対する最後の `close` によって，ファイル記述子に関連付けられたストリームが破壊されます。ストリームの破壊には，ストリーム上のすべてのモジュールのポップ，およびドライバのクローズが含まれます。

5.2.2.3 read 関数

`read` 関数は，ストリーム・ヘッドで待機中の `M_DATA` メッセージの内容を受信するときに使用します。

関数の構文，パラメータ，エラーについては，`read(2)` を参照してください。

`read` 関数は，`M_DATA` 以外のメッセージ・タイプでは失敗し，`errno` に `EBADMSG` がセットされます。

5.2.2.4 write 関数

`write` 関数は，データ・バッファから 1 つ以上の `M_DATA` メッセージを作成するときに使用します。

関数の構文，パラメータ，エラーについては，`write(2)` を参照してください。

5.2.2.5 ioctl 関数

`ioctl` 関数は，ストリーム上でさまざまな制御機能を実行するときに使用します。

`STREAMS` 関数の構文，パラメータ，エラーについては，`streamio(7)` を参照してください。

次の例は，`ioctl` システム・コールの使用法を示しています。

```
int fd;
fd = open("/dev/streams/echo", O_RDWR, 0);
ioctl(fd, I_PUSH, "pass");
```

5.2.2.6 mkfifo 関数

`STREAMS` ベースの `mkfifo` 関数は，単方向の `STREAMS` ベースのファイル記述子を作成するときに使用します。

注意

libc ライブラリの省略時の `mkfifo` 関数は STREAMS ベースではありません。STREAMS ベースの `mkfifo` 関数を使用するには、アプリケーションを `sys5` ライブラリとリンクしなければなりません。関数の構文、パラメータ、エラーについては、`mkfifo(2)` を参照してください。

また、`mkfifo` 関数では、FFM_FS (File on File Mount File System) カーネル・オプションが構成されていなければならないことにも注意してください。カーネル・オプションの構成についての詳細は、『システム管理ガイド』を参照してください。

5.2.2.7 pipe 関数

STREAMS ベースの `pipe` 関数は、双方向の STREAMS ベースの通信チャネルを作成するときに使用します。STREAMS ベースのパイプと STREAMS ベースでないパイプは、次の点が異なります。

- STREAMS ベースでないパイプは単方向である。
- `streamio` や `putmsg` などの STREAMS のオペレーションは、STREAMS ベースでないパイプでは実行できない。

注意

libc ライブラリの省略時の `pipe` 関数は STREAMS ベースではありません。STREAMS ベースの `pipe` 関数を使用するには、アプリケーションを `sys5` ライブラリとリンクしなければなりません。関数の構文、パラメータ、エラーについては、`pipe(2)` を参照してください。

5.2.2.8 putmsg および putpmsg 関数

`putmsg` および `putpmsg` 関数は、指定されたバッファからの情報を使用して、STREAMS メッセージ・ブロックを生成するときに使用します。

関数の構文、パラメータ、エラーについては、`putmsg(2)` を参照してください。

`putpmsg` 関数は、優先帯域データをダウンストリームに送信する場合に使用します。

各引数は `putmsg` 関数の引数と同じ意味です。

関数の構文、パラメータ、エラーについては、`putpmsg(2)` を参照してください。

5.2.2.9 `getmsg` および `getpmsg` 関数

`getmsg` および `getpmsg` 関数は、ストリーム・ヘッ드의読み取りキューにあるメッセージの内容を検索して、ユーザ指定のバッファにいれるときに使用します。

関数の構文、パラメータ、エラーについては、`getmsg(2)` を参照してください。

`getpmsg` 関数は、優先帯域データをストリームから受信する場合に使用します。

引数は `getmsg` 関数の引数と同じ意味です。

関数の構文、パラメータ、エラーについては、`getpmsg(2)` を参照してください。

5.2.2.10 `poll` 関数

`poll` 関数は、ユーザがデータを送受信できるストリームを識別するときに使用します。

関数の構文、パラメータ、エラーについては、`poll(2)` を参照してください。

5.2.2.11 `isastream` 関数

`isastream` 関数は、ファイル記述子が STREAMS ファイルを参照しているかどうかを判断するときに使用します。

次の例は、`isastream` 関数を使用して、ソケット・ベースのパイプでなく、STREAMS ベースのパイプをオープンしていることを確認する方法を示しています。

```
int fds[2];

pipe(fds);
if (isastream(fds[0]))
```

```
        printf("STREAMS based pipe\n");  
else  
        printf("Sockets based pipe\n");
```

関数の構文，パラメータ，エラーについては，`isastream(3)` リファレンス・ページを参照してください。

5.2.2.12 `fattach` 関数

`fattach` 関数は，STREAMS ベースのファイル記述子を，ファイル・システムの名前空間にあるオブジェクトにアタッチするときに使用します。

次の例は，`fattach` 関数を使用して，STREAMS ベースのパイプに名前を付ける方法を示しています。

```
int fds[2];  
  
pipe(fds);  
fattach(fd[0], "/tmp/pipe1");
```

注意

`fattach` 関数を使用するには，FFM_FS カーネル・オプションが構成されている必要があります。カーネル・オプションの構成については、『システム管理ガイド』を参照してください。

関数の構文，パラメータ，エラーについては，`fattach(3)` を参照してください。

5.2.2.13 `fdetach` 関数

`fdetach` 関数は，STREAMS ベースのファイル記述子をファイル名からデタッチするときに使用します。

注意

`fdetach` 関数を使用するには，FFM_FS (File on File Mount File System) カーネル・オプションが構成されている必要があります。カーネル・オプションの構成については、『システム管理ガイド』を参照してください。

関数の構文，パラメータ，エラーについては，`fdetach(3)` を参照してください。

表 5-2 は，STREAMS に関連する情報が記述されているリファレンス・ページについて簡単に説明した一覧です。詳細については，該当するリファレンス・ページを参照してください。

表 5-2: STREAMS リファレンス・ページ

リファレンス・ページ	説明
<code>autopush(8)</code>	システムの自動的にプッシュされた STREAMS モジュールのデータベースを管理するコマンド。
<code>clone(7)</code>	別の STREAMS ドライバで，未使用の主/副デバイスを見つけてオープンする STREAMS ソフトウェア・ドライバ。
<code>close(2)</code> ^a	指定したファイル記述子に関連付けられたファイルをクローズする関数。
<code>dlb(7)</code>	BSD スタイルのデバイス・ドライバと STREAMS プロトコル・スタック間の通信パスを提供する STREAMS 擬似ドライバ。
<code>fattach(3)</code>	STREAMS ベースのファイル記述子をファイル・システムのノードにアタッチするコマンド。
<code>fdetach(8)</code>	STREAMS ベースのファイル記述子をファイル名からデタッチするコマンド。
<code>fdetach(3)</code>	STREAMS ベースのファイル記述子をファイル名からデタッチする関数。
<code>getmsg(2)</code> <code>getpmsg(2)</code>	ストリーム・ヘッ드의読み取りキューに入っているメッセージを参照する関数。
<code>ifnet(7)</code>	データ・リンク・プロバイダ・インタフェース (DLPI) に合わせて作成された STREAMS ベースのデバイス・ドライバとソケット間のブリッジを提供する STREAMS ベースのモジュール。
<code>isastream(3)</code>	ファイル記述子が STREAMS ファイルを参照しているかどうかを判断する関数。
<code>mkfifo(2)</code>	単方向の STREAMS ベースのファイル記述子を作成する関数。
<code>open(2)</code> ^a	ファイルとファイル記述子の間に接続を確立する関数。
<code>pipe(2)</code>	双方向の STREAMS ベースのプロセス間通信チャンネルを作成する関数。

表 5-2: STREAMS リファレンス・ページ (続き)

リファレンス・ページ	説明
<code>poll(2)</code>	1 組のファイル記述子に関連する I/O 状態を報告し、指定された 1 つ以上の条件が真になるまで待機する一般的な機能を提供する関数。
<code>putmsg(2)</code> <code>putpmsg(2)</code>	STREAMS メッセージ・ブロックを生成する関数。
<code>read(2)</code> ^a	データをファイルから読み取って指定のバッファに入れる関数。
<code>strace(8)</code>	STREAMS ログ・ドライバから STREAMS のイベント・トレース・メッセージを検出するアプリケーション。
<code>strchg(1)</code>	ストリームの構成を変更するコマンド。
<code>strclean(8)</code>	STREAMS のエラー・ログ・ファイルを削除するコマンド。
<code>strconf(1)</code>	ストリームの構成について照会するコマンド。
<code>streamio(7)</code>	ストリーム上でさまざまな制御関数を実行するコマンド。
<code>strerr(3)</code>	STREAMS ログ・ドライバからエラー・メッセージを受信するデーモン。
<code>strlog(7)</code>	STREAMS のエラー・ロギング・デーモンおよびイベント・トレース・デーモンが使用する、ログ・メッセージを追跡するインタフェース。
<code>strsetup(8)</code>	適切な STREAMS 擬似デバイスを作成し、STREAMS モジュールの設定を表示するコマンド。
<code>timod(7)</code>	トランスポート・インタフェース (TI) をサポートしているトランスポート・ユーザからの <code>ioctl</code> 呼び出しを、TI をサポートしているトランスポート・プロトコル・プロバイダが使用できるメッセージに変換するモジュール。
<code>tirdwr(7)</code>	トランスポート・インタフェース (TI) をサポートしているトランスポート・ユーザに、TI をサポートしているトランスポート・プロトコル・プロバイダへの別のインタフェースを提供するモジュール。
<code>write(2)</code> ^a	データを指定のバッファからファイルへ書き込む関数。

^aそのページは STREAMS 固有のものではありません。

5.3 カーネル・レベル関数

この節では、STREAMS モジュールおよびドライバを作成するカーネル・プログラマが熟知しておかなければならない情報について説明します。説明する情報は、次のとおりです。

- モジュール・データ構造体
- メッセージ・データ構造体
- モジュールおよびドライバの STREAMS 処理ルーチン

5.3.1 モジュール・データ構造体

モジュールまたはドライバがシステムに組み込まれている場合は、そのモジュールまたはドライバで、読み取りキューと書き込みキュー、および他のモジュール情報を定義しなければなりません。

データ構造体 `qinit`、`module_info`、および `streamtab` は、すべて `<sys/stream.h>` ヘッダ・ファイルに記述されており、読み取りキューおよび書き込みキューを定義します。STREAMS モジュールは、宣言部分で、これらのデータ構造体の内容を指定しておかなければなりません。例については、付録 A を参照してください。

モジュールが提供しなければならない外部データ構造体は、`streamtab` だけです。

次に示す `qinit` 構造体は、キューのためのインタフェース・ルーチンを定義します。読み取りキューおよび書き込みキューそれぞれに、構造体のセットがあります。

```
struct qinit {
    int      (*qi_putp)();          /* put routine */
    int      (*qi_srvp)();          /* service routine */
    int      (*qi_qopen)();         /* called on each open */
                                     /* or a push */
    int      (*qi_qclose)();        /* called on last close */
                                     /* or a pop */
    int      (*qi_qadmin)();        /* reserved for future use */
    struct module_info * qi_minfo;  /* information structure */
    struct module_stat * qi_mstat;  /* statistics structure (op-
                                     tional) */
};
```

`module_info` 構造体には、モジュールまたはドライバの ID および制限値が入ります。次の例を参照してください。

```
struct module_info {
    unsigned short mi_idnum;        /* module ID number */
    ...
};
```



```

char      *mi_idname;      /* module name */
long      mi_minpsz;      /* min packet size, for */
                        /* developer use */
long      mi_maxpsz;      /* max packet size, for */
                        /* developer use */
ulong     mi_hiwat;        /* hi-water mark, for */
                        /* flow control */
ulong     mi_lowat;        /* lo-water mark, for */
                        /* flow control */
};

```

streamtab 構造体は、宣言の最上部に記述します。これは、モジュールまたはドライバの外部に見えなければならない唯一の部分です。次の例を参照してください。

```

struct streamtab {
    struct qinit * st_rdinit;    /* defines read QUEUE */
    struct qinit * st_wrinit;    /* defines write QUEUE */
    struct qinit * st_muxrinit; /* for multiplexing drivers only */
    struct qinit * st_muxwinit; /* ditto */
};

```

5.3.2 メッセージ・データ構造体

Tru64 UNIX STREAMS メッセージは、1 つ以上のリンクされたメッセージ・ブロックから構成されます。各データ・ブロックには、次の 3 つの構成要素があります。

- データ・バッファ

データ・バッファには、メッセージを構成するバイナリ・データが入っています。STREAMS では、データ・バッファ内のデータのフォーマットに関して位置合わせの規則はありません。ただし、ストリーム・ヘッドで処理されるメッセージのために従わなければならない位置合わせの規則があります。

- mblk_t 制御構造体

mblk_t 構造体には、メッセージの所有者が操作できる情報が入っています。この構造体内の 2 つのフィールドは、データ・バッファへの読み取りポインタと書き込みポインタです。

- dblk_t 制御構造体

dblk_t 構造体には、バッファの特性に関する情報が入っています。たとえば、この構造体の 2 つのフィールドは、データ・バッファの制限値を指し示すポインタです。他のフィールドには、メッセージ・タイプが入っています。

ストリーム・ヘッドは、データがアプリケーションからダウンストリームに流れるときに、メッセージ・データ構造体を作成して、その内容を指定します。ストリーム・エンドは、データが外部通信デバイスから流れてくる場合などのように、データがアップストリームに流れるときに、メッセージ・データ構造体を作成して、その内容を指定します。

mblk_t 構造体および dblk_t 構造体を次に示します。どちらの構造体も <sys/stream.h> ヘッダ・ファイルに記述されています。

```
/* message block */
struct msgb {
    struct msgb * b_next;           /* next message on queue */
    struct msgb * b_prev;           /* previous message on queue */
    struct msgb * b_cont;           /* next message block of message */
    unsigned char * b_rptr;         /* first unread data byte in buffer */
    unsigned char * b_wptr;         /* first unwritten data byte */
    struct datab * b_datab;         /* data block */
    unsigned char b_band;           /* message priority */
    unsigned char b_pad1;
    unsigned short b_flag;          /* message flags */
    long b_pad2;
    MSG_KERNEL_FIELDS
};
typedef struct msgb mblk_t;

/* data descriptor */
struct datab {
    union {
        struct datab * freep;
        struct free_rtn * frtnp;
    } db_f;
    unsigned char * db_base;        /* first byte of buffer */
    unsigned char * db_lim;         /* last byte+1 of buffer */
    unsigned char db_ref;           /* count of messages pointing */
                                    /* to block */
    unsigned char db_type;          /* message type */
    unsigned char db_iswhat;        /* message status */
    unsigned int db_size;           /* used internally */
    caddr_t db_msgaddr;            /* used internally */
    long db_filler;
};
#define db_freep db_f.freep
#define db_frtnp db_f.frtnp

typedef struct datab dblk_t;

/* Free return structure for esballoc */
typedef struct free_rtn {
    void (*free_func)(char *, char *); /* Routine to free buffer */
    char * free_arg;                  /* Parameter to free_func */
} frtn_t;
```

メッセージが STREAMS キューに入っている場合、そのメッセージは、ポインタ b_next および b_prev によってリンクされたメッセージ・リストの一部になります。キューに入っている先頭のメッセージは、そのキューの

`q_next` ポインタによって指し示され、キューに入っている最後のメッセージは、そのキューの `q_last` ポインタによって指し示されます。

5.3.3 ドライバおよびモジュールの STREAMS 処理ルーチン

モジュールまたはドライバは、アプリケーションが要求する処理をストリーム上で行うことができます。ただし、要求された処理を行うためには、STREAMS モジュールまたはドライバは、STREAMS フレームワークによって動作が指定される特殊ルーチンを提供しなければなりません。この項では、STREAMS モジュールおよびドライバが提供するルーチン、および処理の種類について説明します。説明する項目は、次のとおりです。

- オープン処理
- クローズ処理
- 構成処理
- 読み取り側プット処理
- 書き込み側プット処理
- 読み取り側サービス処理
- 書き込み側サービス処理

注意

STREAMS モジュールおよびドライバは、オープン、クローズ、および構成処理を提供しなければなりません。この項で説明する他の処理は、オプションです。

この項では、`XX_routine_name` というフォーマットを使用して各ルーチンを説明します。XX は、ユーザ作成の STREAMS モジュール名またはドライバ名に置き換えてください。たとえば、ユーザ作成の STREAMS 擬似デバイス・ドライバ `echo` の `open` ルーチンは、`echo_open` になります。

5.3.3.1 オープンおよびクローズの処理

`open` ルーチンおよび `close` ルーチンだけが、カーネルの `u_area` へのアクセスを提供します。これらのルーチンは、シグナルをキャッチした場合にだけ、スリープすることができます。

オープン処理

モジュールおよびドライバには、open ルーチンがなければなりません。読み取り側の qinit 構造体、st_rdinit は、qi_qopen フィールドに open ルーチンを定義します。ドライバの open ルーチンは、アプリケーションがストリームをオープンするときに呼び出されます。ストリーム・ヘッドは、アプリケーションがモジュールをそのストリームにプッシュすると、モジュール内の open ルーチンを呼び出します。

open ルーチンのフォーマットは、次のとおりです。

```
XX_open(q, devp, flag, sflag, credp)
    queue_t *q;      /* pointer to the read queue */
    dev_t *devp;     /* pointer to major/minor number
                     for devices */
    int flag;        /* file flag */
    int sflag;       /* stream open flag */
    cred_t *credp /* pointer to a credentials structure */
```

open ルーチンは、STREAMS ドライバまたはモジュールが内部で使用するための、データ構造体を割り当てることができます。このデータ構造体へのポインタは、通常 queue_t 構造体の q_ptr フィールドに格納されます。このポインタには、モジュールまたはドライバの他の部分から後でアクセスできます。

クローズ処理

モジュールおよびドライバには、close ルーチンがなければなりません。読み取り側の qinit 構造体、st_rdinit は、qi_qclose フィールドに close ルーチンを定義します。ストリームをオープンしたアプリケーションがそのストリームをクローズするときに、ドライバは close ルーチンを呼び出します。スタックからモジュールをポップするときに、ストリーム・ヘッドはモジュール内の close ルーチンを呼び出します。

close ルーチンのフォーマットは、次のとおりです。

```
XX_close(q, flag, credp)
    queue_t *q;      /* pointer to read queue */
    int flag;        /* file flag */
    cred_t *credp /* pointer to credentials structure */
```

close ルーチンは、内部で使ったデータ構造体を解放して、クリーン・アップするとよい場合があります。

5.3.3.2 構成処理

configure ルーチンは、STREAMS モジュールまたはドライバをカーネルに組み込むときに使用します。これは、Tru64 UNIX 固有のものであり、5.4 節で使用方法を説明します。

configure ルーチンのフォーマットは、次のとおりです。

```
XX_configure(op, indata, indatalen, outdata, outdatalen)
    sysconfig_op_t  op;          /* operation - should be */
                                /* SYSCONFIG_CONFIGURE */
    str_config_t *  indata;      /* for drivers - describes the device */
    size_t          indatalen;   /* sizeof(str_config_t) */
    str_config_t *  outdata;     /* pointer to returned data */
    size_t          outdatalen;  /* sizeof(str_config_t) */
```

5.3.3.3 読み取り側プットおよび書き込み側プットの処理

読み取り側と書き込み側の両方の、XX_xput ルーチンがあります。書き込み側のプット処理用は XX_wput であり、読み取り側のプット処理用は XX_rput です。

書き込み側プット処理

書き込み側のプット・ルーチン、XX_wput は、アップストリーム・モジュールの書き込み側が putnext 呼び出しを発行するときに呼び出されます。XX_wput ルーチンは、アップストリーム・モジュールから現在のモジュールまたはドライバへ引き渡されるメッセージ用の、唯一のインタフェースです。

XX_wput ルーチンのフォーマットは、次のとおりです。

```
XX_wput(q, mp)
    queue_t *q;  /* pointer to write queue */
    mblk_t *mp;  /* message pointer */
```

読み取り側プット処理

読み取り側のプット・ルーチン、XX_rput は、ダウンストリーム・モジュールの読み取り側が putnext 呼び出しを発行するときに呼び出されます。ストリーム・エンドであるドライバにはダウンストリーム・モジュールが存在しないので、読み取り側のプット・ルーチンがありません。XX_rput ルーチンは、ダウンストリーム・モジュールから現在のモジュールへ引き渡されるメッセージ用の、唯一のインタフェースです。

XX_rput ルーチンのフォーマットは、次のとおりです。

```

XX_rput(q, mp)
    queue_t *q; /* pointer to read queue */
    mblk_t *mp; /* message pointer */

```

XX_xput ルーチンは、少なくとも次のうちの 1 つを行わなければなりません。

- メッセージの処理
- 次のキューへのメッセージの引き渡し (putnext の使用)
- メッセージを、モジュールのサービス・ルーチンにプットすることによる、メッセージ処理の遅延 (putq の使用)

XX_xput ルーチンは、大量に処理する場合には、サービス・ルーチンに依頼しなければなりません。

5.3.3.4 読み取り側サービスおよび書き込み側サービスの処理

XX_xput ルーチンが、大量の処理を必要とするメッセージを受信した場合は、ただちに処理を行うと、フロー制御の問題が生じる可能性があります。メッセージをただちに処理する代わりに、XX_rput ルーチンは、(putq システム・コールを使用して) メッセージを読み取り側のメッセージ・キューにいれ、XX_wput ルーチンは、メッセージを書き込み側のキューに入れることができます。STREAMS モジュールは、これらのキューにメッセージが入っていることを通知し、モジュールの読み取り側または書き込み側のサービス・ルーチンをスケジューリングして、それらのメッセージを処理します。モジュールの XX_rput ルーチンが putq を呼び出さなければ、モジュールは、読み取り側のサービス・ルーチンを必要としません。同様に、モジュールの XX_wput ルーチンが putq を呼び出さなければ、モジュールは、書き込み側のサービス・ルーチンを必要としません。

基本的なサービス・ルーチンのコードは、読み取り側も書き込み側も、次のフォーマットです。

```

XXXsrv(q)
queue_t *q;
{
    mblk_t *mp;

    while ((mp = getq(q)) != NULL)
    {
        /*
         * If flow control is a problem, return
         * the message to the queue
         */

        if (!(canput(q->q_next))

```

```

        return putbq(q, mp);
    /*
     * process message
     */
    putnext(q, mp);
}
return 0;
}

```

5.3.4 Tru64 UNIX STREAMS の概念

次の STREAMS の概念は、Tru64 UNIX 独自のものです。この項では、これらの概念と実現方法について説明します。

- 同期
- タイムアウト

5.3.4.1 同期

Tru64 UNIX は、複数のカーネル STREAMS スレッドの使用をサポートします。STREAMS キューおよび関連するデータ構造体への排他的アクセスは保証されていません。メッセージが同じストリームを同時に上下に移動したり、複数のプロセスが同じストリームを下るメッセージを送信することができます。

データ構造体へ同期をとってアクセスするため、各 STREAMS モジュールまたはドライバは許容できる同期レベルを選択します。同期レベルは、モジュールまたはドライバで許可されている並列処理のレベルを決定します。同期レベルは、モジュールまたはドライバの構成ルーチンで定義される streamadm データ構造体の `sa.sa_syn_level` フィールドで定義されます。`sa.sa_syn_level` フィールドは、次のいずれかの値でなければなりません。

SQLVL_QUEUE

キュー・レベル同期。このレベルは、1 つの実行スレッドがモジュールまたはドライバの書き込みキューの任意のインスタンスにアクセスしているときに、別の実行スレッドがモジュールまたはドライバの読み取りキューの任意のインスタンスにアクセスできます。キュー・レベル同期は、読み取りキューおよび書き込みキューが共通のデータを共有していない場合に使用できます。引数 `SQLVL_QUEUE` は、Tru64 UNIX の STREAMS フレームワークにおいて利用できる最下位レベルの同期化を提供します。

たとえば、読み取りキューおよび書き込みキューの `q_ptr` フィールドは、同じメモリ位置を指しません。

SQLVL_QUEUEPAIR

キュー・ペア・レベル同期。一度に1つのスレッドだけが、このモジュールまたはドライバの各インスタンスの読み取りキューおよび書き込みキューにアクセスできます。この同期レベルは、データを処理し、ストリームごとの状態のみを持つほとんどのモジュールおよびドライバに共通です。

たとえば、モジュールのインスタンス内において、読み取りキューおよび書き込みキューの `q_ptr` フィールドは、同じメモリ位置を指します。モジュール内には他に共用データはありません。

SQLVL_MODULE

モジュール・レベル同期。このモジュールまたはドライバ内のすべてのコードはシングル・スレッドです。モジュールまたはドライバのすべてのインスタンスにアクセスできるのは、1つの実行スレッドだけです。たとえば、データは、モジュールまたはドライバのすべてのインスタンスによってアクセスされています。

SQLVL_ELSEWHERE

任意レベル同期。モジュールまたはドライバは、他の任意のモジュールまたはドライバと同期をとります。このレベルは、互いにデータをアクセスするモジュールまたはドライバのグループの同期をとるために使用されます。文字列は、`streamadm` 構造体の `sa.sync_info` フィールドにあるこのオプションと一緒に渡されます。この文字列は、モジュールまたはドライバのセットと関連付けるために使用されます。共同動作するモジュールまたはドライバ間の規約により、この文字列が決定されます。

たとえば、データを共用する TCP モジュールと IP モジュールのようなネットワーキング・スタックが、文字列 `tcp/ip` の受け渡しを同意したとします。この場合には、1つの実行スレッドだけが、この文字列で同期をとるすべてのモジュールまたはドライバにアクセスすることができます。

SQLVL_GLOBAL

グローバル・レベル同期。これより低いレベルのモジュールまたはドライバはすべてシングル・スレッドです。保護の異なる他のレベルを使用しているモジュールまたはドライバがあることに注意してください。このオプションは主にデバッグで利用されます。

5.3.4.2 タイムアウト

`timeout` および `untimeout` へのカーネル・インタフェースは次のとおりです。

```
timeout(func, arg, ticks);
untimeout(func, arg);
```

ただし、AT&T System V Release 4 の STREAMS とソースの互換性を維持するため、`<sys/stream.h>` ヘッダ・ファイルでは `timeout` を次のように再定義して、System V インタフェースにします。

```
id = timeout(func, arg, ticks);
untimeout(id);
```

変数 `id` は `int` として定義されています。

STREAMS のモジュールおよびドライバは、System V インタフェースを使用しなければなりません。

5.4 Tru64 UNIX カーネル内へのユーザ作成の STREAMS ベースのモジュールまたはドライバの組み込み

ユーザが作成した STREAMS ドライバまたはモジュールにシステムがアクセスするためには、そのドライバとモジュールをシステムのカーネルに組み込まなければなりません。

STREAMS ドライバまたはモジュールは構成可能なカーネル・サブシステムとして認識されるべきなので、『プログラミング・ガイド』の記述に従ってカーネル・サブシステムの構成を行ってください。

次の説明では、ソース・ファイルが `mymodule1.c` および `mymodule2.c` である、STREAMS ベースのモジュール (プッシュ可能モジュール、ハー

ドウェア、擬似デバイス・ドライバ) 例 mymod をカーネルに追加する方法を示します。

1. モジュール・ソース・ファイル (この例では /sys/streamsm/mymodule.c) 内で構成ルーチンを宣言する。

このサンプルの mymod_configure は、モジュール用です。ドライバに使用する場合は、次のようにします。

- a. コメント行 /* driver */ に続く次の行のコメントをはずす。

```
/* sa.sa_flags      = STR_IS_DEVICE | STR_SYSV4_OPEN; */
```

- b. コメント行 /* module */ に続く次の行をコメントにする。

```
sa.sa_flags          = STR_IS_MODULE | STR_SYSV4_OPEN;
```

例 5-1: サンプル・モジュール

```
/*
 * Sample mymodule.c
 */
:
:

#include <sys/sysconfig.h>
#include <sys/errno.h>

struct streamtab mymodinfo = { &rinit, &winit };

cfg_subsys_attr_t mymod_attributes[] = {
    {,0,0,0,0,0,0} /* required last element */
};

int
mymod_configure(
    cfg_op_t    op;
    caddr_t      indata;
    ulong        indata_size;
    caddr_t      outdata;
    ulong        outdata_size)
{
    dev_t    devno = NODEV;
    struct streamadm sa;
    if (op != CFG_OP_CONFIGURE)
        return EINVAL;

    sa.sa_version      = OSF_STREAMS_10;
    /* module */
    sa.sa_flags         = STR_IS_MODULE | STR_SYSV4_OPEN;
```

例 5-1: サンプル・モジュール (続き)

```
/* driver */
/* sa.sa_flags      = STR_IS_DEVICE | STR_SYSV4_OPEN; */
sa.sa_ttys          = NULL;
sa.sa_sync_level    = SQLVL_MODULE;          [5]
sa.sa_sync_info     = NULL;
strcpy(sa.sa_name, "mymod");

if ((devno = strmod_add(devno, &mymodinfo, &sa)) == NODEV)
{
    return ENODEV;
}

return ESUCCESS;
}
```

- [1] この例は簡略化されているので、例中のサブルーチンが提供する属性テーブルは空で、サブルーチンに属性が渡されることも予期していません。モジュールに属性を開発する場合は『プログラミング・ガイド』を参照してください。
- [2] cdevsw テーブルの最初の仕様可能なスロットは自動的にモジュールに割り当てられます。特定のデバイス番号を予約する場合は、conf.c プログラムの cdevsw テーブルを調べてから定義してください。cdevsw テーブルおよびデバイス・ドライバ・エントリの追加についての詳細は『*Writing Device Drivers*』を参照してください。
- [3] このルーチン例では、CFG_OP_CONFIGURE オプションだけがサポートされています。構成ルーチンの他のオプションについては『プログラミング・ガイド』を参照してください。
- [4] STR_SYSV4_OPEN は、AT&T System V Release 4 の呼び出しシーケンスを使用して、モジュールまたはデバイスの open および close ルーチンを呼び出すことを指定します。このビットが指定されていない場合には、AT&T System V Release 3.2 の呼び出しシーケンスが使用されます。
- [5] sa_sync_level フィールドの他のオプションについては、5.3.4 項を参照してください。

2. モジュールをカーネルと静的にリンクする。

構成する STREAMS モジュールを動的にロード可能にする場合は『プログラミング・ガイド』のカーネル・サブシステムの構成について参照してください。構成するモジュールがハードウェア・デバイス・ドライバの場合は『*Writing Device Drivers*』を参照してください。

モジュールをカーネルと静的にリンクする場合は、モジュールのソース・ファイル (mymodule1.c および mymodule2.c) を /sys/stream ディレクトリに置き、次の例のように /sys/conf/files ファイルのエントリに各ファイルを追加します。次の例は、mymodule1.c および mymodule2.c の /sys/conf/files ファイルのエントリを示します。

```
streamsm/mymodule1.c    optional mymod Notbinary
streamsm/mymodule2.c    optional mymod Notbinary
```

カーネル構成ファイルに MYMOD オプションを追加します。省略時のカーネル構成ファイルは /sys/conf/HOSTNAME です。(HOSTNAME にはシステムの名前を大文字で指定します) たとえばシステム名が TRU64 の場合、構成ファイル /sys/conf/TRU64 に次の行を追加します。

```
options MYMOD
```

ハードウェア・デバイス・ドライバを組み込んでいる場合は、手順 3 を続け、ハードウェア・デバイス・ドライバを組み込んでいない場合には、手順 4 に進みます。

3. ハードウェア・デバイス・ドライバを組み込んでいる場合は、手順 3a から手順 3d までをすべて実行する。

ハードウェア・デバイス・ドライバを組み込んでいない場合には、手順 4 に進みます。

ハードウェア・デバイス・ドライバを組み込んでいる場合は、事前に XXprobe および interrupt ルーチンを定義しておかなければなりません。probe ルーチンおよび interrupt ルーチンの定義方法についての詳細は、『*Writing Device Drivers*』を参照してください。

- a. 次の行を、デバイス・ドライバ構成ファイルの先頭に追加する。

この例では、デバイス・ドライバ構成ファイルは、
/sys/stream/mydriver.c です。

```
#include <io/common/devdriver.h>
```

- b. コントローラ構造体へのポインタを定義する。

たとえば、次のようにします。

```
struct controller *XXinfo;
```

コントローラ構造体についての詳細は、『*Writing Device Drivers*』を参照してください。

- c. driver 構造体を宣言して初期化する。

たとえば、次のようにします。

```
struct driver XXdriver =
{
    XXprobe, 0, 0, 0, 0, XXstd, 0, 0, "XX", XXinfo
};
```

driver 構造体についての詳細は、『*Writing Device Drivers*』を参照してください。

- d. コントローラ行を、カーネル構成ファイルに追加する。

省略時のカーネル構成ファイルは `/sys/conf/HOSTNAME` です。ここで `HOSTNAME` はマシンの名前です (大文字で示します)。たとえば、使用しているシステムの名前が `TRU64` の場合、次のような 1 行を `/sys/conf/TRU64` 構成ファイルに追加します。

```
controller XX0 at bus vector XXintr
```

`bus` キーワードが取り得る値についての詳細は、『システム管理ガイド』を参照してください。

4. `doconfig` コマンドを使用して、このマシン用の新しいカーネルを、再構成、再構築して、ブートする。

カーネルの再構成についての詳細は、`doconfig(8)`、または『システム管理ガイド』を参照してください。

5. `strsetup -c` コマンドを実行して、デバイスが正しく構成されていることを確認する。

```
# /usr/sbin/strsetup -c
```

```
STREAMS Configuration Information...Wed Jun  2 09:30:11 1994
```

Name	Type	Major	Minor	Module ID
----	----	-----	-----	-----
clone		32	0	
ptm	device	37	0	7609
pts	device	6	0	7608
log	device	36	0	44
nuls	device	38	0	5001
echo	device	39	0	5000

sad	device	40	0	45
pipe	device	41	0	5304
kinfo	device	42	0	5020
xtisoUDP	device	43	0	5010
xtisoTCP	device	44	0	5010
dlb	device	49	0	5010
bufcall	module			0
timod	module			5006
tirdwr	module			0
ifnet	module			5501
ldtty	module			7701
null	module			5003
pass	module			5003
errm	module			5003
spass	module			5007
rspass	module			5008
pipemod	module			5303

Configured devices = 11, modules = 11

5.5 デバイス特殊ファイル

この節では、STREAMS デバイス特殊ファイルとその作成方法について説明します。また、clone デバイスの概要についても説明します。

すべての STREAMS ドライバには、システムに作成された文字型特殊ファイルがなければなりません。これらのファイルは、通常、/dev/streams にあり、インストール時に作成されるか、または /usr/sbin/strsetup ユーティリティを実行することによって作成されます。

STREAMS ドライバには、主デバイス番号が関連付けられています。これは、ドライバをシステムに組み込むときに決定されます。STREAMS 以外のドライバには、通常、主デバイスと副デバイス番号の各組み合わせに対して文字型特殊ファイルが定義されています。次は、/dev/rdisk ディレクトリのエントリの例です。

```
crw----- 1 root    system    8,    1024 Aug 25 15:38 dsk1a
crw----- 1 root    system    8,    1025 Aug 25 15:38 dsk1b
crw----- 1 root    system    8,    1026 Aug 25 15:38 dsk1c
```

この例では、dsk1a の主デバイス番号は 8 であり、副デバイス番号は 1024 です。dsk1b は、主デバイス番号が 8 で、副デバイス番号が 1025、dsk1c は、主デバイス番号が 8 で、副デバイス番号が 1026 です。

また、STREAMS ドライバの場合にも、主デバイス番号と副デバイス番号の各組み合わせに対して文字型特殊ファイルを定義できます。次は、`/dev/streams` ディレクトリのエントリの例です。

```
crw-rw-rw-  1 root  system  32,  0 Jul 13 12:00 /dev/streams/echo0
crw-rw-rw-  1 root  system  32,  1 Jul 13 12:00 /dev/streams/echo1
```

この例では、`echo0` の主デバイス番号は 32 であり、副デバイス番号は 0 です。`echo1` は、主デバイス番号が 32 で、副デバイス番号が 1 です。

アプリケーションが、デバイスに対して一意のストリームをオープンするためには、そのデバイスの未使用の副デバイスをオープンしなければなりません。最初のアプリケーションは、`/dev/streams/echo0` についてオープンでき、2 番目のアプリケーションは、`/dev/streams/echo1` についてオープンできます。これらの各デバイスは、異なる副デバイス番号を持つので、各アプリケーションは、`echo` ドライバに対して一意のストリームを獲得します。この方法では、各デバイス（この場合は `echo`）について、オープンできる各副デバイス用に 1 つの文字型特殊ファイルが必要です。また、この方法では、すでに使用されている文字型特殊ファイルをオープンしないようにするために、アプリケーションがどの文字型特殊ファイルをオープンするべきかを判断する必要があります。

`clone` デバイスは、オープンできる各副デバイスに対して、デバイス特殊ファイルを定義するもう 1 つの方法を提供します。`clone` デバイスを使用すると、各ドライバは、文字型特殊ファイルを 1 つだけ必要とします。また、現在利用できる副デバイスをアプリケーションが判別する必要はなく、その代わりに、`clone` デバイスの主デバイス番号を使用して、2 番目（または 3 番目）のデバイスをオープンできます。副デバイス番号は、オープンされているデバイス（この場合は `echo`）に関連付けられます。`clone` デバイスの主デバイス番号を使用してデバイスをオープンするたびに、STREAMS ドライバは、そのデバイスを一意のストリームとして解釈します。

`strsetup` コマンドは、`/dev/streams` ディレクトリにエントリを設定して、`clone` デバイスを使用できるようにします。次は、`/dev/streams` ファイルのエントリの例です。

```
crw-rw-rw-  1 root  system  32, 18 Jul 13 12:00 /dev/streams/echo
```

この例では、システムによって、`clone` デバイスに主デバイス番号 32 が割り当てられています。18 は、`echo` に関連付けられた主デバイス番号です。アプリケーションが `/dev/streams/echo` をオープンすると、`clone` デバイスは、この呼び出しを遮断して、`echo` ドライバの代わりに `open` ルーチンを

呼び出します。さらに、`clone` は、`echo` ドライバに、クローン・オープンを行うことを通知します。`echo` ドライバがクローン・オープンを認識すると、主デバイス番号 18、および最初に利用できる副デバイス番号を返します。

注意

`/usr/sbin/strsetup` コマンドが作成する文字型特殊ファイルは、省略時には、主デバイス番号と同様に `clone` を使用して `/dev/streams` ディレクトリに作成されます。クローン・オープンを使用しないか、または異なる名前を使用している STREAMS ドライバをカーネルに組み込む場合は、`strsetup.conf(4)` に記述されている `/etc/strsetup.conf` ファイルを変更しなければなりません。

使用しているシステムの `clone` デバイスの主デバイス番号を調べる場合は、`strsetup -c` コマンドを実行します。

5.6 エラーとイベントのロギング

STREAMS のエラーとイベントのロギングは、次の事項に関連しています。

- エラー・ロガー・デーモン
- トレース・ロガー
- `strclean` コマンド

エラー・ロガー・デーモン、`strerr` は、STREAMS のエラー・ロギングおよびイベント・トレース機能に送信されたすべてのエラー・メッセージを、ファイルに記録します。

トレース・ロガー、`strace` は、STREAMS のエラー・ロギングおよびイベント・トレース機能に送信されたトレース・メッセージを、標準出力に書き込みます。

`strclean` コマンドを実行すると、`strerr` デーモンによって生成された古いログ・ファイルをすべて削除することができます。

STREAMS モジュールまたはドライバは、`strlog` カーネル・インタフェースを通して、エラー・メッセージおよびイベント・トレース・メッセージ

を、STREAMS のエラー・ロギングおよびイベント・トレース機能に送信できます。これには、strlog の呼び出しが含まれます。

次の例では、STREAMS ドライバが、オープン・ルーチンの実行中に、STREAMS のエラー・ロギングおよびイベント・トレース機能の両方に、主デバイス番号および副デバイス番号を出力します。

```
#include <sys/strlog.h>

strlog(MY_DRIVER_ID, 0, 0, SL_ERROR | SL_TRACE,
      "My driver: mydriver_open() - major=%d,minor=%d",
      major(dev),minor(dev));
```

また、ユーザ・プロセスも、/dev/streams/log に対してストリームをオープンし、putmsg を呼び出すことによって、STREAMS のエラー・ロギングおよびイベント・トレース機能に、メッセージを送信できます。ログ・メッセージを strlog に引き渡すために、ユーザ・プロセスには、次のようなコードが含まれていなければなりません。

```
struct strbuf ctl, dat;
struct log_ctl lc;
char *message = "Last edited by <username> on <date>";

ctl_len = ctl.maxlen = sizeof (lc);
ctl.buf = (char *)&lc;

dat.len = dat.maxlen = strlen(message);
dat.buf = message;
lc.level = 0;
lc.flags = SL_ERROR|SL_NOTIFY;

putmsg (log, &ctl, &dat, 0);
```



拡張 SNMP アプリケーション・プログラミング・インタフェース

シンプル・ネットワーク管理プロトコル (SNMP) は、ルータ、ブリッジ、ホストなど、ネットワークに接続されているデバイスのリモート管理とデータ収集を可能にするアプリケーション層プロトコルです。

管理されたネットワーク接続デバイスは、そのデバイスのための SNMP エージェントとして機能するソフトウェアを含んでいます。このソフトウェアは、SNMP に対するアプリケーション層プロトコルを処理し、管理コマンドを実行します。管理コマンドは、情報収集パラメータおよび操作に関する設定パラメータで構成されます。

ネットワーク上の各管理デバイスに SNMP コマンドを送って管理タスクを実行するネットワーク管理プログラムもあります。通常このプログラムは、ネットワーク上のいずれかの単一ホストで実行されます。このプログラムが実行する管理タスクとしては、構成管理、ネットワーク・トラフィックの監視、ネットワークに関するトラブル・シューティングなどがあります。

eSNMP (Extensible Simple Network Management Protocol) は、Tru64 UNIX あるいは DIGITAL UNIX の以前のバージョンのための SNMP エージェントです。このエージェントは、マスタ・エージェント・プロセスと eSNMP サブエージェントなどの複数の関連プロセスで構成されます。マスタ・エージェントは SNMP プロトコルを処理し、サブエージェントは要求された管理コマンドを実行します。この章では、次の項目については既知であると仮定して説明します。

- SNMP プロトコル
- MIB (Management Information Base) 定義および RFC (Request for Commnet)
- OID (Object Identifier) および ISO (International Standards Organization) 登録階層 (1.3.6.1.2.1 など)
- C プログラミング言語

この章では次の情報を提供します。

- eSNMP の概要
- eSNMP アプリケーション・プログラミング・インタフェース (API) の概要
- eSNMP ルーチンについての詳細

6.1 eSNMP の概要

以降の節では、eSNMP エージェントの構成要素とアーキテクチャについて下記の順に説明します。

- eSNMP の構成要素
- アーキテクチャ
- SNMP バージョン

6.1.1 eSNMP の構成要素

eSNMP の構成要素は次のとおりです。

- `/usr/sbin/snmpd` — マスタ・エージェント・デーモン
- `/usr/sbin/os_mibs` — ホスト MIB およびネットワーキング・サブエージェント・デーモン
- `/usr/sbin/svrMgt_mib` — サーバ管理サブエージェント・デーモン
- `/usr/sbin/svrSystem_mib` — サーバ・システム・サブエージェント・デーモン
- `/usr/sbin/mosy` — MIB コンパイラ
- `/usr/sbin/snmpi` — オブジェクト・テーブル・コード・ジェネレータ
- `/usr/shlib/libesnmp.so` — eSNMP ライブラリ
- `/usr/include/esnmp.h` — eSNMP 定義
- `/usr/examples/esnmp/*` — サンプル・コード

MIB (Management Information Base) 定義は、ネットワーク管理に関するデータ要素群です。これらの多くは、インターネット協会 (Internet Society) の IETF (Internet Engineering Task Force) ワーキング・グループの標準化作業の成果物として作成された RFC で標準化されています。

RFC で定義されているデータ要素は、階層構造のネーミング・スキームで識別されます。階層構造の各レベルのそれぞれの名前は、それに対応する番号を持っています。MIB 定義のデータ要素は、データ要素の名前あるいはオブジェクト識別子 (OID) と呼ばれる対応するシーケンス番号によって参照することができます。各データ要素の OID は、さらに番号を追加することにより、データ要素の特定のインスタンスを識別するように拡張できます。管理されたデータ要素の集合全体を MIB ツリーと呼びます。

各 SNMP エージェントは、管理されているデバイスに属する MIB 要素と、いくつかの共通 MIB 要素を実現します。これらの MIB 要素をサポート MIB ツリー要素と呼びます。拡張 SNMP エージェントは、このサポート MIB ツリー要素を複数のプロセス間に分散させ、動的に変更することができます。

eSNMP には、1 つのマスタ・エージェントといくつかのサブエージェントがあります。マスタ・エージェントは、SNMP プロトコルを扱い、SNMP 自身に関連した MIB をサポートし、接続されたサブエージェントおよびサブエージェントがサポートする MIB サブツリーの登録の管理を行います。eSNMP のためのマスタ・エージェントは、デーモン・プロセス `/usr/sbin/snmpd` です。

Tru64 UNIX は、IETF 標準もしくは HP 独自の異なる MIB をいくつか実現しているサブエージェントを提供します。詳細は、『QuickSpecs』、『Tru64 UNIX 概要』および `snmpd(8)` を参照してください。これらのサブエージェント（および他社のサブエージェント）はマスタ・エージェントとともにホストに対して、1 つの SNMP エージェントであるかのように機能します。

6.1.2 アーキテクチャ

マスタ・エージェントは、事前に割り当てられている UDP (ユーザ・データグラム・プロトコル) ポート上で、SNMP 要求が送られてくるのを待っています。SNMP 要求を受信すると、マスタ・エージェントはローカル・セキュリティ・データベース (`snmpd(8)` を参照) に対して認証を行い、認証エラーあるいはプロトコル・エラーが発生した場合はそれを処理します。要求が有効な場合、`snmpd` デーモンはその MIB 登録を照会します。要求に含まれているそれぞれの要求 MIB オブジェクトに対して、どの登録済 MIB がそのオブジェクトを含み、どのサブエージェントがその MIB を登録したかを判断します。その後、マスタ・エージェントは (SNMP 要求に含まれる各サブエージェントに対して) 一連のメッセージを作成します。

各サブエージェント・プログラムは、シェアード・ライブラリ `libesnmp.so` にリンクしています。このライブラリには、マスタ・エージェントとサブエージェントとの通信を可能にするプロトコルのインプリメンテーションが含まれています。このコードは、マスタ・エージェントのメッセージを解析し、ローカル・オブジェクト・テーブルを照会します。

オブジェクト・テーブルは、`snmpi` および `mosy` MIB コンパイラ・ツールで生成されたコードで定義および初期化されたデータ構造体です。このデータ構造体には、サブエージェントで実現されている MIB に含まれる各 MIB オブジェクトに対するエントリが含まれています。オブジェクト・テーブル・エントリのある部分は、MIB オブジェクトに対する要求をサービスする関数のアドレスです。この関数をメソッド・ルーチンと呼びます。

eSNMP ライブラリ・コードは、マスタ・エージェントのメッセージにおいて各 MIB 変数に対して指定されたメソッド・ルーチンに呼び込まれます。eSNMP ライブラリ・コードは、戻り値をもとに応答パケットを作成し、それをマスタ・エージェントへ返します。

マスタ・エージェントはタイマを起動し、すべてのサブエージェントからの応答パケットを整理します。(たとえば `GetNext` 要求など) 特定の要求に依存して、マスタ・エージェントは新しいサブエージェント・メッセージを再作成し再送します。必要なデータあるいはエラー応答をマスタ・エージェントがすべて持っている場合 (あるいはサブエージェントからの応答が時間切れになっても届かない場合)、マスタ・エージェントは SNMP 応答メッセージを作成し、もとの SNMP アプリケーションに送ります。要求を出している SNMP 管理アプリケーションには、マスタ・エージェントとサブエージェントとのやり取りは見えません。

サブエージェント・プログラムは、すべてのプロトコル処理および発送を行う `libesnmp.so` シェアラブル・ライブラリにリンクしています。サブエージェント開発者は、それらの定義済みの MIB オブジェクトだけに対するメソッド・ルーチンをコーディングする必要があります。

6.1.3 SNMP バージョン

拡張 SNMP は、次の範囲で SNMPv2c をサポートします。これは、RFC 1901 から RFC 1908 までをベースにしています。

- MIB ツール (mosy および snmpi プログラム) は SNMPv2c Structure of Management Information for SNMPv2 (SMIv2) と textual conventions をサポートしています。
- eSNMP ライブラリ API は、SNMPv2c、変数割り当て例外処理、エラー・コードをサポートします。
- マスタ・エージェントは現在のところ SNMPv1 および SNMPv2c を 2 か国語方式でサポートしています。サブエージェントからの SNMPv2c 固有情報はすべて、必要な場合には、RFC 2089 に従って SNMPv1 関連のデータにマップされます。たとえば、管理アプリケーションが SNMPv1 PDU を使用して要求を作成すると、マスタ・エージェントは SNMPv1 PDU を使用して応答し、サブエージェントから受け取った SNMPv2c SMI 要素をすべてマップします。つまり、以前のバージョンの eSNMP API で作成されたサブエージェントでは、コード変更が不要であり、再コンパイルの必要がありません。

6.1.4 AgentX

マスタ・エージェント間で libesnmp.so ライブラリによって実行される通信はすべて、RFC 2741、『*Agent Extensibility (AgentX) Version 1*』を実現して行われています。RFC 2741 は、拡張可能なエージェント構成要素、マスタ・エージェント、複数のサブエージェントの間の通信に関する標準的なプロトコルを定義しています。すなわち、eSNMP API を使用するサブエージェントは、異なるベンダのマスタ・エージェント (RFC 2741 に準拠しているもの) が Tru64 UNIX ホスト上で実行されていても、修正なしで正しく動作します。

6.2 拡張 SNMP アプリケーション・プログラミング・インタフェースの概要

サブエージェントの機能は、マスタ・エージェントとの通信を確立し、処理しようとしている MIB サブツリーを登録し、マスタ・エージェントからの要求を処理することです。また、ホスト・アプリケーションに代わって SNMP トラップを送ることができます。

サブエージェントは、次のものから構成されます。

- メイン関数 (開発者によって作成される)
- AgentX プロトコル動作を実行する eSNMP ライブラリ・ルーチン

- 特定の MIB 要素を処理するメソッド・ルーチン (開発者によって作成される)
- (`mosy` および `snmpi` プログラムを使用して RFC から生成された) オブジェクト・テーブル構造体

通常、サブエージェントはルータ・デーモンなどのアプリケーション内に組み込まれています。サブエージェントの処理はプロセスが実行する処理のほんの一部分です。この場合、アプリケーションのメイン・イベント・ループは、eSNMP ライブラリを呼び出します。また、サブエージェントが独自のメインルーチンを持つスタンドアロン・デーモンである場合もあります。

マスタ・エージェントからのパケット処理中に eSNMP ライブラリは、要求された各 MIB 変数に対して指定のメソッド・ルーチンを呼び出します。オブジェクト・テーブルの定義済みの各 MIB 変数は、その MIB 変数の要求を処理するためのメソッド・ルーチンへのポインタを含んでいます。サブエージェントのオブジェクト・テーブルは `mosy` および `snmpi` プログラムによって生成されるため、メソッド・ルーチン名は静的です。

オペレーティング・システムで提供される eSNMP 開発者キットには次のものが含まれます。

- `/usr/sbin/mosy` — MIB コンパイラ・ユーティリティ
- `/usr/sbin/snmpi` — オブジェクト・テーブル・コード生成ユーティリティ
- `/usr/examples/esnmp/mib-converter.sh` — MIB テキスト抽出ツール
- `/usr/shlib/libesnmp.so` — eSNMP シェアード・ライブラリ
- `/usr/include/esnmp.h` — eSNMP 定義ファイル
- `/usr/examples/esnmp/*` — サブエージェント・サンプル・ソース・コード

eSNMP シェアード・ライブラリ (`libesnmp.so`) は次のサービスを提供しています。

- マスタ・エージェント/サブエージェント間プロトコル処理ルーチン
- サブエージェントの代わりにマスタ・エージェントと通信する以下のルーチン

- esnmp_init
プロトコルを初期化します (マスタ・エージェントとハンドシェイクを行います)。
 - esnmp_allocate
1 つまたは複数の特定のインデックス・オブジェクト (OID) に値を割り当てるように、マスタ・エージェントに要求します。
 - esnmp_deallocate
1 つまたは複数のインデックス・オブジェクト (OID) の値を割り当て解除するように、マスタ・エージェントに要求します。
 - esnmp_register
マスタ・エージェントに MIB サブツリーを登録します。
 - esnmp_poll
マスタ・エージェントからのパケットを処理します。
 - esnmp_trap
マスタ・エージェントに SNMP トラップを生成するよう要求します。
 - esnmp_are_you_there
マスタ・エージェントに対して ping コマンドを実行します。
 - esnmp_unregister
MIB サブツリーの登録を解除します。
 - esnmp_term
マスタ・エージェントとの通信を終了し、拡張 SNMP を終了します。
 - esnmp_systime — 時間処理および同期化。
- サポート・ルーチン
メソッド・ルーチンの開発の役に立つルーチン。詳細なリストと各 eSNMP サポート・ルーチンの説明については、6.3 節を参照してください。

- eSNMP に関連して `esnmp.h` ヘッダ・ファイルが提供されています。このファイルは、eSNMP API に対するサブエージェントを実現するのに必要なすべてのデータ構造、定数、関数プロトタイプを定義しています。

6.2.1 MIB サブツリー

サブエージェントの動作と eSNMP API について理解するためには、MIB サブツリーについて理解することが非常に重要です。

注意

この項では、SNMP で使用される OID 命名構造については理解しているものとして説明します。OID 命名構造については、RFC 1902 『*Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)*』を参照してください。

SNMP における情報は、逆ツリー構造の階層構造になっています。この階層構造のなかで、データは、リーフ・ノードに対応付けられます。各ノードには名称と番号が付けられます。各ノードは、OID によっても識別することができます。OID は、ツリーのルートからそのノードまでのパス上に存在する、サブ識別子と呼ばれる負でない番号を連鎖したものです。OID の長さは、最低でサブ識別子が 2 つなければならず、サブ識別子が、たかだか 128 個の長さです。有効なサブ識別子 1 の値は、込みで 0 から 2 の範囲であり、サブ識別子 2 の値は、込みで 0 から 39 の範囲であり、残りのサブ識別子の値は、負でない番号です。

たとえば、`/usr/examples/esnmp` ディレクトリのサンプル・コードで提供される chess MIB は、chess という名前の要素を持っています。要素 chess の OID は、ツリーの階層構造における位置を表す 1.3.6.1.4.1.36.2.15.2.99 です。

```
iso(1)
  org(3)
    dod(6)
      internet(1)
        private(4)
          enterprise(1)
            digital(36)
              ema(2)
                sysobjects(15)
```

```
decof(2)
chess(99)
```

MIB 階層構造のどのノードも MIB サブツリーを定義することができます。サブツリー内のすべての要素は、サブツリー・ベースの OID で始まる OID を持っています。たとえば chess を MIB サブツリー・ベースとして定義した場合、chess OID をプレフィックスとして持つ要素は、次のようにすべて MIB サブツリー内に存在します。

chess	1.3.6.1.4.1.36.2.15.2.99
chessProductID	1.3.6.1.4.1.36.2.15.2.99.1
chessMaxGames	1.3.6.1.4.1.36.2.15.2.99.2
chessNumGames	1.3.6.1.4.1.36.2.15.2.99.3
gameTable	1.3.6.1.4.1.36.2.15.2.99.4
gameEntry	1.3.6.1.4.1.36.2.15.2.99.4.1
gameIndex	1.3.6.1.4.1.36.2.15.2.99.4.1.1
gameDescr	1.3.6.1.4.1.36.2.15.2.99.4.1.2
gameNumMoves	1.3.6.1.4.1.36.2.15.2.99.4.1.3
gameStatus	1.3.6.1.4.1.36.2.15.2.99.4.1.4
moveTable	1.3.6.1.4.1.36.2.15.2.99.5
moveEntry	1.3.6.1.4.1.36.2.15.2.99.5.1
moveIndex	1.3.6.1.4.1.36.2.15.2.99.5.1.1
moveByWhite	1.3.6.1.4.1.36.2.15.2.99.5.1.2
moveByBlack	1.3.6.1.4.1.36.2.15.2.99.5.1.3
moveStatus	1.3.6.1.4.1.36.2.15.2.99.5.1.4
chessTraps	1.3.6.1.4.1.36.2.15.2.99.6
moveTrap	1.3.6.1.4.1.36.2.15.2.99.6.1

サブツリー内の要素に関連するすべての要求をこのサブエージェントが処理することを、マスタ・エージェントに知らせるために登録されるのは、この MIB サブツリー・ベースです。

マスタ・エージェントは、登録されている MIB サブツリーに従属するすべてのオブジェクトをサブエージェントが処理することを期待します。この原則を考慮して MIB サブツリーを選択してください。

たとえば、サブエージェントが chess サブツリーの要素に関する要求をすべて処理できると考えるのは現実的なので、chess のサブツリーを登録するのは妥当です。また、特定のアプリケーションはアプリケーション固有の MIB で定義されているすべてのオブジェクトを処理することを期待するため、アプリケーション固有の MIB 全体を登録するのも通常は適切です。

(FDDI や トークンリングなど) transmission に従属するすべての定義済み MIB オブジェクトをサブエージェントが処理できるように用意されてい

ることはありえないので、(MIB-2 の下の) `transmission` のサブツリーを登録するのは間違いです。

サブエージェントは必要な数の MIB サブツリーを登録します。サブエージェントは、そのサブエージェント自身あるいは他のサブエージェントによる別の登録と重複して OID を登録することができます。ただし、サブエージェントは同じ OID を 2 回以上登録することはできません。サブエージェントがマスタ・エージェントとの通信を確立した後は、いつでも MIB サブツリーの登録および登録の解除を行うことができます。

通常、サブツリーとしてマスタ・エージェントに登録されるのは非リーフ・ノードです。ただし、リーフ・ノード(インスタンスと対応する値を持ちうる MIB 変数に相当)、あるいは特定のインスタンスも、サブツリーとして登録することができます。

マスタ・エージェントは、プレフィックスが最長で、最適な優先度の MIB サブツリーを持つサブエージェントへ要求を送ります。

6.2.2 オブジェクト・テーブル

`mosy` および `snmpi` ユーティリティによって、MIB からオブジェクト・テーブルを定義する C 言語コードが生成されます。オブジェクト・テーブルは、サブエージェント内にコンパイルされる `subtree_tbl.h` および `subtree_tbl.c` ファイルで定義されます。

これらのモジュールは、上記のユーティリティによって作成されます。編集しないことをお勧めします。MIB を変更する場合、あるいは将来のバージョンの eSNMP 開発ユーティリティがオブジェクト・テーブルを再作成する場合、これらのファイルに対して編集を加えていなければより簡単にファイルを再作成/再コンパイルできます。

6.2.2.1 `subtree_tbl.h` ファイル

`subtree_tbl.h` ファイルには、次の情報が含まれています。

- MIB サブツリー構造の宣言
- MIB サブツリーにおける各 MIB 変数のためのインデックス定義
- 計数値を持つ MIB 変数のための列挙定義
- MIB グループ・データ構造体定義

- メソッド・ルーチンの関数プロトタイプ

MIB サブツリー構造の宣言

MIB サブツリーは、`subtree_tbl.c` ファイルのコードによって自動的に初期化されます。この構造へのポインタは、マスタ・エージェントに MIB サブツリーを登録するために `esnmp_register` ルーチンへ渡されます。この MIB サブツリーに関するオブジェクト・テーブルへのすべてのアクセスは、このポインタを通して行なわれます。宣言の形式は次のとおりです。

```
extern SUBTREE subtree_subtree;
```

インデックス定義

SUBTREE における各 MIB 変数に対するインデックス定義は、次の形式です。

```
#define I_mib-variable      nnnn
```

これらの値は MIB サブツリー内の各 MIB 変数に対してユニークで、この MIB 変数のためのオブジェクト・テーブルへのインデックスです。また、これらの値は、スイッチ操作でできるように同じメソッド・ルーチンでインプリメントされている変数を区別するのにも使用されます。

列挙定義

“SYNTAX INTEGER” および列挙された値で定義される MIB 変数に対する列挙定義は、次の形式です。

```
#define D_mib-variable_enumeration-name    value
```

列挙定義は、計数 MIB 値がとる可能性のある構造値を記述するため、有用です。次に例を示します。

```
/* enumerations for gameEntry group */
#define D_gameStatus_complete      1
#define D_gameStatus_underway      2
#define D_gameStatus_delete        3
```

データ構造体定義

MIB グループ・データ構造体定義は、次の形式です。

```
typedef struct xxx {
    type mib-variable;
    :
    :
    char mib-variable_mark;
    :
    :
} mib-group_type
```

データ構造体が、MIB サブツリー内の各MIB グループに対して送られます。各構造体定義には、グループ内の各MIB 変数を表すフィールドが含まれています。 *subtree_tbl.h* ファイルが作成されるときに *snmpi* プログラムに渡される MIB のプール内で MIB 変数名がユニークでない場合、それをユニークにするために *snmpi* プログラムは、親変数の名前 (グループ名) で名前を限定しません。MIB 変数フィールドに加えて、各変数に対して 1 バイトの *mib-variable_mark* フィールドが構造体に含まれています。このフィールドは、MIB 変数の状態を管理するために使用することができます。次に示すのは、*chess* MIB のグループ構造体の例です。

```
typedef struct _chess_type {
    OID    chessProductID;
    int    chessMaxGames;
    int    chessNumGames;

    char chessProductID_mark;
    char chessMaxGames_mark;
    char chessNumGames_mark;
} chess_type;
```

MIB グループ構造体はユーザが使用するために提供されますが、必須ではありません。メソッド・ルーチンの中で最も簡単な構造体を使用することができます。

メソッド・ルーチン関数プロトタイプ

MIB サブツリー内の各 MIB グループは、メソッド・ルーチン・プロトタイプを定義してます。MIB グループは、同じ親ノードを持つリーフ・ノードの MIB 変数の集まりです。

Get , *GetNext* , および *GetBulk* 操作を処理するメソッド・ルーチンに対しては関数プロトタイプが常にあります。グループに書き込み可能変数が含まれる場合は、*SET* 操作を処理するメソッド・ルーチン用の関数プロトタイプもあります。これらのルーチンへのポインタは、*subtree_tbl.c*

モジュールで初期化される MIB サブツリーのオブジェクト・テーブルに含まれています。次のように、定義されている各プロトタイプに対してメソッド・ルーチンを作成する必要があります。

```
extern int mib-group_get(  
    METHOD *method);  
extern int mib-group_set(  
    METHOD *method);
```

次に例を示します。

```
extern int chess_get(METHOD *method);  
extern int chess_set(METHOD *method);
```

メソッド・ルーチンについての詳細は、6.3.2.3 項を参照してください。

6.2.2.2 subtree_tbl.c ファイル

subtree_tbl.c ファイルには次の情報が含まれています。

- 各 MIB 変数の OID を表す整数の配列
- OBJECT 構造体の配列 (*esnmp.h* 参照)
- 初期化された SUBTREE 構造体

整数の配列

MIB サブツリーにおける各 MIB 変数の OID として使用される整数の配列は、次の形式です。

```
static unsigned int elems[] = { ...
```

OBJECT 構造体

MIB サブツリー内の各 MIB 変数に対して 1 つの OBJECT があります (*esnmp.h* を参照)。

OBJECT は MIB 変数を表現し、次のフィールドを持ちます。

- *object_index* — *subtree_tbl.h* ファイルからの定数 *I_mib-variable*
- *oid* — 変数の OID (*elems[]* の部分をポイントする)
- *type* — 変数のデータ型
- *getfunc* — Get 操作のために呼び出すメソッド・ルーチンのアドレス

- `setfunc` — Set 操作のために呼び出すメソッド・ルーチンのアドレス

マスタ・エージェントは、オブジェクト・テーブルあるいは MIB 変数についての情報は持っていません。マスタ・エージェントは、MIB サブツリーの登録情報のみを管理します。特定の MIB 変数に対する要求を受け取った場合、次のように処理されます。以下の手順では、MIB 変数は `mib_var` で MIB サブツリーは `subtree_1` です。

1. マスタ・エージェントは、MIB サブツリーの登録情報内で `mib_var` の最も優先度が高い領域として `subtree_1` を見つけます。最も優先度の高い領域は、プレフィックスが最長で、最適な優先度を持つ登録済みの MIB サブツリーとして決定されます。
2. マスタ・エージェントは、`subtree_1` を登録したサブエージェントに eSNMP メッセージを送ります。
3. サブエージェントは登録済み MIB サブツリーのリストを照会して `subtree_1` を探し、`subtree_1` のオブジェクト・テーブルで次のものを探します。
 - `mib_var` (Get および Set 要求の場合)
 - `mib_var` の後の最初のオブジェクト (GetNext あるいは GetBulk 要求の場合)
4. サブエージェントは、適切なメソッド・ルーチンを呼び出します。メソッド・ルーチンが正常に終了すると、マスタ・エージェントへデータが返されます。正常に終了しなかった場合、Get あるいは Set に対してはエラーが返されます。GetNext あるいは GetBulk に対しては、メソッド・ルーチンが正常に終了するかあるいはすべてのオブジェクトを実行するまで、`libesnmp` ライブラリ・コードはその後に続く `subtree_1` のオブジェクト・テーブルのオブジェクトを実行し続けます。どちらの場合も適切な応答が返されます。
5. GetNext あるいは GetBulk ルーチンで `subtree_1` がデータを返せなかったことをマスタ・エージェントが検出した場合、サブエージェントが値を返すか、MIB サブツリーの登録情報がなくなるまで `subtree_1` の後のサブツリーを繰り返し実行します。

初期化済み SUBTREE 構造体

SUBTREE 構造体へのポインタが、MIB サブツリーを登録するために `esnmp_register` eSNMP ライブラリ・ルーチンへ渡されます。ライブラリ・ルーチンは、このポインタを通してオブジェクト構造体を見つけます。次に示すのは chess サブツリー構造体の例です。

```
SUBTREE chess_subtree = { "chess", "1.3.6.1.4.1.36.2.15.2.99",
                          { 11, &elems[0] }, objects, I_moveStatus};
```

SUBTREE 構造体は次の要素を含んでいます。

- *name*
MIB サブツリーのベース・ノードの名前
- *dots*
MIB サブツリー の OID の ASCII 文字列表現。実際に登録される値。
- *oid*
サブツリーのベース・ノードの OID。整数の配列をポイントする。
- *object_tbl*
オブジェクト・テーブルに含まれるオブジェクトの配列へのポインタ。
`subtree_tbl.h` ファイルに記述されている `I_xxxx` 定義によって示される。
- *last*
`object_tbl` ファイルの最後のオブジェクトのインデックス。テーブルの最後に到達したことを検知するために使用されます。

`subtree_tbl.c` の最後のセクションには、`mib-group_type` 構造体を割り当て/解放するための短いルーチンが含まれています。この部分は必ずしも API の一部として提供する必要はありません。

6.2.3 サブエージェントの実現

サブエージェント開発者は、通常、UNIX アプリケーションや、デーモン、ドライバ (たとえば `gated` デーモンや ATM ドライバ) が提供されており、SNMP インタフェースを実現する必要があります。以下に、その手順を示します。

1. MIB 仕様の入手

MIB の開発は MIB 仕様から始まり、通常は RFC の形式です。SNMPv1 については、仕様は RFC 1212 に従って コンサイス MIB フォーマットで作成されます。SNMPv2c については、仕様は、それぞれ RFC 1902 および RFC 1903 で指定されている SMIV2 および textual conventions で作成されます。MIB の設計および仕様決定については、本書では触れません。すでに MIB 仕様をお持ちであることを前提に説明します。

標準の RFC は、次の URL の RFC サイトから入手できます。

<http://www.rfc-editor.org/rfc.html>

ユーザ自身の MIB 仕様を作成する必要がある場合は、他のベンダが作成した同じような MIB を参考にしながら行うことができます。公開 MIB のソースは、次の URL の Network Management ページのアーカイブ・セクションで参照することができます。

<http://smurfland.cit.buffalo.edu/NetMan/index.html>

サブエージェントで実現しようとしているすべての要素に対する MIB、およびそれらが参照する MIB が必要になります。最低限必要なのは、SMIV2 MIB `snmp-smi.my` と textual convention `snmp-tc.my` です。これらは `/usr/examples/esnmp` ディレクトリにあります。

2. MIB をコンパイルします。

MIB 定義を入手したら、それらを使用して新しいサブエージェント用のオブジェクト・テーブルを生成します。目的は、各 MIB の MIB 仕様テキストを取り出し、ASN.1 仕様を抽出して、それらをローカル・オブジェクト・テーブルを含む C 言語モジュールにコンパイルすることです。

次のツールを使用して MIB をコンパイルします。

- `mib-converter.sh`

`mib-converter.sh` は、RFC テキストから MIB ASN.1 定義を抽出するための `awk` シェル・スクリプトです。このステップでは、MIB 定義の前後を削除し、ページ・ヘッダおよびフッタを削除します。

`mib-converter.sh` スクリプトは、不要なものをすべて削除してくれるわけではありません。このため、場合によってはテキスト・エディタを使用して手作業で削除する必要があります。

`mib-converter.sh` スクリプトの使用例を次に示します。

```
# /usr/examples/esnmp/mib-converter.sh mib-def.txt > \
mib-def.my
```

RFC によっては複数の MIB 定義を含む場合がありますので注意してください。 `mib-converter.sh` スクリプトは、単一の MIB 定義を含む RFC に対してのみ使用することができます。 `mosy` コンパイラも、複数の MIB 定義を含むファイルは処理できません。 複数の MIB 定義を含む RFC を使用する場合は、それぞれを別々の入力ファイルに分割します。 抽出された MIB ASN.1 定義を含む結果の出力ファイルは、次のような形式になります。

```
mib-def.my
```

- `mosy`

`mosy` コンパイラは、`mib-converter.sh` スクリプトによって作成された `.my` ファイルを解析し、それらを `.defs` ファイルにコンパイルします。 `.defs` ファイルは、MIB 内のオブジェクト階層構造を記述します。 `.defs` ファイルは、いくつかのツールのフロント・エンドになっています。

`mosy` コンパイラの使用例を次に示します。

```
# mosy mib-def.my
```

`mosy` コンパイラは `mib-def.defs` ファイルを作成します。

この `mosy` プログラムは、4BSD/ISODE SNMPv2 パッケージで配布されている ISODE 8.0 に含まれているものです。

- `snmpi`

MIB データ・イニシャライザ作成プログラム (`snmpi`) は、`mosy` コンパイラによってコンパイルされた `.def` ファイルの連鎖を読み取り、指定された MIB サブツリーのオブジェクト・テーブルの静的な構造を定義する C コードを生成します。

注意

このオペレーティング・システムで提供する `snmpi` プログラムは BSD/ISODE SMUX の `snmpi` プログラムとは異なります。

`mosy` コンパイラによってコンパイルされた `.def` ファイルは、`objects.defs` ファイルに連鎖させてください。 また、コンパイルしたバージョンの `snmp-smi.my` および `snmp-tc.my` を含めてく

ださい。 `objects.defs` ファイルには、サブツリーで使用されていないものも含めて、すべての MIB 名を解決するのに必要な全 MIB を含める必要があります。オブジェクト・テーブル・ファイルは次のコマンドで生成します。

```
# /usr/sbin/snmpd objects.defs subtree
```

`snmpd` プログラムは、`objects.defs` ファイルに含まれているオブジェクトの結果として生成されたツリー全体の内容をダンプするためのプリント・オプションを備えています。サブツリーに問題がある場合、この情報が役に立つ場合があります。リストを生成する場合は次のコマンドを使用してください。

```
# /usr/sbin/snmpd -p objects.defs > objects.txt
```

`snmpd` プログラムは `subtree.tbl.c` および `subtree.tbl.h` を出力します。ここで `subtree` は、MIB サブツリーのベース MIB 変数名です。これらの 2 つのファイルは、指定したサブツリーの MIB オブジェクト・テーブルを初期化するための C コードです (これはローカル・オブジェクト・テーブルです)。サブエージェントで実現されている各 MIB ツリーに対して、この処理を繰り返してください。`snmpd` プログラムは省略時の設定により、メソッド・ルーチンの細分性レベルとして MIB グループを使用するように設定されています。つまり、グループ内のすべての MIB 変数は同じメソッド・ルーチンによってサービスが行われるという前提があります。(このプログラムは、これを支援するための `mib-group_type` データ構造体も提供します。)

`mib-group_type` 構造体は API の一部ではありません。利便性のために提供されています。この構造体は、オブジェクト・テーブルの `mib-group` を使用する際に役立ちます。これは、それらのオブジェクトが論理的に関連していて、通常はグループとしてアクセスされるためです。たとえば `ipRoutes` は、だいたい、カーネルルーティング・テーブルから返されます。

3. メソッド・ルーチンおよび API コールをコーディングします。

マスタ・エージェント (`snmpd`) との通信を初期化し MIB を登録するための eSNMP ライブラリ API を呼び出すコードを作成します (6.2.4 項参照)。

必要なメソッド・ルーチンのためのコードを作成します (6.3 節参照)。通常は、登録された MIB サブツリー内の各 MIB グループに対して、1

つの Get メソッド・ルーチンと 1 つの SET メソッド・ルーチンが必要になります。前のステップで生成された `subtree_tbl.h` ファイルは、各メソッド・ルーチンの名前と関数プロトタイプを定義します。

4. サブエージェントを作成します。

`/usr/examples/esnmp` ディレクトリにサンプルの Makefile (`chess.mk`) が提供されています。

5. サブエージェントの実行とテストを行います。

他のプログラムやデーモンと同じようにサブエージェントを実行します。デバッグ処理でできるように eSNMP ライブラリ・ルーチンにトレース機能が組み込まれています。この機能を使用する場合は、`main` セクションで `set_debug_level` ルーチンを使用してください。

サブエージェントが初期化され、MIB サブツリーが正しく登録されると、HP Insight Manager、HP Openview、あるいはその他の MIB ブラウザなどの標準アプリケーションを使用して、SNMP 要求を送信することができます。SNMP アプリケーションにアクセスできない場合は、`snmp_request`、`snmp_traprcv` および `snmp_trapsnd` プログラムを使用してサブエージェントのデバッグを行うことができます。

対話式デバッグを行うと、サブエージェントが SNMP 要求のタイムアウトの原因になることに注意してください。

通常、すべてのエラーおよび警告メッセージは、システムのデーモン・ログに記録されます。`chess` サブエージェントおよび `os_mibs` サブエージェントを実行する場合、次のようにトレース実行時引数を指定します。

```
os_mibs -trace
```

トレース・オプションが有効な場合、プログラムはデーモンとして実行されず、すべてのトレース出力は `stdout` に送られ、処理される各メッセージを表示します。

この機能は、サブエージェントで `set_debug_level` ルーチンを呼び出し `TRACE` パラメータを渡すことによって使用できます。

デバッグ・マクロで渡されるものは、次のように `stdout` に送られます。

```
ESNMP_LOG ((TRACE, ("message_text \n"));
```

警告メッセージを表示しないですべてをデーモン・ログに送るには、`set_debug_level` ルーチンを呼び出して `WARNING || DAEMON_LOG`

パラメータを渡すか、あるいは、`set_debug_level` ルーチンを呼び出して `ERROR` || `DAEMON_LOG` パラメータを渡します。

6.2.4 サブエージェント・プロトコルの操作

eSNMP API は、マスタ・エージェント (`snmpd`) に密接に結び付かない自主的なサブエージェントのために提供します。サブエージェントは、他のサブシステムあるは製品の 1 部分でもあり、SNMP とは関連しない本来の機能を持っています。たとえば、`gated` デーモンは本来はインターネット・ルーティングに関係していますが、サブエージェントとしても機能します。

特に、`snmpd` デーモンは、スタートアップあるいはシャットダウン・プロセスの中ではどのサブエージェント・デーモンの起動/停止も行いません。また、サブエージェントに関するディスク上の構成情報の保守も行いません。`snmpd` デーモンは、以前のサブエージェントあるいはサブツリーの登録情報については、起動時には何も持っていません。

通常、オペレーティング・システム上の各デーモンは、システムの実行レベルが変わる際にすべて一緒に開始あるいは停止します。しかしサブエージェントは、`snmpd` デーモンの前にそれらをどこで開始するか、あるいは構成ファイルから情報を再ロードするために `snmpd` デーモンを再起動する間それらをどこで実行しているか、正しく状況処理する必要があります。このような状況では、サブエージェントは、次の項で説明するように eSNMP プロトコルを再起動する必要があります。

6.2.4.1 操作の順序

一連のサブエージェント・プロトコルの操作は次のとおりです。

1. 初期化 (`esnmp_init`)
2. 複数のサブエージェント間で共用されるテーブルで必要なインデックスの割り当て (`esnmp_allocate`)
3. 登録 (`esnmp_register [esnmp_register ...]`)
4. データ通信

次のようなループを繰り返し実行します。

```
{
    determine sockets with data pending
    if the eSNMP socket has data pending
```

```

    esnmp_poll

    periodically call esnmp_are_you_there as required
    during periods of inactivity
}

```

5. 終了 (esnmp_term)

サブエージェントを終了する際に `esnmp_term` 関数を呼び出すことは非常に重要です。この操作によって eSNMP は、サブエージェントが使用していたシステム・リソースを解放します。

サブエージェント・プロトコル操作のコーディング方法については、`/usr/examples/esnmp` ディレクトリのサンプル・サブエージェントを参照してください。

6.2.4.2 関数の戻り値

eSNMP API 関数の戻り値は、要求された操作の成功あるいは失敗と、マスタ・エージェントの状態を、サブエージェントに示します。以下に、各戻り値の意味とサブエージェントの対処を示します。

- `ESNMP_LIB_OK`
操作が成功しました。
- `ESNMP_LIB_NO_CONNECTION`
サブエージェントとマスタ・エージェントを接続できません。この値は、`esnmp_init` 関数によって返されます。
 - 原因
マスタ・エージェントが実行されていません。あるいは応答がありません。
 - 対処
一定時間経過後 `esnmp_init` を呼び出してプロトコルを再起動します。
- `ESNMP_LIB_BAD_ALLOC`
1 つまたは複数のインデックスを割り当てることができません。この値は、`esnmp_allocate` 関数によって返されます。
 - 原因は次のとおりです。

- esnmp_allocate 関数を呼び出す前に、esnmp_init 関数が正常に呼び出されていません。
 - esnmp_allocate 関数のいずれかのパラメータが不正です。ログ・ファイルを参照して、不正なパラメータを調べてください。
- 対処

正しい順序と正しい引数で esnmp_allocate 関数を呼び出します。
- ESNMP_LIB_BAD_DEALLOC

1 つまたは複数のインデックスを割り当て解除できません。この値は、esnmp_deallocate 関数によって返されます。

 - 原因は次のとおりです。
 - esnmp_deallocate 関数を呼び出す前に、esnmp_init 関数が正常に呼び出されていません。
 - esnmp_deallocate 関数のいずれかのパラメータが不正です。ログ・ファイルを参照して、不正なパラメータを調べてください。
 - 対処

正しい順序と正しい引数で esnmp_deallocate 関数を呼び出します。
- ESNMP_LIB_LOST_CONNECTION

マスタ・エージェントとの通信が途切れしました。この値は、esnmp_register, esnmp_poll, esnmp_are_you_there, esnmp_unregister, あるいは esnmp_trap 関数によって返されます。

 - 原因

マスタ・エージェント・ソケットへのパケット送信が失敗しました。通常は、マスタ・エージェントの異常終了が原因です。
 - 対処

一定時間経過後、esnmp_init を呼び出してプロトコルを再起動します。
- ESNMP_LIB_BAD_REG

登録の送信が失敗しました。この値は、`esnmp_register`、`esnmp_unregister`、あるいは `esnmp_poll` 関数が返します。

– 原因は次のとおりです。

- ☐ `esnmp_register` 関数を呼び出す前に、`esnmp_init` 関数が正しく呼び出されていません。
- ☐ `esnmp_register` 関数の `timeout` パラメータが正しくありません。
- ☐ `esnmp_register` に渡されたサブツリーはすでに登録のためにキューイングされています。あるいは、すでにこのサブエージェントによって登録されています。
- ☐ 以前の登録が失敗しています (`esnmp_poll` 関数によって返された場合)。失敗の原因および失敗したサブツリーについては、ログ・ファイルを参照してください。
- ☐ 登録されていないサブツリーの登録を解除しようとしています (`esnmp_unregister` 関数によって返された場合)。

– 対処

正しい引数を指定して、適切な順序で `esnmp_register` を呼び出します。

- `ESNMP_LIB_CLOSE`

マスタ・エージェントが停止しています。この値は、`esnmp_poll` 関数によって返されます。

– 原因

マスタ・エージェントを命令に従ってシャットダウンしています。

– 対処

`esnmp_init` 関数でプロトコルを再起動します。

- `ESNMP_LIB_NOTOK`

eSNMP プロトコル・エラーが発生して、パケットが廃棄されました。この値は、`esnmp_poll` あるいは `esnmp_trap` 関数によって返されます。

– 原因

サブエージェント内のメモリ・リソースの不足が原因でパケット・レベルのプロトコル・エラーが発生しています。

- 対処

続行します。

6.3 拡張 **SNMP** アプリケーション・プログラミング・インタフェース

以降の節では、SNMP アプリケーション・プログラミング・インタフェースに関して、次の項目について詳しく説明します。

- 呼び出しインタフェース
- メソッド・ルーチン呼び出しインタフェース
- `libesnmp` サポート・ルーチン

6.3.1 呼び出しインタフェース

呼び出しインタフェースには次のルーチンが含まれます。

- `esnmp_init`
- `esnmp_allocate`
- `esnmp_deallocate`
- `esnmp_register`
- `esnmp_register2`
- `esnmp_unregister`
- `esnmp_unregister2`
- `esnmp_capabilities`
- `esnmp_uncapabilities`
- `esnmp_poll`
- `esnmp_are_you_there`
- `esnmp_trap`
- `esnmp_term`
- `esnmp_systime`

6.3.1.1 esnmp_init ルーチン

`esnmp_init` ルーチンは、eSNMP サブエージェントをローカルに初期化し、マスタ・エージェントとの通信を開始します。

このコールは、マスタ・エージェントからの応答を待つ間ブロックは行いません。`esnmp_init` ルーチンを呼び出したら、サブエージェントが扱う各 MIB サブツリーに対して `esnmp_register` ルーチンを呼び出します。

このルーチンは、プログラム初期化中か、eSNMP プロトコルの再起動のために呼び出してください。再起動の場合、`esnmp_init` ルーチンはすべての登録をクリアするので各サブツリーを再登録する必要があります。

プログラム名 (`argv[0]`) に説明的なテキストを追加して、ユニークな `subagent_identifier` を作成するようにします。

`esnmp_init` ルーチンの構文は次のとおりです。

```
int esnmp_init(  
    int *socket,  
    char *subagent_identifier);
```

引数は次のとおりです。

socket

eSNMP が使用するソケット記述子を受信する整数のアドレス。

subagent_identifier

サブエージェントを識別するヌルで終る文字列 (通常はプログラム名) のアドレス。

戻り値は次のとおりです。

ESNMP_LIB_NO_CONNECTION

初期化できないかあるいはマスタ・エージェントと通信できません。
後で再度実行してください。

ESNMP_LIB_OK

`esnmp_init` ルーチンが正常に終了しました。

ESNMP_LIB_NOTOK

サブエージェントにメモリを割り当てられませんでした。

次に示すのは `esnmp_init` ルーチンの例です。

```
#include <esnmp.h>
int socket;
status = esnmp_init(&socket, "gated");
```

6.3.1.2 `esnmp_allocate` ルーチン

`esnmp_allocate` ルーチンは、1 つまたは複数の特定のインデックス・オブジェクトに対する値の割り当てを要求します。

インデックスの割り当ては、AgentX に準拠したマスタ・エージェントが提供するサービスです。これは、お互いに関する知識を持たないサブエージェント間で、MIB の概念的テーブルを共用するための機能です。マスタ・エージェントはインデックス・オブジェクト (OID) と、各インデックスに割り当てられている値からなるデータベースを保守します。マスタ・エージェントは、各インデックスが表す MIB 変数 (ある場合) を意識しません。規約により、サブエージェントの開発者は、INDEX 句にリストされている MIB 変数を、値を割り当てるインデックス・オブジェクトとして使用する必要があります。

複数の変数でインデックスされるテーブルでは、各インデックスに対してそれぞれ値を割り当てることができますが、多くの場合これは不要です。サブエージェントは、次の割り当てタイプのいずれかで要求することができます。

- 特定のインデックス値
- 現在割り当てられていないインデックス値
- 一度も割り当てられていないインデックス値

2 番目と 3 番目のタイプは、多くの MIB 仕様で、不特定の整数インデックスに対し一意性と不変性が要求されるために必要になります。たとえば、IF-MIB (RFC 2233) の `ifIndex`、FDDI MIB (RFC 1285) の `snmpFddiSMTIndex`、System Application MIB (RFC 2287) の `sysApplInstallPkgIndex` があります。サブエージェントがこのようなインデックスを用いてテーブルを共用しなければならないということが、インデックス割り当てメカニズムが必要な主な理由です。

インデックス割り当てと MIB リージョン登録は、マスタ・エージェント内で連動していません。マスタ・エージェントは、登録要求を処理する際に、インデックス割り当ての現在の状態を考慮しません。また、インデックスの割

り当て要求を処理する際に、現在の登録状況を考慮することはありません。これは主に、非 AgentX のサブエージェントの便宜を図るためです。

サブエージェントの開発者は、まずインデックスの割り当てを要求し、次に対応するリージョンを登録しなければなりません。この順序で実行すると、インデックス割り当て要求に成功したときに、サブエージェントは、何を登録できるかということに対する適切なヒント (ただし保証ではない) を得ることができます。他のエージェントがテーブルのその row にすでに登録しているために、登録に失敗することがあります。

サブエージェントは、共用テーブルの概念的な row を次の順序で登録する必要があります。

1. インデックス値を正常に割り当てます。
2. 割り当てられたインデックス値を `ESNMP_REG` 構造体の `instance` フィールドに設定し、`esnmp_register2` ルーチンに渡します。このようにして、eSNMP サブエージェント開発者は、サブツリーとその `ESNMP_REG` 構造体の任意の `range_subid` および `range_upper_bound` フィールドで指定される MIB リージョンを完全に限定できます。サブエージェントの開発者は、インスタンスにより登録を行う場合は、`priority` フィールドを 255 に設定しなければなりません。
3. 結果コード `ESNMP_REG_STATE_REGDUP` で登録に失敗した場合は、以前に割り当てたインデックス値を `esnmp_deallocate` ルーチンでこの row から割り当て解除し、ステップ 1 からもう一度処理を始めます。

インデックス割り当ては、インデックスが任意の値であり、サブエージェント開発者が、どのインデックス値を使えばよいか判断できない場合にのみ必要になります。インデックス値に固有の意味がある場合、サブエージェントではそのインデックス値を割り当てないようにします。たとえば、RFC 1514 では、実行中のソフトウェア・プロセスのテーブル (`hrSWRunTable`) は、システム固有のプロセス識別子 (`pid`) によってインデックスされます。サブエージェント自身のプロセスに対応する `hrSWRunTable` の row を実装するサブエージェントでは、その row のオブジェクト・インスタンスを定義するリージョンを登録します。インデックス値を割り当てる必要はありません。

`esnmp_allocate` ルーチンの構文は次のとおりです。

```
int esnmp_allocate(  
    ESNMP_ALLOC *alloc_parm);
```

引数の定義は次のとおりです。

alloc_parm

ESNMP_ALLOC 構造体へのポインタ。呼び出す側は、この構造体とこれから参照される VARBIND リストを (メモリ内に) 継続的に確保しなければなりません。これは、eSNMP の実行時ライブラリが、マスタ・エージェントから返される、インデックス割り当て要求の結果をフィールドに格納するために必要です。この構造体には、次のようなフィールドがあります。

vb

1 つまたは複数の VARBIND からなる変数バインディング・リストへのポインタ。リスト内の VARBIND にはそれぞれ、値を割り当てるインデックス・オブジェクトの名前が含まれています。各 VARBIND にはそれぞれ値が必要です。

ESNMP_ALLOC_ANY_INDEX フラグと ESNMP_ALLOC_NEW_INDEX フラグのどちらも指定されていない場合、マスタ・エージェントは、インデックス割り当て要求を処理する際に、指定された値すべてを使用します。ESNMP_ALLOC_ANY_INDEX フラグまたは ESNMP_ALLOC_NEW_INDEX フラグが指定された場合、マスタ・エージェントは、インデックス割り当て要求を処理する際に、指定された値をすべて無視し、各 VARBIND には適切な値を生成します。

options

次の値のビットマスク。

ESNMP_ALLOC_CLUSTER

インデックス割り当て要求は、クラスタ・コンテキストに対するものです。

ESNMP_ALLOC_NEW_INDEX

このインデックス割り当て要求では、新しい値の割り当てを要求します。マスタ・エージェントは、実行を開始して以来使用されていない VARBIND に対してそれぞれ値を生成します。これらの値は、成功応答で返されます。

ESNMP_ALLOC_ANY_INDEX

このインデックス割り当て要求では、未使用の値の割り当てを要求します。マスタ・エージェントは、実行を開始してから使用されたか使用されていないかに関係なく `VARBIND` に対してそれぞれ値を生成します。これらの値は、成功応答で返されます。

status

次に示す整数値のいずれかをとり、呼び出し側に対して、インデックス割り当て要求の状態を示します。この状態コードは非同期に更新されます。 `esnmp_poll` ルーチンから戻った後、呼び出し側は、このパラメータを調べることができます。以下の状態コードでは、`alloc->error_index` フィールドには値ゼロ (0) が格納されています。

ESNMP_ALLOC_STATE_PENDING

インデックス割り当て要求は、現在ローカルに保留されており、マスタ・エージェントへの接続を待っています。

ESNMP_ALLOC_STATE_SENT

インデックス割り当て要求は、マスタ・エージェントに送信されました (最終状態はまだ保留)。

ESNMP_ALLOC_STATE_DONE

マスタ・エージェントは、インデックス割り当て要求の処理に成功し肯定応答を返しました。これでサブエージェントは、`esnmp_register2` 呼び出しに渡す `esnmp_reg->instance` フィールドで、割り当てられたインデックス値を使用できます。

以下の状態コードでは、`alloc->error_index` フィールドにはゼロ以外の値が格納されています。

ESNMP_ALLOC_STATE_ALLOCTYPE

インデックス・タイプまたはオプションが不正のため、マスタ・エージェントはインデックス割り当て要求をリジェクトしました。

ESNMP_ALLOC_STATE_ALLOCINUSE

指定されたインデックス・オブジェクトの値が現在使用されているため、マスタ・エージェントはインデックス割り当て要求をリジェクトしました。

ESNMP_ALLOC_STATE_ALLOCAVAIL

要求されたインデックス・オブジェクトに対して使用できる値がないため、マスタ・エージェントはインデックス割り当て要求をリジェクトしました。この状態は、ESNMP_ALLOC_NEW_INDEX または ESNMP_ALLOC_ANY_INDEX オプションを指定した場合にのみ返されます。

ESNMP_ALLOC_STATE_ALLOCNOCLU

クラスタ・コンテキスト・オプションをサポートしていないため、マスタ・エージェントはインデックス割り当て要求をリジェクトしました。

ESNMP_ALLOC_STATE_REJ

マスタ・エージェントはその他の理由でインデックス割り当て要求をリジェクトしました。

error_index

状態コードが `alloc.vb varbind` リスト内のどの `VARBIND` (1 から開始) に対応しているかを示します。 `esnmp_poll` ルーチンから戻った後、呼び出し側はこのパラメータを調べることができます。

戻り値は次のとおりです。

ESNMP_LIB_NOTOK

`alloc_parm` 引数が指定されていません。

ESNMP_LIB_OK

`esnmp_allocate` ルーチンは正常に終了しました。

ESNMP_LIB_LOST_CONNECTION

サブエージェントとマスタ・エージェントの通信が途切れしました。

ESNMP_LIB_BAD_ALLOC

サブエージェントとマスタ・エージェントが接続されていないか、
`alloc_parm` 引数によって指定された `VARBIND` に有効なものがありま
せん。ログ・ファイルにもメッセージが出力されます。

この戻り値は、インデックス割り当て要求の開始のみを示していること
に注意してください。マスタ・エージェントからの応答で返された実際
の状態コードは、その後 `esnmp_poll` ルーチン呼び出したときに、
`alloc->status` および `alloc->error_index` フィールドに返されます。

`esnmp_allocate` ルーチンの例を、次に示します。

```
#define INDENT_SIZE          4
#define RESPONSE_TIMEOUT    0 /* use the default time set
                                in esnmp_init message */
#define REGISTRATION_PRIORITY 128 /* priority at which the MIB
                                   subtree will register */
#define RANGE_SUBID         10 /* the identifier position in
                                oid->elements just after ifEntry */
#define RANGE_UPPER_BOUND   22 /* the identifier for ifSpecific, under ifEntry */

int rc, status;
unsigned int our_ifIndex_instance = 0;
int ready_to_register = 0;
int have_a_good_registration = 0;
extern SUBTREE ifEntry_subtree; /* generated by /usr/sbin/snmpi -r ifEntry */
OBJECT *ifIndex_object = &ifEntry_subtree.object_tbl[I_ifIndex];
static ESNMP_ALLOC esnmp_alloc_for_ifIndex; /* retain this structure to obtain status
                                             code and index object values. Also,
                                             retain this structure for a subsequent
                                             call to esnmp_deallocate */

static ESNMP_REG esnmp_reg_for_ifEntry; /* retain this structure for a subsequent
                                         call to esnmp_unregister2 */

static OID ifEntry_instance_oid;
VARBIND *vb;
/*
 * initialize the ESNMP_ALLOC structure
 */
memset(&esnmp_alloc_for_ifIndex, 0, sizeof(ESNMP_ALLOC));

esnmp_alloc_for_ifIndex.options = ESNMP_ALLOC_NEW_INDEX;
esnmp_alloc_for_ifIndex.vb = vb = (VARBIND *)malloc(sizeof(VARBIND));
bzero((char *)vb, sizeof(VARBIND));
clone_oid(&vb->name, &ifIndex_object->oid);
o_integer(vb, ifIndex_object, (unsigned long)0); /* master will return actual
                                                  value assigned */

while(!have_a_good_registration) {
    status = esnmp_allocate(&esnmp_alloc_for_ifIndex );
    if (status != ESNMP_LIB_OK) {
        printf("Could not queue the 'ifIndex' \n");
        printf("index object for index allocation\n");
    }
    .
    .
    .
    rc = esnmp_poll();
    .
    .
    if (esnmp_alloc_for_ifIndex.status > ESNMP_ALLOC_STATE_SENT) {
        /*
         * esnmp_alloc_for_ifIndex.status nows contain the final
         * status from the master agent.
         */
        switch(esnmp_alloc_for_ifIndex.status) {
            case ESNMP_ALLOC_STATE_DONE
                our_ifIndex_instance = esnmp_alloc_for_ifIndex.vb->value.ul;
```

```

        ready_to_register = 1;
        printf("\n*** Successful index allocation. Our conceptual row in the");
        printf("\n*** interfaces table was allocated. ifIndex value is %i\n\n",
            our_ifIndex_instance);
        break;
    case ESNMP_ALLOC_STATE_ALLOCTYPE:
        printf("\n*** Failed index allocation - 'allocation type'");
        printf("\n*** associated with supplied varbind #i.\n\n",
            esnmp_alloc_for_ifIndex.error_index);
        break;
    case ESNMP_ALLOC_STATE_ALLOCLINUSE:
        printf("\n*** Failed index allocation - 'allocation in use'");
        printf("\n*** associated with supplied varbind #i.\n\n",
            esnmp_alloc_for_ifIndex.error_index);
        break;
    case ESNMP_ALLOC_STATE_ALLOCAVAIL:
        printf("\n*** Failed index allocation - 'no available values'");
        printf("\n*** associated with supplied varbind #i.\n\n",
            esnmp_alloc_for_ifIndex.error_index);
        break;
    case ESNMP_ALLOC_STATE_ALLOCNOCCLU:
        printf("\n*** Failed index allocation - 'cluster context not supported'");
        printf("\n*** associated with supplied varbind #i.\n\n",
            esnmp_alloc_for_ifIndex.error_index);
        esnmp_alloc_for_ifIndex.options &= ~ESNMP_ALLOC_CLUSTER;
        break;
    case ESNMP_ALLOC_STATE_REJ:
        printf("\n*** failed index allocation - 'other reasons'");
        printf("\n*** associated with supplied varbind #i.\n\n",
            esnmp_alloc_for_ifIndex.error_index);
        break;
    } /* End switch */
} /* End if */
if (ready_to_register) {
    vb = esnmp_alloc_for_ifEntry.vb;
    memset(&esnmp_reg_for_ifEntry, 0, sizeof(ESNMP_REG));
    esnmp_reg_for_ifEntry.subtree = &ifEntry_subtree;
    esnmp_reg_for_ifEntry.priority = REGISTRATION_PRIORITY;
    esnmp_reg_for_ifEntry.timeout = RESPONSE_TIMEOUT;
    esnmp_reg_for_ifEntry.range_subid = RANGE_SUBID;
    esnmp_reg_for_ifEntry.range_upper_bound = RANGE_UPPER_BOUND;

    ifEntry_instance_oid.nelem = 1;
    ifEntry_instance_oid.elements = &our_ifIndex_instance;
    esnmp_reg_for_ifEntry.instance = (OID *)malloc(sizeof(OID));
    esnmp_reg_for_ifEntry.instance = clone_oid(esnmp_reg_for_ifEntry.instance,
        &ifEntry_instance_oid);
    status = esnmp_register2(&esnmp_reg_for_ifEntry);
    if (status != ESNMP_LIB_OK) {
        printf("Could not queue the registration for 'ifEntry'\n");
    }
    else {
        :
        :
        rc = esnmp_poll();
        :
        :
        if (esnmp_reg_for_ifEntry.state > ESNMP_REG_STATE_SENT) {
            /*
             * esnmp_reg_for_ifEntry.status nows contain the final
             * status from the master agent.
             */
            switch(esnmp_reg_for_ifEntry.state) {
                case ESNMP_REG_STATE_DONE:
                    printf("\n*** Successful registration for conceptual row in");
                    printf("\n*** the interfaces table indexed by an ifIndex");
                    printf("\n*** value of %i.\n\n", our_ifIndex_instance);
                    have_a_good_registration = 1;
                    break;
                case ESNMP_REG_STATE_REGDUP:
                    printf("\n*** Failed registration - Duplicate registration.");
                    printf("\n*** We need to deallocate this ifIndex value and");
                    printf("\n*** allocate a new value\n\n");
                    break;
                case ESNMP_REG_STATE_REGNOCCLU:

```

```

        printf("\n*** Failed registration - Cluster context not supported.");
        printf("\n*** Need to deallocate this ifIndex value and to");
        printf("\n*** allocate a new value in the default context\n\n");
        esnmp_alloc_for_ifIndex.options &= ~ESNMP_ALLOC_CLUSTER;
        break;
    case ESNMP_REG_STATE_REJ:
        printf("\n*** Failed registration - Other reasons\n\n");
        break;
    }
    if (!have_a_good_registration) {
        esnmp_deallocate(&esnmp_alloc_for_ifIndex);
        ready_to_register = 0;
        rc = esnmp_poll();
        free_oid(esnmp_reg_for_ifEntry.instance);
        free(esnmp_reg_for_ifEntry.instance);
        esnmp_reg_for_ifEntry.instance = NULL;
    }
}
}
}
}

```

6.3.1.3 esnmp_deallocate ルーチン

esnmp_deallocate ルーチンは、1 つまたは複数のインデックス・オブジェクトについて、このサブエージェントに以前に割り当てた値の割り当てを解除するよう、マスタ・エージェントに要求します。

マスタ・エージェントはインデックスの割り当て解除要求を受けると、インデックス・オブジェクト (OID) のデータベースを更新し、指定された値に割り当て解除のマークを付けます。ここで解放された値は、その後のインデックス割り当て要求での割り当てに使用できます。

esnmp_deallocate ルーチンの構文は、次のとおりです。

```
int esnmp_deallocate(
    ESNMP_ALLOC *alloc_parm);
```

引数の定義は次のとおりです。

alloc_parm

ESNMP_ALLOC 構造体へのポインタ。通常、これは前に呼び出した esnmp_allocate で使用された構造体と同じものです。呼び出す側は、この構造体とこれから参照される VARBIND リストを (メモリ内に) 継続的に確保しなければなりません。これは、eSNMP の実行時ライブラリが、マスタ・エージェントから返される、インデックス割り当て解除要求の結果をフィールドに格納するために必要です。この構造体には、次のようなフィールドがあります。

vb

1 つまたは複数の VARBIND からなる変数バインディング・リストへのポインタ。リスト内の VARBIND にはそれぞれ、割り当て済みインデックス・オブジェクトの名前と値が含まれています。

options

次の値のビットマスク。

ESNMP_ALLOC_CLUSTER

インデックス割り当て要求は、クラスタ・コンテキストに対するものです。

status

次に示す整数値のいずれかをとり、呼び出し側に対して、インデックス割り当て解除要求の状態を示します。この状態コードは非同期に更新されます。 `esnmp_poll` ルーチンから戻った後、呼び出し側は、このパラメータを調べることができます。以下の状態コードでは、`alloc->error_index` フィールドには値ゼロ (0) が格納されています。

ESNMP_ALLOC_STATE_SENT

インデックス割り当て解除要求は、マスタ・エージェントに送信されました (最終状態はまだ保留)。

ESNMP_ALLOC_STATE_DONE

マスタ・エージェントは、インデックス割り当て解除要求の処理に成功し肯定応答を返しました。これでマスタ・エージェントは、その後のインデックス割り当て要求を処理する際に、このインデックス値を再度使用できるようになります。

以下の状態コードでは、`alloc->error_index` フィールドにはゼロ以外の値が格納されています。

ESNMP_ALLOC_STATE_DEALLOC_REJ

割り当てが未知であるため、マスタ・エージェントはインデックス割り当て解除要求をリジェクトしました。

ESNMP_ALLOC_STATE_REJ

マスタ・エージェントはその他の理由でインデックス割り当て解除要求をリジェクトしました。

error_index

状態コードが *alloc.vb* リスト内のどの VARBIND (1 から開始) に対応しているかを示します。 *esnmp_poll* ルーチンから戻った後、呼び出し側はこのパラメータを調べることができます。

戻り値は次のとおりです。

ESNMP_LIB_NOTOK

alloc_parm 引数が指定されていません。

ESNMP_LIB_OK

esnmp_allocate ルーチンは正常に終了しました。

ESNMP_LIB_LOST_CONNECTION

サブエージェントとマスタ・エージェントの通信が途切れました。

ESNMP_LIB_BAD_DEALLOC

サブエージェントとマスタ・エージェントが接続されていないか、*alloc_parm* 引数によって指定された VARBIND に有効なものがありません。 ログ・ファイルを参照してください。

この戻り値は、インデックス割り当て解除要求の開始のみを示していることに注意してください。 マスタ・エージェントからの応答で返された実際の状態コードは、その後 *esnmp_poll* ルーチンを呼び出したときに、*alloc->status* および *alloc->error_index* フィールドに返されます。

esnmp_deallocate ルーチンの例を、次に示します。

```
#include <esnmp.h>

int status;
static ESNMP_ALLOC esnmp_alloc_for_ifIndex; /* structure retained from a previous call
                                              to esnmp_allocation(). Retain this
                                              structure for update with final status
                                              from the master agent */

/* call to unregister2() goes here */

esnmp_alloc_for_ifIndex.options = 0; /* clear options */
status = esnmp_deallocate( &esnmp_alloc_for_ifIndex );
if (status != ESNMP_LIB_OK) {
```

```

    printf("Could not queue the 'ifIndex' \n");
    printf("index object for index allocation\n");
}

:
:

esnmp_poll();

:
:

if (esnmp_alloc_for_ifIndex.status > ESNMP_ALLOC_STATE_SENT) {
    /*
     * the final status from the master agent is available
     */
    switch(esnmp_alloc_for_ifIndex.status) {
        case ESNMP_ALLOC_STATE_DONE:
            printf("Successful index deallocation for value(s) associated\n");
            printf("with the 'ifIndex' index object.\n");
            free_varbind(esnmp_alloc_forifIndex.vb);
            break;
        case ESNMP_ALLOC_STATE_DEALLOC_REJ:
            printf("Failed index deallocation due to 'unknown allocation'\n");
            printf("associated with supplied varbind #i.\n", esnmp_alloc_for_ifIndex.error_index);
            break;
        case ESNMP_ALLOC_STATE_REJ:
            printf("Failed index deallocation due to 'other reasons'\n");
            printf("associated with supplied varbind #i.\n", esnmp_alloc_for_ifIndex.error_index);
            break;
    }
}
}

```

6.3.1.4 esnmp_register ルーチン

esnmp_register ルーチンは、単一の MIB サブツリーの登録を要求します。これは、サブエージェントが登録された MIB サブツリー内で MIB 変数のインスタンスを生成することをマスタ・エージェントに示します。

esnmp_register を呼び出す前に、初期化ルーチン (esnmp_init) を呼び出す必要があります。処理しようとする各 MIB サブツリーに対応するそれぞれの SUBTREE 構造体に対して、esnmp_register ルーチンを呼び出す必要があります。esnmp_unregister を呼び出すと MIB サブツリーの登録が解除され、その後 esnmp_register を呼び出すと再登録されます。

esnmp_init を呼び出して eSNMP プロトコルを再起動すると、すべての MIB サブツリーの登録がクリアされます。この場合、すべての MIB サブツリーを再登録する必要があります。

MIB サブツリーは、MIB 変数名と対応する OID によって識別されます。この集合は、MIB サブツリーに含まれるすべての MIB 変数の親に相当します。たとえば、MIB-2 tcp サブツリーの OID は 1.3.6.1.2.1.6 です。この OID に従属する (同じ 7 つの識別子で始まる) すべての MIB 変数がサブツリーの領域に含まれています。MIB サブツリーは、単一の MIB 変数 (リーフ・ノード) でも特定のインスタンスでもかまいません。

MIB サブツリーを登録すると、サブエージェントは、その MIB サブツリーの領域内のすべての MIB 変数 (もしくは OID) に対する SNMP 要求を処理することを表示します。したがって、サブエージェントは MIB 変数をまだ組み込んでいる MIB サブツリーの中で最も完全に (長く) 修飾されたものを登録します。

マスタ・エージェントは、サブエージェントが同じ MIB サブツリーを 2 回以上登録できないようにします。この制限事項を除いて、サブエージェントは、以前に登録した MIB サブツリーの OID 範囲に重複して、あるいは他のサブエージェントが登録した MIB サブツリーの OID 範囲に重複して MIB サブツリーを登録できます。

たとえば、`os_mibs` および `gated` の 2 つの Tru64 UNIX デーモンを考えてみると、`os_mibs` デーモンは `ip` MIB サブツリー (1.3.6.1.2.1.4) を登録し、`gated` デーモンは `ipRouteEntry` MIB サブツリー (1.3.6.1.2.1.4.21.1) を登録します。`ipRouteIfIndex` (1.3.6.1.2.1.4.21.1.2) などの `ipRouteEntry` 内の `ip` MIB 変数に対する要求は、`gated` サブエージェントへ送られます。`ipNetToMediaIfIndex` (1.3.6.1.2.1.4.22.1.1) などの他の `ip` 変数への要求は、`os_mibs` サブエージェントへ送られます。`gated` プロセスを終了した場合あるいは `ipRouteEntry` MIB サブツリーの登録を解除した場合、すべての `ipRouteEntry` MIB 変数を含む `ip` MIB サブツリーが `ipRouteIfIndex` に対する要求のもっとも優先度が高い領域となっているので、その後の `ipRouteIfIndex` への要求は `os_mibs` サブエージェントへ送られます。

マスタ・エージェントが `SIGUSR1` シグナルを受け取ると、その MIB 登録が `/usr/tmp/snmpd_dump.log` ファイルへ送られます。詳細については `snmpd(8)` を参照してください。

`esnmp_register` ルーチンの構文は次のとおりです。

```
int esnmp_register(  
    SUBTREE *subtree,  
    int timeout,  
    int priority);
```

引数の定義は次のとおりです。

subtree

処理する MIB サブツリーに対応する `SUBTREE` 構造体へのポインタ。`SUBTREE` 構造体は、MIB ドキュメントから直接得られた `mosy` および `snmpi` ユーティリティで生成されたコードで (`xxx_tbl.c` およ

び `xxx_tbl.h` , `xxx` は MIB サブツリー名) 外部的に宣言され初期化されます。

注意

`subtree` フィールドでポイントされているすべてのメモリは、プログラムの存続期間中、`libesnmplib` によって参照されるため、永続的なストレージを持っていなければなりません。`snmpd` ユーティリティによって発行されたデータ宣言を使用してください。

timeout

この MIB サブツリーでデータを要求した際にマスタ・エージェントが応答を待つ秒数。0 ~ 300 の値を指定します。0 を指定した場合は、省略時のタイムアウト (3 秒) が使用されます。省略時の値を使用することをお勧めします。

priority

登録の優先順位。番号の大きいエントリの方が優先順位が高くなります。指定できる範囲は 1 ~ 255 です。OID (オブジェクトID) の範囲内で最も高い優先順位の MIB サブツリーを登録したサブエージェントが、その範囲の OID (オブジェクト識別子) に対するすべての要求を受け取ります。

同じ優先順位で登録されている MIB サブツリーは、重複しているとされ、登録はマスタ・エージェントによってリジェクトされます。

`priority` 引数は、異なる構成を処理する協力関係のある複数のサブエージェントのためのメカニズムです。

戻り値は次のとおりです。

`ESNMP_LIB_OK`

`esnmplib_register` ルーチンが正常に終了しました。

`ESNMP_LIB_BAD_REG`

`esnmplib_init` ルーチンが呼び出されていない、タイムアウト・パラメータが正しくない、あるいはその MIB サブツリーがすでに登録されています。

ESNMP_LIB_LOST_CONNECTION

サブエージェントとマスタ・エージェントとの通信が途絶えています。

状態は要求の開始のみを示します。マスタ・エージェントの応答で返される実際の状態は、その後の `esnmp_poll` ルーチンへの呼び出しで返されます。

次に示すのは `esnmp_register` ルーチンの例です。

```
#include <esnmp.h>
#define RESPONSE_TIMEOUT      0      /* use the default time set
                                       in OPEN message */
#define REGISTRATION_PRIORITY 10     /* priority at which subtrees
                                       will register */
int status;

extern SUBTREE ipRouteEntry_subtree;

status = esnmp_register( &ipRouteEntry_subtree,
                        RESPONSE_TIMEOUT,
                        REGISTRATION_PRIORITY );
if (status != ESNMP_LIB_OK) {
    printf ("Could not queue the 'ipRouteEntry' \n");
    printf ("subtree for registration\n");
}
```

6.3.1.5 esnmp_unregister ルーチン

`esnmp_unregister` ルーチンは、マスタ・エージェントから MIB サブツリーの登録を解除します。

このルーチンは、この MIB サブツリーの変数に対する要求をそれ以上処理しないことを eSNMP サブエージェントに伝えるアプリケーション・コードによって呼び出されます。`esnmp_register` ルーチンを呼び出すことによって、必要に応じて MIB サブツリーを後で再登録することができます。

`esnmp_unregister` ルーチンの構文は次のとおりです。

```
int esnmp_unregister(
    SUBTREE *subtree);
```

引数は次のとおりです。

subtree

登録を解除する MIB サブツリーの SUBTREE 構造体へのポインタ。

戻り値は次のとおりです。

ESNMP_LIB_OK

ルーチンが正常に終了しました。

ESNMP_LIB_BAD_REG

指定した MIB サブツリーが登録されていません。

ESNMP_LIB_LOST_CONNECTION

MIB サブツリーの登録を解除する要求が送信できませんでした。 プロトコルを再起動する必要があります。

次は、esnmp_unregister ルーチンの例です。

```
#include <esnmp.h>
int status

extern SUBTREE ipRouteEntry_subtree;

status = esnmp_unregister( &ipRouteEntry_subtree );

switch (status) {
    case ESNMP_LIB_OK:
        printf ("The esnmp_unregister routine completed successfully.\n");
        break;

    case ESNMP_LIB_BAD_REG:
        printf ("The MIB subtree was not registered.\n");
        break;

    case ESNMP_LIB_LOST_CONNECTION:
        printf("%s%s\n", "The request to unregister the ",
                "MIB subtree could not be sent. ",
                "You should restart the protocol.\n");
        break;
}
```

6.3.1.6 esnmp_register2 ルーチン

esnmp_register2 ルーチンは、esnmp_register ルーチンを拡張したものです。

esnmp_register2 ルーチン呼び出す前に、初期化ルーチン (esnmp_init) を呼び出す必要があります。処理しようとする各 MIB サブツリーに対応するそれぞれのサブツリー構造体に対して、esnmp_register2 関数を呼び出す必要があります。いつでも、esnmp_unregister2 を呼び出すと MIB サブツリーの登録が解除され、その後 esnmp_register2 を呼び出すと再登録されます。

`esnmp_init` を呼び出して eSNMP プロトコルを再起動すると、すべての MIB サブツリーの登録がクリアされます。MIB サブツリーはすべて再登録する必要があります。

MIB サブツリーは、ベース MIB 名と対応する OID によって識別されます。この集合は、MIB サブツリーに含まれるすべての MIB 変数の親を表します。たとえば、MIB-2 `tcp` サブツリーの OID は 1.3.6.1.2.1.6 です。この OID に従属する (同じ 7 つの識別子で始まる) すべての要素は、サブツリーのオブジェクト・テーブルに含まれています。MIB サブツリーは、単一の MIB オブジェクト (リーフ・ノード) でも特定のインスタンスでもかまいません。

MIB サブツリーを登録することにより、サブエージェントは、その MIB サブツリーの領域内部にあるすべての MIB 変数 (または OID) に対する SNMP 要求を処理するということを示します。したがって、サブエージェントは MIB 変数をまだ組み込んでいる MIB サブツリーの中で最も完全に (長く) 修飾されたものを登録します。

`esnmp_register2` ルーチンを使用するサブエージェントは、ローカル・ノードとクラスタに対して、同じ MIB サブツリーを登録できます。MIB サブツリーを両方に登録するには、`esnmp_register2` ルーチンを 2 回呼び出す必要があります。1 回は `options` パラメータで `ESNMP_REG_OPT_CLUSTER` ビットをセットし、もう 1 回は `options` パラメータで `ESNMP_REG_OPT_CLUSTER` ビットをクリアして呼び出します。あるいは、MIB サブツリーをクラスタのみ、またはローカル・ノードのみに対して登録することもできます。この場合は、`options` パラメータで `ESNMP_REG_OPT_CLUSTER` ビットをそれぞれセット/クリアします。

また、サブエージェントは、以前に登録したサブツリーの OID 範囲に重複して、あるいは他のサブエージェントが登録したサブツリーの OID の範囲に重複して MIB サブツリーを登録できます。

たとえば、`os_mibs` と `gated` という 2 つの Tru64 UNIX デーモンを考えてみます。`os_mibs` デーモンは `ip` MIB サブツリー (1.3.6.1.2.1.4) を登録し、`gated` デーモンは `ipRouteEntry` MIB サブツリー (1.3.6.1.2.1.4.21.1) を登録します。これらの登録はどちらも、`options` パラメータで `ESNMP_REG_OPT_CLUSTER` ビットをセットして行われます。

`ipRouteIfIndex` (1.3.6.1.2.1.4.21.1.2) などの `ipRouteEntry` 内の `ip` MIB 変数に対する要求は、`gated` サブエージェントへ送られます。

`ipNetToMediaIfIndex` (1.3.6.1.2.1.4.22.1.1) などの他の `ip` 変数に対する

要求は、os_mibs サブエージェントへ送られます。gated プロセスが終了した場合、またはipRouteEntry MIB サブツリーへの登録を解除した場合、その後の ipRouteIfIndex への要求は、os_mibs サブエージェントへ送られます。これは、すべての ipRouteEntry MIB 変数を含む ip MIB サブツリーが、ipRouteIfIndex に対する要求の最も優先順位が高い領域となっているためです。

esnmp_register2 ルーチンの構文は次のとおりです。

```
int esnmp_register2(  
    ESNMP_REG *reg);
```

引数の定義は次のとおりです。

reg

ESNMP_REG 構造体へのポインタ。次のようなフィールドがあります。

subtree

処理する MIB サブツリーに対応する SUBTREE 構造体へのポインタ。SUBTREE 構造体は、MIB ドキュメントから直接得られた mosy および snmpi ユーティリティで生成されたコードで外部宣言され初期化されます (xxx_tbl.c および xxx_tbl.h , xxx はサブツリー名)。

注意

このフィールドによってポイントされているメモリはすべて、プログラムの存続期間中に libesnmp によって参照されるため、永続的なストレージを持っていない必要があります。snmpi ユーティリティによって発行されたデータ宣言を使用してください。

priority

登録の優先順位。数値が最大のエントリは、優先順位が最も高くなっています。範囲は 1 ~ 255 です。OID (オブジェクト識別子) の範囲内で優先順位が最も高い MIB サブツリーを登録したサブエージェントは、その OID の範囲内の要求をすべて受け取ります。

同じ優先順位で登録されているサブツリーは重複しているとされ、マスタ・エージェントにより登録はリジェクトされます。

priority フィールドは異なる構成を処理するために協力するサブエージェントのためのメカニズムです。

timeout

この MIB サブツリーでデータを要求した際に、マスタ・エージェントが応答を待つ秒数。0 ~ 300 までの値を指定します。0 を指定した場合は、省略時のタイムアウト (3 秒) が使用されます。省略時の値を使用することをお勧めします。

range_subid

整数の値。0 以外の場合、*range_upper_bound* フィールドとともに、MIB サブツリーの OID サブ識別子の 1 つの代わりに範囲を指定します。*range_subid* フィールドは、*range_upper_bound* フィールドによって変更される OID サブ識別子を指定します。

range_upper_bound

整数の値。0 以外の *range_subid* フィールドと組み合わせて、MIB サブツリーの OID サブ識別子の 1 つの代わりに範囲を指定します。MIB サブツリーの OID サブ識別子の範囲に関して、*range_upper_bound* フィールドは上限、*range_subid* は下限になります。

options

整数の値。ESNMP_REG_OPT_CLUSTER に設定されると、登録がクラスタ全体で有効であることを示し、0 に設定されると、登録がローカル・ノードに対して有効であることを示します。

state

次に示す整数の値のいずれかをとり、呼び出す側に対して、この MIB サブツリーの登録の非同期更新を行います。*esnmp_poll* ルーチンが終了した後、呼び出す側はこのパラメータを調べることができます。

ESNMP_REG_STATE_PENDING

登録は現在ローカルに保留されており、マスタ・エージェントとの接続を待っています。

ESNMP_REG_STATE_SENT

登録はマスタ・エージェントに送られました。

ESNMP_REG_STATE_DONE

登録はマスタ・エージェントから正常に肯定応答されました。

ESNMP_REG_STATE_REGDUP

重複しているため、登録はマスタ・エージェントからリジェクトされました。

ESNMP_REG_STATE_REGNOCLU

マスタ・エージェントはクラスタ登録をサポートしていません。

ESNMP_REG_STATE_REJ

マスタ・エージェントは他の理由でリジェクトしました。

instance

ヌルでない場合、この入力パラメータは、登録内の MIB サブツリーに対して部分的にまたは完全に限定されたインスタンスを指定します。テーブル内の row を登録する際は、このパラメータを使用します。テーブル内の row の登録についての詳細は、6.3.1.2 項と `snmpi(8)` を参照してください。

戻り値は次のとおりです。

ESNMP_LIB_OK

`esnmp_register2` ルーチンが正常に終了しました。

ESNMP_LIB_BAD_REG

`esnmp_init` が呼び出されていないか、タイムアウト・パラメータが正しくないか、登録の余地がないか、MIB サブツリーがすでに登録のキューにいれられているかしています。また、ログ・ファイルにもメッセージがあります。

ESNMP_LIB_LOST_CONNECTION

サブエージェントとマスタ・エージェントとの通信が途絶えています。

状態は、要求の開始のみを示しています。マスタ・エージェントの応答に返される実際の状態は、その後の `esnmp_poll` ルーチンへの呼び出しで `reg->state` フィールドに返されます。

次に示すのは `esnmp_register2` ルーチンの例です。

```
#include <esnmp.h>
#define RESPONSE_TIMEOUT      0      /* use the default time set
                                     in esnmp_init message */
#define REGISTRATION_PRIORITY 10     /* priority at which the MIB
                                     subtree will register */
#define RANGE_SUBID           7      /* the identifier position in
                                     oid->elements just after
                                     mib-2 */
#define RANGE_UPPER_BOUND     8      /* the identifier for egp,
                                     under mib-2 */

int status;

extern SUBTREE ip_subtree;
static ESNMP_REG esnmp_reg_for_ip2egp; /* retain this structure
                                     for a subsequent call to
                                     esnmp_unregister2 */

/*
 * initialize the ESNMP_REG structure
 */
memset(&esnmp_reg_for_ip2egp, 0, sizeof(ESNMP_REG));
esnmp_reg_for_ip2egp.subtree      = &ip_subtree;
esnmp_reg_for_ip2egp.priority     = REGISTRATION_PRIORITY;
esnmp_reg_for_ip2egp.timeout      = RESPONSE_TIMEOUT;
esnmp_reg_for_ip2egp.range_subid  = RANGE_SUBID;
esnmp_reg_for_ip2egp.range_upper_bound = RANGE_UPPER_BOUND;

status = esnmp_register2( &esnmp_reg_for_ip2egp );
if (status != ESNMP_LIB_OK) {
    printf("Could not queue the 'ipRouteEntry' \n");
    printf("subtree for registration\n");
}
```

6.3.1.7 esnmp_unregister2 ルーチン

`esnmp_unregister2` ルーチンはマスタ・エージェントから MIB サブツリーの登録を解除します。このルーチンは、MIB サブツリーが

esnmp_register2 ルーチンを用いて登録されている場合にのみ使用してください。

このルーチンをアプリケーションから呼び出して、eSNMP サブエージェントに対して、この MIB サブツリーの変数への要求をそれ以上処理しないように通知することができます。MIB サブツリーは、その後必要に応じて esnmp_register2 ルーチンを呼び出して登録できます。

esnmp_unregister2 ルーチンの構文は次のとおりです。

```
int esnmp_unregister2(  
    ESNMP_REG *reg);
```

引数は次のとおりです。

reg

esnmp_register2 ルーチンが呼び出されたときに使用される、ESNMP_REG 構造体へのポインタ。

戻り値は次のとおりです。

ESNMP_LIB_OK

ルーチンは正常に終了しました。

ESNMP_LIB_BAD_REG

MIB サブツリーが登録されていません。

ESNMP_LIB_LOST_CONNECTION

MIB サブツリーの登録を解除する要求が送信できませんでした。プロトコルを再起動する必要があります。

次に示すのは esnmp_unregister2 ルーチンの例です。

```
#include <esnmp.h>  
int status  
  
extern ESNMP_REG esnmp_reg_for_ip2egp;  
  
status = esnmp_unregister2( &esnmp_reg_for_ip2egp );  
  
switch(status) {  
    case ESNMP_LIB_OK:  
        printf("The esnmp_unregister2 routine completed successfully.\n");  
        break;  
  
    case ESNMP_LIB_BAD_REG:  
        printf("The MIB subtree was not registered.\n");  
        break;
```



```

case ESNMP_LIB_LOST_CONNECTION:
    printf("%s%s\n", "The request to unregister the ",
           "MIB subtree could not be sent. ",
           "You should restart the protocol.\n");
    break;
}

```

6.3.1.8 esnmp_capabilities ルーチン

esnmp_capabilities ルーチンは、サブエージェントの機能をマスタ・エージェントの sysORTable に追加します。sysORTable はエージェントのオブジェクト・リソースを含む概念的なテーブルであり、RFC 1907 で記述されています。

このルーチンは、esnmp_init ルーチンを呼び出して eSNMP を初期化した後の任意の時点で呼び出されます。

esnmp_capabilities ルーチンの構文は次のとおりです。

```

void esnmp_capabilities(
    OID *agent_cap_id,
    char *agent_cap_descr);

```

引数は次のとおりです。

agent_cap_id

最も優先順位が高いエージェント機能識別子を表すオブジェクト識別子へのポインタ。この値は、管理対象ノードの sysORTable で sysORID オブジェクトに対して使用されます。

agent_cap_descr

agent_cap_id を記述するヌル終了文字列へのポインタ。この値は、管理対象ノードの sysORTable で sysORDescr オブジェクトに対して使用されます。

6.3.1.9 esnmp_uncapabilities ルーチン

esnmp_uncapabilities ルーチンは、サブエージェントの機能をマスタ・エージェントの sysORTable から削除します。

このルーチンは、サブエージェントがその機能を動的に変更する場合に呼び出されます。サブエージェントに対する論理接続がクローズされると、マスタ・エージェントは sysORTable 内の関連エントリを削除します。

esnmp_uncapabilities ルーチンの構文は次のとおりです。

```
void esnmp_uncapabilities(  
    OID *agent_cap_id);
```

引数は次のとおりです。

agent_cap_id

sysORTable から削除されるエージェント機能文のオブジェクト識別子へのポインタ。

6.3.1.10 esnmp_poll ルーチン

esnmp_poll ルーチンは、マスタ・エージェントによって送信されているペンディング・メッセージを処理します。このルーチンは、ユーザの select() コールが eSNMP ソケットにデータが用意されていることを示した後に呼び出されます (このソケットは esnmp_init ルーチンに対する呼び出しから返されます)。そのソケットでメッセージがペンディングされていない場合は、esnmp_poll ルーチンはメッセージを受け取るまでブロックします。

受けとったメッセージが問題を指摘している場合、ルーチンは syslog ファイルにエントリを作成し、エラー状態を返します。

受けとったメッセージが SNMP データに対する要求である場合、ルーチンはオブジェクト・テーブルを照会し、適切なメソッド・ルーチンを呼び出します。

esnmp_poll ルーチンの構文は次のとおりです。

```
int esnmp_poll( void );
```

戻り値は次のとおりです。

ESNMP_LIB_OK

esnmp_poll ルーチンが正しく終了しました。

ESNMP_LIB_BAD_REG

マスタ・エージェントによる以前の再登録が失敗しています。ログ・ファイルを参照してください。

ESNMP_LIB_DUPLICATE

重複するサブエージェント ID がマスタ・エージェントによって既に受け取られています。これは、esnmp_init エラーです。

ESNMP_LIB_NO_CONNECTION

マスタ・エージェントは `esnmp_init` 要求の開始に失敗しました。後で再実行してください。また、ログ・ファイルを参照してください。

ESNMP_LIB_CLOSE

CLOSE メッセージが受信されました。

ESNMP_LIB_NOTOK

eSNMP プロトコル・エラーが発生しました。パケットは放棄されました。

ESNMP_LIB_LOST_CONNECTION

マスタ・エージェントとの通信がロストしました。再度接続してください。

6.3.1.11 `esnmp_are_you_there` ルーチン

`esnmp_are_you_there` ルーチンは、ネットワーク・インタフェースが動作しているかどうかをすぐに報告することをマスタ・エージェントに要求します。この呼び出しは応答を待つ間ブロックしません。応答は、`esnmp_poll` ルーチンを呼び出すことによって処理されます。

タイムアウト時間内に応答がない場合、アプリケーションは `esnmp_init` ルーチンを呼び出すことによって eSNMP プロトコルを再起動します。eSNMP ライブラリが管理するタイマはありません。

`esnmp_are_you_there` ルーチンの構文は次のとおりです。

```
int esnmp_are_you_there( void );
```

戻り値は次のとおりです。

ESNMP_LIB_OK

要求が送信されました。

ESNMP_LIB_LOST_CONNECTION

マスタ・エージェントがダウンしているため要求を送信できません。

6.3.1.12 esnmp_trap ルーチン

esnmp_trap ルーチンは、マスタ・エージェントへトラップ・メッセージを送ります。この関数はいつでも呼び出すことができます。マスタ・エージェントが eSNMP プロトコルを実行していない場合、トラップはキュー登録され、通信が確立したときに送られます。

トラップ・メッセージが実際にマスタ・エージェントへ送られるのは、esnmp_init 呼び出しに対するマスタ・エージェントの応答が処理された後です。esnmp_poll ルーチンの後の呼び出し中のほとんどの場合に対して、この処理は API コール内で行われます。esnmp_init, esnmp_poll, および esnmp_trap ルーチンを呼び出すのが、マスタ・エージェントへトラップを送る最も速い方法です。

マスタ・エージェントはトラップを SNMP トラップ・メッセージへフォーマットし、現在の構成のベースになっている管理ステーションへ送ります。マスタ・エージェントの構成については、snmpd(8) および snmpd.conf(4) を参照してください。

マスタ・エージェントから返されるトラップに関する応答はありません。

esnmp_trap ルーチンの構文は次のとおりです。

```
int esnmp_trap(  
    int generic_trap,  
    int specific_trap,  
    char *enterprise,  
    VARBIND *vb);
```

引数は次のとおりです。

generic_trap

汎用トラップ・コード。SNMPv2 トラップに対しては 0 を設定します。

specific_trap

特定トラップ・コード。SNMPv2 トラップに対しては 0 を設定します。

enterprise

ドット表記のエンタープライズ OID 文字列。MIB 仕様を定義する際に NOTIFICATION-TYPE マクロによって定義されるオブジェクト識別子を設定します。この値は、SNMPv2-Trap-PDU 内の SnmpTrapOID.0 の値として受け渡されます。

vb

データの VARBIND リスト (NULL ポインタはデータがないことを示します。)

戻り値は次のとおりです。

ESNMP_LIB_OK

ルーチンが正しく終了しました。

ESNMP_LIB_LOST_CONNECTION

ルーチンは、マスタ・エージェントへトラップ・メッセージを送信できませんでした。

ESNMP_LIB_NOTOK

何らかの原因でメッセージが生成されませんでした。

6.3.1.13 esnmp_term ルーチン

esnmp_term ルーチンは、マスタ・エージェントにクローズ・メッセージを送り、eSNMP プロトコルをシャット・ダウンします。すべてのサブエージェントは、終了時にこのルーチンを呼び出して、マスタ・エージェントが MIB 登録をすぐに更新できるようにして、サブエージェントに代わって eSNMP によって使用されているシステム・リソースを解放できるようにする必要があります。

esnmp_term ルーチンの構文は次のとおりです。

```
void esnmp_term( void );
```

戻り値は次のとおりです。

ESNMP_LIB_OK

esnmp_term ルーチンは、パケットを送信できない場合でも、常に ESNMP_LIB_OK を返します。

6.3.1.14 esnmp_sysuptime ルーチン

esnmp_sysuptime ルーチンは、gettimeofday によって取得した UNIX システム時間を sysUpTime と同じタイム・ベースに変換します。このデータは、1/100 秒の単位で TimeTicks データ・タイプ (マスタ・エージェントが起動されてからの時間) として使用することができます。このタイ

ム・ベースは、`esnmp_init` ルーチンに応答してマスタ・エージェントから取得されます。

このルーチンは UNIX タイムスタンプを SNMP `TimeTicks` に変換するための一般的なメカニズムを提供します。この関数は、特定の UNIX に対する `sysUpTime` の適切な値を返します。ヌルのタイムスタンプを渡すと `sysUpTime` の現在の値が返されます。

構文は次のとおりです。

```
unsigned int esnmp_systime(
    struct timeval *timestamp);
```

引数は次のとおりです。

`timestamp`

`gettimeofday` システム・コールから取得した値を含んでいる `struct timeval` へのポインタです。構造体は `include/sys/time.h` に定義されています。

NULL ポインタは現在の `sysUpTime` を返すことを意味します。

`esnmp_systime` ルーチンの例を次に示します。

```
#include <sys/time.h>
#include <esnmp.h>
struct timeval timestamp;

gettimeofday(&timestamp, NULL);
o_integer(vb, object, esnmp_systime(&timestamp));
```

戻り値は次のとおりです。

0 エラー (`gettimeofday` の失敗) を示します。それ以外の場合は、`timestamp` に、マスタ・エージェント・プロトコルが開始されてからの時間が 1/100 秒の単位で含まれます。

6.3.2 メソッド・ルーチン呼び出しインタフェース

SNMP 要求には、多数の `VariableBindings` (コード化された MIB 変数) が含まれていることがあります。サブエージェントで実行されている `libsnmp` コードは、それぞれの `VariableBinding` をオブジェクト・テーブル項目と照合します。そのとき、オブジェクト・テーブルのメソッド・ルーチンが呼ばれます。

このため、メソッド・ルーチンは単一の MIB 変数をサービスするため呼び出され、単一の SNMP 要求の中で同じメソッド・ルーチンが複数回呼び出されることもあります。

メソッド・ルーチン呼び出しインタフェースには次の関数が含まれます。

- `*_get`
- `*_set`

メソッド・ルーチンについての詳細は、6.3.2.3 項を参照してください。

6.3.2.1 `*_get` ルーチン

`*_get` ルーチンは、MIB グループ (たとえば MIB-2 における `system`) や テーブル・エントリ (たとえば MIB-2 における `ifEntry`) などの、指定した MIB 項目に対するメソッド・ルーチンです。しかし、それはどうとでもできます。詳細については、`snmpi(8)` を参照してください。

`libesnmp` ルーチンは、登録されているサブツリーによって認識されるオブジェクト・テーブルの `Get` 操作に対して指定されているルーチン呼び出します。

この関数は、サブエージェント・オブジェクト・テーブルのいくつかの要素によってポイントされます。要求がオブジェクトに届くと、そのメソッド・ルーチンが呼び出されます。`*_get` メソッド・ルーチンが `Get` SNMP 要求に対する応答の際に呼び出されます。

`*_get` ルーチンの構文は次のとおりです。

```
int mib-group_get(  
    METHOD *method);
```

引数は次のとおりです。

method

次のフィールドを含む `METHOD` へのポインタです。

action

`ESNMP_ACT_GET`、`ESNMP_ACT_GETNEXT`、あるいは
`ESNMP_ACT_GETBULK` のいずれか。

serial_num

この SNMP 要求に対してユニークな整数値。単一の SNMP 要求をサービスしている間に呼び出されるメソッド・ルーチンは、同

じ値 *serial_num* を受け取ります。新しい SNMP 要求は、新しい値 *serial_num* によって示されます。

repeat_cnt

GetBulk に対してのみ使用します。この値は、反復している VARBIND の現在の反復値を示します。この値は、1 ~ *max_repetitions* の間で増加します。0 の値は、VARBIND 構造体を反復しないことを示します。

max_repetitions

GetBulk に使用します。反復数の最大値です。VARBIND 構造体を反復しない場合は 0 になります。呼び出しの繰り返し数の最大値を知ることによって、その後の処理を最適化することができます。

varbind

OID およびデータ・フィールドを埋めなければならない VARBIND 構造体に対するポインタ。メソッド・ルーチンのエントリ時、*method->varbind->name* フィールドは要求された OID です。

メソッド・ルーチンの終了時、要求されたデータが *method->varbind* フィールドに含まれ、返された VARBIND に対する実際のインスタンス OID を反映して *method->varbind->name* フィールドが変更されます。

libsnmp ルーチン (*o_integer*, *o_string*, *o_oid*, および *o_octet*) は一般にデータをロードするために使用されます。*libsnmp instance2oid* ルーチンは、*method->varbind->name* フィールドで OID を更新するために使用されます。

object

MIB 変数が参照されているオブジェクト・テーブル・エントリへのポインタ。*method->object->object_index* フィールドは、(1 つのメソッド・ルーチンが多くのオブジェクトをサービスする際に便利な) オブジェクト・テーブル内でのこのオブジェクトのユニークなインデックスです。

method->object->oid フィールドは、MIB においてこのオブジェクトに対して定義される OID です。要求されたインスタンスは、この OID と method->varbind->name フィールドで発見された要求の OID とを比較することによって引き出されます。この操作には oid2instance が便利です。

*_get ルーチンに対する可能な戻り値は次のとおりです。

ESNMP_MTHD_noError

ルーチンが正常終了しました。

ESNMP_MTHD_noSuchObject

要求したオブジェクトが返されないか、あるいは存在していません。

ESNMP_MTHD_noSuchInstance

オブジェクトの要求したインスタンスが返されないか、あるいは存在していません。

ESNMP_MTHD_genErr

一般的な処理エラー。

6.3.2.2 *_set メソッド・ルーチン

*_set メソッド・ルーチンは、MIB グループ (たとえば MIB-2 の system) あるいはテーブル・エントリ (たとえば MIB-2 の ifEntry) などの、指定された MIB 項目のためのものです。しかし、どうしてもなります。詳細については、snmp(8) を参照してください。

libesnmp ルーチンは、登録されているサブツリーによって認識されるオブジェクト・テーブルの Set 操作に対して指定されているルーチンを呼び出します。

この関数は、サブエージェント・オブジェクト・テーブルのいくつかの要素によってポイントされます。オブジェクトに対する要求が到着すると、そのメソッド・ルーチンが呼び出されます。*_set メソッド・ルーチンは Set SNMP 要求に応答して呼び出されます。

*_set メソッド・ルーチンの構文は次のとおりです。

```
int mib-group_set(  
    METHOD *method);
```

引数は次のとおりです。

method

次のフィールドを含む METHOD 構造体へのポインタです。

action

ESNMP_ACT_SET , ESNMP_ACT_COMMIT ,
ESNMP_ACT_UNDO , あるいは ESNMP_ACT_CLEANUP のい
ずれか。

serial_num

この SNMP 要求に対してユニークな整数値。単一の SNMP 要求
をサービスしている間に呼び出されるメソッド・ルーチンは、同
じ値 *serial_num* を受け取ります。新しい SNMP 要求は、新し
い値 *serial_num* によって示されます。

varbind

MIB 変数の提供するデータおよび名前 (OID) を含む VARBIND 構
造体に対するポインタ。インスタンス情報は OID から展開され、
method->row->instance フィールドに置かれます。

object

MIB 変数が参照されているオブジェクト・テーブル・エントリ
へのポインタ。 *method->object->object_index* フィールド
は、(1 つのメソッド・ルーチンが多くのオブジェクトをサービ
スする際に便利な) オブジェクト・テーブル内でのこのオブジェ
クトのユニークなインデックスです。 *method->object->oid*
フィールドは、MIB においてこのオブジェクトに対して定義さ
れる OID です。

flags

libesnmplib によって設定される読み取り専用の整数ビットマスク
です。 ESNMP_FIRST_IN_ROW ビットを設定すると、この呼
び出しが *row* に設定される最初のオブジェクトであることを示
します。 ESNMP_LAST_IN_ROW ビットを設定すると、この
呼び出しが *row* に設定される最後のオブジェクトであることを
示します。 ESNMP_LAST_IN_ROW ビットが設定されている

METHOD 構造体だけが、commit、undo、cleanup のフェーズのメソッド・ルーチンに渡されます。

row

esnmp.h ヘッダ・ファイルで定義されている ROW_CONTEXT 構造体へのポインタです。同じグループを参照し同じインスタンス番号を持つメソッド・ルーチンに対するすべての Set 呼び出しは、同じ row 構造体に存在します。メソッド・ルーチンは、Set 呼び出し実行中に、commit および undo フェーズで使用する情報を row 構造体に蓄積することができます。累積されたデータは、cleanup フェーズでメソッド・ルーチンを使用して解放することができます。ROW_CONTEXT 構造体には次のフィールドがあります。

instance

概念的 row に対するインスタンス OID を含む配列のアドレスです。libesnmp ルーチンは、要求された変数 binding oid から object oid を引くことによってこの配列を作成します。

instance_len

method->row->instance フィールドのサイズです。

context

要求の処理に必要なデータを参照するために、メソッド・ルーチンが使用するプライベートなポインタです。

save

要求の undo に必要となるデータを参照するために、メソッド・ルーチンがプライベートに使用するポインタです。

state

状態情報を保持するためにメソッド・ルーチンがプライベートに使用する整数です。

*_set メソッド・ルーチンの可能な戻り値は次のとおりです。

ESNMP_MTHD_noError

ルーチンが正常終了しました。

ESNMP_MTHD_notWritable

要求したオブジェクトが設定できないか、あるいは実現されていません。

ESNMP_MTHD_wrongType

要求された値のデータ型が正しくありません。

ESNMP_MTHD_wrongLength

要求した値の長さが正しくありません。

ESNMP_MTHD_wrongEncoding

要求した値の表現が正しくありません。

ESNMP_MTHD_wrongValue

要求した値が範囲外です。

ESNMP_MTHD_noCreation

要求したインスタンスは常に作成できません。

ESNMP_MTHD_inconsistentName

要求したインスタンスは現在作成できません。

ESNMP_MTHD_inconsistentValue

要求した値に矛盾があります。

ESNMP_MTHD_resourceUnavailable

リソースの制約によって失敗しました。

ESNMP_MTHD_genErr

一般的な処理エラーが発生しました。

ESNMP_MTHD_commitFailed

commit フェーズが失敗しました。

ESNMP_MTHD_undoFailed

undo フェーズが失敗しました。

*_set ルーチンの処理

各変数割り当ては解析され、そのオブジェクトはオブジェクト・テーブルに置かれます。各 VARBIND 構造体に対して METHOD 構造体を作成されます。これらの METHOD 構造体は、各フェーズの処理に便利な ROW_CONTEXT 構造体をポイントします。同じ概念的 row のオブジェクトは、すべて同じ ROW_CONTEXT 構造体をポイントします。決定は、次の項目をチェックすることによって行われます。

- 参照されるオブジェクトが同じ MIB グループに存在する。
- VARBIND 構造体が同じインスタンス OID を持つ。

各 ROW_CONTEXT 構造体は、概念的 row に対するインスタンス情報でロードされます。ROW_CONTEXT 構造体の *context* および *save* フィールドは NULL に設定され、*state* フィールドは ESNMP_SET_UNKNOWN に設定されます。

各オブジェクトに対するメソッド・ルーチンが呼び出され、アクション・コードに ESNMP_ACT_SET を持つ METHOD 構造体が渡されます。

すべてのメソッド・ルーチンが正常終了を返す場合、各 row に対して単一のメソッド・ルーチン (最後に呼び出されたメソッド・ルーチン) が `method->action == ESNMP_ACT_COMMIT` で呼び出されます。

いずれかの row が失敗を報告すると、正常にコミットしたすべての row に対してフェーズを行わないよう伝えます。この処理は、`method->action == ESNMP_ACT_UNDO` で、各 row に対して単一のメソッド・ルーチン (commit フェーズで呼び出されたメソッド・ルーチン) を呼び出すことによって行われます。

最後に各 row が解放されます。各 row に対する単一のメソッド・ルーチンは、`method->action == ESNMP_ACT_CLEANUP` で呼び出されます。この処理は、前の処理の結果にかかわらず、各 row ごとに行われます。

次のリストでアクション・コードを説明しています。

ESNMP_ACT_SET

各オブジェクトのメソッド・ルーチンは、Set フェーズですべてのオブジェクトが処理されるかメソッド・ルーチンがエラー状態値を返すまでの間に呼び出されます (これは、各オブジェクトのメソッド・ルーチンが呼び出され

る唯一のフェーズです)。同じ概念 `row` における変数割り当てのために、`method->row` は共通の `ROW_CONTEXT` をポイントします。

呼び出されてるのが `ROW_CONTEXT` に対する最後のオブジェクトである場合、`method->flags` ビットマスクは `ESNMP_LAST_IN_ROW` ビットを設定します。これによって、概念的 `row` に対する各変数割り当てを見ることができるよう、最後の整合性チェックが可能になります。

このフェーズにおけるメソッド・ルーチンのジョブは、`SetRequest` が動作するかどうか判断し、そうでない場合は正しい SNMP エラー・コードを返し、`commit` フェーズで `Set` を実際に実行するのに必要なコンテキスト・データを用意することです。

`method->row->context` はメソッド・ルーチンに対してプライベートで、`libesnmp` はこれを使用しません。典型的な使用法は、概念的 `row` に対する `VARBIND` からのデータでロードされた、エミットされた `foo_type` 構造体のアドレスを保管することです。

ESNMP_ACT_COMMIT

いくつかの変数割り当てが概念的 `row` に存在していたとしても、(`SetRequest` の順序で) 最後の変数割り当てのみが処理されます。そのため、共通の `row` をポイントするすべてのメソッド・ルーチンのうち最後のルーチンのみが呼び出されます。

このメソッド・ルーチンは、操作を実行するのに必要なすべてのデータおよびコンテキストが利用できなければなりません。また、現在のデータのスナップショットあるいは `Set` を `undo` するために必要なデータを保管する必要があります。`method->row->save` フィールドは、この処理を実行するのに必要なデータへのポインタを保持します。典型的な使用法は、概念的 `row` の現在のデータでロードされた、`xxx` 構造体のアドレスを保管することです。`xxx` 構造体は、`snmpi` プログラムが自動的に生成したものです。

`method->row->save` フィールドもメソッド・ルーチンに対してプライベートで、`libesnmp` はこれを使用しません。

`set` 操作が成功した場合、`ESNMP_MTHD_noError` を返します。失敗した場合はコミットを取消し、`ESNMP_MTHD_commitFailed` の値を返します。

`commit` フェーズでエラーが返された場合、`libesnmp` は `undo` フェーズに入ります。それ以外の場合は、`cleanup` フェーズに入ります。

注意

現在のサブエージェントで Set 操作が成功しても、SetRequest が他のサブエージェントに及び、他のサブエージェントが失敗するため、undo フェーズが発生する場合があります。

ESNMP_ACT_UNDO

正しくコミットした各概念的 row に対しては、`method->action == ESNMP_ACT_UNDO` で同じメソッド・ルーチンが呼び出されます。commit フェーズでまだ呼び出されていない `ROW_CONTEXT` 構造体は、undo フェーズで呼び出されることなく、cleanup フェーズで呼び出されます。

メソッド・ルーチンは、commit フェーズで実行されたときの条件をリストアすべきです (この処理は、`method->row->save` フィールドによってポイントされるデータを使用して行われます)。

成功した場合は `ESNMP_MTHD_noError` を返し、失敗した場合は `ESNMP_MTHD_undoFail` を返します。

ESNMP_ACT_CLEANUP

何が発生したかに関係なく、この時点で各 `ROW_CONTEXT` は cleanup フェーズに入ります。commit フェーズで呼び出されたのと同じメソッド・ルーチンは、`method->action == ESNMP_ACT_CLEANUP` で呼び出されます。

これは、SetRequest の処理の終わりを示します。メソッド・ルーチンは、`method->row->context` および `method->row->save` フィールドで割り当てられ保管されていたダイナミック・メモリの解放など、必要な処理を行います。

cleanup フェーズの関数の戻り状態値は無視されます。

6.3.2.3 メソッド・ルーチンのアプリケーションでのプログラミング

`subtree_tbl.h` ファイルで宣言されているメソッド・ルーチンのコードを作成する必要があります。各メソッド・ルーチンは、次のように `METHOD` 構造体へのポインタを引数として持ちます。

```
int mib_group_get(
    METHOD *method,
int mib_group_set(
    METHOD *method);
```

Get メソッド・ルーチンは、Get、GetNext および GetBulk を実行するのに使用されます。

Get メソッド・ルーチンは次のタスクを行います。

1. 必要な OID のインスタンス部分を抽出します。この処理は `oid2instance libesnmp` ルーチンで行います。
`method->object->oid` フィールド (オブジェクトのベース OID) と `method->varbind->name` フィールド (要求された OID) とを比較することによって手作業でこの処理を行うこともできます。
2. インスタンスの妥当性を判断します。インスタンス OID は、ヌル、あるいはいずれかの長さです (何が要求され、オブジェクトがどのように選択されたかに依存します)。インスタンス OID のチェックによって、要求をすぐにリジェクトすることができます。
3. データを抽出します。インスタンス OID と `method->action` フィールドをもとに、どのようなデータを返すべきかを判断します。
4. 応答 OID をそのメソッドの `VARBIND` 構造体にロードします。返されている実際の MIB 変数インスタンスの OID を `method->varbind` フィールドに設定します。この処理は、返したいインスタンス OID を整数の配列にロードし `instance2OID libesnmp` ルーチンを呼び出すことによって行われます。
5. 応答データをメソッドの `VARBIND` 構造体にロードします。

次に示す、データ・タイプに対応する `libesnmp` ライブラリを使用して、返すデータを `method->varbind` フィールドにロードします。

- `o_integer`
- `o_string`
- `o_octet`
- `o_oid`

これらのルーチンは、指定したデータのコピーを作成します。`libesnmp` 関数はコピーされたデータに関するメモリの管理を行います。メソッド・ルーチンはオリジナル・データのメモリ管理を行う必要があります。

ルーチンは、MIB 変数に対するオブジェクト・テーブルで定義されているタイプへの必要な変換を行い、変換したデータを `method->varbind` フィールドへコピーします。

データ値の表現については、値の表現の項を参照してください。

6. 次のように正しい状態値を返します。

- ESNMP_MTHD_noError — ルーチンが正常に終了したか、あるいはエラーが検出されませんでした。
- ESNMP_MTHD_noSuchInstance
— 要求されたオブジェクトにはそのようなインスタンスがありません。
- ESNMP_MTHD_noSuchObject
— そのようなオブジェクトは存在しません。
- ESNMP_MTHD_genErr — エラーが発生し、ルーチンが正常に終了しませんでした。

値の表現

各データ型に対する VARBIND 構造体の値は次のように表現されます。(OCT および OID 構造体については esnmp.h ファイルを参照してください。)

- ESNMP_TYPE_Integer32 (*varbind->value.sl* フィールド)
32 ビットの符号付き整数。整数値を VARBIND に挿入するには `o_integer` ルーチンを使用します。この値の引数のプロトタイプは符号なしロングなので、場合によってはこの値を `signed int` にキャストする必要があります。
- ESNMP_TYPE_DisplayString, ESNMP_TYPE_Opaque, ESNMP_TYPE_OctetString (*varbind->value.oct* フィールド)
オクテット文字の文字列。この値は、長さおよび動的に割り当てられた文字配列に対するポインタを含む OCT 構造体として、VARBIND 構造体に含まれます。
DisplayString は、文字配列が ASCII テキストとして解釈される点で OctetString と異なります。OctetString にビットもしくはビット列がある場合は、OCT 構造体には次のものが含まれます。
 - その値を含むのに必要なバイト数に等しい長さ (すなわち、 $((\text{qty-bits} - 1)/8 + 1)$)。
 - *bbbbbb..bb* の形式の *bitstring* のビットを含むバッファへのポインタ。ここで、*bb* オクテットはビット列自身を表し、ビット

0 が先頭です。最後のオクテットの使用していないビットは、0 に設定されます。

VARBIND 構造体 (バッファおよび長さ) に値を挿入するには `o_string` ルーチンを使用します。新しいスペースが割り当てられ、バッファがその新しいスペースへコピーされます。

値を VARBIND 構造体 (OCT 構造体へのポインタ) に挿入するには `o_octet` ルーチンを使用します。新しいスペースが割り当てられ、OCT 構造体によってポイントされるバッファがコピーされます。

- **ESNMP_TYPE_ObjectId** (`varbind->value.oid` および `varbind->name` フィールド)

オブジェクト識別子。要素の数と、動的に割り当てられた符号なし整数の配列へのポインタを含む OID 構造体として、VARBIND 構造体に含まれます。

`varbind->name` フィールドは、オブジェクト識別子と MIB 変数を識別するインスタンス情報を保持するのに使用されます。要求の OID からインスタンス要素を取り出すには `OID2Instance` 関数を使用します。応答を作成する際に VARBIND 構造体の `name` フィールドを設定するために、インスタンス要素と MIB 変数のベース OID を組み合わせるには、`instance2oid` 関数を使用します。

データとして返される OID 値が OID 構造体へのポインタとなっている場合に、オブジェクト識別子を VARBIND 構造体に挿入するには、`o_oid` 関数を使用します。

データとして返される OID 値が、たとえば 1.3.6.1.2.1.3.1.1.2.0 のようにドット・フォーマットの OID を含む ASCII 文字へのポインタとなっている場合、オブジェクト ID を VARBIND 構造体に挿入するには `o_string` 関数を使用します。

- **ESNMP_TYPE_NULL**

NULL あるいは空のタイプ。この値は、値がないことを示すために使用されます。長さは 0 で、VARBIND 構造体における共用体値は 0 です。

`Get`、`GetNext`、および `GetBulk` での VARBIND は、このデータ・タイプを持ちます。メソッド・ルーチンは、このような値は返しません。`Set` 要求は、VARBIND 構造体でこのような値は持ちません。

- **ESNMP_TYPE_IpAddress** (`varbind->value.oct` フィールド)

IP アドレス。この値は、長さ 4 およびネットワーク・バイト順で 4 バイトの IP アドレスを含む動的に割り当てられるバッファへのポインタを持つ OCT 構造体として、VARBIND 構造体に含まれます。

値がネットワーク順の符号なし整数の場合に IP アドレスを VARBIND 構造体に挿入するには、`o_integer` 関数を使用します。

値が(ネットワーク・バイト順の) バイト配列の場合に IP アドレスを VARBIND 構造体に挿入するには、`o_string` 関数を使用します。長さは 4 を使用します。

- **ESNMP_TYPE_UInteger32 ESNMP_TYPE_Counter32 ESNMP_TYPE_Gauge32** (*varbind->value.ul* フィールド)
32 ビット・カウンタおよび 32 ビット・ゲージ・データ・タイプが、`unsigned int` として VARBIND 構造体に保管されます。
VARBIND 構造体に符号なしの値を挿入するには、`o_integer` 関数を使用します。
- **ESNMP_TYPE_TimeTicks** (*varbind->value.ul* フィールド)
32 ビット・タイムチケット・タイプの値が、`unsigned int` として VARBIND 構造体に保管されます。
符号なしの値を VARBIND 構造体に挿入するには、`o_integer` 関数を使用してください。
- **ESNMP_TYPE_Counter64** (*varbind->value.ul64*)
64 ビット・カウンタは、64 ビット値を使用する Alpha マシンで、`unsigned long` として VARBIND 構造体に保管されます。
符号なしロング値 (64 ビット) を VARBIND 構造体に挿入するには、`o_integer` 関数を使用してください。

6.3.3 libsnmp サポート・ルーチン

以降の節では、libsnmp サポート・ルーチンについて説明します。libsnmp サポート・ルーチンには次のルーチンがあります。

- `o_integer`
- `o_octet`
- `o_oid`
- `o_string`

- str2oid
- sprintoid
- instance2oid
- oid2instance
- inst2ip
- cmp_oid
- cmp_oid_prefix
- clone_oid
- free_oid
- clone_buf
- mem2oct
- cmp_oct
- clone_oct
- free_oct
- free_varbind_data
- set_debug_level
- is_debug_level
- ESNMP_LOG

6.3.3.1 o_integer ルーチン

`o_integer` ルーチンは、整数値を適切なデータ型で `VARBIND` 構造体にロードします。

構文は次のとおりです。

```
int o_integer(
    VARBIND *vb,
    OBJECT *obj,
    unsigned long value);
```

引数は次のとおりです。

vb

データを受け取る `VARBIND` 構造体へのポインタです。この関数は `VARBIND` 構造体は割り当てません。

obj

VARBIND 構造体において OID と関連する MIB 変数に対する OBJECT 構造体へのポインタです。

value

VARBIND 構造体に挿入される値です。

オブジェクト構造体で定義されているように、実数型は次のいずれかでなければなりません。これ以外の場合は、エラーが返されます。

実数型が `IpAddress` の場合、ネットワーク・バイト順で 4 バイト整数が仮定されます。

```
ESNMP_TYPE_Integer32:
    32 ビット INTEGER
ESNMP_TYPE_Counter32:
    32 ビット Counter (符号なし)
ESNMP_TYPE_Gauge32:
    32 ビット Gauge (符号なし)
ESNMP_TYPE_TimeTicks:
    32 ビット TimeTicks (符号なし)
ESNMP_TYPE_UInteger32:
    32 ビット INTEGER (符号なし)
ESNMP_TYPE_Counter64:
    64 ビット Counter (符号なし)
ESNMP_TYPE_IpAddress:
    IMPLICIT OCTET STRING (4)
```

`o_integer` ルーチンの例を次に示します。

```
#include <esnmp.h>
#include "ip_tbl.h"  <-- for ipNetToMediaEntry_type definition
VARBIND      *vb      = method->varbind;
OBJECT       *object   = method->object;
ipNetToMediaEntry_type *data;
:
: assume buffer and structure member assignments occur here
:
switch(arg) {
case I_atIfIndex:
return o_integer(vb, object, data->ipNetToMediaIfIndex);
```

戻り値は次のとおりです。

`ESNMP_MTHD_noError`

ルーチンが正常に終了しました。

ESNMP_MTHD_genErr

エラーが発生しました。

6.3.3.2 o_octet ルーチン

o_octet ルーチンはオクテット値を適切なデータ型で VARBIND 構造体へロードします。

構文は次のとおりです。

```
int o_octet(  
    VARBIND *vb,  
    OBJECT *obj,  
    OCT *oct);
```

引数は次のとおりです。

vb

データを受け取る VARBIND 構造体へのポインタです。この関数は VARBIND 構造体を割り当てません。

注意

vb フィールドのものの値が NULL でない場合、このルーチンはそれを解放しようとします。このため、自身の *vb* 構造体を割り当てるために malloc コマンドを使用する場合は、使用する前にゼロ詰めします。

obj

VARBIND 構造体の OID に関する MIB 変数に対する OBJECT 構造体へのポインタです。

value

VARBIND 構造体に挿入される値です。

オブジェクト構造体で定義されているように、実数型は次のいずれかでなければなりません。これ以外の場合は、エラーが返されます。

ESNMP_TYPE_OCTET_STRING
OCTET STRING
ESNMP_TYPE_IpAddress
IMPLICIT OCTET STRING (4) — オクテット型、
ネットワーク・バイト順

ESNMP_TYPE_DisplayString
DisplayString (Textual Convention)
ESNMP_TYPE_Opaque
IMPLICIT OCTET STRING

o_octet ルーチンの例を次に示します。

```
#include <esnmp.h>
#include "ip_tbl.h"  <-- for ipNetToMediaEntry_type definition
VARBIND      *vb      = method->varbind;
OBJECT       *object   = method->object;
ipNetToMediaEntry_type *data;
:
: assume buffer and structure member assignments occur here
:
switch(arg) {
case I_atPhysAddress:
return o_octet(vb, object, &data->ipNetToMediaPhysAddress);
```

戻り値は次のとおりです。

ESNMP_MTHD_noError

ルーチンが正常に終了しました。

ESNMP_MTHD_genErr

エラー状態が発生しました。

6.3.3.3 o_oid ルーチン

o_oid ルーチンは、OID 値を適切なデータ型で VARBIND 構造体にロードします。

構文は次のとおりです。

```
int o_oid(
    VARBIND *vb,
    OBJECT *obj,
    OID *oid);
```

引数は次のとおりです。

vb

データを受け取る VARBIND 構造体へのポインタです。この関数は VARBIND 構造体の割り当ては行いません。

注意

`vb` フィールドのもとの値が `NULL` でない場合、このルーチンはそれを解放しようとします。このため、自身の `vb` 構造体を割り当てるために `malloc` コマンドを使用する場合は、使用する前にゼロ詰めします。

obj

`VARBIND` 構造体の `OID` と関係する `MIB` 変数に対する `OBJECT` 構造体へのポインタです。

value

`VARBIND` 構造体にデータとして挿入される値です。`OID` の長さについて、6.2.1 項を参照してください。

オブジェクト構造体で定義されているように、実数型は次のいずれかでなければなりません。これ以外の場合は、エラーが返されます。

`ESNMP_TYPE_OBJECT_IDENTIFIER`
`OBJECT IDENTIFIER`

`o_oid` ルーチンの例を次に示します。

```
#include <esnmp.h>
#include "ip_tbl.h"  <-- for ipNetToMediaEntry_type definition
VARBIND      *vb      = method->varbind;
OBJECT       *object   = method->object;
ipNetToMediaEntry_type *data;
:
: assume buffer and structure member assignments occur here
:
switch(arg) {
    case I_atObjectID:
        return o_oid(vb, object, &data->ipNetToMediaObjectID);
}
```

戻り値は次のとおりです。

`ESNMP_MTHD_noError`

ルーチンが正常に完了しました。

`ESNMP_MTHD_genErr`

エラー状態が発生しました。

6.3.3.4 o_string ルーチン

o_string ルーチンは、文字列値を適切なデータ型で VARBIND 構造体へロードします。

構文は次のとおりです。

```
int o_string(  
    VARBIND *vb,  
    OBJECT *obj,  
    unsigned char *ptr,  
    int len);
```

引数は次のとおりです。

vb

データを受け取る VARBIND 構造体へのポインタです。この関数は VARBIND 構造体を割り当てません。

注意

vb フィールドのものの値が NULL でない場合、このルーチンはそれを解放しようとします。このため、自身の *vb* 構造体を割り当てるために malloc コマンドを使用する場合は、使用する前にゼロ詰めします。

obj

VARBIND 構造体の oid と関係する MIB 変数に対する OBJECT 構造体へのポインタです。

ptr

VARBIND 構造体へ挿入するデータを含んでいるバッファへのポインタです。

len

ptr フィールドがポイントするバッファ内のデータの長さです。

オブジェクト構造体で定義されているように、実数型は次のいずれかでなければなりません。これ以外の場合は、エラーが返されます。

ESNMP_TYPE_OCTET_STRING
OCTET STRING

```

ESNMP_TYPE_IpAddress
    IMPLICIT OCTET STRING (4) — オクテット型，ネット
    ワーク・バイト順
ESNMP_TYPE_DisplayString
    DisplayString (Textual Convention)
ESNMP_TYPE_Opaque
    IMPLICIT OCTET STRING
ESNMP_TYPE_OBJECT_IDENTIFIER
    OBJECT IDENTIFIER — ドット表記では 1.3.6.1.4.1.3.6

```

o_string ルーチンの例を次に示します。

```

#include <esnmp.h>
#include "ip_tbl.h"  <-- for ipNetToMediaEntry_type definition
VARBIND      *vb      = method->varbind;
OBJECT       *object   = method->object;
ipNetToMediaEntry_type *data;
:
: assume buffer and structure member assignments occur here
:
switch(arg) {
case I_atPhysAddress:
    return o_string(vb, object, data->ipNetToMediaPhysAddress.ptr,
                    data->ipNetToMediaPhysAddress.len);
}

```

戻り値は次のとおりです。

ESNMP_MTHD_noError

ルーチンが正常に終了しました。

ESNMP_MTHD_genErr

エラー状態が発生しました。

6.3.3.5 str2oid ルーチン

str2oid ルーチンは、ヌルで終了する OID 文字列 (ドット表記) を OID 構造体へ変換します。

このルーチンは要素バッファを動的に割り当て、そのポインタを OID 構造体に挿入します。このバッファは呼び出し元の責任で解放されます。OID は最大 128 の要素を持つことができます。空文字列もしくは空き文字列は、0 の要素をひとつ持つ OID 構造体を返します。

str2oid ルーチンは、OID 構造体を割り当てません。

構文は次のとおりです。

```
OID * str2oid(
    OID *oid,
    char *s);
```

str2oid ルーチンの例を次に示します。

```
#include <esnmp.h>
OID abc;
if (str2oid(&abc, "1.2.5.4.3.6") == NULL)
    DPRINTF((WARNING, "It did not work...\n"));
```

戻り値は次のとおりです。

NULL	エラーが発生しました。そうでない場合は、OID へのポインタ (最初の引数) が返されます。
------	--

6.3.3.6 sprintoid ルーチン

sprintoid ルーチンは OID をヌルで終わるドット表記の文字列に変換します。OID 構造体は要素を 128 まで持つことができます。フル・サイズの OID 構造体は大きなバッファを必要とします。

構文は次のとおりです。

```
char * sprintoid(
    char *buffer,
    OID *oid);
```

sprintoid ルーチンの例を次に示します。

```
#include <esnmp.h>
#define SOMETHING_BIG 1024
OID abc;
char buffer[SOMETHING_BIG];
:
: assume abc gets initialized with some value
:
printf("dots=%s\n", sprintoid(buffer, &abc));
```

戻り値は最初の引数をポイントします。

6.3.3.7 instance2oid ルーチン

instance2oid ルーチンは、オブジェクトのベース OID のコピーを作成しインスタンス配列のコピーを追加して、値に対する完全な OID を作成します。instance は整数の配列で、len は要素数です。インスタンス配列は oid2instance によって作成されるか、あるいは get_next 探索の結果としてキー値から作成されます。

このルーチンは要素バッファを動的に割り当て、呼び出しで渡される `OID` 構造体にそのポインタを挿入します。このバッファの解放は呼び出し元のプログラムもしくはモジュールの責任で行われます。

このルーチンを使うには、新しい `OID` 値を受け取る `OID` 構造体をポインタして、このルーチンを呼び出してください。 `OID` 構造体の以前の値は `free_oid` を最初に呼び出すときに解放され、新しい値が動的に割り当てられ挿入されます。新しい `OID` 構造体の初期値はすべて 0 です。

`instance2oid` ルーチンは `OID` 構造体を割り当てるのではなく、要素を含む配列のみを割り当てることに注意してください。

構文は次のとおりです。

```
OID * instance2oid(  
    OID *new,  
    OBJECT *obj,  
    unsigned int *instance,  
    int len);
```

引数は次のとおりです。

new

新しい `OID` 値を受け取る `OID` 構造体のポインタです。

obj

入手されている MIB 変数に対するオブジェクト・テーブル・エントリのポインタです。新しい `OID` の最初の部分は、MIB オブジェクト・テーブル・エントリから得られた `OID` です。

instance

instance 値の配列のポインタです。これらの値は、MIB オブジェクト・テーブル・エントリから得られたベース `OID` に追加され、新しい `OID` を構成します。

len

instance 配列の要素数です。

`instance2oid` ルーチンの例を次に示します。

```
#include <esnmp.h>  
VARBIND *vb;      <-- filled in  
OBJECT *object;   <-- filled in  
unsigned int instance[6];  
  
-- Construct the outgoing OID in a GETNEXT      --
```

```
-- Instance is N.1.A.A.A where A's are IP address --
instance[0] = data->ipNetToMediaIfIndex;
instance[1] = 1;
for (i = 0; i < 4; i++) {
    instance[i+2]=((unsigned char *)(&data->ipNetToMediaNetAddress))[i];
}
instance2oid(&vb->name, object, instance, 6);
```

戻り値は次のとおりです。

NULL

エラーが発生しました。それ以外の場合は、OID 構造体のポインタ (最初の引数) が返されます。

6.3.3.8 oid2instance ルーチン

oid2instance ルーチンは、OID 構造体からインスタンス値を取り出し、それらを指定した整数配列にコピーします。その後、配列内の要素数を返します。インスタンスは、MIB 変数を識別する要素以外の OID の要素です。それらは、MIB 値の特定のインスタンスを識別するためのインデックスとして使用されます。

OID 構造体が予想より多くの要素を含んでいる場合は (max_len パラメータで指定した値より大きい場合)、この関数は max_len で指定した要素数だけをコピーし、コピーした要素の総数を返します。

構文は次のとおりです。

```
int oid2instance(
    OID *oid,
    OBJECT *obj,
    unsigned int *instance,
    int max_len);
```

引数は次のとおりです。

<i>oid</i>	インスタンスあるいはインスタンスの一部が含まれている OID 構造体へのポインタ。
<i>obj</i>	MIB 変数に対するオブジェクト・テーブル・エントリへのポインタ。
<i>instance</i>	インデックスが置かれる符号なし整数配列へのポインタ。
<i>max_len</i>	インスタンス配列で利用できる要素の数。

```
#include <esnmp.h>
OID      *incoming = &method->varbind->name;
OBJECT    *object   = method->object;
int       instLength;
unsigned int instance[6];

-- in a GET operation --
-- Expected Instance is N.1.A.A.A.A where A's are IP address --
instLength = oid2instance(incoming, object, instance, 6);
if (instLength != 6)
    return ESNMP_MTHD_noSuchInstance;
```

N は instance[0] , IP アドレスは instance[2] , instance[3] , instance[4] および instance[5] になります。

戻り値は次のとおりです。

- <0 — エラーが起きました。この oid を見ることによってオブジェクトが取得された場合、これは返されません。
- 0 — インスタンス要素がありません。
- >0 — インデックスの要素数です (これは *max_len* パラメータの値より大きい場合もあります)。

6.3.3.9 inst2ip ルーチン

inst2ip ルーチンは、OID インスタンスから得られた IP アドレスを返します。Get および Set 操作のインスタンスの評価には、EXACT モードを使用します。GetNext および GetBulk 操作には、NEXT モードを使用します。NEXT モードを使用する場合、このルーチンのロジックは、一致するデータあるいはそれより大きいデータでデータ探索を実行すると想定します。

構文は次のとおりです。

```
int inst2ip(
    unsigned int *inst,
    int length,
    unsigned int *ipAddr,
    int exact,
    int carry);
```

引数は次のとおりです。

inst

IP アドレスへ変換するために oid2instance ルーチンによって返されるインスタンス番号を含む unsigned int の配列へのポインタ。

各要素は 0 ~ 255 の範囲にあります。EXACT モードを使用すると、要素が範囲外の場合 1 を返します。NEXT モードを使用すると、255 より大きな値は要素がオーバーフローする原因となります。0 にセットされ、次に重要な要素が増分されると、辞書的に次に来る `ipAddress` と同等な値を返します。

length

インスタンス配列の要素数です。4 つ目以降のインスタンスは無視されます。長さが 4 より小さい場合、足りない値には 0 が使用されます。長さが負の場合、`ipAddr` の値は 0 です。Get など正確な照合のためには、正確に 4 つの要素でなければなりません。

ipAddr

IP アドレス値を返す場所へのポインタです。ネットワーク・バイト順で最も顕著な要素が最初にきます。

exact

TRUE あるいは FALSE のどちらかです。

TRUE の場合 EXACT を使用します。要素が 255 より大きい場合、あるいは正確に 4 つの要素がない場合は、1 を返します。引数 *carry* は無視されます。

FALSE の場合 NEXT を使用します。*carry* が設定されていて長さが 4 以上ある場合は、辞書的に次にくる IP アドレスが返されます。要素数が 4 より小さい場合、足りない値には 0 が使用されます。いずれかの要素が 255 より大きい場合、0 が使用され次の要素が増加します。

carry

NEXT で IP アドレスを追加するキャリイです。次の IP アドレスを調べる場合は 1 を渡します。それ以外は 0 を渡します。4 より小さい長さはキャリイをキャンセルします。

次に示すのは、`inst2ip` ルーチンの例です。

次の例では、EXACT で、インスタンスを Get のための IP アドレスに変換します。

```
#include <esnmp.h>
OID      *incoming = &method->varbind->name;
OBJECT   *object   = method->object;
int instLength;
```

```

unsigned int instance[6];
unsigned int ip_addr;
int         iface;

-- The instance is N.1.A.A.A where A.A.A.A is the IP address--
instLength = oid2instance(incoming, object, instance, 6);
if (instLength == 6 && !inst2ip(&instance[2], 4, &ip_addr, TRUE, 0)) {
    iface = (int) instance[0];
}
else
    return ESNMP_MTHD_noSuchInstance;

```

次の例は、キーが1つしかないか、または *ipAddr* 値がキーの一番重要でない部分の場合の *GetNext* オペレーションを示しています。これは *NEXT* であるため、*carry* 値に1が渡されます。

```

#include <esnmp.h>
OID      *incoming = &method->varbind->name;
OBJECT    *object   = method->object;
int instLength;
unsigned int instance[6];
unsigned int ip_addr;
int         iface;

-- The instance is N.1.A.A.A where A.A.A.A is the IP address--
instLength = oid2instance(incoming, object, instance, 6);
iface = (instLength < 1) ? 0 : (int) instance[0];

iface += inst2ip(&instance[2], instLength - 2, &ip_addr, FALSE, 1);

```

次の例では、検索キーに複数の部分があり、*GetNext* オペレーションを行っている場合は、検索キーについて次にありそうな値を見つけて、カスケード的に「greater-than」または「equal-to」検索をします。

OID のインスタンス部分で *N.A.A.A.A.B.B.B.B* として表現される1つの数と2つの *ipAddr* 値からなる検索キーがある場合は (*N* は1つの値を持つ整数、*A.A.A.A* は1つのIPアドレス、*B.B.B.B* は2番目のIPアドレスとし、すべて指定された場合の長さが9になる)、キーの一番重要でない部分 (*B.B.B.B* の部分) を変更することによって検索を開始します。*inst2ip* ルーチン呼び出し、キャリイに1を渡し、長さに (*length-5*) を渡すことによってこれを行います。

B.B.B.B 部分の変換でキャリイが生成された場合 (1 が返される) は、それをキーの次に重要な部分に渡します。

したがって、*inst2ip* ルーチン呼び出し、*B.B.B.B* 部分の変換から返されたキャリイおよび長さに (*length-1*) を渡すことにより、*A.A.A.A* 部分を変換します。最も重要な要素である *N* は数です。したがって、*A.A.A.A* 変

換から返されたキャリイをその数に加算します。それもオーバーフローした場合、それは有効な検索キーではありません。

```
#include <esnmp.h>
OID      *incoming = &method->varbind->name;
OBJECT    *object   = method->object;
int instLength;
unsigned int instance[9];
unsigned int ip_addrA;
unsigned int ip_addrB;
int        iface;
int        carry;

-- The instance is N.A.A.A.A.B.B.B.B --
instLength = oid2instance(incoming, object, instance, 9);
iface = (instLength < 1) ? 0 : (int) instance[0];
carry = inst2ip(&instance[5], instLength - 5, &ip_addrB, FALSE, 1);
carry = inst2ip(&instance[1], instLength - 1, &ip_addrA, FALSE, carry);
iface += carry;
if (iface > carry) {
-- a carry caused an overflow in the most significant element
    return ESNMP_MTHD_noSuchInstance;
}
```

戻り値は次のとおりです。

- *carry* が 0 の場合、ルーチンは正常に完了しています。
- EXACT の場合あるいは NEXT に対するキャリイがある場合は、*carry* が 1 だと、エラーを意味します。キャリイがある場合、返される *ipAddr* は 0 です。

6.3.3.10 cmp_oid ルーチン

cmp_oid ルーチンは 2 つの OID 構造体を比較します。このルーチンは、要素 0 から各要素ごとに比較を行います。他の要素がすべて等しければ、最も小さな要素の OID 構造体がより小さいと考えられます。

構文は次のとおりです。

```
int cmp_oid(
    OID *q,
    OID *p);
```

戻り値は次のとおりです。

- +1 — oid *q* が oid *p* より大きい。
- 0 — oid *q* と oid *p* が等しい。
- -1 — oid *q* が oid *p* より小さい。

6.3.3.11 cmp_oid_prefix ルーチン

cmp_oid_prefix ルーチンは、プレフィックスに対して OID 構造体を比較します。プレフィックスは、おそらくオブジェクト・テーブル内のオブジェクトの OID です。プレフィックスを越える要素はインスタンス情報です。

このルーチンは、要素 0 から各要素ごとに比較を行います。プレフィックス OID のすべての要素が OID *q* 構造体の対応する要素と正確に一致する場合は、OID *q* 構造体に別の要素が含まれていても等しいと考えられます。最初の一致しない要素が大きければ、OID *q* 構造体は OID プレフィックスより大きいと考えられます。最初の一致しない要素が小さければ、OID *q* はプレフィックスより小さいと考えられます。

構文は次のとおりです。

```
int cmp_oid_prefix(  
    OID *q,  
    OID *prefix);
```

cmp_oid_prefix ルーチンの例を次に示します。

```
#include <esnmp.h>  
OID *q;  
OBJECT *object;  
if (cmp_oid_prefix(q, &object->oid) == 0)  
    printf("matches prefix\n");
```

戻り値は次のとおりです。

- -1 — oid がプレフィックスより小さい。
- 0 — oid がプレフィックスと等しい。
- +1 — oid がプレフィックスより大きい。

6.3.3.12 clone_oid ルーチン

clone_oid ルーチンは OID 構造体のコピーを作成します。

このルーチンは要素のバッファを動的に割り当て、そのポインタを渡される OID 構造体に挿入します。

このルーチンを使うには、コピーされる古い OID 構造体へのポインタとコピーされた OID 値を受け取る新しい OID 構造体へのポインタを渡します。

このバッファの解放は呼び出し元のプログラムもしくはモジュールの責任で行われます。

新しい OID 構造体によってポイントされる以前の要素のバッファは解放され、動的に割り当てられる新しいバッファへのポインタが挿入されます。解放できる要素バッファを含んでいない場合は、OID 構造体を 0 で初期化します。

このルーチンは OID 構造体を割り当てません。

古い OID 構造体が NULL であるか、要素バッファへの NULL ポインタを含んでいる場合は、{0.0} の新しい OID が生成されます。

構文は次のとおりです。

```
OID *clone_oid(  
    OID *new,  
    OID *oid);
```

引数は次のとおりです。

new

コピーを受け取る OID 構造体へのポインタです。

oid

データを入手する OID 構造体のポインタです。

clone_oid ルーチンの例を次に示します。

```
#include <esnmp.h>  
OID oid1;  
OID oid2;  
:  
: assume oid1 gets assigned a value  
:  
memset(&oid2, 0, sizeof(OID));  
if (clone_oid(&oid2, &oid1) == NULL)  
    ESNMP_LOG((WARNING, "It did not work\n"));
```

戻り値は次のとおりです。

NULL

エラーが生じました。そうでない場合は新しい OID へのポインタ (最初の引数) が返されます。

6.3.3.13 free_oid ルーチン

free_oid ルーチンは、OID 構造体の要素バッファを解放します。

`oid->elements` フィールドによってポイントされるバッファを解放する場合、そのフィールドと `oid->nelem` フィールドをゼロにします。

このルーチンは `OID` 構造体自身の割り当て解除は行いません。要素バッファの割り当て解除のみを行います。

構文は次のとおりです。

```
void free_oid(  
    OID *oid);
```

`free_oid` ルーチンの例を次に示します。

```
#include <esnmp.h>  
OID oid;  
:  
: assume oid was assigned a value (perhaps with clone_oid())  
: and we are now finished with it.  
:  
free_oid(&oid);
```

6.3.3.14 clone_buf ルーチン

`clone_buf` ルーチンは、動的に割り当てられた領域のバッファのコピーを作成します。1つの特別なバイトが最後に割り当てられ、`\0` が埋め込まれます。`len` パラメータが0より小さい場合は、複製バッファ長は0に設定されます。ルーチンは、`malloc` エラーがない限り常にバッファ・ポインタを返します。

割り当てたバッファの解放は呼び出し元の責任で行われます。

構文は次のとおりです。

```
char *clone_buf(  
    char *str,  
    int len);
```

引数は次のとおりです。

`str`

コピーを作成したバッファへのポインタ。

`len`

コピーするバイト数。

`clone_buf` ルーチンの例を次に示します。

```
#include <esnmp.h>  
char *str = "something nice";
```

```
char *copy;
copy = clone_buf(str, strlen(str));
```

戻り値は次のとおりです。

NULL

malloc エラーが生じました。それ以外の場合は、オリジナル・バッファのコピーを含む割り当てバッファへのポインタが返されます。

6.3.3.15 mem2oct ルーチン

mem2oct ルーチンは文字列 (バッファおよび長さ) を OCT 構造体に変換します。

このルーチンは新しいバッファを動的に割り当て、指定されたデータをそこにコピーし、OCT 構造体を新しいバッファのアドレスと長さで更新します。

このバッファの解放は呼び出し元の責任で行われます。

このルーチンは OCT 構造体を割り当てません。以前に割り当てた OCT 構造体でポイントしているデータの解放は行いません。

構文は次のとおりです。

```
OCT * mem2oct (
    OCT *new,
    char *buffer,
    int len);
```

mem2oct ルーチンの例を次に示します。

```
#include <esnmp.h>
char buffer;
int len;
OCT abc;
:
: buffer and len are initialized to something
:
memset(&abc, 0, sizeof(OCT));
if (mem2oct(&abc, buffer, len) == NULL)
    ESNMP_LOG((WARNING, "It did not work...\n"));
```

戻り値は次のとおりです。

NULL

エラーが生じました。それ以外の場合は、OCT 構造体へのポインタ (最初の引数) が返されます。

6.3.3.16 cmp_oct ルーチン

cmp_oct は 2 つのオクテット文字列を比較します。2 つのオクテット文字列は、バイトごとに最も短いオクテット文字列の長さを比較されます。すべてのバイトが等しい場合は、長さが比較されます。ヌル・ポインタのオクテットは、長さ 0 のオクテットと同じであると考えられます。

構文は次のとおりです。

```
int cmp_oct(  
    OCT *oct1,  
    OCT *oct2);
```

cmp_oct ルーチンの例を次に示します。

```
#include <esnmp.h>  
OCT abc, efg;  
:  
: abc and efg are initialized to something  
:  
if (cmp_oct(&abc, &efg) > 0)  
    ESNMP_LOG((WARNING,"octet abc is larger than efg...\n"));
```

戻り値は次のとおりです。

- -1 — 最初のパラメータでポイントされる文字列は 2 つ目よりも小さい。
- 0 — 最初のパラメータでポイントされる文字列は 2 つ目と等しい。
- +1 — 最初のパラメータでポイントされる文字列は 2 つ目より大きい。

6.3.3.17 clone_oct ルーチン

clone_oct ルーチンは、OCT 構造体のコピーを作成します。

呼び出し元は、コピーされる古い OCT 構造体のへのポインタとコピーされた OCT 構造体の値を受け取る新しい OCT 構造体へのポインタを渡します。

このルーチンはバッファを動的に割り当て、バッファのアドレスと長さで新しい OCT 構造体を更新します。

この割り当てられたバッファの解放は呼び出し元の責任で行われます。

新しい OID 構造体によってポイントされる以前のバッファは解放され、動的に割り当てられる新しいバッファへのポインタが挿入されます。解放できる要素バッファを含んでいない場合は、新しい OCT 構造体を 0 で初期化します。

このルーチンは OCT 構造体を割り当てません。OCT 構造体によってポイントされる要素バッファのみを割り当てます。

構文は次のとおりです。

```
OCT *clone_oct(  
    OCT *new,  
    OCT *old);
```

引数は次のとおりです。

new

コピーを受け取る OCT 構造体へのポインタです。

old

データ入手する OCT 構造体のポインタです。

clone_oct ルーチンの例を次に示します。

```
#include <esnmp.h>  
OCT octet1;  
OCT octet2;  
:  
: assume octet1 gets assigned a value  
:  
memset(&octet2, 0, sizeof(OCT));  
if (clone_oct(&octet2, &octet1) == NULL)  
    ESNMP_LOG((WARNING, "It did not work\n"));
```

戻り値は次のとおりです。

NULL

エラーが生じました。それ以外の場合は、OCT 構造体へのポインタ (最初の引数) が返されます。

6.3.3.18 free_oct ルーチン

free_oct ルーチンは OCT 構造体のバッファを解放します。

OCT 構造体がポイントする動的に割り当てられたバッファを解放し、OCT 構造体のポインタと長さフィールドを 0 にします。OCT 構造体がすでに NULL の場合は何もしません。OCT 構造体に接続されたバッファがすでに NULL の場合は、OCT 構造体の length フィールドを 0 にします。

このルーチンは OCT 構造体の割り当て解除は行わず、ポイントするバッファの割り当て解除のみを行います。

構文は次のとおりです。

```
void free_oct(  
    OCT *oct);
```

free_oct ルーチンの例を次に示します。

```
#include <esnmp.h>
OCT octet;
:
: assume octet was assigned a value (perhaps with clone_oct())
: and we are now finished with it.
:
free_oct(&octet);
```

6.3.3.19 free_varbind_data ルーチン

free_varbind_data ルーチンは、VARBIND 構造体に動的に割り当てられたフィールドを解放します。

このルーチンは、free_oid(vb->name) 操作を行います。vb->type フィールドを指定すると、free_out あるいは free_oid ルーチンのどちらかを使用して vb->value データを解放します。

VARBIND 構造体自身の割り当て解除は行わず、それがポイントする名前バッファおよびデータ・バッファのみを割り当て解除します。

構文は次のとおりです。

```
void free_varbind_data(
    VARBIND *vb);
```

free_varbind_data ルーチンの例を次に示します。

```
#include <esnmp.h>
VARBIND *vb;
:
: assume oid and data are declared and
: assigned appropriate values
:
vb = (VARBIND*)malloc(sizeof(VARBIND));
clone_oid(&vb->name, oid);
clone_oct(&vb->value.oct, data);
:
: some processing that uses vb occurs here
:
free_varbind_data(vb);
free(vb);
```

6.3.3.20 set_debug_level ルーチン

set_debug_level ルーチンは、どのようなログ・メッセージが生成されたかを検出するロギング・レベルを設定します。プログラムもしくはモジュールが、実行時オプションに応じて、プログラムの初期化時にこのルーチンを

呼び出します。このルーチンが呼び出されていない場合、省略時の設定として WARNING および ERROR メッセージが `stdout` に送られます。

構文は次のとおりです。

```
void set_debug_level(  
    int stat,  
    LOG_CALLBACK_ROUTINE callback_routine);
```

引数は次のとおりです。

stat

ログ・レベル。次の値が、個別もしくは組み合わせで設定されます。

ERROR

エラーが発生し、再起動を必要とする場合。

WARNING

パケットを処理できない場合。ERROR も含む。

TRACE

すべてのパケットをトレースする場合。ERROR および WARNING を含む。

DAEMON_LOG

標準出力でなく `syslog` へ出力が送られる。

EXTERN_LOG

ログ・メッセージを出力するためにコールバック関数が呼び出される。このビットが設定されている場合は、2 つ目の引数としてユーザが提供する外部コールバック関数のポインタを指定する必要がある。DAEMON_LOG および EXTERN_LOG が指定されていない場合は、標準出力へ送られる。

callback_routine

ユーザ定義の外部コールバック関数。たとえば、次のように定義します。

```
void callback_function(  
    int level,  
    char *message);
```

level には、ERROR、WARNING あるいは TRACE を指定します。

EXTERN_LOG ビットが *stat* に設定されている場合、ESNMP_LOG

マクロが実行されると `callback` 関数が呼び出され、ログ・レベルは生成するログ・メッセージを指定します。

この機能によって、eSNMP ライブラリ関数がログ・メッセージを出力する場所を制御できます。 `EXTERN_LOG` ビットが設定されていない場合は、コールバック関数の引数として `NULL` ポインタが渡されます。

`set_debug_level` ルーチンの例を次に示します。

```
#include <esnmp.h>
extern void log_handler(int level, char *message);

if (daemonize)
    set_debug_level(EXTERN_LOG | WARNING, log_handler);
else
    set_debug_level	TRACE, NULL);
```

6.3.3.21 `is_debug_level` ルーチン

`is_debug_level` ルーチンは、指定したレベルが設定されているかどうかロギングのレベルをテストします。次のようにレベルをテストできます。

- `ERROR` — エラーが発生し、再起動を必要とする場合。
- `WARNING` — パケットを処理できない場合。 `ERROR` も含む。
- `TRACE` — すべてのパケットをトレースする場合。 `ERROR` および `WARNING` も含む。
- `DAEMON_LOG` — `syslog` に出力する場合。
- `EXTERN_LOG` — ログ・メッセージを出力するために `callback` 関数を呼び出す場合。

構文は次のとおりです。

```
int is_debug_level(
    int type);
```

戻り値は次のとおりです。

TRUE 要求したレベルが設定されており `ESNMP_LOG` が出力を生成します。あるいは指定した場所へ出力されます。

FALSE ロギング・レベルが設定されていません。

`is_debug_level` ルーチンの例を次に示します。

```
#include <esnmp.h>

if (is_debug_level(TRACE))
    dump_packet();
```

6.3.3.22 ESNMP_LOG ルーチン

ESNMP_LOG ルーチンは、<esnmp.h> ヘッダファイルで定義されているエラー宣言 C マクロです。取得できる情報を集め、ログに送ります。DAEMON_LOG が設定されている場合、ログ・メッセージはデーモン・ログへ送られます。EXTERN_LOG が設定されている場合、ログ・メッセージは *callback* 関数へ送られます。それ以外の場合は、標準出力へ出力されます。

注意

esnmp_log ルーチンは、ESNMP_LOG マクロを使用して呼び出されます。ESNMP_LOG マクロは、テキスト部分をフォーマットするのにヘルパー・ルーチン *esnmp_logs* を使用します。これらの関数は ESNMP_LOG マクロなしで使用しないでください。

```
#define ESNMP_LOG(level, x) if (is_debug_level(level)) { \
    esnmp_log(level, esnmp_logs x, __LINE__, __FILE__); }
```

x には次の構文で文字列を指定します。

text - format, arguments,

たとえば *printf* 文を指定します。

level

次のいずれかを指定します。

ERROR エラー状態を宣言します。

WARNING 警告を宣言します。

TRACE トレースがアクティブな場合ログ・ファイルを挿入します。

構文は次のとおりです。

```
ESNMP_LOG(level, (format, ...))
```

ESNMP_LOG ルーチンの例を次に示します。

```
#include <esnmp.h>
ESNMP_LOG( ERROR, ("Cannot open file %s\n", file));
```

RSVP アプリケーション・プログラミング・インタフェース

RSVP (ReSerVation Protocol) はネットワーク層プロトコルです。これによって、インターネット・アプリケーションは、特定アプリケーションのデータ・ストリームまたはフロー (単向通信のユニキャストとマルチキャストのいずれも) に対して、より高いサービス品質 (QoS) を要求できます。アプリケーションは、一般には、省略時の値である最善の配送ができない場合 (たとえば、ビデオやオーディオなど) に、RAPI (RSVP Application Programming Interface) を用いてより高い QoS を要求します。アプリケーションが要求できる QoS のタイプは、Internet Integrated Services で定義されています。

この章の説明は、次の事項について理解していることを前提としています。

- IETF Integrated Services (RFC 1633)
- RSVP プロトコル (RFC 2205)
- RSVP および Integrated Services (RFC 2210)
- Controlled-Load Service (RFC 2211)
- C プログラミング言語

この章で説明する内容は、次のとおりです。

- ネットワークのサービス品質の概要
- ネットワークの QoS 構成要素についての説明
- RSVP アプリケーション・プログラミング・インタフェース (RAPI) ルーチンについての説明

7.1 ネットワークのサービス品質

アプリケーションがインターネットのネットワークに要求する帯域幅は増え続けているため、ネットワークの帯域幅を増やしても一時的な解決策にしかなりません。新しいリアルタイムのアプリケーションは、帯域幅の増大と短

い待ち時間の両方を要求しています。明らかに、ネットワーク内での帯域幅の使用を管理するメカニズムが必要になっています。

現在、最善の配信サービスを使用する IP ネットワークでは、すでに受動的な帯域幅管理を行っています。送信キューが満杯、すなわちネットワークのトラフィックが高くて輻輳している場合、パケットは通知なしで捨てられます。上位レベルのプロトコルにはこのデータ損失を検出するものもありますが、検出できないものもあります。

サービス品質 (QoS) とは、一般にネットワークの帯域幅を実際に管理するという考え方に伴って用いられる熟語です。このシナリオでは、ネットワークの要素すべて (ホスト、アプリケーション、ルータなど) とネットワークのプロトコル層すべてが協調して、ネットワーク内のエンド間のトラフィックとサービスが堅実に実行されるようにします。リアルタイム・アプリケーション用のネットワーク帯域幅が予約され、同時に最善の配送用のトラフィックのためにも十分な帯域幅が保持されます。

7.2 ネットワークのサービス品質の構成要素

このオペレーティング・システムでのネットワークのサービス品質の主要な構成要素には、次のものがあります。

- Traffic Control
- RSVP
- RAPI

これらの構成要素は互いに協調してネットワークの QoS を提供しています。以下の各節で、この構成要素それぞれについて説明します。

7.2.1 Traffic Control

このオペレーティング・システムでは、Traffic Control サブシステムは RFC 2211 で定義されているとおりに Controlled-Load サービスを実現しています。このサービスでは、負荷を軽減したネットワーク・インタフェースにより、最善の配送に近い QoS のアプリケーション・データ・フローを実現しています。Traffic Control サブシステムには、次のような構成要素があります。

- Admission control — ローカルのシステム・インタフェースに要求された帯域幅があり、フローとフィルタをインストールしていることを確認します。

- Packet classifier — 出力 IP ヘッダとインストールしたフィルタを照合します。
- Packet rate enforcer — 各出力フローが、合意した境界内にあることを保証します。
- `iftcntl` ユーティリティ — 手動でフローを設定し、存在しているフローとフィルタを表示します。

Traffic Control は、イーサネットと FDDI インタフェースでサポートされています。

7.2.2 RSVP

RSVP は、RFC 2205 で定義されるネットワーク・シグナリング・プロトコルです。これには、ローカル・システムとネットワークで帯域幅を予約するメカニズムがあります。このオペレーティング・システムでは、RSVP は `rsvdpd` デーモンという形で実現されています。このデーモンは、RSVP が有効なアプリケーションとの通信を行い、RSVP メッセージを送信し、RSVP メッセージの受信と処理を行います。`rsvdpd` デーモンは Traffic Control サブシステムを用いて、特定のネットワーク・インタフェースに対するフローとフィルタのインストールと変更を行います。

7.2.3 RAPI

RAPI は、`rsvdpd` デーモンとの通信のためにより高い QoS を必要とするローカル・アプリケーションを有効にします。RAPI ルーチンを使用すると、アプリケーションはローカル・システムでリソース (帯域幅) を予約したり、ネットワークの他のノードにサービスを通知したり、その両方を行えるようになります。このオペレーティング・システムでは、The Open Group が発行している Resource ReSerVation Protocol API (RAPI) で定義されているとおりに、RAPI を実現しています。この技術標準を、この節と一緒にお読みください。サポートされているルーチンのリストとその説明については、7.5 節を参照してください。

7.2.4 構成要素の相互運用

RSVP には、次のような規則があります。

- 送信側 (またはデータ・ソース) — IP ソース・アドレスおよびソース・ポートにより定義。

- 受信側 — トランスポート・プロトコル, IP デスティネーション・アドレス, デスティネーション・ポートにより定義。

アプリケーションは送信側にも受信側にもなります。

一般的なシナリオでは, リモート・ホストはローカル・アプリケーションと通信して, データの受信を依頼します。ローカル・アプリケーションが自分をデータ送信側であると宣言した場合, 次のようなイベントが発生することがあります。

1. アプリケーションが RAPI インタフェースを通じて `rsvpd` との通信を開始する。
2. アプリケーションが送信トラフィックまたはフローの特性 (帯域幅の上限と下限, 遅延, ジッタなど) を `rsvpd` に渡す。このような特性を, **Tspec** といいます。デーモンには **Adspec** も渡されます。
3. `rsvpd` デーモンがこの情報を含む PATH メッセージを作成し, それを Traffic Control に渡す。`rsvpd` デーモンと Traffic Control との相互作用は, 要求しているアプリケーションには見えません。
4. Traffic Control が, 予約に必要な帯域幅がローカル・アダプタにあるかどうかを判断する。存在する場合には, Traffic Control はメッセージを伝送します。ユニキャスト・アドレスにもマルチキャスト・アドレスにも伝送できます。
5. RSVP が有効な各ルータで, Traffic Control は, 直前のソース・アドレス (送信側のすぐ上流のホップ) で PATH メッセージを更新し, ノード上にある QoS 制御サービスに依存する **Adspec** を変更し, メッセージを下流のデスティネーションに伝送する。
6. 受信側で, `rsvpd` は **Tspec** と **Adspec** を RAPI を通じて受信側アプリケーションに渡す。

上記のリストは, 実行される手順を要約したものです。手順がさらに必要になることもあります。

ローカル・アプリケーションが予約の要求を行おうとする (自分を受信側と宣言) と, 次のようなイベントが発生することがあります。

1. アプリケーションが RAPI インタフェースを通じて `rsvpd` デーモンとの通信を開始する。

2. アプリケーションが、受信側が予測するトラフィック (**Tspec**)、必要な QoS レベル (**Rspec**)、パケットに対するトランスポート・プロトコルおよびポート番号 (**filter spec**) を `rsvdpd` に渡す。Rspec および Tspec は、フロー記述子すなわち **flowspec** と見なされます。

また、Rspec はルータまたはホストでパケットのスケジューリングを制御するメカニズムとしても使用されます。また、filter spec はパケットの分類を制御して、どの送信側のデータ・パケットが対応する QoS を受けるかを判断します。
3. `rsvdpd` デーモンがこの情報を含む RESV メッセージを作成し、それを Traffic Control に渡す。`rsvdpd` デーモンと Traffic Control との相互作用は、要求しているアプリケーションには見えません。
4. Traffic Control が、予約に必要な帯域幅がローカル・アダプタにあるかどうかを判断する。存在する場合には、Traffic Control は PATH メッセージ内のソース・アドレスを用いてメッセージを上流に伝送します。
5. RSVP が有効な各ルータで、Traffic Control は、予約要求を認証し、Tspecs および Flowspec で要求された必要なリソースを割り当てる。認証に失敗した場合や十分なリソースがない場合、ルータは受信側に RSVP エラーを返します。
6. RSVP が有効な最後のルータで、要求が受理されると、ルータは RSVP 確認メッセージを受信側に返す。
7. RSVP セッション・データの送信側のそれぞれで、RSVP はマージした Flowspec オブジェクトをアプリケーションに渡す。これによって、要求の送信側とデータ・パスのプロパティを通知します。
8. 送信側は、RSVP 予約パラメータに同意する場合は RSVP を介して予約を受諾し、個別のデータ接続を用いて受信側へのデータを送信し始める。

上記のリストは、実行される手順を要約したものです。手順がさらに必要になることもあります。

異なる受信者からの予約がインターネット内で共有される方法の詳細は、予約スタイルという予約パラメータで制御されています。いろいろな予約スタイルについての詳細は、RFC 2205 を参照してください。

7.3 Traffic Control

Traffic Control サブシステムは、次のタスクを実行します。

- インタフェース・パラメータを保持する承認制御メカニズムを実現する。パラメータには、デバイスのピーク出力レート、Controlled-Load サービスに予約できる帯域幅のパーセンテージ、同時フローの最大数などがあります。
- アプリケーションが、flowspec よりも速いレートでデータを出さないようにする。
- rsvpd デーモンと iftcntl コマンドのインタフェースをとって、フローとフィルタのインストールと削除を行う。
- すべての出力パケットヘッダと既存の filter spec を照合し、どの出力キューにパケットを置くかを判断する。

詳細は、iftcntl(8) を参照してください。システムのトラフィック制御を有効にする際の情報については、『ネットワーク管理ガイド：接続編』を参照してください。

7.4 RSVP

RSVP は QoS を特定の IP データ・フローまたはセッションに割り当てます。これは、マルチポイント-マルチポイントでも、ポイント-ポイントでもかまいません。**RSVP** セッションは、トランスポート・プロトコル、IP デスティネーション・アドレス、デスティネーション・ポートによって定義されます。マルチキャスト・セッションに対するデータ・パケットを受信するには、ホストは、それに対応する IP マルチキャスト・グループに属していなければなりません。セッションには送信側が複数であることもあり、デスティネーションがマルチキャスト・アドレスであれば、受信側も複数です。

以降の節では、オペレーティング・システムの RSVP 構成要素と、rsvpd デーモンでの動作について説明します。

7.4.1 RSVP の構成要素

オペレーティング・システムの RSVP 構成要素は次のとおりです。

- /usr/sbin/rsvdpd — RSVP デーモン。
- /usr/sbin/rsvpstat — リソース予約状態を表示するプログラム
- /usr/shlib/librsvp.so — RSVP ライブラリ
- /usr/include/rapi_lib.h — RAPI の定義
- /usr/include/rapi_err.h — RSVP および RAPI のエラー定義

7.4.2 rsvdpd デーモン

rsvdpd デーモンは次の機能を実行します。

- raw IP ソケットで、受信 RSVP メッセージをリッスンする。
- RAPI (RSVP Application Programming Interface) を通じて、ローカル・ホストのアプリケーションと通信する。
- オペレーティング・システムの Traffic Control サブシステムとのインタフェースをとる。

rsvdpd は、RSVP メッセージをネットワークから受信すると、ローカルのインタフェース・データベースに対して確認を行い、認証またはプロトコル・エラーを処理します。要求が有効であれば、rsvdpd デーモンは以下の動作をします。

- RSVP セッションとフローの動きを見る。
- フローに対するリソース予約を処理する。
- rsvdpd に登録されている場合、RSVP メッセージをローカルのアプリケーションに渡す。
- ローカル・ホストのインタフェースに照合しないデスティネーション・アドレスを持つ RSVP パケットを、次のホップに経路指定する。

デーモンのオプションについての詳細は rsvdpd(8) を参照してください。

rsvdpd の開始と停止についての詳細は、『ネットワーク管理ガイド：接続編』を参照してください。

7.5 RSVP アプリケーション・プログラミング・インタフェース

RAPI (RSVP Application Programming Interface) によって、アプリケーションは `rsvpd` デモンとの通信を確立でき、自分をデータの受信側、データの送信側、またはその両方として宣言できます。

アプリケーションは次のものから構成されます。

- 開発者が作成したメインの関数
- RSVP プロトコル作業を実行する RSVP ライブラリ・ルーチン
- RSVP コールバックを処理するために開発者が作成した関数

以降の節では、サポートされているルーチンと、開発者が RSVP 対応のアプリケーションを作成するために実行する必要がある手順について説明します。

7.5.1 サポートされているルーチン

RAPI は次のものから構成されています。

- クライアント・ライブラリ・サービス — アプリケーション・プログラムと `rsvpd` デモンの間の通信を実現するルーチン
- RAPI フォーマット・ルーチン — RAPI オブジェクトのフォーマットを便利にするためのオプションのルーチン

表 7-1 には、サポートされているクライアント・ライブラリ・サービス・ルーチンと、その説明があります。ルーチンとその構文およびパラメータについての詳細な説明は、各ルーチンのリファレンス・ページを参照してください。

表 7-1: クライアントのライブラリ・サービス・ルーチン

ルーチン	説明
<code>rapi_session(3)</code>	RAPI セッションをローカルに作成する。
<code>rapi_reserve(3)</code>	セッションに対するリソースの予約を作成、変更、削除する。
<code>rapi_sender(3)</code>	送信側のデータ・フロー・パラメータを定義、再定義、削除する。
<code>rapi_strerror(3)</code>	RAPI エラー・コードをエラー・メッセージ文字列にマップする。
<code>rapi_version(3)</code>	オペレーティング・システムで使用されている RAPI のバージョン番号を返す。

表 7-1: クライアントのライブラリ・サービス・ルーチン (続き)

ルーチン	説明
<code>rapi_release(3)</code>	RAPI セッションをクローズし、すべてのリソース予約を削除する。
<code>rapi_dispatch(3)</code>	RAPI イベントをディスパッチする。
<code>rapi_getfd(3)</code>	RAPI セッションに対応するファイル記述子を取得する。
<code>rapi_event_rtn_t(3)</code>	着信した非同期 RSVP イベントを受信するためのユーザ作成ルーチン。

RSVP 要求は、QoS のタイプを記述するオブジェクトを含んでいます。RAPI フォーマット・ルーチンは、アプリケーションが RAPI オブジェクトの内容を表示できるようにします。表 7-2 には、サポートされている RAPI フォーマット・ルーチンとその説明があります。ルーチンとその構文およびパラメータについての詳細な説明は、各ルーチンのリファレンス・ページを参照してください。

表 7-2: RAPI フォーマット・ルーチン

ルーチン	説明
<code>rapi_fmt_adspec(3)</code>	指定した RAPI Adspec を、指定アドレスおよび指定長のバッファにフォーマットする。
<code>rapi_fmt_filtspec(3)</code>	指定した RAPI filter spec を、指定アドレスおよび指定長のバッファにフォーマットする。
<code>rapi_fmt_flowspec(3)</code>	指定した RAPI flowspec を、指定アドレスおよび指定長のバッファにフォーマットする。
<code>rapi_fmt_tspect(3)</code>	指定した RAPI Tspec を、指定アドレスおよび指定長のバッファにフォーマットする。

RAPI オブジェクト、プロトコル動作、RAPI の戻り値についての詳細は、The Open Group が発行している Resource ReSerVation Protocol API (RAPI) Technical Standard を参照してください。

7.5.2 RAPI 対応アプリケーションの作成

UNIX アプリケーションを提供しているアプリケーション開発者として、RSVP インタフェースを実装しなければなりません。

このプロセスを、次の手順で説明します。

1. `rapi.h` ヘッド・ファイルをインクルードする。このファイルは、アプリケーションが RAPI を使用するために必要なデータ構造体、定数、関数プロトタイプをすべて定義しています。
2. RAPI 呼び出しをコーディングする。
`rsvpd` デーモンとの通信を初期化するために RAPI を呼び出すコードを作成します。

RSVP イベントを受信するコールバック・ルーチンを作成します。
3. アプリケーションの実行とテストを行う。

7.5.2.1 アプリケーションのリンク

アプリケーションをコンパイルしたら、それを `librsvp.so` 共有可能ライブラリまたは `librsvp.a` スタティック・ライブラリのいずれかとリンクします。このライブラリには、アプリケーションと `rsvpd` デーモンの間の通信を有効にするプロトコルの実現 (RAPI) が含まれています。

7.5.3 RAPI アプリケーションのデバッグ

アプリケーションのテストとデバッグを行う場合、`rsvpd` に `-d` フラグを付けて実行することができます。これにより、デーモンは強制的にエラー・メッセージとデバッグ出力を `/var/rsvp/rsvpd_dbg.log` ファイルに書き込みます。

また、`rsvpstat` を実行して、接続が行われているか、予約が守られているかを確認することもできます。ローカル・システム上でアクティブな RSVP セッションをモニタするには、次のコマンドを入力します。

```
# /usr/sbin/rsvpstat
```

省略時、`rsvpstat` コマンドは、このシステムでアクティブになっているすべての RSVP セッション、送信側、受信側を表示します。情報にはセッション番号、デスティネーション・アドレス、IP プロトコル、ポート番号、セッションに対する PATH および RESV 状態の番号が含まれます。

送信側からの実際の PATH メッセージの内容などの送信側の情報を表示するには、次のコマンドを入力します。

```
# /usr/sbin/rsvpstat -Sv
```

受信側からの実際の RESV メッセージの内容などの受信側の情報を表示するには、次のコマンドを入力します。

```
# /usr/sbin/rsvpstat -Rv
```

詳細は、`rsvpstat(8)` を参照してください。



Tru64 UNIX における STREAMS とソケットの共存

この章では、`ifnet` STREAMS モジュールと、`dlb` STREAMS 擬似ドライバの通信ブリッジについて説明します。この章は、STREAMS とソケットの基本的な概念に習熟しており、第 4 章および第 5 章の内容を理解していることが前提となっています。

オペレーティング・システムのネットワーク・プログラミング環境は、ネットワーク・プログラミング用に、STREAMS およびソケットの両方のフレームワークをサポートしています。ただし、この 2 つのフレームワーク間には、データ・リンク層における固有の通信パスはありません。共存という用語は、ソケットと STREAMS のフレームワーク間で、データを交換できることを指します。通信ブリッジという用語は、2 つのフレームワークがデータ・リンク層でデータを交換できるようにするソフトウェア (`ifnet` STREAMS モジュールまたは `dlb` STREAMS 擬似ドライバ) を指します。

ソケットおよび STREAMS に合わせて作成されたプログラムは、次の理由から相互通信をしなければなりません。

- システムは、同一デバイス用に 2 つのドライバを持つことができない。
- プログラムは、BSD プロトコル・スタックから STREAMS ベースのデバイス・ドライバにアクセスしなければならない場合がある。また逆に、STREAMS ベースのプロトコル・スタックから、BSD デバイス・ドライバにアクセスしなければならない場合もある。

たとえば、システムで STREAMS デバイス・ドライバを実行しているとき、Tru64 UNIX にインプリメントされたソケット・ベースの TCP/IP をアプリケーションが使用している場合、ソケット・ベースのプロトコル・スタックと STREAMS デバイス・ドライバとの間でデータのやり取りに使用するパスが必要になります。`ifnet` STREAMS モジュールによって、TCP/IP を使用するアプリケーションは、STREAMS デバイス・ドライバとデータを交換できます。`ifnet` STREAMS モジュールについては、8.1 節で説明します。

逆に、システムに STREAMS プロトコル・スタックがインプリメントされていても、Tru64 UNIX にインプリメントされた BSD デバイス・ドライバを使用したい場合は、STREAMS プロトコル・スタックと BSD デバイス・ドライバとの間でデータのやりとりに使用するパスが必要になります。dlb STREAMS 擬似ドライバによって、STREAMS プロトコル・スタックは、データを BSD デバイス・ドライバに経路指定できます。dlb STREAMS 擬似ドライバについては、8.2 節で説明します。

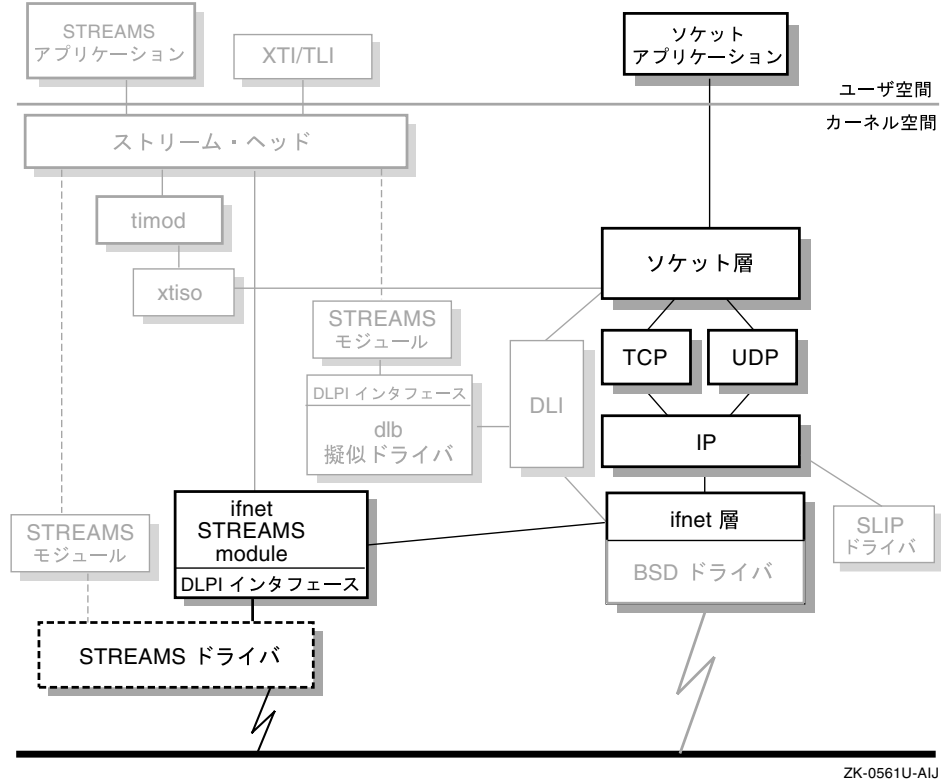
8.1 STREAMS ドライバからソケット・プロトコル・スタックへのブリッジ

ifnet STREAMS モジュールは、通信ブリッジです。これによって、STREAMS ネットワーク・ドライバは、ソケット・ベースのネットワーク・プロトコルにアクセスできます。ifnet STREAMS モジュールは、他の STREAMS モジュールと同様に機能し、STREAMS デバイス・ドライバの上にあるストリーム上にプッシュされます。一度、ストリーム上にプッシュされると、このモジュールは、STREAMS ドライバの DLPI インタフェースと BSD ifnet 層との間で必要なすべての変換処理を行います。ifnet STREAMS モジュールは、標準の STREAMS インタフェースと、ifnet 層インタフェースの両方をエクスポートします。

また、STREAMS ネットワーク・ドライバも、ifnet STREAMS モジュールを使用している間は、STREAMS ベースのネットワーク・プロトコルの使用を継続することができます。

図 8-1 は、ifnet STREAMS モジュールを強調表示して、ネットワーク・プログラミング環境における位置付けを示しています。

図 8-1: ifnet STREAMS モジュール



8.1.1 STREAMS ドライバ

この項では、STREAMS ドライバを実行して ifnet STREAMS モジュールを使用するシステムを準備する方法について説明します。

注意

ifnet STREAMS モジュールは、イーサネット STREAMS デバイス・ドライバだけをサポートします。

また、この項には、ifnet STREAMS モジュールを正常に操作するために、STREAMS ドライバがサポートしなければならない DLPI プリミティブの一覧も記載します。

8.1.1.1 ifnet STREAMS モジュールの使用

デバイス・ドライバが、8.1.1.2 項に示すプリミティブをサポートする場合には、ifnet STREAMS モジュールを使用するために、ドライバまたは STREAMS カーネル・コードのいずれもソース・コードを変更する必要はありません。

ifnet STREAMS モジュールを使用するためには、STRIFNET および DLPI オプションをカーネル構成ファイルに追加し、ドライバに対して STREAMS をセットアップする必要があります。

STRIFNET および DLPI は、インストール時にシステムに組み込むことができます (オプションの構成については、『インストール・ガイド』を参照)。オプションが構成されているかどうかは、次のコマンドで調べることができます。

```
# /usr/sbin/strsetup -c
```

ifnet あるいは dlb が名前の欄に表示されれば、それらのオプションはカーネルに組み込まれています。組み込まれていない場合は、doconfig コマンドを使用してそれらを組み込む必要があります。

STRIFNET および DLPI をカーネルに組み込むには、次の手順に従います。

1. スーパユーザとしてログインする。

2. /usr/sbin/doconfig コマンドを入力する。

構成ファイルをカスタマイズしている場合は /usr/sbin/doconfig -c コマンドを使用します。詳細については、doconfig(8) を参照してください。

3. カーネル構成ファイルの名前を入力する。

この名前は、大文字のシステム名です。通常、省略時の値が大カッコ ([]) の中表示されます。次に例を示します。

```
Enter a name for the kernel configuration file. [HOST1]: RETURN
```

4. システム構成ファイルを置き換えるかどうかのプロンプトに対して、y を入力する。

```
A configuration file with the name 'HOST1' already exists.  
Do you want to replace it? (y/n) [n]: y
```

```
Saving /sys/conf/HOST1 as /sys/conf/HOST1.bck
```

*** KERNEL CONFIGURATION AND BUILD PROCEDURE ***

5. カーネルに組み込むオプションをメニューから選択する。

_____ 注意 _____

STRIFNET および DLPI オプションは、メニューからは選択できません。これらのオプションを組み込むには、以下の手順で構成ファイルを編集する必要があります。

6. カーネル構成ファイルのオプション・セクションに DLPI および STRIFNET を追加する。

カーネル構成ファイルを編集するかどうかを尋ねられたら、`y` を入力する。

`doconfig` コマンドでは、`ed` エディタを使用して構成ファイルを編集できます。`ed` エディタの使用方法についての詳細は、`ed(1)` を参照してください。

次の例は、`ed` 編集セッションで、DLPI オプションおよび STRIFNET オプションを `host1` 用のカーネル構成ファイルに追加する方法を示しています。新しい行を追加する行番号は、カーネル構成ファイルによって異なる場合があります。

```
Do you want to edit the configuration file? (y/n) [n]: y
```

```
Using ed to edit the configuration file. Press return when ready,  
or type 'quit' to skip the editing session:  
2153
```

```
48a  
options          DLPI  
options          STRIFNET  
.  
1,$w  
2185  
q
```

```
*** PERFORMING KERNEL BUILD ***
```

7. 新しいカーネルが構築された後、そのカーネルを、`doconfig` が置いたディレクトリからルート・ディレクトリ (`/`) に移動して、システムをリブートする。

リブートすると、`strsetup -i` コマンドが自動的に実行され、新しい STREAMS モジュールに対してデバイス特殊ファイルが作成されます。

8. `strsetup -c` コマンドを実行して、デバイスが正しく構成されたことを確認する。

次の例は、`strsetup -c` コマンドからの出力を示しています。

```
# /usr/sbin/strsetup -c
STREAMS Configuration Information...Thu Nov  9 08:38:17 1995
```

Name	Type	Major	Module ID
----	----	-----	-----
clone		32	0
dlb	device	52	5010
dlpi	device	53	800
kinfo	device	54	5020
log	device	55	44
nuls	device	56	5001
echo	device	57	5000
sad	device	58	45
pipe	device	59	5304
xtisoUDP	device	60	5010
xtisoTCP	device	61	5010
xtisoUDP+	device	62	5010
xtisoTCP+	device	63	5010
ptm	device	64	7609
pts	device	6	7608
bba	device	65	24880
lat	device	5	5
pppif	module		6002
pppasync	module		6000
pppcomp	module		6001
bufcall	module		0
ifnet	module		5501
null	module		5002
pass	module		5003
errm	module		5003
ptem	module		5003
spass	module		5007
rspass	module		5008
pipemod	module		5303
timod	module		5006
tirdwr	module		0
ldtty	module		7701

Configured devices = 16, modules = 15

カーネルの再構成または `doconfig` コマンドについての詳細は、『システム管理ガイド』および `doconfig(8)` を参照してください。

ドライバ用にストリームをセットアップするには、次の手順に従ってください。

1. 次のようなアプリケーション・プログラムを作成する。

```
/*
 * Application program to set up the "pifnet" streams for IP
 * and ARP. This must be run prior to ifconfig
 */
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <stropts.h>
#include <sys/ioctl.h>
#include <signal.h>
#include "dlpihdr.h"

#define IP_PROTOCOL      0x800
#define ARP_PROTOCOL    0x806
#define PIFNET_IOCTL_UNIT 1236

main(argc, argv)
    int argc;
    char *argv[];
{
    extern char *getenv();
    char *p;
    short unit = 0;
    char devName[256];

    if (argc != 3) usage();
    strcpy(devName, argv[1]);
    unit = atoi(argv[2]);

    sigignore(SIGHUP);
    setupStream(devName, unit, IP_PROTOCOL);
    setupStream(devName, unit, ARP_PROTOCOL);

    /*
     * sleep forever to keep the Streams alive.
     */
    if (fork()) /* detach */
        exit();
    pause();
}

usage()
{
    fprintf(stderr, "usage: pifnetd devname unit-number\n");
    exit(1);
}

setupStream(devName, unit, serviceClass)
    char *devName;
    short unit;
    u_long serviceClass;
{
    int fd, status;
    dl_bind_req_t bindreq;
    dl_bind_ack_t bindack;
    int flags;
    struct strioctl str;
    struct strbuf pstrbufctl, pstrbufdata, gstrbufctl, \
```

```

gstrbufdata;

char ebuf[256];

/*
 * build the stream
 */
fd = open(devName, O_RDWR, 0);
if (fd < 0)
{
    sprintf(ebuf, " open '%s' failed", devName);
    perror(ebuf);
    exit(1);
}
if (ioctl(fd, I_PUSH, "ifnet") < 0)
{
    sprintf(ebuf, " ioctl I_PUSH failed");
    perror(ebuf);
    exit(1);
}

/*
 * tell pifnet the unit number for the device
 */
str.ic_cmd = PIFNET_IOCTL_UNIT;
str.ic_timeout = 15;
str.ic_len = sizeof (short);
str.ic_dp = (char *) &unit;
status = ioctl(fd, I_STR, &str);
if (status < 0)
{
    sprintf(ebuf, " %s - ioctl");
    perror(ebuf);
    exit(1);
}

/*
 * bind the stream to a protocol
 */
bindreq.dl_primitive = DL_BIND_REQ;
bindreq.dl_sap = serviceClass;
bindreq.dl_max_conind = 0;
bindreq.dl_service_mode = DL_CLDLS;
bindreq.dl_conn_mgmt = 0;
bindreq.dl_xidtest_flg = 0;
pstrbufctl.len = sizeof(dl_bind_req_t);
pstrbufctl.buf = (void *)&bindreq;

pstrbufdata.buf = (char *)0;
pstrbufdata.len = -1;
pstrbufdata.maxlen = 0;

status = putmsg(fd, &pstrbufctl, (struct strbuf *)0, 0);
if (status < 0)
{
    perror("putmsg");
    exit(1);
}

/*
 * Check requested binding
 */
gstrbufctl.buf = (char *)&bindack;
gstrbufctl.maxlen = sizeof(dl_bind_ack_t);
gstrbufctl.len = 0;
status = getmsg(fd, &gstrbufctl, (struct strbuf *)0, &flags);

```



```

        if (status < 0)
        {
            perror("getmsg");
            exit(1);
        }

        if (bindack.dl_primitive != DL_BIND_ACK)
        {
            errno = EPROTO;
            perror(" DL_BIND_ACK");
            exit(1);
        }
    }
}

```

このアプリケーション例では、ドライバの名前は `/dev/streams/ln` です。このアプリケーションは、2つのストリームを作成します。1つはインターネット・プロトコル (IP) 用であり、もう1つはアドレス解決プロトコル (ARP) 用です。ストリームをセットアップした後、このアプリケーションは、ストリームを保持しておくために、`pause` コマンドを使用して、実行を継続しなければなりません。

ドライバがスタイル 2 のドライバの場合には、アプリケーション・プログラムに `DL_ATTACH_REQ` プリミティブを追加しなければなりません。`DL_ATTACH_REQ` プリミティブまたはスタイル 2 のドライバについての詳細は、`/usr/share/doc/lib/dlpi/dlpi.ps` の DLPI 仕様を参照してください。

2. アプリケーションの実行可能ファイルを生成する。

エラーなしで実行できるまで、プログラムをコンパイル、リンク、およびデバッグします。

3. ユーザに都合のよいディレクトリに、実行可能プログラムを移動する。

実行可能プログラムは、どのディレクトリにもいれることができます。

4. このプログラムを呼び出す行を、`/sbin/init.d/inet` ファイルに追加する。

このプログラムは、リブートするたびに手動で起動できます。ただし、`/sbin/init.d/inet` ファイルに 1 行追加して、システムのリブート時に自動的に実行させると非常に便利です。この行は必ず、システムの `ifconfig` 行より前に追加してください。

次の例では、システムがリブートするたびに、`/sbin/init.d/inet` ファイルは、`/etc` ディレクトリにある `run_ifnet` というプログラムを実行します。

```

:
#
# Enable network
#
case $1 in

    echo "Configuring network"
    /sbin/hostname $HOSTNAME
    echo "hostname: \c"
    /sbin/hostname
    if [ "$NETDEV_0" != '' ]; then
        echo >/tmp/ifconfig_"$NETDEV_0".tmp
# place command invoking executable BEFORE \
ifconfig lines
    /etc/run_ifnet
        /sbin/ifconfig $NETDEV_0 $IFCONFIG_0 > \
            /tmp/ifconfig_"$NETDEV_0".tmp 2>&1
    if [ $? != 0 ]; then
        ERROR='cat /tmp/ifconfig_"$NETDEV_0".tmp`
        if [ "$ERROR" = "$ERRSTRING" ]; then
            /sbin/ifconfig $NETDEV_0 up
        else
            echo "$0: $ERROR"
        fi
    fi
    rm /tmp/ifconfig_"$NETDEV_0".tmp
fi
:

```

5. システムをリブートする。

/usr/sbin/shutdown -r コマンドを使用して、システムをシャットダウンし、自動的にリブートさせます。次に例を示します。

```
# /usr/sbin/shutdown -r now
```

8.1.1.2 データ・リンク・プロバイダ・インタフェース・プリミティブ

STREAMS デバイス・ドライバは、スタイル 1 またはスタイル 2 のいずれの DLPI プロバイダでもかまいません。これについては、/usr/share/doc/lib/dlpi/dlpi.ps のデータ・リンク・プロバイダ・インタフェース仕様に説明があります。この DLPI 仕様にオンラインでアクセスするには、OSFPGMR_{nnn} サブセットがインストールされていなければなりません。

このドライバでは、次の DLPI プリミティブをサポートしなければなりません。これらのプリミティブとその使用方法についての詳細は、DLPI 仕様を参照してください。

DL_PHYS_ADDR_REQ/DL_PHYS_ADDR_ACK

DL_BIND_REQ/DL_BIND_ACK

DL_UNBIND_REQ

DL_UNITDATA_REQ/DL_UNITDATA_IND/DL_UDERROR_IND

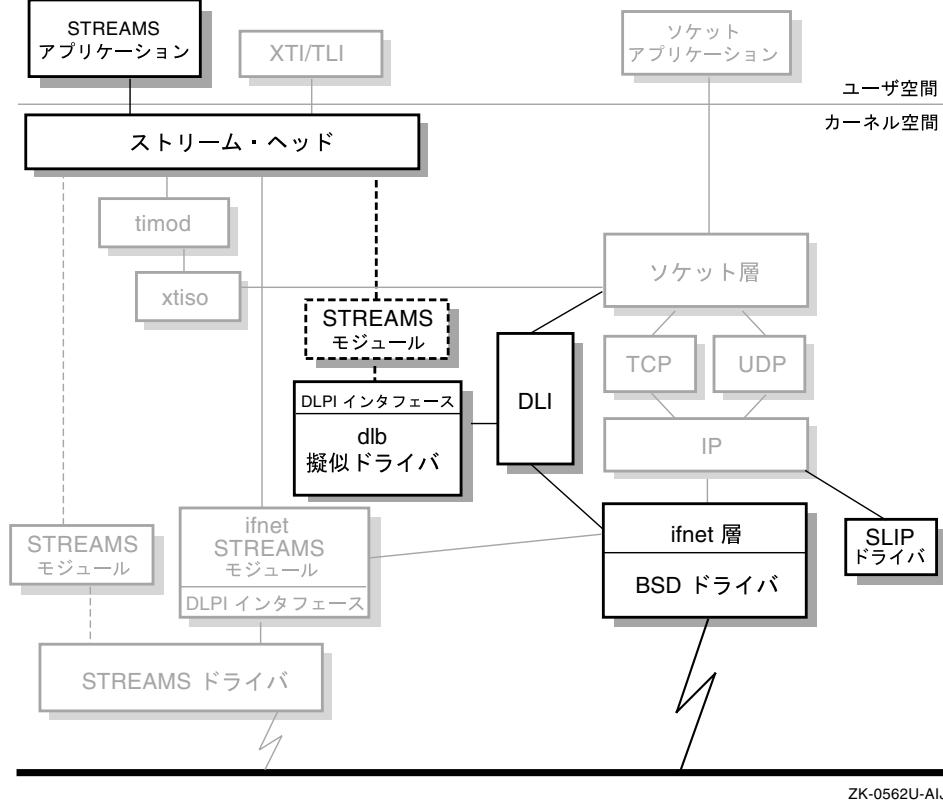
DL_OK_ACK/DL_ERROR_ACK

8.2 BSD ドライバから STREAMS プロトコル・スタックへのブリッジ

d1b STREAMS 擬似デバイス・ドライバによって、BSD スタイルのデバイス・ドライバと、STREAMS プロトコル・スタックをブリッジできます。STREAMS 擬似デバイス・ドライバは、BSD ベースのドライバとの通信を必要としている、ストリーム内のストリーム・エンドです。このオペレーティング・システムで提供されている STREAMS 擬似デバイス・ドライバには、2 つのインタフェースがあります。1 つは、STREAMS プロトコル・スタックと通信する DLPI インタフェースのサブセットであり、もう 1 つは、ソケット・フレームワークの `ifnet` 層インタフェースにアクセスするインタフェースです。

図 8-2 は、d1b STREAMS 擬似ドライバを強調表示して、ネットワーク・プログラミング環境における位置付けを示しています。

図 8-2: DLPI STREAMS 擬似ドライバ



ZK-0562U-AIJ

8.2.1 サポートする DLPI プリミティブおよびメディア・タイプ

dlb STREAMS 擬似ドライバは、次のコネクションレス・モードのプリミティブとメディア・タイプをサポートします。これらのプリミティブ、およびその使用方法についての詳細は、`/usr/share/doc/lib/dlpi/dlpi.ps` のデータ・リンク・プロバイダ・インタフェース仕様を参照してください。

`DL_ATTACH_REQ/DL_DETACH_REQ/DL_OK_ACK`

`DL_BIND_REQ/DL_BIND_ACK/DL_UNBIND_REQ`

`DL_ENABMULTI_REQ/DL_DISABLMULTI_REQ`

`DL_PROMISCON_REQ/DL_PROMISCONOFF_REQ`

`DL_PHYS_ADDR_REQ/DL_PHYS_ADDR_ACK`

`DL_SET_PHYS_ADDR_REQ`

DL_UNITDATA_REQ/DL_UNITDATA_IND

DL_SUBS_BIND_REQ/DL_SUBS_BIND_ACK

DL_SUBS_UNBIND_REQ/DL_SUBS_UNBIND_ACK

イーサネット・バス (DL_ETHER) は、STREAMS 擬似ドライバがサポートするメディア・タイプです。

8.2.2 STREAMS 擬似ドライバの使用

d1b STREAMS 擬似ドライバを使用するには、DLPI オプションをカーネルに組み込まなければなりません。DLPI オプションは、インストール時にシステムに組み込まれている場合もあります。

DLPI オプションが組み込まれているかどうかを調べるには、次のコマンドを実行します。

```
# /usr/sbin/strsetup -c
```

名前の欄に d1b が表示されれば、このオプションはカーネルに組み込まれています。表示されない場合は、doconfig コマンドで追加する必要があります。

doconfig コマンドを使用してカーネルを再構成する方法については、8.1.1.1 項を参照してください。

カーネルの再構成の方法、および doconfig コマンドについての詳細は、『システム管理ガイド』および doconfig(8) を参照してください。インストール時のオプションの構成については、『インストール・ガイド』を参照してください。



STREAMS モジュール例

spass モジュールは、プットされたすべてのメッセージを `putnext()` プロシージャに引き渡す、簡単な STREAMS モジュールです。spass モジュールは、サービス・プロシージャが処理するために、`putnext()` に対する呼び出しを遅延させます。このモジュールには、フロー制御コードが組み込まれており、読み取り側および書き込み側の両方が、1 つのサービス・プロシージャを共用しています。

spass モジュールのコードは、次のとおりです。

```
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/sysconfig.h>

static int      spass_close();
static int      spass_open();
static int      spass_rput();
static int      spass_srv();
static int      spass_wput();

static struct module_info minfo = {
    0, "spass", 0, INFPSZ, 2048, 128
};

static struct qinit rinit = {
    spass_rput, spass_srv, spass_open, spass_close, NULL, &minfo
};

static struct qinit winit = {
    spass_wput, spass_srv, NULL, NULL, NULL, &minfo
};

struct streamtab spassinfo = { &rinit, &winit };

cfg_subsys_attr_t bufcall_attributes[] = {
    {, 0, 0, 0, 0, 0, 0} /* must be the last element */
};

int
spass_configure(op, indata, indata_size, outdata, outdata_size)
    cfg_op_t      op;
    caddr_t      indata;
    ulong         indata_size;
    caddr_t      outdata;
    ulong         outdata_size;
{
    struct streamadm    sa;
    dev_t              devno = NODEV;

    if (op != CFG_OP_CONFIGURE)
```

```

        return EINVAL;

        sa.sa_version      = OSF_STREAMS_10;
        sa.sa_flags        = STR_IS_MODULE | STR_SYSV4_OPEN;
        sa.sa_ttys         = 0;
        sa.sa_sync_level   = SQLVL_QUEUE;
        sa.sa_sync_info    = 0;
        strcpy(sa.sa_name, "spass");
        if ( (devno = strmod_add(devno, &spassinfo, &sa)) == NODEV ) {
            return ENODEV;
        }

        return 0;
    }

/* Called when module is popped or the Stream is closed */
static int
spass_close (q, credp)
    queue_t * q;
    cred_t * credp;
{
    return 0;
}

/* Called when module is pushed */
static int
spass_open (q, devp, flag, sflag, credp)
    queue_t * q;
    int * devp;
    int flag;
    int sflag;
    cred_t * credp;
{
    return 0;
}

/*
 * Called to process a message coming upstream. All messages
 * but flow control messages are put on the read side service
 * queue for later processing.
 */
static int
spass_rput (q, mp)
    queue_t * q;
    mblk_t * mp;
{
    switch (mp->b_datap->db_type) {
    case M_FLUSH:
        if (*mp->b_rptr & FLUSHR)
            flushq(q, 0);
        putnext(q, mp);
        break;
    default:
        putq(q, mp);
        break;
    }
    return 0;
}

/*
 * Shared by both read and write sides to process messages put
 * on the read or write service queues. When called from the
 * write side, sends all messages on the write side queue
 * downstream until flow control kicks in or all messages are
 * processed. When called from the read side sends all messages

```



```

* on its read side service queue upstreams until flow control
* kicks in or all messages are processed.
*/
static int
spass_srv (q)
    queue_t * q;
{
    mblk_t *      mp;

    while (mp = getq(q)) {
        if (!canput(q->q_next))
            return putbq(q, mp);
        putnext(q, mp);
    }
    return 0;
}

/*
* Called to process a message coming downstream. All messages but
* flow control messages are put on the write side service queue for
* later processing.
*/
static int
spass_wput (q, mp)
    queue_t * q;
    mblk_t *      mp;
{
    switch (mp->b_datap->db_type) {
    case M_FLUSH:
        if (*mp->b_rptr & FLUSHW)
            flushq(q, 0);
        putnext(q, mp);
        break;
    default:
        putq(q, mp);
        break;
    }
    return 0;
}

```



B

ソケットおよび XTI プログラム例

この付録では、ソケットおよび XTI プログラム例として、クレジット・カード登録プログラムのサーバ・プログラムとクライアント・プログラム¹を示し、これを注釈付きで説明します。このプログラムでは、クライアントはカード・オペレータ (merchant) に代わってサーバにアクセスし、クライアントのクレジット・カードへの代金のチャージを認めるようにサーバに許可を要求します。サーバは登録されたカード・オペレータ (merchant) とそのパスワード、およびクレジット・カード利用者、クレジットの期限、現在の残高をデータ・ベースで管理し、この情報をもとにクライアントの要求を許可または拒否します。

以下に示されているサーバおよびクライアントのプログラム例は、コネクション指向型モードとコネクションレス型モードの各モードについて、それぞれソケット・コードと XTI コードの 2 種類のコードを使用して作成されたものです。

このプログラムは実際のアプリケーションのネットワーク・プログラミングを使用しますが、次の制限事項があります。

- エラー処理を行わない
- 整数のみを扱う
- 子プロセスのクリーン・アップを行わない
- B.1 節のコネクション指向型のプロトコル例では、サーバ・プログラムは要求を受信するたびにその要求の処理を行なう子プロセスをフォークし、サーバのデータ・ベース情報は子プロセス専用のデータ領域にデタッチされます。本来、この子プロセスは要求を分析し利用者のクレジット残高を差し引く際に、同じ利用者からの次の要求を正しく処理するために、その情報 (およびいくつかのパーシステント・ストレージ) をオリジナルのサーバのデータ領域含ませるためにデータ領域の更

¹ この付録中のクライアントという用語はカード・オペレータ (merchant) の操作によって、サーバ・プログラムとの通信を行なうプログラムのことを指します。

新を行なう必要がありますが、説明上必要以上に複雑になるのをさけるためにこのロジックは含まれていません。

この付録は次のように構成されています。

- コネクション指向型モード・プログラム
 - ソケット
 - ☐ サーバ
 - ☐ クライアント
 - XTI
 - ☐ サーバ
 - ☐ クライアント
- コネクションレス型モード・プログラム
 - ソケット
 - ☐ サーバ
 - ☐ クライアント
 - XTI
 - ☐ サーバ
 - ☐ クライアント
- 共有ファイル

これらのサンプル・プログラムのコピーは `/usr/examples/network_programming` ディレクトリから入手することができます。

B.1 コネクション指向型プログラム

この節では、コネクション指向型モードの通信用に書かれた、ソケット版および XTI 版のサーバおよびクライアントのプログラム例を示します。

B.1.1 ソケット・サーバ・プログラム

例 B-1 はソケット・インタフェースを使用するサーバをインプリメントします。

B-2 ソケットおよび XTI プログラム例

例 B-1: コネクション指向型ソケット・サーバ・プログラム

```
/*
 *
 * This file contains the main socket server code
 * for a connection-oriented mode of communication.
 *
 * Usage:          socketserver
 *
 */

#include "server.h"

char          *parse(char *);
struct transaction *verifycustomer(char *, int, char *);

main(int argc, char *argv[])
{
    int          sockfd;
    int          newsockfd;
    struct sockaddr_in serveraddr;
    struct sockaddr_in clientaddr;
    int          clientaddrlen = sizeof(clientaddr);
    struct hostent *he;
    int          pid;

    signal(SIGCHLD, SIG_IGN);

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) [ 1 ]
    {
        perror("socket_create");
        exit(1);
    }

    bzero((char *) &serveraddr,
          sizeof(struct sockaddr_in)); [ 2 ]
    serveraddr.sin_family = AF_INET; [ 3 ]
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY); [ 4 ]
    serveraddr.sin_port = htons(SERVER_PORT);

    if ( bind(sockfd, [ 5 ]
              (struct sockaddr *)&serveraddr,
              sizeof(struct sockaddr_in)) < 0) {
        perror("socket_bind");
        exit(2);
    }
}
```

例 B-1: コネクション指向型ソケット・サーバ・プログラム (続き)

```
listen(sockfd, 8); 6

while(1) {
    if ((newsockfd =
        accept(sockfd, 7
            (struct sockaddr *) &clientaddr,
            &clientaddrlen)) < 0) {
        if (errno == EINTR) {
            printf("Bye...\n");
            exit(0);
        } else {
            perror("socket_accept");
            exit(3);
        }
    }

    pid = fork();

    switch(pid) {
        case -1:          /* error */
            perror("dosession_fork");
            break;
        default:
            close(newsockfd);
            break;
        case 0:          /* child */

            close(sockfd);
            transactions(newsockfd);

            close(newsockfd);
            return(0);
    }
}

transactions(int fd) 8
{
    int    bytes;
    char    *reply;
    int    dcount;
    char    datapipe[MAXBUFSIZE+1];
```

例 B-1: コネクション指向型ソケット・サーバ・プログラム (続き)

```
/*
 * Look at the data buffer and parse commands,
 * keep track of the collected data through
 * transaction_status.
 */
while (1) {
    if ((dcount=recv(fd, datapipe, MAXBUFSIZE, 0)) < 0) {
        perror("transactions_receive");
        break;
    }
    if (dcount == 0) {
        return(0);
    }

    datapipe[dcount] = '\0';

    if ((reply=parse(datapipe)) != NULL) {
        send(fd, reply, strlen(reply), 0);
    }
}
}
```

- ❶ socket コールでソケットを作成します。

AF_INET はインターネット通信ドメインを指定します。OSI トランスポートがサポートされている場合は AF_INET の代わりに AF_OSI 等の対応する定数が必要となります。ソケット・タイプ SOCK_STREAM は TCP またはコネクション指向型の通信の場合に指定されます。このパラメータはソケットがコネクション指向型であることを示します。

XTI サーバの例 (B.1.3 項) では、socket コールの代わりに t_open コールを使用します。

- ❷ serveraddr はソケットの通信ドメイン (AF_INET) に制御される sockaddr_in 型の構造体です。インターネット通信ドメインのソケット・アドレスにはネットワーク上で固有のインターネット・アドレスおよび 16 ビットのポート番号が含まれます。TCP/IP および UDP/IP では、サーバのインターネット・アドレスとサーバがリッスンしているポート番号になります。

`sockaddr_in` 構造体に含まれる情報は、アドレス・ファミリ (この例では `AF_INET`) に依存する点に注意してください。 `AF_INET` を使用する場合は `bind` コールに `sockaddr_in` が必要とされますが、`AF_INET` の代わりに `AF_OSI` を使用する場合は `sockaddr_osi` が必要になります。

- ③ `INADDR_ANY` はシステムに接続されたインタフェース・アダプタを示します。すべての番号は適当なマクロを使用して、ネットワーク・フォーマットに変換されなければなりません。詳細については、`htonl(3)`、`htons(3)`、`ntohl(3)` および `ntohs(3)` の各リファレンス・ページを参照してください。
- ④ `SERVER_PORT` は `common.h` ヘッダ・ファイルで定義されています。これは `short` 型の整数で、サーバ・プロセスと他のアプリケーション・プロセスの識別に使用します。0 番から 1024 番まではリザーブされています。
- ⑤ `bind` コールでソケットとサーバのアドレスをバインドします。アドレスとポート番号の組み合わせによってネットワーク上で固有のものが識別されます。
- ⑥ 前の `accept` コールの処理を終了するまでにサーバがキュー処理できる保留中の接続の数を指定します。この値はサーバが `accept` コールを処理中の接続の成功レートを制御します。複数のクライアントがサーバに `connect` 要求を送信している場合に成功レートを向上させるには大きい数を使用します。この値はオペレーティング・システムによって上限が決められます。
- ⑦ ソケットへの接続を受け入れます。各接続に対して、サーバはセッションが完成するまでを処理する子プロセスをフォークします。次にサーバは新しい接続要求のリッスンを再開します。この例は並行型サーバのものであるので、データそのものを処理する対話型サーバの例については B.2 節を参照してください。
- ⑧ 着信メッセージ・パケットはサーバに受け入れられた後、カード・オペレータ (merchant) のログインID、パスワードおよびクレジット・カード番号等の情報をトラックする `parse` 関数に渡されます。このプロセスは `parse` 関数がトランザクションの終了を認識し、クライアント・プログラムに送信する応答パケットを返すまで繰り返し実行されます。

クライアント・プログラムは情報パケットを順不同で (および 1 つでも複数でも) 送信することができるので、`parse` 関数は構造化されています。

ないメッセージ・ストリームを処理するために必要なステータス情報を呼び出せる仕様になっています。

このプログラムはコネクション指向型のプロトコルを使用してデータ転送を行なうので、send および recv でメッセージの送受信を行ないます。

9 recv コールでデータを受信します。

10 send コールでデータを送信します。

B.1.2 ソケット・クライアント・プログラム

例 B-2 は例 B-1 に示されている socketserver インタフェースと通信可能なクライアント・プログラムをインプリメントします。

例 B-2: コネクション指向型ソケット・クライアント・プログラム

```
/*
 *
 * This generates the client program.
 *
 * usage: client [serverhostname]
 *
 * If a host name is not specified, the local
 * host is assumed.
 */

#include "client.h"

main(int argc, char *argv[])
{
    int                sockfd;
    struct sockaddr_in serveraddr;
    struct hostent     *he;
    int                n;
    char               *serverhost = "localhost";
    struct hostent     *serverhostp;
    char               buffer[1024];
    char               inbuf[1024];

    if (argc>1) {
        serverhost = argv[1];
    }

    init();
```

例 B-2: コネクション指向型ソケット・クライアント・プログラム (続き)

```
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) 1
{
    perror("socket_create");
    exit(1);
}

bzero((char *) &serveraddr,
      sizeof(struct sockaddr_in)); 2
serveraddr.sin_family = AF_INET;

if ((serverhostp = gethostbyname(serverhost)) == 3
    (struct hostent *)NULL) {
    fprintf(stderr, "gethostbyname on %s failed\n",
            serverhost);
    exit(1);
}
bcopy(serverhostp->h_addr,
      (char *)&(serveraddr.sin_addr.s_addr),
      serverhostp->h_length);

serveraddr.sin_port = htons(SERVER_PORT); 4

/* Now connect to the server */
if (connect(sockfd, (struct sockaddr *)&serveraddr, 5
    sizeof(serveraddr)) < 0) {
    perror("connect");
    exit(2);
}

while(1) {
    /* Merchant record */
    sprintf(buffer, "%%%ms###%%p%s##",
            merchantname, password);

    printf("\n\nSwipe card, enter amount: ");
    fflush(stdout);
    if (scanf("%s", inbuf) == EOF) {
        printf("bye...\n");
        exit(0);
    }
    soundbytes();

    sprintf(buffer, "%s%%a%s###%%n%s##",
            buffer, inbuf, swipecard());
```

例 B-2: コネクション指向型ソケット・クライアント・プログラム (続き)

```
        if (send(sockfd, buffer, strlen(buffer), 0) < 0) {
            perror("send");
            exit(1);
        }

        /* receive info */
        if ((n = recv(sockfd, buffer, 1024, 0)) < 0) {
            perror("recv");
            exit(1);
        }
        buffer[n] = '\0';

        if ((n=analyze(buffer))== 0) {
            printf("transaction failure,"
                  " try again\n");
        } else if (n<0) {
            printf("login failed, try again\n");
            init();
        }
    }
}
```

- ❶ socket コールでソケットを作成します。

AF_INET はインターネット通信ドメインのソケット・タイプです。このパラメータは対応するサーバ・プログラムで選択されたプロトコルおよびタイプと一致する必要がある点に注意してください。

XTI クライアントの例 (B.1.4 項) では、socket コールの代わりに t_open コールを使用します。

- ❷ serveraddr はソケットの通信ドメイン (AF_INET) に制御される sockaddr_in 型の構造体です。インターネット通信ドメインのソケット・アドレスにはネットワーク上で固有のインターネット・アドレスおよび 16 ビットのポート番号が含まれます。TCP/IP プロトコル群では、サーバのインターネット・アドレスとサーバがリスンしているポート番号になります。

sockaddr_in 構造体に含まれる情報は、アドレス・ファミリ (またはプロトコル) に依存する点に注意してください。

- ③ プロトコルまたはアドレス・ファミリに依存するサーバについての情報を取得します。gethostbyname ルーチンを使用してサーバの IP アドレスを知ることができます。
- ④ SERVER_PORT は <common.h> ヘッダ・ファイルで定義されています。ソケット・サーバ・プログラムに接続する際には必ず同じポート番号を使用しなければなりません。サーバおよびクライアントは通信によく使用されるアドレスとして機能するポート番号を選択します。
- ⑤ クライアントはサーバに接続するために、connect コールを実行します。connect コールがコネクション指向型プロトコルで使用される場合、クライアントはデータを送信する前にサーバとの接続を構築することができます。
- ⑥ send コールでデータを送信します。
- ⑦ recv コールでデータを受信します。

B.1.3 XTI サーバ・プログラム

例 B-3 はネットワーク通信に XTI ライブラリを使用するサーバをインプリメントします。この例は、トランスポートを独立させて処理を行なう通信プログラムに相対する仕様のものです。次のプログラムと B.1.1 項のソケット・サーバ・プログラムを比較してください。このプログラムにはこの付録の最初に記述した制限事項が適応されます。

例 B-3: コネクション指向型 XTI サーバ・プログラム

```
/*
 *
 *
 * This file contains the main XTI server code
 * for a connection-oriented mode of communication.
 *
 * Usage:          xtiserver
 *
 */
#include "server.h"

char                *parse(char *);
struct transaction  *verifycustomer(char *, int, char *);

main(int argc, char *argv[])
{
    int                xtifd;
```

例 B-3: コネクション指向型 XTI サーバ・プログラム (続き)

```
int                newxtifd;
struct sockaddr_in serveraddr;
struct hostent     *he;
int                pid;
struct t_bind      *bindreqp;
struct t_call      *call;

signal(SIGCHLD, SIG_IGN);

if ((xtifd = t_open("/dev/streams/xtiso/tcp+", O_RDWR, 1
                    NULL)) < 0) {
    xerror("xti_open", xtifd);
    exit(1);
}

bzero((char *) &serveraddr, sizeof(struct sockaddr_in));
serveraddr.sin_family    = AF_INET;                2
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);    3
serveraddr.sin_port      = htons(SERVER_PORT);     4

/* allocate structures for the t_bind and t_listen call */
if ((bindreqp=(struct t_bind *)
    t_alloc(xtifd, T_BIND, T_ALL))
    == NULL) ||
    ((call=(struct t_call *)
    t_alloc(xtifd, T_CALL, T_ALL))
    == NULL)) {
    xerror("xti_alloc", xtifd);
    exit(3);
}

bindreqp->addr.buf      = (char *)&serveraddr;
bindreqp->addr.len      = sizeof(serveraddr);

/*
 * Specify how many pending connections can be
 * maintained, until finish accept processing
 */
bindreqp->qlen          = 8;                        5

if (t_bind(xtifd, bindreqp, (struct t_bind *)NULL) 6
    < 0) {
```

例 B-3: コネクション指向型 XTI サーバ・プログラム (続き)

```
xerror("xti_bind", xtifd);
exit(4);
}

/*
 * Now the socket is ready to accept connections.
 * For each connection, fork a child process in charge
 * of the session, and then resume accepting connections.
 */
while(1) {

    if (t_listen(xtifd, &call) < 0) {
        if (errno == EINTR) {
            printf("Bye...\n");
            exit(0);
        } else {
            xerror("t_listen", xtifd);
            exit(4);
        }
    }

    /*
     * Create a new transport endpoint on which
     * to accept a connection
     */
    if ((newxtifd=t_open("/dev/streams/xtiso/tcp+",
                        O_RDWR, NULL)) < 0) {
        xerror("xti_newopen", xtifd);
        exit(5);
    }

    /* accept connection */
    if (t_accept(xtifd, newxtifd, call) < 0) {
        xerror("xti_accept", xtifd);
        exit(7);
    }
    pid = fork();

    switch(pid) {
        case -1:
            /* error */
            xerror("dosession_fork", xtifd);
            break;
        default:
            t_close(newxtifd);
    }
}
```

例 B-3: コネクション指向型 XTI サーバ・プログラム (続き)

```
                break;
case 0:          /* child */

    t_close(xtiffd);
    transactions(newxtiffd);
    if ((t_free((char *)bindreqp,
                T_BIND) < 0) ||
        (t_free((char *)call,
                T_CALL) < 0)) {
        xerror("xti_free", xtiffd);
        exit(3);
    }

    t_close(newxtiffd);
    return(0);
    }
}
}
```

```
transactions(int fd) 10
{
    int      bytes;
    char     *reply;
    int      dcount;
    int      flags;
    char     datapipe[MAXBUFSIZE+1];

    /*
     * Look at the data buffer and parse commands, if more data
     * required go get it
     * Since the protocol is SOCK_STREAM oriented, no data
     * boundaries will be preserved.
     */
    while (1) {
        if ((dcount=t_rcv(fd, datapipe, MAXBUFSIZE, 11
                        &flags)) < 0){
            /* if disconnected bid a goodbye */
            if (t_errno == TLOOK) {
                int tmp = t_look(fd);

                if (tmp != T_DISCONNECT) {
                    t_scope(tmp);
                } else {
```

例 B-3: コネクション指向型 XTI サーバ・プログラム (続き)

```
                                exit(0);
                                }
                                }
                                xerror("transactions_receive", fd);
                                break;
                                }
                                if (dcount == 0) {
                                    /* consolidate all transactions */
                                    return(0);
                                }

                                datapipe[dcount] = ' ';

                                if ((reply=parse(datapipe)) != NULL) {
                                    if (t_snd(fd, reply, strlen(reply), 0) 12
                                        < 0) {
                                        xerror("xti_send", fd);
                                        break;
                                    }
                                }
                                }
                                }
```

- ❶ t_open コールは、たとえば /dev/streams/xtiso/tcp+ 等のデバイス特殊ファイルを指定します。このファイル名により IP 上の TCP トラnsポート・プロトコルに必要な抽象化が提供されます。ソケット・インタフェースと異なりアドレス・ファミリを指定する部分 (たとえば AF_INET) では、デバイス特殊ファイルの選択によって既にアドレス・ファミリの情報が取得されています。/dev/streams/xtiso/tcp+ ファイルは TCP トラnsポートと IP を示します。STREAMS デバイスの詳細については第 5 章を参照してください。

B.1.1 項で記述されているとおり、OSI トラnsポートが使用できる場合は /dev/streams/xtiso/cots などのデバイスを使用することが出来ます。

B.1.1 項で使用されている socket コールの代わりに、ここでは t_open コールを使用します。

- ❷ アドレスの選択はトラnsポート・プロトコルの選択に依存します。ソケットの例では socket システム・コールで 사용되는ものと同じ

アドレス・ファミリを使用しましたが、XTI ではアドレスの選択がはっきりしていないため、あらかじめトランスポート・プロトコルから `sockaddr` への適切なマッピングを確認しておく必要があります。詳細については第 3 章を参照してください。

- ③ `INADDR_ANY` はシステムに接続されたインタフェース・アダプタを示します。すべての番号は適当なマクロを使用して、ネットワーク・フォーマットに変換されなければなりません。詳細については、`htonl(3)`、`htons(3)`、`ntohl(3)` および `ntohs(3)` の各リファレンス・ページを参照してください。
- ④ `SERVER_PORT` は `<common.h>` ヘッダ・ファイルで定義されています。これは `short` 型の整数で、サーバ・プロセスと他のアプリケーション・プロセスの識別に使用します。0 番から 1024 番まではリザーブされています。
- ⑤ サーバが最後の要求を処理している際にキュー処理できる、保留中の接続の数を指定します。
- ⑥ `t_bind` コールでサーバのアドレスをバインドします。アドレスとポート番号の組み合わせでネットワーク上で固有のものを識別します。サーバ・プロセスのアドレスがバインドされた後、サーバ・プロセスはシステムに登録されるので、低レベルのカーネル関数はこれを認識し直接要求を出すことができます。
- ⑦ `t_listen` 関数で接続要求をリッスンをします。
- ⑧ 他のコールから `t_open` 関数への新しいトランスポートのエンドポイントを作成します。
- ⑨ `t_accept` 関数で、接続要求を受け入れます。
- ⑩ 各着信メッセージ・パケットはサーバに受け入れられた後、カード・オペレータ (merchant) のログイン ID、パスワードおよびクレジット・カード番号等の情報をトラックする `parse` 関数に渡されます。このプロセスは `parse` 関数がトランザクションの終了を認識し、クライアント・プログラムに送信する応答パケットを返すまで繰り返し実行されます。

クライアント・プログラムは情報パケットを順不同で (および 1 つでも複数でも) 送信することができるので、`parse` 関数は構造化化されていないメッセージ・ストリームを処理するために必要なステータス情報を呼び出せる仕様になっています。

このプログラムはコネクション指向型のプロトコルを使用してデータ転送をしているので、`t_snd` および `t_rcv` を使用してそれぞれメッセージの送受信を行ないます。

11 `t_rcv` 関数でデータを受信します。

12 `t_snd` 関数でデータを送信します。

B.1.4 XTI クライアント・プログラム

例 B-4 は B.1.3 項 に示されている `xtiserver` インタフェースと通信可能なクライアント・プログラムをインプリメントします。例 B-3 に示されているソケット・クライアント・プログラムと比較してください。

例 B-4: コネクション指向型 XTI クライアント・プログラム

```
/*
 *
 * This file contains the main XTI client code
 * for a connection-oriented mode of communication.
 *
 * Usage: xticlient [serverhostname]
 *
 * If a host name is not specified, the local
 * host is assumed.
 */

#include "client.h"

main(int argc, char *argv[])
{
    int                xtifd;
    struct sockaddr_in serveraddr;
    int                n;
    char                *serverhost = "localhost";
    struct hostent      *serverhostp;
    char                buffer[1024];
    char                inbuf[1024];
    struct t_call        *sndcall;
    int                flags = 0;

    if (argc>1) {
        serverhost = argv[1];
    }

    init();
```

例 B-4: コネクション指向型 XTI クライアント・プログラム (続き)

```
if ((xtifd = t_open("/dev/streams/xtiso/tcp+", O_RDWR, 1)
        NULL)) < 0) {
    xerror("xti_open", xtifd);
    exit(1);
}

bzero((char *) &serveraddr, 2
        sizeof(struct sockaddr_in));
serveraddr.sin_family = AF_INET; 3
if ((serverhostp = gethostbyname(serverhost)) == 4
        (struct hostent *)NULL) {
    fprintf(stderr, "gethostbyname on %s failed\n",
            serverhost);
    exit(1);
}
bcopy(serverhostp->h_addr,
        (char *)&(serveraddr.sin_addr.s_addr),
        serverhostp->h_length);
serveraddr.sin_port = htons(SERVER_PORT); 5

if (t_bind(xtifd, (struct t_bind *)NULL, 6
        (struct t_bind *)NULL) < 0) {
    xerror("bind", xtifd);
    exit(2);
}

/* Allocate structures for the t_bind and t_listen calls */
if ((sndcall=(struct t_call *)t_alloc(xtifd, T_CALL,
        T_ALL)) == NULL) {
    xerror("xti_alloc", xtifd);
    exit(3);
}

sndcall.opt.maxlen = 0;
sndcall.udata.maxlen = 0;
sndcall.addr.buf = (char *)&serveraddr;
sndcall.addr.len = sizeof(serveraddr);

if (t_connect(xtifd, sndcall, 7
        (struct t_call *)NULL) < 0) {
    xerror ("t_connect", xtifd);
    exit(3);
}
```

例 B-4: コネクション指向型 XTI クライアント・プログラム (続き)

```
while(1) {
    /* Merchant record */
    sprintf(buffer, "%%%ms###%%p%s##",
        merchantname, password);

    printf("\n\nSwipe card, enter amount: ");
    fflush(stdout);
    if (scanf("%s", inbuf) == EOF) {
        printf("bye...\n");
        exit(0);
    }
    soundbytes();

    sprintf(buffer, "s%%%a%s###%%n%s##",
        buffer, inbuf, swipecard());

    if (t_snd(xtifd, buffer, strlen(buffer), 0) < 0) { 8
        xerror("t_snd", xtifd);
        exit(1);
    }

    if ((n = t_rcv(xtifd, buffer, 1024, &flags)) < 0) { 9
        xerror("t_rcv", xtifd);
        exit(1);
    }

    buffer[n] = '\0';

    if ((n=analyze(buffer))== 0) {
        printf("transaction failure,"
            " try again\n");
    } else if (n<0) {
        printf("login failed, try again\n");
        init();
    }
}

if (t_free((char *)sndcall, T_CALL) < 0) {
    xerror("xti_free", xtifd);
    exit(3);
}
```

例 B-4: コネクション指向型 XTI クライアント・プログラム (続き)

}

- ❶ AF_INET はインターネット通信ドメインのソケット・タイプです。AF_OSI がサポートされている場合には、AF_INET は OSI コミュニケーションのソケットの作成に使用されることもあります。ソケット・タイプ SOCK_STREAM は TCP またはコネクション指向型コミュニケーションに指定されます。

t_open コールは socket コールが必要とするソケット・アドレス・ファミリ、ソケット・タイプおよびプロトコルの代わりにデバイス特殊ファイル名を指定します。

B.1.2 項の socket コールの代わりに t_open コールを使用します。

- ❷ serveraddr はソケットの通信ドメイン (AF_INET) に制御される sockaddr_in 型の構造体です。インターネット通信ドメインのソケット・アドレスにはネットワーク上で固有のインターネット・アドレスおよび 16 ビットのポート番号が含まれます。UDP を含む TCP/IP プロトコル群では、サーバのインターネット・アドレスとサーバがリスンしているポート番号になります。

sockaddr_in 構造体に含まれる情報はアドレス・ファミリ (またはプロトコル) に依存する点に注意してください。

- ❸ AF_INET はインターネット通信ドメインを指定します。AF_OSI がサポートされている場合は OSI 通信用のソケットの作成に使用することもできます。
- ❹ プロトコルまたはアドレス・ファミリに依存するサーバについての情報を取得します。gethostbyname ルーチンを使用してサーバの IP アドレスを知ることができます。
- ❺ SERVER_PORT は <common.h> ヘッダ・ファイルで定義されています。XTI サーバ・プログラムに接続する際には必ず同じポート番号を使用しなければなりません。0 番から 1024 番まではリザーブされています。
- ❻ t_bind 関数でサーバ・アドレスとバインドし、クライアントがデータの送受信を開始できるようにします。

- ⑦ `t_connect` 関数を使用して、サーバとの接続を制御します。
- ⑧ `t_snd` 関数でデータを送信します。
- ⑨ `t_rcv` 関数でデータを受信します。

B.2 コネクションレス型プログラム

この節では、コネクションレス型モードの通信用に書かれた、ソケット版および XTI 版のサーバおよびクライアントのプログラム例を示します。

B.2.1 ソケット・サーバ・プログラム

例 B-5 は、B.1.1 項の通信にコネクション指向型パラダイムを使用したソケット・サーバと同じ方法で、クライアント・プログラムとの通信にコネクションレス型 (データグラム/UDP) パラダイムを使用したサーバをインプリメントします。この付録の冒頭に記述されている制限事項が適応されます。

例 B-5: コネクションレス型ソケット・サーバ・プログラム

```
/*
 *
 * This file contains the main socket server code
 * for a connectionless mode of communication.
 *
 * Usage:          socketserverDG
 *
 */
#include "server.h"

char          *parse(char *);
struct transaction *verifycustomer(char *, int, char *);

main(int argc, char *argv[])
{
    int          sockfd;
    int          newsockfd;
    struct sockaddr_in serveraddr;
    int          serveraddrlen = sizeof(serveraddr);
    struct sockaddr_in clientaddr;
    int          clientaddrlen = sizeof(clientaddr);
    struct hostent *he;
    int          pid;

    signal(SIGCHLD, SIG_IGN);
```

例 B-5: コネクションレス型ソケット・サーバ・プログラム (続き)

```
/* Create a socket for the communications */
if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0))
    < 0) {
    perror("socket_create");
    exit(1);
}

bzero((char *) &serveraddr,
      sizeof(struct sockaddr_in));
serveraddr.sin_family      = AF_INET;
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
serveraddr.sin_port        = htons(SERVER_PORT);

if ( bind(sockfd,
          (struct sockaddr *)&serveraddr,
          sizeof(struct sockaddr_in)) < 0) {
    perror("socket_bind");
    exit(2);
}

transactions(sockfd);
}

transactions(int fd)
{
    int          bytes;
    char         *reply;
    int          dcount;
    char         datapipe[MAXBUFSIZE+1];
    struct sockaddr_in serveraddr;
    int          serveraddrlen = sizeof(serveraddr);

    bzero((char *) &serveraddr, sizeof(struct sockaddr_in));
    serveraddr.sin_family      = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port        = htons(SERVER_PORT);

    /* Look at the data buffer and parse commands.
     * Keep track of the collected data through
     * transaction_status.
     */
    while (1) {
```

例 B-5: コネクションレス型ソケット・サーバ・プログラム (続き)

```
        if ((dcount=recvfrom(fd, datapipe,                                7
                               MAXBUFSIZE, 0,
                               (struct sockaddr *)&serveraddr,
                               &serveraddrlen)) < 0){
            perror("transactions_receive");
            break;
        }
        if (dcount == 0) {
            return(0);
        }

        datapipe[dcount] = '\0';

        if ((reply=parse(datapipe)) != NULL) {
            if (sendto(fd, reply, strlen(reply), 8
                       0,
                       (struct sockaddr *)&serveraddr,
                       serveraddrlen) < 0) {
                perror("transactions_sendto");
            }
        }
    }
}
```

- ❶ `socket` コールでソケットを作成します。

`AF_INET` はインターネット通信ドメインを指定します。ソケットタイプ `SOCK_DGRAM` は UDP またはコネクションレス型の通信の場合に指定されます。このパラメータはプログラムがコネクションレス型であることを示します。

XTI サーバの例 (B.2.3 項) では、`socket` コールの代わりに `t_open` コールを使用します。

- ❷ `serveraddr` はソケットの通信ドメイン (`AF_INET`) に制御される `sockaddr_in` 型の構造体です。インターネット通信ドメインのソケット・アドレスにはネットワーク上で固有のインターネット・アドレスおよび 16 ビットのポート番号が含まれます。UDP を含む TCP/IP プロトコル群では、サーバのインターネット・アドレスとサーバがリスンしているポート番号になります。

`sockaddr_in` 構造体に含まれる情報は、この例では `AF_INET` にあたるアドレス・ファミリに依存する点に注意してください。 `AF_INET` を使用する場合は `bind` コールに `sockaddr_in` が必要とされますが、 `AF_INET` の代わりに `AF_OSI` を使用する場合は `sockaddr_osi` が必要になります。

- ③ `INADDR_ANY` はシステムに接続されたインタフェース・アダプタを示します。すべての番号は適当なマクロを使用して、ネットワーク・フォーマットに変換されなければなりません。詳細については、 `htonl(3)`、`htons(3)`、`ntohl(3)` および `ntohs(3)` の各リファレンス・ページを参照してください。
- ④ `SERVER_PORT` は `<common.h>` ヘッダ・ファイルで定義されています。これは `short` 型の整数で、サーバ・プロセスと他のアプリケーション・プロセスの識別に使用します。
- ⑤ `bind` コールでソケットとサーバのアドレスをバインドします。アドレスとポート番号の組み合わせでネットワーク上で固有のものを識別します。
サーバ・プロセスのアドレスがバインドされた後、サーバ・プロセスはシステムに登録されるので、低レベルのカーネル関数はこれを認識し直接要求を出すことができます。
- ⑥ 各着信メッセージ・パケットはサーバに受け入れられた後、カード・オペレータ (merchant) のログイン ID、パスワードおよびクレジット・カード番号等の情報をトラックする `parse` 関数に渡されます。このプロセスは `parse` 関数がトランザクションの終了を認識し、クライアント・プログラムに送信する応答パケットを返すまで繰り返し実行されます。
このプログラムはコネクションレス型の (データグラム) プロトコルを使用してデータ転送をしているので、`sendto` および `recvfrom` を使用してそれぞれメッセージの送受信を行ないます。
- ⑦ `recvfrom` コールでデータを受信します。
- ⑧ `sendto` コールでデータを送信します。

B.2.2 ソケット・クライアント・プログラム

例 B-6 は例 B-5 に示されているソケット・サーバと通信可能なソケット・クライアント・プログラムをインプリメントします。コネクションレス型モードまたはデータグラム・モードでソケット・インタフェースを使用します。

例 B-6: コネクションレス型ソケット・クライアント・プログラム

```
/*
 *
 * This file contains the main client socket code
 * for a connectionless mode of communication.
 *
 * usage: socketclientDG [serverhostname]
 *
 * If a host name is not specified, the local
 * host is assumed.
 */
#include "client.h"

main(int argc, char *argv[])
{
    int                sockfd;
    struct sockaddr_in serveraddr;
    int                serveraddrlen;
    struct hostent     *he;
    int                n;
    char               *serverhost = "localhost";
    struct hostent     *serverhostp;
    char               buffer[1024];
    char               inbuf[1024];

    if (argc>1) {
        serverhost = argv[1];
    }

    init();

    /* Create a socket for the communications */
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) [1]
    {
        perror("socket_create");
        exit(1);
    }

    bzero((char *) &serveraddr, [2]
        sizeof(struct sockaddr_in));
    serveraddr.sin_family = AF_INET;

    if ((serverhostp = gethostbyname(serverhost)) == [3]
        (struct hostent *)NULL) {
        fprintf(stderr, "gethostbyname on %s failed\n",
```

例 B-6: コネクションレス型ソケット・クライアント・プログラム (続き)

```
        serverhost);
    exit(1);
}
bcopy(serverhostp->h_addr,
      (char *)&(serveraddr.sin_addr.s_addr),
      serverhostp->h_length);

serveraddr.sin_port      = htons(SERVER_PORT);      [ 4 ]

/* Now connect to the server
if (connect(sockfd, (struct sockaddr *)&serveraddr, [ 5 ]
        sizeof(serveraddr)) < 0) {
    perror ("connect");
    exit(2);
}
*/

while(1) {
    /* Merchant record */
    sprintf(buffer, "%%%m%s###%%p%s##",
            merchantname, password);

    printf("\n\nSwipe card, enter amount: ");
    fflush(stdout);
    if (scanf("%s", inbuf) == EOF) {
        printf("bye...\n");
        exit(0);
    }
    soundbytes();

    sprintf(buffer, "%s%%a%s###%%n%s##",
            buffer, inbuf, swipecard());

    if (sendto(sockfd, buffer, strlen(buffer), 0, [ 6 ]
            (struct sockaddr *)&serveraddr,
            sizeof(serveraddr)) < 0) {
        perror("sendto");
        exit(1);
    }

    /* receive info */
    if ((n = recvfrom(sockfd, buffer, 1024, 0, [ 7 ]
            (struct sockaddr *)&serveraddr,
            &serveraddrlen)) < 0) {
        perror("recvfrom");
        exit(1);
    }
}
```

例 B-6: コネクションレス型ソケット・クライアント・プログラム (続き)

```
    }
    buffer[n] = '\0';

    if ((n=analyze(buffer))== 0) {
        printf("transaction failure, "
               "try again\n");
    } else if (n<0) {
        printf("login failed, try again\n");
        init();
    }
}

}
```

- 1 socket コールでソケットを作成します。

AF_INET はインターネット通信ドメインを指定します。AF_OSI がサポートされている場合には、AF_INET は OSI コミュニケーションのソケットの作成に使用されることもあります。ソケットタイプ SOCK_DGRAM は UDP またはコネクションレス型の通信の場合に指定されます。

XTI クライアント例 (B.2.4 項) では socket コールの代わりに t_open コールを使用します。

- 2 serveraddr はソケットの通信ドメイン (AF_INET) に制御される sockaddr_in 型の構造体です。インターネット通信ドメインのソケット・アドレスにはネットワーク上で固有のインターネット・アドレスおよび 16 ビットのポート番号が含まれます。UDP を含む TCP/IP プロトコル群では、サーバのインターネット・アドレスとサーバがリッスンしているポート番号になります。

sockaddr_in 構造体に含まれる情報はアドレス・ファミリ (またはプロトコル) に依存する点に注意してください。

- 3 プロトコルまたはアドレス・ファミリに依存するサーバについての情報を取得します。gethostbyname ルーチンを使用してサーバの IP アドレスを知ることができます。

- ④ SERVER_PORT は <common.h> ヘッダ・ファイルで定義されています。これは short 型の整数で、サーバ・プロセスと他のアプリケーション・プロセスの識別に使用します。
- ⑤ クライアントはサーバに接続するために connect コールを実行します。connect コールがコネクションレス型プロトコルで 사용되는場合、クライアントはサーバのアドレスをローカルに格納することができます。これにより、クライアントはメッセージを送信する毎にサーバのアドレスを指定する必要がなくなります。
- ⑥ sendto コールでデータを送信します。
- ⑦ recvfrom コールでデータを受信します。

B.2.3 XTI サーバ・プログラム

例 B-7 はネットワーク通信に XTI ライブラリを使用するサーバをインプリメントします。この例は、トランスポートを独立させて処理を行なう通信プログラムに相對する仕様のものです。次のプログラムと例 B-5 のソケット・サーバ・プログラムを比較してください。このプログラムにはこの付録の最初に記述した制限事項が適應されます。

例 B-7: コネクションレス型 XTI サーバ・プログラム

```
/*
 *
 * This file contains the main XTI server code
 * for a connectionless mode of communication.
 *
 * Usage:          xtiserverDG
 *
 */
#include "server.h"

char          *parse(char *);
struct transaction *verifycustomer(char *, int, char *);

main(int argc, char *argv[])
{
    int          xtifd;
    int          newxtifd;
    struct sockaddr_in serveraddr;
    struct hostent *he;
    int          pid;
    struct t_bind *bindreqp;
```

例 B-7: コネクションレス型 XTI サーバ・プログラム (続き)

```
signal(SIGCHLD, SIG_IGN);

/* Create a transport endpoint for the communications */
if ((xtifd = t_open("/dev/streams/xtiso/udp+", 1
                  O_RDWR, NULL)) < 0) {
    xerror("xti_open", xtifd);
    exit(1);
}

bzero((char *) &serveraddr, 2
      sizeof(struct sockaddr_in));
serveraddr.sin_family = AF_INET; 3
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY); 4
serveraddr.sin_port = htons(SERVER_PORT) 5

/* allocate structures for the t_bind call */
if ((bindreqp=(struct t_bind *)t_alloc(xtifd,
                                       T_BIND,
                                       T_ALL))
    == NULL) {
    xerror("xti_alloc", xtifd);
    exit(3);
}

bindreqp->addr.buf = (char *)&serveraddr;
bindreqp->addr.len = sizeof(serveraddr);

/*
 * Specify how many pending connections can be
 * maintained, while we finish "accept" processing
 */
bindreqp->qlen = 8; 6

if (t_bind(xtifd, bindreqp, (struct t_bind *)NULL) 7
    < 0) {
    xerror("xti_bind", xtifd);
    exit(4);
}

/*
 * Now the server is ready to accept connections
 * on this socket. For each connection, fork a child
```

例 B-7: コネクションレス型 XTI サーバ・プログラム (続き)

```
    * process in charge of the session, and then resume
    * accepting connections.
    *
    */
transactions(xtifd);

if (t_free((char *)bindreqp, T_BIND) < 0) {
    xerror("xti_free", xtifd);
    exit(3);
}
}

transactions(int fd)
{
    int                bytes;
    char               *reply;
    int                dcount;
    int                flags;
    char               datapipe[MAXBUFSIZE+1];
    struct t_unitdata  *unitdata;
    struct sockaddr_in clientaddr;

    /* Allocate structures for t_rcvudata and t_sndudata call */
    if ((unitdata=(struct t_unitdata *)t_alloc(fd,
                                                T_UNITDATA,
                                                T_ALL))
        == NULL) {
        xerror("xti_alloc", fd);
        exit(3);
    }

    /*
    * Look at the data buffer and parse commands.
    * If more data required, go get it.
    */
    while (1) {
        unitdata->udata.maxlen = MAXBUFSIZE;
        unitdata->udata.buf    = datapipe;
        unitdata->addr.maxlen  = sizeof(clientaddr);
        unitdata->addr.buf     = (char *)&clientaddr;
        unitdata->opt.maxlen   = 0;

        if ((dcount=t_rcvudata(fd, &unitdata, &flags))
            < 0) {
            /* if disconnected bid a goodbye */

```

8

9

例 B-7: コネクションレス型 XTI サーバ・プログラム (続き)

```
        if (t_errno == TLOOK) {
            int tmp = t_look(fd);

            if (tmp != T_DISCONNECT) {
                t_scope(tmp);
            } else {
                exit(0);
            }
        }
        xerror("transactions_receive", fd);
        break;
    }
    if (unitdata->udata.len == 0) {

        return(0);
    }

    datapipe[unitdata->udata.len] = '\0';

    if ((reply=parse(datapipe)) != NULL) {

        /* sender's addr is in the unitdata */
        unitdata->udata.len = strlen(reply);
        unitdata->udata.buf = reply;

        if (t_sndudata(fd, unitdata) < 0) { 10
            xerror("xti_send", fd);
            break;
        }
    }
}
if (t_free((char *)unitdata, T_UNITDATA) < 0) {
    xerror("xti_free", fd);
    exit(3);
}
}
```

- 1 t_open コールは、この例では /dev/streams/xtiso/udp+ にあたるデバイス特殊ファイルを指定します。このファイル名により IP 上の UDP トランスポート・プロトコルに必要な抽象化が提供されます。ソケット・インタフェースと異なりアドレス・ファミリを指定する部分 (たとえば AF_INET) では、デバイス特殊ファイルの選択によって既にアドレス・ファミリの情報が取得されています。 /dev/streams/xtiso/udp+

ファイルはUDP トランスポートとインターネット・プロトコルを示します。STREAMS デバイスの詳細については第 5 章を参照してください。

B.2.1 項で使用されている `socket` コールの代わりに、ここでは `t_open` コールを使用します。

- ② `serveraddr` はソケットの通信ドメインまたはアドレス・ファミリ (`AF_INET`) に制御される `sockaddr_in` 型の構造体です。インターネット通信ドメインのソケット・アドレスにはネットワーク上で固有のインターネット・アドレスおよび 16 ビットのポート番号が含まれます。TCP/IP および UDP/IP では、サーバのインターネット・アドレスとサーバがリスンしているポート番号になります。

`sockaddr_in` 構造体に含まれる情報は、アドレス・ファミリ (またはプロトコル) に依存する点に注意してください。

- ③ `AF_INET` はインターネット通信ドメインまたはアドレス・ファミリを指定します。
- ④ `INADDR_ANY` はシステムに接続されたインタフェース・アダプタを示します。すべての番号は適当なマクロを使用して、ネットワーク・フォーマットに変換されなければなりません。詳細については、`htonl(3)`、`htons(3)`、`ntohl(3)` および `ntohs(3)` の各リファレンス・ページを参照してください。
- ⑤ `SERVER_PORT` は `<common.h>` ヘッダ・ファイルで定義されています。これは `short` 型の整数で、サーバ・プロセスと他のアプリケーション・プロセスの識別に使用します。0 番から 1024 番まではリザーブされています。
- ⑥ サーバが最後の要求を処理している間サーバがキュー処理できる保留中の接続の数を指定します。
- ⑦ `t_bind` コールでサーバのアドレスをバインドします。アドレスとポート番号の組み合わせでネットワーク上で固有のものを識別します。サーバ・プロセスのアドレスがバインドされた後、サーバ・プロセスはシステムに登録されるので、低レベルのカーネル関数はこれを認識し直接要求を出すことができます。
- ⑧ 各着信メッセージ・パケットはサーバに受け入れられた後、カード・オペレータ (merchant) のログイン ID、パスワードおよびクレジット・カード番号等の情報をトラックする `parse` 関数に渡されます。このプロ

セスは `parse` 関数がトランザクションの終了を認識し、クライアント・プログラムに送信する応答パケットを返すまで繰り返し実行されます。

クライアント・プログラムは情報パケットを順不同で (および 1 つでも複数でも) 送信することができるので、`parse` 関数は構造化されていないメッセージ・ストリームを処理するために必要なステータス情報を呼び出せる仕様になっています。

このプログラムはコネクションレス型 (データグラム) のプロトコルを使用してデータ転送をしているので、`t_sndudata` および `t_rcvudata` を使用してそれぞれメッセージの送受信を行ないます。

9 `t_rcvudata` 関数でデータを受信します。

10 `t_sndudata` 関数でデータを送信します。

B.2.4 XTI クライアント・プログラム

例 B-8 は例 B-7 に示されている XTI サーバと通信可能なクライアント・プログラムをインプリメントします。このプログラムはコネクションレス型モードまたはデータグラム・モードで XTI インタフェースを使用します。

例 B-8: コネクションレス型 XTI クライアント・プログラム

```
/*
 *
 * This file contains the main XTI client code
 * for a connectionless mode of communication.
 *
 * usage: client [serverhostname]
 *
 */
#include "client.h"

main(int argc, char*argv[])
{
    int                xtifd;
    struct sockaddr_in serveraddr;
    struct hostent     *he;
    int                n;
    char               *serverhost = "localhost";
    struct hostent     *serverhostp;
    char               buffer[MAXBUFSIZE+1];
    char               inbuf [MAXBUFSIZE+1];
    struct t_unitdata  *unitdata;
    int                flags = 0;
```

例 B-8: コネクションレス型 XTI クライアント・プログラム (続き)

```
if (argc>1) {
    serverhost = argv[1];
}

init();

if ((xtifd = t_open("/dev/streams/xtiso/udp+",
    O_RDWR, NULL)) < 0) { 1
    xerror("xti_open", xtifd);
    exit(1);
}

bzero((char *) &serveraddr, 2
    sizeof(struct sockaddr_in));
serveraddr.sin_family = AF_INET; 3

if ((serverhostp = gethostbyname(serverhost)) == 4
    (struct hostent *)NULL) {
    fprintf(stderr, "gethostbyname on %s failed\n",
        serverhost);
    exit(1);
}
bcopy(serverhostp->h_addr,
    (char *)&(serveraddr.sin_addr.s_addr),
    serverhostp->h_length);
/*
 * SERVER_PORT is a short which identifies
 * the server process from other sources.
 */
serveraddr.sin_port = htons(SERVER_PORT); 5

if (t_bind(xtifd, (struct t_bind *)NULL, 6
    (struct t_bind *)NULL) < 0) {
    xerror("bind", xtifd);
    exit(2);
}

/* Allocate structures for t_rcvudata and t_sndudata call */
if ((unitdata=(struct t_unitdata *)t_alloc(xtifd,
    T_UNITDATA,
    T_ALL))
    == NULL) {
    xerror("xti_alloc", fd);
}
```

例 B-8: コネクションレス型 XTI クライアント・プログラム (続き)

```
        exit(3);
    }

    while(1) {
        /* Merchant record */
        sprintf(buffer, "%%%ms###%%p%s##",
            merchantname, password);

        printf("\n\nSwipe card, enter amount: ");
        fflush(stdout);

        if (scanf("%s", inbuf) == EOF) {
            printf("bye...\n");
            exit(0);
        }

        soundbytes();

        sprintf(buffer, "%s%%a%s###%n%s##",
            buffer, inbuf, swipecard());

        unitdata->addr.buf      = (char *)&serveraddr;
        unitdata->addr.len      = sizeof(serveraddr);
        unitdata->udata.buf     = buffer;
        unitdata->udata.len     = strlen(buffer);
        unitdata->opt.len       = 0;

        if (t_sndudata(xtifd, unitdata) < 0) { 7
            xerror("t_snd", xtifd);
            exit(1);
        }

        unitdata->udata.maxlen  = MAXBUFSIZE;
        unitdata->addr.maxlen   = sizeof(serveraddr);

        /* receive info */
        if ((t_rcvudata(xtifd, unitdata, &flags)) 8
            < 0) {
            xerror("t_rcv", xtifd);
            exit(1);
        }

        buffer[unitdata->udata.len] = '\0';

        if ((n=analyze(buffer))== 0) {
            printf("transaction failure, "
                "try again\n");
        }
    }
}
```

例 B-8: コネクションレス型 XTI クライアント・プログラム (続き)

```
        } else if (n<0) {
            printf("login failed, try again\n");
            init();
        }
    }
    if (t_free((char *)unitdata, T_UNITDATA) < 0) {
        xerror("xti_free", fd);
        exit(3);
    }
}
```

- ❶ `t_open` コールは、たとえば `/dev/streams/xtiso/udp+` 等のデバイス特殊ファイルを指定します。このファイル名により IP 上の UDP トラnsポート・プロトコルに必要な抽象化が提供されます。ソケット・インタフェースと異なりアドレス・ファミリを指定する部分 (たとえば `AF_INET`) では、デバイス特殊ファイルの選択によって既にアドレス・ファミリの情報が取得されています。`/dev/streams/xtiso/udp+` ファイルは UDP トラnsポートと IP を示します。STREAMS デバイスの詳細については第 5 章を参照してください。

B.2.2 項で使用されている `socket` コールの代わりに、ここでは `t_open` コールを使用します。

- ❷ `serveraddr` はソケットの通信ドメイン (`AF_INET`) に制御される `sockaddr_in` 型の構造体です。インターネット通信ドメインのソケット・アドレスにはネットワーク上で固有のインターネット・アドレスおよび 16 ビットのポート番号が含まれます。UDP を含む TCP/IP プロトコル群では、サーバのインターネット・アドレスとサーバがリスンしているポート番号になります。

`sockaddr_in` 構造体に含まれる情報はアドレス・ファミリ (またはプロトコル) に依存します。

- ❸ `AF_INET` はインターネット通信ドメインを指定します。`AF_OSI` がサポートされている場合は OSI 通信用のソケットの作成に使用することもできます。

- ④ プロトコルまたはアドレス・ファミリに依存するサーバについての情報を取得します。 `gethostbyname(3)` ルーチンを使用してサーバの IP アドレスを知ることができます。
- ⑤ `SERVER_PORT` は `<common.h>` ヘッダ・ファイルで定義されています。これは `short` 型の整数で、サーバ・プロセスと他のアプリケーション・プロセスの識別に使用します。
- ⑥ `t_bind` 関数でサーバ・アドレスとバインドし、クライアントがデータの送受信を開始できるようにします。
- ⑦ `t_sndudata` 関数でデータを送信します。
- ⑧ `t_rcvudata` 関数でデータを受信します。

B.3 共通コード

この付録に示されているアプリケーションのクライアントおよびサーバのすべてまたは一部で次のヘッダ・ファイルおよびデータベース・ファイルが必要となります。

- `<common.h>`
- `<server.h>`
- `serverauth.c`
- `serverdb.c`
- `xtierror.c`
- `<client.h>`
- `clientauth.c`
- `clientdb.c`

B.3.1 `common.h` ヘッダ・ファイル

例 B-9 は `<common.h>` ヘッダ・ファイルです。 `<common.h>` にはこの付録のすべてのプログラム例に必要な共通ヘッダ・ファイルと定数が含まれています。

例 B-9: common.h ヘッダ・ファイル

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h> 1
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <fcntl.h>
#include <xti.h>

#define SEPARATOR      ' ,'
#define PREAMBLE      "%%"
#define PREAMBLELEN    2 2
#define POSTAMBLE      "##"
#define POSTAMBLELEN   2

/* How to contact the server */
#define SERVER_PORT     1234 3
/* How to contact the client (for datagram only) */
#define CLIENT_PORT     1235

#define MAXBUFSIZE 4096
```

- 1 インクルードするヘッダ・ファイルのリスト。
- 2 定数を定義するステートメント。この定義によって、サーバとクライアント間のデータ交換のパスがより効率的に行なわれます。
- 3 SERVER_PORT はクライアントがサーバと通信できるようにプログラマによって任意に割り当てられる最も一般的なポートです。SERVER_PORT は接続を望むサービスの識別に使用します。ポート番号の 0 番から 1024 番はシステム用にリザーブされています。プログラマは他のアプリケーションと重複しないものであればどの番号でも選択することができます。デバッグ中はこの番号はランダムに (および試行とエラーによって) 選択されます。分散型のアプリケーションでは、他のアプリケーションとの重複を防ぐために、方針を立てる必要があります。

B.3.2 server.h ヘッダ・ファイル

例 B-10 は <server.h> ヘッダ・ファイルです。<server.h> にはサーバのデータベースにアクセスするためのデータ構造，クライアントからのメッセージを分析するためのデータ構造およびクライアントへのメッセージを合成するためのデータ構造が含まれます。

例 B-10: server.h ヘッダ・ファイル

```
#include "common.h"

struct merchant {
    char *name;
    char *passwd;
};

struct customer {
    char                *cardnum;
    char                *name;
    int                 limit;
    int                 balance;
    struct transaction  *tlist;
    /* presumably other data */
};

struct transaction {
    struct transaction  *nextcust;
    struct transaction  *nextglob;
    struct customer     *whose;
    char                *merchantname;
    int                 amount;
    char                *verification;
};

extern struct transaction *alltransactions;
extern struct merchant  merchant[];
extern int               merchantcount;
extern struct customer  customer[];
extern int               customercount;

#define INVALID (struct transaction *)1

#define MERCHANTAUTHERROR    "%A##"
#define USERAUTHERROR       "%U##"
#define USERAMOUNTERROR     "%V##"
#define TRANSMITERROR        "deadbeef"

/* define transaction_status flags */
```


例 B-10: server.h ヘッダ・ファイル (続き)

```
#define NAME                0x01
#define PASS                0x02
#define AMOUNT              0x04
#define NUMBER              0x08

#define AUTHMASK            0x03
#define VERIMASK            0x0C
```

B.3.3 serverauth.c ファイル

例 B-11 は serverauth.c ファイルです。

例 B-11: serverauth.c ファイル

```
/*
 *
 * Authorization information (not related to the
 * networking interface)
 *
 */

#include "server.h"

/*
 * Currently a simple non-encrypted password method to search db
 *
 */
authorizemerchant(char *merch, char *password)
{
    struct merchant *mp;

    for(mp = merchant; (mp)->name != (char *)NULL; mp++) {
        if (!strcmp(merch, (mp)->name)) {
            return (!strcmp(password, (mp)->passwd));
        }
    }
    return(0);
}

struct transaction *
verifycustomer(char *num, int amount, char *merchant)
{

```

例 B-11: serverauth.c ファイル (続き)

```
char buf[64];
struct customer      *cp;
struct transaction   *tp;

for(cp = customer; (cp)->cardnum != NULL; cp++) {
    if (!strcmp(num, (cp)->cardnum)) {
        if (amount <= (cp)->balance) {
            (cp)->balance -= amount;
            if ((tp = malloc(sizeof(
                struct transaction)))
                == NULL) {
                printf("Malloc error\n");
                return(NULL);
            }
            tp->merchantname = merchant;
            tp->amount = amount;
            sprintf(buf, "v%012d", time(0));
            if ((tp->verification =
                malloc(strlen(buf)+1))
                == NULL) {
                printf("Malloc err\n");
                return(NULL);
            }
            strcpy(tp->verification, buf);
            tp->nextcust = cp->tlist;
            tp->whose = cp;
            cp->tlist = tp;
            tp->nextglob = alltransactions;
            alltransactions = tp;
            return(tp);
        } else {
            return(NULL);
        }
    }
}
return(INVALID);
}

int      transaction_status;
int      authorized = 0;
int      amount = 0;
char     number[256];
char     Merchant[256];
char     password[256];
```

例 B-11: serverauth.c ファイル (続き)

```
char *
parse(char *cp)
{
    char            *dp, *ep;
    unsigned char   type;
    int             doauth = 0;
    char            *buffer;

    dp = cp;
    if ((buffer=malloc(256)) == NULL) {
        return(TRANSMITERROR);
    }

    while (*dp) {
        /* terminate the string at the postamble */
        if (!(ep=strstr(dp, POSTAMBLE))) {
            return(TRANSMITERROR);
        }
        *ep = '\0';
        ep = ep + POSTAMBLELEN; 1

        /* search for preamble */
        if (!(dp=strstr(dp, PREAMBLE))) {
            return(TRANSMITERROR);
        }
        dp += PREAMBLELEN;

        /* Now get the token */
        type = *dp++;

        switch(type) {
            case 'm':
                strcpy(Merchant, dp);
                transaction_status |= NAME;
                break;
            case 'p':
                strcpy(password, dp);
                transaction_status |= PASS;
                break;
            case 'n':
                transaction_status |= NUMBER;
                strcpy(number, dp);
                break;
            case 'a':
                transaction_status |= AMOUNT;
                amount = atoi(dp);
                break;
```

例 B-11: serverauth.c ファイル (続き)

```
        default:
            printf("Bad command\n");
            return(TRANSMITERROR);
    }
    if ((transaction_status & AUTHMASK) == AUTHMASK) {
        transaction_status &= ~AUTHMASK;
        authorized = authorizemerchant(
            Merchant, password);
        if (!authorized) {
            printf("Merchant not
                " authorized\n");
            return(MERCHANTAUTHERROR);
        }
    }

    /* If both amount and number gathered,
     * do verification      */
    if ((authorized) &&
        ((transaction_status & VERIMASK)
         == VERIMASK)) {
        struct transaction *tp;

        transaction_status &= ~VERIMASK;
        /* send a verification back */
        if ((tp=verifycustomer(number,
                                amount,
                                Merchant))
            == NULL) {
            return(USRAMOUNTERROR);
        } else if (tp==INVALID) {
            return(USRAUTHERROR);
        } else {
            sprintf(buffer,
                "%%%s###%%c%s###%%m%s##",
                tp->verification,
                tp->whose->name,
                tp->merchantname);
            return(buffer);
        }
    }

    dp = ep;
}
return(NULL);
```

例 B-11: serverauth.c ファイル (続き)

```
}
```

- ❶ この関数はカード・オペレータ (merchant) の登録情報、利用客のクレジット・カード番号および利用者の支払い金額を含む着信データをパースします。この関数は基礎となる TCP プロトコルがストリーム系なので、上記の情報のすべてが1つのメッセージで入手できると仮定することはできない点に注意してください。この関数は、データグラム型のサービスが使用されている場合、あるいは順次パケット (SEQPACKET) を使用したプロトコルが使用されている場合単純化することができます。この関数は情報のメッセージを順不同で単一でも複数でも受け入れるような仕様になっています。

B.3.4 serverdb.c ファイル

例 B-12 は serverdb.c ファイルです。

例 B-12: serverdb.c ファイル

```
/*
 *
 * Database of valid merchants and credit card customers with the
 * credit limits, etc.
 *
 */
#include "server.h"

struct merchant merchant[] = {
    {"abc", "abc"},
    {"magic", "magic"},
    {"gasco", "gasco"},
    {"furnitureco", "abc"},
    {"groceryco", "groceryco"},
    {"bakeryco", "bakeryco"},
    {"restaurantco", "restaurantco"},
    {NULL, NULL}
};

int merchantcount = sizeof(merchant)/sizeof(struct merchant)-1;

struct customer customer[] = {
```

例 B-12: serverdb.c ファイル (続き)

```
{ "4322546789701000", "John Smith",      1000,    800 },
{ "4322546789701001", "Bill Stone",      2000,    200 },
{ "4322546789701002", "Dave Adams",      1500,    500 },
{ "4322546789701003", "Ray Jones",       1200,    800 },
{ "4322546789701004", "Tony Zachry",     1000,    100 },
{ "4322546789701005", "Danny Einstein",  5000,    50  },
{ "4322546789701006", "Steve Simonyi",  10000,   5800 },
{ "4322546789701007", "Mary Ming",      1100,    700 },
{ "4322546789701008", "Joan Walters",    800,     780 },
{ "4322546789701009", "Gail Newton",    1000,    900 },
{ "4322546789701010", "Jon Robertson",  1000,   1000 },
{ "4322546789701011", "Ellen Bloop",    1300,    800 },
{ "4322546789701012", "Sue Svelter",    1400,    347 },
{ "4322546789701013", "Suzette Ring",    1200,    657 },
{ "4322546789701014", "Daniel Mattis",   1600,    239 },
{ "4322546789701015", "Robert Esconis",  1800,    768 },
{ "4322546789701016", "Lisa Stiles",    1100,    974 },
{ "4322546789701017", "Bill Brophy",    1050,    800 },
{ "4322546789701018", "Linda Smitten",  4000,    200 },
{ "4322546789701019", "John Norton",    1400,    900 },
{ "4322546789701020", "Danielle Smith",  2000,    640 },
{ "4322546789701021", "Amy Olds",       1300,    100 },
{ "4322546789701022", "Steve Smith",    2000,    832 },
{ "4322546789701023", "Robert Smart",    3000,    879 },
{ "4322546789701024", "Jon Harris",      500,    146 },
{ "4322546789701025", "Adam Gershner",   1600,    111 },
{ "4322546789701026", "Mary Papadimis",  2000,    382 },
{ "4322546789701027", "Linda Jones",    1300,    578 },
{ "4322546789701028", "Lucy Barret",     1400,    865 },
{ "4322546789701029", "Marie Gilligan",  1000,    904 },
{ "4322546789701030", "Kim Coyne",       3000,    403 },
{ "4322546789701031", "Mike Storm",     7500,   5183 },
{ "4322546789701032", "Cliff Clayden",   750,    430 },
{ "4322546789701033", "John Turing",     4000,    800 },
{ "4322546789701034", "Jane Joyce",     10000,   8765 },
{ "4322546789701035", "Jim Roberts",     4000,   3247 },
{ "4322546789701036", "Stevw Stephano", 1750,    894 },
{ NULL, NULL }
};

struct transaction *
alltransactions = NULL;
int customercount = sizeof(customer)/sizeof(struct customer)-1;
```

B.3.5 xtierror.c ファイル

例 B-13 は xtierror.c ファイルです。このファイルはエラーの際の説明メッセージの生成に使用されます。非同期エラーまたは非同期イベントについての詳細な情報の取得には、t_look 関数を使用される点に注意してください。

例 B-13: xtierror.c ファイル

```
#include <xti.h>
#include <stdio.h>

void t_scope();

void
xerror(char *marker, int fd)
{
    fprintf(stderr, "%s error [%d]\n", marker, t_errno);
    t_error("Transport Error");
    if (t_errno == TLOOK) {
        t_scope(t_look(fd));
    }
}

void
t_scope(int tlook)
{
    char *tmperr;

    switch(tlook) {
        case T_LISTEN:
            tmperr = "connection indication";
            break;
        case T_CONNECT:
            tmperr = "connect confirmation";
            break;
        case T_DATA:
            tmperr = "normal data received";
            break;
        case T_EXDATA:
            tmperr = "expedited data";
            break;
        case T_DISCONNECT:
            tmperr = "disconnect received";
            break;
        case T_UDERR:
            tmperr = "datagram error";
```

例 B-13: xtierror.c ファイル (続き)

```
        break;
    case T_ORDREL:
        tmperr = "orderly release indication";
        break;
    case T_GODATA:
        tmperr = "flow control restriction lifted";
        break;
    case T_GOEXDATA:
        tmperr = "flow control restriction "
                "on expedited data lifted";
        break;
    default:
        tmperr = "unknown event";
}

fprintf(stderr,
        "Asynchronous event: %s\n",
        tmperr);
}
```

B.3.6 client.h ヘッダ・ファイル

例 B-14 は client.h ヘッダ・ファイルです。

例 B-14: client.h ファイル

```
#include "common.h"

extern char    merchantname[];
extern char    password[];
extern char    *swipecard();
```

B.3.7 clientauth.c ファイル

例 B-15 は clientauth.c ファイルです。このファイルはカード・オペレータ (merchant) の登録情報を取得するコードとサーバから送信されるメッセージの分析を行なうロジックを取得するコードが含まれます。結果として生じるメッセージは解釈され、クライアントの要求がサーバによって許可されたか拒否されたかを判断します。

例 B-15: clientauth.c ファイル

```
#include "client.h"

init()
{
    printf("\nlogin: "); fflush(stdout);
    scanf("%s", merchantname);

    printf("Password: "); fflush(stdout);
    scanf("%s", password);

    srand(time(0));
}

/* simulate some network activity via sound */
soundbytes()
{
    int i;

    for(i=0;i<11;i++) {
        printf();
        fflush(stdout);
        usleep(27000*(random()%10+1));
    }
}

analyze(char *cp)
{
    char *dp, *ep;
    unsigned char type;
    char customer[128];
    char verification[128];

    customer[0] = verification[0] = '\0';

    dp = cp;

    while ((dp!=NULL) && (*dp)) {
        /* terminate the string at the postamble */
        if (!(ep=strstr(dp, POSTAMBLE))) {
            return(0);
        }
        *ep = '\0';
        ep = ep + POSTAMBLELEN;

        /* search for preamble */
        if (!(dp=strstr(dp, PREAMBLE))) {
            return(0);
        }
    }
}
```

例 B-15: clientauth.c ファイル (続き)

```
    }
    dp += PREAMBLELEN;

    /* Now get the token */
    type = *dp++;

    switch(type) {
        case 'm':
            if (strcmp(merchantname, dp)) {
                return(0);
            }
            break;
        case 'c':
            strcpy(customer, dp);
            break;
        case 'U':
            printf("Authorization denied\n");
            return(1);
        case 'V':
            printf("Amount exceeded\n");
            return(1);
        case 'A':
            return(-1);
        case 'v':
            strcpy(verification, dp);
            break;
        default:
            return(0);
    }
    dp = ep;
}
if (*customer && *verification) {
    printf("%s, verification ID: %s\n",
        customer, verification);
    return(1);
}
return(0);
}
```

B.3.8 clientdb.c ファイル

例 B-16 は clientdb.c ファイルです。このファイルにはカード・スワップ・アクションの擬似をするために使用される利用者のクレジット・カード番号のデータベースが含まれています。実際のアプリケーションでは磁気

リーダが適切なインタフェースを通じて番号を読み取ります。また、実際のアプリケーションでは番号のキャッシュは必要とされません。

例 B-16: clientdb.c ファイル

```
/*
 *
 * Database of customer credit card numbers to simulate
 * the card swapping action. In practice the numbers
 * will be read by magnetic readers through an
 * appropriate interface.
 */

#include <time.h>

char    merchantname[256];
char    password[256];

char *numbercache[] = {
    "4322546789701000",
    "4322546789701001",
    "4322546789701002",
    "4222546789701002",           /* fake id */
    "4322546789701003",
    "4322546789701004",
    "4322546789701005",
    "4322546789701006",
    "4322546789701007",
    "4322546789701008",
    "4322546789701009",
    "4322546789701010",
    "4322546789701011",
    "4322546789701012",
    "4322546789701013",
    "4322546789701014",
    "4322546789701015",
    "4322546789701016",
    "4322546789701017",
    "4322546789701018",
    "4222546789701018",           /* fake id */
    "4322546789701019",
    "4322546789701020",
    "4322546789701021",
    "4322546789701022",
    "4322546789701023",
    "4322546789701024",
    "4322546789701025",
    "2322546789701025",           /* fake id */
    "4322546789701026",
```

例 B-16: clientdb.c ファイル (続き)

```
        "4322546789701027",  
        "4322546789701028",  
        "4322546789701029",  
        "4322546789701030",  
        "4322546789701031",  
        "4322546789701032",  
        "4322546789701033",  
        "4322546789701034",  
        "4322546789701035",  
        "4322546789701036",  
};  
  
#define CACHEENTRIES (sizeof(numbercache)/sizeof(char *))  
  
char *  
swipecard()  
{  
    return(numbercache[random()%CACHEENTRIES]);  
}
```

C

IPv4 および IPv6 のソケット・プログラムの例

この付録では、クライアント/サーバ・プログラムの例として、注釈付きのファイルを記載します。クライアントはサーバに要求を送り、サーバからの応答を出力します。サーバはクライアントからの要求をリッスンし、要求を出力して、応答をクライアントに送ります。

このプログラムは実際に使用されているアプリケーションではありませんが、各種のソケット呼び出しのシーケンスと使用方法を示すように構成されています。

情報は、次のように編成されています。

- AF_INET ソケット・プログラム
 - クライアント
 - サーバ
- AF_INET6 ソケット・プログラム
 - クライアント
 - サーバ

これらのプログラムと、これらのクライアント・プログラムのプロトコル非依存バージョンが、`/usr/examples/ipv6/network_programming` ディレクトリにあります。この付録では、これらのプログラムの出力例も記載しています。

C.1 AF_INET ソケットを使用するプログラム

この節には、AF_INET ソケットを使用するクライアントおよびサーバ・プログラムを記載します。

C.1.1 AF_INET ソケットを使用するクライアント・プログラム

例 C-1 は、ユーザのシステムで作成，コンパイル，実行が可能なクライアント・プログラムの例を示しています。このプログラムは，コマンド行で指定したシステムに要求を送信し，そこから応答を受信します。

例 C-1: クライアント・スタブ・ルーチン

```
/*
 * *****
 * *
 * *   Copyright (c) Compaq Computer Corporation, 2000   *
 * *
 * *   The software contained on this media is proprietary to *
 * *   and embodies the confidential technology of Compaq *
 * *   Computer Corporation. Possession, use, duplication or *
 * *   dissemination of the software and media is authorized only *
 * *   pursuant to a valid written license from Compaq Computer *
 * *   Corporation. *
 * *
 * *   RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure *
 * *   by the U.S. Government is subject to restrictions as set *
 * *   forth in Subparagraph (c)(1)(ii) of DFARS 252.227-7013, *
 * *   or in FAR 52.227-19, as applicable. *
 * *
 * * *****
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <arpa/inet.h>

#define SERVER_PORT 7639
#define CLIENT_PORT 7739

#define MAXBUFSIZE 4096

int main (
    int argc,
    char **argv )
{
    int          s;
    char         databuf[MAXBUFSIZE];
    int          dcount;
    struct sockaddr_in  serveraddr;[1]
    struct sockaddr_in  clientaddr;
    int          serveraddrlen;
    const char    *ap;
    const char    *request = "this is the client's request";
    struct hostent *hp;
    char          *server;

    if (argc < 2) {
        printf("Usage: client <server>\n");
        exit(1);
    }
}
```

C-2 IPv4 および IPv6 のソケット・プログラムの例

例 C-1: クライアント・スタブ・ルーチン (続き)

```
}
server = argv[1];

bzero((char *) &serveraddr, sizeof(struct sockaddr_in)); 2
serveraddr.sin_family = AF_INET;
if ((hp = gethostbyname(server)) == NULL) { 3
    printf("unknown host: %s\n", server);
    exit(2);
}
serveraddr.sin_port = htons(SERVER_PORT);

while (hp->h_addr_list[0] != NULL) {
    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) { 4
        perror("socket");
        exit(3);
    }
    memcpy(&serveraddr.sin_addr.s_addr, hp->h_addr_list[0],
        hp->h_length);

    if (connect(s, (struct sockaddr *)&serveraddr,
        sizeof(serveraddr)) < 0) { 5
        perror("connect");
        close(s);
        hp->h_addr_list++;
        continue;
    }
    break;
}
if (send(s, request, strlen(request), 0) < 0) { 6
    perror("send");
    exit(5);
}
dcount = recv(s, databuf, sizeof(databuf), 0); 7
if (dcount < 0) {
    perror("recv");
    exit(6);
}
databuf[dcount] = '\0';

hp = gethostbyaddr((char *)&serveraddr.sin_addr.s_addr, 8
    sizeof(serveraddr.sin_addr.s_addr), AF_INET);
ap = inet_ntoa(serveraddr.sin_addr); 9
printf("Response received from");
```

例 C-1: クライアント・スタブ・ルーチン (続き)

```
if (hp != NULL)
    printf(" %s", hp->h_name);
if (ap != NULL)
    printf(" (%s)", ap);
printf(": %s\n", databuf);

close(s);
}
```

- ❶ `sockaddr_in` 構造体を宣言する。この型の構造体の使用は、ソケット (AF_INET) の通信ドメインにより指示され、これによって通信に IPv4 プロトコルが使用されます。
- ❷ サーバ・アドレスをクリアし、サーバ変数を設定する。IPv4 通信のソケット・アドレスは、32 ビット・インターネット・アドレスと 16 ビットのポート番号からなります。このアドレスは、サーバのインターネット・アドレスとサーバがリッスンするポート番号になります。
- ❸ サーバの IPv4 アドレスを取得する。 `gethostbyname` の呼び出しでは、IPv4 アドレスのみが返されます。
- ❹ `socket` 呼び出しで AF_INET ソケットを作成する。TCP またはコネクション指向型通信用にソケット・タイプ SOCK_STREAM を指定しています。
- ❺ `serveraddr` という名前の `sockaddr_in` 構造体内のアドレスを使用して、サーバに接続する。
- ❻ サーバに要求を送信する。
- ❼ サーバから応答を受信する。
- ❽ `serveraddr` という名前の `sockaddr_in` 構造体内のアドレスを使用して、サーバ名を取り出す。 `gethostbyaddr` 呼び出しは、入力として IPv4 アドレスを期待しています。
- ❾ サーバの 32 ビット IPv4 アドレスを、ドット表記のインターネット・アドレス・テキスト文字列に変換する。 `inet_ntoa` 呼び出しは、入力として IPv4 アドレスを期待しています。

C.1.2 AF_INET ソケットを使用するサーバ・プログラム

例 C-2 は、ユーザのシステムで作成、コンパイル、実行が可能なサーバ・プログラムの例を示しています。このプログラムは他のシステムで稼働しているクライアント・プログラムから要求を受信し、応答を送信します。

例 C-2: サーバ・スタブ・ルーチン

```
/*
 * *****
 * *
 * *   Copyright (c) Compaq Computer Corporation, 2000   *
 * *
 * *   The software contained on this media is proprietary to *
 * *   and embodies the confidential technology of Compaq *
 * *   Computer Corporation. Possession, use, duplication or *
 * *   dissemination of the software and media is authorized only *
 * *   pursuant to a valid written license from Compaq Computer *
 * *   Corporation. *
 * *
 * *   RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure *
 * *   by the U.S. Government is subject to restrictions as set *
 * *   forth in Subparagraph (c)(1)(ii) of DFARS 252.227-7013, *
 * *   or in FAR 52.227-19, as applicable. *
 * *
 * * *****
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <arpa/inet.h>

#define SERVER_PORT 7639
#define CLIENT_PORT 7739

#define MAXBUFSIZE 4096

int main (
    int argc,
    char **argv )
{
    int          s;
    char          databuf[MAXBUFSIZE];
    int          dcount;
    struct sockaddr_in  serveraddr; [1]
    struct sockaddr_in  clientaddr;
    int          clientaddrlen;
    struct hostent      *hp;
    const char          *ap;
    const char          *response = "this is the server's response";
    u_short            port;

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) { [2]
        perror("socket");
    }
```

例 C-2: サーバ・スタブ・ルーチン (続き)

```
        exit(1);
    }

    bzero((char *) &serveraddr, sizeof(struct sockaddr_in)); [3]
    serveraddr.sin_family      = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY); [4]
    serveraddr.sin_port        = htons(SERVER_PORT);

    if (bind(s, (struct sockaddr *)&serveraddr, sizeof(serveraddr)) < 0) { [5]
        perror("bind");
        exit(2);
    }
    if (listen(s, SOMAXCONN) < 0) { [6]
        perror("Listen");
        close(s);
        exit(3);
    }

    while (1) {
        int new_s;
        clientaddrlen = sizeof(clientaddr);
        new_s = accept(s, (struct sockaddr *)&clientaddr, &clientaddrlen); [7]

        dcount = recv(new_s, databuf, sizeof(databuf), 0); [8]
        if (dcount <= 0) {
            perror("recv");
            close(new_s);
            continue;
        }
        databuf[dcount] = '\0';
        hp = gethostbyaddr((char *)&clientaddr.sin_addr.s_addr, [9]
                           sizeof(clientaddr.sin_addr.s_addr), AF_INET);
        ap = inet_ntoa(clientaddr.sin_addr); [10]
        port = ntohs(clientaddr.sin_port);
        printf("Request received from");
        if (hp != NULL)
            printf(" %s", hp->h_name);
        if (ap != NULL)
            printf(" (%s)", ap);
        printf(" port %d \"%s\"\\n", port, databuf);

        if (send(new_s, response, strlen(response), 0) < 0) { [11]
            perror("send");
            close(new_s);
            continue;
        }
        close(new_s);
    }
    close(s);
}
```

- [1] `sockaddr_in` 構造体を宣言する。この型の構造体の使用は、ソケット (AF_INET) の通信ドメインにより指示され、これによって通信に IPv4 プロトコルが使用されます。

- ② AF_INET ソケットを作成する。TCP またはコネクション指向型通信用にソケット・タイプ SOCK_STREAM を指定しています。
- ③ サーバ・アドレスをクリアし、サーバ変数を設定する。IPv4 通信のソケット・アドレスは、32 ビット・インターネット・アドレスと 16 ビットのポート番号からなります。このアドレスは、サーバのインターネット・アドレスとサーバがリスンするポート番号になります。
- ④ サーバ・アドレスに、IPv4 ワイルドカード・アドレス INADDR_ANY を設定する。このアドレスは、システム上に実装されている任意のネットワーク・インタフェースを指定します。
- ⑤ サーバのアドレスを、AF_INET ソケットにバインドする。
- ⑥ ソケット上でコネクションをリスンする。サーバは、以前の accept 呼び出しの処理が終わるまで、最大 SOMAXCONN 個の保留コネクションをキューイングします。ソケット・サブシステムのカーネル属性についての詳細は、sys_attrs_socket(5) を参照してください。
- ⑦ このソケット上のコネクションを受け付ける。accept 呼び出しはクライアントのアドレスを、clientaddr という名前の sockaddr_in 構造体に格納します。
- ⑧ データをクライアントから受信する。
- ⑨ clientaddr という名前の sockaddr_in 構造体内のアドレスを使用して、クライアント名を取り出す。gethostbyaddr 呼び出しは、入力として IPv4 アドレスを期待しています。
- ⑩ サーバの 32 ビットの IPv4 アドレスを、ドット表記のインターネット・アドレス・テキスト文字列に変換する。inet_ntoa 呼び出しは、入力として IPv4 アドレスを期待しています。
- ⑪ クライアントに応答を送信する。

C.2 AF_INET6 ソケットを使用するプログラム

この節には、AF_INET6 ソケットを使用するクライアントおよびサーバ・プログラムを記載します。

C.2.1 AF_INET6 ソケットを使用するクライアント・プログラム

例 C-3 は、ユーザのシステムで作成、コンパイル、実行が可能なクライアント・プログラムの例を示しています。このプログラムは、コマンド行で指

定したシステムに要求を送信し，そこから応答を受信します。すべてのアドレスは，IPv6 アドレス・フォーマットです。

例 C-3: クライアント・スタブ・ルーチン

```
/*
 * *****
 * *
 * *   Copyright (c) Compaq Computer Corporation, 2000   *
 * *
 * *   The software contained on this media is proprietary to *
 * *   and embodies the confidential technology of Compaq *
 * *   Computer Corporation. Possession, use, duplication or *
 * *   dissemination of the software and media is authorized only *
 * *   pursuant to a valid written license from Compaq Computer *
 * *   Corporation. *
 * *
 * *   RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure *
 * *   by the U.S. Government is subject to restrictions as set *
 * *   forth in Subparagraph (c)(1)(ii) of DFARS 252.227-7013, *
 * *   or in FAR 52.227-19, as applicable. *
 * *
 * * *****
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <arpa/inet.h>

#define SERVER_PORT 7639
#define CLIENT_PORT 7739

#define MAXBUFSIZE 4096

int main (
    int argc,
    char **argv )
{
    int          s;
    char          databuf[MAXBUFSIZE];
    int          dcount;
    struct addrinfo *server_info; [1]
    struct addrinfo *cur_info;
    struct addrinfo hints;
    struct sockaddr_in6 serveraddr; [2]
    char          databuf[INET6_ADDRSTRLEN];
    char          node[MAXDNAME];
    char          service[MAXDNAME];
    int          ni;
    int          err;
    int          serveraddrlen;
    const char    *request = "this is the client's request";
    char          *server;

    if (argc < 2) {
        printf("Usage: client <server>\n");
    }
}
```

C-8 IPv4 および IPv6 のソケット・プログラムの例

例 C-3: クライアント・スタブ・ルーチン (続き)

```
    exit(1);
}
server = argv[1];
bzero((char *) &hints, sizeof(hints)); [3]
hints.ai_family = AF_INET6;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_flags = AI_ADDRCONFIG | AI_V4MAPPED;

sprintf(service, "%d", SERVER_PORT);

err = getaddrinfo(server, service, &hints, &server_info); [4]
if (err != 0) {
    printf("%s\n", gai_strerror(err));
    if (err == EAI_SYSTEM)
        perror("getaddrinfo");
    exit(2);
}
cur_info = server_info;

while (cur_info != NULL) {
    if ((s = socket(cur_info->ai_family, cur_info->ai_socktype, 0)) < 0) { [5]
        perror("socket");
        freeaddrinfo(server_info);
        exit(3);
    }
    if (connect(s, cur_info->ai_addr, cur_info->ai_addrlen) < 0 { [6]
        close(s);
        cur_info = cur_info->ai_next;
        continue;
    }
    break;
}
freeaddrinfo(server_info); [7]

if (send(s, request, strlen(request), 0) < 0) { [8]
    perror("send");
    exit(5);
}
dcount = recv(s, databuf, sizeof(databuf), 0); [9]
if (dcount < 0) {
    perror("recv");
    exit(6);
}
databuf[dcount] = '\0';
serveraddrlen = sizeof(serveraddr);
if (getpeername(s, (struct sockaddr*) &serveraddr, &serveraddrlen) < 0) { [10]
    perror("getpeername");
    exit(7);
}
printf("Response received from");
ni = getnameinfo((struct sockaddr*)&serveraddr, serveraddrlen, [11]
                node, sizeof(node), NULL, 0, NI_NAMEREQD);
if (ni == 0)
    printf(" %s", node);
ni = getnameinfo((struct sockaddr*)&serveraddr, serveraddrlen, [12]
                addrbuf, sizeof(addrbuf), NULL, 0, NI_NUMERICHOST);
if (ni == 0)
    printf(" (%s)", addrbuf);

printf(": %s\n", databuf);
```

例 C-3: クライアント・スタブ・ルーチン (続き)

```
        close(s);  
    }
```

- ❶ `addrinfo` 構造体と `hints` 構造体を宣言する。
- ❷ `sockaddr_in6` 構造体を宣言する。この型の構造体の使用は、ソケット (`AF_INET6`) の通信ドメインにより指示され、これによって通信に IPv6 プロトコルが使用されます。プロトコルに依存しないプログラムを作成したい場合は、`sockaddr_storage` 構造体を宣言します。
- ❸ アドレス文字列バッファ、ノード名文字列バッファ、サービス名文字列バッファ、エラー番号変数、サービス・アドレス長変数を宣言する。
- ❹ `hints` 構造体をクリアし、`hints` 変数を設定する。`hints` 構造体には、`getaddrinfo` の処理を指定する値が格納されています。この場合、`AF_INET6` により IPv6 アドレスが返されます。IPv6 アドレスが構成されている場合、`AI_ADDRCONFIG` 値および `AI_V4MAPPED` 値により AAA レコードが返されます。また、IPv6 アドレスがない場合、IPv4 アドレスが構成されていれば、A レコードが返されます。`hints` 構造体の値についての詳細は、`getaddrinfo(3)` を参照してください。
- ❺ サーバ・アドレスを取得する。`getaddrinfo` を呼び出すと、1 つ以上の `addrinfo` 型の構造体で、IPv6 フォーマットのアドレスが返されます。
- ❻ `AF_INET6` ソケットを作成する。ソケット・タイプは、`addrinfo` 構造体で指定します。
- ❼ `cur_info` という名前の `addrinfo` 構造体内のアドレスを使用して、サーバに接続する。
- ❽ すべての `addrinfo` 構造体を解放する。
- ❾ サーバに要求を送信する。
- ❿ サーバから応答を受信する。
- ⓫ コネクションの他端にあるピア・ソケットのアドレスを取得し、そのアドレスを `serverinfo` という名前の `sockaddr_in6` 構造体に格納する。

- [12] serveraddr という名前の sockaddr_in6 構造体内のアドレスを使用して getnameinfo を呼び出し、サーバ名を取得する。NI_NAMEREQD フラグは、このルーチンが、指定されたアドレスのホスト名を返すように指示しています。
- [13] serveraddr という名前の sockaddr_in6 構造体内のアドレスを使用して getnameinfo を呼び出し、サーバの数値アドレスを取得する。NI_NUMERICHOST フラグは、このルーチンが、指定されたアドレスのアドレス値を返すように指示しています。

C.2.2 AF_INET6 ソケットを使用するサーバ・プログラム

例 C-4 は、ユーザのシステムで作成、コンパイル、実行が可能なサーバ・プログラムの例を示しています。このプログラムは他のシステムで稼働しているクライアント・プログラムから要求を受信し、応答を送信します。

例 C-4: サーバ・スタブ・ルーチン

```
/*
 * *****
 * *
 * *      Copyright (c) Compaq Computer Corporation, 2000      *
 * *
 * *      The software contained on this media is proprietary to *
 * *      and embodies the confidential technology of Compaq    *
 * *      Computer Corporation. Possession, use, duplication or  *
 * *      dissemination of the software and media is authorized only *
 * *      pursuant to a valid written license from Compaq Computer *
 * *      Corporation.                                           *
 * *
 * *      RESTRICTED RIGHTS LEGEND Use, duplication, or disclosure *
 * *      by the U.S. Government is subject to restrictions as set *
 * *      forth in Subparagraph (c)(1)(ii) of DFARS 252.227-7013, *
 * *      or in FAR 52.227-19, as applicable.                   *
 * *
 * * *****
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <arpa/inet.h>

#define SERVER_PORT 7639
#define CLIENT_PORT 7739

#define MAXBUFSIZE 4096

int main (
```

例 C-4: サーバ・スタブ・ルーチン (続き)

```
int argc,
char **argv )
{
    int          s;
    char          databuf[MAXBUFSIZE];
    int          dcount;
    struct sockaddr_in6 serveraddr; [1]
    struct sockaddr_storage clientaddr; [2]
    char          addrbuf[INET6_ADDRSTRLEN];
    char          node[MAXDNAME];
    char          port[MAXDNAME];
    int          err;
    int          ni;
    int          clientaddrlen;
    const char    *response = "this is the server's response";

    if ((s = socket(AF_INET6, SOCK_STREAM, 0)) < 0) { [3]
        perror("socket");
        exit(1);
    }

    bzero((char *) &serveraddr, sizeof(struct sockaddr_in6)); [4]
    serveraddr.sin6_family = AF_INET6;
    serveraddr.sin6_addr = in6addr_any;
    serveraddr.sin6_port = htons(SERVER_PORT);

    if (bind(s, (struct sockaddr *)&serveraddr, sizeof(serveraddr)) < 0) { [5]
        perror("bind");
        exit(2);
    }
    if (listen(s, SOMAXCONN) < 0) { [6]
        perror("listen");
        close(s);
        exit(3);
    }
    while (1) {
        int new_s;
        clientaddrlen = sizeof(clientaddr);
        bzero((char *)&clientaddr, clientaddrlen); [7]
        new_s = accept(s, (struct sockaddr *)&clientaddr, &clientaddrlen); [8]

        if (new_s < 0) {
            perror("accept");
            continue;
        }
        dcount = recv(new_s, databuf, sizeof(databuf), 0); [9]
        if (dcount < 0) {
            perror("recv");
            close(new_s);
            continue;
        }
        databuf[dcount] = '\0';

        printf("Request received from");
        ni = getnameinfo((struct sockaddr *)&clientaddr, [10]
            clientaddrlen, node, sizeof(node), NULL, 0, NI_NAMEERQD);
        if (ni == 0)
            printf(" %s", node);
        ni = getnameinfo((struct sockaddr *)&clientaddr, [11]
            clientaddrlen, addrbuf, sizeof(addrbuf), port, sizeof(port),
```


例 C-4: サーバ・スタブ・ルーチン (続き)

```
        NI_NUMERICHOST|NI_NUMERICSERV);
if (ni == 0)
    printf(" (%s) port %d", addrbuf, port);
printf(" \">%s%\n", port, databuf);
if (send(new_s, response, strlen(response), 0) < 0) {12
```

例 C-4: サーバ・スタブ・ルーチン (続き)

```
        perror("send");
        close(new_s);
        continue;
    }
    close(new_s);
}
close(s);
}
```

- ❶ serveraddr という名前の sockaddr_in6 構造体を宣言する。この型の構造体の使用は、ソケット (AF_INET6) の通信ドメインにより指示され、これによって通信に IPv6 プロトコルが使用されます。
- ❷ clientaddr という名前の sockaddr_storage 構造体を宣言する。この型の構造体を使用すると、プログラムをプロトコル非依存にすることができます。
- ❸ AF_INET6 ソケットを作成する。TCP またはコネクション指向型通信用にソケット・タイプ SOCK_STREAM を指定しています。
- ❹ サーバ・アドレスをクリアし、サーバ変数を設定する。
- ❺ サーバのアドレスを、AF_INET ソケットにバインドする。
- ❻ ソケット上でコネクションをリッスンする。サーバは、以前の accept 呼び出しの処理が終わるまで、最大 SOMAXCONN 個の保留コネクションをキューイングします。ソケット・サブシステムのカーネル属性についての詳細は、sys_attrs_socket(5) を参照してください。
- ❼ クライアント・アドレスをクリアする。
- ❽ このソケット上のコネクションを受け付ける。accept 呼び出しは、クライアントのアドレスを、clientaddr という名前の sockaddr_storage 構造体に格納します。
- ❾ データをクライアントから受信する。
- ❿ clientaddr という名前の sockaddr_storage 構造体内のアドレスを使用して getnameinfo を呼び出し、クライアント名を取得する。NI_NAMEREQD フラグは、このルーチンが、指定されたアドレスのホスト名を返すように指示しています。

- [11] `clientaddr` という名前の `sockaddr_storage` 構造体内のアドレスを使用して `getnameinfo` を呼び出し、クライアントの数値アドレスとポート番号を取得する。 `NI_NUMERICHOST` フラグと `NI_NUMERICSERV` フラグは、このルーチンが、指定されたアドレスとポート番号に対応する数値を返すように指示しています。
- [12] クライアントに応答を送信する。

C.3 プログラムの出力例

この節では、サーバ・プログラムとクライアント・プログラムの出力例を示します。サーバ・プログラムは `AF_INET6` ソケット上の要求はすべて、`sockaddr_in6` を用いて受け付け、受信します。IPv4 で受信した要求では、`sockaddr_in6` には IPv4 にマップされた IPv6 アドレスが格納されています。

次の例では、ノード `hostb6` でクライアント・プログラムが実行中であり、ノード `hosta6` に要求を送信しています。このプログラムは `AF_INET6` ソケットを使用します。 `hosta6` ノードの IPv6 アドレスは、DNS (Domain Name System) で `3ffe:1200::a00:2bff:fe97:7be0` になっています。

```
user2@hostb6> ./client hosta6
Response received from hosta6.ipv6.corp.example (3ffe:1200::a00:2bff:fe97:7be0):
this is the server's response
```

サーバ・ノードでは、次の例でサーバ・プログラムの起動とクライアント・ノード `hostb6` からの要求の受信を示します。

```
user1@hosta6> ./server
Request received from hostb6.ipv6.corp.example (3ffe:1200::a00:2bff:fe2d:02b2
port 7739 "this is the client's request"
```

次の例では、ノード `hostb` でクライアント・プログラムが実行中であり、ノード `hosta` に要求を送信しています。プログラムは `AF_INET6` ソケットを使用します。 `hosta` ノードの IPv4 アドレスは、DNS で `10.10.10.13` になっています。

```
user2@hostb> ./client hosta
Response received from hosta.corp.example (::ffff:10.10.10.13): this is the
server's response
```

サーバ・ノードでは、次の例でサーバ・プログラムの起動とクライアント・ノード `hostb` からの要求の受信を示します。

```
user1@hosta6> ./server
Request received from hostb.corp.example (::ffff:10.10.10.251) port 7739
"this is the client's request"
```

次の例では、ノード `hostc` でクライアント・プログラムが実行中であり、ノード `hosta` に要求を送信しています。このプログラムは、IPv4 のみのシステムで `AF_INET` ソケットを用いて作成され、実行されたものです。

```
user3@hostc> ./client hosta
Response received from hosta.corp.example (10.10.10.13): this is the
server's response
```

サーバ・ノードでは、次の例でサーバ・プログラムの起動とクライアント・ノード `hostc` からの要求の受信を示します。

```
user1@hosta6> ./server
Request received from hostc.corp.example (::ffff:10.10.10.63) port 7739
"this is the client's request"
```

TCP 固有のプログラミング情報

この付録では、伝送制御プロトコル (TCP) の性能について説明します。

ソケット・オプションによって次の項目を制御することにより、プログラムで TCP のスループットをチューニングする方法について説明します。

- TCP のウィンドウ・サイズ
- TCP のエラー回復
- TCP の往復時間
- TCP の信頼性

D.1 TCP のスループットとウィンドウ・サイズ

TCP のスループットは2つの要因で決定されます。つまり、ネットワークがパケットを受け入れる速度である転送速度と、TCP セグメントが送信される時間とそのセグメントの肯定応答が着信する時間との間の遅延である往復にかかる時間です。これらの要因により、TCP 接続上で最大のスループットを得るため、肯定応答受信の前に、バッファにいたるデータ量 (ウィンドウ) が決定されます。

転送速度または往復にかかる時間、あるいはその両方の値が大きい場合、TCP が使用する省略時の設定のウィンドウ・サイズでは、パイプを完全にロードしておくには不十分なことがあります。このような状況では、送信側は、以前に送信したデータの肯定応答を受信するまで待機しなければならないため、TCP のスループットが制限されます。

TCP 接続に使用する受信ウィンドウの最大のサイズは、受信ソケットのバッファ・サイズによって決定されます。また、送信側からの転送速度は、送信ソケットのバッファ・サイズによって制限されます。TCP の送信バッファおよび受信バッファでは、省略時の値は 61440 バイトです。

D.1.1 TCP ソケット・バッファ・サイズのプログラミング

アプリケーションは、接続を確立する前に、`setsockopt` システム・コールの `SO_SNDBUF` および `SO_RCVBUF` オプションを使用して、省略時の TCP の送信ソケットおよび受信ソケットのバッファ・サイズの指定を変更することができます。 `SO_SNDBUF` および `SO_RCVBUF` オプションで指定できる最大のサイズは、カーネル変数 `sb_max` によって制限されます。この値を増やす方法については、D.1.2.1 項を参照してください。

最大のスループットを得るために、接続の両端の送信ソケット・バッファおよび受信ソケット・バッファを同じサイズにしてください。

`setsockopt` システム・コールを使用して TCP ソケット・バッファ・サイズ (`SO_SNDBUF` および `SO_RCVBUF`) を変更するプログラムを作成している場合は、TCP 接続に使用される実際のソケット・バッファ・サイズが、指定した値より大きくなる可能性があることに注意してください。この状況は、指定したソケット・バッファ・サイズが、接続に使用される TCP の最大セグメント・サイズ (MSS) の倍数でない場合に起こります。

実際のサイズは、TCP によって決定され、折衝された MSS に最も近い倍数に切り上げられた指定のサイズです。ローカル・ネットワーク接続では、MSS は、一般にネットワーク・インタフェース・タイプおよびその最大転送ユニット (MTU) によって決定されます。

D.1.2 TCP ウィンドウ・スケール・オプション

Tru64 UNIX では、RFC 1323: 『*TCP Extensions for High Performance*』で定義されているように、TCP ウィンドウ・スケール・オプションをインプリメントしています。TCP ウィンドウ・スケール・オプションを使用すると、より大きなウィンドウを使用できるようになりますが、これは、帯域幅が高く遅延が長いネットワーク上の TCP のスループットを上げるために設計されたものです。このオプションを使用すると、ローカルな Gigabit Ethernet および FDDI ネットワークでの TCP のスループットを上げることもできます。

TCP ヘッダ内のウィンドウ・フィールドは 16 ビットです。したがって、ウィンドウ・スケール・オプションを指定しない場合、使用することのできる最大のウィンドウは、 2^{16} (64KB) になります。ウィンドウ・スケール・オプションが共同動作システム間で使用されるとき、ウィンドウは最大 $(2^{30}-1)$ バイトになります。接続の確立時に TCP 対等層間で伝送されるオ

プシオンは、実際のウィンドウ・サイズを取得するために各 TCP ヘッダ内のウィンドウ・サイズ値に適用される、スケール・ファクタを定義します。

最大受信ウィンドウも、また、接続確立時に TCP が提供するスケール・ファクタも、最大受信ソケット・バッファ・スペースによって決定されます。

受信ソケット・バッファ・サイズが 65535 バイトより大きな値の場合、TCP は、接続確立時に、受信ソケット・バッファのサイズに基づいたスケール・ファクタを使用してウィンドウ・スケール・オプションを指定します。TCP 接続された2つのシステムは、接続時にどちらかの方向に発生するウィンドウ・スケーリング用の、SYN セグメント内のウィンドウ・スケール・オプションを送信しなければなりません。前述したように、スループットを最大にするため、接続の両端の送信バッファおよび受信バッファを、同じサイズにしてください。

D.1.2.1 システム・ソケット・バッファ・サイズ制限の増加

ソケット・カーネル・サブシステムの `sb_max` カーネル属性は、各送受信バッファに対して割り当てられるソケット・バッファ・スペースの量を制限します。現在の省略時の値は 1048576 バイト (1MB) ですが、値を増やすこともできます。

ローカルな Gigabit Ethernet 接続では、現在の値で十分です。遅延が長く帯域幅が高いパスに対しては、値を 1MB より大きくする必要があります。

メモリ内に現在あるカーネルの `sb_max` カーネル属性を変更するには、`dxkerneltuner` ユーティリティか `sysconfig -r` コマンドを使用します。詳細については、`dxkerneltuner(8)` または `sysconfig(8)` を参照してください。

D.2 TCP の性能とエラー回復

TCP は、宛先にパケットが到着したかどうかを肯定応答によって判断します。大きなウィンドウを使用する高速のコネクション (Gigabit Ethernet など) では、省略時のメカニズムを使用すると、スループットに大きく影響することがあります。

特に指定しなければ、パケットが紛失すると、TCP はそのパケットとそれ以降のパケットをすべて再送します。アプリケーションでは、コネクションの確立前に `TCP_SACKENA` オプションを指定した `setsockopt` を使用することで、この省略時の動作を無効にできます。このオプションが合意される

と、データ受信側は送信側に、正しく到着したセグメントすべてについての情報を通知できるようになります。このようにすると、送信側が再送する必要があるのは、実際に紛失したセグメントだけになります。このオプションは、複数のセグメントが欠落した場合に便利です。

D.3 TCP の性能と往復時間の計測

TCP は、ウィンドウごとにパケットを 1 つだけ使って、往復時間を計測します。大きなウィンドウを使用する高速のコネクション (Gigabit Ethernet など) では、往復時間の性能が非常に悪いと判断され、再送が多く発生する可能性があります。

特に指定しなければ、TCP は TCP ヘッダにタイムスタンプを設定して送信することはありません。アプリケーションでは、コネクションの確立前に `TCP_TSOPTENA` オプションを指定した `setsockopt` システム・コールを使用することで、この省略時の動作を無効にできます。このオプションの選択後、送信側は各データ・セグメント内にタイムスタンプを格納します。受信側は、これらのタイムスタンプを受け付けるように構成されていれば、ACK セグメントにそのタイムスタンプを入れて返送します。これにより送信側は、往復時間を計測する、信頼性の高いメカニズムを利用できます。

D.4 TCP の信頼性とシーケンス番号

TCP はシーケンス番号を使用して、パケットの順序が正しいかと、重複したパケットを受信していないかを判断します。高速のコネクション (Gigabit Ethernet など) では、シーケンス番号が一巡する可能性があります。つまり、異なる情報を含む 2 つのパケットが、同じシーケンス番号となる可能性があります。これらのパケットは重複しているわけではありませんが、TCP は重複と判断してしまいます。

特に指定しなければ、TCP での古い重複パケットを拒否するメカニズムは有効になっていません。アプリケーションでは、コネクションの確立前に、`TCP_TSOPTENA` オプションを指定した後で `TCP_PAWS` オプションを指定した `setsockopt` システム・コールを使用することで、この省略時の動作を無効にできます。PAWS (Protect Against Wrapped Sequence numbers) オプションが有効になると、受信側は受信した古い重複セグメントを拒否します。このオプションは、同期型 TCP コネクションでのみ使用されます。

E

トークン・リング・ドライバの開発に関連する情報

この付録では Tru64 UNIX オペレーティング・システムのトークン・リング・ドライバの開発に関連する次の情報について説明します。

- ソース・ルーティング
- キャノニカル・アドレスの使用
- データの境界合わせ
- ドライバの `softc` 構造体フィールドの設定

E.1 ソース・ルーティング

ソース・ルーティングはトークン・リング・ローカル・エリア・ネットワーク (LAN) 上のシステムが、相互接続しているトークン・リング LAN 上の他のシステムにメッセージを送信する際に使用するブリッジ機構です。この機構では、メッセージのソースであるシステムはルート・ディスカバリー・プロセスを使用してトークン・リング LAN 上の最適なルートを判別し、デスティネーション・システムにブリッジします。

トークン・リング・ソース・ルーティング・モジュールを使用するためにカーネル構成ファイルに `TRSRCF` オプションを追加する必要があります。 `TRSRCF` オプションを追加するには、次のように `doconfig -c` コマンドを使用します。

1. スーパーユーザのプロンプト (`#`) から `doconfig -c HOSTNAME` コマンドを入力する。

`HOSTNAME` にはシステムの名前を大文字で指定します。たとえば、システム名が `host1` の場合、

```
# doconfig -c HOST1
```

と入力します。

2. カーネル構成ファイルに `TRSRCF` オプションを追加する。

カーネル構成ファイルを編集するかどうかについての質問に `y` と応答します。 `doconfig` コマンドでは `ed` エディタを使用して構成ファイルを編集することができます。 `ed` エディタの詳細についてはリファレンス・ページの `ed(1)` を参照してください。

次のコードは `ed` エディタを使用して `host1` のカーネル構成ファイルに `TRSRCF` オプションを追加する例です。追加する行の番号はカーネル構成ファイルによって異なります。

```
*** KERNEL CONFIGURATION AND BUILD PROCEDURE ***

Saving /sys/conf/HOST1 as /sys/conf/HOST1.bck

Do you want to edit the configuration file? (y/n) [n]: y

Using ed to edit the configuration file. Press return when ready,
or type 'quit' to skip the editing session:
2153

48a
options          TRSRCF
.
1,$w
2185
q

*** PERFORMING KERNEL BUILD ***
```

3. `doconfig` コマンドで作成された新しいカーネルをルート・ディレクトリ (`/`) に移動し、システムをリブートする。

カーネルの再構築および `doconfig` コマンドに関する詳細は『システム管理ガイド』を参照してください。

トークン・リング・ソース・ルーティング機能は `trn_units` 変数が 1 以上の場合に初期化されます。 `trn_units` 変数はシステム上で初期化されたトークン・リング・アダプタの数を表わします。

ドライバは次のように `trn_units` を宣言する必要があります。

```
extern int trn_units;
```

`attach` ルーチンの最後でドライバは次のように `trn_units` をインクリメントする必要があります。

```
trn_units++;
```

ソース・ルーティング管理については『ネットワーク管理ガイド：接続編』を参照してください。

E.2 キャノニカル・アドレス

トークン・リング・ドライバがアドレスをドライバより上層に渡している間、メディア・アクセス・コントロール (MAC) ヘッダ・ファイルのデスティネーション・アドレス (DA) とソース・アドレス (SA) はキャノニカル形式であることが要求されます。

キャノニカル・フォームは最下位ビット (LSB) 形式とも呼ばれます。キャノニカル・フォームは、最上位ビット (MSB) を最初に転送するノンキャノニカル・フォーム (または MSB 形式) と異なり、LSB を最初に転送します。また両者には各オクテット内のビットの順序が逆であるという相違があります。

次のアドレスはノンキャノニカル・フォームの例です。

```
10:00:d4:f0:22:c4
```

上記のアドレスはキャノニカル・フォームでは次のようになります。

```
08-00-2b-0f-44-23
```

ハードウェアが MAC ヘッダのアドレスがキャノニカル・アドレスのドライバを提供していない場合、アドレスを上層に渡す前に、アドレスをキャノニカル・フォームに変換する必要があります。次のフォーマットを含む `haddr_convert` カーネル・ルーチンを使用してキャノニカル・アドレスからノンキャノニカル・アドレスへの変換またはその反対の変換を行うことができます。

```
haddr_convert( addr)
unsigned char *addr
```

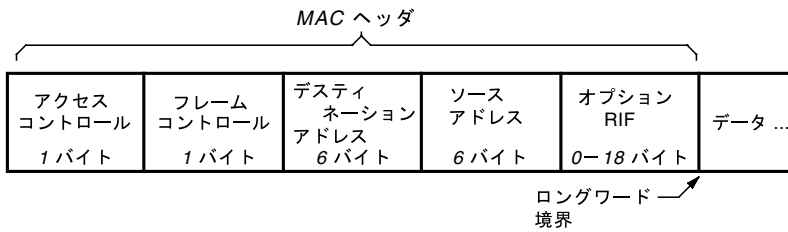
`addr` 変数は変換を必要とする 6 バイトのアドレスへのポインタです。変換されたアドレスは同じバッファに返されます。

E.3 データの境界合わせ

ドライバが受信するフレームは、ルーティング情報フィールド (RIF) とデータをインクルードするメディア・アクセス・コントロール (MAC) ヘッダで構成されます。RIF の長さは 0 バイトから 18 バイトまでの間で変化するので、RIF の後のデータがロングワードの境界に揃わないことがあります。性能の低下を防ぐために、RIF フィールドの埋め込みを行ない、データが常にロングワード境界から開始されるようにすることをお勧めします。

図 E-1 は MCA ヘッダのコンポーネントと典型的なフレームのデータの関係を示した概念図です。

図 E-1: 典型的なフレーム



ZK0894UJR

E.4 ドライバの **softc** 構造体のフィールドの設定

`softc` 構造体にはドライバに特有の情報が含まれています。

ドライバの `attach` ルーチンで `softc` 構造体の次のフィールドを設定する必要があります。

```
sc->isac.ac_arphrd=ARPHRD_802;
```

`sc` は `softc` 構造体のポインタで、`ARPHRD_802` はこのインタフェースから送られたアドレス解決プロトコル (ARP) パケットで使用されるハードウェア・タイプの値です。`ARPHRD_802` の値が 6 の場合 IEEE 802 ネットワークを示します。

データ・リンク・インタフェース

データ・リンク・インタフェース (DLI) はプログラムがデータ・リンク機能を直接使用して、リモート・システムで実行されているデータ・リンク・プログラムとの通信を行なうためのプログラミング・インタフェースです。

クライアントおよびサーバの DLI プログラム例は F.5 節を参照してください。

F.1 DLI プログラミングの前提条件

DLI プログラミングには C 言語の知識とシステム・プログラムの経験が要求されます。また、イーサネット・サブ構造体を使用する場合はイーサネット・プロトコルについての知識が、802 サブ構造体を使用する場合は 802.2、802.3 および FDDI プロトコルについての知識がそれぞれ必要です。

DLI インタフェースへのプログラムを書く場合はさらに次の概念についての知識が必要となります。

- データグラム・ソケット

アプリケーションはソケットを使用してイーサネット、802.3 および FDDI フレームの送受信を行ないますが、DLI はデータグラム・ソケットのみを使用します。

ソケットについての詳細は第 4 章を参照してください。

- 論理リンク・コントロール (LLC)

LLC は 802.2 フレーム・フォーマットの値によって決定されるサービス群を提供する DLI の補助層です。

- 物理アドレスおよびマルチキャスト・アドレス

物理アドレスまたはマルチキャスト・アドレスを使用してネットワーク上にメッセージを送受信することができます。デスティネーションが単一のシステムの場合は物理アドレスを使用してメッセージを送信します。デスティネーションが不特定のシステムの場合はマルチキャスト・アドレスを使用してパケットを送信します。マルチキャスト・

アドレスに送信されたパケットはマルチキャスト・アドレスが有効になっているすべてのシステムに受信されます。

マルチキャスト・アドレスの詳細は 4.7 節を参照してください。

- 標準フレーム・フォーマット

イーサネット・フレーム・フォーマットはコンパックコンピュータ社、インテル社およびゼロックス社が所有する標準フレーム・フォーマットです。IEEE 802.3 フレーム・フォーマットはマルチベンダ・ネットワークの標準フレーム・フォーマットです。FDDI フレーム・フォーマットと IEEE 802.3 フレーム・フォーマットは両者とも LLC (または 802.2) フレームを含む非常に類似しているフレーム・フォーマットです。詳細については F.3.1 項を参照してください。

このオペレーティング・システムでの DLI アプリケーションの実行にはスーパーユーザ特権またはルート特権が要求されます。

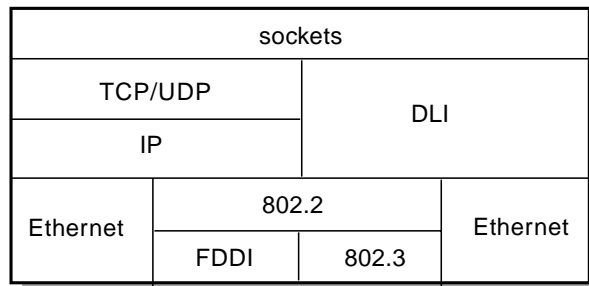
F.2 DLI 概略

DLI プログラムは標準イーサネット・フレーム・フォーマット、オープン・システム・インタコネクト (OSI) 802.3 フレーム・フォーマットまたは FDDI フレーム・フォーマットを使用してネットワーク上にデータを転送します。オペレーティング・システムではインターネット・プログラム、DECnet プログラムおよび DLI プログラムを同時に実行することができます。

オペレーティング・システムはイーサネット・データ・リンク・サービスおよび 802.2 データ・リンク・サービスの両方をサポートします。DLI および IP は両方ともイーサネットおよび 802.2 上で実行します。FDDI と 802.3 は 802.2 論理リンク・コントロール (LLC) をデータ・リンクの補助層として使用します。TCP および UDP は IP 上で実行し、両者を使用するプログラムにデータ配送サービスとメッセージ・ルーティング・サービスを提供します。DLI はデータ・リンク層への直接アクセスを提供するので TCP および UDP が行なう高次レベル・サービスは提供しません。

図 F-1 は DLI と IP、DLI とイーサネットおよび DLI と 802.2 の関係を表した概念図です。

図 F-1: DLI とネットワーク・プログラミング環境



ZK-0821-AI

ソケットは TCP, UDP および DLI へのアクセスを容易にするユーザ・アプリケーション・インタフェースです。DLI 通信ドメイン (AF_DLI) でのソケットのオープンについては第 4 章を参照してください。

F.2.1 DLI サービス

DLI はデータ・リンク層で次のサービスを提供します。

- データグラム・サービス
- 論理リンク・コントロール (LLC) 層
 - ISO 802.2 クラス I, タイプ 1 サービス
- マルチキャスト・アドレス・モード
- 媒体アクセス・コントロール (MAC) 層
 - イーサネット・フレーム
 - 802.3 フレーム
 - FDDI フレーム

F.2.2 ハードウェア・サポート

DLI は特定のシステムがどのドライバを構成しているかを probe ルーチンを使用して識別している, イーサネット・デバイス・ドライバまたは FDDI デバイス・ドライバを使用しますので, DLI ではハードウェアについての知識が必要とされません。サポートされているネットワーク・デバイスの一覧は Tru64 UNIX の『ソフトウェア仕様書』に記載されています。

システムに構成されているネットワーク・デバイスは `/usr/sbin/netstat -i` コマンドを次のように使用して識別します。

```
% /usr/sbin/netstat -i
```

Name	Mtu	Network	Address	Ipkts	Ierrs	Opkts	Oerrs	Coll
ln0	1500	<Link>		746	0	234	0	18
ln0	1500	orange-net	host1	746	0	234	0	18
sl0*	296	<Link>		0	0	0	0	0
sl1*	296	<Link>		0	0	0	0	0
lo0	1536	<Link>		74	0	74	0	0
lo0	1536	loop	localhost	74	0	74	0	0

コマンドを入力すると、システムで構成されているインタフェースとデバイスの情報が画面に出力されます。この例ではイーサネット・ハードウェア・デバイス (ln) が2つのシリアル・ライン・インタフェース・プロトコル・デバイス (sl0 および sl1) として構成されています。sl0 および sl1 の後ろのアスタリスク (*) はインタフェースのサポートが有効になっていないことを示します。

F.2.3 DLI を使用してのローカル・エリア・ネットワークへのアクセス

単一のローカル・エリア・ネットワーク (LAN) コントローラ上のデータ・リンクは同時に複数のユーザをサポートします。各局はネットワーク・チャンネル上の使用可能なポートを表します。

ネットワーク・チャンネルには同時に複数のユーザがアクセスするので、プログラムはメッセージを正しい受信者に確実に配送するアドレス機能を使用する必要があります。ネットワークに転送するメッセージにはデスティネーション・システムを示すイーサネット・アドレスまたは FDDI アドレスが含まれている必要があります。またメッセージがデスティネーション・システムの適切なユーザに届くために必要な識別子も含まれている必要があります (この識別子は使用するフレーム・フォーマットに依存します)。DLI はイーサネット、IEEE 802.3 または FDDI 標準に従ってフレームを構築します。

F.2.4 高次レベル・サービスのインクルード

DLI はデータグラム・サービスのみを提供します。DLI はデータ・リンク層への直接インタフェースなので、インターネットおよび DECnet では通常提供されている高次レベル・サービスを提供しません。したがって、次のサービスをアプリケーションで提供する必要があります。

- パケット・ルーティングおよび保証付き配送 (Guaranteed delivery)
- フロー・コントロール
- エラー回復

F-4 データ・リンク・インタフェース

- データ・セグメンテーション

F.3 DLI ソケット・アドレスのデータ構造体

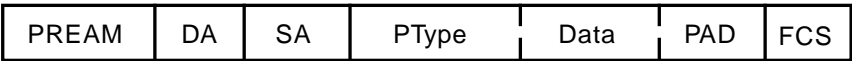
この節ではイーサネット、802.3 および FDDI の標準フレーム・フォーマットおよび DLI ソケット・アドレスのデータ構造体 (sockaddr_dl) の機能について記述します。また、sockaddr_dl を使用してドメイン・アドレス、ネットワーク・デバイスおよびイーサネット、802.3 または FDDI サブ構造体を指定する方法を説明します。

F.3.1 標準フレーム・フォーマット

以下の 4 つの図はイーサネット、802.3 および FDDI フレームの相違点と類似点を示す概念図です。

図 F-2 はイーサネット・フレーム・フォーマットの概念図です。

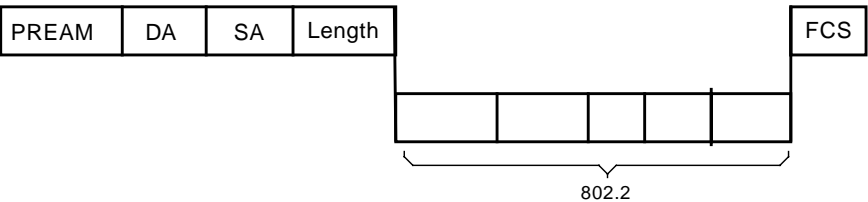
図 F-2: イーサネット・フレーム・フォーマット



ZK-0687U-AI

図 F-3 は 802.3 フレーム・フォーマットの概念図です。802.3 フレーム・フォーマットはその構造の中に図 F-5 の 802.2 構造を含んでいる点に注意してください。

図 F-3: 802.3 フレーム・フォーマット



ZK-0688U-AI

図 F-4 は FDDI フレーム・フォーマットの概念図です。FDDI フレーム・フォーマットもその構造の中に図 F-5 の 802.2 構造を含んでいます。

図 F-4: FDDI フレーム・フォーマット

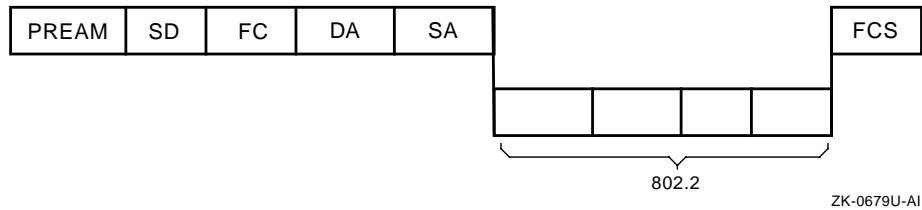
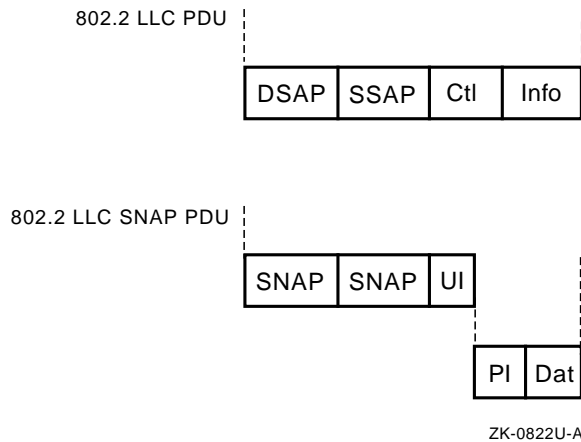


図 F-5 は 802.2 LLC PDU および 802.2 LLC SNAP PDU の概念図です。これらのうちの 1 つが 802.3 および FDDI フレーム・フォーマットに含まれます。

図 F-5: 802.2 構造体



一般的に 802 アプリケーションは 802.2 LLC PDU フォーマットを使用します。しかし、次のような理由から 802.2 LLC SNAP PDU フォーマットを選択することもできます。

- イーサネット上と 802.2 上の両方で操作するアプリケーションおよびイーサネットから 802.2 へ移行するアプリケーションには、SNAP_SAP を使用するとイーサネット・プロトコルを 802.2 プロトコルにマップするのが便利である。
- I/O コントロール・フラグ (DLI_NORMAL, DLI_EXCLUSIVE および DLI_DEFAULT) はイーサネットと 802.2 SNAP フレームのみで有効なので、SNAP 802.2 LLC PDU を使用しない場合にこれらのフラグが意味を持たなくなる。
- 通常の 802.2 LLC PDU は DSAP の上位 7 ビット上に多重化されているが、SNA_SAP は 5 バイトのプロトコル ID を持っているので、こ

れを使用すると 802.2 上でより多くのアプリケーションを実行することができる。

F.3.2 sockaddr_dl 構造体の機能

DLI はデータ・リンク層の通信に必要とされるサービス群の構成に使用するソケット・アドレスのデータ構造体を提供します。データ構造体 `sockaddr_dl` はアプリケーションがネットワークにバインドした時またはネットワークにパケットを転送した時に DLI へ情報を伝送するのに使用されます。また、アプリケーションがネットワークからパケットを受信した時にアプリケーションに情報を伝送するのにも使用されます。この情報にはネットワーク・デバイス情報、使用されるパケット・フォーマットおよびアドレス情報が含まれます。

次の例はヘッダ・ファイル `<dli/dli_var.h>` で定義されている DLI ソケット・アドレス構造体です。

```
#define DLI_ETHERNET    0
#define DLI_802        2
:
:

struct sockaddr_dl {
    u_char dli_len;                /* length of sockaddr */
    u_char dli_family;             /* address family (AF_DLI) */
    struct dli_devid dli_device;    /* id of comm device to use */
    u_char dli_substructype;        /* id to interpret following */
                                    /* structure */
    union {
        struct sockaddr_edl dli_eaddr; /* Ethernet */
        struct sockaddr_802 dli_802addr; /* OSI 802 support */
        caddr_t dli_aligner1;          /* this needs to have */
                                    /* longword alignment */
    } choose_addr;
};
```

すべてのアプリケーションはイーサネット・サブ構造体と 802 サブ構造体の両方を受信することができます。イーサネット・サブ構造体はアプリケーションのイーサネット間での通信を可能にし、802 サブ構造体はアプリケーションの 802.2、802.3 および FDDI プロトコル間での通信を可能にします。

イーサネット・サブ構造体または 802 サブ構造体を使用することで、ソケット・アドレス構造体の中でシステム・コールを使用して値を指定することができます。

サブ構造体のフィールドはシステム・コールの関数として更新されます。たとえば `bind` システム・コールはドメイン、ネットワーク・デバイスおよびサブ構造体の指定に使用されます。データの転送に `sendto` システム・

コールを使用する場合はドメイン、ネットワーク・デバイスおよびサブ構造体の一部を指定する必要があります。データの受信に `recvfrom` システム・コールを使用する場合、DLI は `sockaddr` 構造体のすべてのメンバに値を代入します。

ユーザが作成する `dli_econn` および `dli_802_3_conn` サブルーチンはソケットをオープンし、関連するドメイン、ネットワーク・デバイス名、プロトコル・タイプおよびその他のサブ構造体の情報をソケットにバインドします。ユーザ作成のサブルーチン `dli_econn` と `dli_802_3_conn` の例は F.5 節を参照してください。

以下にイーサネット・サブ構造体と 802.2 サブ構造体が DLI `sockaddr_dl` データ構造体に提供する関数について説明します。

F.3.3 イーサネット・サブ構造体

以下は DLI イーサネット・ソケット・アドレス・サブ構造体の例です。

```
#define DLI_EADDRSIZE 6
:
:

struct sockaddr_edl {
    u_char dli_ioctlflg;           /* i/o control flags */
    u_char dli_options;           /* Ethernet options */
    u_short dli_prototype;        /* Ethernet protocol type */
    u_char dli_target[DLI_EADDRSIZE]; /* Ethernet address of */
                                   /* destination system */
    u_char dli_dest[DLI_EADDRSIZE]; /* Ethernet address used to */
                                   /* address the local system; */
};                                /* DLI places the destination */
                                   /* address of an incoming */
                                   /* packet here to be used in */
                                   /* the recvfrom call. This */
                                   /* address can be the sys- */
                                   /* tem's address or a multi */
                                   /* cast address. */
```

イーサネット・サブ構造体は次の項目を指定します。

- プロトコル・タイプ (`dli_ioctlflg`) の I/O コントロール・フラグ
- イーサネットの埋め込みの有効または無効 (`dli_options`)

PAD は MAC/LLC ヘッダの後のリトル・エンディアン内の 2 バイト長のフィールドです。<dl/dli_var.h> ヘッダ・ファイルの以下のエントリが、埋め込みを有効にする場合に `dli_options` フィールドに設定される必要があるビットです。

```
#define DLI_ETHERPAD 0x01 /* Protocol is padded */
```

F-8 データ・リンク・インタフェース

- DLI プロトコル・タイプ (`dli_prototype`)
- デスティネーション・システムのイーサネット・アドレス (`dli_target`)
- ローカル・システムにアドレスするために使用されるイーサネット・アドレス (`dli_dest`)

この情報はイーサネット・フレーム・フォーマットの作成に使用されます。

F.3.3.1 イーサネット・フレーム

すべてのイーサネット・フレームはイーサネット・プロトコル・タイプ (PType) と呼ばれる 16 ビットの ID 番号を含みます。コントローラにメッセージが届くと、フレームを受信したポートの識別にプロトコル・タイプが使用されます。イーサネット間で通信する DLI アプリケーションは常に同じイーサネット・プロトコル・タイプを有効にしておく必要があります。また、プロトコル・タイプを使用して着信パケットの宛先ユーザを選択する方法に加えて、DLI を構成して、リモート・システムのプロトコル・タイプおよび物理アドレスの両方の関数としてユーザを選択する方法があります。これにより、同一システムの複数のアプリケーションが同一のタイプを使用することができるのでアプリケーションの出入力が簡素化されます。

F.3.3.2 イーサネット・サブ構造体の値の定義

ユーザは次のイーサネット・ソケット・アドレスサブ構造体のフィールドの値を定義します。他方のフィールドはシステム・コールまたは DLI によって代入されます。

- デスティネーションのアドレス (`dli_target [DLI_EADDRSIZE]`)
`dli_target` フィールドを使用してデスティネーションのアドレスを指定することができます。
- プロトコル・タイプ (`dli_prototype`)
`dli_prototype` フィールドを使用してデータ転送に使用されるプロトコルを指定することができます。
- I/O コントロール・フラグ (`dli_ioctlflg`)

以下は、イーサネット・サブ構造体でユーザが定義することができるメンバの値の説明です。

デスティネーション・ノードの物理アドレス

デスティネーション・システムの物理アドレス (図 F-2 の DA) は製造元により各局に割り当てられるイーサネット上で固有の 48 ビットの値です。物理アドレスは、たとえば 08-00-2b-XX-XX-XX (X は 16 進数) という形式で表わされます。DA はローカル・システムから見たリモート・システムのアドレスです。

bind コールで DA 値を指定しない場合は sendto コールでデータの送信時に指定し、recvfrom コールでデータ・メッセージのソースを識別する必要があります。sendto システム・コールへのメッセージの送信は物理アドレスまたはマルチキャスト・アドレスのどちらかを使用して行なうことができます。

プロトコル・タイプ

プロトコル・タイプ (図 F-2 の PType) はソース・アドレスに続くイーサネット・フレームの 16 ビットの値です。イーサネット・ドライバはフレーム内のデータの受信者の決定に使用するプロトコル・タイプを DLI に渡します。リザーブされている値を除いて、工場出荷時に割り当てられたイーサネット・プロトコル・タイプのうちシステム内で他に使用されていないものを任意に使用することができます。

次の 16 進数値はシステム用にリザーブされています。

- 0X 0200 — PUP プロトコル
- 0X 0800 — インターネット・プロトコル
- 0X 0806 — アドレス解決プロトコル
- 0X 6004 — ローカル・エリア・トランスポート
- 0X 6003 — Phase IV DECnet
- 0X 6002 — MOP CCR プロトコル
- 0X 6001 — MOP ダウンライン・ロード・プロトコル
- 0X 9000 — MOP ループバック・プロトコル
- 0X 1000 to 0X 100f — インターネット・トレイラ・プロトコル (VAX のみ使用)

I/O コントロール・フラグ

<dli/dli_var.h> ヘッダ・ファイルで定義されている I/O コントロール・フラグは DLI がプログラムのプロトコル・タイプのリザーブ方法の決定に使用する値です。この値を使用して DLI はユーザをプロトコル・タイプに限った関数として選択するか、プロトコル・タイプとターゲット・オーディエンスの組み合わせの関数として選択するかを決定します。次のリストは指定可能な I/O コントロール・フラグの定義と使用する際の条件の一覧です。

- NORMAL

指定したプロトコル・タイプだけを使用して、プログラムが1つのデスティネーション・システムとメッセージの交換ができるようにします。NORMAL フラグを使用する場合は bind コールにデスティネーション・システムの物理アドレスを指定する必要があります。またデータの送受信にすべてのデータ転送コールを使用することができます。DLI は指定されたターゲットから指定されたプロトコル・タイプを含むすべてのメッセージをユーザにフォワードします。

- EXCLUSIVE

指定されたプロトコル・タイプを排他的に使用して、プログラムが指定されたプロトコル・タイプを使用している任意のシステムとデータの交換ができるようにします。つまり、プログラムは指定されたプロトコル・タイプですべてのメッセージを受信します。EXCLUSIVE フラグを使用する場合、ターゲットのアドレスを bind コールで指定することはできません。この場合は sendto コールおよび recvfrom コールを使用して他のシステムとのデータ交換を行ない、ターゲットのアドレスは sendto コールで指定する必要があります。DLI はアドレス構造体 (recvfrom の戻り値) のターゲット・アドレスにイーサネット・フレームのソース・アドレスを代入します。またデスティネーションのアドレスにイーサネット・フレームのデスティネーション・アドレスを代入します。

- DEFAULT

プログラムは指定されたプロトコル・タイプを含むメッセージでシステムの他のプログラムに向けたものではないメッセージを受信することができます。メッセージのプロトコル・タイプまたはプロトコル・タイプとプロトコル・アドレスの組み合わせに排他的にバインドされたプログラムがほかにない場合、プロトコル・タイプとバインドしたソケットが省略時の設定でメッセージを受け取ります。このオペレーショ

ン・モードはメッセージのリッスンを行なうが送信する必要がないプログラムに対して使用すると効果的です。DEFAULT フラグを使用する場合はターゲット・アドレスの指定に bind コールを使用することはできません。この場合、recvfrom コールを使用して他のシステムからデータを受信してください。DEFAULT フラグを使用する場合、DLI がターゲット・アドレスにイーサネット・フレームのソース・アドレスを代入します。またデスティネーション・アドレスにイーサネット・フレームのデスティネーション・アドレスを代入します。

F.3.4 802.2 サブ構造体

802.2 サブ構造体によりアプリケーションは 802.2、802.3 および FDDI プロトコルを使用した相互通信が可能になります。このサブ構造体はクラス I、タイプ 1 サービスとの 802.2 プロトコルを使用してアプリケーションされるサービスの 2 つのオペレーション・モードを使用します。

以下は DLI 802.3 ソケット・アドレスのサブ構造体の例です。

```
struct sockaddr_802 {           /* 802.3 sockaddr struct */
    u_char iocfl;                /* filter on incoming packets */
                                /* addressed to the SNAP SAP */
    u_char svc;                  /* service class for this portal */
    struct osi_802hdr eh_802;    /* OSI 802 header format */
};
```

802.2 サブ構造体は 802.3 フレーム・フォーマットと FDDI フレーム・フォーマットを包括します。次のフィールドに値を指定することができます。

- デスティネーション・システムの物理アドレス (図 F-3 および図 F-4 の DA)
- サービス・クラス
- デスティネーションのサービス・アクセス・ポイント (図 F-5 の DSAP)
 - 個人
 - グループ
- ソース・サービス・アクセス・ポイント (図 F-5 の SSAP)

有効にする SSAP のタイプにより、プロトコル識別子と I/O コントロール・フィールドが必要とされることがあります。
- コントロール・フィールド

F.3.4.1 802 サブ構造体値の定義

以下は、802 サブ構造体で指定することができるすべてのメンバの値の説明です。

デスティネーション・ノードの物理アドレス

デスティネーション・システムの物理アドレス (DA) は製造元により各局に割り当てられるイーサネット上で固有の 48 ビットの値です。物理アドレスは、たとえば 08-00-2b-XX-XX-XX (X は 16 進数) という形式で表されます。このアドレスはアプリケーションがパケットの交換を試みるリモート・システムのアドレスです。I/O コントロール・フィールドが EXCLUSIVE または DEFAULT で、サービス・アクセス・ポイント (SAP) が SNAP_SAP タイプである場合を除いて、このアドレスは bind コールで指定される必要があります。また、SAP は sendto コールで指定される必要があります。

サービス・クラス

サービス・クラスはデータ・リンク層の論理リンク・コントロール (LLC) 補助層によって提供される機能を設定する 802.2 サブ構造体の値です。指定可能なサービス・クラスは次のとおりです。

- TYPE1

この値により DLI はすべてのヘッダ情報を中断し、クラス I、タイプ 1 サービスを提供します。

注意

タイプ 1 サービスが使用されているときには DLI ソフトウェアが XID および TEST パケットを処理するので、アプリケーションには透過的です。

DLI はソース・サービス・アクセス・ポイントとデスティネーション・サービス・アクセス・ポイントを使用してユーザに代わってコントロール・フィールドを解釈し、メッセージの受信者を決定します。DLI がデータ・フィールドをユーザに渡すかどうかはコントロール・フィールドの値に依存します。

- USER

この値によりごく限られたサービスのみが提供されます。したがってユーザは 802.2 プロトコルの大部分をインプリメントする必要があります。つまり、アプリケーションは XID および TEST パケットを処理しなければなりません。DLI はソース・サービス・アクセス・ポイントおよびデスティネーション・サービス・アクセス・ポイントを使用しますが、データを含むコントロール・フィールドをユーザに渡すので、ユーザはこのコントロール・フィールドを解析する必要があります。アプリケーションがクラス II、タイプ 2 サービスをインプリメントする必要がある場合はこのモードを選択しなければなりません。

デスティネーション・サービス・アクセス・ポイント

デスティネーション・サービス・アクセス・ポイント (DSAP) はメッセージがどのアプリケーションに向けられているかを識別する 802.2 フレームのフィールドです。

1 人または 1 グループのユーザの識別に個人 DSAP またはグループ DSAP を使用することができます。サービス・クラスが USER に設定されている場合に限りグループ DSAP を使用することができます。このフィールドに指定することができる値は次のとおりです。

- 個人 DSAP

- NULL_SAP — すべてゼロで構成される DSAP

TEST および XID の各コマンドとレスポンスを送信することができますが、NULL_SAP にデータを送信することはできません (TEST および XID は後述)。NULL_SAP はデータ・リンク層が主にテストの目的で、他のデータ・リンク層と通信する際に使用されます。

- ユーザ定義 DSAP — メッセージが向けられる 1 人のユーザを識別
ユーザ定義個人 DSAP には 2 以上 254 以下の偶数を指定します。

- SNAP_SAP — 802.3 サブネットワーク・アクセス・プロトコル

- グループ DSAP (ユーザ定義)

メッセージが向けられる複数のユーザを識別します。1 つのソケットで最大 127 のグループ DSAP に対してデータを送信できます。ユーザ定義グループ DSAP には 3 以上 255 以下の奇数を指定します。255 番はグローバル SAP なので、他のグループ SAP と同様に有効でなくてはなら

ない点に注意してください。グループ SAP はサービス・クラスが USER に設定されている場合に限り使用することができます。

ソース・サービス・アクセス・ポイント

ソース・サービス・アクセス・ポイント (SSAP) はメッセージを送信したアプリケーションのアドレスを識別する 802.2 フレームのフィールドです。1 つのソケットで有効にできる SSAP は 1 つに限られます。SSAP は 2 以上 254 以下の偶数です。

注意

SNAP_SAP を使用する場合は DSAP と SSAP の両方が SNAP_SAP に設定されている必要があります。さらに、プロトコル識別子とコントロール・フィールドを指定しなければなりません。プロトコル識別子は 5 バイト、コントロール・フィールドは 1 バイトです。サービス・クラスが TYPE1 の場合に限り SNAP_SAP を有効にすることができます。

また、IEEE 802.2 標準ではすべての SAP アドレスの最下位から 2 番目のビットを 1 に設定して定義しています。SAP の値は、目的にあわせて IEEE 802.2 標準に定義に従って使用することをお勧めします。

コントロール・フィールド

コントロール・フィールドはパケット・タイプを指定します。次の値がクラス I、タイプ 1 サービスに定義されています。以下に記述されている値はクラス I、タイプ 1 サービス用に定義されたものです。これらはクラス II、タイプ 2 サービスを提供するためにユーザ・サブライ・モードで使用することもできます。

注意

このユーザ・モードを使用するアプリケーションは適切なサービスを提供する役割を担います。クラス II サービスでサポートされるその他のオペレーションについては、米国電気電子技術者協会

(IEEE) によって発行されている『*IEEE Standards for Local Area Networks: Logical Link Control*』を参照してください。

- 交換識別

値 `XID` はコマンドまたはレスポンスの交換識別を識別します。8 ビットのフォーマット識別子と 16 ビットのパラメータが `XID` コントロール・フィールドに続きます。16 ビットのパラメータはサポートされている LLC サービスと受信ウィンドウ・サイズを識別します。LLC は IEEE/Std 802 ローカル・エリア・ネットワーク・プロトコルのデータ・リンク層の最上位補助層です。<dlc/dli_var.h> DLI ヘッダ・ファイルで定義されている `XID` 値の値は次のとおりです。

- `XID_PCMD`

ポール・ビットが設定された交換識別コマンド。交換識別コマンドはサポートされた LLC サービスのタイプと受信ウィンドウ・サイズをデスティネーション LLC に伝送します。このコマンドによりデスティネーション LLC は早い順に `XID` レスポンス・プロトコル・データ・ユニットでリプライを行ないます。ポール・ビットは 1 に設定され、PDU のレスポンスを繰り返し要求します。

- `XID_NPCMD`

ポール・ビットが設定されていない交換識別コマンド。ポール・ビットをクリアする点を除いて上記の `XID_PCMD` と同じです。レスポンスは期待しません。

- `XID_PRSP`

ポール・ビットが設定された交換識別レスポンス。データ・リンク層は交換識別レスポンスを使用して早い順に `XID` コマンドにリプライします。`XID` レスポンスの PDU はレスポンスを行なっている LLC を識別し、受信された `XID` コマンド PDU の情報フィールドにある情報の内容にかかわらず、`XID` コマンド PDU に定義されているように情報フィールドをインクルードします。最後のビットが 1 に設定され、このリプライが繰り返し要求を出しているコマンド PDU のリプライとして LLC によって送信されたことを示します。

- `XID_NPRSP`

ポール・ビットが設定されていない交換識別レスポンス。最後のビットがクリアされている点を除いて、上記の `XID_PRSP` と同じです。

- LLC プロトコル・データ・ユニット・テスト

値 `TEST` は LLC PDU コマンド・テストまたはレスポンス・テストを識別します。 `TEST` コントロール・フィールドの後ろにデータ・フィールドを続けることができます。 DLI ヘッダ・ファイル `<dli/dli_var.h>` で定義されている `TEST` の値は次のとおりです。

- `TEST_PCMD`

ポール・ビットが設定されている `TEST` コマンド。 `TEST` コマンドはデスティネーション LLC に早い順に `TEST` レスポンス PDU でレスポンスさせて、LLC 対 LLC の転送パスをテストします。 情報フィールドはこのコントロール・フィールド値ではオプションです。 使用された場合、受信している LLC は情報をユーザに渡すのではなく情報フィールドに代入します。 ポール・ビットが 1 に設定され、レスポンス PDU を繰り返し要求します。

- `TEST_NPCMD`

ポール・ビットが設定されていない `TEST` コマンド。 ポール・ビットがクリアされている点を除き、上記の `TEST_PCMD` と同じです。

- `TEST_PRSP`

ポール・ビットが設定されている `TEST` レスポンス。 `TEST` レスポンス PDU は `TEST` コマンド PDU のリプライです。 `TEST` コマンド PDU に情報フィールドがある場合は、対応する `TEST` レスポンス PDU に返されます。 最終ビットは 1 に設定され、要求を繰り返しているコマンド PDU のリプライとして LLC によってレスポンスが送信されたことを示します。

- `TEST_NPRSP`

ポール・ビットが設定されていない `TEST` レスポンス。 最終ビットがクリアされている点を除いて上記の `TEST_PRSP` と同じです。

- 非番号情報コマンド

ポールが設定されていない非番号情報コマンド (`UI_NPCMD`) は単一または複数の LLC に情報を送信します。 `UI_NPCMD` コマンドは LLC レスポンス PDU を持ちません。 これは通常直接アプリケーションに渡さ

れます。一般的にクラス I、タイプ 1 アプリケーションはこのコマンドを使用してデータの送受信を行ないます。

F.4 DLI プログラムの作成

この節では、DLI プログラムを書く際のシステム・コールの使用方法、およびイーサネット・サブ構造体と 802 サブ構造体内で値を指定する際の手順について説明します。

F.5 節にこの節で説明する手順の DLI プログラミング例が記載されています。

アプリケーション・プログラムを書く際のソケットおよびシステム・コールの使用方法の詳細は第 4 章を参照してください。

F.4.1 データ・リンク・サービスの提供

DLI はデータグラム・サービスのみを提供するので、DLI アプリケーションは高次レベルのネットワーク・ソフトウェアが通常提供するサービスを提供する必要があります。

- フロー・コントロール — 異なるシステムで実行している DLI プログラムはデータを同期転送しないとデータが失われます。
- エラー回復 — DLI はエラー報告をしますが、エラーの回復はアプリケーションで行なわなくてはなりません。
- データ・セグメンテーション — アプリケーションは転送中にデータのセグメントを行なわなければなりません (イーサネット、802.3、および FDDI パケットのバッファ・サイズについての詳細は F.4.7 項を参照してください)。

F.4.2 Tru64 UNIX システム・コール

DLI プログラムは入力引数、構造体および DLI に特化したサブ構造体とともにソケット・インタフェースを使用します。たとえば socket システム・コールを発行するときにプログラムはアドレス・フォーマット AF_DLI およびプロトコル DLPROTO_DLI を使用します。

DLI プログラムの先頭で `<dli/dli_var.h>` ヘッダ・ファイルをインクルードする必要があります。プログラムの本文は表 F-1 の呼び出し順序に従って記述します。

表 F-1: DLI プログラムの呼び出し順序

機能	システム・コール
ソケットの作成	socket
アドレス・ファミリ, フレーム・フォーマットおよび sockaddr_dl 構造体を使用してプログラムがデータを送信するデバイスを指定してソケットとデバイスをバインド	bind
ソケット・オプションの設定。このコールはオプションです。	setsockopt
データの転送	send, recv, read, write, sendto, recvfrom
ソケット・デスク립タの非アクティブ化	close

各システム・コールの詳細は第 4 章およびリファレンス・ページを参照してください。

以降の各項では DLI 関数, 入力引数および構造体について説明します。

F.4.3 ソケットの作成

DLI アプリケーションでは socket システム・コールに次の入力引数を指定してソケットを作成する必要があります。

アドレス・ファミリ: AF_DLI
ソケット・タイプ: SOCK_DGRAM
プロトコル: DLPROTO_DLI

値 AF_DLI は DLI アドレス・ファミリを指定します。SOCK_DGRAM は DLI で使用できる唯一のソケット・タイプのデータグラム・ソケットを作成します。DLI は他のプログラムに接続するために必要なサービスおよび他のソケット・タイプを使用するために必要なサービスを提供していません。値 DLPROTO_DLI は DLI プロトコル・モジュールを指定します。

以下は DLI にソケットをオープンする際の socket コールの使用例です。

```
int so;
:
:
if ( (so = socket(AF_DLI, SOCK_DGRAM, DLPROTO_DLI)) < 0)
{
```

```

        perror("cannot open DLI socket");
        return (-1);
    }

```

F.4.4 ソケット・オプションの設定

setsockopt コールを使用して, sockaddr_dl 構造体に次のソケット・オプションを設定します。

オプション	説明
DLI_ENAGSAP	グループ・サービス・アクセス・ポイント (DSAP) を使用可能にする。
DLI_DISGSAP	グループ・サービス・アクセス・ポイント (DSAP) を使用不可にする。
DLI_SET802CTL	802 コントロール・フィールドを設定する。
DLI_MULTICAST	マルチキャスト・アドレスにアドレスされたすべてのメッセージの受信を可能にする。

以下のコードはソケット・オプションを設定する際の setsockopt コールの例です。

この例は setsockopt コールを使用して GSAP オプションを使用可能にするものです。

```

/* enable GSAPs supplied by user */
j = 3;
i = 0;
while (j < argc) {
    sscanf(argv[j++], "%x", &k);
    out_opt[i++] = k;
}
optlen = i;
if
(setsockopt(sock, DLPROTO_DLI, DLI_ENAGSAP, &out_opt[0], optlen) < 0){
    perror("dli_setsockopt: Can't enable gsap");
    exit(1);
}

```

この例は setsockopt コールを使用して GSAP オプションを使用不可にするものです。

```

/* disable all but the last 4 or all GSAPs, */
/* whichever is smallest */
if (optlen > 4)
    optlen -= 4;
if
(setsockopt(sock, DLPROTO_DLI, DLI_DISGSAP, &out_opt[0], optlen) < 0){
    perror("dli_setsockopt: Can't disable gsap");
}

```


この例は `setsockopt` コールを使用して 802 コントロール・フィールドを設定するものです。

```
/* set 802 control field */
out_opt[0] = TEST_PCMD;
optlen = 1;
if
(setsockopt(sock, DLPROTO_DLI, DLI_SET802CTL,
            &out_opt[0], optlen) < 0) {
    perror("dli_setsockopt: Can't set 802 control");
    exit(1);
}
```

この例は `setsockopt` コールを使用して 2 つのマルチキャスト・アドレスを使用可能にするものです。

```
/* enable two multicast addresses */
bcopy(mcast0, out_opt, sizeof(mcast0));
bcopy(mcast1, out_opt+sizeof(mcast0), sizeof(mcast1));
if ( setsockopt(sock, DLPROTO_DLI, DLI_MULTICAST, &out_opt[0],
               (sizeof(mcast0) + sizeof(mcast1))) < 0 ) {
    perror("dli_setsockopt: can't enable multicast");
}
```

詳細な例については F.5 節を参照してください。

F.4.5 ソケットのバインド

ソケットの作成後、アプリケーションはソケットとネットワーク・デバイスをバインドしなければなりません。この時点でメッセージのフォーマット・タイプを指定します。まずソケットに名前を割り当てます。変数 `name` は `sockaddr_dl` タイプの構造体のポインタとなります。次に、`sockaddr_dl` データ構造体に情報を代入し適当なサブ構造体（イーサネットまたは 802）をインクルードします。

ソケットのバインドには次の `bind` システム・コールを使用します。詳細は `bind(2)` を参照してください。

`bind` システム・コールについての詳細はリファレンス・ページ `bind(2)` を参照してください。

F.4.6 `sockaddr_dl` 構造体

`sockaddr_dl` 構造体には次の情報を代入します。

- アドレス・ファミリ
- I/O デバイス ID
- サブ構造体のタイプ

F.4.6.1 アドレス・ファミリの指定

アドレス・ファミリの指定には、`socket` コール内で値 `AF_DLI` を使用します。

F.4.6.2 I/O デバイス ID の指定

I/O デバイスはプログラムがどのターゲット・システムを送信先および送信元とするかを制御するコントローラです。I/O デバイス ID はデバイス名 `dli_devname` およびデバイス番号 `dli_devnumber` を含みます。各変数の定義は次の情報をもとに行ないます。

- `dli_devname`
`netstat -i` コマンドがシステムで使用可能なデバイスの一覧を表示します。
- `dli_devnumber`
デバイス番号はシステム構成ファイルで設定されています。

F.4.6.3 サブ構造体タイプの指定

サブ構造体はプログラムが使用するフレーム・フォーマットのタイプを指定します。

- `dli_eaddr`
イーサネット・フレーム・フォーマット (`DLI_ETHERNET`)
- `dli_802addr`
802.3 フレーム・フォーマット (`DLI_802`)

各タイプに関連するソケットがある場合、プログラムはイーサネット、802.3 および FDDI フレームを送受信することができます。たとえば、あるシステムではイーサネット・フレームを使用して通信を行ない、他のシステムでは 802.3 または FDDI フレームを使用している場合があります。フレーム・フォーマットの選択はターゲット・プログラムが使用しているフレームのタイプに依存します。しかし、1 つのソケットが対応するフレームのタイプは 1 つに限られます。

プログラムはサブ構造体にユーザの選択を代入することでメッセージの送信用のパケット・ヘッダを指定します。例 F-1 はイーサネット・プロトコル用

に `sockaddr_dl` 構造体に値を代入する例です。例 F-2 は 802 プロトコル用に `sockaddr_dl` 構造体に値を代入する例です。

例 F-1: イーサネット用 `sockaddr_dl` 構造体の代入

```
/*
 * Fill out the sockaddr_dl structure for the bind call
 */
bzero(&out_bind, sizeof(out_bind));
out_bind.dli_family = AF_DLI;
out_bind.dli_substructype = DLI_ETHERNET;
bcopy(devname, out_bind.dli_device.dli_devname, i);
out_bind.dli_device.dli_devnumber = devunit;
out_bind.choose_addr.dli_eaddr.dli_ioctlflg = ioctl;
out_bind.choose_addr.dli_eaddr.dli_protype = ptype;
if ( taddr )
    bcopy(taddr, out_bind.choose_addr.dli_eaddr.dli_target,
          DLI_EADDRSIZE);

if ( bind(sock, &out_bind, sizeof(out_bind)) < 0 )
{
    perror("dli_eth, can't bind DLI socket");
    return(-1);
}

return(sock);
}
```

例 F-2: 802.2 用 `sockaddr_dl` 構造体の代入

```
/*
 * Fill out sockaddr_dl structure for the bind call.
 * Note that we need to determine whether the
 * control field is 8 bits (unnumbered format) or
 * 16 bits (informational/supervisory format). We do this
 * by checking the low order 2 bits, which are both 1 only
 * for unnumbered control fields.
 */
bzero(&out_bind, sizeof(out_bind));
out_bind.dli_family = AF_DLI;
out_bind.dli_substructype = DLI_802;
bcopy(devname, out_bind.dli_device.dli_devname, i);
out_bind.dli_device.dli_devnumber = devunit;
out_bind.choose_addr.dli_802addr.ioctl = ioctl;
out_bind.choose_addr.dli_802addr.svc = svc;
if(ctl & 3)
    out_bind.choose_addr.dli_802addr.eh_802.ctl.U_fmt=(u_char)ctl;
else
    out_bind.choose_addr.dli_802addr.eh_802.ctl.I_S_fmt = ctl;
out_bind.choose_addr.dli_802addr.eh_802.ssap = sap;
out_bind.choose_addr.dli_802addr.eh_802.dsap = dsap;
if ( ptype )
    bcopy(ptype, out_bind.choose_addr.dli_802addr.eh_802.osi_pi, 5);
if ( taddr )
    bcopy(taddr, out_bind.choose_addr.dli_802addr.eh_802.dst,
          DLI_EADDRSIZE);
```

例 F-2: 802.2 用 `sockaddr_dl` 構造体の代入 (続き)

```
if ( bind(sock, &out_bind, sizeof(out_bind)) < 0 )
{
    perror("dli_802, can't bind DLI socket");
    return(-1);
}

return(sock);
}
```

F.4.7 バッファ・サイズの計算

バッファ・サイズは通信しているシステムのコントローラが処理できるサイズを超えてはなりません。超えた場合データは失われます。イーサネット・パケットのバッファ・サイズの最大値は 1500 バイトです。

802.3 パケットのバッファ・サイズの最大値は次の式で算出します。

$$\text{bytes} = 1500 - [2 + (\text{control field} == \text{UI? } 1:2) + (\text{Source SAP} == \text{SNAP SAP ? } 5:0)]$$

コントロール・フィールドおよび Source SAP のバイト数は bind コールに指定されています。

FDDI パケットのバッファ・サイズの最大値は 4352 バイトです。

F.4.8 データ転送

DLI プログラムは `write` , `send` または `sendto` コールを使用してデータを送信 , `read` , `recv` または `recvfrom` コールを使用してデータを受信することができます。表 F-2 の " " は bind コール中に設定される I/O コントロール・フラグの関数として使用することができるシステム・コールの条件を示します。

注意

Normal コントロール・フラグを使用する場合は bind コールでターゲット・アドレスを設定する必要があります。Exclusive または Default コントロール・フラグを使用する場合は bind コールでターゲット・アドレスを指定する必要はありません。しか

し、ターゲット・アドレスを設定しない場合は `sendto` および `recvfrom` システム・コールを使用する必要があります。

表 F-2: DLI で使用されるデータ転送システム・コール

システム・コール	Normal コントロール	Exclusive コントロール	Default コントロール
<code>write</code>			
<code>send</code>			
<code>sendto</code>			
<code>read</code>			
<code>recv</code>			
<code>recvfrom</code>			

コントロール・フラグを `NORMAL` に設定した場合は `bind` コールでターゲット・アドレスを設定して、`write`、`send`、`sendto`、`read`、`recv`、`recvfrom` コールを使用してデータを転送します。

コントロール・フラグを `EXCLUSIVE` に設定した場合は `bind` コールのターゲット・アドレスの値をゼロにして、`sendto` コールでターゲット・アドレスを設定します。データ転送には `sendto` および `recvfrom` コールだけを使用します。

コントロール・フラグを `DEFAULT` に設定した場合は `bind` コールのターゲット・アドレスの値をゼロにして、`sendto` コールを使用してデータの送信を行ない、このコール内でターゲット・アドレスを設定します。データのソース・アドレスの識別には `recvfrom` コールを使用します。

F.4.9 ソケットの非アクティブ化

データの送受信が終了したら、`close` システム・コールを発行してソケットを非アクティブ化します。

F.5 DLI プログラミング例

この節では次の DLI プログラミング例を記述します。

- イーサネット・フォーマット・パケットを使用した DLI クライアント・プログラム例
- イーサネット・フォーマット・パケットを使用した DLI サーバ・プログラム例
- 802.3 フォーマット・パケットを使用した DLI クライアント・プログラム例
- 802.3 フォーマット・パケットを使用した DLI サーバ・プログラム例
- getsockopt および setsockopt システム・コールを使用した DLI プログラム例

上記のプログラム例は /usr/examples/dli ディレクトリにあります。

F.5.1 イーサネット・フォーマット・パケットを使用した DLI クライアント・プログラム例

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <memory.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <net/route.h>
#include <dli/dli_var.h>

/*
 *      d l i _ e x a m p l e : d l i _ e t h
 *
 * Description: This program sends out a message to a node where a
 *              companion program, dli_ethd, echoes the message back.
 *              The ethernet packet format is used. The ethernet
 *              address of the node where the companion program is
 *              running, the protocol type, and the message are
 *              supplied by the user. The companion program should
 *              be started before executing this program.
 *
 * Inputs:      device, target address, protocol type, short message.
 *
 * Outputs:     Exit status.
 *
 * To compile:  cc -o dli_eth dli_eth.c
 *
 * Example:     dli_eth ln0 08-00-2b-02-e2-ff 6006 "Echo this"
 *
 * Comments:    This example demonstrates the use of the "NORMAL" I/O
 *              control flag. The use of the "NORMAL" flag means that
 *              we can communicate only with a single specific node
 *              whose address is specified during the bind. Because
```

```

*           of this, we can use the normal write & read system
*           calls on the socket, because the source/destination of
*           all data that is read/written on the socket is fixed.
*
*/

/*
* Compaq Computer Corporation supplies this software example on
* an "as-is" basis for general customer use. Note that Compaq
* does not offer any support for it, nor is it covered under any
* of Compaq's support contracts.
*/

main(
    int argc,
    char **argv)
{
    struct sockaddr_dl sdl;
    size_t sdllen;
    int ch, fd, rsize, itarget[6], ptype, ioctflg = DLI_NORMAL, errflg = 0;
    u_char inbuf[4800], u_char *src;

    memset(&sdl, 0, sizeof(sdl));
    while ((ch = getopt(argc, argv, "xp:")) != EOF) {
        case 'x': ioctflg = DLI_EXCLUSIVE; break;
        case 'p': {
            if (sscanf(optarg, "%x", &ptype, &ch) != 1) {
                fprintf(stderr, "%s: invalid protocol type %s",
                    argv[0], optarg);
                errflg++;
                break;
            }
        }
        default: errflg++; break;
    }

    if (errflg || argc - optind < 5) {
        fprintf(stderr, "%s %s %s\n",
            "usage:",
            argv[0],
            "device lan-address short-message");
        exit(1);
    }

    /*
     * Get device name and unit number.
     */
    if (sscanf(argv[optind], "[%a-z]hd%c", sdl.dli_device.dli_devname,
        &sdl.dli_device.dli_devnumber, &ch) != 2) {
        fprintf(stderr, "%s: invalid device name",
            argv[0], argv[optind]);
        exit(1);
    }

    /*
     * Get the address to which we will be sending
     */
    if (sscanf(argv[optind+1], "%x%*[:-%]xx%*[:-%]xx%*[:-%]\
        %x%*[:-%]xx%*[:-%]xx%c",
        &itarget[0], &itarget[1], &itarget[2],
        &itarget[3], &itarget[4], &itarget[5], &ch) != 6) {
        fprintf(stderr, "%s: invalid lan address",
            argv[0], argv[optind]);
        exit(1);
    }
}

```

```

/*
 * If the LAN Address is a multicast, then we can't
 * use DLI_NORMAL. Use DLI_DEFAULT instead.
 */
if ((itarget[0] & 1) && ioctflg == DLI_NORMAL)
    ioctflg = DLI_DEFAULT;

/*
 * Fill out sockaddr structure for bind/sento/recvfrom
 */
sdl.dli_family = AF_DLI;
if (ptype < GLOBAL_SAP) {
    sdl.dli_substructype = DLI_802;
    sdl.choose_addr.dli_802addr.ioctl = ioctflg;
    sdl.choose_addr.dli_802addr.svc = TYPE1;
    sdl.choose_addr.dli_802addr.eh_802.dsap = ptype;
    sdl.choose_addr.dli_802addr.eh_802.ssap = ptype;
    sdl.choose_addr.dli_802addr.eh_802.ct1.U_fmt = UI_NPCMD;
    src = sdl.choose_addr.dli_802addr.eh_802.dst;
} else {
    sdl.dli_substructype = DLI_ETHERNET;
    sdl.choose_addr.dli_eaddr.dli_ioctflg = ioctflg;
    sdl.choose_addr.dli_eaddr.dli_prototype = ptype;
    src = sdl.choose_addr.dli_eaddr.dli_target;
}
/*
 * If we are using DLI_NORMAL, we must supply
 */
if (ioctflg == DLI_NORMAL) {
    src[0] = itarget[0]; src[1] = itarget[1]; src[2] = itarget[2];
    src[3] = itarget[3]; src[4] = itarget[4]; src[5] = itarget[5];
}

/*
 * Open a socket to DLI and then bind to our protocol/address.
 */
if ((fd = socket(AF_DLI, SOCK_DGRAM, DLPROTO_DLI)) < 0) {
    fprintf(stderr, "%s: DLI open failed: %s\n",
        argv[0], strerror(errno));
    exit(1);
}

if (bind(fd, (struct sockaddr *) &sdl, sizeof(sdl)) < 0) {
    fprintf(stderr, "%s: DLI bind failed: %s\n",
        argv[0], strerror(errno));
    exit(2);
}

if (ioctflg != DLI_NORMAL) {
    src[0] = itarget[0]; src[1] = itarget[1]; src[2] = itarget[2];
    src[3] = itarget[3]; src[4] = itarget[4]; src[5] = itarget[5];
}

/* send response to originator. */
sdllen = sizeof(sdl);
if (sendto(fd, argv[4], strlen(argv[4]), 0,
    (struct sockaddr *) &sdl, sdllen) < 0) {
    fprintf(stderr, "%s: DLI transmission failed: %s\n",
        argv[0], strerror(errno));
    exit(1);
}

if ((rsize = recvfrom(fd, inbuf, sizeof(inbuf), 0,
    (struct sockaddr *) &sdl, &sdllen)) < 0 ) {

```



```

        fprintf(stderr, "%s: DLI reception failed: %s\n",
                argv[0], strerror(errno));
    exit(1);
}

/* check header */
if (sdllen != sizeof(struct sockaddr_dl)) {
    fprintf(stderr, "%s, incorrect header supplied\n", argv[0]);
    exit(1);
}

if (from.dli_substructype == DLI_802)
    src = from.dli_choose_addr.dli_802addr.eh_802.dst;
else
    src = from.dli_choose_addr.dli_eaddr.dli_target;

/* any data? */
fprintf(stderr, "%s: %sdata received from ", argv[0],
        rsize ? : "NO ");
fprintf(stderr, "%02x-%02x-%02x-%02x-%02x-%02x",
        src[0], src[1], src[2], src[3], src[4], src[5]);
if (from.dli_substructype == DLI_802)
    fprintf(stderr, " SAP %02x\n\n",
            sdl.choose_addr.dli_802addr.eh_802.ssap & ~1);
else
    fprintf(stderr, " on protocol type %04x\n\n",
            sdl.choose_addr.dli_eaddr.dli_prototype);

/* print results */
printf("%s\n", inbuf);
close(fd);
return 0;
}

```

F.5.2 イーサネット・フォーマット・パケットを使用した DLI サーバ・プログラム例

```

#ifdef lint
static char *rcsid = "@(#) $RCSfile: ap-dli.sgml,v $ \
    $Revision: 1.1.6.3 $ (DEC) $Date: 1999/07/08 20:46:48 $";
#endif

#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <dli/dli_var.h>
#include <sys/ioctl.h>

extern int errno;

/*
 *      d l i _ e x a m p l e : d l i _ e t h d
 *
 * Description: This daemon program transmits any message it

```

```

* receives to the originating system, i.e., it echoes the
* message back. The device and protocol type are supplied
* by the user. The program uses ethernet format packets.
*
* Inputs:      device, protocol type.
*
* Outputs:     Exit status.
*
* To compile:  cc -o dli_ethd dli_ethd.c
*
* Example:     dli_ethd de0 6006
*
* Comments:    This example demonstrates the use of the "DEFAULT"
* I/O control flag, and the recvfrom & sendto system calls.
* By specifying "DEFAULT" when binding the DLI socket to
* the device we inform the system that this program will
* receive any ethernet format packet with the given
* protocol type which is not meant for any other program
* on the system. Since packets may arrive from
* different systems we use the recvfrom call to read the
* packets. This call gives us access to the packet
* header information so that we can determine where the
* packet came from. When we write on the socket we must
* use the sendto system call to explicitly give the
* destination of the packet.
*/

/*
* Compaq Computer Corporation supplies this software
* example on an "as-is" basis for general customer use. Note
* that Compaq does not offer any support for it, nor is it
* covered under any of Compaq's support contracts.
*/

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

{
    u_char inbuf[1500], outbuf[1500];
    u_char devname[16];
    u_char target_eaddr[6];
    char *cp;
    int rsize;
    unsigned int devunit;
    int i, sock, fromlen;
    unsigned int ptype;
    struct sockaddr_dl from;

    if ( argc < 3 )
    {
        fprintf(stderr,
            "usage: %s device hex-protocol-type\n", argv[0]);
        exit(1);
    }

    /* get device name and unit number. */
    bzero(devname, sizeof(devname));
    i = 0;
    cp = argv[1];
    while ( isalpha(*cp) )
        devname[i++] = *cp++;
    sscanf(cp, "%d", &devunit);

```

```

/* get protocol type */
sscanf(argv[2], "%x", &ptype);

/* open dli socket */
if
((sock = dli_econn(devname, devunit, ptype, NULL, \
    DLI_DEFAULT)) < 0)
{
    perror("dli_ethd, dli_econn failed");
    exit(1);
}

while ( 1 ) {
    /* wait for message */
    from.dli_family = AF_DLI;
    fromlen = sizeof(struct sockaddr_dl);
    if ((rsize = recvfrom(sock, inbuf, sizeof(inbuf),
        NULL, &from, &fromlen)) < 0 ) {
        sprintf(inbuf, "%s: DLI reception failed", argv[0]);
        perror(inbuf);
        exit(2);
    }

    /* check header */
    if ( fromlen != sizeof(struct sockaddr_dl) ) {
        fprintf(stderr, "%s, incorrect header supplied\n", argv[0]);
        continue;
    }

    /* any data? */
    if ( ! rsize )
        fprintf(stderr, "%s, NO data received from ", argv[0]);
    else
        fprintf(stderr, "%s, data received from ", argv[0]);
    for ( i = 0; i < 6; i++ )
        fprintf(stderr, "%x%s",
            from.choose_addr.dli_eaddr.dli_target[i],
            ((i<5)? "-": " "));
    fprintf(stderr, "on protocol type %x\n",
        from.choose_addr.dli_eaddr.dli_prototype);

    /* send response to originator. */
    if ( sendto(sock, inbuf, rsize, NULL, &from, fromlen) < 0 ) {
        sprintf(outbuf, "%s: DLI transmission failed", argv[0]);
        perror(outbuf);
        exit(2);
    }
}

}

/*
 *          d l i _ e c o n n
 *
 *
 *
 *
 * Description:
 * This subroutine opens a dli socket, then binds an associated
 * device name and protocol type to the socket.
 *
 * Inputs:
 * devname      = ptr to device name
 * devunit      = device unit number
 * ptype        = protocol type
 * taddr        = target address
 * ioctl        = io control flag

```

```

*
* Outputs:
*     returns          = socket handle if success, otherwise -1
*/

dli_econn(devname, devunit, ptype, taddr, ioctl)
char *devname;
unsigned devunit;
unsigned ptype;
u_char *taddr;
u_char ioctl;
{
    int i, sock;
    struct sockaddr_dl out_bind;

    if ( (i = strlen(devname)) >
        sizeof(out_bind.dli_device.dli_devname) )
    {
        fprintf(stderr, "dli_ethd: bad device name");
        return(-1);
    }

    if ((sock = socket(AF_DLI, SOCK_DGRAM, DPROTO_DLI)) < 0)
    {
        perror("dli_ethd, can't open DLI socket");
        return(-1);
    }
    /* Fill out bind structure */

    bzero(&out_bind, sizeof(out_bind));
    out_bind.dli_family = AF_DLI;
    out_bind.dli_substructype = DLI_ETHERNET;
    bcopy(devname, out_bind.dli_device.dli_devname, i);
    out_bind.dli_device.dli_devnumber = devunit;
    out_bind.choose_addr.dli_eaddr.dli_ioctlflg = ioctl;
    out_bind.choose_addr.dli_eaddr.dli_prototype = ptype;
    if ( taddr )
        bcopy(taddr, out_bind.choose_addr.dli_eaddr.dli_target,
            DLI_EADDRSIZE);

    if ( bind(sock, &out_bind, sizeof(out_bind)) < 0 )
    {
        perror("dli_ethd, can't bind DLI socket");
        return(-1);
    }

    return(sock);
}

```

F.5.3 802.3 フォーマット・パケットを使用した DLI クライアント・プログラム例

```

#ifndef lint
static char *scsid = "@(#)dli_802.c 1.1 (DEC OSF/1) 5/29/92";
#endif lint

#include <stdio.h>
#include <ctype.h>
#include <errno.h>

```

```

#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <dli/dli_var.h>
#include <sys/ioctl.h>

extern int errno;

#define PROTOCOL_ID      {0x00, 0x00, 0x00, 0x00, 0x5}
u_char protocolid[] = PROTOCOL_ID;

/*
 *      d l i _ e x a m p l e : d l i _ 8 0 2
 *
 * Description: This program sends out a message to a system
 * where a companion program, dli_802d, echoes the message
 * back. The 802.3 packet format is used. The ethernet
 * address of the system where the companion program is
 * running, the sap, and the message are supplied by the
 * user. The companion program should be started before
 * executing this program.
 *
 * Inputs:      device, target address, sap, short message.
 *
 * Outputs:      Exit status.
 */
#ifdef lint
static char *rcsid = "@(#) $RCSfile: ap-dli.sgml,v $ \
                $Revision: 1.1.6.3 $ (DEC) $Date: 1999/07/08 20:46:48 $";
#endif

#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <dli/dli_var.h>
#include <sys/ioctl.h>

extern int errno;

#define PROTOCOL_ID      {0x00, 0x00, 0x00, 0x00, 0x5}
u_char protocolid[] = PROTOCOL_ID;

/*
 *      d l i _ e x a m p l e : d l i _ 8 0 2
 *
 * Description: This program sends out a message to a system
 * where a companion program, dli_802d, echoes the message
 * back. The 802.3 packet format is used. The ethernet
 * address of the system where the companion program is
 * running, the sap, and the message are supplied by the
 * user. The companion program should be started before
 * executing this program.
 *
 * Inputs:      device, target address, sap, short message.
 */

```

```

* Outputs:      Exit status.
*
* To compile:   cc -o dli_802 dli_802.c
*
* Example:      dli_802 qe0 08-00-2b-02-e2-ff ac "Echo this"
*
* Comments:     This example demonstrates the use of 802 "TYPE1"
*               service. With TYPE1 service, the processing of
*               XID and TEST messages is handled transparently by
*               DLI, i.e., this program doesn't have to be concerned
*               with handling them. If the SNAP SAP (0xAA) is
*               selected, a 5 byte protocol id is also required.
*               This example automatically uses a protocol id of
*               of PROTOCOL_ID when the SNAP SAP is used. Also,
*               note the use of DLI_NORMAL for the i/o control flag.
*               DLI makes use of this only when that SNAP_SAP/Protocol
*               ID pair is used. DLI will filter all incoming messages
*               by comparing the Ethernet source address and Protocol
*               ID against the target address and Protocol ID set up
*               in the bind call. Only if a match occurs will DLI
*               pass the message up to the application.
*/

/*
* Compaq Computer Corporation supplies this software
* example on an "as-is" basis for general customer use. Note
* that Compaq does not offer any support for it, nor is it
* covered under any of Compaq's support contracts.
*/

main(argc, argv, envp)
int argc;
char **argv, **envp;
{
    u_char inbuf[1500], outbuf[1500];
    u_char target_eaddr[6];
    u_char devname[16];
    int rsize, devunit;
    char *cp;
    int i, sock, fromlen;
    struct sockaddr_dl from;
    unsigned int obsiz, byteval;
    u_int sap;
    u_char *pi = 0;

    if ( argc < 5 )
    {
        fprintf(stderr, "%s %s %s\n",
            "usage:",
            argv[0],
            "device ethernet-address hex-sap short-message");
        exit(1);
    }

    /* get device name and unit number. */
    bzero(devname, sizeof(devname));
    i = 0;
    cp = argv[1];
    while ( isalpha(*cp) )
        devname[i++] = *cp++;
    sscanf(cp, "%d", &devunit);

    /* get phys addr of remote system */

```

```

bzero(target_eaddr, sizeof(target_eaddr));
i = 0;
cp = argv[2];
while ( *cp ) {
    if ( *cp == '-' ) {
        cp++;
        continue;
    }
    else {
        sscanf(cp, "%2x", &byteval );
        target_eaddr[i++] = byteval;
        cp += 2;
    }
}

/* get sap */
sscanf(argv[3], "%x", &sap);

/* get message */
bzero(outbuf, sizeof(outbuf));
if ( (obsiz = strlen(argv[4])) > 1500 ) {
    fprintf(stderr, "%s: message is too long\n", argv[0]);
    exit(2);
}
strcpy(outbuf, argv[4]);

/* open dli socket. notice that if (and only if) the */
/* snap sap was selected then a protocol id must also */
/* be provided. */
if ( sap == SNAP_SAP )
    pi = protocolid;
if ( (sock = dli_802_3_conn(devname, devunit, pi, target_eaddr,
        DLI_NORMAL, TYPE1, sap, sap, UI_NPCMD)) < 0 ) {
    perror("dli_802, dli_econn failed");
    exit(3);
}

/* send message to target. minimum message size is 46 bytes. */
if ( write(sock, outbuf, (obsiz < 46 ? 46 : obsiz)) < 0 ) {
    sprintf(outbuf, "%s: DLI transmission failed", argv[0]);
    perror(outbuf);
    exit(4);
}

/* wait for response from correct address */
while (1) {
    bzero(&from, sizeof(from));
    from.dli_family = AF_DLI;
    fromlen = sizeof(struct sockaddr_dl);
    if ((rsize = recvfrom(sock, inbuf, sizeof(inbuf),
        NULL, &from, &fromlen)) < 0 ) {
        sprintf(inbuf, "%s: DLI reception failed", argv[0]);
        perror(inbuf);
        exit(5);
    }
    if ( fromlen != sizeof(struct sockaddr_dl) ) {
        fprintf(stderr, "%s, invalid address size\n", argv[0]);
        exit(6);
    }
    if ( bcmp(from.choose_addr.dli_802addr.eh_802.dst,
        target_eaddr, sizeof(target_eaddr)) == 0 )
        break;
}

if ( ! rsize ) {

```

```

        fprintf(stderr, "%s, no data returned\n", argv[0]);
        exit(7);
    }
    /* print message */
    printf("%s\n", inbuf);

    close(sock);
}

/*
 *          d l i _ 8 0 2 _ 3 _ c o n n
 *
 *
 *
 * Description:
 *   This subroutine opens a dli 802.3 socket, then binds an
 *   associated device name and protocol type to the socket.
 *
 * Inputs:
 *   devname      = ptr to device name
 *   devunit      = device unit number
 *   ptype        = protocol type
 *   taddr        = target address
 *   ioctl        = io control flag
 *   svc          = service class
 *   sap          = source sap
 *   dsap         = destination sap
 *   ctl          = control field
 *
 *
 * Outputs:
 *   returns      = socket handle if success, otherwise -1
 *
 */

dli_802_3_conn (devname,devunit,ptype,taddr,ioctl,svc,sap,dsap,ctl)
char *devname;
u_short devunit;
u_char *ptype;
u_char *taddr;
u_char ioctl;
u_char svc;
u_char sap;
u_char dsap;
u_short ctl;

{
    int i, sock;
    struct sockaddr_dl out_bind;

    if ( (i = strlen(devname)) >
        sizeof(out_bind.dli_device.dli_devname) )
    {
        fprintf(stderr, "dli_802: bad device name");
        return(-1);
    }

    if ((sock = socket(AF_DLI, SOCK_DGRAM, DLPROTO_DLI)) < 0)
    {
        perror("dli_802, can't open DLI socket");
        return(-1);
    }

```



```

/*
 * Fill out bind structure. Note that we need to determine
 * whether the ctl field is 8 bits (unnumbered format) or
 * 16 bits (informational/supervisory format). We do this
 * by checking the low order 2 bits, which are both 1 only
 * for unnumbered control fields.
 */
bzero(&out_bind, sizeof(out_bind));
out_bind.dli_family = AF_DLI;
out_bind.dli_substructype = DLI_802;
bcopy(devname, out_bind.dli_device.dli_devname, i);
out_bind.dli_device.dli_devnumber = devunit;
out_bind.choose_addr.dli_802addr.ioctl = ioctl;
out_bind.choose_addr.dli_802addr.svc = svc;
if(ctl & 3)
    out_bind.choose_addr.dli_802addr.eh_802.ctl.U_fmt=(
        (u_char)ctl;
else
    out_bind.choose_addr.dli_802addr.eh_802.ctl.I_S_fmt = \
        ctl;
out_bind.choose_addr.dli_802addr.eh_802.ssap = sap;
out_bind.choose_addr.dli_802addr.eh_802.dsap = dsap;
if ( ptype )
    bcopy(ptype,out_bind.choose_addr.dli_802addr.eh_802.osi_pi,\
        5);
if ( taddr )
    bcopy(taddr, out_bind.choose_addr.dli_802addr.eh_802.dst,
        DLI_EADDRSIZE);
if ( bind(sock, &out_bind, sizeof(out_bind)) < 0 )
{
    perror("dli_802, can't bind DLI socket");
    return(-1);
}

return(sock);
}

```

F.5.4 802.3 フォーマット・パケットを使用した DLI サーバ・プログラム例

```
#ifndef lint
static char *rcsid = "@(#)RCSfile: ap-dli.sgml,v $ \
$Revision: 1.1.6.3 $ (DEC) $Date: 1999/07/08 20:46:48 $";
#endif

#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <dli/dli_var.h>
#include <sys/ioctl.h>

extern int errno;

#define PROTOCOL_ID {0x00, 0x00, 0x00, 0x00, 0x5}
u_char protocolid[] = PROTOCOL_ID;

/*
 * d l i _ e x a m p l e : d l i _ 8 0 2 d
 *
 * Description: This daemon program transmits any message it
 * receives to the originating system, i.e., it echoes the
 * message back. The device and sap are supplied by the
 * user. The program uses 802.3 format packets.
 *
 * Inputs:      device, sap.
 *
 * Outputs:     Exit status.
 *
 * To compile:  cc -o dli_802d dli_802d.c
 *
 * Example:     dli_802d de0 ac
 *
 * Comments: This example demonstrates the recvfrom & sendto
 * system calls. Since packets may arrive from different
 * systems we use the recvfrom call to read the packets.
 * This call gives us access to the packet header information
 * so that we can determine where the packet came from.
 * When we write on the socket we must use the sendto
 * system call to explicitly give the destination of
 * the packet. The use of the "DEFAULT" I/O control flag
 * only applies (i.e. only has an affect) when the SNAP SAP
 * is used. When the SNAP SAP is used, any arriving packets
 * which have the specified protocol id and which are not
 * destined for some other program will be given to this
 * program.
 */

/*
 * Compaq Computer Corporation supplies this software
 * example on an "as-is" basis for general customer use.
 * Note that Compaq does not offer any support for it, nor
 * is it covered under any of Compaq's support contracts.
 */
```

```

*/

main(argc, argv, envp)
int argc;
char **argv, **envp;
{
    u_char inbuf[1500], outbuf[1500];
    u_char devname[16];
    u_char target_eaddr[6];
    char *cp;
    int rsize, devunit;
    int i, sock, fromlen;
    u_char tmpsap, sap;
    struct sockaddr_dl from;
    u_char *pi = 0;

    if ( argc < 3 )
    {
        fprintf(stderr, "usage: %s device hex-sap\n", argv[0]);
        exit(1);
    }

    /* get device name and unit number. */
    bzero(devname, sizeof(devname));
    i = 0;
    cp = argv[1];
    while ( isalpha(*cp) )
        devname[i++] = *cp++;
    sscanf(cp, "%d", &devunit);

    /* get sap */
    sscanf(argv[2], "%x", &sap);

    /* open dli socket. note that if (and only if) the snap sap */
    /* was selected then a protocol id must also be specified. */
    if ( sap == SNAP_SAP )
        pi = protocolid;
    if ((sock = dli_802_3_conn(devname, devunit, pi, target_eaddr,
        DLI_DEFAULT, TYPE1, sap, sap, UI_NPCMD)) < 0) {
        perror("dli_802d, dli_conn failed");
        exit(1);
    }

    /* listen and respond */
    while ( 1 ) {
        /* wait for message */
        from.dli_family = AF_DLI;
        fromlen = sizeof(struct sockaddr_dl);
        if ((rsize = recvfrom(sock, inbuf, sizeof(inbuf), NULL,
            &from, &fromlen)) < 0 ) {
            sprintf(inbuf, "%s: DLI reception failed", argv[0]);
            perror(inbuf);
            exit(2);
        }

        /* check header */
        if ( fromlen != sizeof(struct sockaddr_dl) ) {
            fprintf(stderr, "%s, incorrect header supplied\n",
                argv[0]);
            continue;
        }
    }
}

```

```

* Note that DLI swaps the source & destination saps and
* lan addresses in the sockaddr_dl structure returned
* by the recvfrom call. That is, it places the DSAP in
* eh_802.ssap and the SSAP in eh_802.dsap; it also places
* the destination lan address in eh_802.src and the source
* lan address in eh_802.dst. This allows for minimal to
* no manipulation of the address structure for subsequent
* sendto or dli connection calls.
*/

/* any data? */
if ( ! rsize )
    fprintf(stderr, "%s: NO data received from ", \
        argv[0]);
else
    fprintf(stderr, "%s: data received from ", argv[0]);
for ( i = 0; i < 6; i++ )
    fprintf(stderr, "%x%s",
        from.choose_addr.dli_802addr.eh_802.dst[i],
        ((i<5)?"-":"" ));
fprintf(stderr, "\n      on dsap %x ",
    from.choose_addr.dli_802addr.eh_802.ssap);
if ( from.choose_addr.dli_802addr.eh_802.dsap == \
    SNAP_SAP )
    fprintf(stderr,
        "(SNAP SAP), protocol id = %x-%x-%x-%x-%x\n",
        from.choose_addr.dli_802addr.eh_802.osi_pi[0],
        from.choose_addr.dli_802addr.eh_802.osi_pi[1],
        from.choose_addr.dli_802addr.eh_802.osi_pi[2],
        from.choose_addr.dli_802addr.eh_802.osi_pi[3],
        from.choose_addr.dli_802addr.eh_802.osi_pi[4]);
fprintf(stderr, " from ssap %x ",
    from.choose_addr.dli_802addr.eh_802.dsap);
fprintf(stderr, "\n\n");

/* send response to originator. */
if ( from.choose_addr.dli_802addr.eh_802.dsap == \
    SNAP_SAP )
    bcopy(protocolid,
        from.choose_addr.dli_802addr.eh_802.osi_pi, 5);
if ( sendto(sock, inbuf, rsize, NULL, &from, fromlen) \
    < 0 ) {
    sprintf(outbuf, "%s: DLI transmission failed", \
        argv[0]);
    perror(outbuf);
    exit(2);
}
}

/*
 *      d l i _ 8 0 2 _ 3 _ c o n n
 *
 *
 *
 * Description:
 *      This subroutine opens a dli 802.3 socket, then binds an
 *      associated device name and protocol type to the socket.
 *
 * Inputs:
 *      devname      = ptr to device name
 *      devunit      = device unit number
 *      ptype        = protocol type
 *      taddr        = target address
 *      ioctl        = io control flag

```

```

*      svc          = service class
*      sap          = source sap
*      dsap         = destination sap
*      ctl          = control field
*
*
* Outputs:
*      returns      = socket handle if success, otherwise -1
*
*
*/

dli_802_3_conn (devname,devunit,ptype,taddr,ioctl,svc,sap,\
                dsap,ctl)
char *devname;
u_short devunit;
u_char *ptype;
u_char *taddr;
u_char ioctl;
u_char svc;
u_char sap;
u_char dsap;
u_short ctl;
{
    int i, sock;
    struct sockaddr_dl out_bind;

    if ( (i = strlen(devname)) >
          sizeof(out_bind.dli_device.dli_devname) )
    {
        fprintf(stderr, "dli_802d: bad device name");
        return(-1);
    }

    if ((sock = socket(AF_DLI, SOCK_DGRAM, DLPROTO_DLI)) < 0)
    {
        perror("dli_802d, can't open DLI socket");
        return(-1);
    }

    /*
     * fill out bind structure.  note that we need to determine
     * whether the ctl field is 8 bits (unnumbered format) or
     * 16 bits (informational/supervisory format).  We do this
     * by checking the low order 2 bits, which are both 1 only
     * for unnumbered control fields.
     */
    bzero(&out_bind, sizeof(out_bind));
    out_bind.dli_family = AF_DLI;
    out_bind.dli_substructype = DLI_802;
    bzero(&out_bind, sizeof(out_bind));
    out_bind.dli_family = AF_DLI;
    out_bind.dli_substructype = DLI_802;
    bcopy(devname, out_bind.dli_device.dli_devname, i);
    out_bind.dli_device.dli_devnumber = devunit;
    out_bind.choose_addr.dli_802addr.ioctl = ioctl;
    out_bind.choose_addr.dli_802addr.svc = svc;
    if(ctl & 3)
        out_bind.choose_addr.dli_802addr.eh_802.ctl.U_fmt=\
            (u_char)ctl;
    else
        out_bind.choose_addr.dli_802addr.eh_802.ctl.I_S_fmt = \
            ctl;
    out_bind.choose_addr.dli_802addr.eh_802.ssap = sap;
    out_bind.choose_addr.dli_802addr.eh_802.dsap = dsap;

```

```

        if ( ptype )
            bcopy(ptype,out_bind.choose_addr.dli_802addr.eh_802.osi_pi,\
                5);
        if ( taddr )
            bcopy(taddr, out_bind.choose_addr.dli_802addr.eh_802.dst,
                DLI_EADDRSIZE);
        if ( bind(sock, &out_bind, sizeof(out_bind)) < 0 )
        {
            perror("dli_802d, can't bind DLI socket");
            return(-1);
        }

        return(sock);
    }
}

```

F.5.5 getsockopt および setsockopt を使用した DLI プログラム例

```

#ifndef lint
static char *scsid = "@(#)dli_setsockopt.c 1.5 3/27/90";
#endif lint

#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <dli/dli_var.h>
#include <sys/ioctl.h>

extern int errno;
int debug = 0;

#define PROTOCOL_ID      {0x00, 0x00, 0x00, 0x00, 0x5}
#define CUSTOMER0        {0xab, 0x00, 0x04, 0x00, 0x00, 0x00}
#define CUSTOMER1        {0xab, 0x00, 0x04, 0x00, 0x00, 0x01}

u_char mcast0[] = CUSTOMER0;
u_char mcast1[] = CUSTOMER1;
u_char protocolid[] = PROTOCOL_ID;

/*
 *      d l i   e x a m p l e : d l i   s e t s o c k o p t
 *
 * Description: This program demonstrates the use of the DLI
 * get- and setsockopt calls. It opens a socket, enables
 * 2 multicast addresses, changes the 802 control
 * field, enables a number of group saps supplied by
 * the user, and reads the group saps that are enabled.
 *
 * Inputs:      device, sap, group-saps.
 *
 * Outputs:     Exit status.
 *
 * To compile:  cc -o dli_setsockopt dli_setsockopt.c
 */

```

```

* Example:      dli_setsockopt qe0 ac 5 9 d
*
* Comments:    When a packet arrives with a group dsap,
*              all dli programs that have that group sap enabled will
*              receive copies of that packet. Group saps are
*              those with the low order bit set. Group sap 1
*              is currently not allowed for customer use. Group
*              saps with the second bit set (eg 3,7,etc) are
*              reserved by IEEE.
*/

/*
* Compaq Computer Corporation supplies this software example
* on an "as-is" basis for general customer use. Note that
* Compaq does not offer any support for it, nor is it covered
* under any of Compaq's support contracts.
*/

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

{
    u_char inbuf[1500], outbuf[1500];
    u_char devname[16];
    u_char target_eaddr[6];
    char *cp;
    int rsize, devunit;
    int i, j, k, sock, fromlen;
    u_short obsiz;
    u_char tmpsap, sap;
    struct sockaddr_dl from;
    u_char *pi = 0;
    u_char out_opt[1000], in_opt[1000];
    int optlen, ioptlen = sizeof(in_opt);

    if ( argc < 4 )
    {
        fprintf(stderr, "usage: %s device hex-sap hex-groupsaps\n",
            argv[0]);
        exit(1);
    }

    /* get device name and unit number. */
    bzero(devname, sizeof(devname));
    i = 0;
    cp = argv[1];
    while ( isalpha(*cp) )
        devname[i++] = *cp++;
    sscanf(cp, "%d", &devunit);

    /* get protocol type */
    sscanf(argv[2], "%x", &sap);

    /* open dli socket */
    if ( sap == SNAP_SAP ) {
        fprintf(stderr,
            "%s: can't use SNAP_SAP in USER mode\n", argv[0]);
        exit(1);
    }
    if ( (sock = dli_802_3_conn(devname, devunit, pi, \
        target_eaddr, \
        DLI_DEFAULT, USER, sap, sap, UI_NPCMD)) \
        < 0 ) {

```

```

        perror("dli_setsockopt: dli_conn failed");
        exit(1);
    }

    /* enable two multicast addresses */
    bcopy(mcast0, out_opt, sizeof(mcast0));
    bcopy(mcast1, out_opt+sizeof(mcast0), sizeof(mcast1));

    if ( setsockopt(sock, DLPROTO_DLI, DLI_MULTICAST, \
        &out_opt[0],
        (sizeof(mcast0) + sizeof(mcast1))) < 0 ) {
        perror("dli_setsockopt: can't enable multicast");
    }

    /* set 802 control field */
    out_opt[0] = TEST_PCMD;
    optlen = 1;
    if
    (setsockopt(sock,DLPROTO_DLI,DLI_SET802CTL,&out_opt[0],\
        optlen)<0){
        perror("dli_setsockopt: Can't set 802 control");
        exit(1);
    }

    /* enable GSAPs supplied by user */
    j = 3;
    i = 0;
    while (j < argc ) {
        sscanf(argv[j++], "%x", &k);
        out_opt[i++] = k;
    }
    optlen = i;
    if
    (setsockopt(sock,DLPROTO_DLI,DLI_ENAGSAP,&out_opt[0],\
        optlen) < 0){
        perror("dli_setsockopt: Can't enable gsap");
        exit(1);
    }

    /* verify all gsaps are enabled */
    bzero(in_opt, (ioptlen = sizeof(in_opt)));
    if
    (getsockopt(sock,DLPROTO_DLI,DLI_GETGSAP,in_opt,\
        &ioptlen) < 0){
        perror("dli_setsockopt: DLI getsockopt 2 failed");
        exit(1);
    }
    printf("number of enabled GSAPs = %d, GSAPS:", ioptlen);
    for(i = 0; i < ioptlen; i++) {
        if ( ! (i % 10) )
            printf("\n");
        printf("%2x ",in_opt[i]);
    }
    printf("\n");

    /* disable all but the last 4 or all GSAPs, */
    /* whichever is smallest */
    if ( optlen > 4 )
        optlen -= 4;
    if
    (setsockopt(sock,DLPROTO_DLI,DLI_DISGSAP,&out_opt[0],\
        optlen) < 0){
        perror("dli_setsockopt: Can't disable gsap");
    }
}

```



```

/* verify some gsaps still enabled */
bzero(in_opt, (ioptlen = sizeof(in_opt)));
if
(getsockopt(sock,DLPROTO_DLI,DLI_GETGSAP,in_opt,\
            &ioptlen) < 0){
    perror("dli_setsockopt: getsockopt 3 failed");
    exit(1);
}
printf("number of enabled GSAPs = %d, GSAPS:", ioptlen);
for(i = 0; i < ioptlen; i++) {
    if ( ! (i % 10) )
        printf("\n");
    printf("%2x ",in_opt[i]);
}
printf("\n");
}

/*
 *          d l i _ 8 0 2 _ 3 _ c o n n
 *
 *
 *
 * Description:
 * This subroutine opens a dli 802.3 socket and then binds
 * an associated device name and protocol type to it.
 *
 * Inputs:
 * devname    = ptr to device name
 * devunit    = device unit number
 * ptype      = protocol type
 * taddr      = target address
 * ioctl      = io control flag
 * svc        = service class
 * sap        = source sap
 * dsap       = destination sap
 * ctl        = control field
 *
 *
 * Outputs:
 * returns    = socket handle if success, otherwise -1
 *
 */

dli_802_3_conn (devname,devunit,ptype,taddr,ioctl,svc,sap,\
                dsap,ctl)
char *devname;
u_short devunit;
u_char *ptype;
u_char *taddr;
u_char ioctl;
u_char svc;
u_char sap;
u_char dsap;
u_short ctl;
{
    int i, sock;
    struct sockaddr_dl out_bind;

    if ( (i = strlen(devname)) >
        sizeof(out_bind.dli_device.dli_devname) )
    {
        fprintf(stderr, "dli_setsockopt: bad device name");
        return(-1);
    }
}

```

```

if ((sock = socket(AF_DLI, SOCK_DGRAM, DLPROTO_DLI)) < 0)
{
    perror("dli_setsockopt: can't open DLI socket");
    return(-1);
}

/*
 * Fill out bind structure
 */
bzero(&out_bind, sizeof(out_bind));
out_bind.dli_family = AF_DLI;
out_bind.dli_substructype = DLI_802;
bcopy(devname, out_bind.dli_device.dli_devname, i);
out_bind.dli_device.dli_devnumber = devunit;
out_bind.choose_addr.dli_802addr.ioctl = ioctl;
out_bind.choose_addr.dli_802addr.svc = svc;
if(ctl & 3)
    out_bind.choose_addr.dli_802addr.eh_802.ctl.U_fmt=(\
        (u_char)ctl;
else
    out_bind.choose_addr.dli_802addr.eh_802.ctl.I_S_fmt = \
        ctl;
out_bind.choose_addr.dli_802addr.eh_802.ssap = sap;
out_bind.choose_addr.dli_802addr.eh_802.dsap = dsap;
if ( ptype )
    bcopy(ptype, out_bind.choose_addr.dli_802addr.eh_802.osi_pi, \
        5);
if ( taddr )
    bcopy(taddr, out_bind.choose_addr.dli_802addr.eh_802.dst,
        DLI_EADDRSIZE);
if ( bind(sock, &out_bind, sizeof(out_bind)) < 0 )
{
    perror("dli_setsockopt: can't bind DLI socket");
    return(-1);
}

return(sock);
}

```

用語集

BSD ソケット・インタフェース

単一のシステム，または複数の接続されたシステム上の 2 つの関連のないプロセス間でプロセス間通信を実行する，アプリケーション用に用意されたトランスポート層インタフェース。このプロセス間通信機能により，プログラムはソケットを使用して，他のプログラム，プロトコルおよびデバイスと通信することができる。

ETSDU

優先トランスポート・サービス・データ・ユニットおよび帯域外データの項を参照。

eSNMP

eSNMP (Extensible Simple Network Protocol) を使用すると，SNMP 管理ステーションによって管理されるサブエージェントを作成することができる。第 6 章を参照。

ICMP

インターネット制御メッセージ・プロトコルの項を参照。

`#include file.h`

指定したファイルを，コンパイルしているファイルへ挿入することを指定する C 言語のプリコンパイラ指示文。挿入するファイルは，(名前を山カッコで囲むことによって示す) 標準のヘッダ・ファイル，または(名前を二重引用符で囲むことによって示す) C 言語コードを含む任意のファイルである。たとえば，次のように指定する。

```
#include <header_file.h>
#include "myfile.c"
```

名前を山カッコ (<>) で囲んだヘッダ・ファイルの絶対パス名は，`/usr/include/file.h` である。

IP マルチキャスト

IP データグラムを，単一の IP デスティネーション・アドレス，つまりホスト・グループによって識別されるホストのグループへ伝送する方式。ホ

スト・グループはクラス D IP アドレスによって識別される。ホスト・グループの項も参照。

ISO

国際標準化機構 の項を参照。

MIB

MIB (Manegement Information Base) 定義は、ネットワーク管理に関するデータ要素の集合です。第 6 章参照。

OSI

開放型システム間相互接続の項を参照。

OID

OID (Object Identifier) は、名前あるいはシーケンス番号によって参照される MIB 定義におけるデータ要素です。第 6 章を参照。

raw ソケット

特権ユーザに、内部ネットワーク・プロトコルおよびインタフェースへのアクセスを提供するソケット。このソケット・タイプを使用すると、通常のインタフェースでは使用不可能なプロトコル機能を利用したり、ハードウェア・インタフェースと通信したりできる。

SLIP

シリアル回線インターネット・プロトコルの項を参照。

STREAMS

AT&T が規定したカーネル機構であり、デバイス・ドライバおよびネットワーク・プロトコル・スタックのインプリメンテーションをサポートする。

STREAMS フレームワークの項も参照。

STREAMS フレームワーク

AT&T STREAMS 機構の構成要素であり、カーネル内およびカーネルとユーザ・レベル間で文字 I/O 用のインタフェース標準を定義する。関数、ユーティリティ・ルーチン、カーネル機能、およびデータ構造体から構成される。

TCP

伝送制御プロトコルの項を参照。

TCP/IP

インターネット・プロトコル群の 2 つの基本プロトコルであり、インターネット・プロトコル群の参照に頻繁に使用される頭文字。TCP は確実なデータ転送を提供するのに対し、IP はデータグラム形式で、ネットワークを介してデータを送信する。伝送制御プロトコルおよびインターネット・プロトコルの項も参照。

TLI

トランスポート層インタフェースの項を参照。

TSDU

トランスポート・サービス・データ・ユニットの項を参照。

UDP

ユーザ・データグラム・プロトコルの項を参照。

X/Open トランスポート・インタフェース

プロトコルに依存しないアプリケーション用のトランスポート層インタフェース。XTI は、TLI に基づく C 言語関数のセットから構成される。TLI は、OSI モデルのトランスポート・サービス定義に基づいている。

XTI

X/Open トランスポート・インタフェースの項を参照。

アクティブ・ユーザ

XTI トランスポート接続において、接続を開始したトランスポート・ユーザ。クライアント・プロセスおよびパッシブ・ユーザの項も参照。

アドレス解決プロトコル (ARP)

インターネット・アドレスを、物理ハードウェア・アドレスに動的に変更することができるインターネット (TCP/IP) プロトコル。ARP は、単一の物理ネットワーク、およびハードウェア・ブロードキャスト機能をサポートする複数のネットワークでのみ使用できる。

イーサネット

衝突検出機能付きキャリア検知多重アクセス (CSMA/CD) を使用する、10 Mビットのベースバンドのローカル・エリア・ネットワーク (LAN)。このネットワークでは、事前に調整しなくても、複数のステーションが自由にメディアにアクセスすることができる。また、キャリア検知と延期、および検出と転送を使用して、送信権の争奪を回避する。

イベント

トランスポート・ユーザにとって意味のある出現事象または偶発事象。イベントは、ユーザが行ったアクションの結果として起こらないという点で、非同期である。

イベント管理

トランスポート・プロバイダがトランスポート・ユーザに重要なイベントが起こったことを通知するメカニズム。

インターネット制御メッセージ・プロトコル (ICMP)

インターネット・プロトコル (IP) のオペレーションに関する、エラー・メッセージや情報メッセージを提供する、インターネット・プロトコル群の中のホスト間プロトコル。

インターネット・プロトコル (IP)

ネットワーク上でデータグラムの転送用にコネクションレス・サービスを提供するインターネット・プロトコル。

エラー

XTI において、実行時にシステム・エラーまたはライブラリ・エラーを検出すると、関数によって返されるインディケータ。これは、アプリケーションが返されたエラー・コードに基づいてアクションをとるためのものである。

オブジェクト識別子

OID を参照。

開放型システム間相互接続 (OSI)

ISO 規格に準拠した、開放型システムの相互接続。

管理情報ベース

MIB を参照。

クライアント・プロセス

通信のクライアント/サーバ・モデルにおいて、サーバ・プロセスからサービスを要求するプロセスのこと。アクティブ・ユーザの項も参照。

国際標準化機構 (ISO)

89 カ国の標準化機構から構成される国際的な組織。ISO は、ネットワーキング・ソフトウェアをはじめ、膨大な数の商品およびサービスに関する規格を制定している。

コネクション指向モード

立された接続を通してデータを送信するために、トランスポート終端によってサポートされるサービス・モード。

コネクションレス・モード

データ伝送用に特定の接続を必要としないトランスポート終端によってサポートされるサービス・モード。データは、データグラムと呼ばれる自己完結型ユニットで配信される。

サーバ・プロセス

通信のクライアント/サーバ・モデルにおいて、クライアント・プロセスに対してサービスを提供するプロセス。パッシブ・ユーザの項も参照。

状態

XTI では、実行中の関数を反映した現在のプロセスの状態。XTI は、8 つの異なる状態を使用して、トランスポート終端上の通信を管理する。

シリアル回線インターネット・プロトコル (SLIP)

非同期シリアル回線を介し、IP データグラムをカプセル化して転送する伝送回線プロトコル。

ストリーム・ソケット

トランスポート接続間に双方向のバイト・ストリームを提供するソケット。帯域外データを処理するための機構も含む。

正常解放

XTI において、トランスポート・ユーザがデータを損失することなく、緩やかにトランスポート接続を終了できるようにするオプション機能。

ソケット

プロセス間通信における通信の終端。または、ソケットおよびそれに関連するデータ構造体を作成するシステム・コール。

ソケットペア

UNIX ドメインで作成できる双方向通信用のソケットのペア。ソケットペアは、パイプと同様に、関連付けられた通信プロセスが必要である。パイプの項も参照。

帯域外データ

緊急時に、通常のデータ・フローとは別に伝送されるデータ。このデータが取り出せるように、受信プロセスにこのデータの存在が通知される。

通常データ

トランスポート・ユーザにより、帯域内において送信または受信される標準データ。帯域外データの項も参照。

通信ドメイン

ネットワークのプロパティを定義するシステムのプロセス間通信機能によって使用される抽象的概念。プロパティには、通信プロトコル、名前の操作および解釈の規則、アクセス権の伝送能力が含まれる。

データグラム

トランスポート・プロバイダのコネクションレス・サービスによって、ネットワークを介して送信されるデータ・ユニット。データグラムには、ユーザ・データに加えて伝送に必要な情報も含まれる。前後に伝送されるデータグラムと関係がないという点で、データグラムは自己完結型である。

データグラム・ソケット

コネクションレス・モードにおいて、転送用の各メッセージを構成するデータグラムを提供するソケット。

伝送制御プロトコル (TCP)

アプリケーションに対して、信頼性の高い全二重のコネクション指向サービスを提供するインターネット・トランスポート層プロトコル。TCP は、IP プロトコルを使用し、ネットワークを介して情報を送信する。

同期実行

トランスポート・ユーザに制御を戻す前に、トランスポート・プリミティブに指定のイベントを待機させる実行モード。

トランスポート・サービス

ユーザ・プロセス間のデータ転送のために、システムのトランスポート層がアプリケーション層に与えたサポート。提供されるサービスは、コネクション指向型およびコネクションレス型の2種類である。トランスポート層インタフェースの項も参照。

トランスポート・サービス・データ・ユニット (TSDU)

OSI の用語で、トランスポート・ユーザがトランスポート・プロバイダに渡す情報の項目またはメッセージ。

トランスポート・プロバイダ

システムで、トランスポート層のサービスを提供する伝送プロトコル。

トランスポート・ユーザ

別のプログラムまたはネットワーク上のポイントとの間でデータを送受信するために、伝送プロトコルのサービスを必要とするプログラム。トランスポート層インタフェースの項も参照。

トランスポート終端

トランスポート・ユーザが、トランスポート・プロバイダとデータをやりとりできる通信パス。トランスポート層インタフェースの項も参照。

トランスポート層インタフェース (TLI)

OSI モデルのトランスポート層へのインタフェースであり、ISO トランスポート・サービス定義に基づいて設計されている。

ネーム・サーバ

クライアント・プロセスが、ホスト・アドレスまたはネットワーク内の他のオブジェクトのアドレスを取得するために交信するシステム上で実行しているデーモン。このデーモンは、マシンのネットワーク名をそのネットワークの IP アドレスに変換する。

ネーム・サービス

クライアント・プロセスに提供されている、通信のための同等システムを識別するサービス。

パイプ

記述子を持つ I/O ストリームであり、関連付けられプロセス間の単方向の通信に使用される。ソケットペアの項も参照。

バークレイ版ソフトウェア

カリフォルニア大学バークレイ校のコンピュータ・システム研究グループ (CSRG: Computer System Research Group) がリリースしている UNIX ソフトウェアのこと。

パッシブ・ユーザ

XTI トランスポート接続において、接続要求に応答した対等トランスポート・ユーザ。アクティブ・ユーザおよびクライアント・プロセスの項も参照。

非同期イベント

イベントの項を参照。

非同期実行

1. すでに開始されている処理またはスレッドの終了を待たずに処理またはスレッドを実行すること。
2. XTI では、トランスポート・ユーザを待たせることなくイベントを通知する実行モード。

非ブロッキング・モード

非同期実行の項を参照。

ファイル記述子

UNIX システムがファイルを識別するために使用する、符号なしの小さな整数。ファイル記述子は、ファイル名に対してオープン・システム・コールを発行するプロセスにより作成される。ファイル記述子は、それを保持するプロセスがなくなったときに消滅する。

ブロッキング・モード

同期実行の項を参照。

ホスト・グループ

IP マルチキャストのため、単一のクラス D IP デスティネーション・アドレスによって識別される 0 個以上のホストから構成されるグループ。クラス D IP アドレスは、上位 4 ビットに 1110 を持つ。IP マルチキャストの項も参照。

優先データ

緊急と考えられるデータ。このデータの意味は、トランスポート・プロバイダによって定義される。帯域外データの項も参照。

優先トランスポート・サービス・データ・ユニット

XTI において、データ・ユニットの ID が、トランスポート接続の一端からもう一方の端まで保持される優先メッセージ。

ユーザ・データグラム・プロトコル (UDP)

リモート・マシン上のアプリケーション・プログラムとの間で、データグラムを相互に送信できるようにするインターネット・プロトコル。UDP は、IP を使用してデータグラムを引き渡す。



数字および記号

802.3 サブ構造体

代入 F-23e

802.3 サブ構造体値

LLC プロトコル・データ・ユニッ

ト・テスト F-17

XID F-16

交換識別 F-16

コントロール・フィールド... F-15

サービス・クラス..... F-13

ソース・サービス・アクセス・ポ
イント F-15デスティネーション・サービス・ア
クセス・ポイント F-14デスティネーション・システムの物
理アドレス F-13

非番号情報コマンド..... F-17

802.3 フレーム・フォーマット

処理 F-13

説明 F-12

例..... F-5

A**accept1** イベント 3-18t**accept2** イベント 3-18t**accept3** イベント 3-18t**accept** システム・コール 4-11t**accept** ソケット呼び出しXTI `t_accept` 関数との相違 ... 3-45**AF_INET6** ソケットアプリケーションでの使用.. 4-41,
4-54**AF_INET6** ドメイン 4-4**AF_INET** ソケット

アプリケーションでの使用... 4-39

AF_INET ドメイン 4-4**AF_UNIX**

(UNIX ドメイン を参照)

AF_UNIX ドメイン 4-4

(UNIX ドメイン も参照)

all hosts グループ

定義 4-81

B**big-endian**

定義 4-16

bind イベント 3-17t**bind** システム・コール 4-11t,
4-28, F-7

構文 F-21

bind ソケット・コールXTI `t_bind` 関数との相違..... 3-45**BSD** ソケット..... 4-49

とネットワーク・アドレス... 4-50

BSD ソケット・インタフェース

4.3BSD msghdr データ構造体 4-51
 4.4BSD msghdr データ構造体 4-52
 raw ソケット 4-6
 ストリーム・ソケット 4-6
 ソケット・オプションの使用 4-32
 ソケットへの接続の確立 4-28
 ソケットへの名前のバインド 4-28
 データグラム・ソケット 4-5
 データの転送 4-33
 ブロッキングおよび非ブロッキング
 のオペレーション 4-26
BSD ドライバ
 STREAMS プロトコル・スタックへ
 のブリッジ 8-11

C

closed イベント 3-17t
close 関数 5-9
close システム・コール 4-38
close 処理 5-20
close ソケット呼び出し
 XTI `t_snddis` 関数との相違... 3-46
CLTS
 (XTI におけるコネクションレ
 ス・サービス を参照)
cmsghdr データ構造体 4-23
connect1 イベント 3-18t
connect2 イベント 3-18t
connect システム・コール 4-11t
 TCP の使用 4-29
 UDP の使用 4-29
COTS
 (コネクション指向型トランス
 ポート・サービス を参照)

D

datagram ソケット F-19
dblk_t データ構造体 5-18
DL_ATTACH_REQ プリミティ
 ブ 2-9, 8-12
DL_BIND_ACK プリミティブ . 2-9,
 8-12
DL_BIND_REQ プリミティブ . 2-9,
 8-12
DL_DETACH_REQ プリミティ
 ブ 8-12
DL_DETTACH_REQ プリミティ
 ブ 2-9
DL_DISABLMULTI_REQ プリミ
 ティブ 8-12
DL_DISABMULTI_REQ プリミティ
 ブ 2-9
DL_ENABMULTI_REQ プリミティ
 ブ 2-9, 8-12
DL_ERROR_ACK プリミティブ 2-9
DL_ETHER メディア 8-13
DL_INFO_ACK プリミティブ .. 2-9
DL_INFO_REQ プリミティブ .. 2-9
DL_OK_ACK プリミティブ 2-9,
 8-12
DL_PHYS_ADDR_ACK プリミティ
 ブ 2-9, 8-12
DL_PHYS_ADDR_REQ プリミティ
 ブ 8-12
DL_PROMISCON_REQ プリミティ
 ブ 8-12
DL_PROMISCONOFF_REQ プリミ
 ティブ 8-12

DL_SET_PHYS_ADDR_REQ プリミ
 タイプ 8-12
DL_SUBS_BIND_ACK プリミティ
 ブ..... 2-9, 8-12
DL_SUBS_BIND_REQ プリミティ
 ブ..... 2-9, 8-12
DL_SUBS_UNBIND_ACK プリミ
 タイプ 8-12
DL_SUBS_UNBIND_REQ プリミ
 タイプ 2-9, 8-12
DL_TEST_CON プリミティブ .. 2-9
DL_TEST_IND プリミティブ... 2-9
DL_TEST_REQ プリミティブ .. 2-9
DL_TEST_RES プリミティブ .. 2-9
DL_UDERROR_IND プリミティ
 ブ..... 2-9
DL_UNBIND_REQ プリミティ
 ブ..... 2-9, 8-12
DL_UNIDATA_IND プリミティ
 ブ..... 2-9
DL_UNIDATA_REQ プリミティ
 ブ..... 2-9
DL_UNITDATA_IND プリミティ
 ブ..... 8-12
DL_UNITDATA_REQ プリミティ
 ブ..... 8-12
DL_XID_CON プリミティブ 2-9
DL_XID_IND プリミティブ 2-9
DL_XID_REQ プリミティブ 2-9
DL_XID_RES プリミティブ 2-9
dlb STREAMS 擬似ドライバ . 1-10,
 2-2, 8-11
dlb STREAMS ドライバ..... 8-1

DLI

sockaddr_dl 構造体への代入 . F-21
 概念 F-1
 サービス..... F-3
 システム・コール..... F-18
 ソケット・オプションの設定 F-20
 ソケットの作成 F-19
 ソケットのバインド..... F-21
 ソケットの非アクティブ化... F-25
 定義 F-1
 データ転送..... F-24
 と伝送 IEEE 802.3 フレーム .. 2-2
 バッファ・サイズの計算 F-24
 プログラミング例..... F-25
 プログラムの作成..... F-18
 ローカル・エリア・ネットワークへ
 のアクセス F-4
dli_802_3_conn サブルーチン
 使用 F-8
 例..... F-42
dli_econn サブルーチン
 使用 F-8
 例..... F-29
DLI アドレス・ファミリ
 指定 F-19
DLI クライアント・プログラム
 802.3 フォーマット・パケットの
 使用
 例 F-32
 イーサネット・フォーマット・パ
 ケットの使用
 例 F-26
DLI サーバ・プログラム

802.3 フォーマット・パケットの使用	
例	F-38
イーサネット・パケットの使用	
例	F-29
DLI サービス	
例	F-3
DLI プログラム	
getsockopt および setsockopt の使用	
例	F-42
高次レベル・サービスのインクルード	F-4
DLI プロトコル・モジュール	
指定	F-19
DLPI	
DLS プロバイダ	2-8
STREAMS ドライバがサポートしなければならないプリミティブ	8-10
アドレッシング	2-7
PPA	2-8
オンラインの仕様へのアクセス	2-1
肯定応答コネクションレス・モード・サービス	2-7
コネクション・モード・サービス	2-5
コネクションレス・モード・サービス	2-6
サポートされているプリミティブ	2-9, 8-12
表	2-9
サポートされているメディア	
DL_ETHER	8-13
サービスの種類	2-4
通信の肯定応答コネクションレス・モード	2-4
通信のコネクションレス・モード	2-3
通信の接続モード	2-3
通信モード	2-3
定義	2-2
と DLS プロバイダ	2-2
と DLS ユーザ	2-2
ローカル管理サービス	2-5
DLPI アドレッシング	
構成要素の識別	2-8
DLPI インタフェース	2-1
DLPI オプション	8-4
カーネル構成ファイルへの追加	8-4
DLPI サービス・インタフェース	2-3
DLPI におけるアドレッシング	2-7
PPA	2-8
構成要素の識別	2-8
DLPI におけるデータ・リンク・サービス・プロバイダ	2-8
DLPI プリミティブ	
Tru64 UNIX でサポート	2-9
DLS プロバイダ	
定義	2-2
DLS ユーザ	
定義	2-2
DSAP	
定義	F-14
E	
EAFNOSUPPORT ソケット・エラー	4-53
EBADF ソケット・エラー	4-53

ECONNREFUSED ソケット・エラー 4-53
EFAULT ソケット・エラー 4-53
EHOSTDOWN ソケット・エラー 4-53
EHOSTUNREACH ソケット・エラー 4-53
EINVAL ソケット・エラー 4-53
EMFILE ソケット・エラー 4-53
endhostent ライブラリ・コール 4-16
endnetent ライブラリ・コール 4-16
endprotoent ライブラリ・コール 4-16
endservent ライブラリ・コール 4-16
ENETDOWN ソケット・エラー 4-53
ENETUNREACH ソケット・エラー 4-53
ENOMEM ソケット・エラー .. 4-53
ENOTSOCK ソケット・エラー 4-53
EOPNOTSUPP ソケット・エラー 4-53
EPROTONOSUPPORT ソケット・エラー 4-53
EPROTOTYPE ソケット・エラー 4-53
eSNMP 1-8
 MIB サブツリー 6-8
 SNMP バージョン 6-4
 subtree_tbl.c ファイル 6-13
 subtree_tbl.h ファイル 6-10
 値の表現 6-63

アプリケーション・インタフェース 6-5
 アプリケーション・プログラミング・インタフェース・ルーチン 6-24
 アーキテクチャ 6-3
 オブジェクト・テーブル 6-10
 概要 1-8, 6-2
 起動 6-20
 関数戻り値 6-21
 構成要素 6-2
 サブエージェントの実現 6-15
 サポート・ルーチン 6-65
 停止 6-20
 関数戻り値 6-21
 メソッド・ルーチン 6-61
 メソッド・ルーチン呼び出しインタフェース 6-52
 呼び出しインタフェース 6-24
eSNMP アプリケーション・プログラミング・インタフェース
 (eSNMP を参照)
ETIMEDOUT ソケット・エラー 4-53
EWouldBLOCK ソケット・エラー 4-53

F

F_GETOWN パラメータ 4-98
F_SETOWN パラメータ 4-98
fattach ライブラリ・コール ... 5-13
fcntl.h ファイル 3-8
fcntl システム・コール

F_GETOWN パラメータ 4-98
F_SETOWN パラメータ 4-98
FDDI
 アクセス F-4
 ソース・サービス・アクセス・ポイント F-15
 フレーム・フォーマット F-5
fdetach ライブラリ・コール... 5-13
fd 変数
 発信イベントにおける使用... 3-17
freehostent ライブラリ・コール 4-16
freehostent ルーチン 4-12

G

getaddrinfo ライブラリ・コール 4-13, 4-16
gethostbyaddr ライブラリ・コール 4-16
gethostbyaddr ルーチン 4-12
 IPv6 4-60
gethostbyname ライブラリ・コール 4-16
gethostbyname ルーチン 4-12
 IPv6 4-60
gethostent ライブラリ・コール 4-16
getipnodebyaddr ライブラリ・コール 4-16
getipnodebyname ライブラリ・コール 4-16
getmsg 関数 5-12
getnameinfo ライブラリ・コール 4-13, 4-16

getnetbyaddr ライブラリ・コール 4-16
getnetbyaddr ルーチン 4-14
getnetbyname ライブラリ・コール 4-16
getnetbyname ルーチン 4-14
getnetent ライブラリ・コール 4-16
getnetent ルーチン 4-14
getpeername システム・コール 4-11t
getpmsg 関数 5-12
getprotobyname ライブラリ・コール 4-16
getprotobyname ルーチン 4-14
getprotobynumber ライブラリ・コール 4-16
getprotobynumber ルーチン . 4-14
getprotoent ライブラリ・コール 4-16
getprotoent ルーチン 4-14
getservbyname ライブラリ・コール 4-16
getservbyname ルーチン 4-15
getservbyport ライブラリ・コール 4-16
getservbyport ルーチン 4-15
getservent ライブラリ・コール 4-16
getservent ルーチン 4-15
getsockname システム・コール 4-11t

H

hostent 構造体 4-59
hostent データ構造体 4-13

htonl ライブラリ・コール 4-16
htons ライブラリ・コール 4-16

I

I/O コントロール・フラグ

機能 F-11

ifnet STREAMS モジュール... 1-9,
8-2

使用 8-4

必要な設定..... 8-4

in_addr 構造体 4-57

IN6_ADDR_EQUAL マク

□ 4-48, 4-63

IN6_IS_ADDR_LINKLOCAL マク

□ 4-48

IN6_IS_ADDR_LOOPBACK マク

□ 4-48

IN6_IS_ADDR_MC_GLOBAL マク

□ 4-48

IN6_IS_ADDR_MC_LINKLOCAL

マク □ 4-48

IN6_IS_ADDR_MC_NODELOCAL

マク □ 4-48

IN6_IS_ADDR_MC_ORGLOCAL マ

ク □ 4-48

IN6_IS_ADDR_MC_SITELOCAL マ

ク □ 4-48

IN6_IS_ADDR_MULTICAST マク

□ 4-48

IN6_IS_ADDR_SITELOCAL マク

□ 4-48

IN6_IS_ADDR_UNSPECIFIED マ

ク □ 4-48, 4-63

IN6_IS_ADDR_V4COMPAT マク

□ 4-48

IN6_IS_ADDR_V4MAPPED マク

□ 4-48, 4-61

in6addr_any ワイルドカード・アド
レス

名前のアドレスへのバインド 4-75

INADDR_ANY ワイルドカード・ア
ドレス

名前のアドレスへのバインド 4-75

inet_addr ライブラリ・コール 4-16

inet_addr ルーチン

IPv6 4-61

inet_lnaof ライブラリ・コール 4-16

inet_makeaddr ライブラリ・コー

ル 4-16

inet_netof ライブラリ・コール 4-16

inet_network ライブラリ・コー

ル 4-16

inet_ntoa ライブラリ・コール. 4-16

inet_ntoa ルーチン

IPv6 4-61

inetd デーモン 4-93

ioctl 関数 5-10

IP_ADD_MEMBERSHIP 4-84

IP_DROP_MEMBERSHIP 4-84

IP_MULTICAST_IF 4-83

IP_MULTICAST_LOOP 4-83

IP_MULTICAST_TTL 4-82

IPv4 マルチキャスト・データグラム

の送信 4-81

IPv6

デスティネーション・オプショ

ン 4-71

パケット・ヘッダ 4-67
 ホップ・バイ・ホップ・オブショ
 ン 4-71
 ルーティング・ヘッダ 4-69
IPV6_JOIN_GROUP 4-88
IPV6_LEAVE_GROUP 4-89
IPV6_MULTICAST_HOPS.... 4-86
IPV6_MULTICAST_IF 4-87
IPV6_MULTICAST_LOOP.... 4-87
IPv6 マルチキャスト・データグラム
 の送信 4-85
IP マルチキャスト 4-80
 all hosts グループ 4-81
 データグラム (IPv4) の受信 .. 4-84
 データグラム (IPv4) の送信 .. 4-81
 データグラム (IPv6) の送信 .. 4-85
 データグラムの受信..... 4-88
 マルチキャスト・グループ... 4-80
isastream ライブラリ・コール 5-12

L

libtli.a ライブラリ 3-7
libxti.a ライブラリ 3-7
listen イベント 3-19
listen システム・コール 4-11t
LLC
 DLI 補助層 F-13
LLC プロトコル・データ・ユニット・
 テスト
 機能 F-17
 定義 F-17

M

mblk_t データ構造体 5-18

MIB サブツリー

eSNMP 6-8
mkfifo 関数 5-10
module_info データ構造体 5-16
msghdr データ構造体..... 4-22,
 4-23, 4-51
recvmsg システム・コールでの使
 用 4-36
sendmsg システム・コールでの使
 用 4-36
 サポートされるタイプ 4-23

N

netdb.h ヘッダ・ファイル 4-12
netent データ構造体..... 4-14
netinet/in.h ヘッダ・ファイル 4-19
ntohl ライブラリ・コール 4-16
ntohs ライブラリ・コール 4-16

O

O_NDELAY 値

 TLI におけるサポート 3-42
ocnt 変数 3-20
 着信イベントにおける使用... 3-18
 発信イベントにおける使用... 3-17
opened イベント 3-17t
open 関数..... 5-9
open 処理..... 5-20
optmgmt イベント 3-17t

P

pass_conn イベント 3-18
PAWS オプション D-4

pipe 関数 5-11
poll 関数 5-12
 XTI アプリケーションにおける使
 用 3-15
PPA
 DLPI におけるアドレッシング 2-8
 定義 2-8
protoent データ構造体 4-14
putmsg 関数 5-11
putpmsg 関数 5-11

Q

qinit データ構造体 5-16
QoS
 QoS 構成要素の動作 7-3
 構成要素 7-2
 定義 7-1

R

RAPI 7-8
 アプリケーションのテスト... 7-10
 アプリケーションのデバッグ 7-10
 サポート 7-1
 ルーチン 7-8
raw ソケット 4-6
rcvconnect イベント 3-19
rcvdis1 イベント 3-19
rcvdis3 イベント 3-19
rcvrel イベント 3-19
rcvudata イベント 3-19
rcvuderr イベント 3-19
rcv イベント 3-19

read 関数 5-10
read システム・コール 4-34
recvfrom システム・コール . 4-11t,
 4-35
recvmsg システム・コール .. 4-11t,
 4-37
 msghdr データ構造体の使用 . 4-36
recv システム・コール. 4-11t, 4-35
resfd 変数
 発信イベントにおける使用... 3-17
Resource ReSerVation Protocol
 (RSVP を参照)

RSVP

 アプリケーションでの実現.... 7-9
 アプリケーション・プログラミン
 グ・インタフェース 7-8
 概要 7-6
 構成要素 7-7
 システムの役割 7-3
rsvpd デーモン 7-7
rsvpstat コマンド 7-10
RSVP アプリケーション・プログラミ
 ング・インタフェース 7-8
 (RAPI も参照)

S

sa_family 4-21
SACK オプション D-3
select ソケット呼び出し
 XTI t_look 関数との相違 3-45
sendmsg システム・コール.. 4-11t,
 4-36
 msghdr データ構造体の使用 . 4-36

sendto システム・コール 4-11t, 4-35, F-7
send システム・コール 4-11t, 4-35
servent データ構造体..... 4-15
sethostent ライブラリ・コール 4-16
setnetent ライブラリ・コール 4-16
setprotoent ライブラリ・コール..... 4-16
setservent ライブラリ・コール 4-16
setsockopt システム・コール 4-11t
 IP_ADD_MEMBERSHIP オプション 4-84
 IP_DROP_MEMBERSHIP オプション 4-84
 IP_MULTICAST_IF オプション 4-83
 IP_MULTICAST_LOOP オプション 4-83
 IP_MULTICAST_TTL オプション 4-82
 IPV6_JOIN_GROUP オプション 4-88
 IPV6_LEAVE_GROUP オプション 4-89
 IPV6_MULTICAST_HOPS オプション 4-86
 IPV6_MULTICAST_IF オプション 4-87
 IPV6_MULTICAST_LOOP オプション 4-87
 SO_REUSEPORT オプション 4-85, 4-89
shutdown システム・コール . 4-11t
SNAP_SAP
 使用 F-15
snddis1 イベント 3-18t
snddis2 イベント 3-18t
sndrel イベント 3-18t
sndudata イベント 3-18t
snd イベント..... 3-18t
SNMP 6-1
 (eSNMP も参照)
 サポートされるバージョン 6-4
SO_REUSEPORT..... 4-85, 4-89
SOCK_DGRAM ソケット..... 4-6
SOCK_RAW ソケット 4-6
SOCK_STREAM ソケット..... 4-6
sockaddr_dl データ構造体 F-7
 イーサネット・サブ構造体 F-8
 および 802.2 サブ構造体 F-12
 代入 F-7
 例..... F-5
sockaddr_in6 データ構造体 ... 4-22
sockaddr_in データ構造体 4-22
sockaddr_storage データ構造体..... 4-21, 4-58
sockaddr_un データ構造体 4-21
sockaddr データ構造体. 4-20, 4-58
 4.3BSD と 4.4BSD の比較 4-50
socketpair システム・コール 4-11t, 4-25
socket システム・コール 4-11t
SSAP
 定義 F-15
strclean コマンド 5-32
STREAMS
 close 関数 5-9
 ifnet STREAMS モジュールの使用 8-4

ifnet STREAMS モジュールを使用 するために必要な設定	8-4	ドライバ処理ルーチン	5-19
ioctl 関数	5-10	ドライバの組み込み	5-25
mkfifo 関数	5-10	ヘッダ・ファイル	5-8
open 関数	5-9	メッセージ	5-6
pipe 関数	5-11	メッセージ・データ構造体 ...	5-17
putmsg 関数	5-11	モジュール処理ルーチン	5-19
putpmsg 関数	5-11	モジュール・データ構造体 ...	5-16
read 関数	5-10	モジュールの組み込み	5-25
write 関数	5-10	モジュール例	A-1
アプリケーション・プログラミン グ・インタフェース	5-8	ライブラリ・コール ...	5-12, 5-13
イベント・ロギング	5-32	STREAMS 概念	5-23
strclean コマンド	5-32	STREAMS 擬似ドライバ	8-13
エラー・ロギング	5-32	STREAMS ドライバ	8-3
関数	5-9	ソケット・プロトコル・スタックへ のブリッジ	8-2
カーネル・レベル関数	5-16	STREAMS フレームワーク 1-4, 5-2	
クローン・デバイス	5-31	XTI との関係	1-6
構成要素	5-3	ソケットとのやりとり	1-8
処理ルーチン		STREAMS プロトコル・スタック	
close 処理	5-20	BSD ドライバへのブリッジ ..	8-11
open 処理	5-20	STREAMS ヘッダ・ファイル	
書き込み側サービスの処理	5-22	strlog.h	5-8
書き込み側ブット処理	5-21	stropts.h	5-8
構成処理	5-21	sys/stream.h	5-8
読み取り側サービスの処理	5-22	STREAMS ベースのドライバ	
読み取り側ブット処理	5-21	ソケット・ベースのプロトコル・ス タックからのアクセス	1-9
ソケットとの共存	8-1	streamtab データ構造体	5-17
ソケット・フレームワークへの通信 ブリッジ	8-1	STRIFNET オプション	8-4
タイムアウトの使用	5-25	カーネル構成ファイルへの追加	8-4
デバイス特殊ファイル	5-30	strlog.h ヘッダ・ファイル	5-8
同期メカニズム	5-23	stropts.h ヘッダ・ファイル	5-8
		struct sockaddr	4-20

struct sockaddr_in 4-22
struct sockaddr_in6 4-22
struct sockaddr_storage 4-21
struct sockaddr_un 4-21
subtree_tbl.c ファイル 6-13
subtree_tbl.h ファイル 6-10
sys/socket.h ヘッダ・ファイル 4-19
sys/stream.h ヘッダ・ファイル 5-8
sys/types.h ヘッダ・ファイル 4-19
sys/un.h ヘッダ・ファイル 4-19

T

t_accept 関数 3-30
 accept ソケット呼び出しとの相
 違 3-45
t_alloc 関数 3-38
t_bind 関数 3-27
 bind ソケット・コールとの相
 違 3-45
t_close 関数 3-35
T_CLTS 定数 3-11
t_connect 関数 3-29
T_CONNECT 非同期イベント 3-12
T_COTS_ORD 定数 3-11
T_COTS 定数 3-11
T_DATAXFER 状態 3-16
T_DATA 非同期イベント 3-12
T_DISCONNECT 非同期イベン
 ト 3-12
t_errno 変数 3-67
T_ERROR イベント
 TLI におけるサポート 3-42
t_error 関数 3-37, 3-39
T_EXDATA 非同期イベント 3-12

t_free 関数 3-38
t_getinfo 関数 3-37
t_getstate 関数 3-38
T_GODATA 非同期イベント ... 3-12
T_GOEXDATA 非同期イベント 3-12
T_IDLE 状態 3-16
T_INCON 状態 3-16
T_INREL 3-16
t_listen 関数 3-29
T_LISTEN 非同期イベント 3-12
t_look 関数 3-37, 3-39
 select ソケット呼び出しとの相
 違 3-45
T_MORE フラグ
 プロトコル独立における使用 3-40
t_optmgmt 関数 3-67
T_ORDREL 非同期イベント... 3-12
T_OUTCON 状態 3-16
T_OUTREL 状態 3-16
t_rcvdis 関数 3-33
 プロトコル独立における使用 3-40
t_rcvrel 関数 3-34
 プロトコル独立における使用 3-40
t_rcvudata 関数 3-36
t_rcvuderr 関数 3-37
 プロトコル独立における使用 3-40
t_rcv 関数 3-32
t_snddis 関数 3-33
 close ソケット呼び出しとの相
 違 3-46
t_sndrel 関数 3-33
 プロトコル独立における使用 3-40
t_sndudata 関数 3-36
t_snd 関数 3-31
t_sync 関数 3-38

T_UDERR 非同期イベント 3-12
t_unbind 関数 3-34
T_UNBIND 状態..... 3-16
T_UNINIT 状態..... 3-16
 意味 3-25
TCP
 connect システム・コールでの使
 用 4-29
 PAWS オプション D-4
 SACK オプション D-3
 ウィンドウ・サイズ..... D-1
 ウィンドウ・スケール・オプション
 カーネルの構成..... D-3
 エラー回復..... D-3
 往復時間 D-4
 往復にかかる時間 D-1
 コネクション指向型通信 4-9
 シーケンス番号 D-4
 スループット D-1
 タイムスタンプ・オプション . D-4
 転送速度 D-1
 プログラミング情報..... D-1
 プロトコル..... 4-9
TCP_PAWS オプション D-4
TCP_SACKENA オプション.... D-3
TCP_TSOPTENA オプション .. D-4
tiuser.h ファイル 3-8, 3-41
TLI
 XTI との互換性 3-41
 XTI との相違 3-42
 および XTI 3-1
 ヘッダ・ファイル..... 3-8

 ライブラリおよびヘッダ・ファイ
 ル 3-7
 リファレンス・ページ 3-9
TLI プログラムの再コンパイル 3-41
TLOOK エラー・メッセージ
 XTI イベントの原因..... 3-14
traffic control サブシステム ... 7-2
 タスク 7-5
Transport Layer Interface
 (TLI を参照)
Transport Service Data Unit
 (TSDU を参照)
trn_units 変数
 ソース・ルーティングの可能化 E-1
TSDU 3-12
 プロトコル独立 3-40

U

UDP
 connect システム・コールでの使
 用 4-29
 プロトコル..... 4-9
unbind イベント..... 3-17t
UNIX 通信ドメイン 4-4
 特性 4-4
UNIX ドメイン 4-76

W

write 関数 5-10
write システム・コール..... 4-34

X

X/Open トランスポート・インタ

フェース

(XTI を参照)

XID

機能 F-16

定義 F-16

XTI

STREAMS およびソケット・フレー

ムワークとの関係 1-6

TLI との相違 3-42

TLI との比較 3-42

XNS4.0 から XNS5.0 へのコードの

移行 3-48

XNS4.0 と XNS5.0 の相違点 .. 3-47

XNS4.0 と XNS5.0 の相互運用

性 3-51

XPG3 から XNS4.0 へのコードの移

行 3-48

XPG3 と XNS4.0 の相違点 ... 3-47

XPG3 と XNS4.0 の相互運用

性 3-51

XPG3 プログラムの使用 3-49

xtiso の構成 3-68

アプリケーション・プログラミン

グ・インタフェース 3-5

エラー処理 3-67

オプション管理 3-67n

および TLI 3-1

および規格 3-1

概要 3-3

関数の呼び出し順序 3-23

コネクション指向型アプリケーショ

ンの作成 3-26

アドレスの終端へのバイン

ド 3-27

終端のオープン 3-26

終端の初期化 3-26

終端の初期化解除 3-34

接続指示のリッスン 3-29

接続の受け入れ 3-30

接続の打ち切りによる解放の使用

方法 3-33

接続の開始 3-29

接続の解放 3-32

接続の確立 3-29

接続の正常解放の使用 3-33

データの受信 3-32

データの送信 3-31

データの転送 3-31

フェーズに依存しない関数の使

用 3-37

コネクション指向型サービス . 3-5

コネクション指向型プログラム

例 B-2

コネクションレス型アプリケーションの作成

終端の初期化 3-35

終端の初期化解除 3-37

データの転送 3-35

コネクションレス型プログラ

ム B-20

コネクションレス・サービス . 3-5

サービス・モード 3-5

サービス・モードを識別する定数

T_CLTS 3-11

T_COTS 3-11

T_COTS_ORD 3-11

実行モード 3-5

正常解放発信状態	3-16	へのアプリケーションのポー	
接続依存のアプリケーションの作成		ト	3-39
プロトコル・オプションの折		ユーザ, プロバイダ, 終端間の関	
衝	3-67	係	3-4
接続指示	3-12	ライブラリおよびヘッダ・ファイ	
ソケット・アプリケーションの書き		ル	3-7
換え	3-43	ライブラリ・コール	3-9
ソケットとの比較	3-43	xti.h ヘッダ・ファイル	3-8
ソケット・ベースのトランスポー		および <code>t_errno</code> 変数	3-68
ト・プロバイダとのデータ・フ		xtiso オプション	
ロー	1-8	構成	3-68
定義	1-6, 3-1	XTI イベント	3-12
データ転送状態	3-16	<code>T_LOOK</code> エラーの原因	3-14
データ・フロー	1-7	コネクションレス・トランスポー	
同期実行	3-6	ト・サービスによる使用 ...	3-36
とネットワーク・プログラミング環		消費関数	3-14
境	3-2	着信	3-19
トランスポート終端	3-4	トラッキング	3-17
発信接続保留状態	3-16	のマップ	3-19
非同期実行	3-6	発信	3-17t
フェーズに依存しない関数 ...	3-37	XTI エラー	
複数のプロセスの同期化	3-22	<code>t_errno</code> 変数	3-67
プログラム例		ソケットとの比較	3-46
<code>client.h</code> ファイル	B-46	XTI 関数	3-9
<code>clientauth.c</code> ファイル	B-46	のマップ	3-19
<code>clientdb.c</code> ファイル	B-48	XTI 状態	
<code>common.h</code> ファイル	B-36	ソケット状態との比較	3-46
<code>server.h</code> ファイル	B-38	トランスポート・プロバイダによる	
<code>serverauth.h</code> ファイル	B-39	管理	3-25
<code>serverdb.h</code> ファイル	B-43	のマップ	3-19
<code>xtierror.c</code> ファイル	B-45	表	3-16
ヘッダ・ファイル	3-8	XTI における打ち切りによる解	
別の終端への接続の引き渡し	3-18	放	3-12

XTI におけるコネクション指向型サービス
 定義 3-5

XTI におけるコネクションレス型トランスポート・サービス
 一般的な状態遷移 3-24

XTI におけるコネクションレス・サービス
 起こり得る状態遷移 3-20
 定義 3-5

XTI におけるサービス
 モード 3-5

XTI における実行
 モード 3-6

XTI における同期実行
 定義 3-6

XTI における非同期イベント... 3-12
 および消費関数 3-14

XTI における非同期実行
 定義 3-6

XTI のオプション
 T_UNSPEC 3-65
 移植性 3-66
 形式 3-53
 使用 3-51
 折衝 3-54
 引数 info..... 3-66

XTI の状態 3-15

XTI 非同期イベント
 および消費関数 3-14
 表..... 3-12

XTI へのアプリケーションのポート 3-39

XTO のオプション
 トランスポート終端の管理... 3-62

あ

アイドル状態
 XTI における 3-16

アクセス権
 recvmsg システム・コールによる引き渡し 4-36
 sendmsg システム・コールによる引き渡し 4-36

アクティブ・ユーザ
 一般的な状態遷移 3-23
 定義 3-4

値の表現
 eSNMP 6-63

アドレス・テスト・マクロ 4-48

アドレスの生成
 TLI と XTI の比較 3-42

アドレス・ファミリ
 DLI のための指定 F-19

アプリケーション
 AF_INET6 ソケットへの移植 4-54
 RSVP 7-9

アプリケーション・プログラミング・インタフェース
 STREAMS..... 1-4, 5-8
 XTI 1-1, 3-5
 ソケット 1-4, 4-7

アプリケーション・プログラム
 XTI のための書き換え 3-43
 XTI へのポート 3-39

ソケット
 netdb.h ヘッダ・ファイルの使用 4-12

い

イベント

- STREAMS におけるロギング 5-32
- XTI における 3-12
- XTI におけるトラッキング... 3-17
- コネクションレス・トランスポート・サービスによる使用... 3-36
- 着信 (XTI) 3-19
- 定義 3-6
- 発信 (XTI) 3-17t

イベント管理

- TLI の互換性における 3-41

インターネット通信ドメイン

- 特性 4-4

イーサネット

- アクセス F-4
- アドレス F-4
- 複数ユーザ F-4
- メッセージ転送 F-4

イーサネット・サブ構造体

- 送受信 F-7
- 代入 F-22
- フレーム構造体 F-8

イーサネット・フレーム構造体

- 機能 F-9
- デスティネーション・システム情報の指定 F-8
- 例 F-5, F-8

え

エラー

- STREAMS におけるロギング 5-32

- XTI と TLI の相違 3-42
- XTI とソケットの比較 3-46
- XTI における処理 3-67
- エラー回復
 - 提供 F-4, F-18

お

往復にかかる時間

- 定義 D-1

オブジェクト・テーブル

- eSNMP 6-10

オプション

- XTI 3-51

オプション管理

- および TCP 3-67n

か

- 書き込み側サービスの処理 5-22

- 書き込み側ブット処理 5-21

書き込み専用アクセス

- TLI におけるサポート 3-42

拡張 **SNMP**

- (eSNMP を参照)

関数

- STREAMS 5-9
- XTI で可能な呼び出し順序 ... 3-23
- XTI とソケットの比較 3-44
- プロトコル独立における使用 3-39

関数の呼び出し順序

- XTI における 3-23

カーネル構成ファイル

- DLPI オプション 8-4

STRIFNET オプション 8-4
カーネルのインプリメンテーション
ソケット 4-7
カーネルのサブシステム
STREAMSドライバの組み込
み 5-25
STREAMSモジュールの組み込
み 5-25
カーネル・レベル関数
STREAMS 5-16

き

擬似端末
スレーブ・デバイス 4-100
定義 4-100
マスタ・デバイス 4-100
キャノニカル・アドレス
トークン・リング・ドライバ E-3
境界合わせ
ルーティング情報フィールド E-3
共存
STREAMS とソケット 8-1
Tru64 UNIX のための定義 8-1

く

クライアント/サーバ相互作用 .. 4-10
クライアント・プロセス
接続の確立 4-28
定義 4-10
クローン・デバイス 5-31

こ

交換識別

機能 F-16
定義 F-16
高次レベル・サービス
提供 F-4, F-18
構成処理 5-21
構造体の境界合わせ E-3
肯定応答コネクションレス・モード・
サービス
DLPI 2-7
コネクション指向型アプリケーション
作成 3-26
終端の初期化 3-26
プログラム例 B-2
ヘッダ・ファイル例 B-36
コネクション指向型通信 4-9
コネクション指向型トランスポート・
サービス
XTI で起こり得る状態遷移... 3-21
関数の一般的な呼び出し順序 3-23
コネクション・モード・サービス
DLPI 2-5
コネクションレス型アプリケーション
作成 3-35
プログラム例 B-20
ヘッダ・ファイル例 B-36
コネクションレス型通信 4-9
コネクションレス・モード・サービス
DLPI 2-6
コントロール・フィールド
機能 F-15

さ

サブエージェント

実現 6-15
サブ構造体

802.2.....	F-12	DLI	F-18
イーサネット・フレーム構造体	F-8	値の指定	F-7
送受信	F-7	ソケット	4-11
代入	F-7	データ転送のための使用	F-24
サブ識別子	6-8	要約	F-18
サブシステム		呼び出し順序	F-18
traffic control	7-2	実行モード	
サーバ/クライアント相互作用 ..	4-10	ソケット	
サーバ・プロセス		非ブロッキング・モード...	4-26
コネクション指向型	4-30	ブロッキング・モード	4-26
コネクションレス型	4-31	受信	
接続の受け入れ	4-30	IPv4 マルチキャスト・データグラ	
定義	4-10	ム	4-84
サービス		IP マルチキャスト・データグラ	
高次レベル	F-18	ム	4-88
サービス・クラス		データ・ユニット	3-36
値	F-13	データ・ユニットに関するエ	
定義	F-13	ラー	3-37
サービスの種類		状態	
DLPI	2-4	XTIでの管理	3-25
サービス品質		XTI とソケットの比較	3-46
(QoS を参照)		XTI における	3-15
し		状態遷移	
シェアード・ライブラリ		初期化フェーズで起こり得る	3-20
TLI での使用	3-7	データ転送で起こり得る	
XTI での使用	3-7	コネクションレス型トランスポー	
シグナル		ト・サービス	3-20
ソケット用にプロセス ID の設		消費関数	
定	4-98	非同期 XTI イベント用	3-14
ソケット用にプロセス・グループの		初期化されていない状態	
設定	4-98	XTI における	3-16
システム・コール		初期化フェーズ	
		起こり得る状態遷移	3-20

す

推奨

コネクション指向型トランスポート
と CLTS の使用..... 3-5

実行モードの使用..... 3-6

ストリーム

エンド

およびデバイス・ドライバ . 5-5

定義 5-3

ヘッド 5-4

モジュール..... 5-6

ストリーム・ソケット..... 4-6

スレーブ・デバイス..... 4-100

せ

正常解放

イベント指示 3-12

定義 3-11

プロトコル独立における使用 3-40

正常解放着信状態

XTI における 3-16

正常解放発信状態

XTI における 3-16

接続

別の終端への引き渡し 3-18

接続確立フェーズ

起こり得る状態遷移..... 3-21

接続指示

XTI における 3-12

接続モード

DLPI での通信..... 2-3

遷移

XTI 状態間..... 3-19

前提条件

DLI プログラミング F-1

そ

ソケット

4.3BSD msghdr データ構造体 4-51

4.4BSD でのプロトコル・データの
受信 4-51

4.4BSD のネットワーク・アドレ

ス 4-50

accept システム・コール.... 4-11t

AF_INET6 4-54

AF_INET6 のプログラム例 ... C-7

AF_INET のプログラム例..... C-1

bind システム・コール..... 4-11t

BSD 4-49

cmsghdr データ構造体 4-23

connect システム・コール... 4-11t

DLI オプションの設定 F-20

DLI でのバインド..... F-21

DLI での非アクティブ化 F-25

DLI 内に作成 F-19

DLI のバッファ・サイズの計

算 F-24

fcntl システム・コール..... 4-98

getpeername システム・コー

ル 4-11t

getsockname システム・コー

ル 4-11t

listen システム・コール.... 4-11t

msghdr データ構造体 4-22

raw (IPv6)..... 4-65

recvfrom システム・コール . 4-11t

recvmsg システム・コール.. 4-11t

recv システム・コール 4-11t

sendmsg システム・コール . 4-11t	クローズ時におけるリソースの再 生 4-38
sendto システム・コール.... 4-11t	高度なプログラミング情報... 4-54
send システム・コール..... 4-11t	コネクション指向型プログラム 例 B-2
setsockopt システム・コール 4-11t	コネクション指向サーバ・プロセ ス 4-30
shutdown システム・コール 4-11t	コネクション指向モード 4-9
sockaddr_dl 構造体への代入 . F-21	コネクションレス型サーバ・プロセ ス 4-31
sockaddr_in6 データ構造体 .. 4-22	コネクションレス型プログラ ム B-20
sockaddr_in データ構造体.... 4-22	コネクションレス型モード.... 4-9
sockaddr_storage データ構造 体 4-21	作成 4-25
sockaddr_un データ構造体... 4-21	サーバ接続の確立..... 4-30
sockaddr データ構造体..... 4-20	サービス名をポート番号にマッ プ 4-15
socketpair システム・コール 4-11t	シグナル用にプロセス ID の設 定 4-98
socket システム・コール.... 4-11t	シグナル用にプロセス・グループの 設定 4-98
STREAMS との共存 8-1	システム・コール..... 4-11
STREAMS フレームワークへの通信 ブリッジ 8-1	実行モード..... 4-26
TCP 固有のプログラミング情 報 D-1	シャットダウン 4-38
TCP ソケット・バッファ・サイズの プログラミング..... D-2	ソケット・オプションの取得 4-32
XTI との比較 3-43	ソケット・オプションの設定 4-32
XTI 用アプリケーションへの書き換 え 3-43	帯域外データの処理..... 4-77
アプリケーション・プログラミン グ・インタフェース 4-7	タイプ 4-5
一般的なエラー 4-53	SOCK_DGRAM 4-6
カーネルのインプリメンテーショ ン 4-7	SOCK_RAW 4-6
クライアント接続の確立 4-28	SOCK_STREAM 4-6
クローズ..... 4-38	通信ドメイン 4-4
クローズ時におけるフラッ シュ 4-39	UNIX ドメイン 4-4

インターネット・ドメイン	4-4	ソケットおよび STREAMS フレー	
通信プロパティ	4-3	ムワーク	
通信モード	4-8	間の通信	1-8
定義	4-3	ソケット・クライアント	
データの転送	4-33	AF_INET6 ソケットを使用するプロ	
特性	4-3	グラム	C-7
名前のバインド	4-28	AF_INET ソケットを使用するプロ	
入出力の多重化	4-94	グラム	C-2
ネットワーク名をネットワーク番号		ソケット・サーバ	
にマップ	4-14	AF_INET6 ソケットを使用するプロ	
バッファ・サイズ制限の増加 . D-3		グラム	C-11
プログラム例		AF_INET ソケットを使用するプロ	
client.h ファイル	B-46	グラム	C-5
clientauth.c ファイル	B-46	ソケット状態	
clientdb.c ファイル	B-48	XTI 状態との比較	3-46
common.h ファイル	B-36	ソケット・タイプ	
server.h ファイル	B-38	SOCK_DGRAM	4-6
serverauth.h ファイル	B-39	SOCK_RAW	4-6
serverdb.h ファイル	B-43	SOCK_STREAM	4-6
xtierror.c ファイル	B-45	ソケット通信プロパティ	4-3
プロトコルの選択	4-74	ソケットのクローズ	4-38
プロトコル名をプロトコル番号に		ソケットのシャットダウン	4-38
マップ	4-14	ソケットのタイプ	4-5
ヘッダ・ファイル	4-19	ソケットの命名	4-7
ホスト名をアドレスにマップ	4-12	ソケット・フレームワーク . 1-4, 4-2	
命名	4-7	STREAMS とのやりとり	1-8
ライブラリ・コール	4-12	XTI との関係	1-6
表	4-16	構成要素	4-2
ソケット I/O		ソケット・プロトコル・スタック	
割り込み駆動	4-97	STREAMS ドライバへのブリッ	
ソケット・インタフェース		ジ	8-2
TCP/IP との使用	4-1	ソケット・ヘッダ・ファイル	4-19
サポートされているタイプ	4-1	ソケットへの名前のバインド	4-28
ソケット・エラー		ソケット・ベースのドライバ	
XTI との比較	3-46		

STREAMS ベースのプロトコル・ス
タックからのアクセス 1-10
ソース・サービス・アクセス・ポイ
ント
(SSAP を参照)
ソース・ルーティング
可能化 E-1

た

帯域外データ
受信 4-78
送信 4-78
ソケット・フレームワークにおける
処理 4-77
帯域幅の管理
ネットワーク 7-1
タイムアウト 5-25
タイムスタンプ・オプション D-4
多重化 4-94

ち

着信イベント
XTI における 3-19
トラッキング (XTI) 3-18
着信接続保留状態
XTI における 3-16

つ

通常データ 3-12
通信ドメイン
ソケット 4-4
UNIX ドメイン 4-4

インターネット・ドメイン . 4-4
通信の肯定応答コネクションレス・
モード
DLPI 2-4
通信のコネクションレス・モード
DLPI 2-3
通信ブリッジ
dlb STREAMS 擬似ドライバ . 1-8,
1-10, 8-1
ifnet STREAMS モジュール.. 1-8,
1-9, 8-1
定義 8-1
通信モード
コネクション指向型 (ソケット) 4-9
コネクションレス型 (ソケット) 4-9
ソケット 4-8

て

デスティネーション・オプション
受信 4-73
送信 4-73
デスティネーション・サービス・アク
セス・ポイント
(DSAP を参照)
デスティネーション・システム
情報の指定 F-8
デスティネーション・システムの物理
アドレス
指定 F-10
定義 F-9, F-13
デバイス特殊ファイル 5-30
デバイス・ドライバ
STREAMS 処理ルーチン 5-19

およびストリーム・エンド	5-5	DLI における	F-24
転送		ソケットによる	4-33
別の終端へ.....	3-16	データ転送状態	
伝送制御プロトコル		XTI における	3-16
(TCP を参照)		データ転送フェーズ	
転送速度		コネクションレス型サービス	3-35
定義	D-1	コネクションレス・トランスポート・サービスで起こり得る状態遷移	3-20
データグラム・ソケット ..	4-5, F-1	データ・フロー	
データ構造体		XTI および STREAMS ベースのトランスポート・プロバイダ .	1-7
4.3BSD msghdr	4-51	XTI およびソケット・ベースのトランスポート・プロバイダ....	1-8
4.4BSD msghdr	4-52	データ・ユニット	
cmsghdr	4-23	エラー情報の受信.....	3-37
dblk_t.....	5-18	受信	3-36
hostent	4-13	データ・リンク・インタフェース	1-3,
mblk_t.....	5-18	2-1, F-1	
msghdr	4-22, 4-23	(DLI も参照)	
netent	4-14	DLPI.....	2-1
protoent	4-14	データ・リンク・サービス・プロバイダ	
servent	4-15	(DLS プロバイダ を参照)	
sockaddr.....	4-20	データ・リンク・サービス・ユーザ	
sockaddr_in	4-22	(DLS ユーザ を参照)	
sockaddr_in6	4-22	データ・リンク・プロバイダ・インタフェース	
sockaddr_storage	4-21	(DLPI を参照)	
sockaddr_un.....	4-21	デーモン	
ソケット	4-20	inetd.....	4-93
メッセージ.....	5-17	rsvpd	7-7
モジュール.....	5-16		
module_info	5-16		
qinit.....	5-16		
streamtab.....	5-17		
データ・セグメンテーション			
提供	F-4		
データ・セグメント			
提供	F-18		
データ転送			

と

同期化

XTIにおける複数のプロセス . 3-22
 同期メカニズム
 STREAMSにおける使用 5-23
 特権
 スーパーユーザ F-1
 ドメイン
 指定 F-7
 ドライバ
 BSD から STREAMS プロトコル・
 スタックへのブリッジ 8-11
 ソケット・プロトコル・スタックへ
 の STREAMS ドライバのブリッ
 ジ 8-2
 トークン・リング E-1
 ドライバのブリッジ
 BSD ドライバから STREAMS プロ
 トコル・スタックへ 8-11
 ソケット・プロトコル・スタックへ
 の STREAMS ドライバ 8-2
 トランスポート終端
 定義 3-3
 トランスポート・プロバイダ
 状態の管理 3-25
 定義 3-3
 トランスポート・ユーザ
 定義 3-3
 トークン・リング・ドライバ
 キャノニカル・アドレス E-3
 ソース・ルーティングの可能化 E-1

な

名前のアドレスへのバインド ... 4-75

in6addr_any ワイルドカード・アド
 レス 4-75
 INADDR_ANY ワイルドカード・ア
 ドレス 4-75
 UNIX ドメインにおける 4-76

に

入出力の多重化 4-94

ね

ネットワーク

DLI による LAN へのアクセス F-4
 QoS 7-1
 QoS アーキテクチャ 7-2
 QoS 構成要素の動作 7-3
 帯域幅の管理 7-1

ネットワーク・アドレス

とソケット 4-50

ネットワーク構成

ブロードキャストと調査 4-89

ネットワーク・ソケット・アプリケー ション

AF_INET6 のプログラム例 ... C-7

AF_INET のプログラム例 C-1

ネットワーク・デバイス

指定 F-7

ネットワーク・バイト順の変換

ネットワーク・プログラミング・イ
 ンタフェース

STREAMS 1-4

ソケット 1-4

ネットワーク・ライブラリ・ルーチン..... 4-12, 4-14, 4-15

は

配送
提供 F-4
バインドされていない状態
XTI における 3-16
パケット・ヘッダ (IPv6) 4-67
パケット・ルーティング
提供 F-4
パッシブ・ユーザ
一般的な状態遷移 3-23
定義 3-4
発信イベント
XTI における 3-17t
トラッキング (XTI) 3-17
発信接続保留状態
XTI における 3-16
バッファ・サイズ
DLI での計算 F-24
TCP ソケット用に増加 D-3

ひ

非番号情報コマンド
機能 F-17
定義 F-17
非ブロッキング・モード
(非同期実行を参照)
標準フレーム・フォーマット
802..... F-1
イーサネット F-1

ふ

ファイル記述子
プロトコル独立における使用 3-40
複数のプロセス
XTI における同期化 3-22
複数ユーザ
イーサネット F-4
物理アドレス..... F-1
使用 F-10
物理的にアタッチするポイント
(PPA を参照)
フレーム
構築 F-4
フレーム・フォーマット
802..... F-1
802.3..... F-5, F-12
FDDI F-5
イーサネット F-1, F-5, F-8
処理 F-13
標準 F-1
フレームワーク
STREAMS..... 5-2
構成要素 5-3
STREAMS の構成要素 5-4
STREAMS メッセージ 5-6
ソケット..... 4-2
構成要素 4-2
プログラミング例
DLI F-25
プロセス
XTI における複数のプロセスの同期化 3-22
複数のプロセス間における単一の終端の共用 3-23
ブロッキング・モード

(同期実行を参照)
プロトコル
 ソケット・システム・コールを使用した選択 4-74
プロトコル固有のオプション
 プロトコル独立における使用 3-40
プロトコル・タイプ
 定義 F-10
プロトコル独立
 XTIアプリケーション用..... 3-39
フロー制御
 XTIとTLIの相違 3-42
 XTIにおける 3-12
 提供 F-4, F-18
ブロードキャストとネットワーク構成の調査 4-89
分散型アプリケーション
 クライアント/サーバ・モデル 4-10

へ

並行プログラム
 実行 F-2
ヘッダ・ファイル
 fcntl.h 3-8
 netinet/in.h..... 4-19
 STREAMS..... 5-8
 sys/socket.h 4-19
 sys/types.h 4-19
 sys/un.h 4-19
 tiuser.h 3-8, 3-41
 xti.h 3-8, 3-43
 XTIおよびTLI..... 3-8
 指定の規約..... 4-5

ソケット..... 4-19

ほ

補助データ 4-23
ホップ・バイ・ホップ・オプション
 受信 4-72
 送信 4-72
ポーティング
 XTIアプリケーション作成のガイド
 ライン..... 3-39
 プロトコル独立 3-39

ま

マクロ
 IN6_ARE_ADDR_EQUAL... 4-48,
 4-63
 IN6_IS_ADDR_LINKLOCAL 4-48
 IN6_IS_ADDR_LOOPBACK . 4-48
 IN6_IS_ADDR_MC_GLOBAL 4-48
 IN6_IS_ADDR_MC_LINKLO-
 CAL..... 4-48
 IN6_IS_ADDR_MC_NODELO-
 CAL..... 4-48
 IN6_IS_ADDR_MC_ORGLO-
 CAL..... 4-48
 IN6_IS_ADDR_MC_SITELO-
 CAL..... 4-48
 IN6_IS_ADDR_MULTICAST 4-48
 IN6_IS_ADDR_SITELOCAL. 4-48
 IN6_IS_ADDR_UNSPECI-
 FIED..... 4-48, 4-63
 IN6_IS_ADDR_V4COMPAT.. 4-48

IN6_IS_ADDR_V4MAPPED 4-48,
4-62
アドレス・テスト 4-48
マスタ・デバイス 4-100
マップ
サービス名をポート番号に... 4-15
ネットワーク名をネットワーク番号
に 4-14
プロトコル名をプロトコル番号
に 4-14
ホスト名をアドレスに 4-12
マルチキャスト 4-80
マルチキャスト・アドレス F-1
使用 F-10
マルチキャスト・グループ
定義 4-80

め

メソッド・ルーチン
eSNMP 6-61
メッセージ・データ構造体 5-17
メッセージのタイプ
通常 5-6
優先 5-6
メッセージ・ブロック
構成要素 5-17
dblk_t 制御構造 5-17
mbblk_t 制御構造 5-17
データ・バッファ 5-17

も

モジュール
STREAMS 処理ルーチン 5-19
close 処理 5-20

open 処理 5-20
書き込み側サービスの処理 5-22
書き込み側プット処理 5-21
構成処理 5-21
読み取り側サービスの処理 5-22
読み取り側プット処理 5-21
モジュール・データ構造体 5-16
module_info 5-16
qinit 5-16
streamtab 5-17

ゆ

優先データ
コネクションレス・トランスポート・サービスにおける使用 3-36
ユーザ・データグラム・プロトコル
(UDP を参照)

よ

読み取り側サービスの処理 5-22
読み取り側プット処理 5-21
読み取り専用アクセス
TLI におけるサポート 3-42

ら

ライブラリ
TLI 3-7
XTI 3-7
ライブラリ・コール
STREAMS
fattach 5-13
fdetach 5-13

isastream 5-12
XTI 3-9
ソケット 4-12

り

リンク
XTIおよびTLIライブラリ 3-7

る

ルーティング情報フィールド E-3
ルーティング・ヘッダ 4-69
受信 4-70

送信 4-70

ろ

論理データ境界
 プロトコル独立における 3-40
論理リンク・コントロール
 (LLC を参照)
ローカル管理サービス
 DLPI 2-5

わ

割り込み駆動のソケット I/O 4-97



Tru64 UNIX ドキュメントの購入方法

Tru64 UNIX ドキュメントのご購入については、弊社担当営業または日本ヒューレット・パッカートの各営業所/代理店にお問い合わせください。

各ドキュメント・キットの注文番号は以下のとおりです。ドキュメント・キットに含まれるマニュアルの内容については『ドキュメント概要』を参照してください。

キット名	注文番号
Tru64 UNIX Documentation CD-ROM	QA-6ADAA-G8
Tru64 UNIX Documentation Kit	QA-6ADAA-GZ
End User Documentation Kit	QA-6ADAB-GZ
- Startup Documentation Kit	QA-6ADAC-GZ
- General User Documentation Kit	QA-6ADAD-GZ
- System and Network Management Documentation Kit	QA-6ADAE-GZ
Developer's Documentation Kit	QA-6ADAF-GZ
Reference Pages Documentation Kit	QA-6ADAG-GZ
TruCluster Server Documentation Kit	QA-6BRAA-GZ
Tru64 UNIX 日本語ドキュメント・キット	QA-6ADJB-GZ
スタートアップ・ドキュメント・キット	QA-6ADJC-GZ
一般ユーザ・ドキュメント・キット	QA-6ADJD-GZ
システム/ネットワーク管理ドキュメント・キット	QA-6ADJE-GZ
プログラミング・ドキュメント・キット	QA-6ADJF-GZ
CDE 翻訳ドキュメント・キット	QA-6ADJG-GZ
TruCluster Server 日本語ドキュメント・キット	QA-05SJA-GZ
Advanced Server for UNIX 日本語ドキュメント・キット	QA-5U2JA-GZ



マニュアルに対するご意見

Tru64 UNIX

ネットワーク・プログラミング・ガイド

AA-RK3QE-TE

弊社のマニュアルに関して、ご意見、ご要望、または内容の不明確な部分など、お気づきの点がございましたら、下記にご記入の上、弊社社員にお渡しくださるようお願い申し上げます。

マニュアルの採点：

	大変良い	良い	普通	良くない
正確さ(説明どおりに動作するか)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
情報量(十分か)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
分かり易さ	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
マニュアルの構成	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
図(役立つか)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
例(役立つか)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
索引(項目の検索性)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
ページ・レイアウト(情報の検索性)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

内容の不明確な部分がありましたら、以下にご記入ください：

ペー ジ

その他お気づきの点がございましたら、以下にご記入ください：

ご使用のソフトウェアのバージョン： _____

貴社名/部課名 _____

御名前 _____

記入日 _____

(注) 当用紙を受け取った弊社社員は、すみやかに下記にお送りください。

ビジネスクリティカルシステム統括本部 **BCS** 技術本部 **Alpha** ソフトウェア技術部