お客様各位

# カタログ等資料中の旧社名の扱いについて

2010年4月1日を以ってNECエレクトロニクス株式会社及び株式会社ルネサステクノロジが合併し、両社の全ての事業が当社に承継されております。従いまして、本資料中には旧社名での表記が残っておりますが、当社の資料として有効ですので、ご理解の程宜しくお願い申し上げます。

ルネサスエレクトロニクス ホームページ（http://www.renesas.com）

2010年4月1日
ルネサスエレクトロニクス株式会社

【発行】ルネサスエレクトロニクス株式会社（http://www.renesas.com）

【問い合わせ先】http://japan.renesas.com/inquiry

# ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されていない用途に当社製品を使用したことによりお客様または第三者に生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   標準水準： コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通用信号機器、防災・防犯装置、各種安全装置、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害などを生じさせないようお客様の責任において冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、機器またはシステムとしての出荷保証をお願いいたします。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、詳細につきましては製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注 1. 本資料において使用されている「当社」とは、ルネサスエレクトロニクス株式会社およびルネサスエレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

アプリケーション・ノート

# RX-NET

## ネットワーク・ライブラリ

## プログラミング・ガイド

---

**対象デバイス**

V850シリーズ™

VRシリーズ™

**対象リアルタイムOS**

RX850 Pro Ver.3.15

RX4000V4 Ver.4.10

**対象ネットワーク・ライブラリ**

RX-NET（TCP/IP） Ver.1.30

RX-NET（PPP） Ver.1.30

**資料番号** U16329JJ2V0AN00（第2版）
**発行年月** February 2003 CP(K)

〔メ　モ〕

# 目次要約

- 本資料に記載されている内容は2003年1月現在のもので，今後，予告なく変更することがあります。量産設計の際には最新の個別データ・シート等をご参照ください。
- 文書による当社の事前の承諾なしに本資料の転載複製を禁じます。当社は，本資料の誤りに関し，一切その責を負いません。
- 当社は，本資料に記載された当社製品の使用に関連し発生した第三者の特許権，著作権その他の知的財産権の侵害等に関し，一切その責を負いません。当社は，本資料に基づき当社または第三者の特許権，著作権その他の知的財産権を何ら許諾するものではありません。
- 本資料に記載された回路，ソフトウエアおよびこれらに関する情報は，半導体製品の動作例，応用例を説明するものです。お客様の機器の設計において，回路，ソフトウエアおよびこれらに関する情報を使用する場合には，お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し，当社は，一切その責を負いません。
- 当社は，当社製品の品質，信頼性の向上に努めておりますが，当社製品の不具合が完全に発生しないことを保証するものではありません。当社製品の不具合により生じた生命，身体および財産に対する損害の危険を最小限度にするために，冗長設計，延焼対策設計，誤動作防止設計等安全設計を行ってください。
- 当社は，当社製品の品質水準を「標準水準」，「特別水準」およびお客様に品質保証プログラムを指定していただく「特定水準」に分類しております。また，各品質水準は，以下に示す用途に製品が使われることを意図しておりますので，当社製品の品質水準をご確認ください。

  標準水準：コンピュータ，OA機器，通信機器，計測機器，AV機器，家電，工作機械，パーソナル機器，産業用ロボット

  特別水準：輸送機器（自動車，電車，船舶等），交通用信号機器，防災・防犯装置，各種安全装置，生命維持を目的として設計されていない医療機器

  特定水準：航空機器，航空宇宙機器，海底中継機器，原子力制御システム，生命維持のための医療機器，生命維持のための装置またはシステム等

当社製品のデータ・シート，データ・ブック等の資料で特に品質水準の表示がない場合は，標準水準製品であることを表します。意図されていない用途で当社製品の使用をお客様が希望する場合には，事前に当社販売窓口までお問い合わせください。

（注）

（1）本事項において使用されている「当社」とは，NECエレクトロニクス株式会社およびNECエレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいう。

（2）本事項において使用されている「当社製品」とは，（1）において定義された当社の開発，製造製品をいう。

M8E　02.11

〔メ　モ〕

# はじめに

**対象者**　このマニュアルは，V850シリーズ，VRシリーズの応用システムを設計，開発するユーザを対象とします。

**目　的**　このマニュアルは，次の構　成に示すRX-NETを用いたアプリケーション･プログラムの作成方法をユーザに理解していただくことを目的としています。

**構　成**　このマニュアルは，大きく分けて次の内容で構成しています。

- RX-NETの概要
- RX-NETのコンフィギュレーション設定
- Ethernetコントローラドライバ
- UARTコントローラドライバ
- ハードウエア・アクセス・ライブラリ
- API関数
- PPP

**読み方**　このマニュアルの読者には，電気，論理回路，マイクロコンピュータ，C 言語，アセンブリ言語に関する一般知識が必要です。

V850 シリーズのハードウエア機能，命令機能を知りたいとき
→各製品のユーザーズ・マニュアルを参照してください。

VR シリーズのハードウエア機能，命令機能を知りたいとき
→各製品のユーザーズ・マニュアルを参照してください。

**凡　例**　注　　：本文中につけた注の説明
注意　　：気をつけて読んでいただきたい内容
備考　　：本文の補足説明
数の表記　：2 進数 …XXXX または B’XXXX
10 進数…XXXX
16 進数…0xXXX または H’XXXX
2 のべき数を示す接頭語（アドレス空間，メモリ容量）：
K（キロ）：$2^{10}$＝1024
M（メガ）：$2^{20}$＝$1024^2$
G（ギガ）：$2^{30}$＝$1024^3$

**関連資料**　**このマニュアルを使用する場合は，次の資料もあわせてご覧ください。**
**関連資料は暫定版の場合がありますが，この資料では「暫定」の表示をしておりません。**
**あらかじめご了承ください。**

V850**シリーズの開発ツールに関する資料（ユーザーズ・マニュアル）**

| 資　料　名 | | 資料番号 | |
|---|---|---|---|
| | | 和文 | 英文 |
| CA850 Ver.2.50**以上** **Cコンパイラ・パッケージ** | **操作編** | U16053J | U16053E |
| | **C言語編** | U16054J | U16054E |
| | PM plus**編** | U16055J | U16055E |
| | **アセンブリ言語編** | U16042J | U16042E |
| ID850 Ver.2.40　**統合ディバッガ** | **操作編** | U15181J | U15181E |
| SM850 Ver.2.40**以上　システム・シミュレータ** | **操作編** | U15182J | U15182E |
| SM850 Ver.2.00**以上　システム・シミュレータ** | **外部部品ユーザ・オープン・インタフェース仕様編** | U14873J | U14873E |
| RX850 Ver.3.13**以上　リアルタイム**OS | **基礎編** | U13430J | U13430E |
| | **インストレーション編** | U13410J | U13410E |
| | **テクニカル編** | U13431J | U13431E |
| RX850 Pro Ver.3.13　**リアルタイム**OS | **基礎編** | U13773J | U13773E |
| | **インストレーション編** | U13774J | U13774E |
| | **テクニカル編** | U13772J | U13772E |
| RX-NET　**ネットワーク・ライブラリ**（TCP/IP） | | U15083J | － |
| RX-NET　**ネットワーク・ライブラリ**（PPP） | | U15303J | － |
| RX-NET　**ネットワーク・ライブラリ**（DNS） | | U15304J | － |
| RX-NET　**ネットワーク・ライブラリ**（DHCP） | | U15382J | － |
| RX-NET　**ネットワーク・ライブラリ**（SMTP） | | U15505J | － |
| RX-NET　**ネットワーク・ライブラリ**（POP） | | U15539J | － |
| RX-NET Ver.1.00 **ネットワーク・ライブラリ**（telnet） | | U16085J | － |
| RX-NET **ネットワーク・ライブラリ　プログラミング・ガイド（アプリケーション・ノート）** | | **このマニュアル** | － |
| RD850 Ver.3.01　**タスク・ディバッガ** | | U13737J | U13737E |
| RD850 Pro Ver.3.01　**タスク・ディバッガ** | | U13916J | U13916E |
| AZ850 Ver.3.10　**システム・パフォーマンス・アナライザ** | | U14410J | U14410E |
| PG-FP3　**フラッシュ・メモリ・プログラマ** | | U13502J | U13502E |
| PG-FP4　**フラッシュ・メモリ・プログラマ** | | U15260J | U15260E |

VR**シリーズの開発ツールに関する資料（ユーザーズ・マニュアル）**

| 資　料　名 | | 資料番号 | |
|---|---|---|---|
| | | 和文 | 英文 |
| RX4000 $\mu$ ITRON4.0　**リアルタイム**OS | **基礎編** | U14833J | U14833E |
| | **テクニカル編** | U14835J | U14835E |
| | **インストレーション編** | U14834J | U14834E |
| AZ4000 Ver.4.00　**システム・パフォーマンス・アナライザ** | | U15031J | U15031E |

# 目 次

# 図の目次

# 表の目次

# 第1章　RX-NETの概要

## 1.1　RX-NET がサポートするプロトコル

RX-NET がサポートしているプロトコルは以下のようになります。

■ 基本ライブラリ

◇　トランスポート層　　TCP，UDP

◇　ネットワーク層　　IP，ICMP，ARP

◇　データリンク層　　MAC（IEEE802.1）

■ PPP ライブラリ

◇　データリンク層　　PPP

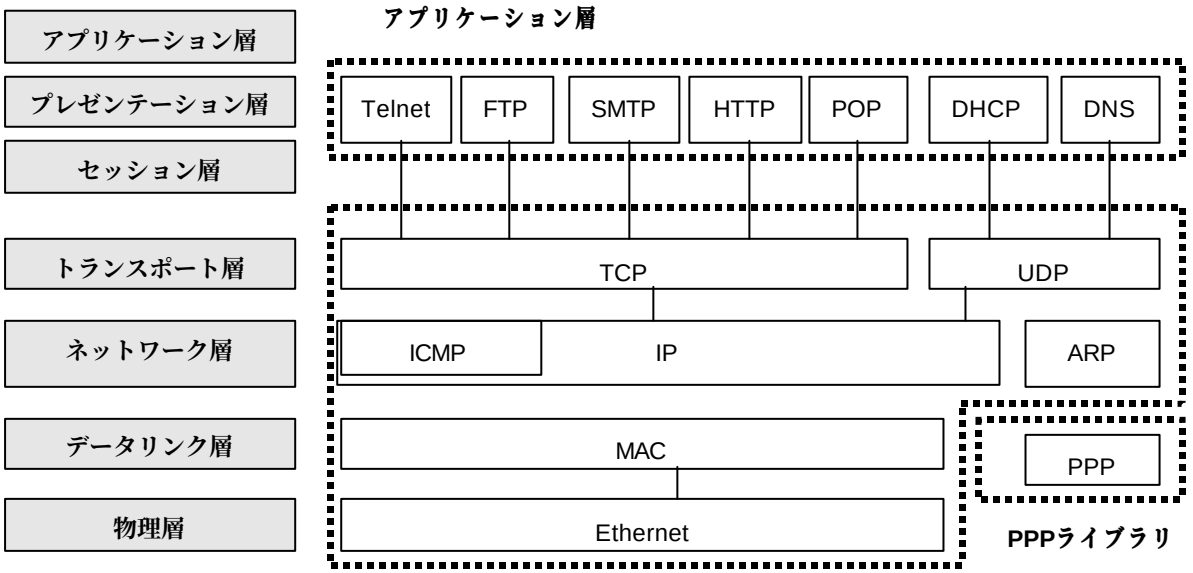
■ クライアント系ライブラリ

◇　アプリケーション層　　DNS，POP，SMTP，DHCP

■ サーバ系ライブラリ

◇　アプリケーション層　　TELNET，FTP，WebServer

図 1－1　OSI 参照モデル

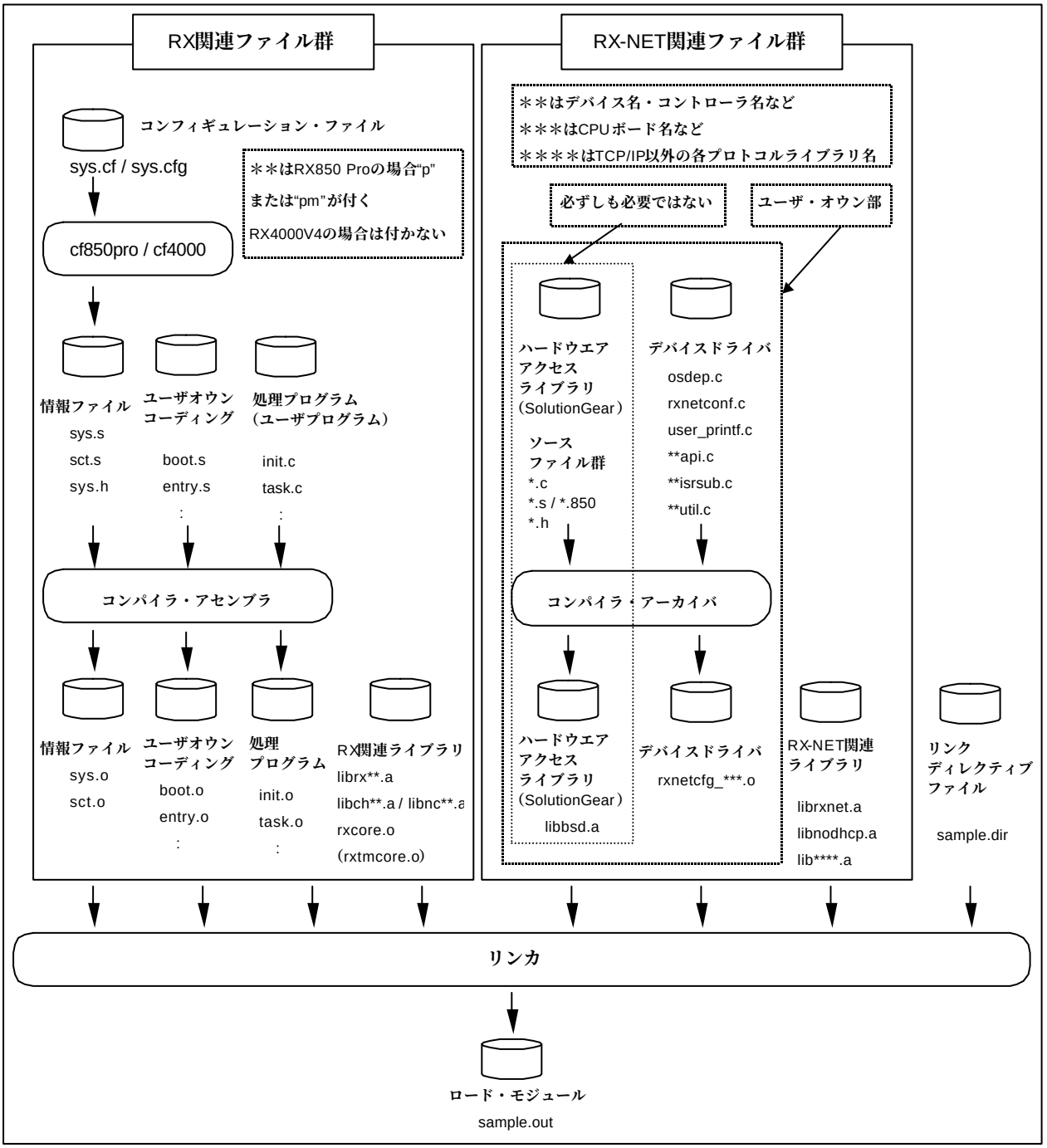


## 1.2　使用するファイルと構築手順の概要

RX-NET のシステム構築について説明します。RX-NET で使用するファイル群と構築手順の概要は以下のようになります。RX-NET は NEC エレクトロニクス製リアルタイム OS の機能を利用して動作しますので，V850 シリーズ V850E を使用する場合は RX850 Pro が，VR シリーズの場合 RX4000V4 が別途必要になります。

下図は使用するファイル群と構築手順です。RX とは，V850E シリーズの場合 RX850 Pro を，VR シリーズの場合 RX4000V4 を意味します。

図 1－2　RX-NET で使用するファイル群と構築手順



図の中の拡張子は使用するコンパイラによって異なり，以下のようになります。

| | アセンブリ・ファイル | リンク時のアドレス情報ファイル ※ |
|---|---|---|
| NEC エレクトロニクス製 CA850 | .s | .dir |
| GHS 製 CCV850（VR 版の場合 CCMIPE） | .850（.mip） | .lx |

※　「リンク時のアドレス情報ファイル」は，NEC エレクトロニクス製では“ リンク・ディレクティブ・ファイル ”，GHS 製では“ リンク・マップ・ファイル ”と呼びます。

RX-NET のパッケージには“ Ethernet コントローラ ”“ UART コントローラ ”といったデバイス・ドライバのサンプルが含まれています。デバイス・ドライバのソース・コードもパッケージに入っていますが，アーカイブやオブジェクトとしてすでに作成されています。また Ver.1.30 からは SolutionGear のドライバ群も添付しており，RX-NET と共通化できる“ マザーボードのドライバ ”は SolutionGear のサンプルを使用するように変更しました[注]。ハードウエア構成の詳細は“ 1. 3 提供サンプルのハードウエア構成 ”を参照してください。

注：SolutionGear のサンプルが入っているのは V850E 版のみです。VR 版は SolutionGearⅡに対応しますが，現状では SolutionGearⅡのサンプルはパッケージに入っていません

RX-NET 関連のライブラリ“ lib****.a ”は，プロトコルやアプリケーション別に用意されています。次の表が，機能とライブラリ名の対応表です。

表 1－1　RX-NET関連のライブラリ

| プロトコル名 | ライブラリ名 | 機　能 |
|---|---|---|
| RX-NET（TCP/IP）部分［基本セット］ | librxnet.a | RX-NET（TCP/IP）プロトコル本体 |
| | libnodhcp.a | DHCP ダミー・エントリー |
| RX-NET（PPP）部分 | libppp.a | PPP ライブラリ |
| RX-NET（DNS）部分 | libdns_rslv.a | DNS リゾルバ・ライブラリ |
| RX-NET（SMTP）部分 | libsmtp.a | SMTP ライブラリ本体 |
| RX-NET（POP）部分 | libpop.a | POP ライブラリ |
| RX-NET（FTP）部分 | libftpc.a | FTP ライブラリ（クライアント側） |
| | libftps.a | FTP ライブラリ（サーバ側） |
| RX-NET（TELNET）部分 | libtelnetc.a | TELNET ライブラリ（クライアント側） |
| | libtelnets.a | TELNET ライブラリ（サーバ側） |
| RX-NET（DHCP）部分 | libdhcp.a | DHCP ライブラリ |
| RX-NET（WebServer）部分 | libwebs.a | Web サーバ・ライブラリ |

上記で“ RX-NET（TCP/IP）部分 ”は必ずリンクする必要があります。ただし“ libnodhcp.a ”は DHCP を使用する場合は“ libdhcp.a ”に差し替えてください。その他は使用するプロトコルやアプリケーションごとにリンクしてください。

## 1.3　提供サンプルのハードウエア構成

RX-NET のパッケージで用意しているサンプルは，マザーボード上に Ethernet コントローラや UART コントローラがある（マザーボードに Ethernet コントローラや UART コントローラが実装されている，または PCI バスや ISA バスにネットワーク・カードが装着されている）システムを想定しています。RX-NET の Ver.1.30 では，SolutionGear に付属しているサンプルもパッケージに入っており，RX-NET と共通化できる“マザーボードのドライバ”は SolutionGear に付属しているドライバを使用するように変更しました。つまり SolutionGear を使用することによって，RX-NET をすぐに評価することが可能です[注]。RX-NET が想定しているハードウエア構成は次のようになっています。

注：SolutionGear のサンプルが入っているのは V850E 版のみです。VR 版は SolutionGearⅡに対応しますが，現状では SolutionGearⅡのサンプルは RX-NET のパッケージに入っていません

表 1－2　サンプルとしてサポートしているハードウエア

| | | |
|---|---|---|
| マザーボード | RTE-MOTHER-A | （株）マイダス・ラボ製（NEC エレクトロニクスより販売） |
| CPU ボード | RTE-V850E/MA1-CB | （株）マイダス・ラボ製（V850E 用 RX-NET の場合） |
| | CMB-VR4131<br>CMB-VR5500 | （株）NEC エレクトロニクス製（VR 用 RX-NET の場合） |
| Ethernet コントローラ | NE2000 互換* | 各社 |
| | LAN91C96 | SMSC（Standard Micro Systems Corporation）製 |
| | i82558 | Intel 製（SolutionGear のサンプル内） |
| UART コントローラ<br>（RX-NET（PPP）のみ） | TL16550C | TEXAS INSTRUMENTS 製 |

※　“Solution Gear”に付属しているマザーボード“ RTE-MOTHER-A”には，Ethernet コントローラとして Intel 製の i82558 が搭載されています。そのため SolutionGear に付属している Ethernet コントローラのサンプルは i82558 を使用したものとなっています。一方，RX-NET のサンプルは，RTE-MOTHER-A 上の PCI バス，または ISA バスに NE2000 互換，LAN91C96 の Ethernet カードを装着し，それを使用したものとなります。

※　NE2000 互換

Ethernet カードのうち，Novell 社の純正ネットワーク・カード NE2000 に対して，ハードウエア的，およびソフトウエア的に互換性があるネットワーク・カードのことを指します。

NE2000 は I/O 命令転送方式を採用した 16 ビットの ISA カードで，互換製品が各社から販売されています。互換カードでも純正の NE2000 と同じように動作しますが，差別化のためにソフトウエア・コンフィギュレーション機能を備えていたり，付属ドライバやユーティリティを充実させたりしていることが多くなっています。なお，現在では ISA カードだけではなく，PCI カードとしても存在します。

また Ethernet コントローラ“LAN91C96 ” のサンプルは，このチップを搭載した ISA バス用 Ethernet カード“EVB90000-6 ”を使用しています。PPP に関しては，UART コントローラ“ TL16550C ”のサンプルを用意しています。

# 1.4　システム構築の手順

RX-NET を使用するシステムの構築手順は以下のようになります。

1. RX-NET コンフィギュレーションの設定（第 2 章）

   使用するリアルタイム OS（RX850 Pro / RX4000V4）の資源の設定，Ethernet コントローラの選択，ディバグ用出力関数の指定などを行います。これらはヘッダ・ファイルやプログラム中で設定します。

2. デバイス・ドライバの作成（第 3 章，第 4 章）

   “ Ethernet コントローラ ”，PPP を使用する場合は “ UART コントローラ ” のドライバを作成します。RX-NET のパッケージに含まれている “ NE2000 用ドライバ ” や “ LAN91C96 用ドライバ ”，“ TL16550 用ドライバ ” 以外のコントローラを使用する場合は，これらのドライバを新規に作成する，またはサンプルを参考に改造する必要があります。また使用するマザーボードが異なる場合も同様です。“ 1. RX-NET コンフィギュレーションの設定 ” で作成したソースと共にコンパイルし，1 つのオブジェクトにします。

3. ハードウエア・アクセス・ライブラリの作成（第 5 章）

   “ Ethernet コントローラ ” や “ UART コントローラ ” の初期化処理とは別に，使用するマザーボードに搭載されているメモリやバスなどの周辺機器の初期化処理が必要になります。Ver.1.30（V850E 版）では周辺装置の初期化／アクセスに，SolutionGear 付属のドライバを使用しています。

   マザーボード上のハードウエア（PCI バス・ISA バスなど）の初期化や，Ethernet コントローラとのアクセスに必要な I/O 関数をライブラリとして作成します。RX-NET にパッケージされているサンプルを SolutionGear 上で動作させる場合は，収録されているライブラリ（libbsd.a）をそのまま使用できますが，SolutionGear 以外の環境にドライバを移植する場合は，ライブラリを作成し直す必要があります。作成する関数等については “ 第 5 章 ハードウエア・アクセス・ライブラリ ” を参照してください。

4. アプリケーションの作成（第 6 章）

   ユーザ・アプリケーションを作成し，SolutionGear のドライバを収録したライブラリ（libbsd.a），RX-NET コンフィギュレーションの設定で作成されたオブジェクト，RX-NET 関連ライブラリをリンクして，ロード・モジュールを作成します。またリアルタイム OS（RX850 Pro / RX4000V4）も使用するので，リアルタイム OS 関連のライブラリもリンクします。

次章から上記 1 より順に具体的な構築方法を説明していきます。

# 第2章　RX-NETのコンフィギュレーション設定

RX-NET を動作させるためには，使用する資源数の情報やデバイス・コントローラの情報など，各種情報を設定する必要があります。これらの情報を“コンフィギュレーション情報”と呼びます。このコンフィギュレーション情報として設定する項目は次のとおりです。

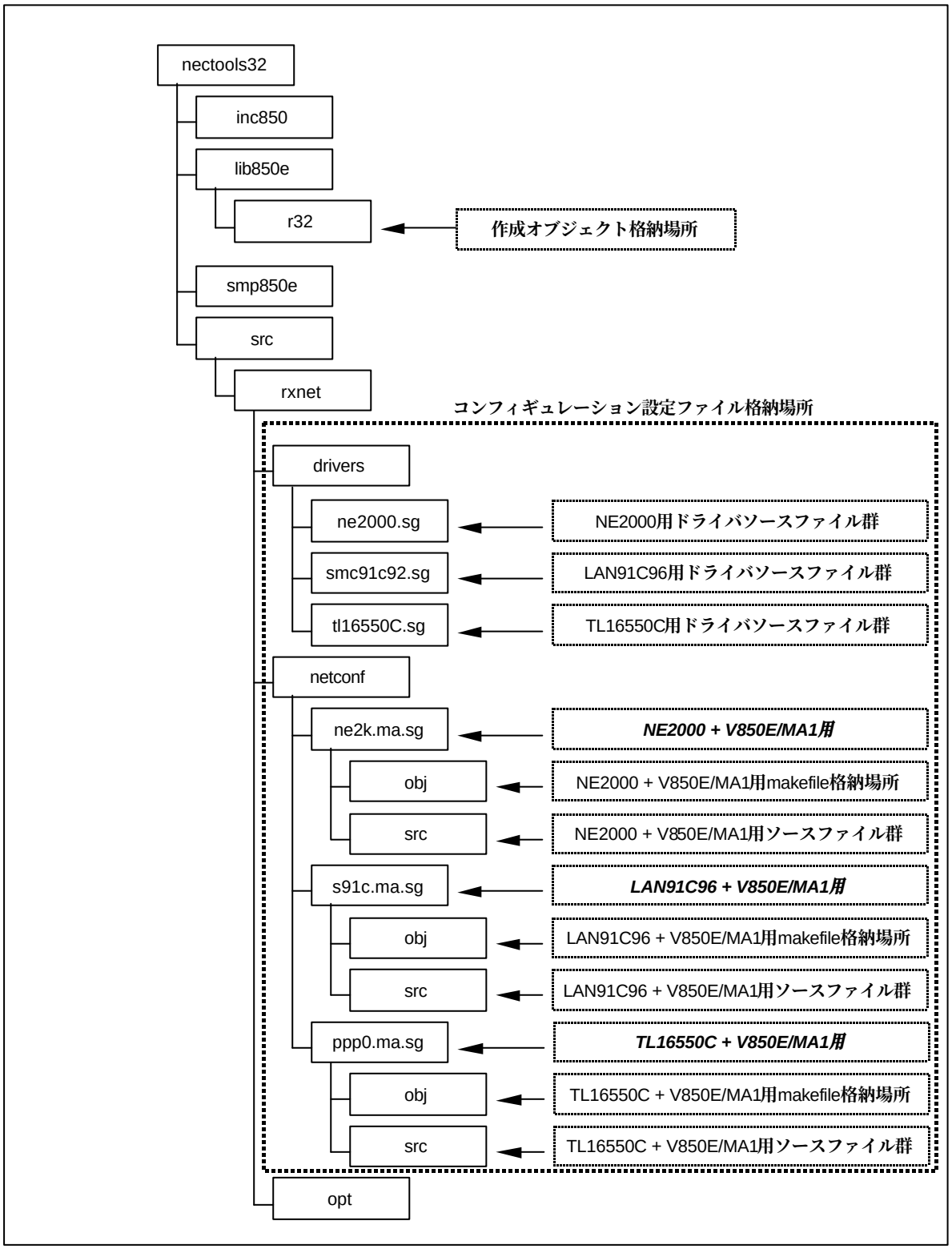
- RX850 Pro（RX4000V4）の資源に関する設定（2. 3，2. 4，2. 5）
- ソケットの最大数（2. 6）
- イベント・テーブルの最大数（2. 7）
- ヒープ・サイズ（2. 8）
- ディバグ用出力関数の設定（2. 9）
- ［PPP 以外の場合］RX-NET に組み込む Ethernet コントローラの設定（2. 10）
- ［PPP の場合］RX-NET（PPP）に組み込む UART コントローラ設定（2. 11）

## 2. 1　コンフィギュレーション設定ファイルの格納場所

コンフィギュレーション設定ファイルのソースファイル，ヘッダファイル，makefile やライブラリ本体が格納されている場所は，図 2－1 のようになります。

RX-NET では Ethernet コントローラとして NE2000 用と LAN91C96 用の 2 つのドライバ・サンプルを用意しているので，それぞれの名前がついたディレクトリが存在します。また RX-NET（PPP）では，UART コントローラとして TL16550C のドライバ・サンプルを用意しているので, この名前のついたディレクトリが存在します。

図 2－1　コンフィギュレーション設定ファイル格納場所




※　GHS 製コンパイラを使用する場合は“ lib850e ”が“ lib850e_ghs ”に，“ smp850e ”が“ smp850e_ghs ”に，“ obj ”が“ obj_ghs ”というディレクトリ名になります。

※　VR シリーズの場合，GHS 製コンパイラのみ対応ですが，ディレクトリ名は上記の“ inc850 ”が“ inc4100 ”“ inc5000 ”に，“ lib850_ghs￥r32 ”が“ lib4100_ghs￥little ”“ lib5000_ghs￥little ”に，“ smp850_ghs ”が“ smp4100_ghs ”“ smp5000_ghs ”に，そして CPU ボード名は“ CMB4131 ”“ CMB5500 ”になります。

## 2.2　コンフィギュレーション設定ファイル

コンフィギュレーション設定ファイルは次のものがあります。

表 2－1　コンフィギュレーション設定ファイル

| ファイル名 | 説 明 |
|---|---|
| src / rxnet / netconf / ne2k.ma.sg / src / **osdep.c** | RX-NET の RX850 Pro（RX4000V4）依存部 |
| src / rxnet / netconf / ne2k.ma.sg / src / **rxnetconf.c** | ・（PPP 以外の場合）RX-NET に組み込む Ethernet コントローラを決定するソース<br>・（PPP の場合）RX-NET（PPP）を PPP 接続で使用することを決定するソース |
| src / rxnet / netconf / ne2k.ma.sg / src / **rxnetconf.h** | ・（PPP 以外の場合）Ethernet コントローラを組み込む際のパラメータを定義（周期ハンドラ番号，タスク優先度，ソケット最大数，イベント・テーブル最大数，ヒープ・サイズ）<br>・（PPP の場合）UART コントローラを組み込む際のパラメータを定義（周期ハンドラ番号，タスク優先度，ソケット最大数，イベント・テーブル最大数，ヒープ・サイズ） |
| src / rxnet / netconf / ne2k.ma.sg / src / **user_printf.c** | ディバグ用の `printf` 関数のソース |
| src / rxnet / netconf / ne2k.ma.sg / src / **user_printf.h** | ディバグ用の `printf` 関数のヘッダ・ファイル |
| src / rxnet / netconf / ne2k.ma.sg / obj / **makefile** | NEC エレクトロニクス版オブジェクトを生成する makefile（GNUmake 用） |
| src / rxnet / netconf / ne2k.ma.sg / obj_ghs / **netconf.bld** | GHS 版オブジェクトを生成するビルド・ファイル |

※ 上記ディレクトリは NE2000 を使用したときのもので，SMC91C96 を使用する場合は“ ne2k.ma.sg ”を“ s91c.ma.sg ”と置き換えてください。

## 2.3　RX-NETが使用するリアルタイムOSの資源

RX-NET は RX850 Pro（RX4000V4）の資源を使用して動作します。そのため RX850 Pro（RX4000V4）関連のライブラリをリンクする必要があります。

また RX-NET が使用する RX の資源，およびサンプルにおけるそれらの資源の定義箇所は以下のとおりです。

・ 周期ハンドラ[注]（1 個）　～ 定義ファイル“ osdep.c / netconf.h ”

RX-NET を定期的に動作させるために使用します。この周期ハンドラの生成と処理内容は osdep.c 内で定義されています。このソースのオブジェクト（osdep.o）はライブラリ（サンプルでは rxnetcfg_ne2k_pci_ma.o など）として収録されています。

また周期ハンドラの生成には，RX850 Pro の場合は def_cyc システム・コールを，RX4000V4 の場合，acre_cyc サービス・コールを使用しています。なお，RX850 Pro ではハンドラの ID（サンプルでは“ OS_CYC_NO ”：rxnetconf.h で定義）をコンフィギュレーション設定ファイルの中で定義しておく必要があります。RX4000V4 の場合は ID が自動的に取得されます。

注：周期ハンドラは RX850 Pro では“ 周期起動ハンドラ ”と呼ばれますが，本マニュアルでは周期ハンドラと表記します。

・ タスク（2 個）　～ 定義ファイル“ osdep.c / **isrsub.c ”

RX-NET ではタスクを 2 個使用します。タスクの ID は自動的に割り当てられます。それぞれのタスクの役割は次のとおりです。

1. Ethernet コントローラ（PPP の場合は UART コントローラ）からの割り込みによって起動され，タスク・レベルでパケットの受信処理等を行います。このタスクは Ethernet コントローラドライバ（UART コントローラドライバ）が使用します。このタスクの生成と処理内容の記述はユーザが行います。サンプルでは，NE2000 互換の Ethernet コントローラの場合，/ rxnet / drivers / ne2000 内にある ne2kisrsub.c ファイルに記述されています。

2. 周期ハンドラから起床されるタスクで，RX-NET のタイマ管理をします。このタスクの生成と処理内容は osdep.c 内で定義されています。このソースのオブジェクト（osdep.o）はライブラリ（サンプルでは rxnetcfg_ne2k_pci_ma.o など）として収録されています。

・ 割り込みハンドラ（1 個以上）　～ 定義ファイル“ sys.cf（RX850 Pro）/ sys.cfg（RX4000V4）”

Ethernet コントローラ（PPP の場合は UART コントローラ）の割り込みによって起動される割り込みハンドラです。RX850 Pro の場合は“ 間接起動割り込みハンドラ ”，RX4000V4 の場合は“ 割り込みサービス・ルーチン ”を使用します。

なお，割り込みハンドラの生成と処理内容の記述はユーザが行います。RX850 Pro の場合は間接起動割り込みハンドラをコンフィギュレーション時に静的に生成・登録するか，初期化時に def_int システム・コールで動的に生成・登録します。RX4000V4 の場合は acre_isr サービス・コールで動的に生成・登録します。

これらの資源の中で，動的に生成されるものに関しては，RX850 Pro（RX4000V4）コンフィギュレーション・ファイルの設定で動的生成可能な状態にします。つまりタスクや周期ハンドラ，間接起動割り込みハンドラ（割り込みサービス・ルーチン）の最大生成数の指定を行います。具体的には次の項目になります。

- RX850 Pro の場合“ prtsk，maxtsk，maxcyc，maxint ”
- RX4000V4 の場合“ MAX_TSK，MAX_CYC，MAX_ISR ”

詳細については RX850 Pro（RX4000V4）マニュアル“ インストレーション編 ”を参照してください。

そして RX-NET では，次のシステム・コールを使用します。RX850 Pro（RX4000V4）コンフィギュレーション・ファイルにて，これらのシステム・コールを使用可能な状態にします。つまり“ SCT 情報 ”にシステム・コール情報を記述します。詳細は RX850 Pro（RX4000V4）マニュアル“ インストレーション編 ”を参照してください。

- RX850 Pro の場合
  cre_tsk，sta_tsk，dis_dsp，ena_dsp，get_tid，slp_tsk，wup_tsk，def_int，dly_tsk，def_cyc，ref_sys，get_tim

- RX4000V4 の場合
  acre_tsk，slp_tsk，wup_tsk，iwup_tsk，dly_tsk，acre_cyc，get_tim，get_tid，dis_dsp，ena_dsp，sns_dpn，acre_isr，del_isr

## 2.4　周期ハンドラ番号の設定

周期ハンドラは，RX-NET を定期的に動作させるために使用されます。周期ハンドラは RX850 Pro（RX4000V4）資源で，def_cyc システム・コールで動的生成するときに指定する番号がこれにあたります。この周期ハンドラ番号は“ rxnetconf.h ”で指定します。

```
#define OS_CYC_NO       1
```

## 2.5　RX-NETタスクの優先度の設定

RX-NET では 2 つのタスクを使用します。1 つはデバイス・ドライバ内でのデータ送受信タスク（***isrsub.c 内の**_Thread），もう 1 つは RX-NET タイマ・タスク（osdep.c 内の fns_cyc_task）です。前者のタスクではデバイス・ドライバによる送受信処理，後者のタスクではタイマ・タスクによる再送，遅延，タイムアウト処理を行います。この 2 つのタスクは同じ優先度で動作させる必要があるため，両者ともここで指定した優先度で生成します。なお，優先度数はなるべく高い優先度にしてください（可能ならば最高優先度）。

なお，サンプルでは，これらのタスクはこの優先度を元に cre_tsk システム・コールで動的に生成されます。このタスクの優先度は“ rxnetconf.h ”で指定します。

```
#define OS_RCVTSK_PRI   1
```

## 2.6　ソケット最大数の設定

RX-NET で使用するソケット数の最大数を指定します（ソケットに関する説明は“ 6. 5 ソケット ”を参照してください）。RX-NET 内部でソケットを 1 つ予約しているので，“ MAX_SOCKET － 1 ”個のソケットをアプリケーションから利用することができます。これを考慮して指定する必要があります。なお，実際にソケットの最大数を定義している箇所は rxnetconf.c ファイル内にあります。usrconf_tbl 構造体のメンバで指定する“ so_cnt ”です。

ソケット最大数は“ rxnetconf.h ”で指定します。

```
#undef  MAX_SOCKET
#define MAX_SOCKET      10
```

## 2.7　イベント・テーブル最大数の設定

イベント・テーブルは RX-NET 内で同期を取るために使用する“ 状態保持テーブル ”です。ソケット API が待ち状態になったり，ネットワークに問い合わせを行ったりしたとき，このイベント・テーブルの要素を 1 つ消費します。指定する値は「最大ソケット数＋通信を行うタスク数」を目安としてください。

イベント・テーブル最大数は“ rxnetconf.h ”で指定します。

```
#define MAX_EVENT       100
```

## 2.8　ヒープ・サイズの設定

ヒープは，プロトコル内部の作業領域として使用されるメモリ領域です。ディフォールトでは 150000 バイトを指定しています。システムによってはこの値を増減する必要がありますが，ヒープが足りない場合，通信のパフォーマンスが落ちたり，API 関数がエラーになったりします。

ヒープ・サイズは“ rxnetconf.h ”で指定します。

```
#undef  HEAP_SIZE
#define HEAP_SIZE       150000
```

実際にヒープ領域を定義している箇所は，rxnetconf.c ファイル内にあります。usrconf_tbl 構造体のメンバで指定する“ heap_size ”で，領域の確保は配列定義 heap_arr[]で行っています。

ヒープ使用量は次のようになります。

1. 静的なヒープ使用量

   ヒープは，1 つのブロックに対し，管理領域として“ 8 バイト ”を使用します。ヒープから領域確保を行うと“ データとして使用する量 ＋ 8 ”バイトずつ，残りの領域が減少します。また初期状態で管理領域を 1 つ持っているので，初期のヒープ・サイズは“ ヒープの配列サイズ － 8 ”となります。

   初期化時（so_initialize()コール時）のヒープの使用量は次のようになります。

   416 × MAX_SOCKET + 956 ［バイト］　（Ver.1.20 から変更）

   ll_config()と ll_route()のコール時にヒープの消費は行われません。また socket()と bind()のコール時もヒープの消費は行われません。

2. UDP 通信における動的なヒープ使用量

   1 パケットの送受信の度に

   align128(データサイズ + 44) + 208 ［バイト］

   の領域を確保します。

3. TCP 通信における動的なヒープ使用量について

　UDP と違って socket()の時にヒープを消費します（UDP は送受信以外でヒープ消費はなし）。コネクションの状態を管理する領域と再送タイマを管理する領域を作成するためです。サイズは

```
296 ［バイト］
```

となります。socket()時に確保したヒープは，closesock()時に解放されます。

　TCP の通信データの送信時，受信時には

```
align128(データサイズ + 56) + 208 ［バイト］
```

の領域確保を行います。

　また，TCP ではデータを含まない SYN や ACK の制御パケットが送受信されます。制御パケット 1 つにつき，

```
128 + 208 = 336 ［バイト］
```

のヒープを消費します。送信や受信のパケットが確保したヒープは，送信や受信が完了したときに解放されます。

　以上が基本的なヒープメモリ消費量です。ただし，ネットワーク・トラフィックやデータ量によってこの通りにならない場合もあります。

# 2.9　ディバグ出力関数の設定

ディバグ時に，ある部分である文字を出力したいような場合，このディバグ出力関数を使います。文字はメモリ上に書き込んだり，シリアルを通してターミナルに表示させたりします。ディバグ出力関数は“osdep.c”で指定します。下記の部分がその指定箇所になります。

```
int     (*os_printf)(char *msg, ...) = user_printf
```

ポインタ“os_printf”にディバグ出力に使用するprintf関数のポインタを指定します。すでにprintf関数が実装されている場合は，そちらを使用しても問題ありません。

## 2.9.1　user_printf関数

サンプルではuser_printf.cにあるuser_printf()を使用しているため，user_printfのポインタを代入しています。

user_printf()は，出力文字を1文字ずつ_BSDF_SendUART0関数で出力しています。SolutionGear以外の環境でuser_printf()関数を使用する場合は，変更する必要があります。サンプルでは，

- _BSDF_OpenUART0()　　サンプルではvarfunc.cで呼ばれています
- _BSDF_SendUART()　　サンプルではuser_printf.cで呼ばれています

を変更してください。（Ver.1.20から変更）

## 2. 10　Ethernetコントローラドライバのユーザ・オウン部設定

この節では Ethernet コントローラドライバのユーザ・オウン部の設定について説明します。UART コントローラの場合は“ 2. 11　UART コントローラドライバのユーザ・オウン部設定 ”を参照してください。

Ethernet コントローラに依存したファイルは以下のようになります。

表 2－2　Ethernetコントローラ依存ファイル

| ファイル名 | 説 明 |
|---|---|
| smc91c92.sg / **s91capi.c**<br>ne2000.sg / **ne2kapi.c** | RX-NET から直接呼び出されるデバイス・ドライバの API 部 |
| smc91c92.sg / **s91cisrsub.c**<br>ne2000.sg / **ne2kisrsub.c** | ドライバの割り込みハンドラから起床されるタスク部 |
| smc91c92.sg / **s91cutil.c**<br>ne2000.sg / **ne2kutil.c** | API 部から利用される関数群 |
| smc91c92.sg / **s91cutil.h**<br>ne2000.sg / **ne2kutil.h** | ドライバ全体で利用されるマクロ定義，プロトタイプ宣言 |

上記で上段は LAN91C96 用，下段は NE2000 用のファイルです。Ethernet コントローラドライバのユーザ・オウン部は，ハード構成によって適宜変更する必要があります。この節では，これらのユーザ・オウン部分でどのような設定を行っているかを説明します。ここでは“ NE2000 ”を使用する場合を例に説明します。

なお，新規に Ethernet コントローラドライバを作成するときは“ 第 3 章 Ethernet コントローラドライバ ”も参照してください。

ユーザ・オウン部で指定・変更する箇所は次の通りです。

- 初期化関数に渡されるパラメータの指定部分（ndevsw[]の設定）（2.10.1）
- I/O 入出力マクロの設定部分（2.10.2）
- ネットワーク割り込みハンドラ（割り込みサービスルーチン）の登録部分（2.10.3）
- ネットワーク割り込みハンドラ（割り込みサービスルーチン）の削除部分（2.10.4）
- 割り込みコントローラの初期化部分（2.10.5）
- 割り込み終了処理部分（2.10.6）

これらについて，順を追って説明していきます。

### 2.10.1　初期化関数に渡されるパラメータの指定

初期化関数は，デバイスを初期化するために呼ばれます。初期化のための情報を関数に渡すため，ndevsw[]という配列を定義しています。なお，この“初期化関数”自体（初期化関数へのポインタ）もこの配列のメンバとして定義されます。

コンフィギュレーション・オブジェクトのndevsw[]デバイス定義がrxnetconf.cにあります。その部分は以下のような記述になっています。

図 2－2　ndevsw[]デバイス定義

```
netdev   ndevsw[] = {
/*      nd_name,     nd_flags,     nd_devid,    0,     0,
 *      nd_p0,       nd_p1,        nd_p2,       nd_p3,
 *      nd_lladdr,   nd_stat,      nd_init,
 *      nd_updown,   nd_send,      nd_start,     nd_ioctl
 */
{       /* intra-machine device; must be in slot 0 */
       "im",         0,           0,            0,     0,
       0,            0,           0,            0,
       {0},          {0},         noent_init,
       noent_updown, im_send,     noent_start,   noent_ioctl
},
{      "ne2k",       0,           0,           0,     0,
       (u32)NE2K_P0, (u32)NE2K_P1, (u32)NE2K_P2, (u32)NE2K_P3,
       {AF_ETHER}, {0},          ne2k_init,
       ne2k_updown, en_scomm,    ne2k_start,   noent_ioctl
},
};
```

初期化関数には，このndevsw[]で指定されたメンバが，デバイス定義として引数に渡されます。つまりこれを使って初期化パラメータを渡すことができます。

上記のように，ndevsw[]配列は「“im”で始まる要素」と「“ne2k”で始まる要素」の2つの要素で構成されています。“im”で始まる要素はRX-NETが内部で使用するものです。ここは変更しないで下さい。デバイス・ドライバの設定は2番目の要素で行ないます。現状は2番目の要素のものが必ず使用されるため，複数定義しておいて選択するということはできません。

netdev[]配列のメンバ名は，上記サンプルのndevsw[]配列内のコメントに記述してあります。以下にその内容について説明します。なおサンプル内で“0”が設定されている箇所は，そのまま“0”を設定してください。

表 2－3　ndevswのメンバ名とその意味

| メンバ名 | 意 味 |
|---|---|
| nd_name | デバイスの名前を表す文字列を記述します。15 文字以内の任意の名前をつけてください。 |
| nd_p0 | 初期化パラメータ －0 |
| nd_p1 | 初期化パラメータ －1 |
| nd_p2 | 初期化パラメータ －2 |
| nd_p3 | 初期化パラメータ －3 |
| nd_lladdr | デバイスの物理層のアドレス・タイプを指定します。Ether の場合は {AF_ETHER} を設定してください。 |
| nd_init | 初期化関数のポインタ |
| nd_updown | 開始・終了関数のポインタ |
| nd_send | RX-NET からデバイス・ドライバを使用して送信する IP パケットに，物理層のヘッダを付加する関数を指定します。Ether の場合は en_scomm を設定します。 |
| nd_start | 送信関数のポインタ |
| nd_ioctl | 現在は未使用。何もしない関数のポインタを設定。 |

この中のメンバである「nd_p0，nd_p1，nd_p2，nd_p3」の内容は，ドライバ依存になっています。サンプル・ドライバでは「I/O ベース・アドレス」と「割り込み番号」を指定する箇所として使用しています。上記の NE2000 の例では「NE2K_P0, NE2K_P1, NE2K_P2, NE2K_P3」と指定されていますが，この値は rxnetconf.h で定義します。サンプルでは以下のような記述になっています。

```
#define NE2K_P0   -1    /* (not used) */
#define NE2K_P1   -1    /* (not used) */
#define NE2K_P2   -1    /* (not used) */
#define NE2K_P3   -1    /* (not used) */
```

Ver.1.30 ではこれらのパラメータは使用していないので，すべて“ -1 ”となっています。ドライバのヘッダに直接パラメータを指定しています。

### 2.10.2　I/O入出力マクロの設定

I/O 処理は，ハードウエアの構成によりメモリマップド I/O であったり，I/O の前後に特別な処理が必要であったりするため，ハードウエア環境によって変更する必要があります。サンプルのドライバでは，I/O 処理はドライバのヘッダ・ファイル内にマクロで記述しており，マクロ定義を書き換えることで I/O 処理を修正するようになっています。

ne2k.h 内の次の箇所が I/O マクロの設定箇所です。（Ver.1.20 より変更されています）

```
#define BIO_RB(addr)       _BSDF_PCIIO_inp8(addr)
#define BIO_RH(addr)       (*(volatile unsigned short*)(addr))
#define BIO_WB(addr, val)  ((*(volatile unsigned char*)(addr)) = (val))
#define BIO_WH(addr, val)  ((*(volatile unsigned short*)(addr)) = (val))
```

BIO_RB（8 ビット IN 命令），BIO_RH（16 バイト IN 命令），BIO_WB（8 ビット OUT 命令），BIO_WH（16 ビット OUT 命令）となっています。

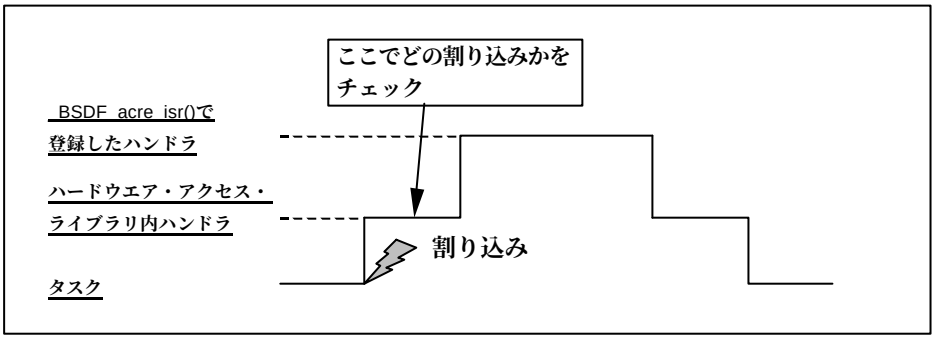
### 2.10.3　ネットワーク割り込みハンドラ（割り込みサービスルーチン）の登録

ハードウエア構成により，使用する割り込み要因が異なるので，割り込みハンドラ（割り込みサービスルーチン）の登録に関しても変更する必要があります。

1. V850E のサンプルの場合

割り込みハンドラは，RX850 Pro の割り込みハンドラとして登録する方法と，そうしない方法があります。“１つの外部割り込みにネットワーク割り込みしか入ってこない場合”は前者を，“１つの外部割り込みに複数の割り込みが入ってくる場合”は後者を選択します。NE2000 サンプルでは後者の方法を採っています。このボードは１つの外部割り込み端子に複数の割り込みが入ってくるためです。そのためハードウエア・アクセス・ライブラリ内の割り込みハンドラで，外部割り込みコントローラを見て，どの割り込みかを判断し，実行する割り込みハンドラを特定しています。つまりハードウエア・アクセス・ライブラリで外部割り込みを一括で処理する関数を経由し，各種割り込みを処理するという二段構成になっています。割り込み要因ごとの関数テーブルに値を設定する関数“ _BSDF_acre_isr()”を使用して割り込みハンドラを登録しています。

図 2－3　サンプル（NE2000）の割り込み処理の流れ




ne2kapi.c の次の箇所が割り込みハンドラの登録箇所です（コメント・ブロックで囲まれた部分）。

図 2－4　ネットワーク割り込みハンドラ定義箇所（V850E 版）

```
/*********************************************************/
/*****         Interrupt handler Definition            *****/
/*****             user own coding part                *****/

ne2k_isr_ID = (BSD_UINT32)_BSDF_acre_isr(INTNO_MB_PCISLOT1_INTA, ne2k_intr);

/*****                                                 *****/
/*****         Interrupt handler Definition            *****/
/*********************************************************/
```

上記のように，サンプル・ドライバのネットワーク割り込みハンドラの関数は ne2k_intr()になります。他のデバイスサンプルの場合，割り込みハンドラは，

*<devicename>_intr()*

となります。g_ndp->nd_p2 は，図 2－2 の ndevsw[]の初期化関数に渡されるパラメータで，この場合 NE2K_P2 の内容が入っています。

RX850 Pro の割り込みハンドラ登録関数で，直接ネットワーク割り込みハンドラの登録を行なう場合，つまり 1 つの外部割り込みにネットワーク割り込みしか入ってこない場合は，次のようにします。この場合，NE2K_P2 を該当する V850E の割り込み要因番号に変更する必要があります。

図 2－5　def_int を使ったネットワーク割り込みハンドラ定義例（NEC エレクトロニクス製ツール）

```
/*********************************************************/
/*****        Interrupt handler Definition           *****/
/*****            user own coding part               *****/
{
#pragma section const begin
extern const int _tp_TEXT;
extern const int _gp_DATA;
#pragma section const end
T_DINT pk_dint;

pk_dint.intatr= TA_HLNG | TA_DPIC | TA_GPID;
pk_dint.inthdr= ne2k_intr;
pk_dint.gp    = &_gp_DATA;
pk_dint.tp    = &_tp_TEXT;
def_int(NE2K_INTR_NUM, &pk_dint);
}
/*****                                               *****/
/*****        Interrupt handler Definition           *****/
/*********************************************************/
```

なお，GHS 製ツールの場合は，

```
#pragma section const begin
extern const int _tp_TEXT;
extern const int _gp_DATA;
#pragma section const end
```

の部分を

```
extern int _gp;
```

と置き換えてください。

2.　VR のサンプルの場合

VR の場合，RX4000V4 の割り込みサービスルーチンとして登録する方法を採ります。ne2kapi.c の次の箇所が割り込みハンドラの登録箇所です（コメント・ブロックで囲まれた部分）。

図 2－6　ネットワーク割り込みハンドラ定義箇所（VR 版）

```
/*********************************************************/
/*****        Interrupt handler Definition           *****/
/*****            user own coding part               *****/

pk_cisr.isratr= TA_HLNG;
pk_cisr.exinf = 0;
pk_cisr.gp    = (VP)getgp();
pk_cisr.tp    = NULL;
pk_cisr.isr   = (void(*)())ne2k_intr;
pk_cisr.intno = INTNO_PCI_D;
NE2KIntHdlID  = acre_isr(&pk_cisr);

/*****                                               *****/
/*****        Interrupt handler Definition           *****/
/*********************************************************/
```

上記のように，サンプル・ドライバのネットワーク割り込みハンドラの関数は ne2k_intr()になります。他のデバイスサンプルの場合，割り込みハンドラは，

*<devicename>_intr()*

となります。g_ndp->nd_p2 は，図 2－2 の ndevsw[]の初期化関数に渡されるパラメータで，この場合 NE2K_P2 の内容が入っています。

### 2.10.4　ネットワーク割り込みハンドラ（割り込みサービスルーチン）の削除

ハードウエア構成により，使用する割り込み要因が異なるので，割り込みハンドラ（割り込みサービスルーチン）の削除の方法も変更する必要があります。

1. V850E のサンプルの場合

割り込みハンドラは，RX850 Pro の割り込みハンドラとして登録したものを削除する方法と，そうしない方法があります。これはハンドラ登録に応じた方法で削除する必要があります。サンプルでは，登録時に割り込みハンドラとして登録していないので，後者の方法を採っています。サンプル・ドライバの削除箇所では，割り込みハンドラ登録時に使用した_BSDF_acre_isr()を使って，登録した割り込みハンドラの箇所に空の関数を再設定して割り込みハンドラを削除しています。

ne2kapi.c の次の箇所が割り込みハンドラの削除箇所です（コメント・ブロックで囲まれた部分）。

図 2－7　ネットワーク割り込みハンドラ削除箇所（V850E 版）

```
/*********************************************************/
/*****         Interrupt handler Undefinition          *****/
/*****             user own coding part                *****/

_BSDF_dis_int(INTNO_MB_PCISLOT1_INTA);
_BSDF_del_isr(INTNO_MB_PCISLOT1_INTA, ne2k_isr_ID);

/*****                                                 *****/
/*****         Interrupt handler Undefinition          *****/
/*********************************************************/
```

RX850 Pro の割り込みハンドラとして登録したものを削除する方法は次のとおりです。この場合，NE2K_P2 を該当する V850E の割り込み要因番号に変更する必要があります。

ne2kapi.c の次の箇所が割り込みハンドラの削除箇所です（コメント・ブロックで囲まれた部分）。

図 2－8　def_int システム・コールを使ったネットワーク割り込みハンドラ削除例

```
/*********************************************************/
/*****         Interrupt handler Undefinition          *****/
/*****             user own coding part                *****/

def_int(NE2K_INTR_NUM, NADR);

/*****                                                 *****/
/*****         Interrupt handler Undefinition          *****/
/*********************************************************/
```

2.　VR のサンプルの場合

VR の場合，RX4000V4 の割り込みサービスルーチンとして登録されているので，同じくサービスルーチンとして削除します。

NE2000 を使用したサンプルで ne2kapi.c の次の箇所が割り込みハンドラの削除箇所です（コメント・ブロックで囲まれた部分）。

図 2－9　ネットワーク割り込みサービスルーチン削除例（VR 版）

```
/*********************************************************/
/*****         Interrupt handler Undefinition          *****/
/*****             user own coding part                *****/

/* 割り込み禁止 */
outp16(GIUINTENL, inp16(GIUINTENL) & ~GPIOINT7);
del_isr(NE2KIntHdlID);

/*****                                                 *****/
/*****         Interrupt handler Undefinition          *****/
/*********************************************************/
```

## 2.10.5　割り込みコントローラの初期化

1.　V850E のサンプルの場合

割り込みコントローラの種類，および割り込みコントローラが接続されている場所（PCI バス，ISA バス，ローカルバス）によって，初期化の方法も異なってきます。割り込みコントローラの初期化は nk2k_updown()内に記述されている Ne2kInit()を呼び出すことによって行われています。Ne2kInit()は ne2kutil.c にあり，実際の初期化内容が記述されています。

ne2kapi.c の次の箇所は，NE2000 の初期化と割り込みハンドラの登録が終わった後に実行される処理内容で，割り込みを許可するコードが書かれています。（コメント・ブロックで囲まれた部分）。

図 2－10　割り込みコントローラに対する割り込み許可部分（V850E 版）

```
/*********************************************************/
/*****         Interrupt Controler Initialization      *****/
/*****               user own coding part              *****/

_BSDF_ena_int(INTNO_MB_PCISLOT1_INTA);

/*****                                                 *****/
/*****         Interrupt Controler Initialization      *****/
/*********************************************************/
```

2.　VR のサンプルの場合

割り込みコントローラの初期化は nk2k_updown()内に記述されている Ne2kInit()を呼び出すことによって行われています。Ne2kInit()は ne2kutil.c にあり，実際の初期化内容が記述されています。

ne2kapi.c の次の箇所は，NE2000 の初期化と割り込みハンドラの登録が終わった後に実行される処理内容で，割り込みを許可するコードが書かれています（コメント・ブロックで囲まれた部分）。

図 2－11　割り込みコントローラの初期化（VR 版）

```
/*********************************************************/
/*****        Interrupt Controler Initialization     *****/
/*****              user own coding part             *****/

/*interrupt for PCI 3 slot*/

/*set level mode*/
outp16(GIUINTTYPL, inp16(GIUINTTYPL) & ~GPIOINT7);
/*set low level mode*/
outp16(GIUINTALSELL, inp16(GIUINTALSELL) & ~GPIOINT7);

/*clear*/
outp16(GIUINTSTATL, inp16(GIUINTSTATL) | GPIOINT7);

/*割り込み許可*/
outp16(GIUINTENL, inp16(GIUINTENL) | GPIOINT7);
outp16(MGIUINTLREG, inp16(MGIUINTLREG) | GPIOINT7);
outp16(MSYSINT1REG, inp16(MSYSINT1REG) | GIUINTR);

/*****                                                *****/
/*****        Interrupt Controler Initialization     *****/
/*********************************************************/
```

### 2.10.6 割り込み終了処理

割り込みコントローラの種類，および割り込みコントローラが接続されている場所（PCI バス，ISA バス，ローカルバス）によって，割り込み終了時の処理も異なってきます。

サンプルでは特に何もしていません。必要な場合，例えば RTE-MOTHER-A の ISA 割り込み要求をクリアするなどの処理が必要なときは，それ相当の記述が必要です。

ne2kapi.c の次の箇所が割り込み終了箇所です（コメント・ブロックで囲まれた部分）。

図 2－12　割り込みコントローラの終了処理

```
/*********************************************************/
/*****                End of Interrupt                *****/
/*****              user own coding part              *****/

/* nothing to do */

/*****                                                *****/
/*****                End of Interrupt                *****/
/*********************************************************/
```

なお，VR のサンプルでも，割り込み終了処理では何もしていません。

# 2. 11　UARTコントローラドライバのユーザ・オウン部設定

この節では PPP 接続のときに使用する UART コントローラドライバのユーザ・オウン部の設定について説明します。Ethernet コントローラの場合は“ 2. 10 Ethernet コントローラドライバのユーザ・オウン部設定 ”を参照してください。

UART コントローラに依存したファイルは以下のようになります。

表 2－4　UARTコントローラ依存ファイル（PPP）

| ファイル名 | 説 明 |
|---|---|
| tl16550c.sg / **com.c** | RX-NET（PPP）から直接呼び出されるデバイス・ドライバの API 部 |
| tl16550c.sg / **com.h** | TL16550C ドライバのヘッダ・ファイル |

UART コントローラドライバのユーザ・オウン部は，ハードウエア構成によって適宜変更する必要があります。ここでは UART コントローラドライバのユーザ・オウン部でどのような設定を行っているかを説明します。サンプルで提供している“ TL16550C ”を使用する場合を例に説明します（V850E における例で説明します）。

なお，新規に UART コントローラドライバを作成するときは“ 第 4 章 UART コントローラドライバ ”も参照してください。

ユーザ・オウン部で指定・変更する箇所は次の通りです。

- 初期化関数に渡されるパラメータの指定部分（ndevsw[]の設定）（2.11.1）
- I/O 入出力マクロの設定部分（2.11.2）
- レジスタ・マップの設定部分（2.11.3）
- ネットワーク割り込みハンドラの登録部分（2.11.4）
- ネットワーク割り込みハンドラの削除部分（2.11.5）
- 割り込みコントローラ操作関数部分（2.11.7）

これらについて順を追って説明していきます。

### 2.11.1　初期化関数に渡されるパラメータの指定

初期化関数は，デバイスを初期化するために呼ばれます。初期化のための情報を関数に渡すため，ndevsw[]という配列を定義しています。なお，この“初期化関数”自体（初期化関数へのポインタ）もこの配列のメンバとして定義されます。

コンフィギュレーション・オブジェクトのndevsw[]デバイス定義がrxnetconf.cにあります。その部分は以下のような記述になっています。

図 2－13　ndevsw[]デバイス定義

```
netdev   ndevsw[] = {
/*      nd_name,     nd_flags,     nd_devid,    0,     0,
 *      nd_p0,       nd_p1,        nd_p2,       nd_p3,
 *      nd_lladdr,   nd_stat,      nd_init,
 *      nd_updown,   nd_send,      nd_start,     nd_ioctl
 */
{       /* intra-machine device; must be in slot 0 */
       "im",         0,          0,            0,     0,
       0,            0,          0,            0,
       {0},          {0},        noent_init,
       noent_updown, im_send,    noent_start,   noent_ioctl
},
{       /* intra-machine device; must be in slot 0 */
       "ppp0",       0,          0,            0,     0,
       PPP0_P0,      PPP0_P1,    PPP0_P2,      PPP0_P3,
       {AF_PPP},     {0},        ppp_init,
       ppp_updown,   ppp_scomm,  ppp_dstart,   noent_ioctl
},
};
```

初期化関数には，このndevsw[]で指定されたメンバがデバイス定義として引数に渡されます。これを使って初期化パラメータを渡すことができます。

netdev構造体の重要メンバの名前は上記サンプルのndevsw[]配列内のコメントに記述してあります。以後はコメントに書かれているメンバに設定する内容について説明します。なおサンプル内で“0”が設定されている箇所はそのまま0を設定してください。またndevsw[]の第一要素，つまり，

```
{       /* intra-machine device; must be in slot 0 */
       "im",         0,          0,            0,     0,
       0,            0,          0,            0,
       {0},          {0},        noent_init,
       noent_updown, im_send,    noent_start,   noent_ioctl
},
```

は，削除／変更しないでください。

以下に，コメントに書かれているメンバに設定する内容について説明します。

表 2－5　ndevswのメンバ名とその意味

| メンバ名 | 意 味 |
|---|---|
| nd_name | デバイスの名前を表す文字列を記述します。RX-NET（PPP）を使用する場合は“ppp0”と記述します。 |
| nd_p0 | 初期化パラメーター0。ドライバ関数の第一引数 ndp 経由でこのメンバにアクセスできます。ndp 経由でドライバ関数に渡すパラメータがない場合は未使用です |
| nd_p1 | 初期化パラメーター1。ドライバ関数の第一引数 ndp 経由でこのメンバにアクセスできます。ndp 経由でドライバ関数に渡すパラメータがない場合は未使用です |
| nd_p2 | 初期化パラメーター2。ドライバ関数の第一引数 ndp 経由でこのメンバにアクセスできます。ndp 経由でドライバ関数に渡すパラメータがない場合は未使用です |
| nd_p3 | 初期化パラメーター3。ドライバ関数の第一引数 ndp 経由でこのメンバにアクセスできます。ndp 経由でドライバ関数に渡すパラメータがない場合は未使用です |
| nd_lladdr | デバイスの物理層のアドレス・タイプを指定します。RX-NET（PPP）を使用する場合は，｛AF_PPP｝を設定してください。 |
| nd_init | 初期化関数のポインタです。RX-NET（PPP）を使用する場合は ppp_init と記述します。 |
| nd_updown | 開始・終了関数のポインタです。RX-NET（PPP）を使用する場合は ppp_updown と記述します。 |
| nd_send | IP パケットに，物理層のヘッダを付加する関数を指定します。RX-NET（PPP）を使用する場合は ppp_scomm と記述します。 |
| nd_start | 送信関数のポインタです。RX-NET（PPP）を使用する場合は ppp_dstart と記述します。 |
| nd_ioctl | 現在は未使用です。RX-NET（PPP）を使用する場合は noent_ioctl と記述します。 |

なお，ここで指定する“ppp_init”“ppp_updown”“ppp_dstart”“ppp_scomm”はRX-NET（PPP）ライブラリに，“noent_ioctl”はRX-NETのライブラリに含まれています。ユーザで定義する必要はありません。

この中のメンバである「nd_p0，nd_p1，nd_p2，nd_p3」は「I/Oベース・アドレス」と「割り込み番号」を指定する箇所で，ドライバ依存になります。上記の TL16550C の例では「PPP0_P0，PPP0_P1，PPP0_P2，PPP0_P3」と指定されていますが，この値を rxnetconf.h で定義します。サンプルでは以下のような記述になっています。

```
#define PPP0_P0    0            /* not used */
#define PPP0_P1    0            /* not used */
#define PPP0_P2    0            /* not used */
#define PPP0_P3    0            /* not used */
```

Ver.1.30では，これらのパラメータは使用しておらず，すべて“0”となっています。

### 2.11.2　I/O入出力マクロの設定

I/O 処理は，ハードの構成によりメモリマップド I/O であったり，I/O の前後に特別な処理が必要であったりするため，ハードウエア環境によって変更する必要があります。サンプルのドライバでは，I/O 処理はドライバのヘッダ・ファイル内にマクロで記述しており，マクロ定義を書き換えることで I/O 処理を修正するようになっています。com.h 内の次の箇所が I/O マクロの設定箇所です。（Ver.1.20 より変更されています）

```
#define BIO_RB(addr)      (*(volatile unsigned char *) addr)
#define BIO_RH(addr)      (*(volatile unsigned short *) addr)
#define BIO_WB(addr, val) ((*(volatile unsigned char *) addr) = val)
#define BIO_WH(addr, val) ((*(volatile unsigned short *) addr) = val)
```

BIO_RB（8 ビット IN 命令），BIO_RH（16 バイト IN 命令），BIO_WB（8 ビット OUT 命令），BIO_WH（16 ビット OUT 命令）となっています。

サンプルでは RTE-V850E/MA1-CB の環境がメモリマップド I/O なので, RTE-MOTHER-A のビット操作が必要な 8 ビット IN 命令以外は, メモリ・アクセスのマクロになっています。なお, RTE-V850E/MA1-CB では GBUS 経由のバイトアクセス前にフラグを立てる必要あります。詳しくは RTE-V850E/MA1-CB のマニュアルを参照してください。

### 2.11.3　レジスタ・マップの設定

サンプル・ドライバのハードウエア環境では，TL16550C のレジスタを，チャネルごとの先頭アドレスから 0x10 置きに I/O 空間に配置しています。使用しているボードでレジスタの配置方法が異なる場合は，各レジスタのアドレスも変更します。com.h 内の次の箇所が TL16550C のレジスタの設定箇所です。

図 2－14　TL16550C のレジスタ定義箇所

```
#define  TLI0Base      0x7807000       /* UART I/O Base Address */

/* UART registers (COM I/O address offsets) */
#define  UART_DLL      (TLI0Base + 0x00) /* divisor latch(LS) (DLAB==1) */
#define  UART_DLM      (TLI0Base + 0x10) /* divisor latch(MS) (DLAB==1) */
#define  UART_RBR      UART_DLL          /* Receiver Buffer Register
                                            (read only) (DLAB==0)       */
#define  UART_THR      UART_DLL          /* Transmitter Holding Register
                                            (write only) (DLAB==0)      */
#define  UART_IER      (TLI0Base + 0x10) /* Interrupt Enable Register
                                            (DLAB==0)                    */
#define  UART_IIR      (TLI0Base + 0x20) /* Interrupt Ident. Register
                                            (read only) (DLAB==0)       */
#define  UART_FCR      (TLI0Base + 0x20) /* FIFO contorol register
                                            (write only) (TL16550 only) */
#define  UART_LCR      (TLI0Base + 0x30) /* Line Contorol Register       */
#define  UART_MCR      (TLI0Base + 0x40) /* Modem Contorol Register     */
#define  UART_LSR      (TLI0Base + 0x50) /* Line Status Register         */
#define  UART_MSR      (TLI0Base + 0x60) /* Modem Status Register        */
#define  UART_SCR      (TLI0Base + 0x70) /* SCratch Register             */
```

### 2.11.4　ネットワーク割り込みハンドラの登録

ハードウエア構成により，使用する割り込み要因が異なるので，割り込みハンドラの登録に関しても変更する必要があります。

割り込みハンドラは，RX850 Pro の割り込みハンドラとして登録する方法と，そうしない方法があります。サンプルでは前者の方法をとっています。V850E と TL16550C コントローラが直結しているような場合は，これに該当します。この場合割り込み要因番号（UART_INTR_NO）を，該当する V850E の割り込み要因番号に変更する必要があります。

com.c の serial_updown()内の次の箇所が，割り込みハンドラの登録箇所です（コメント・ブロックで囲まれた部分）。

図 2－15　ネットワーク割り込みハンドラ定義箇所

```
/*************************************************************/
/*****         Interrupt handler Definition            *****/
/*****             user own coding part                *****/

/* 割り込みハンドラを登録 */
def_int(UART_INTR_NO, &pk_uart0_int);

/* CPU レベルで RTE-V850E/MA1 ボード上 PIC 割り込み要因のマスクを解除 */
SFR_P00IC0 = 0x03;

/* PIC レベルで UART の割り込みマスクを解除 */
pic1_disable;
BIO_WB(_BSDC_CB_PIC_ADR_INT0M, BIO_RB(_BSDC_CB_PIC_ADR_INT0M) | 0x02);
pic1_enable;

/*****                                                 *****/
/*****         Interrupt handler Definition            *****/
/*************************************************************/
```

なお，pk_uart0_int のメンバー設定は com.c 内の先頭部分で次のように行われています。

図 2－16　def_int を使ったネットワーク割り込みハンドラ定義例（NEC エレクトロニクス製ツール）

```
T_DINT pk_uart0_int = {
    TA_HLNG | TA_DPID | TA_DPIC, (FP)serial_intr, &_gp_DATA, &_tp_TEXT
};
```

図 2－17　def_int を使ったネットワーク割り込みハンドラ定義例（GHS 製ツール）

```
T_DINT pk_uart0_int = {
       TA_HLNG | TA_DPID | TA_DPIC, (FP) serial_intr, &_gp, 0
};
```

上記で NEC エレクトロニクス製ツールの“ _gp_DATA”“ _tp_TEXT ”，GHS 製ツールの“ _gp ”は com.h にて定義されています。com.c 内で com.h をインクルードしてください。

### 2.11.5　ネットワーク割り込みハンドラの削除

ハードウエア構成により，使用する割り込み要因が異なるので，割り込みハンドラの削除の方法も変更する必要があります。

割り込みハンドラは，RX850 Pro の割り込みハンドラとして登録したものを削除する方法と，そうしない方法があります。これはハンドラ登録に応じた方法で削除する必要があります。サンプルでは，登録時に割り込みハンドラとして登録しているので，前者の方法をとっています。

com.c の serial_updown 関数内の次の箇所が，割り込みハンドラの削除箇所です（コメント・ブロックで囲まれた部分）。

図 2－18　ネットワーク割り込みハンドラ削除箇所

```
/*********************************************************/
/*****        Interrupt handler Undefinition          *****/
/*****            user own coding part                *****/
{
/* PIC レベルで UART の割り込みをマスク */
pic1_disable;
BIO_WB(_BSDC_CB_PIC_ADR_INT0M, BIO_RB(_BSDC_CB_PIC_ADR_INT0M) & ~0x02);
pic1_enable;

/* CPU レベルで RTE-V850E/MA1 ボード上 PIC 割り込み要因をマスク */
SFR_P00IC0 = 0x43;

/* 割り込みハンドラを削除 */
def_int(UART_INTR_NO, (T_DINT*) NADR);
}
/*****                                                 *****/
/*****        Interrupt handler Undefinition          *****/
/*********************************************************/
```

### 2.11.6 割り込みコントローラの初期化

割り込みコントローラの初期化やバッファの初期化，モデムステータス割り込みのクリアなどを行います。

com.c の次の箇所は，UART からの通信割り込みを PIC レベルでマスクする処理を行っています。（コメント・ブロックで囲まれた部分）。この後に UART の初期化，バッファの初期化，モデムステータス割り込みの初期化を行っています。

図 2－19　UART からの通信割り込みをマスクする処理部分

```
/*********************************************************/
/*****        Interrupt Controler Initialization     *****/
/*****            user own coding part               *****/
{
/* PIC レベルで UART の割り込みをマスク */
pic1_disable;
BIO_WB(_BSDC_CB_PIC_ADR_INT0M, BIO_RB(_BSDC_CB_PIC_ADR_INT0M) & ~0x02);
pic1_enable;
}
/*****                                                   *****/
/*****        Interrupt Controler Initialization     *****/
/*********************************************************/
```

### 2.11.7　割り込みコントローラ操作マクロ

サンプルでは，割り込みコントローラを操作するマクロを用意しています。このマクロは割り込みコントローラに依存する部分なので修正します。サンプルでは“RTE-MOTHER-A”の GINT1 割り込みを使用する場合の設定が書かれています。

割り込みコントローラ依存部分のマクロの名前と内容，そして定義されているサンプル・ソースは下記のとおりです。（Ver.1.20 より変更されています）

表 2－6　割り込みコントローラ依存部の関数／マクロ

| マクロ名 | プロトタイプ／内容 | |
|---|---|---|
| pic1_enable | 機　能 | 割り込み番号で指定された割り込みを許可状態にする |
| | 形　式 | #define　pic1_enable　BIO_WB(_BSDC_CB_PIC_ADR_INTEN, 0x03) |
| | ソース | com.h |
| pic1_disable | 機　能 | 割り込み番号で指定された割り込みを禁止状態にする |
| | 形　式 | #define　pic1_disable　BIO_WB(_BSDC_CB_PIC_ADR_INTEN, 0x02) |
| | ソース | com.h |
| DISINTR | 機　能 | 割り込みを禁止状態にする |
| | 形　式 | #define　DISINTR　{SFR_IMR0 = (0x10 \| SFR_IMR0);} |
| | ソース | com.h |
| RESINTR | 機　能 | 割り込みを許可状態にする |
| | 形　式 | #define　RESINTR　{SFR_IMR0 = ((u8)~0x10 & SFR_IMR0);} |
| | ソース | com.h |

# 第3章　Ethernetコントローラドライバ

サンプルに付属していない Ethernet コントローラを使用する場合，Ethernet コントローラのドライバを新規に作成する必要があります。この節では Ethernet コントローラのドライバ作成に必要な情報を示します。

## 3. 1　ドライバの新規作成

### 3.1.1　デバイス・ドライバの構成要素

デバイス・ドライバを自作する場合，下記の要素を実装します。

表 3－1　デバイス・ドライバ（Ethernetコントローラ）の構成要素

| 構成要素 | 機　能 |
|---|---|
| 初期化関数 | 開始関数を呼ぶ前に実行する必要がある初期化処理を行います。 |
| 開始・終了関数 | デバイスを“有効にする／非有効にする”際に呼び出されます。<br>enable 時（ll_config()呼び出し時）に“ネットワーク割り込みハンドラの登録”と“ネットワークタスクの生成と起動”を行います。<br>disable 時（ll_unconfig()呼び出し時）に“ネットワーク割り込みハンドラの削除”を行います。 |
| 送信関数 | RX-NET の送信処理から呼び出されます。<br>引数で渡されたパケットを送信します。 |
| ネットワーク割り込みハンドラ | 受信割り込み，送信完了割り込み，エラー割り込み等を処理します。 |
| ネットワークタスク | ネットワーク割り込みハンドラから呼び出されます。<br>受信割り込み後なら，Ethernet コントローラからパケットを取り出して RX-NET に渡します。<br>送信完了割り込み後であれば，RX-NET の送信処理を呼び出し，後続の送信を促します。 |

### 3.1.2　新規ドライバの作成手順

次の手順でデバイス・ドライバを作成してください。

1. デバイス・ドライバのディレクトリを作成

   デバイス・ドライバの作成場所は，

   nectools32 / src / rxnet / drivers

   以下に作成してください。この下にデバイス・ドライバ名を使ったディレクトリ名を作成するとよいでしょう。例えば LAN91C96 の場合は s91c96 というようなディレクトリ名です。

2. デバイス・ドライバのディレクトリを作成

   作成したディレクトリ下にドライバ関数のソースを作成します。作成する関数の仕様については，“3.2 ドライバの関数の仕様”を参照してください。

3. RX-NET コンフィギュレーション・オブジェクトの作成

作成するデバイス・ドライバを指定した“コンフィギュレーション・オブジェクト”を作成します。ソースをコンパイルし，nectools32 / lib850e / r32（GHS 版の場合 nectools32 / lib850e_ghs / r32） の下にオブジェクトを配置します。

いちばん簡単な方法は，サンプルの makefile における nectools32 / src / rxnet / netconf / s91c.ma 等の設定を，デバイス・ドライバの部分だけを差し替えて構築する方法です。

### 3.1.3 デバイス・ドライバ関数の登録

作成したデバイス・ドライバ関数を RX-NET から呼び出せるようにするために，その登録を行ないます。

デバイス・ドライバ関数の登録は RX-NET コンフィギュレーション・オブジェクト中のデバイス管理テーブルに関数ポインタを設定することで行ないます。

RX-NET のデバイス管理テーブルは“netdev 構造体”の配列“ndevsw[]”で実装しています。netdev 構造体の定義は nectools32 / inc850 / rxnet / netdev.h で行なわれています。そしてデバイス管理テーブル ndevsw[]は，サンプルの LAN91C96 の場合，nectools32 / src / rxnet / netconf / s91c.ma / src / rxnetconf.c に定義されています。

図 3－1　ndevsw[]デバイス定義

```
netdev   ndevsw[] = {
/*      nd_name,     nd_flags,     nd_devid,    0,     0,
 *      nd_p0,       nd_p1,        nd_p2,       nd_p3,
 *      nd_lladdr,   nd_stat,      nd_init,
 *      nd_updown,   nd_send,      nd_start,     nd_ioctl
 */
{       /* intra-machine device; must be in slot 0 */
       "im",         0,          0,            0,     0,
       0,            0,          0,            0,
       {0},          {0},        noent_init,
       noent_updown, im_send,    noent_start,   noent_ioctl
},
{      "s91c",       0,          0,           0,     0,
       S91C_P0,      S91C_P1,    S91C_P2,     S91C_P3,
       {AF_ETHER}, {0},          S91C_init,
       S91C_updown, en_scomm,    S91C_start,   S91C_ioctl
}
};
```

上記のように，ndevsw[]配列は“im”で始まる要素と“s91c”で始まる要素の 2 つの要素で構成されています。“im”で始まる要素は RX-NET が内部で使用するものです。ここは変更しないで下さい。デバイス・ドライバの設定は 2 番目の要素で行ないます。上記サンプルで網掛け部分が変更する箇所になります。

netdev[]配列のメンバ名は，上記サンプルの ndevsw[]配列内のコメントに記述してあります。以下にその内容について説明します。なおサンプル内で“0”が設定されている箇所は，そのまま“0”を設定してください。

表 3－2　ndevsw[]のメンバ名とその意味

| メンバ名 | 意 味 |
|---|---|
| nd_name | デバイスの名前を表す文字列を記述します。15 文字以内の任意の名前をつけてください。 |
| nd_p0 | 初期化パラメータ－0 |
| nd_p1 | 初期化パラメータ－1 |
| nd_p2 | 初期化パラメータ－2 |
| nd_p3 | 初期化パラメータ－3 |
| nd_lladdr | デバイスの物理層のアドレス・タイプを指定します。Ether の場合は {AF_ETHER} を設定してください。 |
| nd_init | 初期化関数のポインタ |
| nd_updown | 開始・終了関数のポインタ |
| nd_send | RX-NET からデバイス・ドライバを使用して送信する IP パケットに，物理層のヘッダを付加する関数を指定します。Ether の場合は en_scomm を設定します。 |
| nd_start | 送信関数のポインタ |
| nd_ioctl | 現在は未使用。何もしない関数のポインタを設定。 |

### 3.1.4　メッセージ構造体の仕様

RX-NET は，送信関数に送信データを渡す際にメッセージ構造体“ m ”を使用します。またネットワークタスクから受信データを RX-NET に渡す際にもメッセージ構造体“ m ”に受信データを格納して渡す必要があります。メッセージ構造体は nectools32 / inc850 / rxnet / m.h にて定義されています。

ここでは“ 送信 Ether フレームを取り出す方法 ”と“ 受信 Ether フレームを格納する方法 ”について説明します。

1. 送信 Ether フレームをメッセージ構造体“ m ”から取り出す方法

   メッセージ構造体中のデータは m_hp から始まるアドレスに格納されています。ここからデータを取り出すことができますが，その際はメッセージ構造体の Ether フレーム長が必要になります。つまり m_hp から Ether フレーム長分のデータを取得することになります。そのフレーム長を取得するときは，関数“ m_dsize() ”を使用します。

表 3－3　m_dsize()の仕様

| 機　能 | パケット長を取得する |
|---|---|
| 形　式 | nLen = m_dsize(mp) |
| 引　数 | メッセージ構造体“ m ”のポインタ |
| 戻り値 | パケット長 |

m_hp から nLen に格納される Ether フレーム長分を読み出すことで，メッセージ構造体 m からデータを取り出すことができます。

2. 受信 Ether フレームをメッセージ構造体“m”に格納する方法

メッセージを格納するには，まず領域確保が必要となります。領域確保には，関数“m_new()”を使用します。

表 3－4　m_new()の仕様

| 機　能 | 受信データ領域を確保する | |
|---|---|---|
| 形　式 | mp = m_new(packetlen, &err, 0); | |
| 引　数 | packetlen | 受信した Ether フレームの長さを格納した整数 |
| | &err | エラーを格納する領域のアドレス |
| | 0 | 予約（常に 0） |
| 戻り値 | 確保成功時 | メッセージ構造体 m のポインタのアドレスを返し，err に 0 を格納 |
| | 確保失敗時 | NULL を返し，err に非 0 を返す |

m_new()で確保した領域にデータを書き込む前に，次の処理が必要となります。

```
mp -> m_hp -= packetlen;              /* 書き込みポインタを戻す              */
mp -> m_ndp = <device name>_ndp;      /* mp に ndp の情報を設定する          */
```

m_hp はデータの先頭を示すポインタです。領域確保後はデータ領域の最後尾を示しています。確保した領域の分だけポインタを減算する必要があります。

m_ndp は RX-NET がデータをどのデバイスから受信したのか識別するために使用しています。初期化関数の引数として渡された ndp のポインタを指定してください。

上記の処理の後，m_hp のアドレスに受信した Ether フレームをコピーしてください。以上で受信データをメッセージ構造体 m へ格納する作業は完了です。なお，受信処理後，送信完了後の m 構造体の領域開放は RX-NET が行ないます。デバイス・ドライバ内に領域開放の処理を記述する必要はありません。

### 3.1.5　RX-NET の受信キューへのデータの渡し方

デバイス・ドライバが受信したデータを RX-NET に渡す手順を以下に示します。

```
twq_promise();          /* データを渡す前にキューをロック     */
msm(mp, en_up);         /* RX-NET にデータを渡す              */
twq_run();              /* キューのロックを解除               */
```

msm(mp, en_up) の箇所が受信したデータを受信キューに登録している箇所です。

mp は受信データを格納した m 構造体のポインタです。

en_up は RX-NET の Ether フレームの受信関数です。

また受信キューの操作を行なう前にキューのロックが必要です。キューのロックには twq_promise()，ロック解除には twq_run()を使用します。

# 3.2　ドライバの関数の仕様

デバイス・ドライバの関数として，次のような関数を使用しています。

表 3－5　デバイス・ドライバの関数

| 種 別 | 関数名 |
|---|---|
| 初期化関数 | ＜デバイス名＞_init() |
| 開始・終了関数 | ＜デバイス名＞_updown() |
| 送信関数 | ＜デバイス名＞_start() |
| ネットワーク割り込みハンドラ | ＜デバイス名＞_intr() |
| ネットワーク・タスク | ＜デバイス名＞_Thread() |

これらの関数のプロトタイプ，引数仕様，概要，実装例について説明します。
なお，これらの関数のうち“ネットワーク割り込みハンドラ”と“ネットワーク・タスク”はRX850 Proへの登録，削除，参照がドライバ内に閉じているため，関数名は上記のものと同じにする必要はありません。

それ以外の関数は，RX-NET とのインタフェースになるので後述のインタフェース記述部で，同じ名前を使用するようにしてください。

ここで説明しているドライバの実装例では『送信完了時，受信時に割り込みを入れることができる』ことを想定しています。

## 3.2.1　ドライバ内で定義するグローバル変数

ドライバ内で定義しているグローバル変数は次のとおりです。

表 3－6　ドライバ内で定義しているグローバル変数

| 変数名 | 型 | 初期値 | 概 要 |
|---|---|---|---|
| ＜device name＞_ndp | netdev* | NULL | ＜device name＞_init()内で引数の ndp の値をコピーして保存する。<br>送信関数やネットワーク・タスク等の ndp が引数として渡されない関数からデバイスの管理情報を参照するときに利用する。 |

### 3.2.2　スケジューリング制御のインタフェース

RX-NET は，スケジューリング制御（スケジューリングの禁止）のインタフェースとして，次のものを用意しています。

表 3－7　スケジューリング制御のインタフェース

| 種 別 | マクロ名 |
|---|---|
| 関数内で RX-NET タスク間のスケジューリング制御の使用を宣言 | use_critical |
| RX-NET タスク間のスケジューリング禁止区間の開始 | critical |
| RX-NET タスク間のスケジューリング禁止区間の終了 | normal |

マクロ“ use_critical ”は，スケジューリング制御を行う関数内のローカル変数宣言の末尾に記述します。マクロ“ critical ”とマクロ“ normal ”に挟まれた区間がスケジューリング禁止区間になります。スケジューリング制御区間では，割り込みは禁止されていません。割り込み禁止が必要な場合は，別途割り込み禁止処理を記述してください。また関数内でスケジューリング制御区間をネストしてはいけません。

### 3.2.3　初期化関数

**初期化関数のプロトタイプ**

```
int   <device name>_init ( netdev* ndp );
```

**デバイス・ドライバの初期化の準備を行ないます。**so_initialize()**内で呼び出されます。**

- **パラメータ“*ndp*”は，**RX-NET **からデバイス・ドライバの情報が渡されます。**

**この関数の戻り値は以下のようになります。**

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | **正常終了** |
| | 0x0 **以外** | **異常終了** |

**【 実装例 】**

**ここでは，デバイス管理情報ポインタの保存，**Ethernet **コントローラの存在の確認，**MAC **アドレスの取得，ネットワーク・タスクの生成と起動を行います。**

Ethernet **コントローラの割り込み設定等は，次の**＜device name＞_updown()**内で全ての初期化が終わった後に行ないます。**

```
int
<device name>_init(struct netdev* ndp)
{
    u8  etAddress[6];          /* MAC アドレス    */

    /*** このデバイス・ドライバの管理情報のポインタを保存する ***/
    <device name>_ndp = ndp;

    ＜ポート等をチェックして Ethernet コントローラが存在することを確認＞

    if (Ethernet コントローラが見つからなかったら) {
       return ENODEV;
    }

    ＜Ethernet コントローラから MAC アドレスを読み出して etAddress 変数に格納します。
    具体的には 48 ビットの MAC アドレスのポインタを受け取り，MAC アドレスの値を
    プロトコル内で記憶します＞

    brd_host_addr(&etAddress); /* MAC アドレスを RX-NET に通知      */

    ＜ネットタスク・タスクを作成・起動する＞

    if (タスクの作成・起動に失敗したら) {
       return ERR;
    }
    return ENOERR;     /* 正常終了 */
}
```

### 3.2.4　開始・終了関数

**開始・終了関数のプロトタイプ**

```
int   <device name>_updown ( netdev* ndp, u16 flags );
```

**開始するときは「Ethernet コントローラの初期化」，終了するときは「Ethernet コントローラの停止」を行います。**

**この関数は** `ll_config()` **内と** `ll_unconfig()` **内で呼び出されます。ただし** DHCP **を使用しているときは** `so_initialize()` **内で呼び出されます。**

- **第一パラメータ“*ndp*”は，**RX-NET **からデバイス・ドライバの情報が渡されます。**
- **第二パラメータ“*flags*”は，呼び出し元の判定を行ないます。**
  `ll_config()` **からの呼び出しの場合は“** `flag == 1` **”**
  `ll_unconfig()` **からの呼び出しの場合は“** `flag == 0` **”**

**この関数の戻り値は以下のようになります。**

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| | 0x0 以外 | 異常終了 |

【 実装例 】

開始時：Ethernet コントローラの初期化，ネットワーク割り込みハンドラの登録，割り込み許可

終了時：Ethernet コントローラの停止，ネットワーク割り込みハンドラの削除，割り込み禁止

```
int
<device name>_updown(struct netdev* ndp, u16 flags)
{
    if (flag) {

    /*** ndp->nd_lladdr.a_ena （型は a48）に MAC アドレスを格納する ***/
    ndp->nd_lladdr.a_type = AF_ETHER;    /* アドレスタイプに AF_ETHER 型を設定 */
    ndp->nd_lladdr.a_len = sizeof(a48);  /* アドレス長に MAC アドレス長を設定 */

    ＜PSW を操作して CPU を割り込み禁止にする＞
    ＜Ethernet コントローラの初期化＞
      （コントローラ内の割り込み設定等を行なう）
      （サンプルでは Ne2kInit() / S91C_DriverReset()）
    ＜ネットワーク割り込みハンドラの登録（_setvect / def_int）＞
    ＜ネットワーク割り込みが入るように CPU の割り込みマスクを変更＞
    ＜PSW を操作して CPU を割り込み許可にする＞

    } else {

    ＜PSW を操作して CPU を割り込み禁止にする＞
    ＜Ethernet コントローラの動作停止
      （コントローラ内の割り込み設定等を行なう）
      （サンプルでは Ne2kShutdown() / S91C_Shutdown()）
    ＜ネットワーク割り込みハンドラの削除（_setvect / def_int）＞
    ＜CPU の割り込みマスクを変更（必要があれば）＞
    ＜PSW を操作して CPU を割り込み許可にする＞

    }
    return ENOERR;     /* 正常終了 */
}
```

### 3.2.5　送信関数

**送信関数のプロトタイプ**

```
int   <device name>_start ( struct m* mp );
```

RX-NET 内から呼び出される送信関数です。

・ パラメータ“*mp*”は，Ether フレームを含む送信データの構造体のアドレス。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了（常にこの値） |

【 実装例 】

RX-NET から引数として受け取ったデータを，Ethernet コントローラを使って送信します。

引数として渡されるデータには，物理層（Ether）のヘッダも含まれているので，送信関数内でこの部分を追加する必要はありません。

Ethernet コントローラのヘッダがある場合は，この関数内で送信データに付加してください。

送信関数は送信を要求したユーザ・タスク，RX-NET のネットワーク・タスク，RX-NET のタイマ・タスクから呼び出される可能性があります。

Ethernet コントローラの送信コマンドの実行中に，別のタスクから送信関数が呼ばれることのないように関数内ではスケジューリング制御を行なってください。

```
int
<device name>_start(struct m* mp)
{
    int  nLen;          /* パケット長 */
    use_critical;       /*関数内でスケジュール禁止区間を使用する宣言 */

    nLen = m_dsize(mp);         /* mp からパケット長を取り出す */

    /*** <device name>_start()が多重に動かないようにスケジュール禁止区間にする ***/
    critical;

    ＜nLen 長のパケットを Ethernet コントローラで送信（先頭アドレスは mp->m_hp）＞

    /*** <device name>_start()のスケジュール禁止区間の終了 ***/
    normal;

    return ENOERR;
}
```

### 3.2.6　ネットワーク割り込みハンドラ

ネットワーク割り込みハンドラのプロトタイプ

```
ID   <device name>_intr ( void );
```

＜device name＞_updown( )内で登録される割り込みハンドラです。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| TSK_NULL | 0x0 | 正常終了（常にこの値） |

【 実装例 】

割り込み時間を短くするために，割り込みハンドラではネットワーク・タスクの起床要求のみを行います。データの送受信を実際に行なうのはネットワーク・タスクです。

サンプルでは，ネットワーク・タスクが処理を終えるまでの間に次の Ethernet コントローラの割り込みが入らないように Ethernet コントローラの割り込みマスクを操作しています。

```
ID
<device name>_intr(void)
{

    ＜Ethernet コントローラ内の割り込みマスクを操作して割り込みを禁止＞
      （サンプルでは Ne2kDisableInts()/ S91C_DisableInts()）
    ＜ネットワーク・タスク（<device_name>_Thread）を起床させる（wup_tsk/iwup_tsk）＞
    ＜CPU の割り込み終了処理を行う＞

    return TSK_NULL;
}
```

なお，このハンドラ内で割り込み要因を取得して起床させるタスクに通知する方法もよいでしょう。その場合，タスク内で Ethernet コントローラの ISR（割り込みステータスレジスタ）を読み出す処理がいらなくなります。

### 3.2.7　ネットワーク・タスク

**ネットワーク・タスクのプロトタイプ**

```
void   <device name>_Thread (int stacd );
```

**ネットワーク割り込みハンドラからの要求で起床するネットワーク送受信タスクです。**

- **パラメータ“*stacd*”は，タスク起動時に渡されるパラメータです。これは** RX850 Pro / RX4000V4 **の仕様によるものですが，このタスク内でこの値を使用することはありません。詳しくは** RX850 Pro / RX4000V4 **のユーザーズ・マニュアルを参照してください。**

**【 実装例 】**

**受信したパケットは** msm() **を使って** RX-NET **に渡します。このとき，メッセージ構造体** m **を使用します。メッセージ構造体** m **の領域を確保する関数は** m_new() **です。**

**送信完了割り込みを受信した場合は，**ndq_restart() **を呼んで次の送信を行ないます。**

```
void
<device name>_Thread(int stacd)
{
    int  err;     /* m_new関数の引数用 */
    int  length; /* 受信フレーム長   */
    m*   mp;      /* 書き込みポインタ */

    use_critical;       /* 関数内でスケジュール禁止区間を使用する宣言 */
    while(1) {

        slp_tsk();      /* ネットワーク割り込みハンドラから起床されるまで起床待ち */
        critical;       /* 送信・受信処理中はスケジュール禁止区間にする */

        ＜Ethernetコントローラの ISR（割り込みステータスレジスタ）を読み出し
        読み出した割り込み要因をクリア＞

        /*** 割り込み要因で場合分け ***/
        if (割り込み要因が受信割り込みだったとき) {
            length = 受信Ether フレーム長     /* 受信 Ether フレーム長を取得する */
            mp = m_new(length, &err, 0);/* Ether フレーム格納領域の確保 */

            ＜確保できたら受信 Ether フレームを mp->m_hp のアドレスから書き込む
            終わったら RX-NET にデータを渡す（以下の処理）＞

            if (mp != NULL ) {          /* NULL の時はメモリ不足 */
            mp->m_hp -= length;         /* 書き込みポインタを戻す */

            /*** 受信Ether フレームを mp->m_hp のアドレスから書き込む ***/
            mp->m_ndp = <device name>_ndp; /* mp に ndp の情報を設定する */
            twq_promise();              /* データを渡す前にキューロック */
            msm(mp, en_up);             /* RX-NET にデータを渡す */
            twq_run();                  /* キューのロックを解除 */
            }
        }
        if (割り込み要因が送信割り込みだったとき) {
            /*** 次の送信処理を実行する ***/
            ndp_restart(<device_name>_ndp);
        }
        if (割り込み要因がエラー割り込みだったとき) {
            ＜エラー処理を実行＞
            ＜ネットワーク割り込みハンドラで操作した Ethernet コントローラの
            割り込みマスクを元に戻して再び割り込みが入るようにする＞
        }
        normal; /* スケジュール禁止区間を終了する */
    }
}
```

# 第4章　UARTコントローラドライバ

サンプルに付属していない UART コントローラを使用する場合，UART コントローラのドライバを新規に作成する必要があります。この節では UART コントローラのドライバ作成に必要な情報を示します。

## 4. 1　ドライバの新規作成

### 4.1.1　デバイス・ドライバの構成要素

デバイス・ドライバを自作する場合，下記の要素を実装します。

表 4－1　デバイス・ドライバ（UARTコントローラ）の構成要素

| 構成要素 | 機　能 |
|---|---|
| 開始・終了関数 | 接続と切断の度に呼び出されます。<br>PPP 接続時に UART 割り込みハンドラの登録，タスクの生成と起動，送受信バッファの初期化を行います。<br>PPP 切断時に UART 割り込みハンドラの削除を行います。 |
| 送信関数 | PPP デバイスに出力があった時，RX-NET から呼び出されます。またモデム初期化時等にも利用されます。 |
| 受信バッファ読み出し関数 | PPP 接続の開始前と切断後にモデムの通信設定等に使用されます。PPP 接続中には使用されません。 |
| UART 割り込みハンドラ | 受信割り込み，送信完了割り込み，エラー割り込み等を処理します。 |
| UART タスク | 受信／送信完了割り込み後に UART 割り込みハンドラから呼び出されます。<br>受信バッファからデータを取り出して RX-NET（PPP）に渡します。<br>送信バッファに空きがある場合は RX-NET（PPP）にデータを格納するように促します。 |
| 送信バッファ | RX-NET が送信関数からデータを入力します。<br>送信完了割り込み時に UART 割り込みハンドラが UART にデータを送ります。 |
| 受信バッファ | 受信割り込み時に UART 割り込みハンドラが UART の受信データを入力します。<br>受信タスクがデータを取り出して RX-NET に渡します。 |

### 4.1.2　新規ドライバの作成手順

次の手順でデバイス・ドライバを作成してください。

1. デバイス・ドライバのディレクトリを作成

   デバイス・ドライバの作成場所は,

   nectools32 / src / rxnet / drivers

   以下に作成してください。この下にデバイス・ドライバ名を使ったディレクトリ名を作成するとよいでしょう。例えば TL16550C 用のサンプル・ドライバの場合は tl16550c という名前のディレクトリです。

2. 送受信バッファの実装

   UART 割り込みハンドラと UART ドライバ関数間でデータの受け渡しを行うためのバッファを実装します。

3. UART ドライバ関数の実装

   作成したデバイス・ドライバのディレクトリにドライバ関数のソースを作成します。作成する関数の仕様については, “4.2 ドライバの関数の仕様”を参照してください。

4. RX-NET コンフィギュレーション・オブジェクトの作成

   ppp 仮想デバイス“ppp0”を指定したコンフィギュレーション・オブジェクトを作成します。サンプルの nectools32 / src / rxnet / netconf / ppp0.ma.sg の設定から, UART ドライバ部分だけを差し替えてオブジェクトを作成します。

### 4.1.3　デバイス・ドライバ関数の登録

RX-NET から RX-NET（PPP）を使用して PPP 接続を行うための設定を行います。

RX-NET（PPP）には PPP 接続のための RX-NET ドライバ関数のインタフェースがあります。RX-NET に RX-NET（PPP）に含まれるドライバ関数を登録することで, RX-NET（PPP）のドライバ関数を経由して UART ドライバ関数を呼び出すことができるようになります。

デバイス・ドライバ関数の登録は RX-NET コンフィギュレーション・オブジェクト中のデバイス管理テーブルに関数ポインタを設定することで行ないます。

RX-NET のデバイス管理テーブルは“netdev 構造体”の配列“ndevsw[]”で実装しています。netdev 構造体の定義は nectools32 / inc850 / rxnet / netdev.h で行なわれています。そしてデバイス管理テーブルを記述しているのは nectools32 / src / rxnet / netconf / ppp0.ma.s g / src / rxnetconf.h の次の箇所です。

図 4－1　ndevsw[]デバイス定義

```
netdev   ndevsw[] = {
/*      nd_name,     nd_flags,     nd_devid,    0,     0,
 *      nd_p0,       nd_p1,        nd_p2,       nd_p3,
 *      nd_lladdr,   nd_stat,      nd_init,
 *      nd_updown,   nd_send,      nd_start,     nd_ioctl
 */
{       /* intra-machine device; must be in slot 0 */
       "im",         0,          0,            0,     0,
       0,            0,          0,            0,
       {0},          {0},        noent_init,
       noent_updown, im_send,    noent_start,   noent_ioctl
},
{      "ppp0",       0,          0,            0,     0,
       PPP0_P0,      PPP0_P1,     PPP0_P2,      PPP0_P3,
       {AF_PPP},    {0},         PPP_init,
       ppp_updown,  ppp_scomm,   ppp_dstart,   noent_ioctl
}
};
```

上記のように，ndevsw[]配列は“ im ”で始まる要素と“ ppp0 ”で始まる要素の 2 つの要素で構成されています。“ im ”で始まる要素は RX-NET が内部で使用するものです。ここは変更しないで下さい。デバイス・ドライバの設定は 2 番目の“ ppp0 ”で始まる要素で行ないます。

ただし，RX-NET（PPP）で使用する場合は，一度 RX-NET（PPP）のライブラリを経由してから UART ドライバの関数が呼ばれる仕様になっています。デバイス管理テーブルには UART ドライバの関数を直接記述することはありません。どのような UART ドライバを使用する場合も，デバイス管理テーブルは上記のとおりに記述してください。

netdev 構造体の重要メンバの名前は上記サンプルの ndevsw[]配列内のコメントに記述してあります。以後はコメントに書かれているメンバに設定する内容について説明します。なおサンプル内で“ 0 ”が設定されている箇所はそのまま 0 を設定してください。

表 4－2　ndevsw[]のメンバ名とその意味

| メンバ名 | 意 味 |
|---|---|
| nd_name | デバイスの名前を表す文字列を記述します。RX-NET（PPP）を使用する場合は“ ppp0 ”と記述します。 |
| nd_p0 | 初期化パラメータ－0。ドライバ関数の第一引数 ndp 経由でこのメンバにアクセスできます。ndp 経由でドライバ関数に渡すパラメータがない場合は未使用です |
| nd_p1 | 初期化パラメータ－1。ドライバ関数の第一引数 ndp 経由でこのメンバにアクセスできます。ndp 経由でドライバ関数に渡すパラメータがない場合は未使用です |
| nd_p2 | 初期化パラメータ－2。ドライバ関数の第一引数 ndp 経由でこのメンバにアクセスできます。ndp 経由でドライバ関数に渡すパラメータがない場合は未使用です |
| nd_p3 | 初期化パラメータ－3。ドライバ関数の第一引数 ndp 経由でこのメンバにアクセスできます。ndp 経由でドライバ関数に渡すパラメータがない場合は未使用です |
| nd_lladdr | デバイスの物理層のアドレス・タイプを指定します。RX-NET（PPP）を使用する場合は，｛AF_PPP｝を設定してください。 |
| nd_init | 初期化関数のポインタです。RX-NET（PPP）を使用する場合は ppp_init と記述します。 |
| nd_updown | 開始・終了関数のポインタです。RX-NET（PPP）を使用する場合は ppp_updown と記述します。 |
| nd_send | IP パケットに，物理層のヘッダを付加する関数を指定します。RX-NET（PPP）を使用する場合は ppp_scomm と記述します。 |
| nd_start | 送信関数のポインタです。RX-NET（PPP）を使用する場合は ppp_dstart と記述します。 |
| nd_ioctl | 現在は未使用です。RX-NET（PPP）を使用する場合は noent_ioctl と記述します。 |

なお，ここで指定する“ ppp_init ”“ ppp_updown ”“ ppp_dstart ”“ ppp_scomm ”は RX-NET（PPP）ライブラリに，“ noent_ioctl ”は RX-NET のライブラリに含まれています。ユーザで定義する必要はありません。

### 4.1.4 UART送受信バッファの仕様

RX-NET（PPP）では，送受信バッファは実装依存です。

1. RX-NET（PPP）とUART送受信バッファ間のアクセス方法

RX-NET（PPP）とUART送受信バッファのアクセス方法は次の通りです。

| | |
|---|---|
| ドライバの受信バッファ → RX-NET（PPP） | 1バイト単位で書き込みを行う |
| RX-NET（PPP）→ ドライバの送信バッファ | 1パケット単位で書き込みを行う |

上記より，受信バッファには特にサイズの制限はありません。しかし送信バッファは 1 パケット単位の書き込みのため，パケットの最大サイズより大きくなくてはなりません。

2. パケットの最大サイズの計算方法

送信バッファのサイズ決定の目安として PPP パケットサイズの計算方法を示します。

PPP パケットのフォーマットは次の通りです。

| | | | | |
|---|---|---|---|---|
| フラグ（1バイト） | PPPヘッダ（4バイト） | PPPデータ（最大でMTUバイト） | FCS（2バイト） | フラグ（1バイト） |

MTU（Maximum Transmission Unit）は通常 1500 バイトが使用されます。MTU が 1500 の場合，上記のパケットの最大値は 1508 バイトになります。ただし PPP では 1 バイトのデータを 2 バイトに符号化することがあるので，1508 バイトのパケットは最大で 2 倍の 3016 バイトになる可能性があります。すべてのバイトが 2 バイトに符号化することはありえないので，実際にパケットサイズが 2 倍になることはありませんが，MTU が 1500 の場合はバッファサイズを 3016 バイト以上確保しておくことを推奨します。

3. バッファ操作時のスケジューリング制御

後述のサンプル UART ドライバ関数の実装では，バッファを次のように利用しています。

| ハンドラ／関数／タスク | データの移動 | | |
|---|---|---|---|
| 割り込みハンドラ内 | 送受信バッファ | ←→ | UART |
| 送信関数 | RX-NET | → | 送受信バッファ |
| UARTタスク | RX-NET | ← | 送受信バッファ |

送受信バッファが上記のような仕様を満たすためには，送受信バッファを利用する“UART 割り込みハンドラ”“UART タスク”“送信関数”の間で，送受信バッファ操作のための同期を取り，誤動作を防ぐ仕組みを実装する必要があります。

また，バッファから RX-NET（PPP）にデータを渡す途中で，RX-NET（PPP）の状態が“接続 → 切断”と変化することも避けなければなりません。そのため，スケジューリング制御の仕組みは RX-NET（PPP）で使用しているスケジューリング制御と同じものにする必要があります。

ここでは UART ドライバ関数の実装例で使用している，送受信バッファ操作の同期の方法を説明します。

### (1) スケジューリング制御のインタフェース

RX-NET は，スケジューリング制御のインタフェースとして，次のものを用意しています。

表 4－3　スケジューリング制御のインタフェース

| 種 別 | マクロ名 |
|---|---|
| 関数内で RX-NET タスク間排他処理の使用を宣言 | use_critical |
| RX-NET タスク間排他区間の開始 | critical |
| RX-NET タスク間排他区間の終了 | normal |

マクロ“ use_critical ”は，スケジューリング制御を行う関数内のローカル変数宣言の末尾に記述します。マクロ“ critical ”とマクロ“ normal ”に挟まれた区間がスケジューリング制御区間になります。スケジューリング制御区間では，割り込みは禁止されていません。割り込み禁止が必要な場合は，別途割り込み禁止処理を記述してください。また関数内でスケジューリング制御区間をネストしてはいけません。

ドライバ内でスケジューリング制御を使用しているところは次の箇所です。

- “送信関数実行中”。次の送信関数の実行が始まらないようにするため
- “受信関数が受信データを渡している最中”。PPP の上位層が終了してしまわないようにするため。

### (2) 待ち合わせのインタフェース

UART タスクは，UART 割り込みハンドラ，または UART 開始・終了関数からイベントがあるまで待機します。この待ち合わせには，RX850 Pro / RX4000V4 の slp_tsk システム・コールと wup_tsk システム・コールを使用しています。

ドライバ内で待ち合わせを使用しているのは次の箇所です。

- 受信バッファにデータが格納されたとき（UART 割り込みハンドラから通知）
- PPP の上位層が終了したとき（開始・終了関数から通知）

4.　バッファの空き通知

受信バッファに空きがない時は，ドライバ内の UART コントロールのレベルでフロー制御を行うことにより，相手からの送信を抑制，再開することができます。

しかし，送信バッファに空きがないとき，RX-NET（PPP）からデータをドライバに送らないようにするためには，これとは別の仕組みが必要です。

そこで RX-NET では，送信パケットを一度 RX-NET（PPP）内の送信キューに蓄積し，ドライバからバッファの空き通知関数“ ndq_restart() ”が呼ばれるまで，送信関数“ serial_write() ”が実行されるのを待つ仕組みになっています。

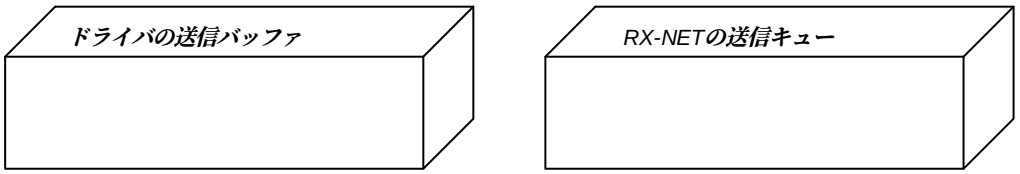
ドライバの送信バッファ，RX-NET の送信キュー，バッファ空き通知関数“ ndq_restart() ”，送信関数“ serial_write() ”の関係を説明します。

・ 前提条件

ドライバの送信バッファのサイズは，最大パケットサイズよりも大きいこと（＝バッファが空なら１パケット受け入れ可能であること）。つまり初期状態では，RX-NET（PPP）は無条件に送信関数“ serial_write() ”を呼び出して送信可能です。

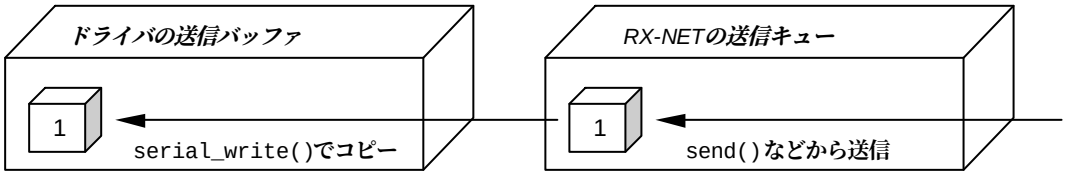
・ 初期状態

初期状態では RX-NET（PPP）の送信キューが空で，ドライバの送信バッファに１パケット分以上の空きがあります。




・ １パケット送信状態

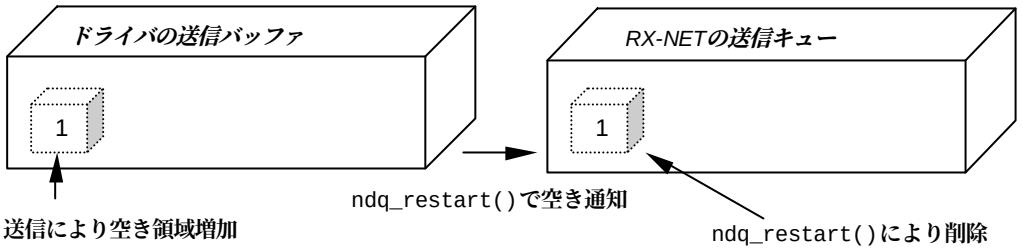
RX-NET（PPP）は初期状態からパケットを送信する場合，RX-NET（PPP）の送信キューにパケットを格納し，serial_write()でドライバの送信バッファにコピーします。




RX-NET（PPP）の送信キューに格納されたパケットは，ドライバから「送信キューの空き通知」されるまで保持されます。

・ **1パケット送信状態から送信キュー空き通知**

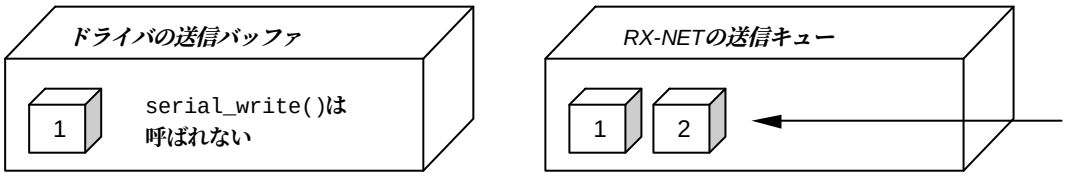
**データ送信を行うとドライバの送信バッファの空きサイズが 1 パケットより大きくなったとき，ドライバは** ndq_restart() **を呼び出して“送信バッファの空き通知”を行います。**




ndq_restart() **は，送信キューにパケットがある場合，先頭のパケットを削除します。先頭パケット削除後にキューが空になった場合，** ndq_restart() **は処理を終了します。これにより，キューとバッファは初期状態へと遷移します。**

・ **2パケット送信状態**

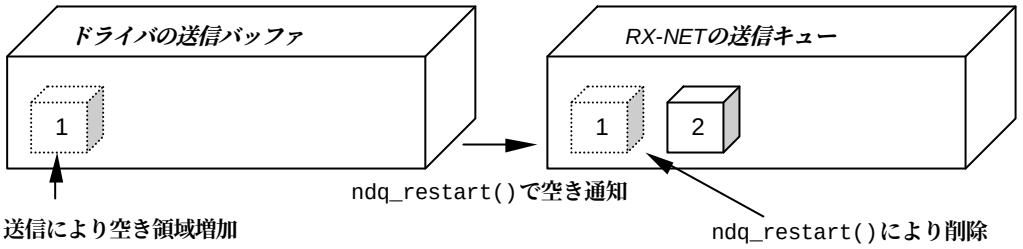
**1 パケット送信状態からさらにパケットを送信する場合，**RX-NET（PPP）**の送信キューの末尾にデータを格納します。**RX-NET（PPP）**の送信キューが空ではないことで，初期状態と違うことを判定しています。**



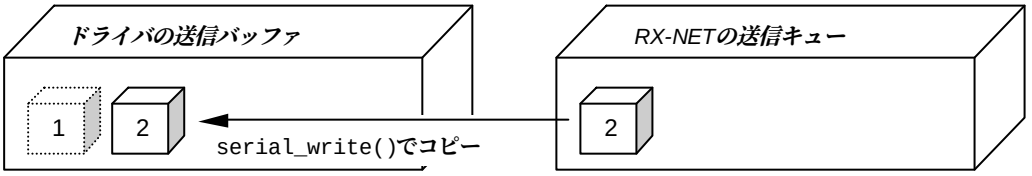

**なお 3 パケット目，4 パケット目の送信がある場合も同一です。**RX-NET（PPP）**の送信キューにパケットが増えていくだけで，2 パケット送信状態から遷移しません。**

・ **2パケット送信状態から送信キュー空き通知**

**データ送信を行うと，ドライバの送信バッファの空きサイズが1パケットより大きくなったとき，ドライバは** ndq_restart()**を呼び出して“送信バッファの空き通知”を行います。ドライバからは** RX-NET（PPP）**の送信キューの状態はわかりませんので，ドライバ側で行う処理は1パケット送信状態からの“送信キュー空き通知”と全く同じです。**



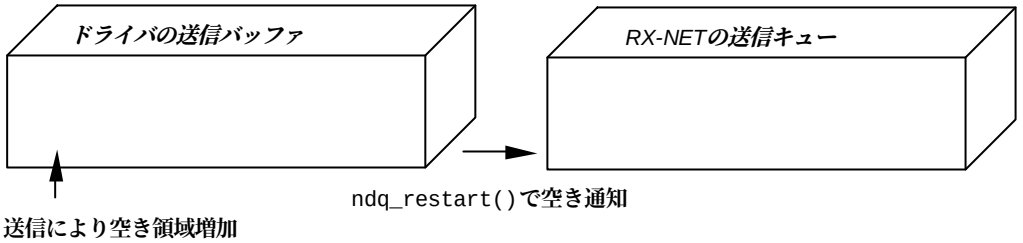

ndq_restart()**は，送信キューにパケットがある場合，先頭のパケットを削除します。先頭パケット削除後にキューにパケットがある場合，**serial_write()**で送信バッファにパケットをコピーします。**




**これにより，キューとバッファは** 1 **パケット送信状態へと遷移します。キューに** 3 **パケット目，**4 **パケット目がある場合も，**RX-NET **の送信キューから先頭パケットを取り除き，**2 **番目のパケットを** serial_write()**でコピーします。キューのパケットが** 2 **つ以上残っているときは，**2 **パケット送信状態からの“送信キューの空き通知”状態から遷移しません。**

・ **初期状態からの送信キュー空き通知**

RX-NET（PPP）の送信キューが空のときに ndq_restart()が呼ばれても誤動作はしません。




RX-NET（PPP）の送信キューが空の場合，ndq_restart()は何もせずに終了します。

送信処理は上記の仕組みになっているので，ドライバ内で serial_write()が呼び出された回数だけ ndq_restart()を呼び出すようにすると，次々とパケットを送信することができます。（送信バッファが空の場合，割り込みが入るたびに無条件に ndq_restart()を呼び出すことは，オーバヘッドが生じるので推奨しません）。

またPPP 接続が切断されたときに，送信キューに残っていたデータは自動的に削除されるので，ドライバ終了処理に送信キューを空にするための ndq_restart()呼び出し記述をする必要はありません。

例外として，一部の送信処理（DIAL，LOGIN 文字列送信等）は，送信キューを利用せずに直接 serial_write()を呼び出します。したがって serial_write()の呼び出し回数分 ndq_restart()を実行するようにドライバを作成すると，キューが空のときに ndq_restart()が呼ばれますが，PPP の初期化時に数回呼び出される程度なので，通信速度に影響することはありません。

## 4.2　ドライバの関数の仕様

UART ドライバの関数として，次のような関数を使用しています。

表 4－4　デバイス・ドライバの関数

| 種 別 | 関数名 |
|---|---|
| 開始・終了関数 | serial_updown() |
| 送信関数 | serial_write() |
| 受信関数 | serial_read_buf() |
| UART 割り込みハンドラ | serial_intr() |
| 受信タスク | serial_task() |

なお“ UART 割り込みハンドラ ”と“ 受信タスク ”は RX850 Pro（RX4000V4）への登録，削除，参照がドライバ内で閉じているため，関数名は上記のものと同じにする必要はありません。それ以外の関数は，RX-NET（PPP）とのインタフェースになるため，関数名を変更しないで下さい。

ここで説明しているドライバ実装例で，想定しているモデム設定は以下のとおりです。

- DCD 信号は相手モデムのキャリアに従う
- DTR 信号 ON で回線切断
- CTS / RTS フロー制御を使用する
- UART の FIFO の空き数によるフロー制御は UART コントローラが自動で行うので，ドライバでは操作しない

### 4.2.1　ドライバ内で定義しているグローバル変数

ドライバ内で定義しているグローバル変数のうち，主要なものは次のとおりです。

表 4−5　UARTドライバ内で定義しているグローバル変数（主要なもの）

| 変数名 | 型 | 初期値 | 概 要 |
|---|---|---|---|
| serial_isup | int | 0 | UARTのupdown状態のフラグ<br>serial_updown()でup時に1，down時に0を設定します。<br>UARTタスクでこのフラグをチェックして終了処理を開始します。 |
| send_npkts | int | 0 | serial_write()を実行するときに加算します。値が1以上なら送信バッファに空きができたときに ndq_restart()を呼び出す必要があります。ndq_restart()を呼び出したときに減算します。 |
| com_task_id | ID | なし | UARTタスクのIDを格納します。<br>UART割り込みハンドラやppp_updown()からwup_tskシステム・コールでUARTタスクを起床させるために使用します。 |
| com_intr_no | int | なし | UART割り込みハンドラの割り込み番号を格納します。<br>def_intシステム・コールでUART割り込みハンドラを登録するために使用します。 |
| uart_prev_mstat | u8 | なし | 前回のモデムステータスレジスタの内容を格納しています。<br>UARTドライバ内でDCD信号の変化を検出するために使用します。 |
| uart_last_mstat | u8 | なし | モデムステータスの検査時に作業領域として使用します。 |

### 4.2.2　スケジューリング制御のインタフェース

RX-NETは，スケジューリング制御のインタフェースとして，次のものを用意しています。

表 4−6　スケジューリング制御のインタフェース

| 種 別 | マクロ名 |
|---|---|
| 関数内でRX-NETタスク間排他処理の使用を宣言 | use_critical |
| RX-NETタスク間排他区間の開始 | critical |
| RX-NETタスク間排他区間の終了 | normal |

マクロ“use_critical”は，スケジューリング制御を行う関数内のローカル変数宣言の末尾に記述します。マクロ“critical”とマクロ“normal”に挟まれた区間がスケジューリング制御区間になります。スケジューリング制御区間では，割り込みは禁止されていません。割り込み禁止が必要な場合は，別途割り込み禁止処理を記述してください。また関数内でスケジューリング制御区間をネストしてはいけません。

### 4.2.3　ドライバ内で参照する RX-NET（PPP）のグローバル変数

ドライバ関数内で参照する必要のある RX-NET（PPP）のグローバル変数は次のとおりです。

表 4－7　UARTドライバ関数内で参照する必要のあるRX-NET（PPP）のグローバル変数

| 変数名 | 概 要 |
|---|---|
| PPP_IS_UP | PPP の状態判定に使用します。<br>ドライバから RX-NET（PPP）の関数を呼び出す前に，PPP の状態を調べるために使用します。<br>・ PPP_IS_UP が 1 のとき<br>PPP 層はデータを送受信する準備ができているので，ドライバから RX-NET（PPP）の関数の呼び出しを行うことが可能です。<br>・ PPP_IS_UP が 0 のとき<br>PPP 層は未初期化か，切断によって終了しているので，ドライバから RX-NET（PPP）の関数の呼び出しは行えません。<br>PPP_IS_UP の参照とその後の RX-NET（PPP）関数の呼び出しは必ずスケジューリング制御区間内で行ってください。 |
| RXNET_MODEM_LINE | モデム接続かヌルモデム接続かを判定するために使用します。<br>ppp_connect()で指定したモデム／ヌルモデム接続の種別が格納されています。<br>モデム接続が選択された場合は“1”，ヌルモデム接続が選択された場合は“0”，接続前の状態なら“-1”が入っています。<br>ドライバ内でモデム信号線を制御する／しないの判定に使用してください。 |
| slppp_head | RX-NET（PPP）にデータを渡すために使用します。<br>UARTから取得したデータを RX-NET（PPP）に渡す際に ppp_onechar_input()の引数として使用します。<br>UART の送信バッファに十分な空きができたことを通知する際に ndq_restart()の引数として使用します。 |
| ppp_serial_raw | DIAL，LOGIN 文字列の受信制御判定に使用します。<br>接続方法の選択によっては，接続開始時に PPP のパケット以外のデータ（例：モデム初期化コマンド，ダイアルコマンド，LOGIN 文字列処理のプロンプト等）を送受信する必要があります。<br>ドライバの UART タスクで受信したデータは PPP に渡されてしまうので，PPP のパケット以外のデータの送受信が終わるまでドライバの UART タスクを待たせる必要があります。<br>ppp_serial_rawはUARTタスク生成時には非0の値になっており，DIAL やLOGIN 文字列の処理が終了すると 0 に変化します。<br>ドライバの UART タスクの受信ループに入る前に ppp_serial_raw を参照して PPP パケット以外のデータの送受信が終わっているか確認してください。 |

| 変数名 | 概 要 |
|---|---|
| RXNET_DCD_SEM | DCD 信号の変化（回線状態の変化）を通知するために使用します。<br>一般にアナログモデムによる接続では，接続／切断の状態をキャリアの有無で判断し，キャリアは UART の CD 信号で確認することができます。<br>キャリアの発生するタイミングは，ダイアル後にモデム間の通信設定が終わって，データを流せる状態になったときです。接続方法としてモデム接続を選択した場合，RX-NET（PPP）はダイアル後（ATDT コマンド発行後）に回線接続通知のイベント（キャリア検出のイベント）を待ちます。<br>RXNET_DCD_SEM はドライバから回線接続通知のイベントを起こす際に回線状態を RX-NET（PPP）に通知するために使用します。<br>回線接続通知イベントの発行の方法と RXNET_DCD_SEM の値の設定箇所について説明します。<br><br>・ **接続前**<br>`RXNET_DCD_SEM = 0;`<br>接続前は値を 0 に設定してください。<br><br>・ **回線接続通知イベント発行時（DCD 信号 ON を検出）**<br>`RXNET_DCD_SEM = 1;`<br>`os_wakeup(&RXNET_DCD_SEM);`<br>値を 1 に書き換えた後 os_wakeup()を実行して RX-NET（PPP）に DCD 信号の変化を通知してください。<br><br>・ **ドライバ関数内で DCD 信号が OFF になったことを検出した場合**<br>`RXNET_DCD_SEM = 0;`<br>DCD 信号が OFF になった場合は値を 0 に書き換えてください。RX-NET（PPP）で DCD 信号が OFF になるイベントを待っているわけではないので，os_wakeup()の呼び出しは行いません。この値の変更の目的は，次回の接続前の値クリアです。<br>なお，DCD 信号 OFF を検出した場合，回線切断を検出したことになるので，別途 UART タスク終了処理等を行ってください。 |

### 4.2.4　ドライバ内で使用する RX-NET（PPP）の関数

ドライバ内で使用する RX-NET（PPP）の関数について説明します。

表 4－8　ドライバで使用するRX-NET（PPP）の関数

| 関数名 | 概 要 |
|---|---|
| ppp_onechar_input() | 受信データがある時，RX-NET（PPP）に 1 バイトずつ渡します。<br><br>【プロトタイプ】<br>void　ppp_onechar_input (char ***ch***, netdev* ***ndp***);<br><br>UART ドライバが受信したデータを PPP に渡すための関数です。<br>引数 ***ch*** に指定する値は，ドライバで受信した 1 バイトのデータです。<br>引数 ***ndp*** に指定する値は，RX-NET（PPP）内のグローバル変数 slppp_head です。なお，ppp_onechar_input()の呼び出しは，**必ずスケジューリング制御区間内で行ってください**。 |
| ndq_restart() | 送信バッファに空きがある時，RX-NET（PPP）にデータの格納を促します。<br><br>【プロトタイプ】<br>void　ndq_restart (netdev* ***ndp***);<br><br>ndq_restart()は RX-NET（PPP）に次のパケットを送信するように促す関数です。ndq_restart()は，RX-NET（PPP）が最後に送信したパケットを RX-NET 内の送信キューから取り除き，次に送信すべきパケットがある場合は serial_write()を呼び出してパケット送信バッファに格納します。<br>引数 ***ndp*** に指定する値は，RX-NET（PPP）内のグローバル変数 slppp_head です。なお，ndq_restart()の呼び出しは，**必ずスケジューリング制御区間内で行ってください**。 |
| ppp_reset() | UART レベルで切断が終了したときに RX-NET（PPP）に切断を通知します。<br><br>【プロトタイプ】<br>void　ppp_reset (netdev* ***ndp***);<br><br>UART ドライバレベルで切断を検出したとき，PPP に回線の切断による終了を行わせるための関数です。PPP_IS_UP を参照して値が 0 の場合は，PPP の終了処理はすでに行われているので ppp_reset()をドライバから呼び出す必要はありません。<br>引数 ***ndp*** に指定する値は，RX-NET（PPP）内のグローバル変数 slppp_head です。なお，ppp_onechar_input()の呼び出しは，**必ずスケジューリング制御区間内で行ってください**。 |

### 4.2.5　開始・終了関数

**開始・終了関数のプロトタイプ**

```
int   serial_updown ( netdev* ndp, u16 flags, char* opt );
```

ppp_connect()と ppp_disconnect()内で呼び出されます。開始時は UART 初期化の後半部分，終了時は UART の停止を行ないます。

- 第一パラメータ“ ***ndp*** ”は，RX-NET（PPP）から PPP 仮想デバイスの情報が渡されます。ただし関数内で特に使用する必要はありません。
- 第二パラメータ“ ***flags*** ”は，呼び出し元の判定を行ないます。
  ppp_connect()からの呼び出しの場合は“ flag == 1”
  ppp_disconnect()からの呼び出しの場合は“ flag == 0”

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| | 0x0 以外 | 異常終了 |

【 実装例 】

開始時：DCD 信号線フラグの初期化，バッファの初期化，UART 割り込みハンドラの登録，UART タスクの生成と起動，UART 割り込みの許可

終了時：UART 割り込みの禁止，UART 割り込みハンドラの削除，電話回線の切断

```
int
serial_updown(netdev* ndp, u16 flags)
{
    /*** UART タスク作成状態フラグを宣言（初期値は未生成状態）***/
    static int rtsk_cre = 0; /* UART タスク作成状態フラグの宣言（初期値は未生成）*/
    if (flags) {
        /*** ppp_connect の場合の処理 ***/

        ＜送信バッファ，受信バッファを初期化する＞

        serial_initproc();      /* ドライバ状態変数などの初期化 */

        ＜UART コントローラを初期化する＞
        ＜必要があれば UART 関連レジスタを初期化＞

        /*** UARTの状態フラグ（serial_isup）をup にする ***/
        serial_isup = 1;        /* UARTの状態フラグを up に */

        /*** UARTのタスクが生成されていない場合は生成する（cre_tsk）***/
        if (rtsk_cre == 0) {
            ＜UART タスクを作成する（cre_tsk）＞
            rtsk_cre = 1;       /* UART タスク作成状態フラグを生成済み状態にする */
        }
        ＜UART タスクを起動する（sta_tsk）＞
        ＜UART 割り込みハンドラを登録（def_int）＞

    } else {
        /*** ppp_disconnect の場合の処理 ***/

        ＜割り込みハンドラを削除（def_int）＞

        /***UART状態フラグ（serial_isup）を down にし、UART タスクに終了処理を行わせる***/
        serial_isup = 0;        /* UARTの状態フラグを down に  */

        /*** モデム接続の場合は切断処理をする ***/
        if (RXNET_MODEM_LINE == 1) {
            ＜モデムの DTR 信号を OFF にして回線切断する＞
        }
        ＜必要があれば，UART 関連レジスタをリセット＞

        /*** UARTタスクを起床し（そのタスクで）終了処理を行う ***/
        wup_tsk(com_tsk_id);    /* UARTタスクを起こして終了処理を行う */
    }
    return 0;
}
```

### 4.2.6　送信関数

**送信関数のプロトタイプ**

```
    void   serial_write ( char* buf, int bytes, netdev* ndp );
```

RX-NET（PPP）内から呼び出される PPP の送信関数です。

- 第一パラメータ“*buf*”は，送信データの先頭バイトのポインタ。
- 第ニパラメータ“*bytes*”は，送信データの長さ。
- 第三パラメータ“*ndp*”は，RX-NET（PPP）から PPP 仮想デバイスの情報が渡されます。ただし関数内で特に使用する必要はありません。

**【 実装例 】**

RX-NET（PPP）から引数として受け取ったデータを送信バッファに書き込みます。この関数では UART への出力は行いません。UART への出力は UART 割り込みハンドラで行います。この関数でバッファ操作中に UART 割り込みハンドラがバッファの状態を変化させないように，関数内では割り込みを禁止してください。

```
void
serial_write(char* buf, int bytes, netdev* ndp)
{
    use_critical;       /* 関数内でスケジュール禁止区間を使用する宣言 */
    critical;           /* 送信中はスケジュール禁止区間にする */
    DISINTR;            /* バッファ操作前に割り込みを禁止する */

    /*** データの長さが 0 か，UART タスク終了フラグが立っているときは終了 ***/
    if (bytes == 0 || serial_isup == 0) {
        RESINTR;        /* リターン前に割り込みを許可しておく */
        normal;         /* スケジュール許可にする */
        return;         /* エラー終了 */
    }
    /*** ndq_restart()のために「送信中のデータあり」状態にする（send_npkts）***/
    send_npkts++;       /* 「送信中のデータあり」状態にする */

    ＜ここで送信バッファにデータを書き込む＞

    RESINTR;    /* バッファ操作完了で割り込みを許可する */
    normal;     /* 書き込み終了後，スケジュール禁止区間を終了する */
    return;     /* 正常終了 */
}
```

### 4.2.7　受信関数

受信関数のプロトタイプ

```
int   serial_read_buf ( char* buf, int buflen );
```

モデム設定時等に使用される UART 受信関数です。

- 第一パラメータ“*buf*”は，受信データを格納する領域の先頭ポインタ。
- 第ニパラメータ“*buflen*”は，*buf* の差す領域のサイズ（バイト数）。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| | 0 または非 0 | 受信バイト数 |

【 実装例 】

受信バッファ内のデータを取り出して *buf* から始まるアドレスに書き込みます。この関数では UART からの読み出しは行いません。UART からデータを読み出して受信バッファに格納する処理は，UART 割り込みハンドラで行います。この関数でバッファ操作中に UART 割り込みハンドラがバッファの状態を変化させないように，関数内では割り込みを禁止にしてください。

```
int
serial_read_buf(char* buf, int buflen)
{
    use_critical;       /* 関数内でスケジュール禁止区間を使用する宣言 */
    int  len = 0;       /* 受信バイト数のカウント */
    critical;           /* 送信中はスケジュール禁止区間にする */

    ＜バッファ操作前に割り込みを禁止する（DISINTR）＞

    /*** UART タスク終了フラグ（serial_isup）が立っているときは終了 ***/
    if (serial_isup == 0) {
       RESINTR;         /* リターン前に割り込みを許可しておく */
       normal;          /* スケジュール許可にする */
       return -1;       /* エラー終了 */
    }

    /*** 受信バッファからデータ取り出し ***/
    while (受信バッファにデータがある && len < buflen) {
       *(buf++) =  *(受信バッファ++);     /* 受信バッファから１文字取り出す */
       len++;                             /* 受信した文字数カウントを１つあげる */
       ＜バッファの最後に達したら，バッファの先頭に戻る＞
    }

    /*** 受信バッファあふれで受信が停止中だが，受信バッファに空きができた場合 ***/
    if (受信バッファあふれで受信停止中 && 受信バッファの空きが十分) {
       ＜受信バッファから１文字取り出す＞
    }

    RESINTR;            /* バッファ操作完了で割り込みを許可する */
    normal;             /* 読み込み終了後，スケジュール禁止区間を終了する */
    return len;         /* 正常終了（読み出した長さを返す） */
}
```

### 4.2.8　UART 割り込みハンドラ

UART 割り込みハンドラのプロトタイプ

```
ID   serial_intr ( void );
```

serial_updown()内で登録される割り込みハンドラです。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| TSK_NULL | 0x0 | 正常終了（常にこの値） |

【 実装例 】

割り込み時間を短くするために，割り込みハンドラでは送受信バッファの操作のみを行います。RX-NET（PPP）と直接データを交換するのは，送信関数と受信タスクです。

```
ID
serial_intr(void)
{
    ＜UART 割り込み禁止（多重割り込みでバッファを破壊しないように）＞

    /*** ペンディング中の割り込みをすべて処理するまでループ ***/
    while(1) {
       /*** ライン・ステータス・レジスタの値を読み,                          ***/
       /*** データレディ（DR），オーバーランエラー（OE），パリティエラー（PE），  ***/
       /*** フレームエラー（FE），ブレークインタラプト（BI），               ***/
       /*** 送信バッファエンプティ（THRE）フラグのいずれかが立っている間ループ   ***/

       while (ライン・ステータス・レジスタの値が上記のいずれかだった場合) {
           if (データレディのとき) {
                ＜受信データ取得＞
           }
           if (オーバーランのとき) {
                ＜オーバランエラーを記録＞
           }
           if (パリティエラー，フレームエラー，ブレークインタラプトのとき) {
                ＜データを破棄（while ループ抜け）＞
           }
           if (受信バッファの空きが十分ではないとき) {
                ＜受信割り込みを禁止にする＞
           }
           if (受信バッファの空きがあるかをもう一度念のためチェック) {
                ＜受信データをバッファに格納＞
           } else {
                ＜受信バッファオーバフローのため抜ける＞
           }
       }
       ＜モデム・ステータス・レジスタを読み込む＞
       if (モデム・ステータス・レジスタで DCD 信号の変化があったとき) {
           if (モデム制御信号 DCD が 0→1 に変化（接続したとき）) {
               RXNET_DCD_SEM = 1;       /* DCD の状態フラグを 1 にする */
               os_wakeup((char *) & RXNET_DCD_SEM);       /* DIAL 待ちを解除 */
           } else {     /*** DCD 信号が 1 → 0 へ変化した場合 ***/
               RXNET_DCD_SEM = 0;       /* DCD フラグを 0 にする */
                serial_isup = 0;        /* UART 状態フラグを down */
           }
       }
       ＜ライン・ステータス・レジスタを読み込む＞
       if (ライン・ステータス・レジスタで送信バッファに空きがあるとき) {
           if (送信データがある場合) {
                ＜送信バッファからデータを取り出し，UART にデータを渡す＞
           } else {     /*** 送信バッファが空だった場合 ***/
                ＜送信完了割り込みを禁止にする＞
           }
       }
       ＜割り込み確認レジスタを読み込む＞
       if (エラー割り込み要求あり) (
           break;       /* while ループを抜ける */
       }
    }
    ＜UART 割り込みを許可する＞
    return com_tsk_id; /* UART タスクを起床する */
}
```

### 4.2.9 UART タスク

**受信タスクのプロトタイプ**

```
void   serial_task (int stacd );
```

UART 割り込みハンドラからの要求で起床する UART タスクです。

- パラメータ“*stacd*”は，タスク起動時に渡されるパラメータです。これは RX850 Pro の仕様によるものですが，このタスク内でこの値を使用することはありません。詳しくは RX850 Pro のユーザーズ・マニュアルを参照してください。

【 実装例 】

起動待ちループ，送受信ループ，終了処理の 3 つのブロックに分かれています。

UART タスクが送受信ループを終了するのは ppp_disconnect()が呼ばれた場合と，DCD の 0 をドライバで検出した場合の 2 通りの状況があります。DCD が 0 の場合は，PPP を停止するため ppp_reset()と serial_updown()を呼び出します。ppp_disconnect()から終了処理が呼ばれた場合は，PPP_IS_UP が 0 になっています。

```
void
serial_task(int stacd)
{
    unsigned char  ch; /* 受信バッファから取り出した文字を格納する作業変数 */
    use_critical;      /* タスク内でスケジュール禁止区間を使用する宣言 */
    critical;          /* スケジュール禁止区間にする */

    /*** ↓ PPP の起動，DIAL，LOGIN の終了を待つループ。serial_isup が 0 でも中止 ***/
    while ((psp == 0 || ppp_serial_raw) && serial_isup) {
       normal;         /* スケジュール許可 */
       tslp_tsk(1000); /* 定期的にポーリング */
       critical;       /* 起床後，スケジュール禁止区間に変更 */
    }
    normal;    /* whileループを抜けたら，スケジュール許可 */

    /*** UART 状態フラグが down（serial_isup を 0）になるまで while ループで実行 ***/
    while (serial_isup != 0) {
       slp_tsk();      /* serial_intr()から起床されるまで待機 */
       critical;       /* スケジュール禁止区間にする */

       /*** PPP が送受信可能状態のとき（PPP_IS_UP）***/
       if (PPP_IS_UP) {
          DISINTR;     /* バッファ操作中は割り込み禁止 */
          while (受信バッファにデータがある) {
             ch = 受信バッファからデータ取り出し;
             RESINTR; /* 割り込み許可 */
             ppp_onechar_input(c, sleep_head); /* PPP にデータを渡す */
             DISINTR; /* 割り込み禁止 */
          }
          if (受信バッファあふれで受信停止中のとき) {
             if (受信バッファに半分の空きがあるとき) {
                ＜受信割り込みの処理を再開＞
             }
          }
          if (snd_npkts > 0 && 送信バッファに半分の空きがある) {
             snd_npkts--;
             RESINTR; /* 割り込み許可 */
             ndq_restart(slppp_head); /* 次の送信処理 */
          } else {
             RESINTR;
          }
       }
       normal; /* スケジュール許可 */
    }

    /*** 終了処理 ***/
    critical;  /* スケジュール禁止区間にする */
    if (PPP_IS_UP) {
       ppp_reset(slppp_head);          /* RX-NET(PPP)に切断を通知 */
       serial_updown(slppp_head, 0);   /* PPP 処理の終了 */
    }
    normal;    /* スケジュール許可 */

    ext_tsk(); /* タスク終了 */
}
```

# 第5章　ハードウエア・アクセス・ライブラリ

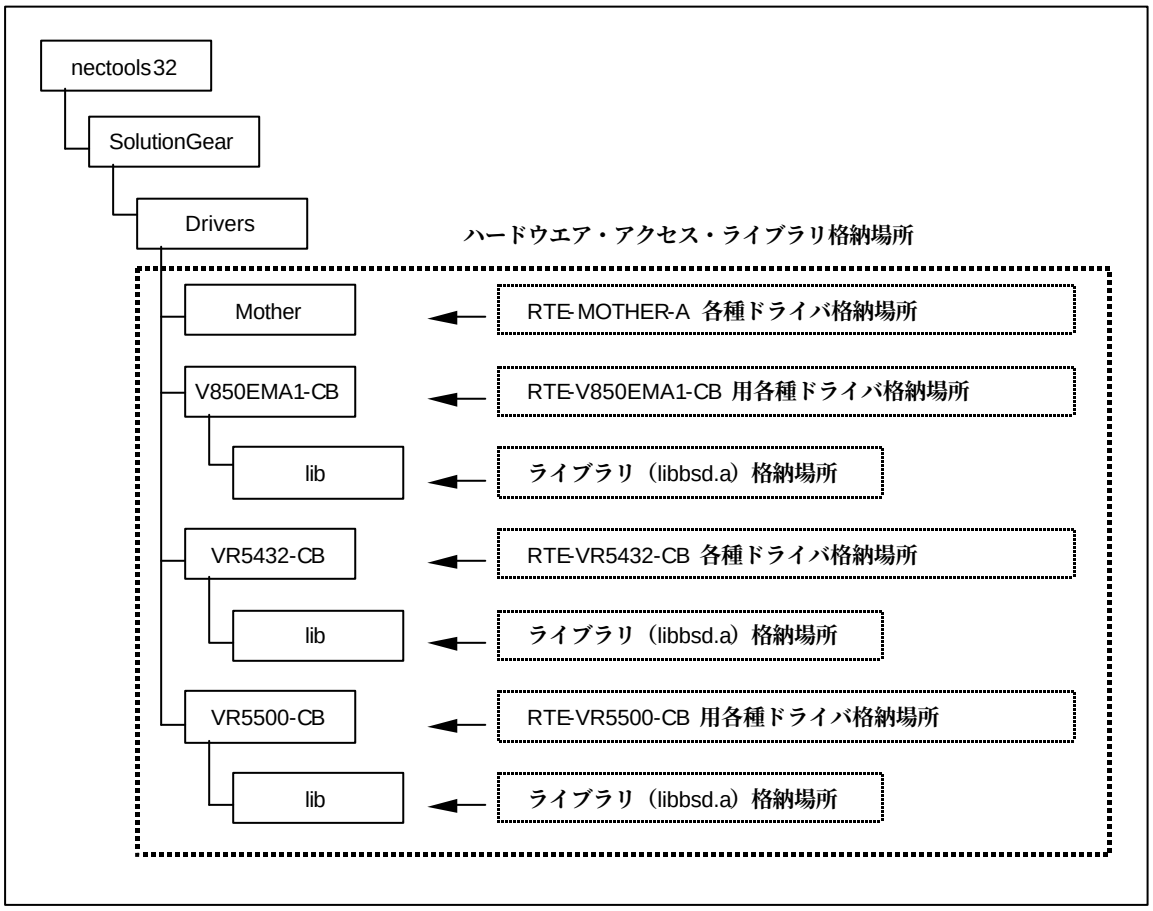
## 5. 1　サンプル・ライブラリの説明

### 5.1.1　ハードウエア・アクセス・ライブラリの格納場所

ハードウエア・アクセス・ライブラリは，Ethernet コントローラや UART コントローラの初期化処理とは別に，メモリやバスなどの周辺機器の初期化処理を行なう関数をまとめたライブラリです。SolutionGear で使用されているライブラリと同じものを使用しています。

ハードウエア・アクセス・ライブラリのある場所は，図 5－1の通りです。それぞれの CPU ボード名のフォルダー以下に，ソース・ファイル，ヘッダ・ファイル，ライブラリ・ファイルが格納されているフォルダーがあります（ライブラリの格納場所だけを表示しています）。またライブラリ・ファイルを構築する際に必要なプロジェクト・ファイル（NEC エレクトロニクス製），ビルド・ファイル（GHS 製）は，lib フォルダー以下に存在します。

図 5－1　ハードウエア・アクセス・ライブラリ格納場所



### 5.1.2　リファレンス・プラットフォーム・サンプル・ドライバ関数

RX-NET のサンプル・アプリケーション中で使用しているリファレンス・プラット・フォーム・サンプル・ドライバ関数について説明します。SolutionGear 以外の環境へ移植する場合は，以下の関数を使用している箇所を移植先のハードウエア用に書き換える必要があります。

1. `_BSDF_InitPCI()`

   SolutionGear の PCI バスを初期化します。この関数はサンプル・アプリケーションの varfunc.c で使用しています。すべてのサンプルで使用される関数です。

   **格納場所：** Drivers / Mother / PCI / PCI_9080.c

2. `_BSDF_InitISA()`

   SolutionGear の ISA バスを初期化します。この関数を発行する前に`_BSDF_InitPCI()`を実行しておく必要があります。この関数はサンプル・アプリケーションの varfunc.c で使用しています。ISA バスのネットワーク・デバイスを使用するサンプルで使用される関数です。

   **格納場所：** Mother / ISA / ISA_1523.c

3. `_BSDF_MB_InitInt1()`

   SolutionGear の GINT1 割り込みを初期化します。この関数はサンプル・アプリケーションの varfunc.c で使用しています。すべてのサンプルで使用される関数です。

   **格納場所：** Mother / INT / MB_INT.c

4. `_BSDF_ISA_InitInt()`

   SolutionGear の ISA バス割り込みを初期化します。先に`_BDDF_MB_InitInt1()`を実行しておく必要があります。この関数はサンプル・アプリケーションの varfunc.c で使用しています。ISA バスのネットワーク・デバイスを使用するサンプルで使用される関数です。

   **格納場所：** Mother / INT / ISA_INT.c

5. `_BSDF_OpenUART0()`

   SolutionGear の UART（RTE-MOTHER-A 上の JSIO1）を初期化します。この関数はサンプル・アプリケーションの varfunc.c で使用しています。すべてのサンプルで使用される関数です。

   **格納場所：** Mother / UART / UART_16552_ch0.c

6. `_BSDF_SendUART0()`

   SolutionGear の UART（RTE-MOTHER-A 上の JSIO1）の出力関数です。RX-NET のディバグメッセージ出力に使用します。この関数はコンフィギュレーション・オブジェクトの user_printf.c で使用しています。すべてのサンプルで使用される関数です。

   **格納場所：** Mother / UART / UART_16552_ch0.c

7. _BSDF_PCIIO_inp8()

PCI I/O 空間用の 1 バイトの I/O リード関数です。RX-NET のディバグメッセージ出力に使用します。この関数はコンフィギュレーション・オブジェクトのドライバ・ソースで使用しています。PCI バスのネットワークデバイスを使用するサンプルで使用される関数です。

**格納場所：** Drivers / ***ボード名*** / common / CB_CPU.c

8. _BSDF_inp8()

1 バイトの I/O リード関数です。この関数はコンフィギュレーション・オブジェクトのドライバ・ソースで使用しています。すべてのサンプルで使用される関数です。

**格納場所：** Drivers / ***ボード名*** / common / CB_CPU.c

9. _BSDF_inp16()

2 バイトの I/O リード関数です。この関数はコンフィギュレーション・オブジェクトのドライバ・ソースで使用しています。すべてのサンプルで使用される関数です。

**格納場所：** Drivers / ***ボード名*** / common / CB_CPU.c

10. _BSDF_outp8()

1 バイトの I/O ライト関数です。この関数はコンフィギュレーション・オブジェクトのドライバ・ソースで使用しています。すべてのサンプルで使用される関数です。

**格納場所：** Drivers / ***ボード名*** / common / CB_CPU.c

11. _BSDF_outp16()

2 バイトの I/O ライト関数です。この関数はコンフィギュレーション・オブジェクトのドライバ・ソースで使用しています。すべてのサンプルで使用される関数です。

**格納場所：** Drivers / ***ボード名*** / common / CB_CPU.c

12. _BSDF_acre_isr()

SolutionGear の割り込みハンドラ登録関数です。この関数はコンフィギュレーション・オブジェクトのドライバ・ソースで使用しています。すべてのサンプルで使用される関数です。

**格納場所：** Drivers / ***ボード名*** / common / RXdepend.c

13. _BSDF_del_isr()

SolutionGear の割り込みハンドラ削除関数です。この関数はコンフィギュレーション・オブジェクトのドライバ・ソースで使用しています。すべてのサンプルで使用される関数です。

**格納場所：** Drivers / ***ボード名*** / common / RXdepend.c

14. _BSDF_ena_int()

SolutionGear の割り込み許可関数です。この関数はコンフィギュレーション・オブジェクトのドライバ・ソースで使用しています。すべてのサンプルで使用される関数です。

**格納場所：** Drivers / ***ボード名*** / common / RXdepend.c

15. _BSDF_dis_int()

    SolutionGear の割り込み禁止関数です。この関数はコンフィギュレーション・オブジェクトのドライバ・ソースで使用しています。すべてのサンプルで使用される関数です。

    **格納場所：** Drivers / ***ボード名*** / common / RXdepend.c

16. _BSDF_SetPCICfgReg32()

    SolutionGear の 32 ビット PCI コンフィギュレーション・レジスタ設定関数です。この関数はコンフィギュレーション・オブジェクトのドライバ・ソースで使用しています。PCI バスのネットワーク・デバイスを使用するサンプルで使用される関数です。

    **格納場所：** Drivers / Mother / PCI / PCI.c

17. _BSDF_SetPCICfgReg8()

    SolutionGear の 8 ビット PCI コンフィギュレーション・レジスタ設定関数です。この関数はコンフィギュレーション・オブジェクトのドライバ・ソースで使用しています。PCI バスのネットワーク・デバイスを使用するサンプルで使用される関数です。

    **格納場所：** Drivers / Mother / PCI / PCI.c

18. _BSDF_ISA_EOI()

    SolutionGear の ISA バス割り込み終了処理関数です。この関数はコンフィギュレーション・オブジェクトのドライバ・ソースで使用しています。ISA バスのネットワーク・デバイスを使用するサンプルで使用される関数です。

    **格納場所：** Drivers / Mother / INT / ISA_INT.c

## 5.2　ハードウエア・アクセス・ライブラリの新規作成

使用するマザーボードが RTE-MOTHER-A ではないときなどは，使用するボードの初期化コードや関数を自作する必要があります。

まず必須のファイルや関数は次のようになります。

1. ボード初期化関数

   使用するハードウエアの初期化処理を行なう関数を作成してください。サンプルの InitMOTHER_A()がこれにあたります。この関数の呼び出しは RX850 Pro（RX4000V4）のハードウエア初期化部や初期化ハンドラ等から行ってください。

2. ヘッダファイル“ bsp.h ”

   添付のデバイス・ドライバに必ずインクルードされるヘッダファイルです。bsp のソース・ディレクトリ上に作成してください。このファイルでは BSP に依存したヘッダ・ファイルのインクルード等を行なっています。

次の関数も，たいていは必要となります。

1. I/O 関数

I/O 関数はデバイス・ドライバ，マザーボード等で C ソースから I/O アクセスを行なうときに使用するものです。

- １バイトの I/O リード関数（サンプルでは_inpb()）
- ２バイトの I/O リード関数（サンプルでは_inph()）
- ４バイトの I/O リード関数（サンプルでは_inpw()）
- １バイトの I/O ライト関数（サンプルでは_outpb()）
- ２バイトの I/O ライト関数（サンプルでは_outph()）
- ４バイトの I/O ライト関数（サンプルでは_outpw()）

2. １文字出力関数

サンプルでは user_printf.c にある user_printf()を使用しています。user_printf()は，出力文字を１文字ずつ_BSDF_SendUART0 関数で出力しています。SolutionGear 以外の環境で user_printf()関数を使用する場合は，変更する必要があります。サンプルでは，

- _BSDF_OpenUART0()　　サンプルでは varfunc.c で呼ばれています
- _BSDF_SendUART()　　サンプルでは user_printf.c で呼ばれています

を変更してください。

その他，必要な関数を用意してください。

これらの関数を記述したファイルを格納するディレクトリですが，図 5－1 中の“ MOTHER ”と同じ階層に新規ディレクトリを作成し，そこに格納するとよいでしょう。

# 第6章　API 関数

RX-NET を使用してソケットプログラミングをするとき，ソケットを生成する前にさまざまな初期化作業をする必要があります。RX-NET が必要とするテーブル情報やキューの作成，使 用する LAN コントローラ（ネットワーク・インタフェース）などの初期化です。

RX-NET には，初期化を行うための API 関数を用意しています。この章ではこれらについて説明します。

## 6. 1　RX-NETの初期化

RX-NET が提供する機能を実現する上で必要となる各種初期化処理を実行します。

### 6.1.1　so_initialize

so_initialize()プロトタイプ

```
int   so_initialize (void);
```

RX-NET が提供する機能を実現する上で必要となる各種初期化処理を実行します。RX-NET を使用するとき，最初に使用する関数になります。ソケット API の利用に先立って，この関数を呼び出す必要があります。

so_initialize()では，

- RAM 領域の初期化
- ソケットテーブル作成
- RX-NET で使用する RX850 Pro（RX4000V4）の資源（タスク・周期ハンドラ・セマフォ）の作成

を行います。

ハードウエア初期化は使用するボードによって異なりますが，V850E 用 RX-NET では，

- Midas Lab.製 “ RTE-V850E/MA1-CB（V850E/MA1 ボード） ”
- Midas Lab.製 “ RTE-MOTHER-A ”

用のサンプルを用意しています。なお, RTE-MOTHER-A のサンプルは SolutionGear と同じものを使用します。RX-NET のサンプルを動作させると, これらのボードの初期化処理が so_initialize()内で呼び出されます。ユーザの使用する環境に合わせて，これらの処理を参考に書き換える必要があります。これらの処理が記述されたサンプルの存在するディレクトリに関しては “ 5.1.2 リファレンス・プラットフォーム・サンプル・ドライバ関数 ” を参照してください。

これらの初期化関数の実行が正常終了した場合，次のメッセージが os_printf()により出力されます。

```
FNS device ＜デバイス名＞ installed at ＜アドレス＞
```

逆にデバイス・ドライバを見つけられなかった場合や，デバイス・ドライバの初期化関数がエラー終了した場合は，以下のメッセージが os_printf()により出力されます。

```
FNS starting up intra-machine only
```

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EALREADY | 0x52 | すでに so_initialize()が発行されています。 |

# 6.2　ネットワーク・インタフェースの起動・遮断

so_initialize()で初期化したネットワーク・インタフェースを起動し，指定した情報をもとに初期化します。

## 6.2.1 ll_config

ll_config()プロトタイプ

```
int  ll_config (int *devname, u32 ipaddress, u32 ipmask, unsigned  mtu, u32 ospfarea );
```

ll_config()（Link Layer Configuration）は，***devname*** で指定されたネットワーク・インタフェースを起動し，***ipaddress***，***ipmask***，***mtu*** で指定された情報を元に初期化します。

- 第一パラメータ“***devname***”は，ネットワーク・デバイス名へのポインタを指定します。ここで指定する値は，rxnetconf.c 内で定義される ndevsw に登録されているネットワークデバイス名に限られます。そのため，使用するネットワークデバイス情報を rxnetconf.c で指定する必要があります。この詳細については“2. 10　Ethernet コントローラ”を参照してください。
- 第二パラメータ“***ipaddress***”は，ネットワーク・インタフェースの IP アドレスを指定します。ここで指定する値はネットワーク・バイト・オーダー形式で指定します。ネットワーク・バイト・オーダーについては“6. 12　ネットワーク・バイト・オーダー”を参照してください。
- 第三パラメータ“***ipmask***”は，使用するネットワークのネットマスクの値を指定します。ここで指定する値はネットワーク・バイト・オーダー形式で指定します。ネットワーク・バイト・オーダーについては“6. 12　ネットワーク・バイト・オーダー”を参照してください。
- 第四パラメータ“***mtu***”は，TCP/IP の最大転送単位（MTU：Maximum Transmission Unit）の値を指定します。ディフォルト値は 1500 バイトです。ここで 0 を指定すると 1500 が指定されたものとみなして処理をします
- 第五パラメータ“***ospfarea***”は，システム予約領域で“0”固定です。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| ENODEV | 0x13 | ネットワーク・デバイス名 ***devname*** の指定が不正です |
| EINVAL | 0x16 | パラメータの指定が不正です |
| EALREADY | 0x52 | すでに ll_config()が発行されています。 |
| EAFNOSUPPORT | 0x5b | IP アドレス ***ipaddress*** の指定が不正です |

### 6.2.2 ll_unconfig

ll_unconfig()プロトタイプ

```
int  ll_unconfig ( int *devname, u32 ipaddress, u32 ipmask );
```

ll_unconfig()は，***devname***，***ipaddress***，***ipmask*** 指定されたネットワーク・インタフェースを遮断します。

- 第一パラメータ“***devname***”は，ネットワーク・デバイス名へのポインタを指定します。ここで指定する値は，rxnetconf.c 内で定義される ndevsw に登録されているネットワークデバイス名に限られます。そのため，使用するネットワークデバイス情報を rxnetconf.c で指定する必要があります。
- 第二パラメータ“***ipaddress***”は，ネットワーク・インタフェースの IP アドレスを指定します。ここで指定する値はネットワーク・バイト・オーダー形式で指定します。ネットワーク・バイト・オーダーについては“6. 12　ネットワーク・バイト・オーダー”を参照してください。
- 第三パラメータ“***ipmask***”は，使用するネットワークのネットマスクの値を指定します。ここで指定する値はネットワーク・バイト・オーダー形式で指定します。ネットワーク・バイト・オーダーについては“6. 12　ネットワーク・バイト・オーダー”を参照してください。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| ENODEV | 0x13 | ネットワーク・デバイス名 ***devname*** の指定が不正です |
| EALREADY | 0x52 | so_initialize()が発行されていません。 |

## 6.3　ルーティング・テーブルの登録・登録解除

RX-NET が管理する静的ルーティング・テーブルを操作します。ルーティング・テーブルとは，ホストやルータが管理している“宛先のネットワーク”と“そのネットワークと通信するためのネットワーク・インタフェイスやルータの対応表”を指します。Windows やUNIXでは“netstat –r” でその内容を表示できます。

### 6.3.1　ll_route

ll_route()プロトタイプ

```
int  ll_route ( char *devname, u32 gateway, u32 dipaddress, u32 dipmask, int  hops );
```

ll_route()は，***devname*** で指定されたネットワーク・インタフェースのルーティング・テーブルに，***gateway*** で指定されたアドレスを登録します。

- 第一パラメータ“***devname***”は，ネットワーク・デバイス名へのポインタを指定します。ここで指定する値は，rxnetconf.c 内で定義される ndevsw に登録されているネットワークデバイス名に限られます。
- 第二パラメータ“***gateway***”は，登録したいルータの IP アドレスを指定します。ここで指定する値はネットワーク・バイト・オーダー形式で指定します。ネットワーク・バイト・オーダーについては“6.12　ネットワーク・バイト・オーダー”を参照してください。
- 第三パラメータ“***dipaddress***”は，到達先のネットワーク・アドレスを指定しますが，現在は使用していません。“0”を指定してください。
- 第四パラメータ“***dipmask***”は，到達先のネットワークのネットマスクを指定しますが，現在は使用していません。“0”を指定してください。
- 第五パラメータ“***hops***”は，HOP の数を指定しますが，現在は使用していません。“0”を指定してください。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| ENODEV | 0x13 | ネットワーク・デバイス名 ***devname*** の指定が不正です |
| EINVAL | 0x16 | 引数が不正です |
| EALREADY | 0x52 | 指定されたルータの IP アドレスはすでに登録されています。 |

### 6.3.2 ll_del_static_route

ll_del_static_route()プロトタイプ

```
int  ll_del_static_route ( char *devname, u32 gateway, u32 dipaddress, u32 dipmask );
```

ll_del_static_route()は，***devname*** で指定されたネットワーク・インタフェースのルーティング・テーブルから，***gateway*** で指定されたアドレスを削除します。

- 第一パラメータ“***devname***”は，ネットワーク・デバイス名へのポインタを指定します。ここで指定する値は，rxnetconf.c 内で定義される ndevsw に登録されているネットワークデバイス名に限られます。
- 第二パラメータ“***gateway***”は，削除したいルータの IP アドレスを指定します。ここで指定する値はネットワーク・バイト・オーダー形式で指定します。ネットワーク・バイト・オーダーについては“6.12　ネットワーク・バイト・オーダー”を参照してください。
- 第三パラメータ“***dipaddress***”は，到達先のネットワーク・アドレスを指定しますが，現在は使用していません。“0”を指定してください。
- 第四パラメータ“***dipmask***”は，到達先のネットワークのネットマスクを指定しますが，現在は使用していません。“0”を指定してください。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| ENODEV | 0x13 | ネットワーク・デバイス名 ***devname*** の指定が不正です |
| EALREADY | 0x52 | 指定されたルータの IP アドレスはすでに登録されています。 |

# 6.4　テスト・コマンド

IP パケットが通信先まで届いているかどうかや，IP が到達可能かどうかを調べるために利用します。

## 6.4.1　ping

ping()プロトタイプ

```
int  ping ( u32 host, int datalen, int npackets );
```

ping()は，ICMP（Internet Control Message Protocol）ECHO パケットを送信する関数です。IP パケットが通信先まで届いているかどうかや，IP が到達可能かどうかを調べるために利用される，最も基本的なコマンドです。

ping を実行したとき，返答が返ってくれば相手のノードは存在し（少なくとも IP 層レベルにおいては）ネットワーク・ソフトウェアはアクティブになっていることが分かります。

- 第一パラメータ“***host***”は，パケットの送信先 IP アドレスをネットワーク・バイト・オーダーで指定します。ネットワーク・バイト・オーダーについては“6.12　ネットワーク・バイト・オーダー”を参照してください。
- 第二パラメータ“***datalen***”は，ICMP ECHO メッセージのデータ部分の長さを指定します。“1”以下の値を指定すると“1”が指定されたものとして動作します。4088 以上の値を指定すると 4088 が指定されたものとして動作します。パケットサイズとして表示されるパケットの大きさは，***datalen*** の値に 8 を加えたサイズになります。
- 第三パラメータ“***npackets***”は，送信パケット数を指定します。“0”以下の値を指定したときは“1”が指定されたものとして動作します。

この関数は ICMP ECHO パケットを送信しますが，送信したパケットに対する ECHO REPLY を ***npackets*** で指定した回数受信するか，受信する前にタイムアウト時間（1000ms）が経過すると関数から復帰します。通信結果は os_printf()経由で出力します。出力フォーマットは次のようになります。

- 1 パケットの応答があるたびに次の行を出力します

```
＜パケット・サイズ＞bytes from＜host で指定した IP＞：icmp_sn=<ID> time=＜往復にかかった時間＞
```

- npackets 回の受信後，統計情報を表示します

```
----＜host で指定した IP＞ PING Statistics＞----
＜送信した回数＞packets transmitted,＜npackets で指定した回数＞packets received,
＜損失率＞packet loss round-trip (ms)  min/avg/max = ＜最小時間＞/＜平均時間＞/＜最大時間＞
```

・ 全く応答がなかった場合，次の行を出力します

```
PING failed：‘peer not responding’ ‘Connection timed out’
----＜host で指定した IP＞ PING Statistics＞----
7 packets transmitted, 0 packets received, 100% packet loss
```

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| ETIMEDOUT | 0x67 | ICMP ECHO REPLYメッセージを受信できませんでした |

## 6.5　ソケット

次に RX-NET を使用した TCP/IP のプログラム作成において必要な“ ソケット ”について述べます。

ソケットとは TCP/IP を利用するアプリケーションのための API（Application Program Interface）を指します。もともとは UNIX に TCP/IP が実装されたとき，アプリケーション間通信の概念として考えられたことが始まりです。1983 年に ARPA（Advanced Research Projects Agency：米国国防総省の高等研究計画局）の支援を受けて作られた 4.2BSD（Berkeley Standard Distribution）に TCP/IP が実装されたことにより，プログラムから TCP/IP を使う API として用意されました。

ネットワーク上で通信する 2 つのプログラムは，それぞれソケットを 1 つ以上定義します。つまり，お互いが通信を始めるときは，まずそれぞれのソケットを使ってコネクション（通信路）をオープンし，そのソケットを通してデータの送受信を行うことになります。

## 6.6　コネクション型とコネクションレス型

### 6.6.1　TCPとUDP

インターネット・プロトコル群で多く使われるプロトコルは TCP（Transmission Control Protocol）と UDP（User Datagram Protocol）です。ネットワークのプログラムを作成する場合，この TCP と UDP を用途によって使い分けます。TCP は“ コネクション型 ”のプログラム，UDP は“ コネクションレス型 ”のプログラムに多く使われます。このコネクション型とコネクションレス型では，通信の流れが違ってきます。

### 6.6.2　コネクション型プログラムの通信の流れ

コネクション型プログラムの通信とは，通信を行う前に“ 通信路の確保 ”と“ 通信後の通信路の開放 ”を必要とする通信方法をいいます。そのため“ データを送受信する順番が決まっていないプログラム ”に多く用いられます。つまり，データを送受信するためには，始めに“ サーバープログラム ”と“ クライアントプログラム ”の接続を確立しておく必要があります。サーバープログラム，クライアントプログラムの接続方法を，それぞれの観点から見ると次のような流れになります。

1. サーバープログラム
   - ソケットを作成し（socket），それに名前を付ける（bind）
     → ソケットに名前を付けるとき，使用する“ポート番号”と“プロトコル（TCP または UDP）”を決定する。なおポート番号については“付録 Well-Known-Port”を参照してください。
   - クライアントからの接続を待つ（listen）
     → サーバープログラムはいつでもクライアントプログラムからの接続要求を受け入れる（accept）ことが可能になる

2. クライアントプログラム
   - ソケットを作成する（socket）
   - サーバーへ接続要求する（connect）
     → サーバーへの接続要求が通ると，この時点でサーバー側とクライアントの両側の識別情報（IP アドレスやポート番号）が確認され，サーバーとクライアントの間に一本の通信経路が確立します。これを仮想回線（virtual circuit）の確立といいます。

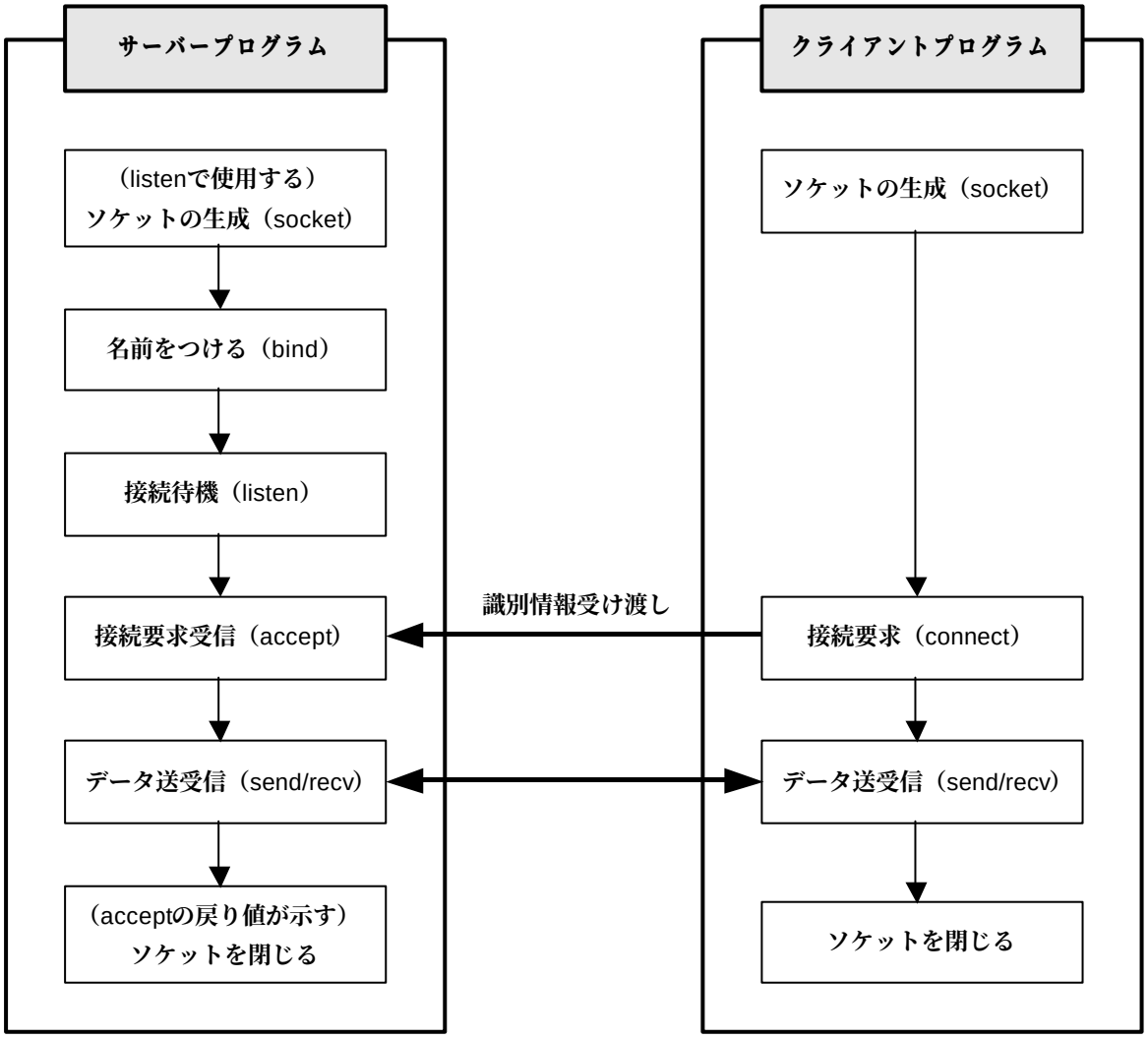
この後は両方のプログラムがソケットに対して送受信する（send / receive）ことができます。送受信が終了したらソケットを閉じます（closesock）。

なおソケットを閉じるタイミングですが，通常は“クライアントプログラムから”ソケットを閉じます。これによりサーバープログラムへ FIN パケットが送信され，クライアントプログラムがソケットを閉じたことがわかります。

クライアントプログラムに“bind”がありませんが，“bind”をしても間違いではありません。bind すると，そのポート番号が使用されます。上記のように bind を使用しなければ，RX-NET が適当なポートを割り当てます。

コネクション型プログラムの流れを図で表すと次のようになります。

図 6－1　コネクション型プログラムの通信の流れ




ここで注意が必要なのは，サーバー側のソケットは，クライアント側が connect する際に使われる時と，send / recv する際に使われる時で異なるということです。クライアント側がサーバー側に connect する際に使うソケットは，サーバー側が listen で待っているときのソケットです。クライアント側の connect 要求を受け入れると，サーバー側は accept を発行して通信用のソケットを用意し，このソケットを使ってクライアント側と send / recv します。そのためサーバー側のソケットを閉じるときは，accept したときに用意したソケットを閉じることになります。

### 6.6.3　コネクションレス型プログラムの通信の流れ

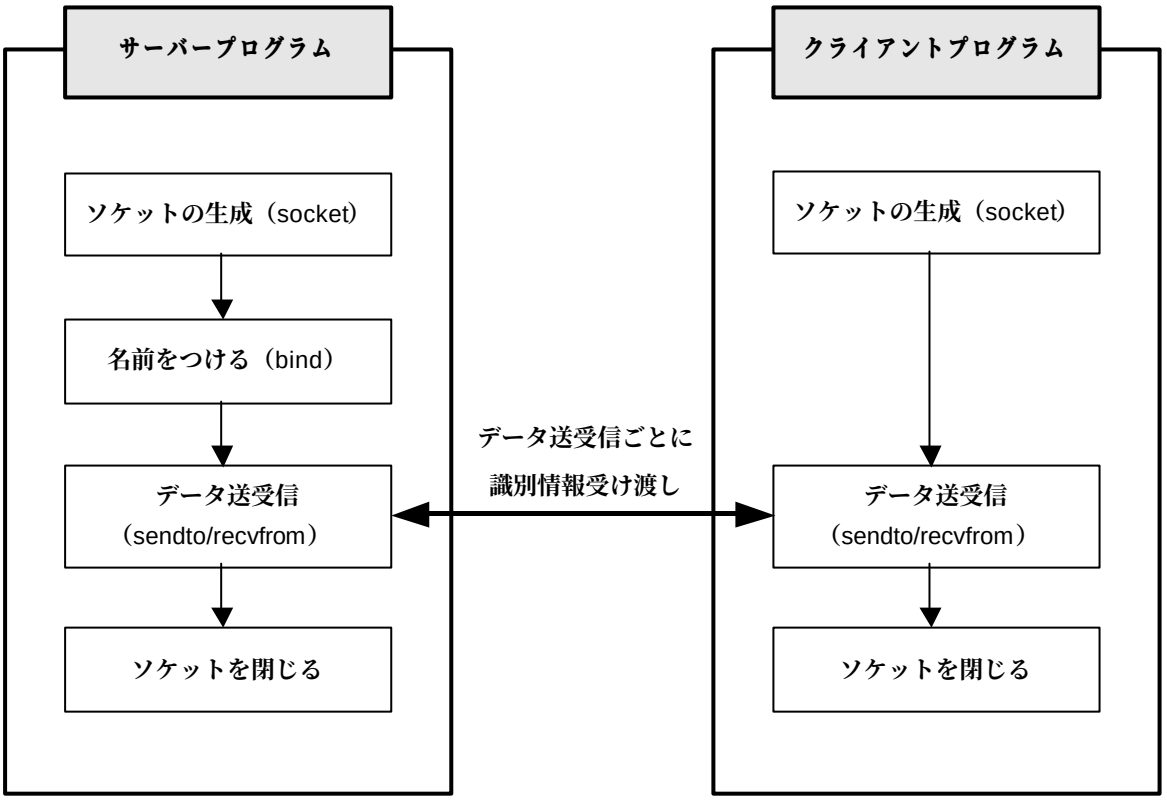
コネクション型プログラムの通信とは，通信を行う前に“通信路の確保”と“通信後の通信路の開放”が不要な通信方法をいいます。そのため“データをデータグラム（データを受信したら送信する，といったような順番が決まっている）として送受信するプログラム”に多く用いられます。コネクション型プログラムとは異なり，サーバーとクライアントで接続を確立せずにデータの送受信ができます。そのため，データを送受信するたびに識別情報（IP アドレスやポート番号）をデータに付けて送ることになります。また“ソケットを閉じるときにクライアントからサーバに通知が行われない”ところもコネクション型と違うところです。

1. サーバープログラム
   - ソケットを作成し（socket），それに名前を付ける（bind）
     → ソケットに名前を付けるとき，使用する“ポート番号”と“プロトコル（TCP または UDP）”を決定する。なおポート番号については“付録 Well-Known-Port”を参照してください。
   - データ受信を待つ（recvfrom）

2. クライアントプログラム
   - ソケットを作成する（socket）
   - データを送信する（sendto）
     → サーバープログラムは recvfrom の状態で待っているので，クライアント情報がわかり，データ受信後もクライアントへデータを返すことができるようになります。なお，クライアントプログラムもサーバープログラムへデータ送信後，recvfrom でデータ受信を待ちます。

送受信が終了したらソケットを閉じます。
なおソケットを閉じるタイミングですが，特に手順は特にありません。

図 6－2　コネクションレス型プログラムの通信の流れ



# 6. 7　ソケットの生成

## 6.7.1　ソケットの生成

サーバープログラムが接続を待つため，またはクライアントプログラムがサーバープログラムに接続するためには，まずソケットを生成する必要があります。ソケットを作成するのに必要な情報は「アドレス・ファミリ」「ソケットタイプ」「プロトコル」の 3 つです。それぞれ使用するプロトコルによって，ほぼ一意に決まります。

| | アドレス・ファミリ | ソケットタイプ | プロトコル |
|---|---|---|---|
| コネクション型プログラム | インターネット・アドレス・ファミリ | バイトストリーム | TCP |
| コネクションレス型プログラム | インターネット・アドレス・ファミリ | データグラム | UDP |

プロトコルは「アドレスファミリがインターネットアドレスファミリ」であり「ソケットタイプがバイトストリーム」であれば「TCP」を指定することになります。ですから「ソケットタイプがバイトストリーム」なのに「UDP」を指定した場合は，システム側（ソケット内部）での保証外となってしまいます。

またプロトコルは，サーバーとクライアントで必ず同じものを指定しなければ通信ができません。

## 6.7.2　ソケット記述子

ソケットを生成すると，システムからソケット記述子を取得できます。ソケット記述子とは，複数のソケットがあった場合に，それらを識別するため使用する名前のようなものです。ソケット記述子取得後は，ソケットに対しての接続要求や，データの送受信をしている間は，このソケット記述子でコネクション（仮想回線：virtual circuit）を区別することになります。

## 6.7.3　ソケットとポート番号の関連付け

サーバープログラムは，クライアントプログラムからの接続要求を待つために，そのソケットがどのポート番号への接続要求を待つかを指定します。これを行なうのが“ bind ”です。ポート番号とは，1 つの IP アドレス（コンピュータ）上で動作する Telnet や FTP などのアプリケーションに対し，データを渡すためのアドレスを意味します。ポート番号の付け方や詳細については“ 付録 Well-Known-Port ”を参照してください。この bind により，システムに対して“ プログラムのポート番号 ”を登録したことになります。これで「同一システム上のプログラム」と「自分のプログラム」とのデータの振り分けが可能になります。相手側のクライアントプログラムは，このポート番号に向けて接続することになります。これでサーバープログラムは待機する準備ができたことになります。

### 6.7.4 socket

socket()プロトタイプ

```
int   socket ( int af,  int type, int protocol,  int *errp );
```

ソケットを生成します。socket()が完了すると，戻り値としてソケット記述子が返されます。これ以降，そのソケットに対する操作は，ここで与えられたソケット記述子を使用します。サーバープログラムではlisten()用のソケット記述子として使用します。accept()発行後に行うデータ送受信には，accept()の戻り値として返されるソケット記述子を使います。

socket()の処理内容は socket.h で定義されているソケットレベルオプションで制御されます。オプションを設定したり，状態を取得したりするには，それぞれ setsopt()，getsopt()を使用します。またディフォールトでは，ソケットは“ブロッキング・モード”で生成されます。また，ソケットのクローズは std.h で定義されている closesock()で行うことができます。

- 第一パラメータ“***af***”は，アドレスファミリを指定します。RX-NET では“ AF_INET ”（インターネットアドレスファミリ）のみをサポートしているので，必ず“ AF_INET（=0x2） ”を指定します。
- 第二パラメータ“***type***”は，ソケットタイプを指定します。つまり通信の方法を定めるために，ソケットの型を指定します。
  TCP を使用する場合は，バイトストリームなので“ SOCK_STREAM（=0x1） ”を指定します。UDP を使用する場合は，データグラムなので“ SOCK_DGRAM（=0x2） ”を指定します。その他“ SOCK_RAW（=0x3） ”も指定可能です。これは生データで，内部ネットワーク・インタフェースにアクセスを提供します。
- 第三パラメータ“***protocol***”は，プロトコルを指定します。RX-NET では第一パラメータ，第二パラメータから自動的に判別するので“ 0 ”を指定します。
- 第四パラメータ“****errp***”は，戻り値が“ -1 ”（異常終了）だった場合のエラーコードが格納されます。ここに格納されるエラーは次の通りです。

| マクロ | 値 | 作 用 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x17 | 最大ソケット生成数を越えました。最大ソケット生成数はユーザオウンで，rxnetconf.h に定義（MAX_SOCKET）されています。 |
| EPROTONOSUPPORT | 0x58 | ***protocol*** の指定が不正です |
| ESOCKTNOSUPPORT | 0x59 | ***type*** の指定が不正です。 |
| EAFNOSUPPORT | 0x5b | ***af*** の指定が不正です |
| ENETDOWN | 0x5e | ネットワークがダウンしています。または RX-NET が初期化されていません。 |

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| その他 | -1 以外 | 正常終了。生成されたソケットのソケット記述子が返されます。 |
| EINVAL | -1 | 異常終了。対応するエラーコードが ***errp*** で指定された領域に格納されます。 |

備考　SOCK_STREAM タイプを実装するのに使われる通信プロトコルを用いると，データ損失や二重生成をされないことが保証されます。ピア・プロトコルがバッファ領域を提供しているデータが適正な時間内に伝送できない場合には，その接続は中断されていると判断され，“ -1 ”が返るとともに ***errp*** が ETIMEDOUT に設定されます。

SO_KEEPALIVE オプションが ON の場合，ソケットは“ ウォーム ”され続け，他の活動はせずにプロトコルに依存した周波数で強制的に伝送を継続します。他の用途には使われていない接続であるにもかかわらず，時間（この値もプロトコルに依って異なります）が過ぎても応答が得られない場合，エラーとみなされます。

SOCK_DGRAM ソケットを使うと，送信コールと受信コールで指名されている相手にデータグラムやメッセージを送信したり，これらを受け取ったりすることができます。

### 6.7.5 bind

bind()プロトタイプ

```
int   bind ( int sd,  saddr *name,  int name_len );
```

作成したソケットと TCP/UDP プロトコルのポート番号との関連付け（ソケットに名前をつける）を行ないます。

- 第一パラメータ“ *sd* ”は，ソケット記述子を指定します。
- 第二パラメータ“ *name* ”は，sockaddr_in 構造体のポインタを渡します。ただし関数プロトタイプ宣言との矛盾を回避するために *saddr** へのキャストが必要となります。

［ sockaddr_in 構造体プロトタイプと in_addr 構造体プロトタイプ ］

```
struct  sockaddr_in {
    i16     sin_family;
    u16     sin_port;
    struct in_addr  sin_addr;
    char    sin_zero[8];
}
```

```
struct  in_addr {
    union {
        struct{char s_b1, s_b2, s_b3, s_b4;}S_un_b;
        struct{u16 s_w1, s_w2;}S_un_w;
        long    S_addr;
    }S_un;
};
#define  s_addr  S_un.S_addr
```

| メンバ | 説 明 |
|---|---|
| sin_family | アドレスファミリを指定（必ず AF_INET） |
| sin_port | ポート番号を指定 |
| sin_addr | in_addr 構造体。IP アドレスを 3 種類の共用体で表現している構造体。<br>bind()の場合は s_addr を使って，接続可能な IP アドレスを指定する。 |
| sin_zero | 未使用領域 |

- 第三パラメータ“ *name_len* ”は，第二パラメータ *name* で指定された sockaddr_in 構造体の大きさ（バイト）を指定します。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・`listen()`発行後に呼び出した<br>・`connect()`発行後に呼び出した<br>・不正なソケット記述子です<br>・***name*** が NULL です<br>・***namelen*** が“0”です<br>・RX-NET が初期化されていません |
| EAFNOSUPPORT | 0x5b | 対象ソケットのアドレスファミリが AF_INET ではありません |
| EADDRINUSE | 0x5c | IP アドレスの指定が不正です |
| ENETDOWN | 0x5e | ネットワークがダウンしています |

## 6.8　ソケットの接続

ソケットが生成されると，次は接続準備に入ります。サーバー側とクライアント側で，接続される側と接続する側となり，それぞれ違う工程になります。

### 6.8.1　コネクション型におけるサーバー側の接続

ソケットを生成（socket）し，bind によってポート番号との関連付けが行われると，次にクライアント側からの接続を待ちます。接続を待つためには“ listen ”を発行します。ただし“ 待つ ”といっても，接続を受け入れるための準備ができたことをシステムに知らせ，クライアントプログラムからの接続要求をシステム側で保持してくれるためのキューを用意することを意味します。そのため listen を呼んでも，サーバープログラムがクライアントプログラムからの接続要求を待つために，待ち状態に入ってしまうことはありません。この時点で接続要求が来た場合，接続要求はキューに格納されます。

次に accept を発行し，実際にクライアントプログラムからの接続要求を受け入れます。つまりキューに接続要求が入っている場合，それを取り出します。取り出すと accept は新しいソケット記述子を返します。今後はこの新しいソケット記述子を使ってデータの送受信をすることになります。この accept に関しては，システムによって「クライアントプログラムの接続要求が来るまで待ってしまう場合」と「待たずにエラーとなる場合」があります。RX-NET では前者になります。ただし listen に使用するソケットがノンブロッキングモードだった場合は後者になります。

### 6.8.2　コネクション型におけるクライアント側の接続

クライアント側もソケットの作成に関してはサーバー側と全く同じ工程になります。クライアントプログラムがサーバープログラムに接続するために必要な情報は“ IP アドレス ”と“ ポート番号 ”です。

接続要求するには“ connect ”を発行します。この connect を発行するときに“ IP アドレス ”と“ ポート番号 ”が必要になります。connect が完了すると，サーバーとのコネクション（仮想回線）が確立した（サーバーが accept した）ことになり，サーバーに対してデータの送信が行える状態になります。これは「クライアントのソケット」と「目的のサーバーのソケット」との関連付けができたことになり，ソケット記述子を使ってサーバーにデータの送受信ができることを意味します。

### 6.8.3　コネクションレス型の接続

コネクションレス型の場合，前にも述べたように接続という概念はありません。データを送受信する前に，サーバーの場合は bind し，ポート番号とソケットの関連付けをしてデータの受信待ちになります。クライアントの場合は，ソケットを作るとすぐにデータ送信が可能になります。

### 6.8.4 listen

listen()プロトタイプ

```
int   listen ( int sd,  int backlog );
```

クライアントからの接続要求を待機するため，接続要求をシステム側で保持するためのキューを用意します。

- 第一パラメータ“ ***sd*** ”は，ソケット記述子を指定します。SOCK_STREAM型のソケットだけに適用します。
- 第二パラメータ“ ***backlog*** ”は，接続を保留できるキューの最大値を指定します。一般的には 1 から 4 ぐらいで，もし無効な値を指定すると近い値が採用されます。RX-NET では 1 から 5 まで指定可能で，5 よりも大きい値を指定した場合は 5 が指定されたものとして処理されます。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正なソケット記述子です<br>・RX-NET が初期化されていません |
| EOPNOTSUPP | 0x5a | 対象ソケットのサービスタイプがSOCK_STREAMではありません |
| EISCONN | 0x64 | すでに listen 発行済のソケットが指定されました |

注意　接続キューの空きがない状況で接続要求が来ると，クライアントに対してエラー“ ECONNREFUSED ”を返す場合があります。

### 6.8.5 accept

accept()プロトタイプ

```
int   accept ( int sd,  saddr *addrp, int *addrlen,  int *errp );
```

listen()で待機中に接続要求があったので，接続を受け入れます。具体的には，確立されたコネクションに対応した新しいソケット（引数 ***sd*** とは別のもの）を，接続要求を待たせているキューから取り出し，返却します。この関数は SOCK_STREAM 型のソケットだけで利用可能です。

accept()が完了すると，関数の戻り値としてソケット記述子が返されます。このソケット記述子はクライアントと関連付けされているソケット記述子になります。以降，クライアントと送受信する場合は，このソケット記述子を使用します。また ***addrp*** にはクライアントの情報が格納されてきます。

- 第一パラメータ“ ***sd*** ”は，ソケット記述子を指定します。このソケット記述子は listen()で使用しているものです。
- 第二パラメータ“ ***addrp*** ”は，**sockaddr_in** 構造体のポインタを渡します。ただし関数プロトタイプ宣言との矛盾を回避するために ***saddr**へのキャストが必要となります。***addrp*** が NULL ポインタ（0）の場合には，接続相手のアドレスは格納されません。相手のアドレスが不要な場合に利用します。

［ **sockaddr_in 構造体プロトタイプと in_addr 構造体プロトタイプ** ］

```
struct  sockaddr_in {
    i16     sin_family;
    u16     sin_port;
    struct in_addr  sin_addr;
    char    sin_zero[8];
}
```

```
struct  in_addr {
    union {
        struct{char s_b1, s_b2, s_b3, s_b4;}S_un_b;
        struct{u16 s_w1, s_w2;}S_un_w;
        long    S_addr;
    }S_un;
};
#define  s_addr  S_un.S_addr
```

| メンバ | 説 明 |
|---|---|
| **sin_family** | アドレスファミリを指定（必ず AF_INET） |
| **sin_port** | ポート番号を指定 |
| **sin_addr** | **in_addr** 構造体。IP アドレスを 3 種類の共用体で表現している構造体。<br>**accept()**の場合は **s_addr** を使って，接続相手の IP アドレスを指定する。 |
| **sin_zero** | 未使用領域 |

- 第三パラメータ“ ***addrlen*** ”は，第二パラメータの **sockaddr_in** 構造体の大きさを入れた整数値のポインタを指定します。***addrlen*** が NULL ポインタ（0）の場合には，接続相手のアドレスは格納されません。相手のアドレスが不要な場合に利用します。
- 第四パラメータ“ ****errp*** ”は，戻り値が“ -1 ”（異常終了）だった場合のエラーコードが格納されます。ここに格納されるエラーは次の通りです。

| マクロ | 値 | 作 用 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EACCESS | 0xd | ソケットへのアクセスが拒否されました。次のような原因が考えられます。<br>・listen()未呼び出しのソケットを使用しています |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正なソケット記述子です<br>・***addrlen*** が短いです<br>・RX-NET が初期化されていません |
| EWOULDBLOCK | 0x50 | ソケットがノンブロッキング・モードです |
| EOPNOTSUPP | 0x5a | サービス・タイプが SOCK_STREAM ではありません |
| ENETDOWN | 0x5e | ネットワークがダウンしています |

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| － | -1 以外 | 正常終了。クライアントと関連付けされたソケット記述子が返ります。 |
| － | -1 | 異常終了。対応するエラーコードが ***errp*** で指定された領域に格納されます。 |

### 6.8.6 connect

connect()プロトタイプ

```
int   connect ( int sd,  saddr *name, int name_len );
```

ソケット記述子に対し，接続を確立します（アクティブオープンをかけます）。

SOCK_STREAM 型ソケットの場合，connect()は ***name*** で指定されたアドレスへの接続を行います。SOCK_DGRAM 型ソケットや SOCK_RAW 型ソケットのようなコネクションレスソケットの場合には，***name*** で指定されたアドレスを，***sd*** で指定されたソケットに“宛て先アドレス”を設定します。この設定は send() のためのディフォールトの宛て先アドレスとして利用されます。このディフォールトのアドレスは sendto() では無視されます。

- 第一パラメータ“***sd***”は，ソケット記述子を指定します。
- 第二パラメータ“***name***”は接続相手のアドレスを格納する格納するアドレス構造体へのポインタを指定します。ここは **sockaddr_in** 構造体のポインタを渡します。ただし関数プロトタイプ宣言との矛盾を回避するために ***saddr****へのキャストが必要となります。

［ **sockaddr_in** 構造体プロトタイプと **in_addr** 構造体プロトタイプ ］

```
struct  sockaddr_in {
    i16     sin_family;
    u16     sin_port;
    struct in_addr  sin_addr;
    char    sin_zero[8];
}
```

```
struct  in_addr {
    union {
        struct{char s_b1, s_b2, s_b3, s_b4;}S_un_b;
        struct{u16 s_w1, s_w2;}S_un_w;
        long    S_addr;
    }S_un;
};
#define  s_addr  S_un.S_addr
```

| メンバ | 説 明 |
|---|---|
| **sin_family** | アドレスファミリを指定（必ず AF_INET） |
| **sin_port** | ポート番号を指定 |
| **sin_addr** | **in_addr** 構造体。IP アドレスを 3 種類の共用体で表現している構造体。<br>**connect()**の場合は **s_addr** を使って，接続相手の IP アドレスを指定する。 |
| **sin_zero** | 未使用領域 |

・ 第三パラメータ“ *addrlen* ”は，第二パラメータの *name* で指定される **sockaddr_in** 構造体の大きさ（バイト）を入れた整数値のポインタを指定します。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正なソケット記述子です<br>・*name* が NULL です<br>・*name_len* が 0 です<br>・RX-NET が初期化されていません |
| EWOULDBLOCK | 0x50 | 対象ソケットがノンブロッキング・モードです |
| EAFNOSUPPORT | 0x5b | 対象ソケットのアドレスファミリが AF_INET ではありません |
| ETIMEDOUT | 0x67 | 接続要求がタイムアウトしました |
| ECONNREFUSED | 0x68 | 遠隔側のソケットにおける IP アドレス，ポート番号の割り付け（bind），および受動モードへの移行（listen）が完了していません。 |

# 6. 9　コネクション型のデータ送受信

データの送受信については，サーバー／クライアントとも考え方は同じです。ソケット記述子を使用して send / recv を行ないます。

データの送受信で重要となるのが再送処理，つまり“ リトライ ”です。LAN／WAN 環境では自分の通信相手とは全く関係ない部分から受ける影響が非常に大きいため，1 回で send／recv したデータの送受信が完了するとは限らないためです。プログラムを作成する上でこのリトライの処理は必須になります。

またコネクション型とコネクションレス型では，送受信で使用する API 関数が異なります。

## 6.9.1　コネクション型のデータ送信

コネクション型のデータ送信は“ send ”を使用します。サーバーからでもクライアントからでも，データを送信するタイミングは TCP/IP のプロトコル規定上, どちらが先に送信しなければならないといったような決まりごとはありません（TCP の上位プロトコルで決まる場合はあります）。send で送信したデータをすべて相手に送信できたかどうかを判定する責任はプログラムにあります。send を発行すると，送信できたバイト数が戻り値に格納されます。もしすべてのデータを送信しきれていなければ, もう一度送信できなかったデータを send する必要があります。

## 6.9.2　コネクション型のデータ受信

コネクション型のデータ受信は“ recv ”を使用します。send と同じように一度にすべてが受信できるとは限らないため，データをすべて取り込むまで受信する必要があります。すべて正常に受信できたかを判定する責任はプログラムにあります。この確認はアプリケーションで行なう必要があります。

recv はシステムによって受信すべきデータがない場合の待機方法が異なる場合と，選択できる場合があります。RX-NET ではこれらを選択することが可能です。相手のプログラムからデータが来ないとき「来るまで待つ方法」と「エラーを返す方法」を選択できます。前者を「ブロッキング・モード（同期型）」，後者を「ノンブロッキング・モード（非同期型）」と呼びます。

これらはシステムによって異なります。受信データの退避方法として，割り込みや一定時間ごとに調査する“ ポーリング ”方式などがあります。

### 6.9.3 send

send()プロトタイプ

```
int   send ( int sd,  char *buf,  int buf_len, int flags,  int *errp );
```

データを送信します。一度にすべてのデータが送信されるとは限りません。戻り値に送信したバイト数が返されるので，すべてが送信されるまで何度も send する必要があります。

- 第一パラメータ“ ***sd*** ”は，ソケット記述子を指定します。（正常終了した socket()の戻り値，または正常終了した accept()の戻り値）
- 第二パラメータ“ ***buf*** ”は，送信するデータのバッファ（送信バッファ）ポインタを指定します。
- 第三パラメータ“ ***buf_len*** ”は，第二パラメータ ***buf*** で指定したバッファのサイズ（バイト）を指定します。
- 第四パラメータ“ ***flags*** ”は，データの送信方法を指定します。ここで指定する値は socket.h で定義されている下記のフラグ値そのもの，あるいはこれらの論理和になります。

| 送信制御フラグ | 値 | 説 明 |
|---|---|---|
| MSG_OOB | 0x1 | 緊急データを送信する |
| MSG_DONTROUTE | 0x4 | 経路制御情報（ルーティング・テーブル）を使用せずに送信する（診断／ルーティングプログラムで使用するオプション） |
| MSG_FDBROADCAST | 0x2000 | 全ニ重ブロードキャストを送信する |
| MSG_NONBLOCKING | 0x4000 | ノンブロッキング・モードで送信する |
| MSG_BLOCKING | 0x8000 | ブロッキング・モードで送信する |

備考1　MSG_OOB は緊急データを送信するためのものです。緊急データの引渡しに対して，どのような特別処理を行うかは，アプリケーションに任されています。緊急データは，IST プロトコル，IDG プロトコル，および TCP に対してだけ実装されています。RX-NET の送信ルーチンでは，MSG_OOB が受信ルーチンで使われる MSG_URGENT に対応しています。両方とも緊急データを扱います。

備考2　MSG_FDBROADCAST を使うと，ブロードキャストされたパケットがこのノードでも受信されることになります。ディフォールトでは自身がブロードキャストしたものは受信しないようになっています。

・ 第四パラメータ“ ***errp*** ”は，戻り値が“ -1 ”（異常終了）だった場合のエラーコードが格納されます。
ここに格納されるエラーは次の通りです。

| マクロ | 値 | 作 用 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正なソケット記述子です<br>・RX-NET が初期化されていません |
| EWOULDBLOCK | 0x50 | ソケットがノンブロッキング・モードで，かつメッセージ送信処理の実行条件が整っていません |
| ENETDOWN | 0x5e | ネットワークがダウンしています |
| ENOBUFS | 0x63 | 送信するメッセージのサイズが大きすぎます |
| ENETCONN | 0x65 | 接続が確立していません |

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| － | -1 以外 | 正常終了。送信されたバイト数が格納されます。 |
| － | -1 | 異常終了。対応するエラーコードが ***errp*** で指定された領域に格納されます。 |

注意1　send を発行したとき，対象ソケットが“ ノンブロッキング・モード ”でかつ“ メッセージ送信処理の実行条件が整っていなかった ”場合，メッセージ送信処理は行わず，戻り値として“ -1 ”を，エラーコードとして“ EWOULDBLOCK ”を返します。

注意2　対象ソケットが“ データグラム型（SOCK_DGRAM） ”であった場合，送信可能なメッセージ・サイズは“ 0x1000（4096）バイト ”です。このため，メッセージを格納した ***buf*** のサイズ“ ***buf_len*** ”に 0x1000 以上の値を指定した場合には，メッセージの送信は行わず，戻り値として“ -1 ”を，エラーコードとして“ ENOBUFS ”を返します。また，メッセージが長すぎて，基本のプロトコルを少しも通過できなかった場合や，プロトコルがメッセージの断片化をサポートしていない場合には“ -1 ”の値が戻され，***errp*** に EMSGSIZE が設定されます。

注意3　利用可能なメッセージ領域がないために（プロトコルのウインドウの空きがないなど），メッセージをすぐには送信できない場合，送信ルーチンはソケットがブロッキングモードにあるかどうかを検出し，検出結果に適応した処置を行います。送信ルーチンは，ブロッキングモードにあるソケットに対し，データが送出されるまで呼び出し処理をブロックします。ソケットがノンブロッキング I/O モードにあるか，MSG_NONBLOCKING フラグが設定されている場合には，ERR（-1）の値が直ちに戻され，***errp*** は EWOULDBLOCK に設定されます。

**注意4**　send()が正常終了したということが，メッセージが相手先ホストに無事届いたということを意味しているわけではありません。“-1”が戻ったということは，ほとんどのプロトコルの場合，ローカルなエラーが検出されたことを示しています。nselect()を使って，いつデータを送信再開できるかを調べることができる場合もあります。

### 6.9.4 recv

recv()プロトタイプ

```
int   recv ( int sd,  char *buf,  int buf_len, int flags,  int *errp );
```

データを受信します。一度にすべてのデータが受信できるとは限りません。戻り値に受信したバイト数が返されるので，すべてが受信されるまで何度もrecvする必要があります。

- 第一パラメータ“***sd***”は，ソケット記述子を指定します。
- 第二パラメータ“***buf***”は，メッセージを格納するバッファ（受信バッファ）ポインタを指定します。
- 第三パラメータ“***buf_len***”は，第二パラメータ***buf***で指定した受信バッファのサイズ（バイト）を指定します。
- 第四パラメータ“***flags***”は，データの受信方法を指定します。

| 送信制御フラグ | 値 | 説 明 |
|---|---|---|
| MSG_PEEK | 0x2 | データを受信。読み出し位置の変更はなく，続くrecvで同じデータを参照できます。 |
| MSG_URGENT | 0x800 | 緊急データを受信する |
| MSG_TRUNCATE | 0x1000 | 受信パケットの切り捨て |
| MSG_NONBLOCKING | 0x4000 | ノンブロッキング・モードで受信する |
| MSG_BLOCKING | 0x8000 | ブロッキング・モードで受信する |

備考1　MSG_PEEKを使うと，読み出し位置を変更することなく，そのデータを読み取ることができます。この場合，次の（MSG_PEEKを指定しない）recv()呼び出しでそのデータを読み取り，読み出し位置が変更されることになります。

備考2　MSG_TRUNCATEを用いると，パケット内のデータが，recv()で指定された長さに切り捨てられます。このフラグのみがSOCK_DGRAMのようなパケット型プロトコルに有効です。MSG_TRUNCATEを指定しない場合は，複数のrecv()呼び出しで，パケット内のすべてのデータを読み出すことができます。

備考3　MSG_URGENTは緊急データを受信します。緊急データが存在しない場合，recv()は“-1”を返し，***errp***はENOURGENTDATAに設定されることになります。緊急データとは，MSG_OOBフラグをオンにして送信されるデータのことをいいます。緊急データの引渡しに対して，どのような特別処理を行うかは，アプリケーションに任されています。緊急データはISTプロトコル，IDGプロトコルおよびTCPに対してだけ実装されています。ソケットにデータがなく，かつソケットがブロッキングモードにあると，受信コールはパケットが届くのを待ちます。しかしながらソケットがノンブロッキングモードにあるか，MSG_NONBLOCKINGフラグが使われていた場合，-1という値が直ちに戻され，***errp***がEWOULDBLOCKに設定されます。いつ残りのデータが届くかを調べるためには，nselect()を使うことができます。

・ 第四パラメータ“ ***errp*** ”は，戻り値が“ -1 ”（異常終了）だった場合のエラーコードが格納されます。
ここに格納されるエラーは次の通りです。

| マクロ | 値 | 作 用 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正なソケット記述子です<br>・RX-NET が初期化されていません |
| EWOULDBLOCK | 0x50 | ソケットがノンブロッキング・モードです |
| ENETCONN | 0x65 | 接続が確立していません |

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ― | -1 以外 | 正常終了。受信されたバイト数が格納されます。<br>なお 0x0 が返された場合は，メッセージの受信処理が完了する以前に対象ソケットの解放が行われたことを意味します。 |
| ― | -1 | 異常終了。対応するエラーコードが ***errp*** で指定された領域に格納されます。 |

# 6. 10　コネクションレス型のデータ送受信

## 6.10.1　コネクションレス型のデータ送信

コネクションレス型のデータ送信は“ sendto ”を使用します。コネクションレス型の場合，データの送信前に接続が行われていないので，データ送信の際に送信先のアドレスを指定することになります。送信・受信の流れが決まっているため，sendto を最初に行なうのはクライアントになります。

## 6.10.2　コネクションレス型のデータ受信

コネクションレス型のデータ受信は“ recvfrom ”を使用します。recvfrom の行なう処理は recv とほとんど同じですが，送信元のアドレスを受け取るという点が異なります。

またコネクションレス型の場合，sendto と recvfrom の組み合わせが必要となります。クライアントが sendto で 10 バイト送信を 2 回行なった場合，サーバーは合計 20 バイト受け取る必要がありますが，recvfrom では 1 回で 1 パケット分のデータ（10 バイト）しか受信できないため，2 回の recvfrom が必要となります。

### 6.10.3 sendto

sendto()プロトタイプ

```
int  sendto (int sd, char *buf, int buf_len, int flags, saddr *to, int tolen, int *errp );
```

データを送信します。sendto は送信先のアドレス情報を含めて送信します。

- 第一パラメータ“ ***sd*** ”は，ソケット記述子を指定します。（正常終了した socket()の戻り値）
- 第二パラメータ“ ***buf*** ”は，送信するデータのバッファ（送信バッファ）ポインタを指定します。
- 第三パラメータ“ ***buf_len*** ”は，第二パラメータ ***buf*** で指定したバッファのサイズ（バイト）を指定します。
- 第四パラメータ“ ***flags*** ”は，データの送信方法を指定します。ここで指定する値は socket.h で定義されている下記のフラグ値そのもの，あるいはこれらの論理和になります。

| 送信制御フラグ | 値 | 説 明 |
|---|---|---|
| MSG_OOB | 0x1 | 緊急データを送信する |
| MSG_DONTROUTE | 0x4 | 経路制御情報（ルーティング・テーブル）を使用せずに送信する（診断／ルーティングプログラムで使用するオプション） |
| MSG_FDBROADCAST | 0x2000 | 全ニ重ブロードキャストを送信する |
| MSG_NONBLOCKING | 0x4000 | ノンブロッキング・モードで送信する |
| MSG_BLOCKING | 0x8000 | ブロッキング・モードで送信する |

備考1　MSG_OOB は緊急データを送信するためのものです。緊急データの引渡しに対して，どのような特別処理を行うかは，アプリケーションに任されています。緊急データは，IST プロトコル，IDG プロトコル，および TCP に対してだけ実装されています。RX-NET の送信ルーチンでは，MSG_OOB が受信ルーチンで使われる MSG_URGENT に対応しています。両方とも緊急データを扱います。

備考2　MSG_FDBROADCAST を使うと，ブロードキャストされたパケットがこのノードでも受信されることになります。ディフォールトではユーザがブロードキャストしたものは受信しないようになっています。

・ 第五パラメータ“ ***to*** ”は，データを送信するターゲットホストの“ アドレス構造体へのポインタ ”を指定します。ここは **sockaddr_in** 構造体のポインタを渡します。ただし関数プロトタイプ宣言との矛盾を回避するために ***saddr**** へのキャストが必要となります。

［ **sockaddr_in** 構造体プロトタイプと **in_addr** 構造体プロトタイプ ］

```
struct  sockaddr_in {
    i16     sin_family;
    u16     sin_port;
    struct in_addr  sin_addr;
    char    sin_zero[8];
}
```

```
struct  in_addr {
    union {
        struct{char s_b1, s_b2, s_b3, s_b4;}S_un_b;
        struct{u16 s_w1, s_w2;}S_un_w;
        long    S_addr;
    }S_un;
};
#define  s_addr  S_un.S_addr
```

| メンバ | 説 明 |
|---|---|
| **sin_family** | アドレスファミリを指定（必ず AF_INET） |
| **sin_port** | ポート番号を指定 |
| **sin_addr** | **in_addr** 構造体。IP アドレスを 3 種類の共用体で表現している構造体。<br>**sendto()**の場合は **s_addr** を使って，接続相手の IP アドレスを指定する。 |
| **sin_zero** | 未使用領域 |

・ 第六パラメータ“ ***tolen*** ”は，第五パラメータ ***to*** で指定された **sockaddr_in** 構造体の大きさ（バイト）を指定します。

・ 第七パラメータ“ ***errp** ”は，戻り値が“ -1 ”（異常終了）だった場合のエラーコードが格納されます。ここに格納されるエラーは次の通りです。

| マクロ | | 作 用 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正なソケット記述子です<br>・RX-NET が初期化されていません |
| EWOULDBLOCK | 0x50 | ソケットがノンブロッキング・モードです |
| ENETDOWN | 0x5e | ネットワークがダウンしています |
| ENOBUFS | 0x63 | 送信するメッセージのサイズが大きすぎます |
| ENETCONN | 0x65 | 接続が確立していません |

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ― | -1 以外 | 正常終了。送信されたバイト数が格納されます。 |
| EINVAL | -1 | 異常終了。対応するエラーコードが ***errp*** で指定された領域に格納されます。 |

注意1　send を発行したとき，対象ソケットが“ ノンブロッキング・モード ”でかつ“ メッセージ送信処理の実行条件が整っていなかった ”場合，メッセージ送信処理は行わず，戻り値として“ EINVAL ”を，エラーコードとして“ EWOULDBLOCK ”を返します。

注意2　対象ソケットが“ データグラム型（SOCK_DGRAM） ”であった場合，送信可能なメッセージ・サイズは“ 0x1000（4096）バイト ”です。このため，メッセージを格納した ***buf*** のサイズ“ ***buf_len*** ”に 0x1000 以上の値を指定した場合には，メッセージの送信は行わず，戻り値として“ EINVAL ”を，エラーコードとして“ ENOBUFS ”を返します。

### 6.10.4 recvfrom

recvfrom()プロトタイプ

```
int recvfrom ( int sd, char *buf, int buf_len, int flags, saddr *from, int *fromlen, int *errp );
```

データを受信します。一度にすべてのデータが受信できるとは限りません。戻り値に受信したバイト数が返されるので，すべてが受信されるまで何度も recvfrom する必要があります。

- 第一パラメータ“ *sd* ”は，ソケット記述子を指定します。
- 第二パラメータ“ *buf* ”は，メッセージを格納するバッファ（受信バッファ）ポインタを指定します。
- 第三パラメータ“ *buf_len* ”は，第二パラメータ *buf* で指定した受信バッファのサイズ（バイト）を指定します。
- 第四パラメータ“ *flags* ”は，データの受信方法を指定します。

| 送信制御フラグ | 値 | 説 明 |
|---|---|---|
| MSG_PEEK | 0x2 | データを受信。読み出し位置の変更はなく，続く recv で同じデータを参照できます。 |
| MSG_URGENT | 0x800 | 緊急データを受信する |
| MSG_TRUNCATE | 0x1000 | 受信パケットの切り捨て |
| MSG_NONBLOCKING | 0x4000 | ノンブロッキング・モードで受信する |
| MSG_BLOCKING | 0x8000 | ブロッキング・モードで受信する |

備考1　MSG_PEEK を使うと，読み出し位置を変更することなく，そのデータを読み取ることができます。この場合，次の（MSG_PEEK を指定しない）recvfrom()呼び出しでそのデータを読み取り，読み出し位置が変更されることになります。

備考2　MSG_TRUNCATE を用いると，パケット内のデータが，recvfrom()で指定された長さに切り捨てられます。このフラグのみが，SOCK_DGRAM のようなパケット型プロトコルに有効です。MSG_TRUNCATE を指定しない場合は，複数の recvfrom()呼び出しで，パケット内のすべてのデータを読み出すことができます。

備考3　MSG_URGENT は緊急データを受信します。緊急データが存在しない場合，recvfrom()は“ -1 ”を返し，*errp* は ENOURGENTDATA に設定されることになります。緊急データとは，MSG_OOB フラグをオンにして送信されるデータのことをいいます。緊急データの引渡しに対して，どのような特別処理を行うかは，アプリケーションに任されています。緊急データは IST プロトコル，IDG プロトコルおよび TCP に対してだけ実装されています。ソケットにデータがなく，かつソケットがブロッキングモードにあると，受信コールはパケットが届くのを待ちます。しかしながらソケットがノンブロッキングモードにあるか，MSG_NONBLOCKING フラグが使われていた場合，-1 という値が直ちに戻され，*errp* が EWOULDBLOCK に設定されます。いつ残りのデータが届くかを調べるためには，nselect()を使うことができます。

・ 第五パラメータ“ ***from*** ”は，受信したパケットの発信元アドレスを格納する“ アドレス構造体へのポインタ ”を指定します。ここは **sockaddr_in** 構造体のポインタを渡します。ただし関数プロトタイプ宣言との矛盾を回避するために ***saddr**へのキャストが必要となります。

［ **sockaddr_in** 構造体プロトタイプと **in_addr** 構造体プロトタイプ ］

```
struct  sockaddr_in {
    i16     sin_family;
    u16     sin_port;
    struct in_addr  sin_addr;
    char    sin_zero[8];
}
```

```
struct  in_addr {
    union {
        struct{char s_b1, s_b2, s_b3, s_b4;}S_un_b;
        struct{u16 s_w1, s_w2;}S_un_w;
        long    S_addr;
    }S_un;
};
#define  s_addr  S_un.S_addr
```

| メンバ | 説 明 |
|---|---|
| **sin_family** | アドレスファミリを指定（必ず AF_INET） |
| **sin_port** | ポート番号を指定 |
| **sin_addr** | **in_addr** 構造体。IP アドレスを 3 種類の共用体で表現している構造体。<br>recvfrom()の場合は **s_addr** を使って，送信元の IP アドレスが格納される。 |
| **sin_zero** | 未使用領域 |

・ 第六パラメータ“ ***fromlen*** ”は，第五パラメータ ***from*** で指定された **sockaddr_in** 構造体の大きさ（バイト）を指定します。

・ 第七パラメータ“ ***errp** ”は，戻り値が“ -1 ”（異常終了）だった場合のエラーコードが格納されます。ここに格納されるエラーは次の通りです。

| マクロ | 値 | 作 用 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正なソケット記述子です<br>・RX-NET が初期化されていません |
| EWOULDBLOCK | 0x50 | ソケットがノンブロッキング・モードです |
| ENETCONN | 0x65 | 接続が確立していません |

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| － | -1 以外 | 正常終了。受信されたバイト数が格納されます。<br>なお 0x0 が返された場合は，メッセージの受信処理が完了する以前に対象ソケットの解放が行われたことを意味します。 |
| － | -1 | 異常終了。対応するエラーコードが ***errp*** で指定された領域に格納されます。 |

注意　recvfrom()を発行したとき，対象ソケットが“ ノンブロッキング・モード ”でかつ“ メッセージ受信処理の実行条件が整っていなかった ”場合，メッセージ受信処理は行わず，戻り値として“ EINVAL ”を，エラーコードとして“ EWOULDBLOCK ”を返します。

# 6. 11　ソケットの終了

ソケットを終了する場合は，正しい手順に従って行う必要があります。つまり 2 つのプログラム間で通信を行なっているため，データを送信している最中であったり，受信データが蓄積されている状態であったりします。このような状態のときに，片側のプログラムが一方的にソケットを破棄してしまうのは良くありません。

通常，クライアント/サーバ型のプログラムでは“クライアント側から”通信を終了します。サーバーから通信を終了するケースもありますが，このほとんどは強制的にサーバーからクライアントのコネクションを切断したいときです。

## 6.11.1　ソケットの終了（コネクション型の場合）

正しい手順でソケットの終了をするには，まず shutdown [注]を使用します。これはソケットに対して“送信の禁止”“受信の禁止”“送受信両方の禁止”をすることができます。これにより，クライアントからサーバーに対して send が禁止され，サーバーからのデータが，今溜まっているデータ以外はなくなります。その後，その溜まっているデータをクライアント側で recv します。これが終了すると，ソケットを終了（closesock）することができます。

なお，サーバープログラム，およびクライアントプログラムのソケットをクローズするまでは，ソケットのハンドルは有効です。ただし，ソケットに対して“送信の禁止”や“受信の禁止”をした後に，また connect するなどの通信を開始するようなことはできません。ソケットに対して送受信の禁止をした後は，必ずソケットを close します。送受信を禁止したソケットから connect することは，通常のシステムでは許可されていません。

注：shutdown()は LINGER TIMEOUT を使用することができません。タイムアウト処理を行いたい場合は shutdown()のあとに closesock()を発行し，closesock()の条件として LINGER TIMEOUT を使用してください。

## 6.11.2　ソケットの終了（コネクションレス型の場合）

コネクションレス型の場合，基本的に sendto と recvfrom の対で処理されているので，気づかないうちにデータが蓄積されていることはありません。したがって，コネクションレス型の場合は closesock するだけで，ソケットを終了できます。ただし bind 後のポートには受信データが蓄積されますので，コネクション型の場合と同様な処理をする方がよいです。

### 6.11.3 shutdown

shutdown()プロトタイプ

```
int   shutdown ( int sd,  int direction );
```

ソケットに対して，送信または受信を禁止状態にします。ソケットをクローズする前にデータの受信を停止して，すべてのデータをキューから取り出すために使用します。

- 第一パラメータ“***sd***”は，ソケットの識別子を指定します。
- 第二パラメータ“***direction***”は，次の３つの値で禁止状態を指定します。

| 値 | 作 用 |
|---|---|
| 0 | ソケットに対して受信を禁止します |
| 1 | ソケットに対して送信を禁止します |
| 2 | ソケットに対して送受信の両方を禁止します |

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です |
| EOPNOTSUPP | 0x5a | 対象ソケットのサービスタイプがSOCK_STREAMではありません |

### 6.11.4 closesock

closesock()プロトタイプ

```
int   closesock ( int sd );
```

***sd*** で指定されるソケットをクローズします。クローズすることによって，そのソケット用のすべてのリソースが解放されます。

setsopt()で変更可能な SO_LINGER オプションの値によって closesock()の動作が変化します。詳しくは下記の“ 注意 ”を参照してください。

なお closesock()は std.h で次のようなマクロとして実装されています。

```
#define  closesock(fd)  so_close(kdev(fd))
```

・ 第一パラメータ“ ***sd*** ”は，ソケットの識別子を指定します。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正なソケット記述子です<br>・RX-NET が初期化されていません |
| EALREADY | 0x5a | 対象ソケットが生成されていません |
| ECONNABORTED | 0x5a | Linger タイムアウトで指定された時間が経過しました |

注意1　SENDQEMPTY_NOTIFYイベントを受信した場合，closesock()を使いたくなるかもしれません。TCPで送信キューが空になると，nselect()を使用中のソケットにSENDQEMPTY_NOTIFYイベントが送付されてきます。このことは，TCPの送信キューがさらにデータを受けられる状態にあるか，アプリケーションにデータがなくなれば，ソケットをクローズできるということを意味しています。この時点でどうするかは，アプリケーション次第です。

注意2　この関数が正常に終了すると，CLOSE_NOTIFYイベントが通知されます。CLOSE_NOTIFYは，ソケットが下位層によりクローズされ，もう使用できないことをアプリケーションに知らせます。この通知によってどのような処理を行うかはアプリケーション依存ですが，このソケットを使ってさらにデータを送受信することはできません。

注意3　アプリケーションが closesock()をコールした際に，LINGER オプションが使用可能で，LINGER タイムアウトがブロッキングソケット上で非ゼロに設定されていると，TCP の接続状態が CLOSED あるいは TIME_WAIT へ移行するまで，あるいは LINGER タイムアウトが時間切れになるまで，のいずれか早い方が生じるまで，RX-NET はブロック動作を継続することになります。TCP CLOSED あるいは TIME_WAIT への移行が先に生じると，RX-NET はソケットを解放し，プログラムの制御をユーザに返します。TCP 状態ベクトルはソケットから切り離され，TIME_WAIT 状態にある場合には，CLOSED 状態への移行が完了するまで留まります。LINGER タイムアウトが先に生じると，RX-NET は TCP コネクションをリセットし，状態ベクトルとソケットを解放し，プログラムの制御をユーザに戻します。SO_LINGER の詳細については“ 6.13.6　setsopt ”を参照してください。

注意4　ソケットがノンブロッキング状態の場合や，LINGER オプションが設定されていない場合には，ソケットはただちに解放されます。受信キューに受信されたデータがあると，TCP コネクションがリセットされ，状態ベクトルが解放されます。そうでない場合には，TCP の状態ベクトルがソケットから切り離され，TCP 自身で CLOSE 状態に移行することになります。SO_LINGER の詳細については“ 6.13.6 setsopt ”を参照してください。

注意5　LINGER オプションが設定されている場合でも，LINGER タイムアウト値がゼロに設定されると，ソケットはただちに解放されます。SO_LINGER の詳細については“ 6.13.6　setsopt ”を参照してください。

## 6. 12　ネットワーク・バイト・オーダー

バイト・オーダー（マルチバイトを使用する値のメモリへの格納方法），いわゆる“ リトルエンディアン ”と“ ビッグエンディアン ”について説明します。

コンピュータのアーキテクチャにより，マルチバイト（2 バイト以上を必要とする値）をメモリ上に格納する場合，格納方法が異なります。例えば“ 0x1234 ”をメモリに格納するとき，2 バイトの大きさが必要となります。ビッグエンディアンの場合 12 を上位バイト，34 を下位バイトに格納します。リトルエンディアンの場合 34 を上位バイトに，12 を下位バイトに格納します。Motorola 社，Sun Microsystems 社の SPARC プロセッサはビッグエンディアン，Intel や Compaq 社の Alpha，そして V850 シリーズ，VR シリーズはリトルエンディアンが採用されています。このそれぞれ違った方式を“ ホスト・バイト・オーダー ”といいます。

IP アドレスやポート番号などソケットの要素となりうる数値に関しては，それぞれのプラットホームのソケットが処理してくれるので，プログラム開発者はバイトオーダーを意識する必要はありません。しかし send や recv で送受信するデータに関しては，プログラムの責任になるため，プログラム開発にはこのバイト・オーダーを意識する必要があります。

問題となるのは，ネットワーク上で違うアーキテクチャのコンピュータ同士が通信した場合，このホスト・バイト・オーダーが異なると，送信側と受信側で同じ値にも関わらず，違う値を示してしまうことになります。

このような場合，通信ネットワーク上では「ネットワーク・バイト・オーダー」という決まりごとが作られています。つまりホスト・バイト・オーダーがビッグエンディアンであろうがリトルエンディアンであろうが「ネットワーク・バイト・オーダーに変換する」ためのコーディングをしておくと，ネットワーク・バイト・オーダーがビッグエンディアンであろうがリトルエンディアンであろうが関係なくなります。つまり移植性の高いプログラムを作成することができます。なおネットワークバイトオーダーはビッグエンディアンです。

RX-NET では，ホスト・バイト・オーダーからネットワーク・バイト・オーダーにするための関数，ネットワーク・バイト・オーダーからホスト・バイト・オーダーにするための 4 つの関数が用意されています。

表 6－1　バイト・オーダー変換関数

| 変換関数 | 意 味 | 説 明 |
|---|---|---|
| htons | Host-to-network-short | 16 ビット値をホスト・バイト・オーダーからネットワーク・バイト・オーダにする |
| htonl | Host-to-network-long | 32 ビット値をホスト・バイト・オーダーからネットワーク・バイト・オーダにする |
| ntohs | Network-to-host-short | 16 ビット値をネットワーク・バイト・オーダからホスト・バイト・オーダーにする |
| ntohl | Network-to-host-long | 32 ビット値をネットワーク・バイト・オーダからホスト・バイト・オーダーにする |

RX-NET では，これらの関数はマクロとして用意されています（nectools32 / inc850 / rxnet / bsd_in.h に定義されています）。次にこれらの関数の使い方を説明します。

### 6.12.1 htons

htons()プロトタイプ

```
short  htons ( short a );
```

ホストバイトオーダーの16ビットデータを,ネットワークバイトオーダーの16ビットデータに変換します。

・ 第一パラメータ“*a*”は，ホストバイトオーダーの16ビットデータを指定します。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| － | 任意 | ネットワーク・バイト・オーダーの16ビットデータ |

### 6.12.2 htonl

htonl()プロトタイプ

```
int  htonl ( int a );
```

ホストバイトオーダーの32ビットデータを,ネットワークバイトオーダーの32ビットデータに変換します。

・ 第一パラメータ“*a*”は，ホストバイトオーダーの32ビットデータを指定します。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| － | 任意 | ネットワーク・バイト・オーダーの32ビットデータ |

### 6.12.3 ntohs

ntohs()プロトタイプ

```
short  ntohs ( short a );
```

ネットワークバイトオーダーの16ビットデータを,ホストバイトオーダーの16ビットデータに変換します。

・ 第一パラメータ“*a*”は，ネットワークバイトオーダーの16ビットデータを指定します。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| － | 任意 | ホストバイトオーダーの16ビットデータ |

### 6.12.4 ntohl

ntohl()プロトタイプ

```
int  ntohl ( int a );
```

ネットワークバイトオーダーの32ビットデータを,ホストバイトオーダーの32ビットデータに変換します。

・ 第一パラメータ“*a*”は，ネットワークバイトオーダーの32ビットデータを指定します。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| － | 任意 | ホストバイトオーダーの32ビットデータ |

### 6.12.5 inet_addr

RX-NET では，ドット形式の IP 文字列をネットワーク・バイト・オーダーの数値に変換する関数として，“inet_addr()”を用意しています。

inet_addr()プロトタイプ

```
u32   inet_addr ( u8 *cp );
```

inet_addr()は，ドット形式 IP 文字列（例えば ”192.168.1.1”）をネットワーク・バイト・オーダーの数値に変換します。ドット形式の IP 文字列とは次のような形式のドットで区切られた数字の列です。

- 8 ビット値.8 ビット値.8 ビット値.8 ビット値
- 8 ビット値.8 ビット値.16 ビット値
- 8 ビット値.24 ビット値
- 32 ビット値

数字の部分には 8 進数（012 など），10 進数（12 など），16 進数（0x12 など）が使用できます。各フィールドのビット数で表せる数値の範囲を越える値を記述した場合，正しくない文字列として扱われます。

- 第一パラメータ“***cp***”は，ドット形式 IP 文字列の先頭アドレスを指定します。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| － | 非 0x0 | ネットワーク・バイト・オーダーの数値に変換された IP アドレス |
| － | 0xffffffff | IP アドレスとして正しくない文字列が ***cp*** に渡された場合 |

# 6. 13　その他のソケット API

その他に RX-NET で用意されているソケット API を説明します。

## 6.13.1　blocking

blocking()プロトタイプ

```
int   blocking ( int sd );
```

***sd*** で指定されたソケットをブロッキング・モードに設定します。ブロッキング・モードでは，すべての I/O 要求に対し，要求が（失敗も含めて）完了するまで，要求動作をブロックします。ディフォールトでは，ソケットはブロッキングモードで生成されます。

・ 第一パラメータ“***sd***”は，ブロッキング・モードにするソケット記述子を指定します。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正なソケット記述子です<br>・RX-NET が初期化されていません |
| ENETDOWN | 0x5e | ネットワークがダウンしています。または RX-NET が初期化されていません。 |

### 6.13.2 nonblocking

nonblocking()プロトタイプ

```
int   nonblocking ( int sd );
```

***sd*** で指定されたソケットをノンブロッキング・モードに設定します。ソケットがノンブロッキング・モードだった場合，完了を待たされる API（送信データ未到着時の recv 等。ただし nselect は除く）は，err（-1）の値と，（もしある場合は）エラーコード EWOULDBLOCK を持って直ちに戻ることになります。
なお，ソケット生成時は，ディフォールトはブロッキングモードで生成されます。

・ 第一パラメータ“ ***sd*** ”は，ノンブロッキング・モードにするソケット記述子を指定します。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正なソケット記述子です<br>・RX-NET が初期化されていません |

### 6.13.3 getpeername

getpeername()プロトタイプ

```
int   getpeername (int sd,  saddr *name, int name_len );
```

***sd*** に接続しているソケットのアドレスを取得します。この関数を呼び出すことによって，***name*** には接続相手のアドレスが格納されますが，その構造はアドレス・ファミリによって異なります。RX-NET は AF_INET だけをサポートしているので，***name*** として ***struct sockaddr_in*** へのポインタを指定してください。

- 第一パラメータ“***sd***”は，アドレスを取得したい接続相手のソケット記述子を指定します。
- 第二パラメータ“***name***”は，**sockaddr_in** 構造体のポインタを渡します。取得したアドレスを格納します。ただし関数プロトタイプ宣言との矛盾を回避するために，***saddr ***へのキャストが必要となります。

［ **sockaddr_in** 構造体プロトタイプと **in_addr** 構造体プロトタイプ ］

```
struct  sockaddr_in {
    i16     sin_family;
    u16     sin_port;
    struct in_addr  sin_addr;
    char    sin_zero[8];
}
```

```
struct  in_addr {
    union {
        struct{char s_b1, s_b2, s_b3, s_b4;}S_un_b;
        struct{u16 s_w1, s_w2;}S_un_w;
        long    S_addr;
    }S_un;
};
#define  s_addr  S_un.S_addr
```

| メンバ | 説 明 |
|---|---|
| **sin_family** | アドレスファミリが返却されます（必ず AF_INET） |
| **sin_port** | ポート番号が返却されます |
| **sin_addr** | **in_addr** 構造体。IP アドレスを 3 種類の共用体で表現している構造体。<br>**getpeername()**の場合は **s_addr** を使った IP アドレスが返却されます。 |
| **sin_zero** | 未使用領域 |

・ 第三パラメータ“ *name_len* ”は，第二パラメータ *name* で指定された **sockaddr_in** 構造体の大きさ（バイト）を指定します。*name* が指す領域の大きさを超える値を指定することもできますが，その場合には正常終了後，*name* に書き込まれる接続相手のアドレスの大きさを反映するように変更されます。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正なソケット記述子です<br>・*name* が NULL<br>・*name_len* が 0，または名前を格納するのに十分な大きさではない。<br>・**name* が 0<br>・RX-NET が初期化されていません |

### 6.13.4 getsockname

getsockname()プロトタイプ

```
int   getsockname (int sd,  saddr *name,  int name_len );
```

***sd*** で指定されたソケットのアドレスを取得します。この関数を呼び出すことによって，***name*** にはソケットのアドレスが格納されますが，その構造はアドレス・ファミリによって異なります。RX-NET は AF_INET だけをサポートしているので，***name*** として ***struct sockaddr_in*** へのポインタを指定してください。

- 第一パラメータ“***sd***”は，名前を取得したいソケット記述子を指定します。
- 第二パラメータ“***name***”は，取得した名前を格納するアドレス構造体へのポインタを指定します。ここは **sockaddr_in** 構造体のポインタを渡します。取得した名前を格納します。ただし関数プロトタイプ宣言との矛盾を回避するために，***saddr*** *へのキャストが必要となります。

［ **sockaddr_in** 構造体プロトタイプと **in_addr** 構造体プロトタイプ ］

```
struct  sockaddr_in {
    i16     sin_family;
    u16     sin_port;
    struct in_addr  sin_addr;
    char    sin_zero[8];
}
```

```
struct  in_addr {
    union {
        struct{char s_b1, s_b2, s_b3, s_b4;}S_un_b;
        struct{u16 s_w1, s_w2;}S_un_w;
        long    S_addr;
    }S_un;
};
#define  s_addr  S_un.S_addr
```

| メンバ | 説 明 |
|---|---|
| **sin_family** | アドレスファミリが返却されます（必ず AF_INET） |
| **sin_port** | ポート番号が返却されます |
| **sin_addr** | **in_addr** 構造体。IP アドレスを 3 種類の共用体で表現している構造体。<br>**getsockname()**の場合は **s_addr** を使った IP アドレスが返却されます。 |
| **sin_zero** | 未使用領域 |

・ 第三パラメータ“*name_len*”は，第二パラメータ *name* で指定される **sockaddr_in** 構造体の大きさを指定します。*name* が指す領域の大きさを超える値を指定することもできますが，その場合には正常終了後，*name* に書き込まれる接続相手のアドレスの大きさを反映するように変更されます。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正なソケット記述子です<br>・*name* が NULL<br>・*name_len* が 0，または名前を格納するのに十分な大きさではない。<br>・**name* が 0<br>・RX-NET が初期化されていません |

### 6.13.5 getsopt

getsopt()プロトタイプ

```
int  getsopt ( int sd, int level, int optname, char *optval, int *optlen );
```

***sd*** で指定されたソケットのソケット・オプションの現在値を検索します。この API と setsopt()を使用することによって，特定ソケットに対するオプション操作が可能となります。TCP/IP では，基本の IP に対してオプションが設定されますので，この関数は TCP/IP にはきわめて有効です。

- 第一パラメータ“***sd***”は，ソケット記述子を指定します。ここで指定できるのは，IP を使用しており，かつ現在オープン中の TCP，UDP あるいはその他のプロトコル・ソケット用のファイル記述子です。
- 第二パラメータ“***level***”は，オプションが対象としているネットワーク層を指定します。ここで指定できるのは，次のものです。

| マクロ | 値 | 説 明 |
|---|---|---|
| IP_IP | 0x0 | IP プロトコルレベル |
| SOL_SOCKET | 0xffff | ソケットインタフェースレベル |

- 第三パラメータ“***optname***”は，検索するソケットオプションの種類を指定します。第二パラメータの“***level***”の値によって指定可能なオプションが異なります。

【 ***level*** を **IP_IP** に設定する場合，】

| 送信制御フラグ | 値 | 説 明 |
|---|---|---|
| IP_O_TOS | 0x1 | サービスのタイプを決定します |
| IP_O_FRAG | 0x2 | 断片化制御のタイプ（もしあれば）を決定します |
| IP_O_MAXTTL | 0x3 | 最大生存時間（秒）の設定値を決定します |
| IP_O_MINTTL | 0x4 | 最小生存時間（秒）の設定値を決定します |
| IP_O_SECURE | 0x5 | セキュリティ・オプションを決定します |
| IP_O_LSRR | 0x6 | Loose Source とルートの記録を決定します |
| IP_O_SSRR | 0x7 | Strict Source とルートの記録を決定します |
| IP_O_RR | 0x8 | ルートの記録を決定します |
| IP_O_STREAM | 0x9 | ストリーム識別子を決定します |
| IP_O_TIME | 0xa | 現在使用中のインターネット・タイムスタンプを調査します |

【 *level* を SOL_SOCKET に設定する場合，】

| 送信制御フラグ | 値 | 説 明 |
|---|---|---|
| SO_REUSEADDR | 0x4 | バインド状態にあるローカル・アドレスの再使用が可能か不可能かを調査 |
| SO_KEEPALIVE | 0x8 | 接続されたソケット上で定期的なメッセージ伝送が可能か不可能かを調査 |
| SO_DONTROUTE | 0x10 | “ don’t route ”オプションが使用可能か不可能化を調査 |
| SO_LINGER | 0x80 | SO_LINGER オプションが ON か OFF かを調べ, ON であればアプリケーションが“ LINGER ”する予定の時間長（ユーザ・アプリケーションが closesock()を発行してからソケットが解放されるまでの時間）を調査 |
| SO_TOROUGHPUT | 0x100 | このオプションを設定すると正常な状況では，完全な MSS サイズのパケットのみを送出することになります。より多くのデータが最終的に設定される（あるいはコネクションがクローズされ，残りがフラッシュされる）ことが知られているストリーム転送（FTP のようなアプリケーション）で，この機能は有効です。 |
| SO_EXOEDITE | 0x200 | SO_EXPEDITE が使用可能か不可能かを調査 |

- 第四パラメータ“ *optval* ”は，獲得したオプションの値を格納するバッファへのポインタを指定します。第三パラメータの“ *optname* ”が SO_REUSEADDR， SO_KEEPALIVE， SO_DONTROUTE，SO_THROUGHPUT，SO_EXPEDITE の場合は，この“ *optval* ”にはソケットオプションが格納されます。その際の値のサイズは 4 バイト（32 ビット）ですので，第五パラメータの“ *optlen* ”には 4（マクロ：MAXOPTVALEN）を指定してください。第三パラメータの“ *optname* ”が上記以外だった場合には，“ *optname* ”の指定に従って戻される情報のタイプによって異なります。詳細は“ 6.13.6　setsopt ”を参照してください。

- 第五パラメータ“ *optlen* ”は，第四パラメータの“ *optval* ”でポイントされるバッファの大きさ（単位：バイト）を格納する領域のアドレスを指定します。リターン時，“ *optlen* ”には検索したオプション値の大きさを含んでいます。最大長は 32 ビットで，マクロ“ MAXOPTVALEN ”において指定されます。この値は“ *optval* ”でポイントされる情報と整合が取れていなければなりません。詳細は“ 6.13.6　setsopt ”を参照してください。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 作 用 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正なソケット記述子です<br>・***optval*** が NULL です<br>・***optlen*** が NULL です<br>・****optlen*** が 0 です<br>・RX-NET が初期化されていません |
| ENOPROTOOPT | 0x57 | プロトコルでサポートされていないオプションです |
| EAFNOSUPPORT | 0x5b | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正な ***level*** 指定です |
| ENETDOWN | 0x5e | ネットワークがダウンしています |
| ENETCONN | 0x65 | 接続が確立していません |

以下に getsopt()の使用例を示します。

例1　オープンしている特定のソケットに対する REUSEADDR オプションの取得

```
/* オープンしている特定のソケットに対するREUSEADDRオプションの取得 */
/* 記述子が“sd”であるソケット                                    */

int  optval;
int  optlen;
optlen = sizeof(optval);

getsopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *)&optval, &optlen);
optlen = sizeof(optval);
if(optval & SO_REUSEADDR){

/* SO_REUSEADDR が設定されている */

} else {

/* SO_REUSEADDR が設定されていない */

}
```

例2　オープンしている特定のソケットに対する REUSEADDR オプションの取得

```
/* オープンしている特定のソケットに対するLINGERオプションの取得 */
/* 記述子が“sd”であるソケット                                  */

lingeropt linger;
int lingerlen;
lingerlen = sizeof(lingeropt);

getsopt(sd, SOL_SOCKET, SO_LINGER, (char *)&lingers, &lingerlen);
```

### 6.13.6 setsopt

setsopt()プロトタイプ

```
int  setsopt ( int sd, int level, int optname, char *optval, int optlen );
```

***sd*** で指定されたソケットのソケット・オプションを設定します。この setsopt()と getsopt()を使用することによって，特定ソケットに対するオプションに操作することが可能となります。TCP/IP では，基本の IP に対してオプションが設定されますので，この関数は TCP/IP にはきわめて有効です。

setsopt()によって変更された値は，次に変更されるまでその値のままとなります。

- 第一パラメータ“ ***sd*** ”は，ソケット記述子を指定します。ここで指定できるのは，IP を使用しており，かつ現在オープン中の TCP，UDP あるいはその他のプロトコル・ソケット用のファイル記述子です。
- 第二パラメータ“ ***level*** ”は，オプションが対象としているネットワーク層を指定します。ここで指定できるのは，次のものです。

| マクロ | 値 | 説 明 |
|---|---|---|
| IP_IP | 0x0 | IP プロトコルレベル |
| SOL_SOCKET | 0xffff | ソケットインタフェースレベル |

- 第三パラメータ“ ***optname*** ”は，ソケット上で実行される操作の種類を指定します。このパラメータはレベル固有です。つまり第二パラメータの“ ***level*** ”の値によって指定可能なオプションが異なります。

【 ***level*** を **IP_IP** に設定する場合，】

| 送信制御フラグ | 値 | 説 明 |
|---|---|---|
| IP_O_TOS | 0x1 | サービスのタイプを定義します。このオプションでは，setsopt()はコネクションが確立される前にコールされなければなりません。1つでもコネクションが確立してからコールされると最初に設定された値は折衝設定値と同じか，それより大きい値でなければなりません。 |
| IP_O_FRAG | 0x2 | 断片化制御を定義します。詳細については後述を参照してください。 |
| IP_O_MAXTTL | 0x3 | 最大生存時間のオプションを定義します。詳細については後述を参照してください。 |
| IP_O_MINTTL | 0x4 | 最小生存時間のオプションを定義します。詳細については後述を参照してください。 |
| IP_O_SECURE | 0x5 | IP により生成されるすべてのパケットに挿入して利用されるセキュリティ・オプションを指定します。 |

| 送信制御フラグ | 値 | 説 明 |
|---|---|---|
| IP_O_LSRR | 0x6 | Loose Source and Record Route オプションを指定します。Loose Source Routing を行うことでゲートウエイルートに沿って IP モジュールで利用されるルーティング情報を供給することができます。ゲートウエイは，可能な場合，このリストで指定された次のホストへパケットを向かわせます。可能でない場合は，通常の方法でルートを決定します。Loose Source ルートリストは，パケットが通過する予定の中間ゲートウエイを識別する１つあるいは複数のインターネットアドレスから構成されています。 |
| IP_O_SSRR | 0x7 | Strict Source and Record Route オプションを指定します。Strict Source Routing を使うと，ゲートウエイルートに沿って通過すべき IP モジュールの正確な設定を指定することができます。ゲートウエイは，可能な場合，このリストで指定された次のホストにパケットを向かわせます。可能でない場合は失敗となり，ICMP エラーが戻されます。 |
| IP_O_RR | 0x8 | Record Route オプションを指定します。Record Routing は IP_O_LSRR と同じです。違いは，ルーティングについての制限がないことです。Record Route オプションを使うことで，データグラムのゲートウエイルートを記録する方法が得られることになります。`setsopt()`を介して渡される配列により，このオプションでパケットが利用できるスロットの数が決まります。 |
| IP_O_STREAM | 0x9 | Stream Identifier（ストリーム識別子）を指定します |
| IP_O_TIME | 0xa | Internet Timestamp（現在使用中のインターネット・タイムスタンプ）を指定します。 |

【 ***level*** を **SOL_SOCKET** に設定する場合，】

| 送信制御フラグ | 値 | 説 明 |
|---|---|---|
| SO_REUSEADDR | 0x4 | バインド状態にあるローカル・アドレスの再使用を可能にします。このオプションを無効にする場合には~SO_REUSEADDR を指定してください。 |
| SO_KEEPALIVE | 0x8 | 接続されたソケット上で定期的なメッセージ送信を可能にします。コネクションをベースとするほとんどのプロトコルが，ディフォールトではこのモードになります。このオプションを無効にするには~SO_KEEPALIVE を指定してください。 |
| SO_DONTROUTE | 0x10 | 出力されるデータが，下位プロトコルによる通常の経路制御機構を迂回することを指定します。データは宛て先アドレスのネットワーク部のインタフェース・アドレスに直接送信されるようになります。通常の経路制御を使用したい場合には，~SO_DONTROUTE を指定します。 |

| | | |
|---|---|---|
| SO_LINGER | 0x80 | SO_LINGER オプションが ON か OFF かを調べ, ON であればアプリケーションが“ linger ”する予定の時間長（ユーザアプリケーションが closesock()を発行してからソケットが解放されるまでの時間）を調査 |
| SO_TOROUGHPUT | 0x100 | スループットを増やしたり減らしたりします。正常な状態でこのオプションを設定すると，完全な MSS サイズのパケットを送出することになります。より多くのデータが最終的に送信される（あるいはコネクションがクローズされ，残りが捨てられる）ことが知られているストリーム転送（FTP のようなアプリケーション）ではこの機能は有効です。このオプションは SO_EXPEDITE と排他です。このオプションを無効にする場合には~SO_EXPEDITE を指定してください。 |
| SO_EXPEDITE | 0x200 | 上位層プロトコルでの引渡しと同様に，すべてのデータは大きさに関係なく直ちに送出されます。このオプションは，ユーザが結果をすぐ希望する TELNET のようなアプリケーションに有効です。このオプションは SO_THROUGHPUT と排他です。このオプションを無効にする場合には~SO_EXPEDITE を指定してください。 |

- 第四パラメータ“ ***optval*** ”は，実行中の操作によって異なります。

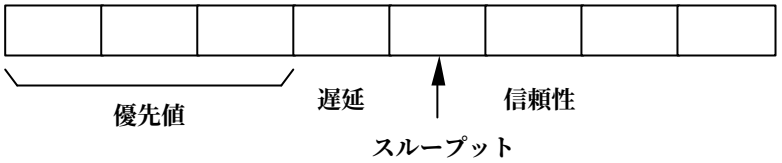
1. ***optname*** が SO_REUSEADDR， SO_KEEPALIVE， SO_DONTROUTE， SO_THROUGHPUT，SO_EXPEDITE の場合“ ***optval*** ”は無視されます

2. ***optname*** が SO_LINGER の場合“ ***optval*** ”は下記のフォーマットのデータ構造体をポイントします。

```
lingeropt  lingers;
lingers.linger_on = 1;        /* 1 は ON を，0 は OFF を意味します */
lingers.linger_timeout = 1;  /* 1 秒 */
```

linger_on を 0 または 1 に設定してください。“ 1 ”はこのオプションが使用可能，“ 0 ”はこのオプションが使用不可ということを意味します。
ユーザのアプリケーションがどのくらい長く（ユーザのアプリケーションが closesock()を発行してから，ソケットが解放されるまでの時間）“ linger ”しておきたいかを指定するには, linger_timeout を設定してください。ディフォールトでは，オープンされたソケットはブロッキングモードにあり，かつ LINGER オプションは ON で，LINGER タイムアウトは 60 秒に設定されています。このオプションは，ソケットのクローズの方法にも影響を及ぼします。詳細は“ 6.11.4　closesock ”をご参照ください。

3. ***optname*** が IP_O_TOS の場合“***optval***”は IP パケットに対するサービスの質を定義します。“***optval***”はソケットで生成されるどの IP パケットにも存在する所定のフィールドに挿入される情報を含んでいるバイトに対するポインタです。***optlen*** が“0”の場合，ディフォールトのサービスタイプが選択されます（ルーチンの優先，および通常の遅延，スループットならびに信頼性）。***optlen*** が“0”でない場合は，***optval*** でポイントされるバイトは1バイトで，4つのフィールドからなります。




| マクロ名 | 値 | 意 味 |
|---|---|---|
| IP_PRECEDENCE | 0xe0 | 優先度のマスクビットパターン |
| IP_DELAY | 0x10 | 遅延 |
| IP_THROUGHPUT | 0x8 | スループット |
| IP_RELIABLE | 0x4 | 信頼性 |

優先値は IP パケットが引き渡される順番を示しています。高い優先値（例えばネットワーク制御が最高の優先値を持っています）を有するパケットが，低い優先値を有するパケットより先に引き渡されます。

| マクロ名 | 値 | 意 味 |
|---|---|---|
| IP_NC | 0xe0 | ネットワーク制御 |
| IP_IC | 0xc0 | インターネットワーク制御 |
| IP_CRITIC | 0xa0 | CRITIC / ECP |
| IP_FLASH | 0x80 | フラッシュ |
| IP_FLSHO | 0x60 | フラッシュを無視 |
| IP_IMMED | 0x40 | 即時 |
| IP_PRIORITY | 0x2 | 優先度 |
| IP_ROUTINE | 0x0 | ルーチン |

遅延，スループット，信頼性は二進数で表されます。“0”は通常のケースを意味します。遅延値“1”は，遅延量を小さくしなければならないことを意味します（通常は遅延してはいけないことです）。スループット値“1”は，高いスループットの要求を意味しています。また，信頼性の値“1”は，高信頼性の要求を意味しています。
すべてのサブネット（リンク層）が，すべての伝送サービスをサポートしているわけではありません。しかし，配送パスにある各サブネットプロトコルは，サブネットで利用可能なサービスを要求されているサービスの質に，できるだけ近づくように動作します。

4. ***optname*** が IP_O_FLAG の場合“ ***optval*** ”は“ 0 ”か“ 1 ”を示すバイトに対するポインタとなります。“ 0 ”の場合，断片化制御は OFF になります（断片化が発生します）。“ 1 ”の場合，断片化制御は ON になります。***optlen*** が“ 0 ”でない場合，モードは ***optval*** でポイントされる値によって選択されます。それ以外の場合は，ディフォールトの制御モードが選択されることになります（断片化が許されます）。ディフォールトでは断片化が許されます。

5. ***optname*** が IP_O_MAXTTL の場合“ ***optval*** ”は MAXTTL（最大生存期間）として利用され，新しい値を含むバイトに対するポインタとなります。***optlen*** が“ 0 ”でない場合，この新しい値は，***optval*** でポイントされるバイトによって設定されます。そうでない場合，ディフォールトの MAXTTL 値が選択されます。ディフォールトの MAXTTL 値は構成依存です。

6. ***optname*** が IP_O_MINTTL の場合“ ***optval*** ”は MINTTL（最小生存期間）として利用される新しい値を含むバイトに対するポインタとなります。***optlen*** が“ 0 ”でない場合，この新しい値は，***optval*** でポイントされるバイトによって設定されます。そうでない場合，ディフォールトの MINTTL 値が選択されます。ディフォールトの MINTTL 値は構成依存です。***optval*** でポイントされる値は“ 1 秒（ディフォールト値）以上 255 秒以下の値 ”をとることができます。この値が IP_O_MAXTTL による設定値より大きい場合，この値は常に MAXTTL が MINTTL よりも大きいか，同じになるように調整されます。

7. ***optname*** が IP_O_SECURE の場合，セキュリティ・オプションを指定します。Security はセキュリティ，コンパートメント，処理制限および伝送制御コートの 4 フィールドで成り立っています。ディフォールトは“ オプションなし ”です。このオプションは，国防省のホストがインターネットを使って，標準方式で機密データを送信するために必要とされています。***optval*** は socket.h で定義されている下記の構造体に対するポインタを受け入れます。

```
typedef struct
{
     a16  ipos_security;          /* セキュリティ */
     a16  ipos_compartments;      /* コンパートメント */
     a16  ipos_handling;          /* 処理制限 */
     a32  ipos_tcc;               /* 伝送制御コード */
} ipos_t;
```

ipos_tcc フィールドは，わずか 24 ビット長で，そのうち最下位バイトは使われておりません。***optlen*** がその値（***ipos_t***）です。***optlen*** が“ 0 ”の場合，セキュリティ・オプションは使用不可となります。なお，このオプションの詳細は RFC を参照してください。

8. ***optname*** が IP_O_LSRR の場合“ ***optval*** ”は losse route オプションで使われる loose route であるインターネット・アドレスの配列をポイントします。***optlen*** はこの構造体の大きさ（バイト数）でなければなりません。***optlen*** が 0（ディフォールト値）の場合，すべての発信元ルーティングは不可となります（このことは Strict Source and Record Route オプションにも適用されます）。

9. ***optname*** が IP_O_SSRR の場合“ ***optval*** ”は strict source and record route で使われる loose route となるインターネット・アドレスの配列をポイントします。***optlen*** はこの構造体の大きさ（バイト数）でなければなりません。***optlen*** が 0（ディフォールト値）の場合，すべての発信元ルーティングは不可となります（このことは Strict Source and Record Route オプションにも適用されます）

10. ***optname*** が IP_O_RR（Record Route）の場合“ ***optval*** ”は record route オプションで使われる loose route となるインターネット・アドレスの配列をポイントします。***optlen*** はこの構造体の大きさ（バイト数）でなければなりません。***optlen*** が 0（ディフォールト値）の場合，すべての発信元ルーティングは不可となります（このことは Strict Source and Record Route オプションにも適用されます）

11. ***optname*** が IP_O_STREAM の場合“ ***optval*** ”はストリーム識別子を含む 2 バイトからなるフィールドに対するポインタです。***optlen*** は 2 か 0 でなければなりません。***optlen*** が 0（ディフォールト値)の場合,ストリームオプションは使用不可となります。16 ビットのストリーム識別子は SATNET のようなストリーム概念をサポートしているサブネットで使われます。

12. ***optname*** が IP_O_TIME の場合“ ***optval*** ”はインターネット・タイムスタンプ・オプションを指定します。このオプションを使うと，データグラムが宛て先に向かってインターネット上を伝送されていく中で，時間情報を収集できるようになります。このオプションには，3 つの基本的な形式があります。

| | |
|---|---|
| タイムスタンプのみ | 各 IP モジュールに連続する 32 ビットワードのタイムスタンプが格納されている。 |
| タイムスタンプ／IP アドレスの一組 | 各 IP モジュールにインターネット・アドレスとタイムスタンプが格納されている。 |
| 前もって指定されている IP アドレス | インターネット・アドレスはユーザにより指定される。各 IP モジュールがデータグラムを受信すると，そのモジュールがリスト上で次に指定されている場合には，データグラムにタイムスタンプが記録される。 |

***optval*** は socket.h で定義されている下記の構造体に対するポインタを受け入れます。

```
typedef struct ipot_t
{
   a16  ipot_overflow_type;      /* セキュリティ */
   a16  ipot_timestamps[9];      /* コンパートメント */
} ipot_t;
```

ipot_overflow_type は IPO_T_TO（タイムスタンプだけ），IPO_T_IT（タイムスタンプ／IP アドレスの一組），あるいは IPO_T_PIT（前もって指定されている IP アドレス）のいずれかです。ipot_timestamps はインターネット・アドレスあるいはインターネット・タイムスタンプの配列です。インターネット・タイムスタンプ・オプションにおけるビットは下記のようになっています。すなわち，IPO_T_OVER（オーバーフロー・マスク），IPO_T_TYPE（タイムスタンプ・タイプ・マスク），IPO_T_TO（タイムスタンプのみ），IPO_T_IT（インターネット・アドレスとタイムスタンプとの一組）あるいは，IPO_T_PIT（前もって指定された IP アドレス一組）です。***optlen*** が 0（ディフォールト）の場合，タイムスタンプ・オプションは使用不可となります。そうでない場合，***optlen*** は 2 から 38 までの数を取ります。この値はタイムスタンプ・オプション用に割り当てられるスロットの数となります。計算は次のようになります。

```
optlen = (4 * number_of_slots) + 2
```

・ 第五パラメータ“ ***optlen*** ”は，第四パラメータの“ ***optval*** ”でポイントされるバッファの大きさ（単位：バイト）を指定します。

この関数の戻り値は以下のようになります。

| マクロ | 値 | 作 用 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正なソケット記述子です<br>・不正な ***level*** 指定です<br>・***optval*** が NULL です<br>・***optlen** が 0 です<br>・RX-NET が初期化されていません |
| EPROTOTYPE | 0x56 | ソケットに違うタイプのプロトコルが使われています |
| ENOPROTOOPT | 0x57 | プロトコルでサポートされていないオプションです |

以下に setsopt()の使用例を示します。

例　記述子が“ *sd* ”であるソケット上で LINGER タイムアウト値を 1 秒に変更する

```
/* 記述子が“ sd ”であるソケット上で          */
/* LINGERタイムアウト値を1秒に変更するには  */
/* このプログラムシーケンスを使用する        */

lingeropt linger;
lingers.linger_on = 1;                /* 1はONを，0はOFFを意味する */
lingers.linger_timeout = 1;   /* 1秒 */
setsopt(sd,SOL_SOCKET,SO_LINGER,(char *)&lingers, sizeof(lingeropt));
/* LINGER を OFF にするには，下記のシーケンスを使用する */
lingeropt linger;
lingers.linger_on = 0;        /* 1はONを，0はOFFを意味する */
setsopt(sd, SOL_SOCKET, SO_LINGER, (char *)&lingers, sizeof(lingeropt));
```

注意1　断片化について

IP には，大きなデータグラムを小さなデータグラムに分割するための断片化機構が用意されています。サブネットごとに容量が違うことから生じる問題に対するこの解決法は，データグラムの大きさを小さくして，インターネット上のどのサブネットでも受け入れられるようにするよりも優れた柔軟性を提供しています。断片化を無視するには，このソケットオプションをご利用ください。TCP コネクションは適正サイズのパケットを生成しますので，フラグメントは通常生じません。しかしながら，物理リンク層には大きすぎる UDP パケットは断片化される場合があり，途中にあるゲートウエイによっては，パケットの断片化もあり得ます。断片化制御が ON の場合，生成されたパケットは断片化されません。パケットを断片化しなければならない状況があるとエラーとなります。

注意2　生存時間について

IP にはデータグラムの生存時間を制御する機構が備わっています。この機構を用いることにより，ルートが迂回路を指定したとき，データグラムがずっとループしていたり，宛て先に大幅に遅れて到着したりするような問題を回避することができます。ユーザは，このオプションを使い，プロトコルで生成される生存時間フィールトの生存時間の最大バウンドを指定することができます。この値は秒単位で 1 秒以上 255 秒（4 分 15 秒）以下の数値を取ることができます。この値が IP_O_MINTTL オプションによる設定値より小さい場合は，この値は MAXTTL が常に MINTTL より大きくなるか同じになるように調整されます。

### 6.13.7 nselect

nselect()プロトタイプ

```
int  nselect (struct sel *selp, int cnt, u32 *waitp,
              int (*pfi)(struct sel *, int, u32, void *), void *arg, int *errp );
```

複数のソケット上で指定されたイベントを待ちます。*selp* で指定された選択構造体 **sel** のメンバ **se_inflags** に設定されているイベント（要求イベントの論理和）のうち，どのイベントが発生していたのかを調べ，実際に発生していたイベント（発生イベントの論理和）がメンバ **se_outflags** に格納されます。

ただし，この関数を発行した際，メンバ **se_inflags** に設定されているイベントが１つも発生していなかった場合には，要求動作の実行が中断されます。

なお，要求動作の実行再開は，メンバ **se_inflags** に設定されているイベントが１つでも発生した際，または *waitp* で指定された待ち時間が経過した際に行われます。

- 第一パラメータ“ *selp* ”は，**sel** 構造体の配列に対するポインタです。この構造体における **se_inflags** の値は，イベントに組み合わせに対して設定されなければなりません。関数から戻ると，どんなイベントが発生したかが分かるように **se_outflags** が変更されることになります。

［ **sel** 構造体プロトタイプ ］

```
typedef struct  sel {
    u16     se_inflags;
    u16     se_outflags;
    i16     se_fd;
    i16     se_1reserved;
    u32     se_user;
    u32     se_2reserved;
} sel;
```

**se_outflags** に設定される値は次のとおりです。

| イベント | 値 | 内 容 | TCP | UDP |
|---|---|---|---|---|
| READ_NOTIFY | 0x1 | データ読み取りが可能な状態にあります。READ_NOTIFY が発生すると，受信キューにある全てのデータを読み出さないと nselect()を使ってもう一つREAD_NOTIFYを登録できるようにはなりません。 | ○ | ○ |
| WRITE_NOTIFY | 0x2 | データ書き込み領域がまだあります。WRITE_NOTIFY は送信キューにまだ空きが残っている場合に発生します。 | ○ | ○ |
| ACCEPT_NOTIFY | 0x4 | 受動コネクションが確立されました。 | ○ | |
| CLOSE_NOTIFY | 0x8 | 終了処理が完了しました。このイベントはclosesock()操作が正常終了した際に設定されます。ソケットの終了処理が完了すると，非同期ポートの整理が強制的に行われます。 | ○ | ○ |
| CONNECT_NOTIFY | 0x10 | 能動コネクションが確立しました。 | ○ | |
| EXCEPT_NOTIFY | 0x20 | 例外事項が発生しました。 | ○ | ○ |
| RSHUTDOWN_NOTIFY | 0x40 | 受信方向が相手から遮断されています。下記の条件が正しい場合に設定されます。<br>・相手が shutdown()により接続を遮断<br>・受信キューが空である | ○ | ○ |
| TIMEOUT_NOTIFY | 0x80 | イベントがタイムアウトになっています。 | ○ | |
| WSHUTDOWN_NOTIFY | 0x100 | 書き込み方向が相手から遮断されています。 | ○ | |
| URGENT_NOTIFY | 0x200 | 緊急データが用意されています。 | ○ | |
| SENDQFULL_NOTIFY | 0x1000 | 送信キューに空きがありません。 | ○ | ○ |
| SENDQEMPTY_NOTIFY | 0x2000 | 送信キューが空です。このイベントが発生したことが通知されると，アプリケーションがclosesock()をコールしようとする可能性があります。 | ○ | ○ |

- 第ニパラメータ“ ***cnt*** ”は，***selp*** にある選択構造体の数です。***cnt*** は選択するソケットの数にも対応していなければなりません。
- 第三パラメータ“ ***waitp*** ”は，応答を待つ時間長（ミリ秒）へのポインタです。***waitp*** の指す内容が“ -1 ”だった場合や，ポインタ自身が NULL の場合には，nselect()は，たとえノンブロッキングモードなソケットだったとしても待ちつづけます。時間長が正の整数値の場合，nselect()はある期間（ ****waitp*** ミリ秒）経過後戻ります。終了後 ***waitp*** の時間長の値は，時間切れの前の残り時間に変更されます。
- 第四パラメータ“ ***pfi*** ”は，現状では***pfi*** をゼロポインタ（usel_nilpfi）に設定する必要があります。usel_nilpfi は select.h で定義されています。
- 第五パラメータ“ ***arg*** ”は，予約されています。

・ 第六パラメータ“ ***errp*** ”は，エラーコードを格納する整数値に対するポインタです。戻り値が“ -1 ”（異常終了）だった場合のエラーコードが格納されます。ここに格納されるエラーは次の通りです。

| マクロ | 値 | 作 用 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | 無効な引数です。以下のような原因が考えられます。<br>・***selp*** が NULL です<br>・***cnt*** が 0 です<br>・***cnt*** が-1 です |
| ENETDOWN | 0x5e | ネットワークがダウンしています。または RX-NET が初期化されていません。 |

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| ― | -1, 0 以外 | 正常終了。イベントが発生したソケット数が返却されます。 |
| ― | 0 | 選択したイベントが発生する以前にタイムアウトしました。 |
| ERR | -1 | 異常終了。対応するエラーコードが ***errp*** で指定された領域に格納されます。 |

例

```
sel  sel_arr[3], *sep;
char buf[1024];
int  nfound, cnt, bytes, err;
:
sel_arr[0].se_fd = fd0;
sel_arr[0].se_inflags = READ_NOTIFY;
sel_arr[1].se_fd = fd1;
sel_arr[1].se_inflags = READ_NOTIFY|URGENT_NOTIFY;
sel_arr[2].se_fd = fd2;
sel_arr[2].se_inflags = READ_NOTIFY;

nfound = nselect((sel *)sel_arr, 3, (u32 *)0, usel_nilpfi, (u32)0, &err);
if(nfound == ERR){
return(err);
}
```

注意1　SENDQEMPTY_NOTIFY イベントは，TCP の送信キューが空になった際に nselect()を使ってソケットに送られます。このことは，TCP の送信キューが，さらにデータを収容できるということを，あるいはアプリケーションにデータが残っていなければ，ソケットをクローズすることができることを意味しています。この時点でどうするかはアプリケーション次第です。

注意2　CLOSE_NOTIFY は，ソケットが下位層によりクローズされ，もう使用できないことをアプリケーションに知らせます。この時点で何をするかはアプリケーション次第ですが，このソケットを使ってさらにデータを送ることはできません。非同期ポート上の closesock()コールに応じた CLOSE_NOTIFY が，すべてのデータがソケットで受け取られ，最後の TCP ハンドシェークが完了した後に，送付されてきていた可能性もあります。あるいは接続を遮断中のもう一方の側から突然届く可能性もあります。アプリケーションには障害許容力が必要で，さらに予想した範囲だけではなく，可能性のある全ての NOTIFY とエラーとをチェックしてみる必要があります。

### 6.13.8 reject

reject()プロトタイプ

```
int  reject ( int sd );
```

新しく確立されたコネクションを拒否するべきであるとプロトコルに知らせます。具体的には ***sd*** で指定されたソケットの，待機ソケットキューの先頭にキューイングされている接続要求を削除（接続の確立拒否）し，コネクションをクローズします。

なお，この関数では削除要求のキューイングが行われません。このため，この関数を発行した際，対象ソケットの待機ソケットキューに接続要求がキューイングされていなかった場合には，戻り値として EINVAL を返します。

・ 第一パラメータ“ ***sd*** ”は，待機ソケットのソケット記述子を指定します。

| マクロ | 値 | 説 明 |
|---|---|---|
| ENOERR | 0x0 | 正常終了 |
| EINVAL | 0x16 | パラメータの指定が不正です。次のような原因が考えられます。<br>・不正なソケット記述子です<br>・接続待機ソケットがありません |

# 6. 14　BSD の Socket-API との差異について

RX-NET の API と BSD の Socket-API の差異について以下に説明します。

UNIX ソケット・インタフェースは，一般的に BSD ソケットといわれるインタフェースを用いていますが，その実装の詳細についてはシステムによって多少の違いがあると思われます。以下では Solaris 2.5.1 と RX-NET V1.20 の違いについて記述します。ただし，他のシステム（4.4BSD など）では異なる可能性があります。

## 6.14.1　全体における変更点

- UNIX でのプロトコルファミリは、RX-NET ではアドレスファミリとして扱います。例えば `socket()`の第一パラメータは以下のように異なります。

  ・UNIX　→　PF_INET
  ・RX-NET　→　AF_INET

  これはアドレス構造体（`saddr / struct sockaddr`）における第一フィールドも同様です。

- UNIX では通常 `struct sockaddr` を利用しますが，RX-NET では同構造体を `typedef` した `saddr` を利用します。
  両構造体は基本的に同じ形をしていますが，前述したように UNIX ではプロトコルファミリ(PF_)を指定するところに RX-NET ではアドレスファミリ(AF_)を指定します。
  なお，実際の使用にあたっては `struct sockaddr_in` 構造体を利用できる点は同じです。

- UNIX では送受信バッファのアドレス型として `void *` を利用していますが，RX-NET では `char *` を利用します。

- UNIX ではサイズ型は `size_t` や `socklen_t` 等の型を利用しますが，RX-NET では `int` を利用します。

### 6.14.2　各 API の差異

API の差異について，下記にまとめます。

| API 名 | 差 異 |
|---|---|
| accept | パラメータ ***errp*** を追加しました。RX-NET では ***errno*** 変数へのエラーコードの格納は行いませんので，***errp*** を利用してエラーコードを取得してください。 |
| bind | UNIX ソケットから変更はありません。 |
| blocking | API を新設しました。ソケットのブロッキングオプションを変更する場合には，この API または nonblocking をご利用ください。 |
| closesock | close から名前を変更しています。ソケットのクローズにはこの API をご利用ください。 |
| connect | UNIX ソケットから変更はありません。 |
| getpeername | UNIX ソケットから変更はありません。 |
| getsockname | UNIX ソケットから変更はありません。 |
| getsopt | getsockopt から名前を変更しています。また，利用できるオプションに違いがあります。 |
| listen | UNIX ソケットから変更はありません。 |
| nonblocking | API を新設しました。ソケットのブロッキングオプションを変更する場合には，この API または blocking をご利用ください。 |
| nselect | API を新設しました。複数ソケットの待ち受けを行うことができます。UNIX の select に相当しますが，イベント発生時にコールバック関数を呼び出すことはできません。<br>主にタイムアウト付きのポーリングを行うために利用します。 |
| recv | パラメータ ***errp*** を追加しました。RX-NET では ***errno*** 変数へのエラーコードの格納は行いませんので，***errp*** を利用してエラーコードを取得してください。 |
| recvfrom | パラメータ ***errp*** を追加しました。RX-NET では ***errno*** 変数へのエラーコードの格納は行いませんので，***errp*** を利用してエラーコードを取得してください。 |
| reject | API を新設しました。コネクト要求を拒否するために利用します。 |
| send | パラメータ ***errp*** を追加しました。RX-NET では ***errno*** 変数へのエラーコードの格納は行いませんので，***errp*** を利用してエラーコードを取得してください。 |
| sendto | パラメータ ***errp*** を追加しました。RX-NET では ***errno*** 変数へのエラーコードの格納は行いませんので，***errp*** を利用してエラーコードを取得してください。 |
| setsopt | setsockopt から名前を変更しています。また，利用できるオプションに違いがあります。 |
| shutdown | UNIX ソケットから変更はありません。 |
| socket | パラメータ ***errp*** を追加しました。RX-NET では ***errno*** 変数へのエラーコードの格納は行いませんので，***errp*** を利用してエラーコードを取得してください。 |

# 第7章　PPP

RX-NET（PPP）の説明にあたり，まず PPP についての説明をします。

## 7. 1　PPP（Point-to-Point Protocol）

PPP はシリアル接続でデータグラムを送るプロトコルです。異なるプロトコルのデータを運べるようにしたいという考えから開発されました。

PPP の基本的な役割は，2 点間で正しくデータの交換ができることを確認し（接続の確立），データをカプセル化して運ぶということです。その 2 点間では，お互いに IP で接続することができます。

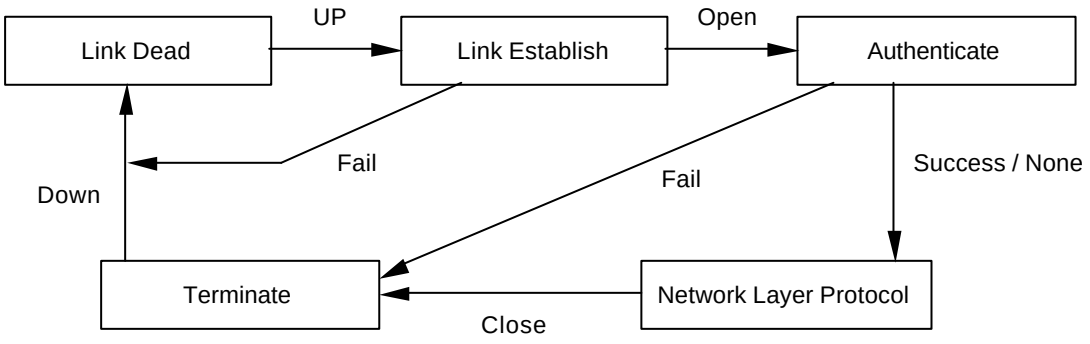
PPP には次のような特徴があります。

- HDLC（High level Data Link Control Procedure）とよく似たフレーム構成を採用し，非同期シリアル回線だけではなく，同期シリアル回線でも利用できる。
- 異なるプロトコルを取り扱うことができる。IP，IPX，DECnet，AppleTalk など，さまざまなプロトコルを単一リンクで運ぶことが可能。ただし，1 つのパケットに入るプロトコルは 1 つだけでなければならない。
- IP アドレス，MRU（Max Receive Unit：最大受信単位）などについて，送受信側相互間でやりとりする。

## 7. 2　PPP の状態フェーズ

PPP のリンクが確立し，それが開放されるまで，に示すようなフェーズの遷移が起こります。

図 7－1　PPP リンクの確立



各フェーズの意味は次の通りです。

- Link Dead
  この状態では PPP は休止しており，物理層からの準備完了信号を待っています。この信号には外部のイベントからの信号と，シリアルリンクから送られるデータキャリア検出信号があります。この時点では LCP は初期状態，あるいは起動中と呼ばれる状態にあり，リンク上にはいかなる活動もあり得ません。
  物理層がリンクの準備完了を知らせると，PPP は Link Establish 状態に移行します。

- Link Establish
  この状態では，コネクションの確立に LCP が使われ，接続された両端が設定パケットの交換と応答確認を行います。リンクが確立されると認証フェーズ（Authenticate）に移行します。

- Authenticate
  コネクションのどちらかの側が " 認証が有効 " になっていた場合，開設中状態の間に認証プロトコルの使用を要求します。PPP がサポートしている標準認証プロトコルには，パスワード認証プロトコル（PAP：Password Authentication Protocol）と，チェレンジハンドシェーク認証プロトコル（CHAP：Challenge Handshake Authentication Protocol）の 2 つがあります。他の独自プロトコルがありますが，両端が使用に合意しなければなりません。
  認証が成功した場合，または認証が要求されていない場合，PPP は Network Layer Protocol に移行します。

- Network Layer Protocol
  この状態では，要求されたそれぞれのネットワーク制御プロトコル（NCP）を使って，そのネットワーク層で動作するようにリンクを設定します。リンクが開いている間は，いつでも NCP を使うことができます。これによって新しいネットワークプロトコルをリンクに追加したり，不要になったものを削除したりすることが可能になっています。特定の NCP が開設状態に達すると，対応するネットワーク層プロトコルを PPP が運べるようになります。特定の NCP が失敗した場合，あるいは最初から開かれていない場合，そのプロトコルに対して送られたパケットは，全て拒絶パケットとして送信側に返送されます。

- Terminate
  終了パケットと応答確認の交換によってリンクが閉鎖される間の状態です。リンクの障害などのイベントがあると，いつでも終了中状態になります。リンク障害の場合は，もちろん終了応答確認は受信されないので，一定のタイムアウト後に終了するのが普通です。ただし通常の終了の場合は，PPP が正常な閉鎖を保証します。
  正常に終了すると，リンクは Link Dead の状態に戻り，リンクを開く要求か，物理層の再確立を示す信号を待つことになります。

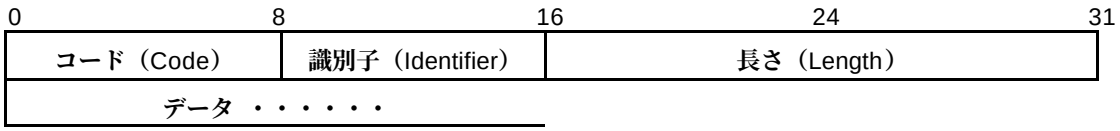
## 7. 3　リンク制御プロトコル（LCP：Link Control Protocol）

電話回線を使っての接続は，LAN 接続の場合と違い，常時接続されていないのが普通です。そのためデータを送る前に本当に相手とつながっていることを確認する必要があります。また，電話が途中で切れたり，長い間データのやり取りがないような状態が続いたりしたときは，しかるべき手続きでセッションを終了させる必要があります。このようにリンクの確立や設定，およびテストが行えるプロトコルが“ リンク制御プロトコル（LCP）”です。

### 7.3.1　LCP パケットの一般的なフォーマット

LCP パケットの一般的なフォーマットは以下のようになります。

図 7－2　LCP パケットの一般的なフォーマット




“ コード（Code）”にはパケットのタイプが入ります。パケットのタイプには次のものがあります。

表 7－1　LCPコード表

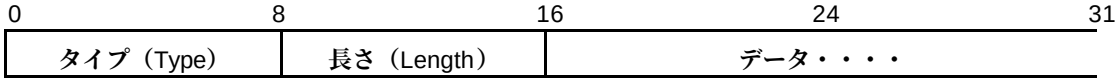
| コード | 説 明 |
|---|---|
| 1 | 設定要求 |
| 2 | 設定に対する肯定応答確認（ACK） |
| 3 | 設定に対する否定応答確認（NAK） |
| 4 | 設定拒絶（拒絶） |
| 5 | 終了要求 |
| 6 | 終了応答確認 |
| 7 | コード拒絶 |
| 8 | プロトコル拒絶 |
| 9 | エコー要求 |
| 10 | エコー応答 |
| 11 | 廃棄要求 |
| 12 | 識別 |
| 13 | 残存時間 |

### 7.3.2　LCP 設定要求

リンクの初期設定や，リンクが現在使用しているパラメータを変更しようとする場合，必ず設定要求を送ります。この要求では，LCP パケットのコードフィールドは 1 に設定され，データフィールドにはオプションのリストが入ります。

オプションのフォーマットは以下のようになります。

図 7－3　LCP の設定オプションの一般的なフォーマット




“タイプ”フィールドには，表 7－2で示された値の 1 つが設定されます。“長さ”フィールドには，タイプ，長さ，データの各フィールドを含め，オプションの全長が設定されます。“データ”フィールドには，取り決める設定オプション固有情報が入ります。指定したどのオプションにも交渉が行われ，適切な応答が返送されます。

RX-NET が PPP ネゴシエーションを行う際に使用するオプション設定は以下のようになります。

表 7－2　LCP設定要求のタイプ値

| タイプ | 説 明 |
|---|---|
| 1 | 最大受信単位（MRU） |
| 3 | 認証プロトコル |
| 5 | マジックナンバー |
| 7 | プロトコル・フィールド圧縮 |
| 8 | アドレスおよび制御フィールド圧縮 |

- 最大受信単位（MRU：Maximum Receive Unit）

  RX-NET（PPP）では，ディフォールトの 1500 を使用するので，このオプションは使用しません。一般的に“最大受信単位”オプションを使うのは，最大受信単位（MRU）の変更要求をリモートの相手に伝える場合です。つまり“パケットサイズを小さくしてほしい”あるいは“もっと大きなパケットを受信できる”のどちらかを示すことになります。

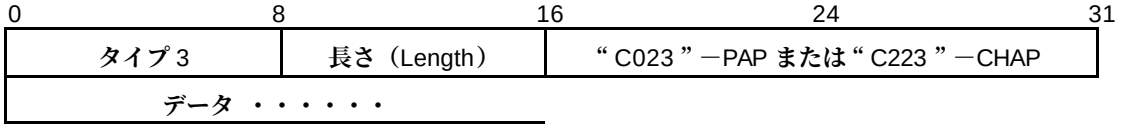
・ 認証プロトコル

信用できる装置とだけ通信したい場合，PPP では標準認証プロトコルを利用することができます。

RX-NET（PPP）では ppp_connect()発行時に“ 認証サーバモード ”を指定した場合，標準認証プロトコルである PAP または CHAP を送信します。PAP と CHAP を両方有効にした場合，CHAP 要求に対してクライアントから PAP への変更要求が来た場合には，PAP 要求に変更します。それ以外の場合は，クライアントからの認証プロトコルの変更要求は受け付けません。

なお，認証プロトコルについての詳細は“ 7. 4 PPP 認証プロトコル ”を参照してください。

図 7－4　LCP 認証プロトコル設定

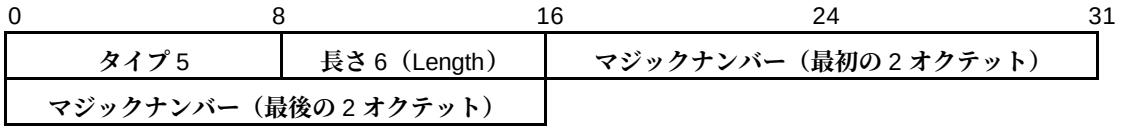



・ マジックナンバー

RX-NET（PPP）では使用しないので，このオプションは送信しません。

一般的に“ マジックナンバー ”は特定の相手を個別に認識するものです。それぞれのピアが自分の 32 ビットナンバーを選択し，LCP を使ってそれが固有であることを確認します。装置の製造番号や MAC アドレスを基礎にするのが一般的ですが，固有であることが確認できれば任意の値を使うことができます。

図 7－5　LCP マジックナンバー取り決め




・ プロトコル・フィールド圧縮（PFC：Protocol Field Compression）

RX-NET（PPP）では ppp_connect()発行時に“ プロトコル・フィールド圧縮を有効 ”にした場合に送信します。NAK や REJ が返されたときは，相手の設定に従います。

プロトコル・フィールド圧縮において圧縮されるのは，通常のデータパケットのみで，LCP パケットは必ず非圧縮フォームで送られます。

・ アドレスおよび制御フィールド圧縮（ACFC：Address Control Field Compression）

RX-NET（PPP）では ppp_connect()発行時に“ アドレスおよび制御フィールド圧縮を有効 ”にした場合に送信します。NAK や REJ が返されたときは，相手の設定に従います。

アドレスおよび制御フィールド圧縮において圧縮されるのは，通常のデータパケットのみで，LCP パケットは必ず非圧縮フォームで送られます。

・ 非同期文字マップ（0xa0000）

0x1d，0x1f をエスケープ送信するように相手に依頼します。

低速リンクでデータを送る場合は，オーバーヘッドをできるだけ小さくする必要があります。したがって PPP パケットで節約できるところがあれば，それを十分に活用すべきです。これらの“プロトコル・フィールド圧縮”や“アドレスおよび制御フィールド圧縮”を活用するのがよいでしょう。

### 7.3.3 LCP 設定要求に対する応答

相手からの要求に対しての応答には，肯定応答確認（ACK），否定応答確認（NAK），および拒絶（REJ）があります。

設定 ACK を相手に送るのは，設定要求のオプションがすべて認識でき，受け入れ可能な場合です。コードフィールトの値は 2 で，オプションフィールドには応答確認するオプションのリストが入ります。オプションの順序は設定要求パケットのときの順序と同じになります。

設定 NAK を相手に送るのは，設定要求のオプションは認識できるけれども，受け入れられないオプションがある場合です。コードフィールドの値は 3 で，オプションフィールドには，元の設定要求が使ったものと同じになります。またオプションフィールドには，受け入れられないオプションのリストが要求での順序どおりに示されます。

値フィールドを持たないオプションについては，受け入れ不可能な場合，NAK ではなく REJ が使われます。

- 認証プロトコル
  PAP または CHAP の場合は ACK を返します。

- マジックナンバー
  ACK を返します。

- プロトコル・フィールド圧縮
  ACK を返します。

- アドレスおよび制御フィールド圧縮
  ACK を返します。

- 非同期文字マップ
  0 から 0xffffffff まで，すべての値に対して ACK を返します

## 7.4　PPP 認証プロトコル

RX-NET には PPP 接続で一般に用いられる 3 種類の認証方法を実装しています。通信設定を開始する前に行う“ login 認証 ”，そして通信設定中に認証を行う“ PAP による認証 ”“ CHAP による認証 ”です。

### 7.4.1　LOGIN 認証

PPP の通信設定を開始する前に認証を行い，PPP の通信設定中には認証を行わない旧来の方法です。login 認証を行う PPP サーバに接続した場合，相手から以下の文字列が送られてきます。ただし，実際には相手によって送られてくる文字列は異なるので，事前にどのような文字列が送られてくるか確認が必要です。

```
login:
```

ここで login 名を入力すると，次の文字列が現れ，login 名に対応したパスワードの入力を求められます。ただし，ここで送られてくる文字列も相手によって異なります。

```
Password:
```

正しいパスワードを入力して認証に成功すると，初めて PPP の通信設定を送受信できる状態になります。

login 認証を使用する PPP サーバに接続する場合は，モデム制御シーケンス login_array[]に login プロンプトの期待値文字列と認証に用いる login 名とパスワードを設定した上で，PPP 接続関数 ppp_connect()の引数 auth に AUTH_LOGIN を指定して接続を行います。

### 7.4.2　PAP（Password Authentication Protocol）による認証

パスワード認証プロトコル“ PAP ”は，装置が相手を確認するための簡単な方法です。実際の手順が行われるのは，最初のリンク確立段階が終了した後だけです。リンク確立段階が終了すると，認証とパスワードの組み合わせを連続して送ります。これを応答確認の受信か，リンクの終了段階まで続けます。

サーバはクライアントから送られてきたユーザ名とパスワードの組み合わせが，サーバ上のパスワードデータベースと一致した場合に接続を許可します。

PAP は比較的簡単な認証方法であるため，製品に実装しやすい反面，盗聴に弱いという欠点があります。つまりユーザ ID とパスワードをそのまま回線上に流し，認証が成功するか，回線の接続が切れるまで ID とパスワードを流すためです。

PAP 認証を使用する PPP サーバにクライアントとして接続する場合は，ppp_connect()の引数 param に PPP_CL_PAP を指定し，name 引数と passwd 引数にユーザ名とパスワードを設定して接続を行います。

PAP 認証を使用する PPP サーバとして動作させる場合は，パスワードデータベース登録関数 ppp_setPassDB()を使ってクライアントのユーザ名とパスワードを設定した上で ppp_connect()の引数 param に PPP_SV_PAP を指定して接続を行います。

### 7.4.3　CHAP（Challenge Handshake Authentication Protocol）による認証

チャレンジハンドシェーク認証プロトコル“CHAP”は，PAP と違い，認証情報を暗号化してから送信します。ユーザ ID およびパスワードをルータ内部で暗号化し，その暗号化された ID の照合によってリンクを確立します。また，PPP リンク確立後，一定周期で認証を繰り返す（チャレンジする）ことができます。このような点から PAP よりも安全性が高くなります。

CHAP の認証手順は次のようになります。

- サーバがランダムなデータ（Challenge メッセージ）をクライアントに送信します。
- クライアントは受信した Challenge メッセージとパスワードを使って暗号文を作り，サーバに送信します。このとき暗号化アルゴリズムとして MD5（Message Digest Algorithm）と呼ばれる方式を使用します。
- サーバも同じアルゴリズムで Challenge メッセージとパスワードを使って暗号文を作り，クライアントが送信した暗号文と比較します。一致した場合はパスワードが合っていることになります。合致すれば認証が成功したものとして接続を許可し，合致しなければ切断します。

この方式の利点はパスワードを直接送信しないため，パスワードを盗まれる危険性が少ないこと，そして暗号文を元にランダムなデータを含ませることで毎回結果を変えて結果の予測を困難にしていることです。つまり毎回同じ Challenge メッセージを使用した場合は，結果として出力される暗号文も同じものになり，パケット盗聴によるなりすましを防ぐ CHAP 認証の利点が生かされません。サーバは Challenge メッセージを毎回変更することを推奨します。

また CHAP では，Challenge メッセージを送る繰り返し回数や時間間隔を制限することもできます。

CHAP 認証を使用する PPP サーバにクライアントとして接続する場合は，ppp_connect()の引数 param に PPP_CL_CHAP を指定し，引数 name と passwd にユーザ名とパスワードを設定して接続を行います。

CHAP 認証を使用する PPP サーバとして動作させる場合は，パスワードデータベース登録関数 ppp_setPassDB() を使ってクライアントのユーザ名とパスワードを設定し，関数 ppp_setChapChallengeMsg()を使って Challenge メッセージを設定した上で，ppp_connect()の引数 param に PPP_SV_CHAP を指定して接続を行います。CHAP 認証サーバとして動作させる場合の記述例を以下に示します。

```
/* サーバIPを "10.30.180.1" に指定 */
u32   ipaddr  = inet_addr("10.30.180.1");
/* サブネットマスクを "255.255.255.0" に指定 */
u32   netmask = inet_addr("255.255.255.0");
so_initialize();      /* RX-NETの初期化は最初に実行 */
/* ↓ パスワードデータベースの0番目のエントリにユーザ名foo        */
/* ↓ パスワードbar,IPアドレスはクライアント側から指定 と登録する  */
ppp_setPassDB(0, "foo", "bar", inet_addr("0.0.0.0"));
/* ↓ ppp_connect()呼び出し前に今回のChallengeメッセージを指定する   */
/* ↓ メッセージabcdefg, サーバ名pppservと指定する */
ppp_setChapChallengeMsg("abcdefg", strlen("abcdefg"), "pppserv");
/* ↓ CHAP認証サーバとして接続開始 */
ppp_connect(NULL, NULL, &ipaddr, netmask, 100000, PPP_SV_CHAP);
```

## 7.5　IPCP（Internet Protocol Control Protocol）

IPCP は，IP 圧縮プロトコル，IP アドレスなどを取り決めるプロトコルです。RX-NET（PPP）が PPP ネゴシエーションを行う際は，ユーザが API を発行すると，その内容に従って RX-NET（PPP）内で IPCP のオプション設定を行い，交渉します。

圧縮プロトコルには，Van Jacobson の“ TCP Header Compression Protocol（VJ 圧縮）”があり，TCP/IP ヘッダを圧縮します。このヘッダ圧縮を利用するかどうかを取り決めます。VJ 圧縮によって TCP/IP のヘッダは大幅に小さくすることができます。つまりヘッダ圧縮を利用することによって通信効率は上がりますが，PPP の実装には必須となっていないので，お互いの実装を確認して圧縮を使うかどうかを決める必要があります。この交渉は，ppp_connect()発行時に VJ 圧縮を有効したか否かで決定します。

またダイアルアップで接続するとき，通常は自分自身の IP アドレスは持っていません。常時接続していないコンピュータは IP アドレスを固定せず，ある範囲のアドレスを用意しておき，接続のたびに利用可能な IP アドレスを割り当てるという方法をとっていることが多くなっています。PPP には IP アドレスを取り決める機能があるので，これを利用して IP アドレスをサーバから取得します。PPP による接続が成功すると，サーバからそのとき空いている IP アドレスを受け取り，自分自身のアドレスとして使用します。ここで初めて通信準備が整います。RX-NET（PPP）では，ppp_connect()発行時に指定した IP アドレスを要求します。これにより否定応答（NAK）が返ってきた場合は，相手が指定した IP アドレスで再度要求します。

また RX-NET（PPP）では IPCP の拡張として，ネームサーバの IP アドレスをアクセスサーバから取得することも可能です。これは DNS サーバの IP アドレスをアクセスサーバから取得します。ppp_connect()発行時に“ DNS アドレスの取得 ”を指定した場合，DNS アドレスを要求します。

次に IPCP 通信の設定要求と設定要求応答について説明します。

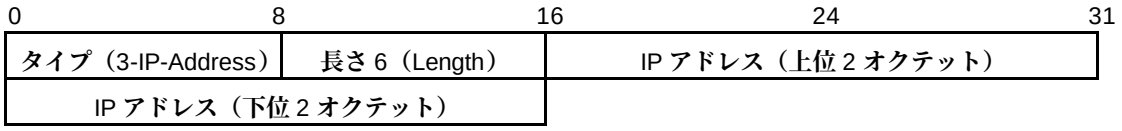
### 7.5.1 IPCP 設定要求

RX-NET（PPP）では，IPCP 通信において次のような設定要求があります。

1. IP-Address

   IP-Address オプションを使用すると，使用したい IP アドレスをピアが示したり，IP アドレスの提供をリモートピアに要求したりすることができます。RX-NET（PPP）では ppp_connect()発行時に指定した IP アドレスを要求します。一般的にはローカル側で使うべき IP アドレスをピアが交渉するためのものです。次の図が IP-Address オプションのフォーマットです。

図 7－6　IP-Address オプション

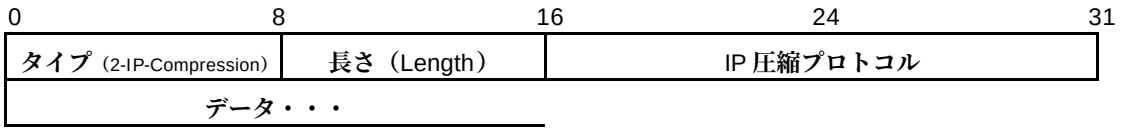



   IP-Address フィールドには 2 つの 2 オクテット値が置かれ，ピアがローカルアドレスとして使用を希望する IP アドレス，または 0 が入ります。4 つのオクテットがすべて 0 であれば，リンクのリモートエンド側に IP アドレスの割り当てを依頼することを示します。値が非 0 であっても，リモートピアが受け入れられないものであれば，リモート側がこのオプションを否定応答確認（NAK）して有効な IP アドレスを送り，それをローカル側が使用します。RX-NET（PPP）では，NAK が返された時は相手が指定した IP アドレスで再度要求を出します。

2. IP-Compression-Protocol（IP 圧縮プロトコル）

   初期設定では PPP 上の IP は圧縮を使用しません。圧縮が必要なときは，IPCP を使用して IPCP 設定オプションで圧縮プロトコルを交渉することができます。このオプションの基本フォーマットは以下のとおりです。

図 7－7　IP 圧縮プロトコルのオプション




   このオプションは必ず 4 オクテット以上です。よって長さフィールドが示す値は 4 以上になります。IP 圧縮プロトコルフィールドは，使用する圧縮プロトコルを示すために使用します。RFC1332 が定義しているのは VJ 圧縮のみで，このフィールドに使うべき値として 0x002D が示されています。データフィールドには，それぞれの圧縮プロトコルが必要とする追加情報を入れるために使います。VJ 圧縮では長さフィールドは 6 に設定され，データフィールドの 2 つの 1 オクテットフィールドが“ Max スロット IP ”と“ Comp スロット ID ”を示します。

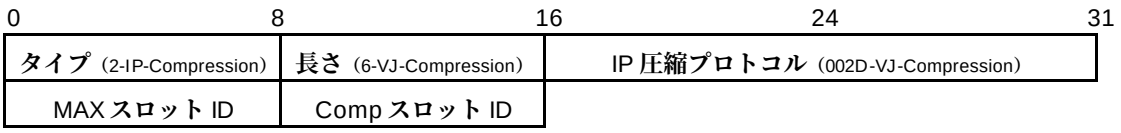
3. Van Jacobson 圧縮

Van Jacobson 圧縮（VJ 圧縮）を使用すると，TCP/IP ヘッダサイズを大幅に小さくすることが可能です。詳細については“ 7.6.3 Van Jacobson TCP ヘッダ圧縮（VJ 圧縮） ”を参照してください。

ppp_connect( )で VJ 圧縮を有効にした場合“ スロット数 16 ”“ connection-ID 圧縮有効 ”を要求します。NAK や REJ が返されたときは，相手の設定に従います。

このオプションの基本フォーマットは以下のとおりです。

図 7－8　Van Jacobson 圧縮の IP 圧縮オプション




MAX スロット ID フィールドは，ピアがサポートするコネクションスロット（配列項目）の最大数を示します（スロット数は 0 から数えるため，実際の値は“ 最大数-1 ”）。

Comp スロット ID フィールドは，スロット番号が圧縮できるかどうかを示します。この値が 0 の場合，スロット番号フィールドを作り，VJ ヘッダの C フラグを設定する必要があります。この値が 1 の場合，コネクションが変更されていない限り，VJ ヘッダのコネクション番号を省略することができます。

交渉が成功すると，IP の入った PPP パケットが送信されます。

## 7.5.2　IPCP 設定要求に対する応答

相手からの要求に対しての応答には，肯定応答確認（ACK），否定応答確認（NAK），および拒絶（REJ）があります。

- IP-Address

サーバモード時は認証したユーザ名に割り振られた IP アドレスを通知します。IP アドレスを割り当てていない場合は，クライアントからの要求に従います。

クライアントモード時は常に ACK を返します。

- Van Jacobson 圧縮

スロット数 2～16，connection-ID 圧縮有効・無効のいずれの場合も ACK を返します。

## 7.6　圧縮

低速のシリアルリンクで行う通信では，利用できる帯域幅をできるだけ多く絞り出さなければならないという問題があります。リモートからダイアルアップ回線でインターネットにアクセスする場合，一般的に使用されるモデムは 14.4kbps か 28.8kbps です。これを使用して大きなファイルをダウンロードしようとすると，帯域幅の有効利用が必要になってきます。

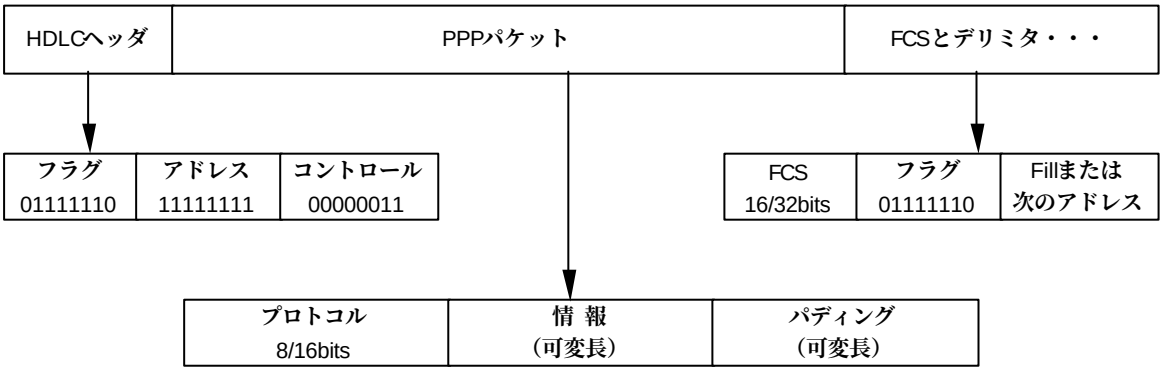
そこで使用される技術が圧縮です。RX-NET（PPP）では，

・アドレスおよび制御フィールド圧縮（ACFC：Address Control Field Compression）
・プロトコル・フィールド圧縮（PFC：Protocol Field Compression）
・Van Jacobson TCP ヘッダ圧縮（TCP Header Compression Protocol（VJ 圧縮））

の 3 つの圧縮をサポートしています。

RX-NET（PPP）で採用している HDLC 方式フレームは，次のような構造になっています。

図 7－9　HDLC 方式フレームの PPP




これらのフレームのうち，圧縮されるフィールドは HDLC ヘッダ部分と，PPP パケットのプロトコル部分になります。それぞれの圧縮について，圧縮部分および圧縮を使用する方法を説明します。

### 7.6.1　アドレスおよび制御フィールド圧縮

HDLC ヘッダのアドレス部分とコントロール部分の 2 バイトは，常に同じ値を使用するフィールドになるので，省略が可能です。この圧縮では，この 2 バイトを省略します。

アドレスおよび制御フィールド圧縮を使用する場合，`ppp_connect()`の引数 `param` に PPP_CMP_ADDR を指定して行います。

### 7.6.2　プロトコル・フィールド圧縮

PPP パケットフォーマットでは，プロトコル・フィールドは 2 バイトになっています。しかし 1 バイトに収まる（255 以下）値を持つプロトコル番号は 1 バイトで送信することができます。このプロトコル・フィールド圧縮では，プロトコル番号を 1 バイトで送信します。

プロトコル・フィールド圧縮を使用する場合，ppp_connect()の引数 param に PPP_CMP_PROT を指定して行います。

### 7.6.3　Van Jacobson TCP ヘッダ圧縮（VJ 圧縮）

Van Jacobson TCP ヘッダ圧縮（VJ 圧縮）は，連続して送信される TCP ヘッダ内の冗長なデータを圧縮するプロトコルです。TCP 通信の同一コネクションのパケットでは，ソースと通信相手のアドレス，ポート番号等のデータは常に同じで変化しません。そこで VJ 圧縮では，TCP のパケットを送信するときに以前に送信した TCP パケットとの TCP ヘッダの差分情報だけを送信することで，TCP ヘッダサイズを小さくしています。

VJ 圧縮を使用する場合，ppp_connect()の引数 param に PPP_CMP_VJ を指定して行います。

# 7. 7　接続の形態

RX-NET（PPP）は，PPP による TCP/IP ネットワークへの接続機能を提供します。

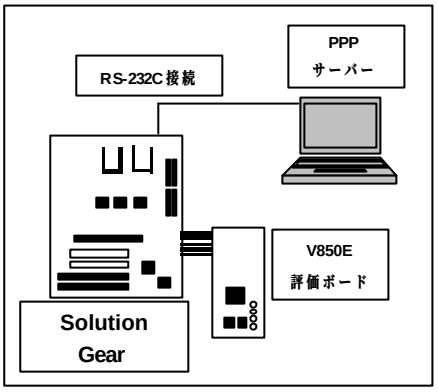
TCP/IP ネットワークへの接続形態には“ ヌルモデム接続 ”と“ モデム接続 ”の 2 種類があります。

ここでは、それぞれの接続形態の接続方法、認証方法、RX-NET（PPP）が相手側の PPP サーバと通信に用いるディフォールトの通信設定について説明します。

## 7.7.1　ヌルモデム接続

RS-232C のクロスケーブルで，RX-NET（PPP）の載ったターゲットと PPP サーバを接続する形態です。

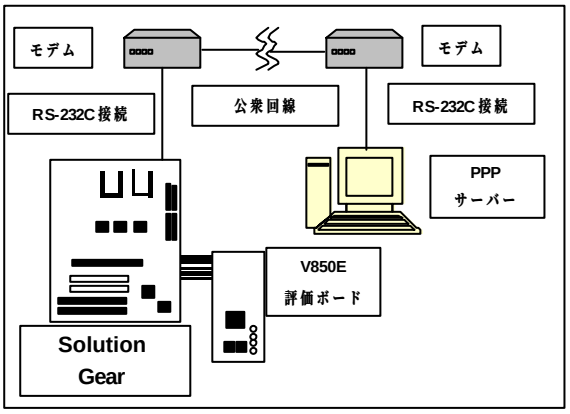
ユーザ・アプリケーションで，モデムの初期化と公衆回線の接続を行ってから RX-NET（PPP）で PPP を利用する場合も，この機能を使用します。




ヌルモデム接続機能を使用する場合は，PPP 接続関数 ppp_connect()の引数 param に PPP_DIAL を“ 指定せず ”に接続を行います。

## 7.7.2　モデム接続

モデムを使って公衆回線経由で RX-NET（PPP）の載ったターゲットと PPP サーバを接続する形態です。




モデム接続機能を使用する場合は，PPP 接続関数 ppp_connect()の引数 param に PPP_DIAL を指定して接続を行います。

# 7.8　RX-NET（PPP）における処理の流れ

次に RX-NET（PPP）を使用したときの処理の流れについて説明します。
モデム経由で ISP（Internet Service Provider）に接続し，切断するまでの流れを説明します。

## 7.8.1　モデム設定

モデムを使用するため，モデムの設定を行います。ここでは，

- モデム初期化コマンド &F
- 電話回線の種別　　トーン
- ISP の電話番号　　012-345-6789

とします。これらの設定は配列“ dial_array[] ”にて行います。dial_array[]は，モデム制御シーケンス構造体“ SE_SEQUENCE 型 ”です。SE_SEQUENCE 型の詳細は以下のとおりです。

［ モデム制御シーケンス構造体 SE_SEQUENCE 型のプロトタイプ ］

```
#define MAX_SEND_EXPECT  20
typedef struct se_sequence {
    char   send[MAX_SEND_EXPECT];
    char   expect[MAX_SEND_EXPECT];
    long   timeout;
    int    trys;
}SE_SEQUENCE;
```

| メンバ | 説 明 |
|---|---|
| **send** | 送信文字列を格納します |
| **expect** | 送信文字列に対する応答の期待値文字列を格納します |
| **timeout** | 送信文字列の送信から，期待値文字列の検査を行うまでの待ち時間を格納します（ms） |
| **trys** | 期待値文字列がマッチしなかった場合の再試行の回数を格納します |

なお，これらの 2 つの配列の最後の要素には，終端要素の目印として，

```
{“¥0”, “¥0”, 0, 0}
```

を配置する必要があります。

SE_SEQUENCE を参照する個所では，次の手順でこの構造体を利用します。

1. 終端要素だったら終了
2. send に格納された文字列を UART に送信する
3. メンバ timeout に格納された時間だけ待ち状態に入る
4. UART から文字列を受信する
5. 受信した文字列とメンバ expect の文字列比較を行う
   2 つの文字列の終端から，短い方の文字列の長さだけ比較します。長い方の文字列の先頭部分は無視されます。
6. 比較の結果が一致していたら，次の要素に進んで 1 番に戻る。一致していない場合は 3 番に戻る。すでに **trys** 回の試行を行っていたら終了

dial_array[]で“ ATD ”コマンドを使用した時は特別で，モデムの DCD 信号線が“ 1 ”になるまで待ち状態に入ります。このとき expect メンバは使用されません。

このときの dial_array[]の定義は次のようになります。

```
SE_SEQUENCE  dial_array[] = {
{“AT&F¥r”, “OK¥r¥n”, 500L, 2},
{“ATDT012-345-6789¥r”, “CONNECT 115200¥r”, 200000L, 2},
{“¥0”, “¥0”, 0, 0}
};
```

この定義の意味は，次のようになります。

1. 「AT&F¥r」を送信します。（AT コマンドでモデム初期化コマンド&F を送信）
2. 500ms（0.5 秒）待ちます
3. 応答結果が期待値「OK¥r¥n」と同じだった場合，次に進む
4. 「ATDT012-345-6789¥r」を送信します。（AT コマンドでトーン回線に 012-345-6789 をダイアル）
5. 200000ms（200 秒）待ちます
6. 200000ms（200 秒）経過する前に DCD 線が“ 1 ”になれば正常終了

このとき，dial_array[]に記述する ATD コマンドは大文字で記述してください。小文字で記述した場合，RX-NET（PPP）が ATD コマンドを認識できません。

上記の 2 つ目の要素において，ATDT コマンドの後は DCD 変化のイベントを待つため，ダイアルアップ応答の期待値「CONNECT 115200¥r」は現在使用されていません。

また，モデムを使用せずにヌルモデム接続を行う場合は，dial_array[]は使用されないので，設定の変更は必要ありません。なおヌルモデム接続の方法については“ 7.8.4 ppp_connect ”を参照してください。

### 7.8.2　LOGIN 認証の設定

LOGIN 認証をする場合の送受信文字列の設定を行います。LOGIN 認証を行わない場合は，この設定を行う必要はありません。

ここでは，

- ログイン・プロンプト　“ogin: ”
- パスワード・プロンプト　“assword”
- ログイン名　“pppusername”
- パスワード　“ppppassword”
- ログイン・プロンプトが出るまでの待ち時間　10 秒（3 回）
- パスワード・プロンプトが出るまでの待ち時間　10 秒
- パスワード入力後の待ち時間　10 秒

とします。このときの login_array[]の定義は次のようになります。

```
SE_SEQUENCE  login_array[] = {
{“¥r”, “ogin: ”, 10000, 3},
{“pppusername¥r”, “assword: ”, 10000, 1},
{“ppppassword¥r”, “¥0”, 10000, 1},
{“¥0”, “¥0”, 0, 0}
};
```

これらの情報は ISP のアカウントにあわせて修正してください。

### 7.8.3　ISP の IP アドレス設定

ISP のアドレス設定を行います。設定する項目は“ IP アドレス ”と“ ネットマスク ”です。値は変数に代入し，その変数を PPP 接続関数“ ppp_connect() ”の引数として使用します。

ただし，通常は IP アドレスの指定は行いません。ISP が動的に IP アドレスを割り当てることが多いためです。IP アドレスをしてしない場合は，変数に“ 0 ”を代入します。IP アドレスを“ ipaddr ”，ネットマスクを“ netmask ”とすると，

```
ipaddr = 0;
netmask = 0;
```

となります。値が必要な場合はネットワーク・バイト・オーダーでアドレスを使用する必要あるので，inet_addr 関数を使用して値を代入します。

```
#define  IP_ADDR  "10.30.180.182"
#define  RXNET_NETMASK  "255.255.255.0"

ipaddr = inet_addr(IP_ADDR);
netmask = inet_addr(NETMASK);
```

### 7.8.4 ppp_connect

次に PPP 接続関数を発行します。PPP 接続関数は ppp_connect()です。

ppp_connect()プロトタイプ

```
int  ppp_connect(char* name,char* passwd,u32* ipaddr,u32 netmask,int timeout,int param );
```

PPP の接続を開始します。まず PPP の接続を開始する前に RX-NET の初期化関数“ so_initialize() ”を呼び出します。この関数の詳細については“ 6.1.1 so_initialize ”を参照してください。

- 第一パラメータ“ ***name*** ”は，クライアントとして PAP，CHAP で使用するユーザ名文字列のポインタを指定します。ユーザ名には 1 バイト以上 24 バイト以下の文字列が使用できます。なお，このパラメータは param 引数に PPP_CL_PAP または PPP_CL_CHAP を指定した場合にのみ参照されます。
- 第二パラメータ“ ***passwd*** ”は，クライアントとして PAP，CHAP で使用するパスワード文字列のポインタを指定します。パスワードには 1 バイト以上 14 バイト以下の文字列が使用できます。なお，このパラメータは param 引数に PPP_CL_PAP または PPP_CL_CHAP を指定した場合にのみ参照されます。
- 第三パラメータ“ ***ipaddr*** ”は，ネットワーク・デバイスに割り当てる IP アドレスをネットワーク・バイト・オーダーで指定します。IP アドレスを接続相手に割り当ててもらう場合には“ 0 ”を指定します。このときは ppp_connect()の戻り値として決定した IP アドレスが返却されます。
- 第四パラメータ“ ***netmask*** ”は，ネットマスクをネットワーク・バイト・オーダーで指定します。
- 第五パラメータ“ ***timeout*** ”は，PPP 接続設定（LCP ネゴシエーション等）の完了を待つ時間（ms）を指定します。PPP の接続設定開始から ***timeout*** 時間までに接続設定が決定しない場合は，回線を切断します。0～0x7fffffff の値が指定可能です。

・ 第六パラメータ“ ***param*** ”は，通信設定のパラメータを指定します。複数の値を指定する場合は，“ | ”（論理和）を使用します。パラメータ名とその意味は次の通りです。

| パラメータ名 | 意 味 |
|---|---|
| PPP_DIAL | モデム接続（ダイアルアップ）します。<br>ダイアル・アップ処理内容については“ 7.8.1 モデム設定 ”を参照してください。<br>ヌルモデム接続の場合は PPP_DIAL を指定しないでください。 |
| PPP_LOGIN | LOGIN 文字列を処理します。<br>LOGIN 文字処理の内容については“ 7.8.2　LOGIN 認証の設定 ”を参照してください。 |
| PPP_CL_PAP | クライアントとして PAP 認証を使用します。<br>通信相手から PAP 認証を要求された場合，引数 ***name*** と ***password*** の値を使って認証を試みます。 |
| PPP_CL_CHAP | クライアントとして CHAP 認証を使用します。<br>通信相手から CHAP 認証を要求された場合，引数 ***name*** と ***password*** の値を使って認証を試みます。 |
| PPP_CL_MSDNS | DNS アドレスの取得を要求します。<br>取得したアドレスは<br>`ppp_getNameServerIP()`，`ppp_getNameServerIP2()`で参照可能です。 |
| PPP_SV_PAP | サーバとして PAP 認証を使用します。<br>これを指定した場合は，認証サーバモードで動作します。PPP_SV_PAP と PPP_SV_CHAP を両方指定した場合は，PPP_SV_CHAP が優先されます。 |
| PPP_SV_CHAP | サーバとして CHAP 認証を使用します。<br>これを指定した場合は，認証サーバモードで動作します。PPP_SV_PAP と PPP_SV_CHAP を両方指定した場合は，PPP_SV_CHAP が優先されます。 |
| PPP_CMP_VJ | Van Jacobson TCP ヘッダ圧縮を使用します。 |
| PPP_CMP_ADDR | Address/Control フィールド圧縮を使用します。 |
| PPP_CMP_PROT | プロトコル・フィールド圧縮を使用します。 |

この関数の戻り値は以下のようになります。

| マクロ | 値 | 内 容 |
|---|---|---|
| EPPP_NOERR | 0x0 | 正常終了 |
| EPPP_INVAL | 0x1300 | パラメータの指定が不正です。次のような原因が考えられます。<br>・クライアントとして PAP, CHAP 使用時に ***name*** が NULL になっています。<br>・クライアントとして PAP, CHAP 使用時に ***passwd*** が NULL になっています。<br>・クライアントとして PAP, CHAP 使用時に ***name*** のデータ長が不正です。<br>・クライアントとして PAP, CHAP 使用時に ***passwd*** のデータ長が不正です。<br>・***ipaddr*** に NULL を指定しています。<br>・***timeout*** に負の数が指定されています。 |
| EPPP_NOPPP | 0x1301 | PPP が初期化されていません |
| EPPP_NETDOWN | 0x1302 | RX-NET が初期化されていません |
| EPPP_DIALFAIL | 0x1303 | ダイアルアップ処理に失敗しました |
| EPPP_LOGINFAIL | 0x1303 | LOGIN 文字処理に失敗しました |
| EPPP_DEVICEFAIL | 0x1305 | UARTの初期化に失敗しました。次のような原因が考えられます。<br>・`ll_config()`が異常終了しています。 |
| EPPP_TIMEOUT | 0x1307 | PPP のネゴシエーションに失敗しました |
| EPPP_NOMEM | 0x1308 | ヒープメモリが不足しています |
| EPPP_CONNECT | 0x1309 | すでに PPP 接続されています |

次のように `ppp_connect()`を発行したとします。

```
#define  PAPNAME  "papname"
#define  PAPPASSWD  "pappasswd"

ppp_connect(PAPNAME, PAPPASSWD, &ipaddr, netmask, 5000,
          PPP_LOGIN | PPP_DIAL | PPP_CMP_VJ | PPP_CMP_ADDR | PPP_CMP_PROT);
```

意味は次のようになります。

- PAP 認証のアカウント名　　“papname”
- PAP 認証のパスワード　　“pappasswd”
- ダイアル後の接続タイムアウト時間　　“5000ms（5 秒）”
- ダイアルアップで接続
- VJ 圧縮使用
- Address/Control フィールド圧縮使用
- プロトコル・フィールド圧縮使用

### 7.8.5 ppp_disconnect

次に PPP 接続切断関数について説明します。PPP 接続を切断するには ppp_disconnect()を発行します。

ppp_disconnect()プロトタイプ

```
int  ppp_disconnect(u32 ipaddr,u32 netmask);
```

PPP の接続を終了し，回線を切断します。

- 第一パラメータ“ *ipaddr* ”は，ネットワーク・デバイスに割り当てる IP アドレスをネットワーク・バイト・オーダーで指定します。ppp_connect()の戻り値である IP アドレスを指定します。
- 第二パラメータ“ *netmask* ”は，ネットマスクをネットワーク・バイト・オーダーで指定します。ppp_connect()の呼び出し時に使用したネットマスクを指定します。

| マクロ | 値 | 内 容 |
|---|---|---|
| EPPP_NOERR | 0x0 | 正常終了 |
| EPPP_NOPPP | 0x1301 | PPP が初期化されていません |
| EPPP_NETDOWN | 0x1302 | RX-NET が初期化されていません |
| EPPP_DEVICEFAIL | 0x1305 | UART の終了処理に失敗しました。次のような原因が考えられます。<br>・ll_unconfig()が異常終了しています。 |
| EPPP_NOTCONN | 0x130a | すでに PPP 接続は切断されています |

# 7. 9　RX-NET（PPP）のその他の API 関数

RX-NET（PPP）で用意されている，その他の API 関数について説明します。

## 7.9.1　ppp_getNameServerIP

ppp_connect()の引数 ***param*** に PPP_CL_MSDNS を指定して DNS アドレスの取得を要求した場合，取得したアドレスを参照するときに ppp_getNameServerIP()／ppp_getNameServerIP2()を発行します。

プライマリ DNS アドレスを参照する場合は“ ppp_getNameServerIP()”，セカンダリ DNS アドレスを参照する場合は“ ppp_getNameServerIP2()”を使用します。

ppp_getNameServerIP()プロトタイプ

```
int  ppp_getNameServerIP(u32* ipaddr);
```

ppp_connect()の引数 ***param*** に PPP_CL_MSDNS を指定して DNS アドレスを要求し，接続相手が DNS アドレスを通知した場合，この関数でプライマリ DNS アドレスを取得することができます。

ppp_connect()の引数に PPP_CL_MSDNS を指定していない，または接続相手が DNS アドレスの通知を行っていなかった場合はエラーが返ります。

- 第一パラメータ“ ***ipaddr*** ”は，プライマリ DNS の IP アドレスを格納する領域のポインタを指定します。プライマリ DNS の IP アドレスはネットワーク・バイト・オーダーで格納されます。

| マクロ | 値 | 内 容 |
|---|---|---|
| EPPP_NOERR | 0x0 | 正常終了 |
| EPPP_INVAL | 0x1300 | パラメータの指定が不正です。次のような原因が考えられます。<br>・***ipaddr*** に NULL を指定しています。 |
| EPPP_NOPPP | 0x1301 | PPP が初期化されていません |
| EPPP_NETDOWN | 0x1302 | RX-NET が初期化されていません |
| EPPP_NOTCONN | 0x130a | PPP 接続されていないため，情報がありません |
| EPPP_NOMSDNS | 0x130b | DNS アドレス情報がありません |

### 7.9.2　ppp_getNameServerIP2

ppp_connect()の引数 ***param*** に PPP_CL_MSDNS を指定して DNS アドレスの取得を要求した場合，取得したアドレスを参照するときに ppp_getNameServerIP()／ppp_getNameServerIP2()を発行します。

プライマリ DNS アドレスを参照する場合は“ ppp_getNameServerIP()”，セカンダリ DNS アドレスを参照する場合は“ ppp_getNameServerIP2()”を使用します。

ppp_getNameServerIP2()プロトタイプ

```
int  ppp_getNameServerIP2(u32* ipaddr);
```

ppp_connect()の引数 ***param*** に PPP_CL_MSDNS を指定して DNS アドレスを要求し，接続相手が DNS アドレスを通知した場合，この関数でセカンダリ DNS アドレスを取得することができます。

ppp_connect()の引数に PPP_CL_MSDNS を指定していない，または接続相手が DNS アドレスの通知を行っていなかった場合はエラーが返ります。

- 第一パラメータ“ ***ipaddr*** ”は，セカンダリ DNS の IP アドレスを格納する領域のポインタを指定します。セカンダリ DNS の IP アドレスはネットワーク・バイト・オーダーで格納されます。

| マクロ | 値 | 内 容 |
|---|---|---|
| EPPP_NOERR | 0x0 | 正常終了 |
| EPPP_INVAL | 0x1300 | パラメータの指定が不正です。次のような原因が考えられます。<br>・***ipaddr*** に NULL を指定しています。 |
| EPPP_NOPPP | 0x1301 | PPP が初期化されていません |
| EPPP_NETDOWN | 0x1302 | RX-NET が初期化されていません |
| EPPP_NOTCONN | 0x130a | PPP 接続されていないため，情報がありません |
| EPPP_NOMSDNS | 0x130b | DNS アドレス情報がありません |

### 7.9.3　ppp_setPassDB

認証サーバのパスワードデータベースを変更します。

ppp_setPassDB()プロトタイプ

```
int  ppp_setPassDB(int number, char* name, char* passwd, u32 ipaddr);
```

認証サーバモード時に参照するパスワードデータベースにデータを設定する関数です。

RX-NET（PPP）はパスワードデータベースに 10 個のエントリを用意しており，最大で 10 のクライアントを識別することが可能です。認識時には，パスワードデータベースは先頭から検索されます。データベース中に重複するユーザ名がある場合，先に見つかったデータだけが使用されます。

- 第一パラメータ“ ***number*** ”は，データの格納先の番号を指定します。0～9 までの値が指定可能です。
- 第二パラメータ“ ***name*** ”は，認証相手のユーザ名を指定します。ユーザ名には 1 バイト以上 24 バイト以下の文字列が使用可能です。
- 第三パラメータ“ ***passwd*** ”は，引数 ***name*** のユーザ名に対応したパスワードを指定します。パスワードには 1 バイト以上 24 バイト以下の文字列が使用可能です。
- 第四パラメータ“ ***ipaddr*** ”は，引数 ***name*** のユーザ名に割り当てる IP アドレスをネットワーク・バイト・オーダーで指定します。サーバ側で IP アドレスの割り当てを行わない場合は，IP アドレス“ 0.0.0.0 ”を指定します。

| マクロ | 値 | 内 容 |
|---|---|---|
| EPPP_NOERR | 0x0 | 正常終了 |
| EPPP_INVAL | 0x1300 | パラメータの指定が不正です。次のような原因が考えられます。<br>・***number*** に 0～9 以外の値を指定しています。<br>・***name***，***passwd*** に NULL を指定しています。<br>・***name***，***passwd*** のデータ長が不正です。 |
| EPPP_NETDOWN | 0x1302 | RX-NET が初期化されていません |

### 7.9.4 ppp_setChapChallengeMsg

CHAP 認証サーバの Challenge メッセージを設定します。

ppp_setChapChallengeMsg()プロトタイプ

```
int  ppp_setChapChallengeMsg(char* msg, int msglen, char* name);
```

CHAP 認証サーバがクライアントに対して送信する Challenge メッセージに使用する文字列を設定します。セキュリティ上 Challenge メッセージは毎回変更することを推奨します。

- 第一パラメータ“***msg***”は，Challenge メッセージのポインタを指定します。
- 第二パラメータ“***msglen***”は，引数 ***msg*** のポインタに指定した Challenge メッセージのバイト数を指定します。メッセージ長には 1～32 の値が指定可能です。
- 第三パラメータ“***name***”は，Challenge メッセージと共にクライアントに送信するサーバのマシン名を指定します。マシン名には 1 バイト以上 24 バイト以下の文字列が使用可能です。

| マクロ | 値 | 内 容 |
|---|---|---|
| EPPP_NOERR | 0x0 | 正常終了 |
| EPPP_INVAL | 0x1300 | パラメータの指定が不正です。次のような原因が考えられます。<br>・***msg***，***name*** に NULL を指定しています。<br>・***msglen*** に 1～32 以外の値を指定しています。<br>・***name*** のデータ長が不正です。 |
| EPPP_NETDOWN | 0x1302 | RX-NET が初期化されていません |

## 7. 10　RX-NET から RX-NET（PPP）への変更点

PPP 機能を持たない RX-NET のアプリケーションを，RX-NET（PPP）を使用して PPP 機能を持ったアプリケーションに変更する手順を説明します。ここでは，

- RX-NET → RX-NET（PPP）の順にインストールしていること
- RX-NET と RX-NET（PPP）を同一のディレクトリ（nectools32 など）にインストールしていること

を前提に説明します。

1. RX-NET コンフィギュレーション・オブジェクトの変更
   RX-NET コンフィギュレーション・オブジェクトを，Ethernet ドライバ使用のものから UART 使用のものへ変更します。リンクする RX-NET コンフィギュレーション・オブジェクトを変更してください。
   UART 使用の RX-NET コンフィギュレーション・オブジェクトを新規に作成する場合は“ 第 4 章　UART コントローラドライバ ”を参照してください。

2. ライブラリの追加
   RX-NET（PPP）を使用するアプリケーションでは“ libppp.a ”のリンクが必要です。RX-NET アプリケーションに libppp.a のリンクを追加してください。

3. インクルード・ファイルの追加
   RX-NET（PPP）のアプリケーションでは rxnet_ppp.h のインクルードが必要です。rxnet.h のインクルード文の次に rxnet_ppp.h のインクルード文を追加してください。

4. RX-NET 初期化部の変更
   初期化部の変更点は，次の 3 点です。

   - `ll_config()`，`ll_route()`の削除
   - `ppp_connect()`の追加
   - `ppp_disconnect()`の追加

   まず，`ll_config()`と `ll_route()`の呼び出し部分を削除します。`ll_config()`，`ll_route()`は，それぞれネットワーク・インタフェースの IP アドレスとデフォルト・ゲートウエイの IP アドレスを設定するための関数です。`ll_config()`，`ll_route()`に相当する処理は，PPP の接続関数 `ppp_connect()` 内で行われるので必要ないためです。
   また，PPP の通信では，通信前に PPP 回線の接続，通信完了後に PPP 回線の切断が必要です。PPP 回線の接続を行うときには“ `ppp_connect()` ”を，PPP 回線の切断を行うときには“ `ppp_disconnect()` ”を使用してください。

# 付録　Well-Known-Port

プログラム同士がソケットを通してやり取りをするのに必要な情報は“IP アドレス”と“ポート番号”です。“IP アドレス”は，コンピュータごとに決まり事に従って割り振られます。一方“ポート番号”は，1 つの IP アドレス（コンピュータ）上で動作している“ FTP ”や“ Telnet ”などのアプリケーションに，データを渡すために使用するアドレスです。これによってどのアプリケーションに対するデータかを振り分けることができます。

例えば，サーバ上で動作するネットワークアプリケーションを作成したとき，IP アドレスはサーバに割り振られているので問題ないですが，ポート番号はどのようにして決めればよいでしょうか？ ポート番号は“16 ビットで表せる数”と決まっているので，その中から勝手につけてもよいのですが，通常のサーバは OS レベルでいろいろなサービスが組み込まれて提供されています。そのため勝手にポート番号を決めてアプリケーションを作成してしまうと，バッティングしてしまう可能性があります。このため RFC1700（ASSIGNED NUMBERS）に 3 種類のポート番号が規定されています。

表 7－3　ポート番号

| 種 類 | 説 明 | 範 囲 |
|---|---|---|
| Well-Known-Port | IANA（Internet Assigned Number Authority）によってインターネット全体に対して管理されている“よく知られている”ポート番号 | 0～1023 |
| Registered-Port | IANA に対して登録されていて，利用方法が公式登録されているポート番号 | 1024～49151 |
| Private-Port | ローカルシステムにおいて私的に使用できるポート番号 | 49152～65535 |

※　Registered-Port の範囲は変更される可能性があります

独自にネットワークアプリケーションを作成した場合は，一般的には 1024 以上で，できるだけ大きい数，もしくは 49152 以上を使用するのがよいとされています。企業などでいろいろなソフトウエアがインストールされているようなサーバの場合，バッティングする可能性が高くなります。アプリケーションの仕組みとしてポート番号をユーザが与えられるような設計にし，サーバ管理者に確認を取ったうえでポート番号を設定するのがよいでしょう。

表 7－4　主なWell-Known-Port

| ポート番号 | プロトコル／サービス名 | 説 明 |
|---|---|---|
| 21 | ftp | File Transfer Protocol（ファイル転送プロトコル） |
| 23 | telnet | Telnet（仮想端末プロトコル） |
| 25 | smtp | Simple Mail Transfer Protocol（電子メール転送プロトコル） |
| 80 | http | Hyper-Text Transfer Protocol（WWW データ転送プロトコル） |
| 110 | pop3 | Post Office Protocol Version3（電子メール受信プロトコル） |
| 119 | nntp | Network News Transport Protocol（インターネットニュース配信） |

# 索 引

## 記号・数字



## A



## B



## C



## D



## E



## G



## H



## I



## L

## M



## N



## O



## P



## R



## S

〔メ　モ〕

【発　行】

**NECエレクトロニクス株式会社**

〒211-8668　神奈川県川崎市中原区下沼部1753

電話（代表）：044(435)5111

---

**【ホームページ】**

NECエレクトロニクスの情報がインターネットでご覧になれます。

URL（アドレス）　http://www.necel.co.jp/

---

**【営業関係お問い合わせ先】**

下記のページに最新版のお問い合わせ先が記載されています。

URL（アドレス）　http://www.necel.com/ja/contact/contact_j.html

---

**【技術的なお問い合わせ先】**

半導体テクニカルホットライン

（電話：午前 9:00～12:00，午後 1:00～5:00）

電　話　：044−435−9494
FAX　：044−435−9608
E-mail　：info@lsi.nec.co.jp

---

**【資料請求先】**

NECエレクトロニクス特約店または上記ホームページ記載の営業関係お問い合わせ先へお申し付けください。

---