--- 目次 ---

1. 概要 1

1. 1. 用語説明 1

1. 2. プログラミング上の注意点 2

1. 3. 使用上の注意点 2

1. 4. 電波状態(圏内外)表示について 3

2. ソフトウェア構成 3

3. PPP プロトコルサポート範囲 4

4. アプリケーション作成方法 5

4. 1. ソースコード作成 5

4. 2. メーク環境 5

4. 3. 回線接続までの処理方法 5

4. 3.1. 通信アプリケーションの作成 5

4. 3.2. 電波状態を表示アプリケーションの作成 5

5. 環境ファイル 6

5. 1. PPP で必要となるデータ 6

5. 2. ファイルレイアウト 7

6. ライブラリ関数一覧 9

6. 1. 電源 OFF/ON 発生時の処理 9

6. 2. 回線切断発生時の処理 9

6. 3. ハードウェアエラー発生時の処理 9

6. 4. PHS モジュールローバッテリー発生時の処理 9

6. 5. 回線接続／切断タイミング 9

6. 6. エラーコード 10

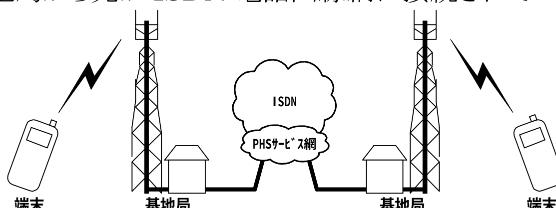
6. 7. 関数一覧 11

1. 概要

本マニュアルには DT-900M50P/51P の PHS データ通信機能を使用して TCP/IP 通信アプリケーションを作成する方法や注意点が記載されています。

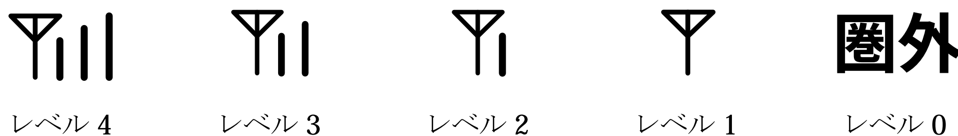
1. 1.用語説明

基地局
PHS を用いた通信は、お互いの端末同士が直接電波のやり取りをしているのではなく、基地局と呼ばれる設備を介して行われます。通常、基地局は数百メートル毎に設置されており、基地局から先が ISDN 電話回線網に接続されています。



エリア
複数の基地局によって構成されます。通常、PHS からは、エリア内に設置された複数の基地局を検出しており、中でも電波状態の良い基地局を選んで通信を行います。

電波強度
電波強度とは、基地局から届く電波の強さを、0 から 4 までの数字で表したものです。この数値を、音声通話用の PHS や携帯電話で馴染み深い表示に対応させたのが下図です。(**注意！　本機には、下の記号を表示する機能はありません。**)



レベル 4　　レベル 3　　レベル 2　　レベル 1　　レベル 0

ｻｲﾄ・ｻｰﾍﾞｲ
電波強度を測定するためのプログラムです。

PIAFS
PHS を用いたデータ通信の業界標準規格で、PHS Internet Access Forum Standard の略です。PIAFS 対応のアクセスポイント、および PIAFS 対応の TA(ターミナルアダプタ)に接続することにより、32Kbps の高速通信が可能です。現在、PIAFS 通信では 64Kbps に対応していますが、本機では 32Kbps の通信になります。

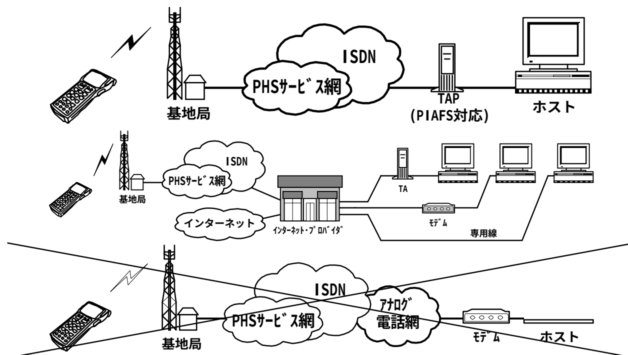
TAP
ターミナルアダプタ for PIAFS の略。PIAFS 対応の TA を意味します。

1. 2. プログラミング上の注意点

- アプリケーションジェネレータを使用して PHS 通信アプリケーションを作成することはできません。
- PHS 通信アプリケーションの設計には、必ず pppsock ライブラリを使用して下さい。通信 BIOS を使用することはできません。従って、アプリケーション・プログラムから直接 AT コマンド(モデム制御コマンド)を発行することはできません。
- pppsock ライブラリは発信のみをサポートしています。着信アプリケーションを作成することはできません。

1. 3.使用上の注意点

- 本機は、PIAFS 以外のデータ通信、および音声通話をサポートしていません。そのため、アナログモデム（パソコン通信用のモデム）と通信することはできません。この制限は、PTE（プロトコル変換機）を介したとしても変わりません。

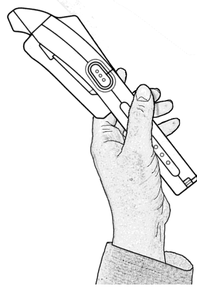


TAP を使用し、ホストコンピュータに直接ダイアルアップする方式です。

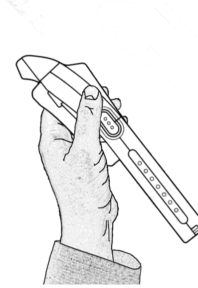
インターネット・プロバイダを介してインターネット接続を行う方式です。

プロトコル変換機を介しても、アナログモデムと通信を行うことはできません。

- 無線インターネット方式(DDI ポケット電話グループの「メディア変換機能」を利用した通信方式で、PIAFS 非対応の相手とも 32Kbps の通信が可能)、および無線モデム方式(DDI ポケット電話グループの「メディア変換機能」を利用した通信方式で、最大 14.4Kbps の通信が可能)を利用することはできません。
- サイト・サーベイで取得・表示される電波状態は、リアルタイムには変化しません。これは電波状態の変化を平均化していること、および、表示のデータの更新周期が 6 秒であることによります。詳しくは、「電波状態（圏内外）表示について(p.3)」を参照して下さい。
- 圏外から圏内に移動しても、表示が変化するまでには時間を要します。これは電池の消費を押さえるために、圏外での電波強度測定の周期を延ばしているためです。通常は、1.2 秒毎の計測を繰り返しますが、圏外では 10 秒毎に計測を行います。さらに取得された電波強度は平均化されるため、圏内に移動しても、表示が速やかに変化しないことになります。
- 同じ場所で電波状態を測定していても、常に一定の値を示すわけではありません。これは通信事業者によって電波強度に強弱が付けられているためであり、故障ではありません。この目的は、同一エリア内に存在する PHS が、電波の強い基地局に集中しないようにするためであると考えられます。
- PHS 通信部とアンテナは、副電池カバー内に収められています。そのため、この部分を覆うと、通信に悪影響を及ぼしますのでお止め下さい。



良い持ち方



悪い持ち方

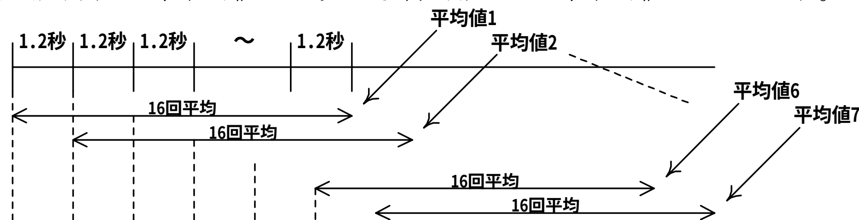
- PHSによる通信は、電波強度が3以上の場所で行って下さい。電波強度が1や2の場所で通信を行うと、繋がらなかったり、通信の途中で切れてしまったりします。[1]

レベル4	:	安定したデータ通信を行うことができます。
レベル3	:	データ通信を行うことができます。
レベル2〜1	:	電話が繋がらないか、途中で切れてしまうことがあります。電波状態の良い場所へ移動して通信を行って下さい。
レベル0	:	通信できません。通信可能な場所へ移動して下さい。

1.4.電波状態（圏内外）表示について

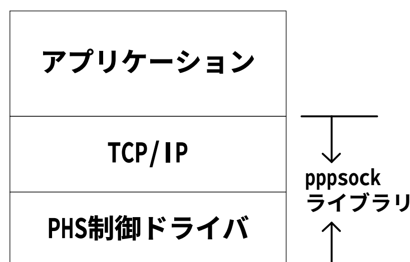
電波状態の取得は 1.2 秒毎に行っています。しかし、この値は急激な変化を繰り返すため、電波状態の判定には不適当です。そのため、本機では「平均化」と呼ばれる手法によって電波状態を判断し易い値に変換しています。

本機で使用している平均化アルゴリズムは、下図のように 1.2 秒毎に取得されるデータの 16 回平均です。但し、この 1.2 秒毎に更新される平均値は、通信モジュールの内部で使用される値であり、アプリケーション・プログラム（サイト・サーベイを含む）から取得可能な値は、この値を 6 秒毎にサンプリングした値です。これを下図で説明すると、平均値 1 の次に取得可能なのは、平均値 7 になります。



2. ソフトウェア構成

ソケットライブラリはプロトコル部、ドライバ部、共にまとめてライブラリ形式になり、アプリケーションとリンクして使用します。



[1] 実際には、安定した通信ができないのは電波強度が1に近い2のところです。しかし、このような状態を判定することはできませんので、運用に際しては3以上の強度を確保して下さい。

3. PPP **プロトコルサポート範囲**

本ライブラリでサポートする PPP の機能は以下の通りです。

認証プロトコル：PAP，CHAP，無し

MS-CHAP はサポートしません。

IPヘッダ圧縮：サポートしません

4. アプリケーション作成方法

4. 1.ソースコード作成

必要に応じて以下のファイルをインクルードします。

#include "net.h"　　　必須
#include "local.h"　　必須
#include "support.h"　必須
#include "socket.h"　　socket 使用時
#include "icmp.h"　　　ICMP プロトコル使用時

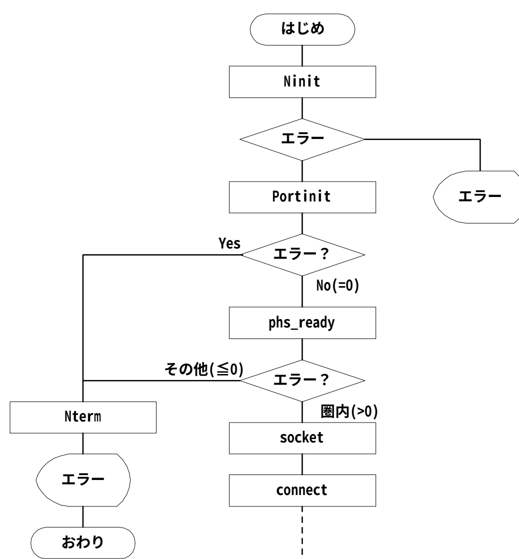
4. 2.メーク環境

DT-900 のアプリケーションの作成環境に順じますが、PPPSOCK ライブラリ(PPPSOCK.LIB)のリンクを追加してください。

4. 3.回線接続までの処理方法

4. 3.1. 通信アプリケーションの作成

Ninit／Portinit後、回線接続(connect 等)するまでの間に電波状態の監視を行い、圏内の場合のみ回線接続を行ってください。回線状態の確認には、phs_ready 関数を使用します。



4. 3.2. 電波状態を表示アプリケーションの作成

Ninit／Portinit後、正しい電波状態が計測できるまでには数秒の時間を要します。Portinit 後に phs_ready 関数を繰り返しコールするとレベル1（エラー時を除く）が返ります。接続処理はこの時点以降で可能ですが、正しい電波の状態を表示したい場合には、更に数秒間 phs_ready 関数を呼び続けて下さい。

5. 環境ファイル

環境ファイルは PPPSOCK.CFG です。A ドライブのルートに格納して下さい。Ninit 関数実行時に参照されます。このファイルを作成、編集される場合、ファイルの最後に改行のみの 1 行を挿入して下さい。

5. 1.PPP で必要となるデータ

- 相手先 TEL 番号(最大 40 桁、2 件まで)

  必須項目です。最大 2 項目の設定が可能です。

  1 件目の電話番号につながらない場合は、2 件目の記述が有れば 2 件目の電話番号にダイヤルを試みます。

- 認証形態(無し、PAP、CHAP)

  必須項目です。

- 認証ユーザ名(最大 30 文字)

  認証が PAP または CHAP の時、必須項目です。

  アプリケーションで認証ユーザ名(PPP_USER)を設定した場合、この項目内容は使用されませんが、ダミーとしてこの項目は必要です。

  ユーザー名にスペースは使用できません。

- 認証パスワード(最大 20 文字)

  認証が PAP または CHAP の時、必須項目です。

  アプリケーションで認証パスワード(PPP_PSWD)を設定した場合、この項目内容は使用されませんが、ダミーとしてこの項目は必要です。

  パスワード文字列は必ずダブルクォーテーション(“)で囲みます。(パスワードの文字列にダブルクォーテーションの使用は不可です。)

- 自 IP アドレス

  認証が PAP または CHAP の時、オプションです。

  認証が無しの時、必須項目です。

  自 IP アドレス、サブネットマスクを設定した場合、そのアドレス情報を使用します。設定しない場合、サーバより取得します。

- サブネットマスク

  自 IP アドレス項目を設定した時、必須項目です。

- ドメインネームサーバ IP アドレス

  今回は使用しません。(拡張用です。)

- 初期化 AT コマンド(最大 50 桁)

  項目名のみ必要です。値は設定する必要ありません。

- ダイヤル AT コマンド

  必須項目です。

- シリアルボーレート(38400)

  必須項目です。

5. 2.**ファイルレイアウト**

```
TEL:01-2345-6780        # 電話番号(第 1)
TEL:01-2345-6781        # 電話番号(第 2)*
AUT:2                   # 認証形態(0:無し 1:PAP 2:CHAP)
USR:USERNAME            # ユーザ名
PAS:"PASSWORD"          # パスワード
IPA:192.168.0.1         # 自 IP アドレス
SNM:255.255.255.0       # サブネットマスク
DNS:192.168.0.10        # ドメインネームサーバ IP アドレス
SAI:                    # 初期化 AT コマンド
SAD:ATD                 # ダイヤル AT コマンド
SSP:2                   #  ボーレート(固定)
```

※ "TEL:"項目は 2 個まで入力可能。

各項目の順番は自由です。

各項目は省略不可です。(2 個目の"TEL:"項目を除く。)

2 個以上同じ項目("TEL:"項目は 3 個以上)がある場合、ファイルエラーです。

必須でない項目を設定しない場合、":"の後に区切り文字を入れてください。

区切り文字はスペース、タブ、改行です。

例. 電話番号 1 件、CHAP、ユーザ名 DT900、パスワード「dt900 pass」

```
TEL:01-2345-6789        # 電話番号(第 1)
AUT:2                   # 認証形態(0:無し 1:PAP 2:CHAP)
USR:DT900               # ユーザ名
PAS:"dt900 pass"        # パスワード
IPA:                    # 自 IP アドレス
SNM:                    # サブネットマスク
DNS:                    # ドメインネームサーバ IP アドレス
SAI:                    # 初期化 AT コマンド
SAD:ATD                 # ダイヤル AT コマンド
SSP:2                   #  ボーレート(固定)
```

例 2. 電話番号 1 件、CHAP、ユーザ名とパスワードはアプリケーション設定

```
TEL:01-2345-6789          # 電話番号(第 1)
AUT:2                     # 認証形態(0:無し 1:PAP 2:CHAP)
USR:DUMMY_USER            # ユーザ名
PAS:"DUMMY_PASSWORD"      # パスワード
IPA:                      # 自 IP アドレス
SNM:                      # サブネットマスク
DNS:                      # ドメインネームサーバ IP アドレス
SAI:                      # 初期化 AT コマンド
SAD:ATD                   # ダイヤル AT コマンド
SSP:2                     #  ボーレート(固定)
```

アプリケーション

```
strcpy(PPP_USER, "MyUserName");      /* ユーザ名設定 */
strcpy(PPP_PSWD, "My Password");     /* パスワード設定 */
if (Ninit() < 0)
{
    ...
```

Ninit をコールする前に PPP_USER、PPP_PSWD を設定します。

設定しない場合、またはヌルストリング(文字長 0)を設定した場合、環境ファイルの内容が使用されます。

6. ライブラリ関数一覧

6. 1.電源 OFF/ON 発生時の処理

Portinit から Portterm(Nterm)までの間で電源 OFF/ON が発生した場合、電源 ON 後 PPPSOCK ライブラリ関数で"電源 OFF/ON あり(-20)"をリターンします。

この場合、Nterm を実行後 Ninit から処理を始める必要があります。

6. 2.回線切断発生時の処理

回線接続中に回線切断が発生した場合、発生後の PPPSOCK ライブラリ関数で"DisConnect(-21)"をリターンします。

この場合、Nterm を実行後 Ninit から処理を始める必要があります。

6. 3.ハードウェアエラー発生時の処理

Portinit から Portterm(Nterm)までの間で PPPSOCK ライブラリ関数で"ハードウェアのエラー(-13)"でリターンがあった時、Nterm を実行後 Ninit から処理を始める必要があります。

6. 4.PHS モジュールローバッテリー発生時の処理

Portinit から Portterm(Nterm)までの間で、PHS モジュールがローバッテリーを検出した場合、PPPSOCK ライブラリ関数は"LBP エラー(-24)"をリターンします。

この場合は、Nterm を実行後電池を充電して Ninit から処理を始める必要があります。

6. 5.回線接続／切断タイミング

回線接続

- Nopen
- connect
- bind
- send
- sendto
- sendmsg
- recv
- recvform
- recvmsg

回線切断

- Nterm
- Portterm

FTPput、FTPget は内部で回線接続、回線切断をおこないます。

6. 6.エラーコード

回線接続時に、受信レベルを調べて、圏外の場合はerrno2に7をセットします。

また、モデムから以下のリザルトコードが返ってきた場合は、グローバル変数 errno2 に
エラーコードをセットします。

アプリケーションは、errno2 を参照することで、回線接続に失敗した理由を知ることが出来ます。

リザルトコード／状態	errno2 の値	
CONNECT	0	
NO CARRIER	1	
BUSY	2	
NO DIALTONE	3	＊発生しない
NO ANSWER	4	
DELAYED	5	＊発生しない

6. 7.関数一覧

Ninit	テーブル・バッファの初期化
Nterm	終了
Portinit	通信ポートの初期化
Portterm	シャットダウン
Nopen	コネクションオープン
Nclose	コネクションクローズ
Nread	読み込み
Nwrite	書き込み
Nportno	ポート番号取得

accept	受動オープン待ち
bind	名前をバインド
closesocket	ソケットをクローズ
connect	コネクション開始
fcntlsocket	fcntl 制御
gethostbyname	ホスト名から IP アドレスの取得
getpeername	Peer 名の獲得
getsockname	ソケット名の獲得
getsockopt	ソケットのオプションの獲得
ioctlsocket	ioctl 制御
listen	Listen
readsocket	読み出し
recv	メッセージの受信
recvfrom	〃
recvmsg	〃
selectsocket	Select
send	メッセージの送信
sendmsg	〃
sendto	〃
setsockopt	ソケットのオプションの設定
shutdown	シャットダウン
sleepsocket	ディレイ
socket	ソケットを作成
writesocket	書き込み

FTPget	FTP 受信
FTPput	FTP 送信
phs_ready	PHS の電波受信レベル取得

int Ninit(void)

テーブルとバッファの初期化を行います。Ninit()は、最初にコールする必要があります。

引数　　無し

戻り値　0 :　　正常終了。
　　　　-1 :　　環境ファイルの記述が間違っています。

[Ninit 例]

```
main()
{
  ...
  if (Ninit() < 0)
      ...<< Process error >>
}
```

対応するシャットダウン関数は、Nterm です。

PPP ライブラリ関数を使用する場合は,必ず以下の手順で行ってください。
PPP ライブラリの初期化前及び、終了後は PPP ライブラリ関数は正常動作しません。

初期化時

Ninit(),Portinit()の順に関数コールしてください。正常終了後他の PPP ライブラリ関数が使用可能になります。

終了時

PPP ライブラリを使った後は、必ず Portterm(),Nterm()の順に関数コールを行ない、PPP ライブラリの終了処理を行ってください。

int Nterm(void)
Ninit に対応するシャットダウン関数です。

引数　　無し

戻り値　　0 :　　　正常終了

[Nterm 例]

```
Nterm();
```

アプリケーション終了時は必ずコールしてください。

int Portinit(char *port)

通信ポートの初期化処理です。

引数　char *port : "*"　必ず"*"を指定してください。

戻り値		
	0 :	初期化成功
	-11 :	パラメータエラー。
	-13 :	ハードウェアのエラーが発生、又は通信ポートが使えません。
	-20 :	電源 OFF/ON あり。
	-24 :	LBPエラー

[Portinit 例]

```
main()
{
  ...
  if (Ninit() < 0)
      ...<< process error >>
  if (Portinit("*") < 0)
      ...<< process error >>
  ...
}
```

対応するシャットダウン関数は、Portterm です。

Portinitを実行後、Porttermを実行するまでの間でディレイをおきたい場合、必ずsleepsocketを使用してください。ソフトウェアループ、キー入力待ち関数実行等をおこなう場合、TCP/IP の受信処理が動作できずに接続 NG になる場合があります。

int Portterm(char *port)

通信ポートの終了をおこないます。

引数　　char *port : "*"　　　　必ず"*"を指定してください。

戻り値　　0 :　　　　正常終了

[Portterm 例]

```
Portterm("*");
```

int Nopen(char *to, char *portoc, int p1, int p2, int flags)

コネクションをオープンする関数です。

引数

char *to : "n1.n2.n3.n4" リモートホストの IP アドレス。
"*" 受動オープン時。

char *protoc : 使用するプロトコルを設定します。"TCP/IP"、"UDP/IP"、あるいは、"ICMP/IP"です。

int p1,p2 : ローカルとリモートのポート番号。ポート番号は、2 つのポートを一致させるために使われます。
受動的オープンの場合は、p2 を 0 に、to を"*"に指定します。
受動的オープンは、何もしないで能動的オープンからメッセージが来るのを待ちます。
ポート番号(固定の意味を持つもの以外)を、すぐに再使用することは出来ません。多くのシステムは、ポートの切断処理を行う前に、かなり長い遅延時間が費やされます。同じポート番号に再コネクションを要求した場合、その要求は拒絶されるでしょう。その場合は、Nportno()サブルーチンを使って、ポート番号を取得してください。

int flags : S_NOWA は、コネクションを開始しリターンします。TCP の能動的オープンの場合、コネクションが確立されたときはリターンしますが、応答が数分たっても返されないときはタイムアウトになります。

```
if (SOCKET_ISOPEN(conno))
        CONNECTION IS OPEN;
```

受動的オープンは、永遠に待ちます。S_NOWA オプション無しの TCP の能動的オープンの場合は、コネクションが確立したときはリターンしますが、応答が数分経っても返されない場合は、タイムアウトになります。

戻り値

正の数 : コネクションが成功した場合のコネクションナンバー。
-11 : リモートホストのアクセスが不可能。
-12 : タイムアウト。
-14 : 接続先ホストがコネクションを拒否。
-20 : 電源 OFF/ON あり。
-21 : 回線切断。
-24 : LBPエラー

[Nopen 例 1]
host1から"192.168.1.1"のTCPポート1000にオープン要求を出す能動的オープンです。
ローカルポート番号は、Nportnoファンクションを使って、動的に割り当てられます。

```
host1:
int conno, myport;                              /* connection number */
...
myport = Nportno();
conno = Nopen("192.168.1.1", "TCP/IP", myport, 1000, 0);

if (conno < 0)
      ...<< process error >>
```

[Nopen 例 2]
TCPポート番号1000でコールを待つ、受動的オープンです。

```
host2:
int conno;                                      /* connection number */

conno = Nopen("*", "TCP/IP", 1000, 0, 0);
```

[Nopen 例 3]
ICMPのオープンです。

```
host1:
conno = Nopen("192.168.1.1", "ICMP/IP", 1000, 1010, 0);
```

PINGユーティリティ等で使用します。

[Nopen 例 4]
ノンブロッキングモードのOPENを行い、OPENコネクションをポーリングしている間に、何らかの処理を行います。

```
conno = Nopen("192.168.1.1", "TCP/IP", 1001, 1000, S_NOWA);
if (conno < 0)
      << ERROER >>
while(! SOCKET_ISOPEN(conno))
```

```
<< perform other processing >>
```

一般的に、受動的オープンはサーバアプリケーション、能動的オープンはクライアントアプリケーションで使用します。

int Nclose(int conno)

コネクションナンバー"conno"をクローズします。このファンクションは、コネクションテーブルがクリアされるまで待ちます。

引数　int conno :　コネクションナンバー

戻り値
- 0 :　成功
- -14 :　プロトコル関連の問題が発生しました。リモートホストにデータを書き込んでいた場合は、データは、安全な状態でないと思わなければなりません。コネクションはクローズされます。
- -16 :　コネクションナンバーが間違っている可能性があります。クローズは行われません。
- -20 :　電源 OFF/ON あり。
- -21 :　回線切断。
- -24 :　LBPエラー

[Nclose 例]

```
int conno;              /* connection number */

error = Nclose(conno);

                                /* close connection */
if (error < 0)
      << process error >>
```

アプリケーションはクローズを再試行してはいけません。クローズ後、内部的にコネクションナンバーが使用される場合があるため、アプリケーションはクローズ後にコネクションナンバーを再使用してはいけません。

int Nread(int conno, char *buff, int len)

コネクション"conno"から最長"len"サイズのメッセージを"buff"に読み込みます。

引数	int conno :	コネクションナンバー。
	char *buff :	メッセージ格納バッファへのポインタ。
	int len :	メッセージ長。

戻り値	0を含む正の数 :	リターンされた文字数。
	-12 :	タイムアウト。読み出しを再び試みます。
	-14 :	プロトコル関連の問題。アプリケーションは、コネクションをクローズする必要があります。
	-16 :	コネクションナンバーが無効です。
	-18 :	ノンブロッキングコネクションを処理することが出来ません。読み出しは、再び試されています。
	-19 :	メッセージが長すぎて、バッファに格納出来ません。
	-20 :	電源OFF/ONあり。
	-21 :	回線切断
	-24 :	LBPエラー

Nreadマクロ

readマクロを使った場合、以下で示すように、コネクションの処理の融通性を高める事が出来ます。

SOCKET_RXTOUT(conno, tout) :	コネクション"conno"に新しい読み出しのタイムアウトを tout mSEC で指定します。 リターン値はありません。
SOCKET_HASDATA(conno) :	コネクション"conno"のメッセージがあるかどうかテストします。メッセージがある場合は、ゼロ以外の値を返します。
SOCKET_TESTFIN(conno) :	コネクション"conno"がクローズされたかテストします。

[Nread 例]

```
/* user defined input buffer size */
#define MAX_BUFFER_SIZE 80 ...

int error;                              /* error code */
int conno;                              /* connection number */
char buff[MAX_BUFFER_SIZE];             /* data input buffer */
...
/* read data into "buff" from connection number "conno" */
error = Nread(conno, buff, sizeof(buff));
if (error < 0)
      << process error >>
```

int Nwrite(int conno, char *buff, int len)
"buff"にあるサイズ"len"バイトのメッセージを、コネクション"conno"に書き込みます。

引数		
	int conno :	コネクションナンバー。
	char *buff :	書き込み先のバッファのポインタ。
	int len :	メッセージ長。(最大長は SOCKET_MAXDAT 参照)

戻り値		
	正の数 :	正常終了
	-12 :	タイムアウト。ブロッキングモードの TCP の場合、もう一方のポートが肯定応答を正しく送信しなかった可能性があります。又、システムに大量のロードが行われた為に、肯定応答を受け取る前に、タイムアウトになった可能性があります。コネクションをクローズする必要があります。ノンブロッキングモードの場合は、書き込みが再度試されます。
	-14 :	プロトコル関連の問題。アプリケーションはコネクションをクローズする必要があります。
	-16 :	コネクションナンバーが無効です。
	-19 :	メッセージが長すぎて、バッファに格納出来ません。
	-20 :	電源 OFF/ON あり。
	-21 :	回線切断
	-24 :	LBPエラー

Write のエラーは Nclose で報告されます。従って、常に Nclose のステータスを確認しなければなりません。
Nwrite は、ストリーム I/O はおこないません。データの最長サイズは、バッファサイズ MAXBUF から必要なヘッダサイズを引いた値になります。

Nwrite マクロ

SOCKET_MAXDAT(conno) :	コネクション"conno"のレコードの最長サイズを返します。
SetTxtout(sec):	MAXTXTOUT(TCP 送信タイムアウト値)を設定します。
short GetTxtout():	MAXTXTOUT(TCP 送信タイムアウト値)を取得します。

[Nwrite 例 1]
```
#define MAX_BUFFER_SIZE 80            /* user defined input buffer size
...
int conno;                            /* connection number */
char buff[MAX_BUFFER_SIZE];           /* data input buffer */

/* write data stored in "buff" to open connection "conno" */
error = Nwrite(conno, buff, sizeof(buff));
```

```
if (error < 0)
        << process error >>
```

unsigned short Nportno(void)

ポート番号取得

引数　　無し

戻り値　　ポート番号

int accept(int s, struct sockaddr *name, int *namelen)

accept は、能動オープンの要求を待ち、新しいソケット番号をリターンします。オリジナルのソケットは、どのような場合も変化しません。

引数	int s :	ソケット番号。
	struct sockaddr *name :	name 構造体へのポインタ。
	int *namelen :	name 構造体のサイズへのポインタ。

戻り値	正の数 :	ソケット番号。
	-1 :	失敗。
	-20 :	電源 OFF/ON あり。
	-21 :	回線切断
	-24 :	LBPエラー

int bind(int s, struct sockaddr *name, int namelen)

ソケットは、socket コールで作成されています。bind を、能動的オープン(connect)の前に使うことは出来ますが、その必要性はありません。ただし、accept の前には必要です。

引数	int s :	ソケット番号。
	struct sockaddr *name :	name 構造体へのポインタ。
	int namelen :	name 構造体のサイズ。

戻り値	0 :	成功
	-1 :	失敗
	-20 :	電源 OFF/ON あり。
	-21 :	回線切断
	-24 :	LBPエラー

int closesocket(int s)

ソケットをクローズします。

closesocketは、UNIXの場合は、closeです。ファイルのopne/closeと区別するためclosesocketになっています。

引数	int s :	ソケット番号

戻り値	0 :	成功
	-1 :	失敗
	-20 :	電源OFF/ONあり。
	-21 :	回線切断
	-24 :	LBPエラー

int connect(int s, struct sockaddr *name, int namelen)

能動的に接続します。

ソケットは、socket コール時に作成されています。name 構造体は、ポート番号および Internet アドレスを指定するのに使われます。

引数	int s :	ソケット番号
	struct sockaddr *name :	name 構造体へのポインタ
	int namelen :	name 構造体のサイズ

戻り値	0 :	成功
	-1 :	失敗
	-20 :	電源 OFF/ON あり。
	-21 :	回線切断
	-24 :	LBPエラー

```
struct sockaddr {                /* generic socket address */
    unsigned short  sa_family;  /* address family */
    char            sa_data[14];/* up to 14 bytes of direct address */
};
```

```
int fcntlsocket(int s, int cmd, int arg)
```

ソケットのノンブロッキング属性を取得、設定します。

引数

int s :	ソケット番号		
int cmd :	ネットワークコマンド		
		F_GETFL	フラグを取得します。
		F_SETFL	フラグを設定します。
int arg :	パラメータ(F_SETFL のとき有効)		
		O_NDELAY	ノンブロッキング
		0	非ノンブロッキング

戻り値

SETFL 0 :	成功
GETFL フラグの現在の値 :	成功
-1 :	失敗
-20 :	電源 OFF/ON あり。
-21 :	回線切断。
-24 :	LBPエラー

```
struct hostent *gethostbyname(char *name)
```

名前から IP アドレスを取得します。

このルーチンで使われる名前は"ホスト名"です。

引数	char *name :	ホスト名へのポインタ

戻り値	構造体のアドレス :	成功
	0 :	失敗
	-20 :	電源 OFF/ON あり。
	-21:	回線切断
	-24 :	LBPエラー

回線接続中に使用できます。

ホストの IP アドレスは、構造体"hostent"に入れられます。

ホストの IP アドレスを、構造体"hostent"から取得する場合のコード例として、以下のコードがあります。

```
memcpy((char *)&socksav.sin_addr, (char *)hostentp->h_addr_list[0], 4);
```

```
int getpeername(int s, struct sockaddr *name, int *namelen)
```

通信先のホストの情報を取得します。

リモートアドレスを構造体"name"(リモート Internet アドレスおよびポート番号)に設定します。

受信処理終了後にコールしてください。

引数		
	int s :	ソケット番号
	struct sockaddr *name :	name 構造体へのポインタ
	int *namelen :	name 構造体のサイズ

戻り値		
	0 :	成功
	-1 :	失敗
	-20 :	電源 OFF/ON あり。
	-21 :	回線切断。
	-24 :	LBPエラー

int getsockname(int s, struct sockaddr *name, int *namelen)

自ホストの情報を取得します。

ローカル Internet アドレスおよびポート番号を構造体"name"に設定します。

引数	int s :	ソケット番号
	struct sockaddr *name :	name 構造体へのポインタ
	int *namelen :	name 構造体のサイズへのポインタ。

戻り値	0 :	成功
	-1 :	失敗
	-20 :	電源 OFF/ON あり。
	-21 :	回線切断。
	-24 :	LBPエラー

```
int setsockopt(int s, int level, int optname, char *optval, int optlen)
int getsockopt(int s, int level, int optname, char *optval, int *optlen)
```

ソケットオプションをセットします。

ソケットオプションを操作するルーチン

レベル	IP_OPTIONS	意味
IPPROTO_IP	IP_OPTIONS	IP ヘッダのオプション
IPPROTO_TCP	TCP_NODELAY	送信を遅らせない
SOL_SOCKET	SO_BROADCAST	ブロードキャストを許可
	SO_DEBUG	デバッグフラグ
	SO_DONTROUTE	ルーティング抜き
	SO_KEEPALIVE	ブローブのキープアライブ
	SO_LINGER	クローズを延ばす
	SO_OOBINLINE	URG データをインラインのまま残す
	SO_RCVBUF	バッファサイズを獲得する
	SO_SNDBUF	バッファサイズを送信する
	SO_REUSEADDR	ローカルアドレスの再使用

戻り値		
	0 :	正常終了
	-1 :	エラー
	-20 :	電源 OFF/ON あり。
	-21 :	回線切断。
	-24 :	LBPエラー

int setsockopt(int s, int level, int optname, char *optval, int optlen)

ソケットオプションを設定します。

引数	int s :	ソケット番号
	int level :	レベル
	int optname :	IP_OPTIONS
	char *optval :	オプション
	int optlen :	オプション

戻り値	0 :	正常終了	
	-1 :	エラー(errno にそのエラーを示す以下の値が設定されます。)	
		EBADF(-16)	s が有効な記述子ではありません。
		EFAULT(-17)	optval が指すアドレスは、プロセス・アドレス空間の有効部分にありません。(パラメータエラー)
		ENOPROTOOPT(-53)	オプションは指定されたレベルでは認識されていません。
	-20 :	電源 OFF/ON あり。	
	-21 :	回線切断。	
	-24 :	LBPエラー	

int getsockopt(int s, int level, int optname, char *optval, int *optlen)

setsockopt()でセットした値を取得します。

引数	int *optlen :	オプション
	その他の引数は、上記を参照してください。	

戻り値　上記を参照してください。

int ioctlsocket(int s, int request,...);

ソケットインターフェースの動作を指定します。

オプションの三番目の引数は、結果のポインタとして使われます。このファンクションが BSD ソケットで使われる方法と違いがある場合があります。二番目の引数は、符号無し long 型を指定することが出来ます。可変引数は、非 ANSI C では違う方法で処理されます。

引数	int s :	ソケット番号
	int requset :	FIONBIO 　　ノンブロッキングモードの設定 FIONREAD 　　受信待ちデータサイズ取得 SIOCATMARK 　　帯域外フラグ取得

戻り値	0 :	その他	
	-1 :	エラー（errno にそのエラーを示す以下の値が設定されます。）	
		EBADF(-16)	s が有効な記述子ではありません。
		EFAULT(-17)	request は arg が指すバッファへ／からのデータ伝送を必要としますが、そのバッファの一部はプロセスに割り当てられた空間の範囲外にあります。（パラメータエラー）
	-20 :	電源 OFF/ON あり。	
	-21 :	回線切断。	
	-24 :	LBPエラー	

int listen(int s, int backlog)

受動オープンが行われる事を指定します。

引数 int s : ソケット番号

int backlog : 受動オープン待ちの最大コネクション数

（実装では最大ソケット数:8 で固定されています）

戻り値		
	0 :	正常終了
	-1 :	エラー（errno にそのエラーを示す以下の値が設定されます。）
		EBADF(-16)　s が有効な記述子ではありません。
	-20 :	電源 OFF/ON あり。
	-21 :	回線切断。
	-24 :	LBPエラー

```
int recv(int s, char *buf, int len, int flags)
int recvfrom(int s, char *buf, int len, int flags, struct sockaddr *from, int *fromlen)
int recvmsg(int s, struct msghdr *msg, int flags)
int readsocket(int s, char *buf, int len)
```

ソケットからメッセージを受け取る BSD ルーチンです。
recv は、コネクション型で使われ、残りの二つはコネクションレス型でも使用する事が出来ます。
recvmsg は、UDP 用の関数です。

引数
- int s : ソケット番号
- char *buf : バッファへのポインタ
- int len : バッファサイズ
- int flags : フラグ
  - MSG_OOB ソケット上にある「帯域外」のデータの受信をします。
  - MSG_PEEK データを取りますが、再読み出しができるように、データをそのまま残します。
- struct sockaddr *from : メッセージ発信者の sockaddr へのポインタ
- int *fromlen : from のサイズへのポインタ
- struct msghdr *msg : msghdr へのポインタ

戻り値
- 正の数 : 成功したバイト数。
- -1 : エラー
- -20 : 電源 OFF/ON あり。
- -21 : 回線切断
- -24 : LBPエラー

構造体"msghdr"

```
struct msghdr {                          /* Message header for recvmsg and sendmsg. */
    char           *msg_name;            /* optional address */
    int             msg_namelen;         /* size of address */
    struct iovec   *msg_iov;             /* scatter/gather array */
    int             msg_iovlen;          /* # elements in msg_iov */
    char           *msg_accrights;       /* access rights sent/received */
    int             msg_accrightslen;    /* access rights length */
};
struct iovec{                            /* address and length */
      char *iov_base;                    /* base */
      int iov_len;                       /* size */
};
```

マクロ

```
#define readsocket(s, buf, len) recv(s, buf, len, 0);
```

int selectsocket(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceprfds, struct timeval *timeout)

ソケットの状態をチェックします。

引数	int nfds :	対象となるソケット番号
	fd_set *readfds :	データを獲得するソケットへのポインタ
	fd_set *writefds :	データを送信するソケットへのポインタ
	fd_set *exceprfds :	緊急データを獲得するソケットへのポインタ
	struct timeval *timeout :	タイムアウト値へのポインタ ゼロポインタは、無制限のタイムアウトを意味します。

戻り値	正の数 :	準備出来ているソケット数
	-1 :	エラー
	0 :	タイムアウト
	-20 :	電源 OFF/ON あり。
	-21 :	回線切断。
	-24 :	LBPエラー

```
struct timeval{                    /* time-out format for select */
      long tv_sec;                 /* seconds */
      long tv_usec;                /* microseconds */
};
```

fd_set は、以下のマクロを使用して操作することが出来ます。

FD_ZERO(&fd_set)	ソケットリストをクリアします。
FD_SET(s, &fd_set)	ソケット s を追加します。
FD_CLR(s, &fd_set)	ソケット s を削除します。
FD_ISSET(s, &fd_set)	s が含まれた場合は、ゼロ以外になります。

```
int send(int s, char *buf, int len, int flags)
int sendto(int s, char *buf, int len, int flags, struct sockaddr *to, int tolen)
int sendmsg(int s, struct msghdr *msg, int flags)
int writesocket(int s, char *buf, int len)
```

ソケットにメッセージを送る BSD ルーチンです。

send は、コネクション型で使われ、残りの二つは、コネクションレス型でも使用することが出来ます。

sendmsg は、UDP 用の関数です。

引数		
	int s :	ソケット番号
	char *buf :	データバッファへのポインタ
	int len :	データバッファのサイズ
	int flags :	フラグ MSG_OOB 「帯域外」のデータを送信します。 MSG_DONTROUTE ルーティング抜きで送信します。
	struct sockaddr *to :	sockaddr 構造体の to へのポインタ
	int tolen :	to のサイズ
	struct msghdr *msg :	msghdr 構造体の msg へのポインタ

戻り値		
	正の数 :	送信されたバイト数
	-1 :	失敗
	-20 :	電源 OFF/ON あり。
	-21 :	回線切断
	-24 :	LBPエラー

マクロ

```
#define writesocket(s, buf, len) send(s, buf, len, 0);
```

int shutdown(int s, int how)

ソケットのシャットダウンします。

本実装では、受信のシャットダウンをサポートしていません。

引数		
	int s :	ソケット番号
	int how :	0 受信をシャットダウンします。 1 送信をシャットダウンします。TCP の場合は、FIN を送ります。 2 受信、送信の両方をシャットダウンします。

戻り値		
	0 :	成功
	-1 :	失敗
	-20 :	電源 OFF/ON あり。
	-21 :	回線切断
	-24 :	LBPエラー

void sleepsocket(int sec)

秒単位のディレイです。

引数　　int sec :　　　　　　ディレイする秒(0も可 : キー入力待ち等に使用します。)

戻り値　　無し

Portinitを実行後、Porttermを実行するまでの間でディレイをおきたい場合、必ずsleepsocketを使用してください。ソフトウェアループ、キー入力待ち等を行うと、TCP/IPの受信処理が動作できずに接続NGとなる場合があります。

```
int socket(int domain, int type, int protocol)
```

ソケットを作成します。socket ルーチンでは、実際のネットワーク操作は行われません。
本実装では type に SOCK_RAW はサポートしません。

引数	int domain :	PF_INET	TCP/IP および関連プロトコル。
	int type :	SOCK_STREAM	ストリームソケット=TCP/IP
		SOCK_DGRAM	データグラムソケット=UDP/IP
	int protocol :	0 固定	

戻り値	正の数 :	ソケット番号
	-1 :	失敗
	-20 :	電源 OFF/ON あり。
	-21 :	回線切断。
	-24 :	LBPエラー

```
int FTPget(char *host, char *file, int mode)
```

FTP プロトコルでファイルを受信します。

引数	char *host :	サーバホスト名。"192.168.1.1" など。
	char *file :	"受信ファイル名" "ローカルファイル名 サーバホストのファイル名 "
	int mode :	テキストファイルの場合は ASCII バイナリファイルの場合は IMAGE

戻り値	0 :	成功
	-1 :	失敗
	-20:	電源 OFF/ON あり。
	-21:	回線切断
	-24 :	LBPエラー

サーバホストにログインしてファイル受信後、ログアウトします。
ユーザ名、パスワードは変数 userid, passwd にそれぞれセットしておく必要があります。

例

```
extern char userid[];
extern char passwd[];

strcpy(userid, "DT900");
strcpy(passwd, "casiocasio");

error = FTPget("192.168.1.1", "test1", ASCII);
```

ホスト(192.168.1.1)からファイル(test1)をテキストモードで受信します。ローカルファイル名は test1 になります。

```
error = FTPget("192.168.1.1", "test2 test3", IMAGE);
```

ホスト(192.168.1.1)からファイル(test3)をバイナリモードで受信します。ローカルファイル名は test2 になります。

int FTPput(char *host, char *file, int mode)

FTP プロトコルでファイルを送信します。

引数	char *host :	サーバホスト名。"192.168.1.1" など。
	char *file :	"送信ファイル名"
		"ローカルファイル名 サーバホストのファイル名 "
	int mode :	テキストファイルの場合は ASCII
		バイナリファイルの場合は IMAGE

戻り値	0 :	成功
	-1 :	失敗
	-20:	電源 OFF/ON あり。
	-21:	回線切断
	-24 :	LBPエラー

サーバホストにログインしてファイル送信後、ログアウトします。

ユーザ名、パスワードは変数 userid, passwd にそれぞれセットしておく必要があります。

例

```
extern char userid[];
extern char passwd[];

strcpy(userid, "DT900");
strcpy(passwd, "casiocasio");

error = FTPput("192.168.1.1", "test1", ASCII);
```

ホスト(192.168.1.1)にローカルファイル(test1)をテキストモードで送信します。サーバホストの受信ファイル名は test1 になります。

```
error = FTPput("192.168.1.1", "test2 test3", IMAGE);
```

ホスト(192.168.1.1)にローカルファイル(test2)をバイナリモードで送信します。サーバホストの受信ファイル名は test3 になります。

int phs_ready(void)

PHS の受信レベルを取得します。

引数　　無し

戻り値		
	0 :	圏外
	1 :	圏内(レベル1)
	2 :	圏内(レベル2)
	3 :	圏内(レベル3)
	4 :	圏内(レベル4)
	-1:	エラー(Ninit／Portinitを行っていない、又は回線接続状態)
	-20:	電源 OFF/ON あり
	-24 :	LBPエラー

Ninit／Portinit後、回線接続するまでの間に使用します。

回線接続前に電波圏内にある事を確認してから回線接続してください。