PIC C Compiler Reference Mamnual

PIC C コンパイラー

日本語リファレンス　マニュアル

バージョン **3 -** Revision 3.6

Version 3

July 2003

Custom Computer Services Inc.

P.O.BOX 2452

Broolfield, WI 53008

E-mail: ccs@ccsinfo.com

URL: http://www.ccsinfo.com

Custom Computer Services

日本総代理店

有限会社 データ ダイナミクス

〒579-8062　東大阪市上六万寺町 13-10

TEL:　0729-81-6332

FAX:　0729-81-6085

E-mail: support@datadynamics.co.jp

URL: http://www.datadynamics.co.jp

コンパイラーの概要

PCB,PCM と PCH の概要

PCB,PCM と PCH コンパイラーはそれぞれ別のコンパイラーです。
PCB はマイクロチップテクノロジー社 12bit オペコード用で PCM は 14bit オペコード用、そして、PCH は 16bit PIC18 用となっております。このリファレンス・マニュアルでも多くの点で共通していますが、機能と制限はそれぞれのコントローラーに対応しています。るコードは異なりますが、同じユーザー・インターフェースを持ち同じ操作性を持っています。マニュアルも本書で統一されていますが、一部組み込み関数やプリプロセッサーに違いがありますが、異なる部分については都度明記されています。また、PIC の内蔵ペリフェラルを意識した組み込み関数を多数内蔵しています。このため、既存の C コンパイラーとは少し異なる言語体系を持っていますが、すでに C 言語を習得されているユーザーは大きな戸惑いもなく開発を行って頂けます。
コンパイラーを含む統語開発環境は、迅速にこれらのコントローラに対する読みやすくてハイレベルなアプリケーション開発環境を提供します。

このコンパイラーは他のＣコンパイラーにはない制限が多少あります。ハードウェアの制限も多くの古典的なＣコンパイラーとは異なっているかもしれません。一例を上げると、関数内で定数へのポインターは使えないなどです。これはハードウェア上でコードとデータセグメントが分離されていてコードエリアからデータにアクセスすることができないからです。しかし、コンパイラーはプロセッサーのハードウェアの制限をよく知っていて、どのようにユーザー・アプリケーションのアルゴリズムを実装すればよいか知っています。コンパイラーは大変効率的な入出力操作を行ない、ビット操作を行なえます。

テクニカル・サポート

内容等に対してのお問合せについては販売店までご連絡下さい。
動作不良、不明点などをご連絡いただきます際には次のような内容を電子メールでお問い合わせください。お問い合わせの場合、その内容などによっては即答できかねる場合もございますのでご了承ください。

- インストール環境
  - ・インストールされているマシン名、メモリ容量、CPU 名など *必要な場合のみ
  - ・AUTOEXEC.BAT、CONFIG.SYS の内容 *必要な場合のみ
  - ・ご使用中の OS の名称とバージョン
- ターゲット・プロセッサー
- 動作内容
  - ・ソースファイルなどが必要になる場合があります。
  - ・できるだけ詳しくご連絡ください。

尚、ターゲットの動作やハードウェアの内容、アプリケーション・プログラムの内容や C 言語、OS など本プログラムと直接関係ない内容についてはお答えいたしかねる場合がございます。また、本プログラムをインストール、実行したときのいかなる障害や出力ファイルの結果などについては責任を負いかねますのでご了承下さい。

ご購入後、ユーザー登録されました時点から 30 日以内のバージョンアップは無料で行われます。それ以降のプログラムのアップデートは、必要に応じて有料で行なわれます。

技術的なサポートはメール又は、FAX のみにて受け付けております。
support@datadynamics.co.jp
FAX: 0729-81-6085
*ファイルが必要な場合は FAX での長いファイルは送らないで下さい。

動作環境

PCB,PCM,PCH,PCW は IBM-PC/AT、IBM-PC 互換機上で動作する 32 ビットのソフトウエアです。Windows95/98/2000/NT/XP 上で動作します。

動作確認が取れている環境はおおよそ次のようなものです。

PCB,PCM,PCH コマンド・ライン・コンパイラー　PCB,PCM,PCH
- ・386、486、Pentium プロセッサー
- ・4M バイト以上のメモリ、HDD、CD-ROM ドライブ
- ・VGA モニタ、マウス
- ・MS-Windows95/98SE/2000/NT/XP

PCW,PCWH Windows 版コンパイラー
- ・386、486、Pentium プロセッサー
- ・8M バイト以上のメモリ、HDD、CD-ROM ドライブ
- ・マウス
- ・MS-Windows95/98SE/2000/NT/XP

PCB,PCM,PCH は DOS 環境で動作いたしますが、このマニュアルで触れられている DOS 環境は**英語 DOS** を示しています。DOS 環境で英語モードにするには下記の通りおこなって下さい。
日本語 MS-DOS（日本語 PC-DOS）の日本語モードでは動作致しません。
コマンドライン・コンパイラーでご使用になられる場合は日本語 DOS モードでも問題はありませんが DOS-IDE 環境をご使用になられる場合は、必ず**英語モード**から起動してください。
また、日本語モードから英語モードに切り替えた場合でも、コンベンショナルメモリ不足などでそのままの環境で起動できない場合もあります。このような場合はできる限り不要なデバイス・ドライバーや常駐プログラムを削除したり、UBM、XMS などに追い出したりしてメモリなどを確保してください。マルチコンフィグレーションによってピュアな英語モードでの起動を行われるとより確実です。コマンドライン・コンパイラーをご使用になられる場合は、別途、テキスト・エディターなどソースファイル作成環境が必要です。

PCW をイントールするには、Windows 95/98/2000/NT/XP が確実に動作している環境をご用意ください。日本語 Windows の環境でご使用いただけます。

インストール

PCB,PCM と PCH プログラムのインストール：

ディスクの 1 枚目をフロッピー・ドライブに挿入して、ウィンドウズのスタートの”ファイル名を指定して実行”で次の様にタイプしてください。

A:¥SETUP

PCW/PCWH のインストール：

CD-ROM を挿入し、PCW Installation をクリックしますとインストールが開始されます。

コマンドライン・コンパイラーの呼び出し：

下記のコマンドでコマンドライン・コンパイラーを呼び出します・：

CCSC option cfilename

有効オプション：

+FB	PCB(12bit)選択	-D	デバッグ・ファイルを作成しない
+FM	PCM(14bit)選択	+DS	標準.COD 形式デバッグ・ファイル
+FH	PCH(PIC18)選択	+DM	.MAP 形式デバッグ・ファイル
+FS	PCS(SX)選択	+DC	拡張 .COD 形式デバッグ・ファイル
+ES	標準エラー・ファイル	+EO	古いエラー・ファイル形式
+T	コール・ツリー作成(.TRE)	-T	コール・ツリーを作成しない
+A	スタッツ・ファイル作成(.STA)	-A	スタティクス・ファイルを作成しない
+EW	警告メッセージを表示	-EW	警告メッセージを隠す(+EA と使用)
+EA	すべてのエラー・メッセージと警告を表示	+E	最初のエラーのみを表示
+Yx	オプティマイズ・レベル x (0-9)		

下記の xxx はオプションです。

もし、インクルードされますと、ファイル拡張子をセット:

+LNxxx	ノーマル・リスト・ファイル	+O8xxx	8bit Intel HEX 出力ファイル
+LSxxx	MPASM 形式リスト・ファイル	+OWxxx	16 bit Intel HEX 出力ファイル
+LOxxx	古い MPASM リスト・ファイル	+OBxxx	バイナリー出力ファイル
+LYxxx	シンボリック・リスト・ファイル	-0	オブジェクト・ファイルを作成しない
-L	リスト・ファイルを作成しない		

+P	コンパイル後にコンパイル・ステータス・ウィンドウをキープ
+Pxx	コンパイル後に xx seconds に対するステータス・ウィンドウをキープ
+PN	エラーがないときのみステータス・ウィンドウをキープ
+PE	エラーが有るときのみステータス・ウィンドウをキープ
+Z	コンパイル後にスクラッチとデバッグ・ファイルをディスクにキープ
-Z	コンパイラー・スクラッチ・ファイルを保持しない
+DF	COFF デバッグ・ファイル

I=+"..." パス・リストがカレント・リストに加えらることを除いては I="..."T と同じです。

I="..." インクルード・ディレクトリー・サーチ・パスをセット、例えば：

I="c:¥picc¥examples;c:¥picc¥myincludes"

I が無い場合は、コマンド・ラインで.PJT ファイルはインクルード・ファイルのパスを指定するのに使用されます。

-P	コンパイルが完了した後、コンパイル・ウィンドウを閉じる
+M	シンボル・ファイル(.SYM)を作成
-M	シンボル・ファイルを作成しない
+J	プロジェクト・ファイル(.PJT)を作成
-J	PJT ファイルを作成しない
+ICD	ICD で使用するためにコンパイル
+?	ヘルプ・ファイルを呼び出す

-?　　+?と同じ

#xxx="yyy"　　yyy で id xxx に対するグローバル #define をセット、例えば:
#debug="true"
+Gxxx=”yyy”　#xxx="yyy" と同じ

+STDOUT STDOUT にエラーを出力(サード・パーティのエディターのために使用されます。)
+SETUP　CCSC を MPLAB にインストール(コンパイルはされません。)
+V　　コンパイラー・バージョンを表示(コンパイルはされません。)
+Q　　データベース内の全有効デバイスを表示(コンパイルはされません。)

キャラクター / を+ に置き換えて使用することが出来ます。デフォルト・オプションは
下記になります。:

+FM +ES +J +DC +Y9 ?T ?A +M +LNIst +O8hex ?P -Z

もし、@ファイル名が CCSC コマンド・ラインにありますと、コマンド・ライン・
オプションは指定されたファイルから読み込まれます。パラメータはファイルの複数行に
表わされます。

もし、CCSC.INI が CCSC.EXE と同じディレクトリーにありますと、コマンド・ライン・
パラメータは、コマンド・ラインで実行される前に、そのファイルから読み込まれます。

例：　　CCSC +FM C:¥PICSTUFF/TEST.C
CCSC :FM +P +T TEST.C

MPLAB 統合環境
MPLAB 5:
MPLAB がコンパイラーの前にインストールされていますと、MPLAB との統合は自動でおこなわれます。そうでない場合は、次のコマンドを使用して下さい。

CCSC +SETUP

MPLAB 6:
MPLAB 6 で CCS C コンパイラーを使用するには、その前にプラグイン・プログラムを実行してください。 もし、このプラグインがご使用の MPLAB のバージョンに無い場合は CCS の web サイトからダウンロードしてください。
一般的には MPLAB 6.x 以上をお使いの場合は mplab6.exe をインストールする必要があります。

MPLAB からのコンパイルや実行のための特定のインストラクションはバージョンによりことなります。一般的にはプロジェクトを作成するときに Tool Suite として CCS C コンパイラーを選択の上通常の MPLAB のインストラクションに従ってください。

http://www.microchip.comから最新のバージョンをダウンロードして下さい。

ディレクトリー
コンパイラーは次のディレクトリーに対してインクルード・ファイルをサーチします。

- コマンド・ラインにリストされているディレクトリー
- .PJT ファイルで指定されているディレクトリー
- ソース・ファイルと同じディレクトリー

デフォルトでは、コンパイラー・ファイルは C:¥Programe Files¥PICC に、そして、サンプル・プログラムと全てのインクルード･ファイルは C:¥Program File¥PICC¥EXAMPLES に置かれます。

コンパイラー自身は DLL ファイルです。DLL ファイルはデフォルトでは C:¥Program File¥PICC¥DLL の DLL ディレクトリーにあります。古いバージョンではこのディレクトリーはリネームされています。

ファイル・フォーマット
コンパイラーは 8bit/16bit のヘキサ・ファイルとバイナリー・ファイルのいずれかを出力します。また、2 つのリスティング・フォーマットを利用できます。これらは標準的なマイクロチップ社のツールやサードパーティ製の各種ツールとの整合がとられています。デバッグ・ファイルは、マイクロチップ社の **.COD ファイル**又は、アドバンスド・トランスデータ社の **.MAP ファイル**のいずれかを出力できます。
全てのファイル形式と拡張子は DOS IDE のオプション|ファイル(Option|file)と Windows IDE ではコンパイラー｜オプション(Compiler|Options)のから設定します。

ダイレクト・デバイス・プログラミング
IDE 環境のメイン・メニュー・バーにプログラム・オプションがあります。
IDE からデバイス・プログラマーのプログラムを呼び出すことができます。このコマンドをオプション｜プログラム(Option|Programmer Option)ウィンドゥで指定しておきます。
%H はヘキサ・ファイル名に置き換えられ、そして、%D はデバイス番号に置き換えられます。
プログラムから IDE へ戻る前に一時停止を行なわたいときは、最後に **！** を付けて下さい。
このオプションではコマンドによりプログラムのみを呼び出すことが出来ます。

デバイス・カリブレーション・データ
マイクロチップのデバイスのいくつかは出荷時にプログラム・エリアの一部にキャリブレーション・データが書き込まれています。これらはデバイス毎に異なったユニークなデータとなっています。開発中にはこれらのデータが必要となる場合があります。

UV 消去型のデバイス（セラミック・パッケージに窓のついたデバイス）はデータを消したときこのキャリブレーション・データも消去されます。このような場合、プリプロセッサーの＃ROM ディレクティブにより再プログラミングすることができます。

PCW パッケージにはこの手順をサポートするユティリティが付属しています。
新しいデバイスを購入しデバイスを読み込んで HEX ファイルに落とします。
そしてメニューのからツール|エクストラクト・キャリブレーション・データ[**Tools|Extract Cal Data**]を実行、そして、このデバイスのための名前(.C)を選択します。ユティリティはこのパーツに対する正しい#ROM ディレクティブを持った指定した名前でインクルード・ファイルを作成します。プロトタイプの開発中は #Include ディレクティブを追加し、そして、プログラムされる各パーツ#を作成する前に名前を変更して下さい。

OTP などデバイス・キャリブレーションデータの書き込みが必要ないデバイスの場合は、単にこの #Include をコメントアウトしておけばいいだけです。

ユティリティ・プログラム

SIOW
SIOW は、単なるターミナル・プログラムです。PIC プロセッサで RS232C などのアプリケーションを作成されたとき、ターミナルとしてご使用いただけます。
このユティリティは表示できない文字を HEX で表示させることができます。

DEVEDIT
DEVEDIT は、PCW のみに付属しているデバイス・データベース編集ユティリティです。
PCW コンパイラーは、コンパイル時にいつも指定されたデバイスのデータベースを参照しています。このユティリティはデバイスを追加したり変更したり削除したりできます。

デバイスを追加する場合は、ベースとなるようなデバイスを選択して追加ボタン（ADD）をクリックします。編集の場合は編集したいデバイスをリストから選択して編集ボタン（EDIT）をクリックします。
削除する場合も削除したいデバイスを選択して削除ボタン（DELETE）をクリックします。

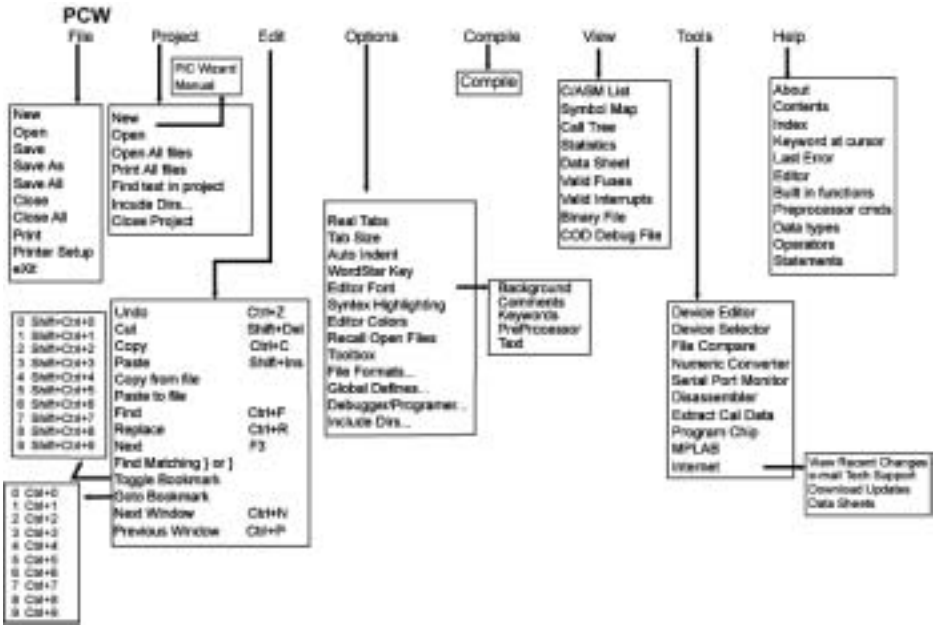
【注意】デバイス・データベースを不要に削除、変更、追加を行うとコンパイル時に検出されるべきエラー（メモリ容量など）や新規プロジェクト作成時のデータなどコンパイラが正常な動作ができなくなります。
【注意】デバイス・データベースの編集は、十分デバイスのスペックをご理解された上で行ってください。

PCONVERT
PCONVERT は、PCW のみに付属している Windows ユティリティで様々なデータタイプを他のデータタイプに変換します。例えば、浮動小数点を 4 バイト HEX に変換します。
このユティリティは変換するのに小さなウィンドウを開きます。このウィンドウは
PCW 又は、MPLAB を使用中はアクティブになっています。
デバッグなどにご利用ください。

CCSC +Q
これはコンパイラー・データベースの中のすべてのデバイスをリストします。

CCSC +FM +V
これはコンパイラーのバージョンを表示します。他のコンパイラーの場合は、+FM を +FB 又は、+FH に置き換えてください。

PCW IDE



ファイル・メニュー

New[新規]　　新規ファイルの作成

Open[開く]　エディターにファイルをオープンするします。他のファイルがオープンされていないときは、このオープンされたファイルに対してプロジェクト名が設定されます。Ctrl-O がショートカットです。

Reopen[再オープン]　最近使用されたファイルをリストにし、そのファイルを選択することでオープンできます。

Save[保存]　　編集時に現在選択されているファイルが保存されます。

Save As[名前を付けて保存]　　現在選択されているファイルを保存するためにファイル名を付けて保存します。

Save All[すべてを保存]　　すべてのオープンされたファイルが保存されます。

Close[閉じる]　　編集のために開かれているファイルを閉じます。PCW では１つのファイルがオープンされている場合、他のプログラムを編集のために開くことは出来ません。Shift+ F11 がショートカットです。

Close All[全てのファイルを閉じる]　全てのファイルを閉じます。

Print[印刷]　現在選択されているファイルを印刷

Printer Setup[プリンター設定]　　プリンターの設定

Exit[終了]　　PCW を終了

プロジェクト・メニュー

New[新規]　　新規プロジェクトを作成します。プロジェクトは手動又は、ウィザードを使って作成することが出来ます。 手動で作成した場合は .PJT ファイルのみが作成されます。他の C メイン・ファイルは指定しなければいけません。ウィザードではユーザーがプロジェクトのパラメータを指定し、.C, .HT と .PJT ファイルが作成されます。

New｜PICWIZARD[新規プロジェクト・ウィザード]

このコマンドはブランク形式になっており、RS232 I/O と I2C の特性、タイマー・オプション、使用するインターラプト、A/D オプション、必要なドライバーやピン名等をフォームに指定することで新しいプロジェクトを作成することが出来ます。

Open[開く] 指定された .PJT ファイルとメイン・ソースがロードされます。

Open All File[すべてのファイルを開く]
指定された .PJT ファイルとプロジェクトで使用される全てのファイルがオープンされます。このためにはインクルート・ファイルがわかるようにプログラムがコンパイルされていなければいけません。

Reopen[再オープン] 最近使用されたファイルをリストにし､そのファイルを選択することでオープンできます。

Find Text in Project[プロジェクトでテキストを検索]は与えられたテキストの文字列に対して、プロジェクトの全てのファイルを検索します。

Print All File[すべてのファイルを印刷]
プロジェクトのすべてのファイルを印刷します。このためにはインクルート・ファイルがわかるようにプログラムがコンパイルされていなければいけません。

Include Dirs[ディレクトリーを含める]
そのプロジェクトのためのインクルード・ファイルを検索するために各々のディレクトリーの指定を行います。この情報は .PJT ファイルにセーブされます。

Close Project[プロジェクトを閉じる]
そのプロジェクトに関連したファイルを閉じます。

編集メニュー

Undo[アンドゥ] 最後の動作をアンドゥ

Cut[カット] ファイルから選択したテキストをクリップ・ボードへ移動

Copy[コピー] 選択したテキストをクリップ・ボードへコピー

Paste[ペースト] クリップ・ボードの内容をカーソル位置へコピー

Select All[すべて選択] ファイル内のすべてのテキストをハイライト

Copy from File[ファイルからコピー] ファイルの内容をカーソル位置へコピー

Paste to File[ファイルへペースト] 選択したテキストをファイルにペースト

Find[検索] 指定した文字列でファイルの中を検索

Replace[置き換え] 指定した文字列を新しい文字列の置き換え

Next[次へ] 他の検索、又は、置き換えを行います。

Match Brace[括弧の一致] 一致した }又は、) がハイライトされます。エディターはオープンとクロース括弧のカウントを開始し、対応する閉じる括弧をハイライトします。カーソルを括弧の前に置くか、又は、マッチングを見つける必要のある括弧をクリックしますとマッチングする括弧がハイライトされます。

Match Brace Extended[括弧の一致の拡張] 対応した }又は､)までテキストがハイライトされます。

Indent Selection[インデント選択] ハイライトされたテキストをインデント

Toggle Bookmark[トグル・ブックマーク] カーソル位置でブックマーク(0 から 9)をセット

Goto Bookmark[ブックマークへ] カーソルを指定したブックマークへ移動

Next Windows[次のウィンドウへ] 編集のために次にオープンするファイルを選択

Previous Windows[前のウィンドウへ] 編集のために前にオープンしたファイルを選択

オプション・メニュー

Recall Open Files[リコール・オープン・ファイル]
選択しますと、PCW は常に最後にシャット・ダウンされた時にオープ

ンされていたファイルと同じファイルで開始します。

Editor Properties[エディター・プロパティ]
クリックしますと、Editor Properties ウィンドウがオープンされ、ユーザーが各種オプションを設定することが出来ます。下記のタブがあります。

General Tab: **Window Settings[ウィンドウズ・セッティング]**
エディター(horizontal[水平]と vertical[垂直])のスクロール・バーが選択出来ます。

Editor Options[エディター・オプション]
Syntax Highlighting[構文ハイライト]
チェックされますと、エディターはカラーで C キーワードとコメントをハイライトします。

Auto indent[オート・インデント]
選択されて、ENTER を押しますと、カーソルは前の行の最初のキャラクターの次の行にカーソルが移動します。選択されませんと、ENTER は常に次の行の最初に移動します。

TABS[タブ]
Tab size
タブ位置間のキャラクター数が決定されます。タブに相当するスペース数もセット出来ます。
Keep Tabs
選択されて、TAB キーが押された時、エディターはタブ・キャラクター(ASCII 9)を挿入。
Insert Spaces
選択されて、TAB キーが押された時、次のタブ位置までスペースが挿入されます。

Display Tab: **Margin[マージン]**
Visibl Left Margin
エディターの左マージンが見えるようになります。
Visibl Right Margin
エディターの右マージンが見えるようになります。
Left Margin Width
左マージンの幅
Right Margin Width
右マージンの幅
Editor Font
エディターのフォントを選択します。
Font Size
エディター・フォントのサイズ
Font Style
エディター・フォントのスタイル(イタリック/ボールド/下線)

Color Tab: 構文ハイライトのカラーを選択することが出来ます。

Customize[カスタマイズ] デバッガの機能にすばやくアクセス出来るようにツールバーにアイコンを追加することが出来ます。

File Formats[ファイル形式] 出力ファイルの形式を選択することが出来ます。

デバッグ・ファイル・オプション

Microchip COD	標準 PICmicro MCU デバッグ・ファイル
RICE16 MAP	古い RICE16 にのみ使用されます。
拡張 COD	アドバンスト・デバッグ情報を持った COD ファイル

リスト・フォーマット・オプション

Simple	C コードと ASM を持った基本的なフォーマット
Standard	マシーン・コードを持った MPASM 標準フォーマット
Old	古い MPASM フォーマット
Object file extension	HEX ファイルのためのファイル拡張子
List file extension	リスト・ファイルのためのファイル拡張子

オブジェクト・ファイル・オプション

8bit HEX	8bit HEX ファイル
16bit HEX	16bit HEX ファイル
Binary	ヒューズ情報を含んでいないストレート・バイナリー

エラー・ファイル・オプション

Standard	マイクロチップの現在の標準
Original	マイクロチップの古い標準

Include Dirs[インクルード・ディレクトリー]
新規作成されたプロジェクトのためにデフォルトでインクルード・ファイルを検索するのに使用する各ディレクトリーを指定する事が出来ます。これは既に作成されたプロジェクトには効果はありません。

Debugger/Programmer[デバッガ/プログラマー]
デバイス・プログラマーやデバッガを指定することが出来ます。

Global Definitions[グローバル定義] コンパイルで使用される #defines の設定を行います。

コンパイル・オプション

PCW コンパイル

現在のプロジェクト(下右に名前があります。)を現在のコンパイラー(ツール・バーに名前があります。)を使ってコンパイルします。

ビュー・メニュー

C/ASM　リード・オンリー(READ ONLY)モードでリスティング・ファイルをオープンします。リスト・ファイルを見るにはファイルはコンパイルしなければいけません。
コンパイル後このファイルをアップデートします。リスティング・ファイルは各 C ソース行とその行のために生成された関連したアセンブリー・コードを表示します。
サンプル：

```
……..delay_ms(3)
0F2:    MOVLW 05
0F3:    MOVWF 08
0F4:    DESCZ 08,F
0F5:    GOTO 0F4
……..while input(pin_0));
0F6:    BSF 0B,3
```

シンボル・マップ　リード・オンリー(READ ONLY)モードでシンボル・ファイルをオープンします。シンボル・ファイルを見るにはソースファイルをコンパイルしなければなりません。コンパイルされる度にシンボル・ファイルは更新されます。シンボル・ファイルは各レジスター位置と各位置でどのようなプログラム変数をセーブしたかを表示します。
最後にコンパイルされたプログラムの RAM メモリー・マップを表示します。
マップは各 RAM 位置の使用を表示します。ある位置では現在の実行されている状況による再使用されていますのでマルチになっています。
サンプル：

```
08      @SCRATCH
09      @SCRATCH
0A      TRIS_A
0B      TRIS_B
0C      MAIN.SCALE
0D      MAIN.TIME
0E      GET_SCALE.SCALE
0E      PUTHEX.N
0E      MAIN.@SCRATCH
```

コール・ツリー　View メニューの Call Tree コマンドでコールツリーが表示されますが、修正することはできません。リード・オンリー(READ ONLY)モードでツリー・ファイルをオープンします。コールツリーを見るにはソースファイルをコンパイルしなければなりません。コンパイルされる度にコールツリーは更新されます。コールツリーは各関数が使用する ROM 及び RAM、関数の中でコールされる関数を表示します。

(inline)は@で始まるインライン・プロシージャの後に現れます。
プロシージャ名の後は s/n の形式で数字が表示されます。ここで、s はプロシージャのページ番号、 n は必要とされるコードのサイズです。
もし、s の個所が?となっている場合は、コンパイラが ROM 領域を使い果たしたことを示します。もし、s の個所が?となっている場合は、コンパイラが ROM 領域を使い果たしたことを示します。
サンプル： Main 0/30
INIT 0/6

WAIT_FOR_HOST 0/23 (Inline)
DELAY_US 0/12
SEND_DATA 0/65

スタティクス　リード・オンリー(READ ONLY)モードでスタッツ・ファイルをオープンします。スタッツ・ファイルを見るにはソースファイルをコンパイルしなければなりません。コンパイルされる度にスタッツ・ファイルは更新されます。スタッツ・ファイルは各関数が使用する ROM 及び RAM を表示します。

データ・シート　このツールは選択されたパーツのメーカーのデータ・シートをアクロバット・リーダー形式で表示します。

バイナリー・ファイル　リード・オンリー(READ ONLY)モードでバイナリー・ファイルをオープンします。このファイルは HEX と ASCII で表示されます。

COD デバッグ・ファイル　リード・オンリー(READ ONLY)モードでデバッグ・ファイルをオープンします。このファイルは翻訳された形式で表示されます。

有効ヒューズ　このデバイスの#uses ディレクティブのための全ての有効なキーワードのリストを表示します。

有効インターラプト　このデバイスの#int_xxxx ディレクティブと enable/disable _interrupts のための全ての有効なキーワードのリストを表示します。

ツール・メニュー

デバイス・エディター　サポートされている各プロセッサーに対する主要な特性を指定します。このツールはコンパイルをコントロールするためにコンパイラーによって使用されるデータベースを編集します。

デバイス・セレクター　デバイスのパラメトリック選択を行えるようにデバイスのデータベースを使用します。

ファイル比較　２つのファイルを比較します。ソース・ファイルが選択されますと、行ごとの比較が行われます。リスト・ファイルが選択されますと、比較はより意味のあるものにするために RAM と ROM 又は、いずれかを無視するようにセットされます。
例えば、プログラムの最初に ASM 行が追加されますと、ノーマル比較は各行にフラッグを立てます。ROM アドレスを無視することで、特別な行のみが変更されてフラッグされます。２つの出力形式が利用出来ます。１つは表示のため、１つはファイル又は、印刷のためです。

ニューメリック変換　デシマル、HEX と実数間をコンバートするためのツールです。

シリアル・ポート・モニター　シリアル・ポートに接続する簡単に使用できるツールです。このツールは RS232 を使用してターゲット・プログラムと通信するのに便利なツールです。データは ASCII キャラクターとローHEX として示されます。

ディスアセンブラー　HEX ファイルを入力として、そして、ASM を出力します。ASM はインライン ASM として使用できる形式です。
このコマンドは HEX ファイルを入力とし、選択された部分を抽出してイ

ンライン・アセンブリーとして C プログラムに挿入することが出来ます。オプションはアセンブリー・フォーマットを選択することが出来ます。
●12 又は、14bit オペコード
●アドレス、C, MC ASM ラベル
●HEX 又は、Binary
●Simple, ASM, C numbers

カリブレーション・データ抽出　HEX ファイルを入力し、カリブレーション・データを C インクルード・ファイルへ抽出します。
これは UV 消去パーツのカリブレーション・データを保持するのに使用されます。

プログラム・チップ　**Option/Program**s ウィンドウで指定されたとき、デバイス・プログラマー・ソフトウエアを呼び出すことが出来ます。コマンド・ラインの確立はコンパイル・オプションを使用して下さい。

MPLAB　現在のプロジェクトで MPLAB を呼び出します。プロジェクトがクローズされますと MPLAB は必要な場合はファイルを修正します。この方法で MPLAB が呼び出されますと、PCW は MPLAB が終了し、プロジェクトが再ロードされるまで最小化されて留まります。

PCW エディター・キー

カーソルの動き	
[←]左矢印キー	カーソルを１文字左へ移動
[→]右矢印キー	カーソルを１文字右へ移動
[↑]上矢印キー	カーソルを１行上へ移動
[↓]下矢印キー	カーソルを１行下へ移動
Ctrl+左矢印キー	カーソルを１ワード左へ移動
Ctrl+右矢印キー	カーソルを１ワード右へ移動
Home	カーソルを行頭へ移動
End	カーソルを行末へ移動
Ctrl+PgUp	カーソルをウィンドゥの最上行へ移動
Ctrl+PgDn	カーソルをウィンドゥの最下行へ移動
PgUp	1 画面分上へスクロール
PgDn	1 画面分下へスクロール
Ctrl+Home	カーソルをファイルの先頭へ移動
Ctrl+End	カーソルをファイル・エンドへ移動
Ctrl S	カーソルを１文字左へ移動
Ctrl D	カーソルを１文字右へ移動
Ctrl E	カーソルを１行上へ移動
Ctrl X	**カーソルを１行下へ
Ctrl A	カーソルを１語左へ移動
Ctrl F	カーソルを１語右へ移動
Ctrl Q S	カーソルをウィンドウのトップへ移動
Ctrl Q D	カーソルをウィンドウのボトムへ移動
Ctrl R	カーソルをファイルの始めに移動
Ctrl C	*カーソルをファイルの最後に移動
Shift ～	上記のいずれにも～がある場合：カーソル移動したとき選択されたエリアを拡張します。

編集コマンド	
F4	マッチした()又は、{}で次のテキストを選択
Ctrl #	ブックマーク #0-9 へ
Shift Ctrl #	ブックマーク #0-9 を設定
Ctrl Q #	ブックマーク #0-9 へ
Ctrl K #	ブックマーク #0-9 を設定
Ctrl W	スクロール・アップ
Ctrl Z	* スクロール・ダウン
Del	次のキャラクターを削除
BkSp	前のキャラクターを削除
Shift BkSp	前のキャラクターを削除
Ins	挿入/上書きモードをトグル
Ctrl Z	** 最後の操作をアンドゥ
Shift Ctrl Z	最後のアンドゥをリドゥ(再実行)
Alt BkSp	オリジナルの内容にリストア
Ctrl Enter	新しい行を挿入
Shift Del	選択したテキストをファイルからカット
Ctrl Ins	選択したテキストをコピー
Shift Ins	ペースト
Tab	タブ又は、スペースを挿入
Ctrl Tab	タブ又は、スペースを挿入
Ctrl P ～	コントロール・キャラクター～をテキストに挿入
Ctrl G	次のキャラクターを削除
Ctrl T	次のワードを削除
Ctrl H	前のキャラクターを削除
Ctrl Y	行の削除
Ctrl Q Y	行の終わりまで削除
Ctrl Q L	オリジナルの内容にリストア
Ctrl X	* 選択したテキストをファイルからカット
Ctrl C	** 選択したテキストをコピー
Ctrl V	ペースト
Ctrl K R	カーソル位置でファイルを読込み
Ctrl K W	選択したテキストをファイルへ書込み
Ctrl-F	* ファインド・テキスト
Ctrl-R	** テキスト置き換え
F3	最後のファイド/置き換えをリピート

*Wordstar モード選択時のみ

**Wordstar モードが選択されいない時のみ

プロジェクト・ウィザード
新規プロジェクト・ウィザードは新しいプロジェクトを簡単に始めることが出来ます。
ウィザードを開始しますと、メインの C ファイル名を入力する必要があります。このファイルは関連する .h ファイルに従って作成されます。メニューのプロジェクト｜新規（Project｜New）かスピードボタンで選択すると新しいプロジェクトを作成するためにブランクのフォームを用意します。RS232C I/O や I^2C,タイマー,割り込み,AD コンバーター設定,ドライバー設定などそれぞれのダイアログ・ボックスから選択入力します。以下にそれぞれの設定ダイアログ・ボックスを示します。

●最初の General タブではプロセッサー名、クロック, オシレータ等一般的な設定を行います。
●Communication タブでは RS232C、I^2C, PSP, SPI 等の設定を行います。
ここで設定される内容に添って後の設定が変化します。例えば、AD コンバータを内蔵しないデバイスの場合、“アナログ入力”（Analog Input）の設定は無意味になります。
Timer タブではタイマーの設定が行えます。
Analog Input タブでは A/D ピン, A/D クロックの設定が行えます。
AD コンバーターを持たないデバイスでは無意味です。
Other タブでは CCP1, CCP2 のモード等の設定が行えます。
Interrupts タブでは割り込みの設定を行います。必要とする割り込みイベントにチェックを入れます。
Drivers タブではチェックを入れることで各種ドライバーの設定を行います。
サンプルに入っているドライバーを設定します。それぞれのドライバーは PCW をインストールしたディレクトリーやフォルダの“EXAMPLES”ディレクトリーにある対応するファイルがインクルードされます。ドライバーの内容についてはそれぞれインクルードされるファイルを参照してください。ドライバーはすべてが一度に選択できるわけではありません。

【注意】“EXAMPLES”ディレクトリーのソースファイルを変更するとここで使用されるドライバーの内容も変化します。この場合、入出力ピン設定などが重複したり設定が間違ったりしますのでご注意ください。

●I/O ピンの入出力方向や用途を設定します。
ドライバー設定やアナログ設定、RS232C、I^2C 設定などによりすでに決まっている場合もあります。このようにして設定を終了して OK ボタンをクリックすると最初に指定したファイル名でソースファイルが作成されます。

CCS デバッガー

デバッガー – 概要

PCW IDE にはビルトイン・デバッガーが付いています。デバッガーは Debug｜Enable メニューで開始できます。このセクションは下記のトピックスを含みます。

- デバッグ・メニュー
- コンフィギュア
- コントロール
- ウォッチ
- ブレーク
- RAM
- ROM
- DATA EEPROM
- スタック
- エバルエーション[評価]
- ログ
- モニター
- ペリフェラル
- スナップショット
- イネーブル/ディスエーブル

デバッグ – メニュー

このメニューは ICD が PC と C のプログラムをデバッグするためのプロトタイプ・ボードに接続されているときはすべてのデバッガー・オプションを含んでいます。

デバッガー – コンフィギュア

コンフィギュア・タブはどのハードウエア・デバッガーが接続されるかを選択します。他のコンフィギュレーション・オプションは使われるハードウエア・デバッガーにより異なります。また、コンフィギュア・タブはターゲットにコードを手動で再ロードすることが出来ます。

もし、デバッガー・ウィンドウが開かれていて、”Reload target after every compile”ボックスが選択されていますと、プログラムがコンパイルされる度にプログラムがターゲットにダウンロードされます。

デバッガー・プロファイルはウォッチされる変数、デバッガー・ウィンドウの位置とサイズとブレークポイント設定のようなすべてのデバッガー・タブの選択が含まれています。プロファイルはファイルにセーブされ、そして、コンフィギュア・タブからロードできます。セーブ又は、ロードされた最後のプロファイル・ファイルはプロジェクト .PJT ファイルにセーブされます。

ICD ユーザーのためのスペシャル・ノート

ICDを使用するときは、CCSファームウエアがICDにインストールされていなければいけません。ファームウエアをインストールするには “Configure Hardware”をクリックしてから一番上の真ん中のボタンをクリックしてください。

デバッガー – コントロール

リセット・ボタンはターゲットをリセット状態にします。ソース・ファイル・ウィンドウでリスティング・ウィンドウと ROM ウィンドウのカレント・プログラム・カウンター行は黄色にハイライトされます。これは実行すべき次の行です。

Go ボタンでプログラムの実行を開始します。 デバッガー・ウィンドウが実行されていない間に現在の情報で更新されます。プログラムはブレーク条件に行ったとき、又は、STOP ボタンが押されたときに止まります。

STEP ボタンは、もし、ソース・ファイルがアクティブ・エディター・タブの場合、１つのＣ行だけを、もし、リスト・ファイルがアクティブ・エディター・タブの場合は、１アセンブラー行を実行します。STEP OVER は STEP の様に動きます、異なる点は、もし、行が他の関数を呼んでいますと、全ての関数が１つの STEP OVER で実行されます。

GO TO ボタンはエディター・カーソルがその行に達するまで実行されます。

デバッガー – イネーブル/ディスエーブル
このオプションはデバッガーがその状態にない場合にイネーブル/ディスエーブルします。メニュー・オプションは自動的に他のものに変更します。PCW デバッガーIDE を表示したり、隠したりします。

デバッガー – ウォッチ
ウォッチ・タブがウォッチする新しい式を入力する選択がされているときに+アイコンをクリックして下さい。ヘルパー・ウィンドウがポップアップしウォッチするプログラムの識別子を見つけることができます。通常の C 表現は下記の様にウォッチされます。

```
X
X+Y
BUFFER[X]
BUFFER[X].NAME
```

ウォッチに入る時にソース・ファイルにエディター・カーソルのあるところの式がどのように影響するかを見て下さい。例えば、

デバッガー – ブレーク
ブレークポイントをセットするのに、ソースかリスト・ファイルの必要なライン位置へエディタ・カーソルを動かしてください。そして、デバッガのブレーク・タブを選択し、+アイコンをクリックしてください。

ブレークの動作はハードウエア・ユニットの種類で異なります。
例えば、ICD で PIC16 を使うと、ただ１つのブレーク設定が可能な事と、プロセッサの実行はストップ動作の前の状態でブレークポイントをセットされたライン（アセンブラのライン）までを実行します。

デバッガー – RAM
デバッガ RAM タブはターゲット RAM を示します。赤い番号はプログラムが最後に停止されてから変更された状態を示しています。ブラック・アウトされている位置は物理的にレジスタが存在しないか、デバッグ中に利用できない事を表わしています。RAM を変更するには変更したい位置でダブル・クリックして変更内容を入力してください。全ての番号はヘキサです。

デバッガー – ROM
ROM タブはターゲットのプログラム・メモリーの内容をヘキサ値とディスアセンブルの両方で示します。このデータは最初にロードした HEX ファイルからのもので、ユーザーが要求しないかぎりターゲットからは変更されません。ターゲットから再ロードするにはウィンドウで右クリックしてください。

デバッガー – DATA EEPROM
デバッガの Data EEPROM タブはターゲットの Data EEPROM を示します。赤い番号はプログラムが最後に停止されてから変更された状態を示しています。変更したい Data EEPROM の位置でダブル・クリックして変更したい内容を入力してください。全ての番号はヘキサです。

デバッガー – スタック
このタブは現在のスタックを示します。最後に呼び出された関数とすべてのパラメータはリストのトップに表示されます。

PIC16 でスタックを見るにはソース・ファイルで#DEVICE CCSICD=TRUE がなければいけません。これによりコンパイラーはエキストラ・コードを生成しデバッガでスタックが見える様になります。

PIC18 でスタックを見るにはソース・ファイルに #device ICD=TRUE があるだけで十分です。

デバッガー – エバルエーション[評価]
このタブは C 記述の評価のためのものです。これはウォッチ機能をよく似ていますが、大きな構造や配列のために用いられ、さらに空き領域があることが異なります。

エバルエーションはターゲットの C 関数を呼ぶことができます。この場合、すべてのパラメータを与えなければいけません。関数の結果は Result ウィンドウに表示されます。
この機能はすべてのデッバガ・プラットフォームで利用可能ではありません。

デバッガー – ログ
ログ機能はブレーク、ウォッチとスナップショットの組み合わせです。ブレーク番号と評価する数式を各ブレーク毎に指定してください。プログラムは数式が評価され結果がログ・ウィンドウに記録された後、再スタートします。複数の数式はセミコロンで別々に指定します。ログ・ウィンドウはファイルにセーブすることが出来ます。ファイルの中の各数式結果はスプレッド・シート形式のプログラムにインポートし易い様にタブでセパレートされています。

デバッガー – モニター
モニター・ウィンドウはターゲットからのデータを示し、そして、ターゲットに送るデータの入力が出来ます。これはターゲット上で下記の様に行われます。

```
#use RS232(DEBUGGER)
…
printf("Test to run? ");
test=getc();
```

デバッガー – ペリフェラル
このタブはターゲット・スペシャル・ファンクション・レジスターの状態を示します。このデータは関数によって構成されます。
レジスターの下にレジスターの各フィールドがどのビット・パターンがリストされます。

デバッガー – スナップショット
スナップショト・ウィンドウを画面に現すにはカメラ・アイコンをクリックします。スナップショット機能はパーツや各種デバッガー・ウィンドウの内容の記録が出来ます。右側で記録したい項目を選ぶことが出来ます。右上はデータをどこへ記録するかを選びます。オプションは：

- プリンター
- 新規ファイル
- 既存のファイルに追加

さらに、いつスナップショットをおこなうかを選択することが出来ます。

- 今
- 毎ブレーク
- 毎シングル・ステップ

更に、コメントをファイルに挿入するために APPEND COMMENT ボタンをクリックすることが出来ます。

プリプロセッサー

プリプロセッサー・コマンド一覧	
標準 C	デバイス・スペシフィケーション
＃DEFINE ID STRING	＃DEVICE CHIP
＃ELSE	＃ID NUMBER
＃ENDIF	＃ID“filename”
＃ERROR	＃ID CHECKSUM
＃IF expr	＃FUSES options
＃IFDEF　id	＃TYPE type=type
＃INCLUDE“FILENAME”	組み込みライブラリー
＃INCLUDE <FILENAME>	＃USE DELAY CLOCK
＃LIST	＃USE FAST_IO
＃NOLIST	＃USE FIXED_IO
＃PRAGMA cmd	＃USE I2C
＃UNDEF id	＃USE RS232
関数修飾	＃USE STANDARD_IO
＃INLINE	メモリー・コントロール
＃INT_DEFAULT	＃ASM
＃INT_GLOBAL	＃BIT id=const.const
＃INT_xxx	＃BIT id=id.const
＃SEPARATE	＃BYTE id=const
コンパイラー内部定義	＃BYTE id=id
_ _DATE_ _	＃LOCATE id=const
_ _DEVICE_ _	＃ENDASM
_ _FILE_ _	＃RESERVE
_ _LINE_ _	＃ROM
_ _PCB_ _	＃ZERO_RAM
_ _PCM_ _	コンパイラー制御
_ _PCH_ _	＃CASE
_ _TIME_ _	＃OPT n
	＃PRIORITY
	＃ORG
	＃IGNORE_WARNINGS

プリプロセッサー・ディレクティブ
すべてのプリプロセッサー・ディレクティブは、#で始まります。その後スペースを空けずにディレクティブ・コマンドが続きます。
プリプロセッサー・ディレクティブは、表に示すようなコマンドがあります、コマンドの詳細については次章のそれぞれのプロプロセッサー・コマンドを参照してください。一般的にプリプロセッサー・ディレクティブは、“#PRAGMA”に続いてコマンドが書かれますが、PCB,PCM,PCW コンパイラーは直接、“#”に続いてコマンドを記入しても同じです。これは、コンパイラーの互換性のために設けてあります。

サンプル： 下記の両方が有効です。

```
#INLINE                 //  インラインプリプロセッサー・コマンド
#PRAGMA INLINE          //  このように書いても同じです
```

＃ASM
＃ENDASM
構文：#asm 又は、#asm ASIS
code
#endasm
エレメント: ***code*** はアセンブリ言語命令
目的: #ASM と#ENDASM の間に直接アセンブラー・コードを書きます。ここに使用できるアセンブラー・ニーモニックは、マイクロチップ社のニーモニックで次の表に示してあります。
関数が戻り値を持つ場合は、“_RERURN_”が定義されています。基本的には C の構文ですので、コメントなど C の構文を挿入することができます。もし、ASIS と共に 2 番目のフォームが使用されますと、コンパイラーは現在のバンクからアクセス出来ない変数に対して自動バンク切り替えを行いません。アセンブリ・コードはそのまま使用されます。このオプションがない場合、アセンブリが添字されますので変数は常に必要とされるところから追加されたバンク切り替えで正しくアクセスされます。
サンプル：

```
int         find_parity(int data) {
int         count;
#asm
movlw       8
movwf       count
movlw       0
loop:
xorwf       data,w
rrf         data,f
decfsz      count,f
goto        loop
movwf       _return_        // 関数の戻り値
#endasm
}
```

サンプル・ファイル： ex_glint.c

#ASM で使用できるニーモニック一覧

12bit(PCB)と 14bit(PCM)	
ADDWF f,d	ANDWF f,d
CLRF f	CLRW
COMF f,d	DECF f,d
DECFSZ f,d	INCF f,d
INCFSZ f,d	IORWF f,d
MOVF f,d	MOVPHW
MOVPLW	MOVWF f
NOP	RLF f,d
RRF f,d	SUBWF f,d
SWAPF f,d	XORWF f,d
BCF f,b	BSF f,b
BTFSC f,b	BTFSS f,b
ANDLW k	CALL k
CLRWDT	GOTO k
IORLW k	MOVLW k
RETLW k	SLEEP
XORLW	OPTION
TRIS k	
	14bit(PCM)のみ
	ADDLW k
	SUBLW k
	RETFIE
	RETURN

f：定数(ファイル番号)又は、シンプル変数
d：0 か 1 を持つ定数又は、W 又は、F
f, b: ファイルと定数(0-7)又は、ビット変数リファレンス
k：定数

すべての表現とコメントは C ライクな構文です。

PIC18					
ADDWF	f,d,a	ADDWFC	f,d,a	ANDWF	f,d,a
CLRF	f,a	COMF	f,d,a	CPFSEQ	f,a
CPFSGT	f,a	CPFSLT	f,a	DECF	f,d,a
DECFSZ	f,d,a	DCFSNZ	f,d,a	INCF	f,d,a
INFSNZ	f,d,a	IORWF	f,d,a	MOVF	f,d,a
MOVFF	fs, fd	MOVWF	f,a	MULWF	f,a
NEGF	f,a	RLCF	f,d,a	RLNCF	f,d,a
RRCF	f,d,a	RRNCF	f,d,a	SETF	f,a
SUBFWB	f,d,a	SUBWF	f,d,a	SUBWFB	f,d,a
SWAPF	f,d,a	TSTFSZ	f,a	XORWF	f,d,a
BCF	f,b,a	BSF	f,b,a	BTFSC	f,b,a
BTFSS	f,b,a	BTG	f,d,a	BC	n
BN	n	BNC	n	BNN	n
BNOV	n	BNZ	n	BOV	n
BRA	n	BZ	n	CALL	n,s
CLRWDT	-	DAW	-	GOTO	n
NOP	-	NOP	-	POP	-
PUSH	-	RCALL	n	RESET	-
RETFIE	s	RETLW	k	RETURN	s
SLEEP	-	ADDLW	k	ANDLW	k
IORLW	k	LFSR	f,k	MOVLB	k
MOVLW	k	MULLW	k	RETLW	k
SUBLW	k	XORLW	k	TBLRD*	
TBLRD*+		TBLRD*-		TBLRD+*	
TBLWT*		TBLWT*+		TBLWT*-	
TBLWT+*					

＃BIT
構文：#bit ***id=x.y***
エレメント: ***id*** は有効な C の識別子
X は定数、又は、C 変数
Y は定数 0-7
目的: このディレクティブは“id”のビットを“SHORT　INT”（1bit）で定義します。
“id”は有効 C の識別子
“x”は定数や定義済み変数を指定します。“x”アドレスのビット“y”を定義します。
“y”は定数　0-7
サンプル： #bit T0IF = 0xb.2

```
…
T0IF = 0 ; //  タイマー0 のインターラプト・フラッグをクリア

int resuilt;
#bit result_odd = result.0              //   変数 result の 0bit を定義
…
if ( result_odd )
…
```

サンプル・ファイル： ex_glint.c
参照: #byte, #reserve, #locate

＃BYTE
構文：#byte ***id=x***
エレメント: ***id*** は有効 C の識別子

x は C 変数又は、定数
目的: id をバイトとして定義し、INT（１バイト）として使用します。
メモリー・ロケーションの x を参照することと同じで、 x は定数でも変数でもかまいません。
もし、 x が変数ならば、この変数は他の変数のように同じアドレス上に取られます。
また、もし id がすでに存在し、１バイトであれば、id は指定されたアドレス上に取られます。
サンプル：#byte status = 3

```
#byte b_port = 6                    // b_port をアドレス 6h と定義
struct {                            // a_port の構造体を定義
  short int r_w;
  short int c_d;
  int unused : 2;
  int data     : 4; } a_port;
#byte a_port = 5                    // 定義された a_port をアドレス 5h に定義
…
a_port.c_d = 1;
```

サンプル・ファイル： ex_glint.c
参照: #bit, #locate, #reserve

＃CASE
構文： #case
エレメント: なし
目的: コンパイラーをケース・センシティブにします。デフォルトはケース・センシティブです。
注意：すべてのサンプル・プログラム、ヘッダーとドライバーがケース・センシティブかテストされていません。
サンプル：#case

```
int STATUS;
void func() {
int status;
…
STATUS = status; // ローカル・ステータスをグローバルへコピー
```

サンプル・ファイル：ex_cust.c

_ _DATE_ _
構文： _ _date_ _
エレメント: なし
目的: このプリ・プロセッサーの識別子はコンパイラーが内蔵している定数でコンパイルが行われた日付を保持してします。日付の表現形式は、“01-OCT-97”（1997 年 10 月 1 日）となります。
サンプル：

```
printf( “Software was compiled on “ );
printf( __DATE__ );
```

＃DEFINE
構文： #define ***id*** text 又は、#define ***id(x,y…)*** text
エレメント: id はプリプロセッサ識別子、text はテキスト、***x,y*** 等はローカル・プリプロセッサ識別子、この形式ではカンマにより分けられた１つ又は、2 つ以上の識別子がありえます。
目的: ID で示される識別子を“string”で置き換えます。マクロ定義にも使用されます。
サンプル：

```
#define BITS 8              // BITS を 8 で置き換える
a = a + BITS;               // a = a + 8 と同じ
#define hi(x) (x<<4)
a = hi(a);                  // a = (a<<4)と同じ
```

サンプル・ファイル： ex_stwt.c, ex_macro.c
参照: #undef, #ifdef, #ifndef

＃DEVICE
構文： #device ***chip options***
エレメント: ***chip*** は指定するプロセッサーの名前です。(例えば、PIC16C74)
サポートされるデバイスは次の通りです。
START | RUN | CCSC +Q
Options[オプション]はデバイスの標準操作に対するクフォリファイアです。
● ****=5-*** 5bit ポインターを使用(12bit パーツ)
● ****=8-*** 8bit ポインターを使用(12 と 14bit パーツ)
● ****=16-*** 16bit ポインターを使用(14bit パーツ)
● ***ADC=x*** x がビット数のところは read_adc()を返してください。
● ***ICD=TRUE*** ICD デバッギング・ハードウエア対応のコードを作成
chip と option はオプションですので、複数の #device 行をデバイス定義に使用することも出来ます。しかし、#device でチップを指定しますと、前のすべての #device と #fuse 設定がクリアされてしまいます。
目的: ターゲット・プロセッサーを定義します。すべてのプログラムはチップに１つの#devine を持っていなければいけません。
サンプル： #device PIC16C74
#device PIC16C67 *=16
#device *=16 ICD=TRUE
#device PIC16F877 *=16 ADC=10
サンプル・ファイル：ex_mxram.c, ex_icd.c, 16c74.h
参照： read_adc()

_ _DEVICE_ _
構文：_ _ device _ _
エレメント: なし
目的: このコンパイラー内部変数には＃DEVICE で指定されたデバイス名が格納されています。ただし、デバイス名にはプロセッサのナンバー部のみです。例えば、PIC16C622 なら__DEVICE__ 変数には、“622”が格納されています。
サンプル： #if __DEVICE__ == 71
setup_port_a(ALL_DIGITAL);
#endif
参照: #device

#ERROR
構文：#error ***text***
エレメント: ***text*** はオプションです。如何なるテキストでもかまいません。
目的: 何らかの理由でコンパイラーがコンパイルを中止したとき、＃ERROR ディレクティブ以降に示される文字列を画面に表示してエラーメッセージとして使用されます。マクロが含まれるとその結果を表示します。例で示されるように、定義された値がエラーを出す場合（BUFFER_SIZE が 16 以上のとき）＃ERROR 以降のメッセージを表示してコンパイルが停止されます。
このディレクティブは注意深くプログラムを作成されるユーザーには大変重宝します。
サンプル: #if BUFFER_SIZE>16
#error Buffer size is too large
#endif
#error Macro test: min (x, y)
サンプル・ファイル：ex_psp.c

_ _FILE_ _
構文：_ _FILE_ _
エレメント: なし
目的: プリプロセッサ識別子はコンパイル時にコンパイルされるファイル名に置き換わります。

サンプル： if(index>MAX_ENTRIES)
printf(“Too many entries, source file: “__FILE__
“ at line “ __LINE__ “¥r¥n”) ;
サンプル・ファイル：assert.h
参照: _ _line_ _

#FUSES
構文：#use ***options***
エレメント: ***options*** はデバイスによりそれぞれ違います。
● LP, XT, HS, RC - クロックソース
● WDT, NOWDT - ウォッチドッグ・タイマー
● PROTECT, NOPROTECT - コードプロテクト
● PUT, NOPUT - パワアップ・タイマー
● BROWNOUT, NOBROWNOUT – ブローアウト、ノー・ブローアウト
目的: #FUSE ディレクティブは、CPU のヒューズデータの書き込み内容を指示します。
このディレクティブは、コンパイルにはなんの影響も与えませんが、この情報は、出力ファイルに書き出され、CPU の動作が決定されます。“PAR”はパララックス社のフォーマットを指定する場合に使用します。SWAP は出力される HEX ファイルのプログラムコード以外のデータの上位、下位バイトを入れ替える命令です。(マイクロチップ社標準)使用できるオプションは、以下の通りです。
サンプル:#fuses HS, NOWDT
サンプル・ファイル：ex_sqw.c

#ID
#ID ***number 16***
#ID ***number, number, number, number***
#ID ***“filename”***
#ID ***CHECKSUM***
エレメント: 最初の指定例では、16bit の値を HEX で指定します。また、4bit ずつ 4 つの数値で表すことも可能です。***number16*** は 16bit 数、***number*** は 4bit 数です。
Filename は有効な PC のファイル名、そして、***checksum*** はキーワード
目的: このディレクティブはパーツにプログラムされる ID を定義します。このディレクティブはコンパイルに影響は与えませんが、情報は出力ファイルに出力されます。
次の例ではファイルから ID 設定を読み込みます。ファイル名が指定されますと、ID はファイルから読み込まれます。ファイルのフォーマットは単に数値を指定し CR/LF で区切られたテキスト・ファイルです。“CHECKSUM”を指定するとコンパイルされたコードのチェックサムが ID として採用されます。
サンプル: #id 0x1234
#id “serial.num”
#id CHECKSUM
サンプル・ファイル： ex_cust.c

＃IF expr
＃ELSE
＃ELIF
＃ENDIF
構文：#if ***expr***
code
#elif ***expr*** ***//***オプション、いかなる番号も使用できます。
code
#else ***//***オプション
code
#endif

エレメント: ***expr*** は定数、スタンダード・オペレーター又は/と、プリプロセッサーの識別子
code はスタンダードな C ソース・コード
目的: このディレクティブは、“expr”の条件によりプログラムの制御を行わせます。
“expr”が真（0 でない）ならその後の文がコンパイルされ、偽（0 なら）#ELSE 以降がコンパイルされます。また文の制御は、#ENDIF で終了します。#ELSE は必要がない場合は書く必要はありません。プロセッサー表現 DEFINED(id)は id が定義され、そして、0 でない場合に 1 を返えすために使用されます。
サンプル： #if MAX_VALUE > 255
long value;
#else
int value;
#endif
サンプル・ファイル：ex_extee.c
参照：#ifdel, #ifndef

＃IGNORE_WARNINGS
構文：#ignore_warnings ALL
#ignore_warnings none
#ignore_warnings ***warnings***
エレメント: ***warnings*** はカンマで区切られた１又は、それ以上の警告ナンバー
目的: この関数はコンパイラからの警告メッサージを隠します。 ALL は全ての警告を隠します。NONE はすべての警告を報告します。もし、警告ナンバーがリストされていますと、リストにあるナンバーの警告は隠されます。
サンプル： #ignore_warnings 203
while(TRUE) {
#ignore_warnings NONE
サンプル・ファイル：なし
参照：警告メッセージ

＃IFDEF
＃IFNDEF
＃ELSE
＃ELIF
＃ENDIF
構文：#ifdef ***if***
code
#elif
code
#else
code
#endif
#ifndef ***id***
code
#elif
code
#else
code
#endif
エレメント: ***id*** はプリプロセッサーの識別子。***code*** は有効な C ソース・コード
目的: 識別子“id”が定義されている場合(#IFDEF)と定義されていない場合(#IFNDEF)によりコンパイル条件を設定します。識別子は#DEFINE により定義されこれらのディレクティブより以前に設定されていることが必要です。
サンプル： #define debug

```
…
#ifdef DEBUG
printf( “debug point a” );
#endif
```

サンプル・ファイル：ex_sqw.c
参照: #if

＃INCLUDE
構文：#include ***<filename>*** 又は、#include ***“filename”***
エレメント: ***filename***[ファイル名]は有効な PC ファイル名ドライブとパス情報を含みます。
目的: コンパイラーのインストールされたディレクトリーから filename を記述されたところに読み込みます。普通、コンパイラーのヘッダ・ファイルの取り込みに用いられます。
もし、フル・パスが指定されませんと、コンパイラーはプロジェクトで指定されたディレクトリーのリストを使用します。もし、ファイル名が “ “ の場合はメイン・ソースファイルのディレクトリーを最初に読み込みます。もし、ファイル名が < > の場合はメイン・ソースファイルのディレクトリーを最後に読まれます。
サンプル: #include <PIC16C54.H>
#include <C:¥INCLUDES¥COMLIB/MYRS232.C>
サンプル・ファイル：ex_sqw.c

＃INLINE
構文：#inline
エレメント: なし
目的: このディレクティブが指定されると、コンパイラーは、以後の関数をインライン展開します。これはスタック・スペースの節約や関数呼び出しのオーバーヘッドを少なくする目的で使用されますが、コード領域は、呼び出される分だけ必要となります。
サンプル：#inline

```
swapbyte( int &a, int &b ){
int t;
t = a;
a = b;
b = t;
}
```

サンプル・ファイル：ex_sqw.c
参照：#separate

＃INT_xxx

#INT_AD	AD コンバーター
#INT_ADOF	AD コンバーター・オーバーフロー
#INT_BUSCOL	バス衝突
#INT_BUTTON	プッシュ・ボタン
#INT_CCP1	CCP1 キャプチャー
#INT_CCP2	CCP2 キャプチャー
#INT_COMP	コンパレータ
#INT_EEPROM	EEPROM 書き込み完了
#INT_EXT	外部割り込み
#INT_EXT1	外部割り込み #1
#INT_EXT2	外部割り込み #2
#INT_I2C	14000 I^2C 割り込み
#INT_LCD	LCD 動作
#INT_LOWVOLT	低電圧検知

#INT_PSP PSP（パラレル・スレーブ・ポート）データ
#INT_RB B4-B7 でのポート B 変更
#INT_RC C4-C7 でのポート C 変更
#INT_RDA シリアルデータ受信
#INT_RTCC タイマー0（RTCC）オーバーフロー
#INT_SSP SPI 又は、I^2C 動作
#INT_TBE RS232 送信バッファ空
#INT_TIMER0 タイマー0(RTCC)オーバーフロー
#INT_TIMER1 タイマー1 オーバーフロー
#INT_TIMER2 タイマー2 オーバーフロー
#INT_TIMER3 タイマー3 オーバーフロー
ノート：上記以外の多くの #INT_オプションが特定のチップで利用することが出来ます。
すべてのリストに関しては、.h ファイルをチェックして下さい。
エレメント: なし
目的: これらのディレクティブは割り込みサービス・ルーチンを指定します。割り込み関数は引数、戻り値は取れません。すべてのチップですべてのディレクティブが使用できるわけではありません。コンパイラーはこのディレクティブに出会うと割り込みが発生したとき、現在のマシン・ステートを保存、リストアする命令とそれぞれの割り込みエントリーにジャンプする命令を生成します。割り込みは ENABLE_INTERRUPT 関数によりそれぞれのイベントが有効に設定され、グローバル・インターラプト・フラグが立っていることが必要です。
このディレクティブが有効な範囲は、ディレクティブが現れて、関数が終了するまでです。
PCH コンパイラでは、キーワード FAST を高い優先度の割り込みをマークするために使用することができます。ファースト・インタラプトは他の割り込みハンドラを遮ることができます。コンパイラはファースト ISR ではセーブ／リストアはしません。よって、セーブが必要なレジスタは保存しなければなりません。
サンプル：
```
#int_ad
adc_handler( void ){
adc_active = FALSE;
}
#int_rtcc noclear
isr() {
}
```
サンプル・ファイル： ex_sisr.c と ex_stwt.c
参照：enable_interrupts(), disable_interrupts(), #int_default, #int_global

#INT_DEFAULT
構文：#int_default
エレメント: なし
目的: 割り込み関数がセットされていない時や割り込みがかかった時にこの関数が呼び出されます。
サンプル：
```
#int_default
default_isr() {
        printf("Unexplained interrupt¥r¥n");
}
```
参照：#int_xxxx, #int_global

#INT_GLOBAL
構文： #int_global
エレメント: なし
目的: このディレクティブはすべての割り込みサービスをこのディレクティブ以降の関数に割り当てます。この関数は通常は必要としませんが，使用には注意が必要です。
これを使用しますと、コンパイラーはスタート・アップ・コード又は、クリーン・アップ・コード

を作成しません。そして、レジスターはセーブしません。
サンプル：#int_global
```
isr() {                         // 4 の位置に置かれます。
#asm
bsf     isr_flag
retfie
#endasm
}
```
サンプル・ファイル： ex_glint.c
参照：#int_xxxx

_ _LINE_ _
構文：_ _LINE_ _
エレメント: なし
目的: プリプロセッサ識別子はコンパイル時にコンパイルされようとしているファイル行数に置き換わります。
サンプル： if(index>MAX_ENTRIES)
```
    printf("Too many entries, source file: "__FILE__
            " at line " __LINE__ "¥r¥n") ;
```
サンプル・ファイル：assert.h
参照: _ _file_ _

＃LIST
構文： #list
エレメント: なし
目的: リスティング・ファイルへ出力します。#NOLIST で抑止された出力を再開します。
サンプル：#NOLIST   //リスト・ファイルをまとめます。
```
#include <cdriver.h>
#LIST
```
サンプル・ファイル： 16c74.h
参照：#nolist

＃LOCATE
構文： #locate ***id=x***
エレメント: ***id*** は C の変数
x は定数メモリー・アドレス
目的: #LOCATE は#BYTE をよく似た働きをしますが、使用エリアから C を見つけます。
サンプル： // 50-53 に浮動変数を置き、そして、C は自動的に置かれた他の変数のためにこのメモリーを使用しません。
```
Float x;
#locate x=0x50
```
サンプル・ファイル： ex_glint.c
参照：#byte, #bit, #reserve

＃NOLIST
構文： #NOLIST
エレメント: なし
目的: このディレクティブから以降はリスティング・ファイルに出力されません。出力を再開するには#LIST ディレクティブを用います。
サンプル：#NOLIST
```
#include <cdriver.h>
#LIST
```

サンプル・ファイル： 16c74.h
参照：#list

＃OPT
構文： #OPT ***n***
エレメント: ***n*** はオプティマイズ・レベル 0-9
目的: このディレクティブはオプチマイズ・レベル（最適化レベル）を設定します。“n”が 5 のとき、DOS 版のコンパイラと同じオプチマイズ・レベルとなります。
PCW のオプチマイズ・レベルのデフォルトは 9 です。
サンプル： #opt 5

＃ORG
構文： #org ***start, end*** 又は、
#org ***segment*** 又は、
#org ***start, end {}*** 又は、
#org ***start, end auto=0***
#org ***start, end DEFAULT*** 又は、
#org ***DEFAULT***
エレメント: ***start*** は最初の ROM 位置（ワード・アドレス）、***end*** は最後の ROM 位置、***segment*** は前の#org からの ROM 開始位置
目的: このディレクティブは指定された ROM エリアに関数又は、定数をフィックスします。
End はその他の関数をこのセグメントに追加された時のみ、前に定義されているときは省略することが出来ます。
ORG の後の {}はコンパイラーにより何も挿入されないでエリアのみリザーブします。
ORG 関数のための RAM はロー・メモリーにリセットしますので、ローカル変数とスクラッチ変数はロー・メモリーに置かれます。これは ORG された関数が呼び出し元に戻らない場合のみ使用してください。使用された RAM はメイン・プログラムの RAM にオーバラップします。#ORG 行の最後に AUTO=0 を加えて下さい。
キーワード DEFAULT が使用されると、ファイル中で#ORG DEFAULT に遭遇（アドレス範囲なし）するまでのアドレスの範囲はユーザ及びコンパイラが生成する全ての関数に使用されます。
DEFAULT が有効である間、コンパイラ関数が生成コードからコールされたときは、コンパイラは指定されたアドレスの範囲の中で関数を新規に生成します。

サンプル：
```
#ORG 0x1E00, 0x1FFF
Myfunc () {                 //この関数は 1E00 に置かれます。
}

#ORG 0x1E00
Anotherfunc () {            //1E00-1F00 のどこかに置かれます。
}

#ORG 0x800, 0x820{ }  //800-820 にはなにも置かれません。
}

#ORG 0x1C00, 0x1C0F
CHAR CONST ID{10}={“123456789”}
//この ID は 1C00 に置かれます。いくつかの特別なコードは 123456789 を処理します。

#ORG 0x1E00, 0x1FF0
Void loader () {
.
.
.
```

```
        }
```
サンプル・ファイル：loader.c
参照：#rom

_ _PCB_ _
構文：_ _PCB_ _
エレメント: なし
目的: コンパイラーが PCB のときこの内部変数が設定されます。
コンパイラーによって異なるデバイスを使用するときなどに使用できます。
サンプル：
```
#ifdef      __PCB__
#device     pic16c54
#endif
```
サンプル・ファイル：ex_sqw.c
参照：__pcb__, __pch__

_ _PCM_ _
構文：_ _PCM_ _
エレメント: なし
目的: コンパイラーが PCM のときこの内部変数が設定されます。
コンパイラーによって異なるデバイスを使用するときなどに使用できます。
サンプル：
```
#ifdef      __PCM__
#device     pic16c71
#endif
```
サンプル・ファイル：ex_sqw.c
参照：__pcb__, __pch__

_ _PCH_ _
構文： _ _PCH_ _
エレメント: なし
目的: コンパイラーが PCH のときこの内部変数が設定されます。
コンパイラーによって異なるデバイスを使用するときなどに使用できます。
サンプル：
```
#ifdef      __PCH__
#device     PIC18C452
#endif
```
サンプル・ファイル：ex_sqw.c
参照：__pcb__, __pcm__

＃PRAGMA
構文：#pragma ***cmd***
エレメント: ***cmd*** は有効なプリプロセッサー・ディレクティブ
目的: このディレクティブは、C コンパイラーの互換性維持のため用意されています。
このコンパイラーでは、あえてこのディレクティブは必要ありません。
サンプル： #pragma device PIC16C54
サンプル・ファイル：ex_cust.c

＃PRIORITY
構文： #priority ***ints***
エレメント: ***ints*** はカンマ(、)によってセパレートされた１つ又は、それ以上の割り込みのリスト
目的: 割り込みの優先順位を設定します。最初に書かれた割り込みが最優先になります。
サンプル： #priorty　rtcc,rb
参照：#int_xxxx

＃RESERVE
構文：#reserve ***address*** 又は、
#reserve ***address, address, address*** 又は、
#reserve ***start:end***
エレメント: ***address*** は ROM アドレス、***start*** は最初のアドレス、***end*** は最後のアドレス
目的: コンパイラーにより使用される RAM 領域を指定したときリザーブします。このディレクティブは#DEVICE の後に記述されないと有効になりません。
サンプル：#DEVICE PIC16C74
#RESERVE 0x60:0x6f
サンプル・ファイル：ex_cust.c
参照：#org

＃ROM
構文：#rom ***address = {list};***
エレメント: ***address*** は ROM ワード・アドレス、***list*** はカンマによりセパレートされたワードのリストです。
目的: 出力ファイル中に指定されたアドレスから指定されたデータを挿入します。例では、PIC16C84 の EEPROM データエリアにデータを設定しています。また、一部のデバイスでコンフィグレーション･エリアのデータの再設定などのためにも＃ROM ディレクティブは使用されます。
サンプル： #rom 0x2100 ={1,2,3,4,5,6,7,8}
参照：#org

＃SEPARATE
構文：#separate
エレメント: なし
目的: このディレクティブは、コンパイラーが、以後の関数をインライン展開することを防ぎます。ROM スペースを節約しますが、スタックは消費されます。
デフォルトでは、コンパイラーは全ての関数を SEPARATE でコンパイルします。
サンプル：#separate
```
swapbyte( int *a, int *b ){
int t;
t = *a;
*a = *b;
*b = t;
}
```
サンプル・ファイル： ex_cust.c
参照：#inline

_ _TIME_ _
構文：_ _TIME_ _
エレメント: なし
目的: プリプロセッサ識別子はコンパイル時にコンパイルした時刻（"hh:mm:ss"形式）に置き換わります。
サンプル： printf(“Software was compiled on ”) ;
printf(__TIME__);
サンプル・ファイル： なし
参照：なし

＃TYPE
構文： #type ***standard-type=size***
エレメント: ***standard-type*** は short, int 又は、long の C キーワードです。
サイズは 1，8，16 又は,32

目的: デフォルトではコンパイラーは SHORT を 1bit、INT を 8bit、そして、LONG を 16bit として扱います。 通常C言語では INT はターゲット・プロセッサーに最も効果的なサイズで定義されます。PIC で INT を 8bit としているのはこのためです。
＃TYPE ディレクティブはコードの互換性を助けるためにタイプの変更を行います。
＃TYPE はこれらのキーワードを再定義することができます。
ノート：カンマはオプションです。

＃TYPE では 4 つの bit サイズ(1, 8, 16, 32)を使用する場合、SHORT, INT, LONG では全ての bit サイズを割り当てることができないので、通常は整数として 4 つのキーワード　INT1, INT8, INT16, INT32 を使用します。もし、＃TYPE をプログラムで使用する場合、CCS サンプルプログラム及びインクルードファイルが正しく動作しないかも知れないことに注意してください。

サンプル：#TYPE SHORT=8, INT=16, LONG=32
サンプル・ファイル： ex_cust.c

＃UNDEF
構文： #undef ***id***
エレメント: ***id*** は#define を介して定義されるプリプロセッサーid です。
目的: 定義済の識別子“id”を未定義にします。
サンプル：#if MAXSIZE<100
#undef MAXSIZE
#define MAXSIZE 100
#endif
参照 : #define

＃USE DELAY
構文： #use delay***(clock=speed)*** 又は、#use delay***(clock=speed, restart_wdt)***
エレメント: ***speed*** は定数 1-100000000(1hz から 100mhz)
目的: ビルトイン関数 DELAY_MS、DELAY_US や RS232C、I^2C などの組み込みライブラリーのために、クロックスピードを設定します。speed には、Ｈｚ（ヘルツ）単位で数値を指定します。オプションの“RESTART_WDT”を付け加えるとディレイ中の WDT のクリアを行なうコードを生成します。
サンプル：#use delay(clock=20000000)
#use delay(clock=32000,RESTART_WDT)
サンプル・ファイル：ex_sqw.c
参照 : delay_ms(), delay_us()

＃USE FAST_IO
構文： #use fast_io***(port)***
エレメント: ***port*** は A-G
目的: このディレクティブは、入出力命令のコードを作成するときに効果を持つようになるディレクティブです。高速 I/O という方法は、ディレクション・レジスターのプログラミング以外は、直接 I/O ピンを操作します。つまり I/O ポートの方向はすでに決定されているものとして入出力操作の際に方向設定は行われません。
ディレクション・レジスターが正しく set_tris_X()を介してセットされてなければいけません。
サンプル：#use fast_io(A)
サンプル・ファイル：ex_cust.c
参照 : #use fixed_io, #use standard_io, set_tris_x( )

＃USE FIXED_IO
構文： #use fixed_io***(port_outputs=pin,pin,?)***

エレメント: ***port*** は A-G, ***pin*** はデバイス .h ファイルで定義されたピン定数の１つ
目的: このディレクティブは、入出力命令のコードを作成するときに効果を持つようになるディレクティブです。固定 I/O という方法は、I/O ピンへのアクセスがあるたびにピンの入出力を決めるコードを生成します。“port”には A-G が指定され、出力とするピンを＝の後に記入します。I/O ピンはこれに一致するようにプログラムされます。スタンダード I/O と同じように RAM エリアが消費されます。
サンプル：#use fixed_io(a_outputs=PIN_A2,PIN_A3)
参照 : #use fast_io, #use standard_io

＃USE I2C
構文： #user i2c ***(options)***
エレメント: options はカンマによってセパレートされます。

MASTER-	マスタ・デバイスを設定
SLAVE-	スレーブ・デバイスを設定
SCL=pin-	SCL ピンを指定（ビットアドレス）
SDA＝pin-	SDA ピンを指定
ADDRESS=nn-	スレーブ・デバイス時のアドレスを設定
FAST-	高速 I^2C を設定
SLOW-	低速 I^2C を設定
RESTART_WDT-	I2C_READ でデータを取得中に WDT クリアを行う
NOFORCE_SW-	ハードウェア I^2C を使用する。

目的: 内蔵の I^2C ライブラリーを使って I^2C シリアルバスを利用できるように設定します。
このディレクティブを設定することにより、組み込み関数の I2C_START, I2C_STOP, I2C_READ, I2C_WRITE, I2C_POLL が利用できるようになります。
ソフトウェアで処理する関数を生成させるには、“NOFORCE_SW”をオプションに付けません。また、スレーブモードで使用するときは、内蔵 I^2C（SSP）を持たないデバイスでは設定できません。
サンプル: #use i2c(master, sda=PIN_B0, scl=PIN_B1)
#use i2c(slave, sda=PIN_C4, scl=PIN_C3,address=0xa0,NOFORCE_SW)
サンプル・ファイル： ex_extee.c with 2464.c
参照 : i2c_read(), i2c_write()

＃USE RS232
構文： #use rs232***(options)***
エレメント: ***options*** はカンマでセパレートされます。

BAUD=x	ボーレートをセット
XMIT=pin	送信ピンをセット
RCV=pin	受信ピンをセット
RESTART_WDT	GETC（）で受信待ちを行っているときに WDT を止めないようなコードを生成します。
INVERT	入出力ピンの極性を反転します。 通常、RS232 の送受信ポートは負論理です。
PARITY=x	パリティを設定します。 x には N,E,O が入ります。デフォルトは N です。
BITS=x	データ長を 5-9bit で指定します。 5-7bit 長は内蔵 SCI では使用できません。
FLOAT_HIGH	オープンコレクター出力とします。
ERRORS	受信の際のエラーを“RS232_ERRORS”に返します。 エラーが発生するとポートをリセットします。
BRGH10K	ボーレードに問題が有るチップ上で誤ボーレードを許可
ENABLE=pin	指定したピンを送信中ハイにします。これは 485 送信を可能にします。

目的: このディレクティブは、シリアル I/O のために、ボーレートと使用するピンを設定します。

このディレクティブの使用には、あらかじめ#USE DELAY によるクロックの設定が必要です。このディレクティブを設定すると、ビルトイン関数の GETCH、PUTCHAR や PRINTF 関数などが使用できるようになります。もし、STANDARD_IO 設定がなされていないならば、このディレクティブの設定前に FIX ED_IO や FAST_IO ディレクティブ設定を行なってください。
RESTART_WDT は、オプションで、GETC()関数で文字の受信待を行なっているとき、WDT を止めないためのものです。シリアル・ポートに使用するピンの極性は、INVERT キーワードをオプションに追加することにより反転させることができます。
パリティは PARITY オプションに N（なし）、E（偶数）、O（奇数）を設定します。

SCI ポートの送受信ピンに割り当てられると内蔵の SCI が使用されます。
ボーレートは、設定ボーレートに対して約 3%の誤差を持つ場合がありあます。
また、クロックの周波数の精度や安定度に影響を受けます。特に RC 発信を行なっている場合は、誤差が大きくなる場合があります。クロック設定によっては、指定されたボーレートが設定できない場合や設定が無効（まったく違ったボーレートとなる）になる場合があります。

RS232_ERRORS の定義は下記の通りです。

UART 無し：
● Bit 7 は 9bit データ・モード(get and put)のための 9 番目のビット
● Bit 6 はフローティング高速モードで

UART 有り：
● get によりのみ使用することが出来ます。
● RCSTA レジスター以外のコピー
● Bit 0 はパリティー・エラーを表示するために使用されます。

サンプル：#use rs232(baud=9600,xmit=PIN_A2,rcv=PIN_A3)
サンプル・ファイル：ex_sqw.c
参照: getc(), putc(), printf()

＃USE STANDARD_IO
構文： #USE STANDARD_IO***(port)***
エレメント: ***port*** は A-G
目的: このディレクティブは、入出力命令のコードを作成するときに効果を持つようになるディレクティブです。標準 I/O という方法は、いつでもプログラム中で I/O ピンを入力ポート、出力ポートして使用できるようコードを生成します。5X では、設定されたポート毎１バイトの RAM を必要とします。
サンプル：#use standard_io(A)
サンプル・ファイル： ex_cust.c
参照 : #use fast_io, #use fixed_io

＃ZERO_RAM
構文：＃ZERO_RAM
エレメント: なし
目的: プログラム実行に先立ち、保持する変数を内蔵レジスタに全て割り付けます。
サンプル： #zero_ram
```
void main() }
}
```
サンプル・ファイル： ex_cust.c

データ定義

データ・タイプ

データ定義を行なうための構文は、次の表に示されます。
TYPEDEF 宣言は、宣言時には実体が生成されません。（単なる宣言だけです）
メモリに割り付けられるのは、その宣言を使って定義を行なったときです。
CONST 宣言は、定数を宣言するために使用されるキーワードです。定数は、初期化が必要で、プログラム中での変更は行なえません。ポインターの定数は、許されていません。

SHORT 型は、I/O とビット操作のために効率の良いコードを出力するために用いられます。また、SHORT 型の配列と SHORT 型のポインターは許されていません。
ノート：下記の表の [ ] はオプションです。

データ定義一覧
[type-qualifier] [type-spcifier] [declarator];
enum [id] { [id [=cexpr]] ↑ １つ、又は、それ以上のカンマでセパレート
struct [id] { [type-qualifier[[*] id cexpr[cexpr]]]} 又は、 **Union** ↑ ↑ １つ、又は、それ以上 0 又は、それ以上 のセミコロンでセパレート
Typedef [type-qualifier] [type-spcifier] [declarator];

Type Qualifier	
static	変数はグローバルにアクティブで、そして、0 にイニシャライズされます。
auto	変数は処理が行われているときのみ存在します。これはデフォルトですので AUTO を使用する必要はありません。
double	リザーブされていますが、サポートされたデータ・タイプではありません。
extern	qualifier として許可されますが、影響しません。
register	qualifier として許可されますが、影響しません。

Type-Specifier	
int1	1bit 符号を定義
int8	8bit 符号を定義
int16	16bit 符号を定義
int32	32bit 符号を定義
char	8bit キャラクターを定義
float	32bit 浮動小数点を定義
short	デフォルトでは int1 と同じ
int	デフォルトでは int8 と同じ
long	デフォルトでは int16 と同じ
double	リザーブ・ワードですが、サポートされたデータ・タイプではありません。
void	指定タイプを示しません。

float を除きすべてのタイプはデフォルトでは unsigned[符号なし] ですが、unsigned 又は、signed で始めることが出来ます。Short と long は int を影響無く続けることが出来ます。
type-qualifiler は、データのスコープを示し、次の指定が行えます。
STATIC ： グローバルメモリにおかれます。

AUTO　：　関数内のみ有効で、デフォルトは AUTO で、AUTO 宣言は必要ありません。

type-id	TYPEDEF 定義型
enum	列挙型
struct	構造体
union	共用体

declarator				
[const]	[*]	id	[cexpr]	[=int]
		↑		
		0 又は、それ以上		

declarator は、データの性格を決めます。
[CONST]　[*]　id　[cexpr]　[=init]
CONST　：　定数宣言
*　：　ポインター
id　：　データ型
cexpr　：　データ名称
=init　：　初期値指定

Enum		
enum	[id]	{[id[=cexpr]]
		↑
		１つ、又は、それ以上のカンマでセパレート

ENUM キーワードの次の id は、データ型名を与えます。{}内でデータ宣言を行ないます。デフォルトは、初期値 0(ゼロ）から取られますが、=cexpr を与えることにより、任意の値を初期値として設定することができます。

struct と union				
struct	[id]	{[type-qualifier[[*]	id	cexpr[cexpr]]]}
union				
		↑		↑
		１つ、又は、それ以上のセミコロン		0 又は、それ以上

構造体、共用体の定義は次の通りです。

STRUCT [id]　{　[type-qualifier [　[*] [:cexper]　]]　}
STRUCT [id]　{　[type-qualifier [　[*] [cexper]　]]　}
UNION　[id]　{　[type-qualifier [　[*] [:cexper]　]]　}
UNION　[id]　{　[type-qualifier [　[*] [cexper]　]]　}

STRUCT と UNION の宣言は同じやり方ですが、UNION 宣言されたデータはメモリの同じ位置に取られます。id としてデータを定義する場合はビット変数を 1～8(1～8bit)を指定します。

サンプル：
```
int a,b,c,d;  // int 型変数宣言
typedef int    byte;        // typedef 宣言    byte 型を int 型とする
typedef short    bit;       //                 bit 型を short 型とする

bit    e,f;                 // short    e,f;と同じ
byte    g[3][2];            // int 型の多次元配列 g を宣言
char    *h;                 // char 型のポインター宣言
enum boolean {false,ture};// boolean は列挙型
```

```
  boolean   j; // j を boolean 型として宣言
  byte   k=5;                 //  初期化付き変数
  byte const WEEKS = 52;                // WEEKS は 52 をもつ定数として宣言
  byte const FACTORS[4] = {8,16,64,128};   // FACTORS を配列定数として宣言
  struct data_record {        //  構造体 data_record の定義
     byte   a[2];             //  メンバー   byte 型配列
     byte   b:2;              //     〃          2bit を持つ b
     byte   c:3;              //     〃          3bit を持つ c
     int    D;                //     〃          1bitD
}                             // data_record 構造体 d1、d2 を宣言
```

関数定義

関数定義

関数定義は、次のように行なわれます。

[qualifier] id ([type-specifier id]) { [stmt] }
↑ オプション　↑ 0 又は、それ以上のカンマでセパレートされます。　↑ 0 又は、それ以上のセミコロンでセパレートされます。

qualifier は、戻り値の型を示し、次の型が使用できます。

関数定義に先立ち、プリプロセッサー・ディレクティブを与えるとその関数は特別な関数として扱われます。

関数のための qualifier は下記です。
●VOID
●type-specifier
●#separate
●#inline
●#int_xxxx

このコンパイラーだけの特徴として、１つの char 型の引数を持つ関数を文字列定数のポインターを与えて呼び出すと、自動的に文字列がなくなるまで関数をループさせます。

サンプル：
```
void lcd_putc( char c ){
    ...
}

lcd_puts( "Hi There" );     // lcd_putc()が文字列分呼び出されます。
```

関数の引数

コンパイラーは、制限がありますが、関数に引数パラメータを与えられることをサポートします。これは、読みやすいコード記述のためとインライン展開で効率を上げるためにサポートされています。下記に示されるような２つの宣言は、同じ宣言方法です。パラメータを参照渡しとして宣言する方がインライン展開でより効率が高くなります。

```
funct_a(int *x, int *y){          // 一般的な書き方による
    if(*x!=5)
        *y = *x+3;
}
funct_a(&a, &b);                  // funct_a を呼び出す

funct_b(int & x,int & y){         // 参照渡しの宣言
    if(x!=5)
        y=x+3;
}

funct_b(a,b);                                 // funct_b を呼び出す
```

C ステートメントと式

プログラム構文

プログラムはファイルの次の４つの要素から出来ています。

- コメント
- プリプロセッサー・ディレクティブ
- データ定義
- 関数定義

コメント

コメントは quoted string 内を除きファイル内のどこに置いても構いません。
/* と */ の間のキャラクターは無視されます。//の後のキャラクターも無視されます。

ステートメント

ステートメント	サンプル

if(expr)stmt;[**else** stmt;]	If (x==25)
	x=1;
	else
	x=x+1;

wihle(expr)stmt;	while (get_rtcc()!=0) putc(`n`);

do stmt **while** (expr);	do {
	putc(c=getc());
	} while (c!=0);

for (expr1 ; expr2 ; expr3) stmt	for (i=1;I<=10;++i)

switch(expr) {
case cexpr: stmt;//1 又は、2 ケース
[default:stmt]
…}
	switch (cmd) {
	case 0: printf(“cmd 0“);
	break;
	case 1: printf(“cmd 1“);
	break;
	default: printf(“bad cmd“);
	break;

return [exor];	return (5);
Goto label;	goto loop;
Label:stmt;	loop: I++;
Break;	break;
Continue;	continue;
Expr;	i=1;
;	;
{[**stmt**]} ↑ 0 又は、それ以上の；でセパレート	{a=1; b=1;}

ノート：[ ]内の項目はオプションです。

式

定数:	
123	10 進定数(デシマル)
0123	8 進定数(オクタル)
0x123	16 進定数(ヘキサ)
0b010010	バイナリー
`x`	文字定数
`¥010`	8 進文字定数
`¥xA5`	16 進文字定数
`¥c`	特別な文字定数で　¥c には次の１つが入ります。 ¥n ﾗｲﾝ・ﾌｨｰﾄﾞ - ¥x0a　と同じ ¥r 改行 - ¥x0d　と同じ ¥t TAB - ¥x09　と同じ ¥b ﾊﾞｯｸ・ｽﾍﾟｰｽ - ¥x08　と同じ ¥f ﾌｫｰﾑ・ﾌｨｰﾄﾞ - ¥x0c　と同じ ¥a ベル - ¥x07　と同じ ¥v ﾊﾞｰﾃｨｶﾙ・ｽﾍﾟｰｽ - ¥x0b　と同じ ¥? ｸｴｽﾁｮﾝ・ﾏｰｸ - ¥x3f　と同じ ¥` ｼﾝｸﾞﾙ・ｸｫｰﾄ - ¥x60　と同じ ¥" ﾀﾞﾌﾞﾙ・ｸｫｰﾄ - ¥x22　と同じ ¥¥ ｼﾝｸﾞﾙ・ﾊﾞｯｸｽﾗｯｼｭ - ¥x5c　と同じ
“abcdef”	文字列（最後に null が付加されます）

識別子:	
ABCDE	変数名は 32 文字以内で宣言します。また、数字を先頭につけた変数名、関数名を宣言することは出来ません。
ID[X]	１次元の配列
ID[X][X]	多次元配列 -最大 5 次元まで取ることができます。
ID.ID	直値参照
ID->ID	アドレス参照

演算子

+	加算
+=	加算　x+=y は、x=x+y と同じ
&=	ビット論理積　x&=y は x=x&y と同じ
&	アドレス演算子　&x は x のアドレスを示す。
&	ビット論理積
^=	ビット XOR　x^=y は x=x^y と同じ
^	ビット XOR（エクスクルーシブ OR, EX-OR）
|=	ビット論理和　x|=y は x=x|y と同じ
|	ビット論理和
?:	条件演算子　x ? y=x : z=x　x が真なら y=x、偽なら z=x が実行される
--	デクリメント演算子　x--は x=x-1 と同じ
/=	除算　　x/=y は x=x/y と同じ
/	除算
==	イコール
>	より大きい　x>y は x が大きいと真
>=	以上　　x>=y は x が y より大きいか等しければ真
++	インクリメント演算子　x++は x=x+1 と同じ
*	乗算
!=	ノットイコール、等しくない
<<=	左シフト　x<<=y は x=x<<y と同じ
<	未満　　x<y は x が y 未満のであれば真
<<	左シフト
<=	以下　　x<=y は x が y に等しいか未満であれば真
&&	論理積
!	論理否定
||	論理和
%=	剰余　　x%=y は x=x%y と同じ
%	剰余
*=	乗算　　x*=y は x=x*y と同じ
*	乗算
~	１の補数　ビットを反転したもの
>>=	右シフト　x>>=y は x=x>>y と同じ
>>	右シフト
->	構造体のポインタアクセス
-=	減算　　x-=y は x=x-y と同じ
-	減算
sizeof	オペランドのバイトのサイズを決定

演算子先行順位

順降順位					
(expr)				- -expr	
!expr	～expr	++expr	expr++		expr- -
(type)expr	*expr	&value	sizeof(type)		
expr*expr	expr/expr	expr%expr			
expr+expr	expr-expr				
expr<<expr	expr>>expr				
expr<expr	expr<=expr	expr>expr	expr>=expr		
expr==expr	expr!=expr				
expr&expr					
expr^expr					
expr｜expr					
expr&&expr					
expr｜｜expr					
!value ? expr: expr					
value=expr	value+=expr	value-=expr			
value*=expr	value/=expr	value%=expr			
value>>=expr	value<<=expr	value&=expr			
value~=expr	value｜=expr	expr,expr			

expr ： 式、もしくは数値
cexpr ： 定数式、定数
stmt ： 文
label ： ラベル名

IF 文　　IF (expr) stmt [ELSE stmt]
expr が真のとき（0 以外）その後のステートメント stmt が実行されます。
ELSE 節がないとそのまま次の文に実行がうつります。
条件が偽のときで ELSE 節があると ELSE 節以降が実行されます。

WHILE 文　WHILE (expr) stmt
expr が真（0 以外）の間、ステートメント stmt が実行されます。
stmt が複文の場合は、{} で括ります。
expr の評価によっては stmt が一度も実行されない場合もあります。

DO WHILE 文　　DO stmt WHILE (expr);
一度、ステートメント stmt が実行された後、expr が評価されます。
評価が真の間は stmt が繰り返されます。

FOR 文　　FOR ([expr1]; [expr2]; [expr3]) stmt
評価式 expr2 が真の間、ステートメント stmt が実行されます。
初期値 expr1、増分 expr3 として使用される場合が最も多く、場合によっては expr すべて省略される場合があります。この場合は無限ループとなります。

SWITCH CASE 文　　SWITCH (expr)　{ [CASE cexpr: stmt] [DEFAULT: stmt] }
評価式 expr を cexpr と評価して行き一致しない場合は DEFAULT 節の stmt が実行されます。
CASE 節の stmt の終わりに BREAK 文がないと以後の CASE 節も実行されます。

RETURN 文　　RETURN [expr];
関数の終了を示します。
戻り値がある場合は expr が値を戻します。

GOTO 文　GOTO label;
ラベル label にジャンプします。ジャンプできる範囲は関数内だけで関数をまたがる GOTO 文は実行できません。

ラベル　　label : stmt
GOTO 分のジャンプ先を示します。

BREAK 文 BREAK;
　ループからの脱出や SWITCH CASE などに使用されます。
CONTINUE 文　　CONTINUE;
　ループの先頭に実行を戻します。
式文　　expr;
　演算式を含む文です。
空文　　;
　セミコロン“；”だけのからの文
式 - 式には、演算子、数値、定数などがあり、以下に示す通りです。
数式 - 数式は、次に示す演算子とともに用いられます。また、評価の優先順位は、ここに書かれた順で低くなります。
　括弧は、優先順位をつけます。
　　(expr)
　単項演算子を含む数式
　　!expr　~expr　++expr　expr++　--expr　expr--
　　(type)expr　*expr　&value　sizeof expr　sizeof(type)
　二項演算子を含む数式
　　expr * expr　expr / expr　expr % expr
　　expr + expr　expr - expr
　　expr >> expr　expr << expr
　　expr > expr　expr >= expr　expr < expr　expr <= expr
　　expr == expr　expr != expr
　　expr & expr　expr ^ expr　expr | expr
　　expr && expr　expr || expr
　三項演算子を含む数式
　　value ? expr : expr
　その他の数式
　　value = expr　value += expr　value -= expr
　　value *= expr　value /= expr　value %= expr
　　value >>= expr　value <<= expr　value &= expr
　　value ^= expr　value |= expr
　　expr,expr

三連文字
コンパイラーはある種のキーボードでは利用できない特別なキャラクターの変わりに 3 つの連続したキャラクターを使用することが出来ます。

Sequence	Same as
??=	#
??(	[
??/	¥
??)	]
??`	^
??<	{
??!	|
??>	}
??-	～

組み込み関数

組み込み関数一覧	
RS232 I/O	パラレル・スレーブ I/O
getc()	setup_psp()
putc()	psp_input_full()
fgetc()	psp_output_full()
gets()	psp_overflow()
puts()	I2C　I/O
fgets()	i2C_start()
fputc()	I2c_stop()
fputs()	I2c_read()
printf()	I2c_white()
kbhit()	I2c_poll()
fprintf()	プリプロセッサー・コントロール
set_uart_speed()	sleep()
perror()	reset_cpu()
assert()	restart_cause()
getchar	disable_interrupts()
putchar	enable_interrupts()
SPI two wire I/O	ext_int_edge()
setup_spi()	read_bank()
spi_read()	write_bank()
spi_write()	label_address()
spi_data_is_in()	goto_address()
ディスクリート I/O	getenv()
output_low()	ビット/バイト操作
output_high()	shift_right()
output_float()	shift_left()
output_bit()	rotate_right()
input()	rotate_left()
output_X()	bit_clear()
input_X()	bit_set()
port_b_pullups()	bit_test()
set_tris_X()	swap()
キャプチャー/コンペア/PWM	make8()
setup_ccpX()	make16()
set_pwmX_duty()	make32()

標準 C Math	標準 C Char
abs()	atoi()
acos()	atoi32()
asin()	atoll()
atan()	atof()
ceil()	tolower()
cos()	toupper()
exp()	isalnum()
floor()	isalpha()
labs()	isamoung()
sinh()	isdigit()
log()	islower()
log10()	isspace()
pow()	isupper()
sin()	isxdigit()
cosh()	strlen()
tanh()	strcpy()
fabs()	strncpy()
fmod()	strcmp()
atan2()	stricmp()
frexp()	strncmp()
ldexp()	strcat()
modf()	strstr()
sqrt()	strchr()
tan()	strrchr()
div	strtok()
idiv	strspn()
電圧 Ref	strcspn()
setup_vref()	strpbrk()
A/D コンバータ	strlwr()
setup_adc_ports()	sprintf()
setup_adc()	isgraph()
set_adc_channel()	iscntrl()
read_adc()	isprint()
	ispunct
	strtod()
	strtol()
	strtoul()
	strncat()
	strcoll(),strxfrm

タイマー	内部 EEPROM
setup_timer_X()	read_eeprom()
set_timer_X()	write_eeprom()
get_timer_X()	read_program_eeprom()
setup_counters()	write_program_eeprom()
setup_wdt()	read_calibration()
restart_wdt()	write_program_memory()
標準 C メモリー	read_program_memory()
memset()	write_external_memory()
memcpy()	erase_program_memory()
offsetof()	setup_external_memory()
offsetofbit()	標準 C スペシャル
malloc()	rand()
calloc()	srand()
free()	ディレイ
realloc()	delay_us()
memmove()	delay_ms()
memcmp()	delay_cycles()
memchr()	アナログ・コンペア
	setup_comparator()

ABS( )
構文：value=abs***(x)***
パラメータ：***x*** は符号無し 8,16 又は、32bit の整数、又は、float[実数]
戻り値：パラメータと同じ
機能: 整数型 j の絶対値を返します。型の異なる引数を与えたときの動作は未定義です。
対象デバイス: 全デバイス
必要：#include<stdlib.h>
サンプル：signed int target,actual;

```
. . .
error=abs(target-actual);
```

参照: labs()

ACOS( )
SIN()を参照
機能: x のアークコサイン COS^{-1}（逆余弦）を計算します。戻り値の x には、０～π までのラジアンで返されます。 x の範囲が－１～１にないときの戻り値は未定義です。

ASIN( )
SIN()を参照
機能: x のアークサイン SIN^{-1}（逆正弦）を計算します。戻り値の x には、-π/2 ~ π/2 までのラジアンで返されます。 x の範囲が－１～１にないときの戻り値は未定義です。この関数は MATH.H が含まれる必要があります。この関数は MATH.H が含まれる必要があります。

ASSERT( )
構文： assert***(condition)***
パラメータ：***condition*** は比較式
戻り値： なし
機能: この関数は condition をテストします。そして、テストが FALSE であれば STDERR へのエラーメッセージを生成します。STDERR の出力先はデフォルトでプログラム中に最初に使われた RS232 となります。エラーメッセージにはファイル名と assert()の行番号が含まれます。もし、#define NODEBUF とした場合は、assert() はエラーメッセージを生成しません。このように assert() をソースコードに埋め込むことで、必要な場合にはコードのテストが行え、テストが終了すれば簡単にテスト機能を無効にすることができます。

対象デバイス: 全デバイス
必要：assert.h と #use rs232
サンプル：assert(number_of_entries is >= TABLE_SIZE) ;
もし、number_of_entries >= TABLE_SIZE であるとき、下記のメッセージが RS232 に出力します。:
Ascertain failed in file : myfile.c at line : 56
サンプル・ファイル： なし
参照: #use rs232

ATAN( )
SIN()を参照
機能: x のアークタンジェント TAN^{-1}（逆正接）を計算します。戻り値の x には-π/2 ~ π/2 までのラジアンで返されます。

ATAN2( )
SIN()を参照

ATOF()
構文：result = atof***(string)***

パラメータ：***string*** は NULL(0x00)で終端された文字列のポインター
戻り値： 32bit 浮動小数点
機能: 指定した文字列を浮動小数点形式に変換します。
32bit 浮動小数点に変換できない場合の戻り値は未定義です。
対象デバイス: 全デバイス
必要：#include<stdlib.h>
サンプル：char string [10];
```
float x;

strcpy (string, “123.456”);
x = atof(string)
// x は 123.456
```
サンプル・ファイル： ex_tank.c
参照: atoi(), atoll(), atoi32(), printf()

ATOI( )
ATOL( )
ATOI32( )
構文：ivalue=atoi***(string)*** 又は、ivalue=atol***(string)*** 又は、i32value=atoi32***(string)***
パラメータ：***string*** は NULL(0x00)で終端された文字列のポインター
戻り値：ivalue は 8bit 整数
ivalue は 16bit 整数
i32value は 32bit 整数
機能: アスキー文字列から整数・倍長整数に変換します。デシマルとヘキサが使用できます。
対象デバイス: 全デバイス
必要： #include <stdlib.h>
サンプル：char string[10];
```
int x;

strcpy(string, “123”);
x = atoi(string);                // x は 123
```
サンプル・ファイル：input.c
参照: printf()

BIT_CLEAR( )
構文： bit_clear***(var, bit)***
パラメータ：***var*** は 8,16 又は、32bit 長の何れかの変数。***Bit*** は 0～31 の整数でビット位置を示す。
bit=0 は最下位ビット(LSB)
戻り値：未定義
機能: 渡された var の bit で指定されたビットをクリアします。bit は 0～7（var が 8bit）か 0～15（16bit）を指定します。最下位ビットは 0 です。
この関数は、次の計算式と同じです。var &= ~(1<<bit);
対象デバイス: 全デバイス
必要： なし
サンプル：int x;
```
x = 5;
bit_clear(x,2);  //  x は 5 から 1 になる
bit_clear(*11,7);                // 割り込みを禁止する粗雑な方法
```
サンプル・ファイル： ex_patg.c
参照: bit_set(), bit_test()

BIT_SET( )
構文：bit_set***(var,bit)***

パラメータ：***var*** は 8,16 又は、32bit 長の何れかの変数。***Bit*** は 0～31 の整数でビット位置を示す。
bit=0 は最下位ビット(LSB)
戻り値：未定義
機能: 渡された var の bit で指定されたビットをセットします。 bit は 0～7(var が 8bit)か
0～15(16bit)を指定します。最下位ビットは 0 です。
この関数は、次の計算式と同じです。var |= (1<<bit);
対象デバイス: 全デバイス
必要： なし
サンプル：
```
int   x;
x = 5;
bit_set(x,3);              //  x は 5 から 13 になる
bit_set(*6,1);             // ピン B1 をセットする粗雑な方法
```
サンプル・ファイル： ex_patg.c
参照: bit_clear(), bit_test()

BIT_TEST( )
構文：value = bit_test***(var,bit)***
パラメータ：***var*** は 8,16 又は、32bit 長の何れかの変数。***Bit*** は 0～31 の整数でビット位置を示す。
bit=0 は最下位ビット(LSB)
戻り値：0 又は、1
機能: 渡された var の bit で指定されたビットをテストします。bit は 0～7(var が 8bit)か 0～15(16bit)
を指定します。最下位ビットは 0 です。
この関数は、次の計算式と同じです。 ((var & (1<<bit)) != 0);
対象デバイス: 全デバイス
必要： なし
サンプル：
```
if ( bit_test(x, 3) || !bit_test(x,1) ) {
                    // ビット 3 が 1 又は、ビット 1 が 0
}

if(data!=0)
for(I=31;!bit_test(data,i);i- ;
```
サンプル・ファイル： ex_patg.c
参照: bit_clear(), bit_set()

CALLOC( )
構文：ptr=calloc***(nmem, size)***
パラメータ：***nmem*** はオブジェクトの数を表す整数です。***size*** は割り当てられるバイトの数又は、
それらの１つです。
戻り値：割り当てられたメモリーへのポインタを返します。さもなくば null を返します。
機能: calloc 関数はサイズで指定されたオブジェクトのサイズで割り当てられたメモリー容量を
割り当てます。
対象デバイス: 全デバイス
必要：STDLIB.H
サンプル:
```
int * iptr;
iptr=calloc(5,10) //iptr は 50 バイトのメモリー・ブロックにポイントされます。
```
参照: realloc(), malloc()

CEIL( )
構文：result=ceil***(value)***
パラメータ：***value*** は float[実数]です。
戻り値：float[実数]

機能: float x より大きい最小の整数を計算します。float[12.67]は 13.00 です。
対象デバイス: 全デバイス
必要：#include<math.h>
サンプル：cost = ceil (weight) // 重さの単位ポンド当りの US ドルを計算
参照: floor()

COS( )
参照 : SIN()

COSH( )
参照 : SIN()

DELAY_CYCLES( )
構文：delay_cycles***(count)***
パラメータ：***count*** は定数又は、変数 1～255
戻り値：未定義
機能: count で示されるマシンステート数のディレイを入れます。
ディレイ時間は、クロック周波数に依存し、クロック発振周波数の 4 倍となります。
対象デバイス: 全デバイス
必要： なし
サンプル：delay_cycles(1); // NOP と同じ
delay_cycles(25); // 20MHz で 5us ディレイ
サンプル・ファイル： ex_cust.c
参照: delay_us(), delay_ms()

DELAY_MS( )
構文：delay_ms***(time)***
パラメータ：***time*** は変数 0～255、又は、定数 0～065535
戻り値：未定義
機能: time ミリ秒のディレイをいれます。
time には、定数であれば 0～65535 まで入力できます。また、変数ならば 0～255 となります。
ディレイ関数は SEPARATE で呼び出されコードエリアを節約します。
対象デバイス: 全デバイス
必要：#use delay
サンプル：#use delay(clock=20000000)
delay_ms(2); // 2mS 待ち

```
void delay_seconds( int n ){          // n 秒待ちルーチン
  for (;n!=0; n- -)
  delay_ms( 1000 );
}
```

サンプル・ファイル： ex_sqw.c
参照: delay_us(), delay_cycles(), #use delay

DELAY_US( )
構文： delay_us***(time)***
パラメータ：***time*** は変数 0-255、又は、定数 0-65535
戻り値：未定義
機能: timeμ 秒のディレイをいれます。
time には、定数であれば 0～65535 まで入力できます。また、変数ならば 0～255 となります。
短いディレイ関数は INLINE 関数としてコールすることにより誤差を最小にすることができますがコードエリアは必要になります。長いディレイの場合は SEPARATE で呼び出してコードエリアを節約します。

対象デバイス: 全デバイス
必要：#use delay
サンプル：#use delay(clock=20000000)

```
do {
output_high(PIN_B0);
delay_us(duty);
output_low(PIN_B0) ;
delay_us(period-duty);
} while(TRUE);
```

サンプル・ファイル： ex_sqw.c
参照: delay_ms(), delay_cycles(), #use delay

DISABLE_INTERRUPTS( )
構文：disable_interrupts(***level***)
パラメータ：***level*** はデバイスの .h ファイルで定義された定数
戻り値：未定義
機能: level で指定されたレベルでの割込を不許可します。level に GLOBAL を与えて割込を全て不許可することができます。
対象デバイス: インターラプトを持ったデバイス(PCM と PCH)
必要：#INT_XXXX を持っている必要があります。定数は各デバイスの .h ファイルで定義されます。
割込レベルには次の指定が可能です。
・GLOBAL ： 割込許可
・ADC_DONE ： AD コンバータ変換終了
・RTCC_ZERO： RTCC オーバーフロー
・RB_CHANGE： RB ピン変化
・EXT_INT ： 外部割込信号入力
・INT_TIMER1： タイマー1 オーバーフロー
・INT_TIMER2： タイマー2 オーバーフロー
・INT_CCP1： CCP1
・INT_CCP2： CCP2
・INT_SSP ： SSP
・INT_PSP ： PSP
・INT_RDA： シリアル受信
・INT_TBE ： シリアル送信
・INT_COMP： コンパレーター
・INT_ADOF： 14000 AD コンバーター・オーバーフロー
・INT_RC ： 14000 RC ポート変化
・INT_I2C ： 14000 I2C
・INT_BUTTON： 14000 プッシュボタン
・INT_LCD ： 92 x LCD

```
サンプル：disable_interrupts(GLOBAL);          // 全ての割込オフ
          enable_interrupts(INT_RDA);          // RS232 オフ
          enable_interrupts(ADC_DONE);         // 割込許可ビット設定
          enable_interrupts(RB_CHANGE);        //          〃
          enable_interrupts(GLOBAL);           // ここで設定した割込が許可される
```

サンプル・ファイル: ex_sisr.c, ex_stwt.c
参照: enable_interrupts(), #int_xxxx

DIV()
LDIV( )
構文：idiv=div***(num, denom)***
　　idiv=ldiv***(lnum, ldenom)***
パラメータ：***num*** と ***denom*** は Signed interger
　　num は分子、そして、denom は分母
　　lnum と ***ldenom*** は Signed long
　　lnum は分子、そして、***ldenom*** は分母
戻り値：idiv は商及び剰余を含む div_t の構造体を返します。
　　ldiv は商及び剰余を含む ldiv_t の構造体を返します。
機能: div と ldiv 関数は分子 と分母による除算の商(quot)と剰余(rem)を計算します。割り切れないとき、代数的な商にもっとも近く、それより絶対値の小さい整数になります。結果を表せないときの動作は未定義です。それ以外のときは quot*denom(ldenom)+rem は num(lnum)
対象デバイス: 全デバイス
必要：#include<STDLIB.H>
サンプル：div_t idev;
　　ldiv_t lidiv;
　　idiv=div(3,2); //idiv は quot=1 と rem=1 を含みます。
　　ldiv=ldiv(300,250); //lidiv は quot=1 と rem=50 を含みます。

ENABLE_INTERRUPTS( )
構文：enable_interrupts(level)
パラメータ：level - デバイスの .h ファイルで定義された定数
戻り値：未定義
機能: level で指定されたレベルでの割込を許可します。
最初に level に GLOBAL を与えて割込を全て許可することはできません。しかし、前もって許可した意外の割込を指定することはできますので、必要な割込について許可し、その後 GLOBAL で許可します。前の DISABLE_INTERRUPTS を参照してください。
対象デバイス: インターラプトを持ったデバイス(PCM と PCH)
必要：#INT_XXXX を持っている必要があります。デバイスの .h ファイルで定義された定数
サンプル・ファイル：ex_sisr.c, ex_stwt.c
参照: disable_enterrupts(), #int_xxxx

ERASE_PROGRAM_EEPROM( )
構文：erase_program_eeprom(***address***);
パラメータ：***address*** は PCM チップでは 16bit、PCH チップでは 32bit です。最下位 bit は無視されます。
戻り値：未定義
機能: プログラム・メモリーの 0xFFFF までの FLASH_ ERASE_SIZE 分のメモリーをイレースします。
FLASH_ERASE_SIZE 値（バイト）はチップにより変わります。例えば、64 バイトでは、アドレスの最下位 6bit が無視されます。
対象デバイス: プログラム・メモリー書き込めるデバイスのみ
必要：なし
サンプル： for(i=0x1000;i<=0x1fff;i+=getenv(" FLASH_ERASE_SIZE "))
　　erase_program_memory(i);
サンプル・ファイル：なし
参照: write_program_eeprom(), write_program_memory()

EXP( )
構文： result = exp***(value)***
パラメータ：***value*** は float[実数]
戻り値：float[実数]
機能: 引数 x のエクスポネンシャル（e^x）を計算します。 x の値が大き過ぎる時の関数の結果は未定義です。
対象デバイス: 全デバイス
必要：MATH.H が含まれている必要があります。
参照: pow(), log(), log10()

EXT_INT_EDGE( )
構文： ext_int_edige***(source, edge)***
パラメータ：***source*** は PIC18 では定数 0,1 又は、2、そして、0 です。ソースがオプションでない限りデフォルトは 0 です。***Edge*** はハイからローとローからハイを現す定数 H_TO_L 又は、L_TO_H
戻り値：未定義
機能: 外部割込みのエッジ方向を設定します。
edge には“L_TO_H”（立ち上がりエッジ）又は、“H_TO_L”（立ち下がりエッジ）を指定します。
対象デバイス: インターラプトを持ったデバイス(PCM と PCH)
必要：各デバイスの .h ファイル内に定数として
サンプル： ext_int_edge (2, L_TO_H); //PIC18 EXT2 セットアップ
ext_int_edge (H_TO_L); //EXT セットアップ
サンプル・ファイル： ex_wakup.c
参照: #int_ext, enable_interrupts(), disable_interrupts()

FABS( )
構文： result=fabs(***value***)
パラメータ：***value*** は float[実数]
戻り値：float[実数]
機能: fabs 関数は float[実数]の絶対値を計算します。
対象デバイス: 全デバイス
必要：MATH.H が含まれている必要があります。
サンプル： float result;
result=fabs(-40.0); //result は 40.0
サンプル・ファイル： なし
参照: abs(), labs()

FLOOR( )
構文： result=floor(***value***)
パラメータ：***value*** は float[実数]
戻り値：float[実数]
機能: 引数 x を超えない最大の整数を返します。（小数点以下の切り捨て）
対象デバイス:全デバイス
必要：MATH.H が含まれている必要があります。
サンプル：frac=value – floor(value);
参照: ceil()

FMOD( )
構文： result=fmod***(val1, val2)***
パラメータ：***val1*** と ***val2*** は float[実数]
戻り値：float[実数]

機能：fmod 関数は val1 = i * val2 + f となる val1 / val2 の浮動小数点剰余 f を計算します。ここで i は整数です。f は val1 と同じ符号であり、その絶対値は val2 の絶対値より小さい値です。
対象デバイス:全デバイス
必要：MATH.H が含まれている必要があります。
サンプル：float result;
```
results=fmod(3,2);      // 結果は 1
```
サンプル・ファイル: なし
参照: なし

FREE( )
構文： free(***ptr***)
パラメータ：***ptr*** は calloc、malloc、又は、realloc 関数によって前に返されたポインタ
戻り値：なし
機能：free 関数により ptr が指定するスペースを解放し、さらに割り当て出来るようにします。ptr が null ポインタの場合、動作しません。それ以外の場合、実引数が calloc、malloc、又は、realloc 関数によって前に返されたポインタと一致しないとき、又は、そのエリアが free もしくは realloc の呼出しによって解放されているとき、その動作は未定義です。
対象デバイス:全デバイス
必要：STDLIBM.H が含まれている必要があります。
サンプル：int * iptr;
```
iptr=malloc(10);
free(iptr)      //iptr が再度割り当てられます。
```
サンプル・ファイル: なし
参照: realloc(), malloc(), calloc()

FREXP( )
構文： result=frexp(***value, &exp***);
パラメータ：***value*** は float[実数]
exp は signed int[符号付整数]
戻り値：float[実数]
機能：frexp()は浮動小数点数 value を仮数(m)と指数(exp) に分解し、仮数(m)の絶対値が 0.5 以上かつ 1.0 未満になるように、また value = m * (2 の exp 乗) になるようにします。
戻り値は仮数(m)となります。指数は整数値として、exp オブジェクトに格納されます。
もし、value がゼロの場合は仮数と指数ともゼロになります。
対象デバイス:全デバイス
必要：MATH.H が含まれている必要があります。
サンプル：float result;
```
signed int exp;
result=frexp(.5, $exo);      //result ha .5   そして、exp は 0
```
サンプル・ファイル: なし
参照: idexp(), exp(), log(), log10(), modf()

GETENV( )
構文：value=getenv***(cstring)***;
パラメータ：***cstring*** は認識されたキーワードをもった文字列
戻り値：定数、文字列、又は、0
機能: 実行環境の情報を取得します。指定するキーワードは下記の通りです。
キーワードごとに Value に値が設定されて戻ります。キーワードが理解されない場合は 0 が戻されます。
・FUSE_SET:fffff　　:ヒューズ fffff が有効の場合 1 を返します。
・FUSE_VALID.fffff　:ヒューズ fffff が正しい場合 1 を返します。
・INT:iiiii　　:割り込み iiiii が正しい場合 1 を返します。

・ID :#ID によってセットされたデバイス ID を返します。
・DEVICE :デバイス名を文字列で返します。(“PIC16C74”のように)
・VERSION :float としてコンパイラーのバージョンを返します。
・VERSION_STRING :文字列としてコンパイラーのバージョンを返します。
・PROGRAM_MEMORY:コードのためのメモリー・サイズを返します。
・STACK :スタック・サイズを返します。
・DATA_EEPROM :データ EEPROM のバイト数を返します。
・READ_PROGRAM :コード・メモリーが読まれますと 1 を返します。
・PIN:pb :ポート p のビット b がその部分であるとき 1 を返します。
・ADC_CHANNELS :A/D チャンネルの数を返します。
・ADC_RESOLUTION :READ_ADC()から返された bit の数を返します。
・ICD :ICD のためにコンパイルされようとするときは 1 を返します。
・SPI :デバイスが SPI を持っていますと 1 を返します。
・USB :デバイスが USB を持っていますと 1 を返します。
・CAN :デバイスが CAN を持っていますと 1 を返します。
・I2C_SLAVE :デバイスが I2C スレーブ H/W を持っていますと 1 を返します。
・I2C_MASTER :デバイスが I2C マスターH/W を持っていますと 1 を返します。
・PSP :デバイスが PSP を持っていますと 1 を返します。
・COMP :デバイスがコンパレータを持っていますと 1 を返します。
・VREF :デバイスが電圧リファレンスを持っていますと 1 を返します。
・LCD :デバイスがダイレクト LCD H/W を持っていますと 1 を返します。
・UART :H/W UART の数を返します。
・CCCPx :デバイスが CCP 番号 x を持っているとき 1 を返します。
・TIMERx :デバイスが TIMER 番号 x を持っているとき 1 を返します。
・FLASH_WRITE_SIZE:FLASH に書き込めるバイトの最小数
・FLASH_ERASE¥SIZE:FLASH で消去出来るバイトの最小数
対象デバイス: 全デバイス
サンプル：#IF getenv(“VERSION”)<3.050
#ERROR Compiler version too old for this program
#ENDIF

for(i=0;I<getenv(“DATGA_EEPROM”);i++)
write_eeprom(i,0);

#IF gedtenv(“FUSE_VALID:BROWNOUT”)
#FUSE BROWNOUT
#ENDIF

GET_TIMERx( )
構文：**value=get_timer0 ( )** **value=get_rtcc ( ) と同じ**
value=get_timer1 ( )
value=get_timer2 ( )
value=get_timer3 ( )
パラメータ：なし
戻り値：Timer1 と 3 は 16bit 整数
Timer2 は 8bit 整数
Timer0(AKA RTCC)は 8bit 整数 *PIC18 では 16bit 整数
機能: リアルタイム・カウンター(RTCC)のカウント値を返します。
また、TIMER0 と RTCC は同じカウンタのことで、TIMER1 は 16bit の値が戻されます。
それ以外の値は、8bit で返されます。
対象デバイス:Timer0 - 全デバイス

Timer1,2 – PCM デバイスのほとんど *全部ではありません。
Timer3 – PIC18XXX
必要：なし
サンプル：set_timer0 (0);
while (get_timer0() < 200);
サンプル・ファイル： ex_stwt.c
参照: set_timerx(), setup_timerx()

GETC ( )
GETCH ( )
GETCHAR ( )
FGETC ( )
構文： value=getc()
value=fgetc(stream)
パラメータ：stream はストリーム識別子(定数バイト)
戻り値：8 ビット・キャラクター
機能: RS-232C ポートから文字を読み込みます。
読み込む文字がない場合は、文字が入力されるまで待ちます。
関数の使用に先立ち、#USE RS232 ディレクティブによって、有効なポート設定がなされている必要があります。GETC()と GETCHAR()は、STDIO.H でマクロとして定義されています。
シリアル I/O ポートに関係する関数は、#USE DELAY ディレクティブでシステムのクロックを設定し、その設定からボーレートを設定しています。また、蛇足ですが、PIC の CPU の入出力レベルは、RS232C のレベルとはなっていません。(適切なレベル変換が必要です)
対象デバイス: 全デバイス
必要：#use rs232
サンプル：printf("Continue (Y, N)? ");
do {
answer=getch();
} while (answer != 'Y' && answer != 'N');
サンプル・ファイル: ex_stwt.c
参照: putc(), kbhit(), printf(), #use rs232, input.c

GETS( )
FGETS( )
構文：gets(***string***)
value=fgets(***string, stream***)
パラメータ：***string*** は文字配列へのポインター, ***stream*** はストリーム識別子(定数バイト)
戻り値：未定義
GETC()関数を使用して、シリアル・ポートからリターン・コード（13）が入力されるまで string に読み込みます。リターン・コードは string に含まれず、string の最後には NULL(0)が入れられます。INPUT.C にはより便利な GET_STRING 関数があります。もし、fgets()が使用されると、指定されたストリームは gets() がデフォルトとする STDIN として使用されます。
対象デバイス: 全デバイス
必要：#use rs232
サンプル：char string[30];
printf(“Password: “);
gets(string);
if(strcmp(string,password))
printf(“OK”);
サンプル・ファイル: なし
参照：getc(), get_string in input.c

GOTO_ADDRESS( )
構文：goto_address(location);
パラメータ：location は ROM アドレス, 16 又は、32 ビット int.
戻り値：なし
機能：この関数は location で指定されたアドレスにジャンプします。現在の関数の外部へのジャンプは注意して行わなければなりません。これは非常に特別な場合を除いて通常使用されません。
対象デバイス: 全デバイス
サンプル：
```
#define LOAD_REQUEST PIN_B1
#define LOADER 0x1f00

if(input(LOAD_REQUEST))
    goto_address(LOADER);
```
サンプル・ファイル： setjmp.h
参照:

I2C_POLL ( )
構文：i2c_poll()
パラメータ：なし
戻り値：1(TRUE)又は、0(FALSE)
機能: この関数は、内蔵の SSP を使っているときのみ使用できます。関数が TRUE を返すときハードウェアはバッファに受信データがあることを示します。関数が TRUE を返した後、I2C_READ 関数でデータを読み込みます。
対象デバイス: 内臓 I2C を持ったデバイス
必要：#use i2c
サンプル：
```
i2c_start();                    // Ｉ2C ポート使用開始
i2c_write(0xc1);                // デバイス・アドレス/読み込み
count = 0;
while(count!=4){
 while(!i2c_poll()){
 buffer[count+] = i2c_read(); // 次を読み込む
   }
}
i2c_stop();                     // Ｉ2C ポート使用停止
```
サンプル・ファイル: ex_slave.c
参照: i2c_start, i2c_write, i2c_stop, i2c_poll

I2C_READ ( )
構文：data=i2c_read(); 又は、data=i2c_read***(ack)***;
パラメータ：***ack*** はオプション。デフォルト 1
0 受信終了時にアクノリッジ・ビットを送信しない
1 受信終了時にアクノリッジ・ビットを送信する
戻り値：data – 8bit 整数
機能: I^2C インターフェースから読み込み、data を返します。
関数の使用に先立ち、#use I2C が必要で、各種の設定が行なわれていることが要求されます。
マスターモードの時、クロックを出力して、クロックに同期して data を入力します。
スレーブモードの時、マスターからのクロックを待ち、クロックが入力されると同期し data が読み込まれます。この関数は、データがあるまで待ち、タイムアウトは使用されません。
RESTART_WDT()が#USE I2C に含まれると WDT を止めずに I2C_READ 関数はデータを待ちます。
0 が返されるとき、受信したデータが無効の場合です。
必要：#use i2c
サンプル：
```
i2c_start();
i2c_write( 0xa1 );
```

```
        data1 = i2c_read();
        data2 = i2c_read();
        i2c_stop();
```

サンプル・ファイル: ex_extee.c　2416.C
参照: i2c_start, i2c_write, i2c_stop, i2c_poll

I2C_START ( )
構文：i2c_start()
パラメータ：なし
戻り値：未定義
機能: I²C マスター・モードのとき、I²C 通信が開始できる状態にします。
関数の使用に先立ち、#use I2C が必要で、各種の設定が行なわれていることが要求されます。
関数が実行されると、クロック・ラインが I2C_READ、I2C_WRITE が実行されるまでローレベルに保持されます。I2C_WRITE の例を参考にしてください。
対象デバイス: 全デバイス
必要：#use i2c
サンプル：

```
i2c_start();
i2c_write(0xa1);              // デバイス・アドレス
i2c_write(address);           // データをデバイスへ
i2c_start();                  // 再スタート
i2c_write(0xa1);              // データ変更
data2 = i2c_read();           // スレーブから読込み
i2c_stop();
```

サンプル・ファイル：ex_extee.c with 2416.C
参照：i2c_start, i2c_write, i2c_stop, i2c_poll

I2C_STOP( )
構文：i2c_stop()
パラメータ：なし
戻り値：未定義
機能: I²C マスターモードのとき、I²C 通信を停止できる状態にします。
関数の使用に先立ち、#use I2C が必要で、各種の設定が行なわれていることが要求されます。
I2C_WRITE の例を参考にしてください。
対象デバイス: 全デバイス
必要：#use i2c
サンプル：

```
i2c_start();          // I2C ポート使用開始
i2c_write(0xa1);      // デバイス・アドレス
i2c_write(5);         // デバイス・コマンド
i2c_write(12);        // デバイス・データ
i2c_stop();           // Ｉ2C ポート使用停止
```

サンプル・ファイル：ex_extee.c with 2416.C
参照：i2c_start, i2c_write, i2c_stop, i2c_poll

I2C_WRITE( )
構文：i2c_write***(data)***
パラメータ：***data*** は 8bit 整数
戻り値：この関数は ACK ビットを返します。
0 は ACK, 1 は NO ACK
機能: I²C インターフェースに byte を出力します。
関数の使用に先立ち、#use I2C が必要で、各種の設定が行なわれていることが要求されます。
マスターモードの時、クロックを出力して、クロックに同期して byte を出力します。
スレーブモードの時、マスターからのクロックを待ち、クロックが入力されると同期して byte

が出力されます。内蔵の SSP が使用されていないときは、関数はタイムアウトを検出しません。
対象デバイス: 全デバイス
必要：#use i2c
サンプル：
```
i2c_start();              // Ｉ2C ポート使用開始
i2c_write(0xa0);          // デバイス・アドレス出力
i2c_write(cmd);           // ロー・バイト・コマンド
i2c_write(cmd>>8);        // ハイ・バイト・コマンド
i2c_stop();               // Ｉ2C ポート使用停止
```
サンプル・ファイル：ex_extee.c with 2416.C
参照：i2c_start, i2c_write, i2c_stop, i2c_poll

INPUT( )
構文：value=input***(pin)***
パラメータ：***pin*** で指定されたピンの状態を読み込みます。
ピンは各デバイスのヘッダー・ファイル(*.h)で定義されています。実際の値はビット・アドレスです。例えば、ポート A(レジスター・アドレス 5)の bit3 は 5x8+3=43 となります。よって、ヘッダー・ファイルでは、#define PIN_A3 43 の様に定義されています。
戻り値：pin がローの場合は 0(又は、FALSE)
pin がハイの場合は 1(又は、TRUE)
機能: 示されたピンの状態を返します。I/O の方法は最後に指定された #USE *_IO ディレクティブに依存します。デフォルトのスタンダード I/O では、入力が行われる前にデータ・ディレクションは入力にセットされます。
対象デバイス: 全デバイス
必要：各デバイスの .h ファイルでピン定数が定義されている必要があります。
サンプル：
```
while ( ! input( PIN_B1 ) );
    if( input(PIN_A0) )
        printf("A0 is now high¥r¥n");
```
サンプル・ファイル：ex_pulse.c
参照：input_x(), output_low(), output_high(), #use xxxx_io

INPUT_x( )
構文：value=input_a()
value=input_b()
value=input_c()
value=input_d()
value=input_e()
パラメータ：なし
戻り値：8bit 整数がポート入力データを現します。
機能: ポートからの全バイトを入力。最後に指定された #USE *_IO ディレクティブに従ってディレクション・レジスターは変更されます。デフォルトのスタンダード I/O では、入力が行われる前にデータ・ディレクションは入力にセットされます。
対象デバイス: 全デバイス
サンプル・ファイル： ex_psp.c
参照：input(), output_x(), #use xxxx_io

ISAMOUNG( )
構文：result = isamoung***(value, cstring)***
パラメータ：***value*** はキャラクター、***cstring*** は文字列定義
戻り値：value が cstring に無いときは 0(又は、FALSE)
value が cstring に有れば 1(又は、TRUE)
機能: もし指定したキャラクター(value)が文字列定数(cstring)中に有れば TRUE を返します。

対象デバイス: 全デバイス
サンプル： char x;

```
…
if( isamoung( x,
      “0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ”) )
  printf(“The character is valid”);
```

サンプル・ファイル：ctype.h
参照：isalnum(), isalpha(), isdigit(), isspace(), islower(), isupper(), isdigit(9

ISALNUM(char)
ISALPHA(char)
ISDIGIT(char)
ISLOWER(char)
ISSPACE(char)
ISUPPER(char)
ISXDIGIT(char)
ISCNTRL(x)
ISGRAPH(x)
ISPRINT(x)
ISPUNCT(x)
構文：value=isalnum***(datac)***
value=isalpha***(datac)***
value=isdigit***(datac)***
value=islower***(datac)***
value=isspace***(datac)***
value=isupper***(datac)***
value=isxdigit***(datac)***
iscntrl(x) X is less than a space
isgraph(x) X is greater than a space
isprint(x) X is greater than or equal to a space
ispunct(x) X is greater than a space and not a letter or number
パラメータ：datac は 8bit キャラクター
戻り値：datac が標準にマッチしないときは 0(又は、FALSE)
datac が標準にマッチすれば 1(又は、TRUE)
機能: これらはすべて ctype.h でマクロとして定義されています。これらの関数は、文字のチェックと変換が含まれます。
対象デバイス: 全デバイス
必要：ctype.h
文字チェック関数： x が示される値を取るとき TRUE が返される。

isalnum(x)	：	'0'..'9','A'..'Z','a'..'z'
isalpha(x)	：	'A'..'Z','a'..'z'
isdigit(x)	：	'0'..'9'
islower(x)	：	'a'..'z'
isupper(x)	：	'A'..'Z'
isspace(x)	：	スペース(0x20)
isxdigit(x)	：	'0'..'9','A'..'F','a'..'f'

対象デバイス: 全デバイス
必要: ctype.h
サンプル： char id[20];

```
…
if(isalpha(id[0])) {
                valid_id=TRUE;
for(i=1;i<strlen(id);I++)
   valid_id=valid_id&& isalnum(id[i]);
} else
```

```
                        valid_id=FALSE
```
サンプル・ファイル： ex_str.c
参照：isamoung()

KBHIT( )
構文：value=kbhit()
パラメータ：なし
戻り値：getc()がキャラクターを待つ必要がある場合は 0(又は、FALSE)
getc()に対してキャラクターが用意されているときは 1(又は、TRUE)
機能: RS232C ポートの RCV（受信）ポートにスタート・ビットが入ると TRUE を返します。
関数の使用に先立ち、#USE RS232 ディレクティブによって有効なポート設定がなされている必要があります。
対象デバイス: 全デバイス
必要：#use rs232
サンプル：
```
char timed_getc() {

 long timeout;

 timeout_error=FALSE;
timeout=0;
while(!kbhit&&(++timeout<50000)) //1/2 秒
      delay_us(10);
if(kbhit() )
       return(getc() );
else {
      timeout_error=TRUE
      return(0);
     }
}
```
サンプル・ファイル： ex_tgetc.c
参照：getc(), #use rs232

LABEL_ADDRESS( )
構文： value=label_address***(label)***;
パラメータ：***label*** は関数のいかなる C ラベル
戻り値：PCB,PCM では 16 ビット int そして、PCH では 32 ビット int
機能：この関数はラベル後の次の命令の ROM アドレスを取得します。
これは非常に特別な場合を除いて通常使用されません。
対象デバイス: 全デバイス
必要：なし。
サンプル：
```
start:
    a = (b+c)<<2;
end:
   printf("It taks %1u ROM locations . ¥r¥n",
            label_address(end)-label_address(start));
```
サンプル・ファイル: setjmp.h
参照：goto_address

LABS( )
構文： result = labs***(value)***
パラメータ：***value*** は符号無し 16bit 倍長整数
戻り値：符号無し 16bit 倍長整数
機能: 倍長整数 x の絶対値を計算します。もし、結果を表わせることが出来ない場合は未定義です。

対象デバイス: 全デバイス
必要：stdlib.h が含まれる必要があります。
サンプル： if(labs(target_value – actual_value) > 500)
printf("Error is over 500
points¥r¥n")
参照：abs()

LCD_LOAD ( )
構文： lcd_load***(buffer_pointer, offset, length)***;
パラメータ：***buffer_pointer*** は LCD へ送るユーザー・データを示します。***Offset*** はデータを LCD セグメント・メモリーへ書くためのオフセットです。***length*** は転送するバイトの数です。
戻り値：未定義
機能: バッファ・ポインターbuffer_pointer で示されるバッファから length で指定されたバイト数 923/924 の LCD セグメント・データエリアのオフセット offset にロードします。
オフセットは 0～15 です。Lcd_symbol はセグメント・メモリーにデータを書く簡単な方法です。
対象デバイス: LCD ドライブ・ハードウエアを持ったデバイスにのみ利用することが出来ます。
必要：定数は各デバイスの .h ファイルで定義されます。
サンプル：lcd_load(buffer, 0, 16);
サンプル・ファイル：ex_92lcd.c
参照：lcd_symbol(), setup_lcd()

LCD_SYMBOL( )
構文: lcd_symbol ***(symbol, b7_adrs, b6_adrs, b5_adrs, b4_adrs, b3_adrs, b2_adrs, b1_adrs, b0_adrs)***
パラメータ：***symbol*** は 8bit 定数。***bX_addr*** はシンボルの bit X のために使用されるセグメント位置を現すビット・アドレスです。
戻り値：未定義
機能: 指定されたビットアドレスから 8bit のセグメント・データを LCD にロードします。
シンボルのビット 7 がセットしセグメントの B7_sdrs がセットしたときクリアされます。
それ以外のシンボルのビットも同様です。B7_adrs は LCD RAM 内のアドレスです。
対象デバイス: LCD ドライブ・ハードウエアを持ったデバイスにのみ利用することが出来ます。
必要：定数は各デバイスの .h ファイルで定義されます。
サンプル： byte CONST DIGIT_MAP[10] =
{ 0x90, 0xb7, 0x19, 0x36, 0x54, 0x50, 0xb5, 0x24 };

#define DIGIT_1_CONFIG
COM0+2, COM0+4, COM05, COM2+4, COM2+1, COM1+4, COM1+5

for(i=1; i<=9; i++){
LCD_SYMBOL(DIGIT_MAP[i], DIGIT_1_CONFIG);
delay_ms(1000);
}
サンプル・ファイル：ex_92lcd.c
参照： setup_lcd(),lcd_load()

LDEXP( )
構文： result = ldexp***(value, exp)***
パラメータ：***value*** は float[実数]です。***exp*** は signed int[符号付整数]
戻り値：結果は実数で、value と 2 の exp 乗を掛けた値となります。
機能: ldexp 関数は実数と 2 の整数乗の掛け算を実行します。
対象デバイス: 全デバイス
必要：MATH.H が含まれている必要があります。
サンプル：float result;

```
        signed int exp;
        result=ldexp(.5,0); //   結果は  .5
```
サンプル・ファイル: なし
参照：frexp(), exp(), log(), log10(), modf()

LOG( )
構文： result = log***(value)***
パラメータ：***value*** は float[実数]です。
戻り値：float[実数]
機能: 自然対数（$\log_e$）を計算します。引数 x が 0 や大きすぎる場合、結果は未定義です。
対象デバイス: 全デバイス
必要：MATH.H が含まれている必要があります。
サンプル：1nx=log(x):
サンプル・ファイル: なし
参照：log10(), exp(), pow()

LOG10( )
構文： result = log10***(value)***
パラメータ：***value*** は float[実数]です。
戻り値：float[実数]
機能: 常用対数（$\log_{10}$）を計算します。引数 x が 0 や大きすぎる場合、結果は未定義です。
対象デバイス: 全デバイス
必要：#include<math.h>
サンプル：db = log10(read_adc()*(5.0/255))*10
サンプル・ファイル: なし
参照：log(), exp(), pow()

MAKE8( )
構文： i8 = MAKER8***(var,offset)***
パラメータ：***var*** は 16 又は、32bit 整数
offset は 0,1,2 又は、3 のバイト・オフセット
戻り値：8bit 整数
機能: var からバイト単位(8bit)のデータを取り出し 8bit 変数にします。バイト単位での移動であることを除けば、i8=(((var>>(offset*8)) & 0xff)と同じです。
対象デバイス: 全デバイス
必要：なし
サンプル：
```
int32 x;
int y;
y = make8(x,3);     //x の MSB を取得
```
サンプル・ファイル：なし
参照：make16(), make32()

MAKE16( )
構文： i16 = MAKE16***(varhigh,varlow)***
パラメータ：***varhigh*** と ***varlow*** は 8bit 整数
戻り値：16bit 整数
機能: 2 つの 8bit 数から 16bit 数を作ります。パラメータが 16bit 又は、32bit の時は最下位のバイトが使用されます。パラメータを 8bit 変数としたときは、
i16=(int16)(varhigh&0xff)*0x100+(varlow&0xff)と同じです。
対象デバイス ： 全デバイス
必要：なし

サンプル：long x;
int hi, lo;
x = make16(hi,lo);
サンプル・ファイル：ltc1298.c
参照：make8(), make32()

MAKE32( )
構文： i32 = MAKE32***(var1,var2,var3,var4)***
パラメータ：***var1-4*** は 8 又は、16 bit 整数。***var2-4*** はオプション数
戻り値：32bit 整数
機能: 8 と 16bit 数のいかなる組み合わせからでも 32bit 数を作ります。パラメータの数は 1 から 4 です。パラメータの最初に指定したものが Msb（最上位バイト）になります。
但し、bit の合計が 32b 以下のときは、msb（最上位バイト）は 0 で埋められます。
対象デバイス ： 全デバイス
必要：なし
サンプル：
int32 x; int y; long z;
x = make32(1,2,3,4); // x is 0x01020304
y=0x12; z=0x4321;
x = make32(y,z); // x is 0x00124321
x = make32(y,y,z); // x is 0x12124321
サンプル・ファイル： ex_freqc.c
参照：make8(), make16()

MALLOC( )
構文： ptr=malloc***(size)***
パラメータ：***size*** は割り当てられるバイトの数を整数で表します。
戻り値：割り当てられたメモリーへのポインタを返します。さもなくば null を返します。
機能: malloc 関数はサイズで指定されたオブジェクトのサイズで割り当てられたメモリー容量を割り当てます。
対象デバイス: 全デバイス
必要：STDLIB.H
サンプル: int * iptr;
iptr=malloc(10) //iptr は 10 バイトのメモリー・ブロックにポイントされます。
参照: realloc(), free(), calloc()

MEMCPY( )
MEMMOVE()
構文： memcpy***(destination, source, n)***
memmove***(destination, source, n)***
パラメータ：***destination*** はディスティネーション・メモリーへのポインターです。
Source はソース・メモリーへのポインターです。***n*** は転送するバイトの数です。
戻り値：未定義
機能: Memcpy 関数はソースから RAM 内のデストネーションに n バイトをコピーします。
source、dest ともポインターです。
Memmove 関数はソースから RAM 内のディスティネーションに n バイト移動します。(memcpy 関数と違いソースとディスティネーションのオーバーラップが可能です。)
対象デバイス ： 全デバイス
必要: なし
サンプル： memcpy (&structA, &structB, sizeof (structA));
memcpy (arrayA, arrayB, sizeof (arrayA));
memcpy (&structA, &databyte, 1);

```
        char a[20]="hello";
        memmove(a,a+2,5);       //a は "llo"
```
参照：strcpy(), memset()

MEMSET( )
構文： memset***(destination, value, n)***
パラメータ：***destination*** はメモリーへのポインターです。***value*** は 8bit の整数です。
n は 8bit の整数です。
戻り値：未定義
機能: value 値を destination で指定した RAM アドレスから n バイト分コピーします。
配列変数名はポインターですが、他の変数名及び構造体名はポインターではないので変数名及び構造体名の前に＆が必要です。
Memmove は安全なコピーを行います（オブジェクトがオーバーラップしていても問題は発生しません）。ソースから n キャラクタをコピーする場合、最初に n キャラクタのテンポラリ配列（ソース及び destination とはオーバーラップしない）にコピーされ、テンポラリ配列から destination にコピーされます。
対象デバイス : 全デバイス
必要: なし
サンプル：
```
        memset( arrayA, 0, sizeof( arrayA ) );
        memset( arrayB, '?', sizeof( arrayB ) );
        memset( &structA, 0xFF, sizeof( structA ) );
```
参照：memcpy()

MODF( )
構文： result=modf***(value, & integral)***
パラメータ：***value*** と ***integral*** は float[実数]
戻り値：結果は float[実数]
機能: modf 関数は引数 value を整数部と小数部に分けます。整数部と小数部の符号は引数 value と同じです。整数部は実数のオブジェクト integral に格納されます。
対象デバイス : 全デバイス
必要: MATH.H が含まれていなければいけません。
サンプル：
```
        float result, integral;
        result=modf(123.987, &integral);
        // 結果は .987 と integral は 123.0000
```
サンプル・ファイル: なし
参照：なし

OFFSETOF( )
OFFSETOFBIT( )
構文：
```
    value=offsetof(stype, field);
    value=offsetofbit(stype, field);
```
パラメータ：***stype*** はストラクチャー・タイプ名。***field*** は上記のストラクチャーからのフィールドです。
戻り値：8 ビット・バイト
機能: これらの関数は構造体 stype の指定されたメンバ field のオフセットを返します。
Offsetof はバイト単位で、Offsetofbit は bit 単位でオフセット返します。
対象デバイス: 全デバイス
必要 : stddef.h
サンプル：
```
struct time_structure {
        int hour, min, sec;
        int zone : 4;
```

```
                short daylight_savings;
            }
        x = offsetof(time_structure, sec);                  // x は 2
        x = offsetofbit(time_structure, sec);               // x は 16
        x = offsetof(time_structure, daylight_savings);     // x は 3
        x = offsetofbit(time_structure, daylight_savings);// x は 28
```

サンプル・ファイル: なし
参照：なし

OUTPUT_A( )
OUTPUT_B( )
OUTPUT_C( )
OUTPUT_D( )
OUTPUT_E( )
構文： output_a***(value)***
output_b***(value)***
output_c***(value)***
output_d***(value)***
output_e***(value)***
パラメータ：***value*** は 8bit 整数
戻り値：未定義
機能: ポートからの全バイトを出力。最後に指定された #USE *_IO ディレクティブに従ってディレクション・レジスターは変更されます。
対象デバイス: 全デバイスですが、すべては A-E の全ポートを持っているわけではありません。
必要: なし
サンプル： OUTPUT_B(0xf0);
サンプル・ファイル: ex_patg.c
参照：input(), output_low(), output_high(), output_bit(), #use xxxx_io

OUTPUT_BIT( )
構文： output_bit***(pin, value)***
パラメータ：***pin*** は各デバイスのヘッダー・ファイル(*.h)で定義されています。
実際の値はビット・アドレスです。例えば、ポート A(バイト 5)の bit3 は 5x8+3、又は、43 となります。よって、ヘッダー・ファイルでは、#define PIN_A3 43 の様に定義されています。
Value は 1 又は、0
戻り値：未定義
機能: 指定されたピン pin に value（0 か１）を出力します。この関数での出力方法は、最後に指示された#USE*_IO ディレクティブに依存します。ピン指定は、INPUT 関数のところを参照してください。
対象デバイス: 全デバイス
必要：各デバイスの .h ファイルでピン定数が定義されいる必要があります。
サンプル：

```
output_bit( PIN_B0, 0 );                                // output_low(PIN_B0)と同じ
output_bit( PIN_B0, input( PIN_B1 ) );                  // B0 が B1 に追従
output_bit( PIN_B0,
            shift_left( &data, 1, input( PIN_B1 ) );
                                                        // data の MSB が B0 に出力されると
                                                    //  同時に B1  のデータが LSB にシフトされる
```

サンプル・ファイル：ex_extee.c with 9356.c
参照：input(), output_low(), output_high(), output_float(), output_x(), #use xxxx_io

OUTPUT_FLOAT( )
構文： output_float***(pin)***
パラメータ： ***pin*** は各デバイスのヘッダー・ファイル(*.h)で定義されています。

実際の値はビット・アドレスです。例えば、ポート A(バイト 5)の bit3 は 5x8+3、又は、43 となります。よって、ヘッダー・ファイルでは、#define PIN_A3 43 の様に定義されています。
戻り値：未定義
機能: 指定されたピン pin が入力に設定されます。これによりピンはハイ・インピーダンス状態となりオープン・コレクター出力のように振る舞います。ピン指定は、INPUT 関数のところを参照してください。ピン定数は各デバイスの .h ファイルで定義されます。
対象デバイス: 全デバイス
必要：各デバイスの .h ファイルでピン定数が定義されいる必要があります。
サンプル： if ((data & 0x80) == 0)
output_low(pin_A0);
else
output_float(pin_A0);
参照：input(), output_low(), output_high(), output_bit(), output_x(), #use xxxx_io
【注意】フロート状態のピンをそのままご使用なられますと、静電気などでポートがダメージを受けたり、ノイズで誤動作します。フロートとなるようなピンはきちんと処理しておいてください。（一般的にプルアップしておきます）

OUTPUT_HIGH ( )
構文： output_high***(pin)***
パラメータ：***pin*** で指定されたピンの状態を読込みます。ピンは各デバイスのヘッダー・ファイル(*.h)で定義されています。
実際の値はビット・アドレスです。例えば、ポート A(バイト 5)の bit3 は 5x8+3、又は、43 となります。よって、ヘッダー・ファイルでは、#define PIN_A3 43 の様に定義されています。
戻り値：未定義
機能: 与えられた出力ピン pin をハイレベル（ほぼ電源電圧）にします。
この関数での出力方法は、最後に指示された#USE*_IO ディレクティブに依存します。
ピン指定は、INPUT 関数のところを参照してください。
対象デバイス: 全デバイス
必要：各デバイスの .h ファイルでピン定数が定義されいる必要があります。
サンプル：output_high(PIN_A0);
サンプル・ファイル：ex_sqw.c
参照：input(), output_low(), output_float(), output_bit(), output_x(), #use xxxx_io

OUTPUT_LOW ( )
構文： output_low***(pin)***
パラメータ：***pin*** は各デバイスのヘッダー・ファイル(*.h)で定義されています。
実際の値はビット・アドレスです。例えば、ポート A(バイト 5)の bit3 は 5x8+3、又は、43 となります。よって、ヘッダー・ファイルでは、#define PIN_A3 43 の様に定義されています。
戻り値：未定義
機能: 与えられた出力ピン pin をローレベル（0V）にします。この関数での出力方法は、最後に指示された#USE*_IO ディレクティブに依存します。ピン指定は、INPUT 関数のところを参照してください。
対象デバイス: 全デバイス
必要：各デバイスの .h ファイルでピン定数が定義されいる必要があります。
サンプル：output_low(PIN_A0);
サンプル・ファイル：ex_sqw.c
参照：input(), output_high(), output_float(), output_bit(), output_x(), #use xxxx_io

PERROR( )
構文： perror*(string)*;
パラメータ： ***string*** は定数 string 又は, 文字配列(null でターミネートされた)

戻り値：なし
機能: この関数は STDERR へ指定した文字列とシステムエラーメッセージ（通常は演算エラー）を出力します。
対象デバイス: 全デバイス
必要: #use rs232, errno.h
サンプル： x = sin(y);
if(errno!=0)
perror(“problem in find_area”);
サンプル・ファイル：なし
参照：なし

PORT_A_PULLUPS( )
構文： port_a_pull-ups***(value)***
パラメータ：***value*** は TRUE 又は、FALSE
戻り値：未定義
機能: ポート A のプルアップの指定を行ないます。TRUE でプアウアップされます。FALSE で非アクティベートされます。
対象デバイス: 14 と 16 ビット・デバイス(PCM と PCH) *PCB(12bit では SETUP_COUNTERS を使用
必要: なし
サンプル：port_a_pullups(FALSE);
サンプル・ファイル：ex_lcdkb.c with kbd.c
参照：input(), input_x(), output_float(),

PORT_B_PULLUPS( )
構文： port_b_pull-ups***(value)***
パラメータ：ほとんどのデバイスで ***value*** は TRUE 又は、FALSE。いくつかのデバイスではプルアップをそれぞれのピンでおこなうことが出来ます。この場合は value は 8bit 整数で各 bit が各ポートピンに対応します。
戻り値：未定義
機能:ポート B のプルアップの指定を行ないます。TRUE でプルアップされます。FALSE で
非アクティベートされます。
対象デバイス: 14 と 16 ビット・デバイス(PCM と PCH) *PCB(12bit では SETUP_COUNTERS を使用
必要: なし
サンプル：port_b_pullups(FALSE);
サンプル・ファイル：ex_lcdkb.c with kbd.c
参照：input(), input_x(), output_float(),

POW( )
構文： f=pow***(x,y)***
パラメータ：***x*** と ***y*** と float[実数]
戻り値：float[実数]
機能: Y パワーへの X を計算します。
対象デバイス: 全デバイス
必要：#include<math.h>
サンプル：area = (size, 3.0);

PRINTF( )
FPRINTF( )
構文：printf***(string)***又は、
printf***(cstring, values…)***又は、

printf***(fname, cstring, values…)***
value=fprintf***(stream, cstring, values..)***

パラメータ：***string*** は null 終端された文字列定数、又は、文字配列です。***Values*** はカンマで
セパレートされた変数のリストです。***Fname*** はストリーム識別子（定数バイト）です。
戻り値：未定義
機能: PUTS()を利用して、string を書式付きフォーマットで送信します。
書式付き printf 関数は、RS-232C ピンに出力される文字列と指定された function に対して実行されます。書式指定は文字列で行なわれ、文字列定数で指定します。
“%”キャラクターが変数を置き換え、“%”自身を出力するには“%%”とします。
“%wt”で%に続く“w”は変数の桁数を指定します。1～9 を指定すると変数は 1～9 桁にフォーマッティングされ、01～09 を指定すれば、リーディングゼロが行なわれます。
1.1～9.9 のように小数点を入れると浮動小数点のフォーマットとなります。
“ｔ”には次の書式表示指定文字が入ります。

・C：文字
・U：符号なし int 型
・X：16 進で小文字で表現される　0x1a　→　1a
・X：16 進で大文字で表現される　0x1a　→　1A
・D：10 進数符号付きで表現される
・e:　浮動小数点を指数形式（nnnnX10ˣ　1.2e5 など）で表現されます。
・f:　浮動小数点
・Lx: 16 進で小文字表現
・Lx: 16 進で大文字表現
・lu:　10 進で小文字表現
・ld:　10 進で大文字表現
・%:　%自身を出力

対象デバイス: 全デバイス
必要： #use rs232(fname が使用されていないこと)
出力例：

フォーマット	数値を 0x12 としたとき	数値を 0xfe としたとき
%03u	018	254
%u	18	254
%2u	18	*（未定義）
%5u	18	254
%d	18	-2
%x	12	Fe
%X	12	FE
%4x	0012	00FE

サンプル：
```
byte x,y,z;
printf(“HiThere”);
printf(“RTCCVALUE=>%2x¥n¥r”, get_rtcc());
printf(“%2u %x %4X¥n¥r”, x,y,z);
printf(LCD_PUTC, “n=%v”, n);
```
サンプル・ファイル： ex_admm.c, ex_lcdkb.c
参照：atoi(), puts(), putc()

PSP_OUTPUT_FULL( )
PSP_INPUT_FULL( )
PSP_OVERFLOW( )

構文：
```
result=psp_output_full()
result=psp_input_full()
result=psp_overflow()
```

パラメータ：なし
戻り値：0(FALSE)又は、1(TRUE)
機能: パラレル・スレーブ・ポート(PSP)の状態により、TRUE か FALSE を返します。
対象デバイス: PSP ハードウエアを持ったデバイス
必要: なし
サンプル：
```
while ( psp_output_full() );
psp_data = command;
while ( !psp_input_full() );
if ( psp_overflow() )
    error = TRUE;
else
    data = psp_data;
```
サンプル・ファイル：ex_psp.c
参照：setup_psp()

PUTC( )
PUTCHAR( )
FPUTC( )
構文： putc***(cdata)***
putchar***(cdata)***
value=fputc***(cdata, stream)***
パラメータ：***cdata*** は 8 ビット・キャラクター。***Stream*** はストリーム識別子(定数バイト)
戻り値：未定義
機能: RS232 XMIT ピンへ文字を出力します。
関数の使用に先立ち、#USE RS232 ディレクティブによって有効なポート設定がなされている必要があります。PUTC()は、STDIO.H でマクロとして定義されています。
もし、fputc()が使用されると、指定されたストリームは gets() がデフォルトとする STOUT として使用されます。
対象デバイス ： 全デバイス
必要：#use rs232
サンプル：
```
putc('*');
for(i=0; i<10; i++)
    putc(buffer[i]);
putc(13);
```
サンプル・ファイル：ex_tgetc.c
参照：getc(), printf(), #use rs232

PUTS( )
FPUTS( )
構文： put***(string)***, value=fputs***(string, stream)***
パラメータ：***string*** は文字列定数、又は、文字列(null で終端された)。
Stream は stream identifier(a constant byte)
戻り値：未定義
機能: PUTC()関数を使用して、シリアルポートから string を送信します。
string 送信後、リターンコード(13)と改行コード(10)が送られます。
もし、fputc()が使用されると、指定されたストリームは gets() がデフォルトとする STOUT として使用されます。
対象デバイス ： 全デバイス
必要：#use rs232
サンプル：
```
puts( "----------" );
puts( "|     HI     |" );
puts( "----------" );
```
サンプル・ファイル: なし

参照：printf(), gets()

RAND( )
構文：　re=rand()
パラメータ：なし。
戻り値：擬似乱数
機能: rand 関数は 0 から RAND_MAX の範囲の擬似乱数の乱数系列のうち直前に実行された次の乱数を戻します。
対象デバイス ： 全デバイス
必要：#include<STDLIB.H>
サンプル：int I;
　　I=rand();
サンプル・ファイル: なし
参照：srand()

READ_ADC( )
構文：　value=read_adc([***mode***])
パラメータ：***mode*** はオプション・パラメータです。もし、使用されますと：
ADC_START_AND_READ(これはデフォルト)
ADC_START_ONLY(変換をスタートさせてリターンします。)
ADC_READ_ONLY(最後に変換された結果を読み出します。)
戻り値：8 又は、16bit 整数 *#DEVICE ADC=ディレクティブによります。
機能: AD コンバーターからデタジル値を読み込みます。
事前に、setup_adc()、setup_adc_ports()と set_adc_channel()でコンバータの設定を済ませておく必要があります。戻り値の範囲は各デバイスの A/D コンバーターにビット数と下記の様に #DEVICE ADC=ディレクティブの設定に依存します。

#DEVICE	8bit	10bit	11bit	16bit
ADC=8	00-FF	00-FF	00-FF	00-FF
ADC=10	x	0-3FF	x	x
ADC=11	x	x	0-7FF	x
ADC=16	0-FF00	0-FFC0	0-FFE0	0-FFFF

ノート：　x は未定義
対象デバイス: A/D ハードウエアを持ったデバイス
必要: #include<STDLIB.H>
サンプル：setup_adc(ADC_CLOCK_INTERNAL);
　　setup_adc(ALL_ANALOG);
　　set_adc_channel(1);
　　while (input(PIN_B0)){
　　delay_ms(5000);
　　value = read_adc();
　　printf("A/D value = %2x¥n¥r", value);
　　}
　　read_adc(ADC_START_ONLY);
　　sleep();
　　value=read_adc(ADC_READ_ONLY);
サンプル・ファイル：ex_admm.c, ex_14kad.c
参照： setup_adc()、setup_adc_ports()と set_adc_channel(), #device

READ_BANK( )
構文：　value=read_bank***(bank, offset)***
パラメータ：***bank*** は物理 RAM バンク 1-3（デバイスに依存）, ***offset*** はそのバンク(0 でスタート)のユーザーRAM へのオフセット

戻り値：8bit 整数
機能: 指定されたメモリー・バンクのユーザーRAM からデータ・バイト読込みます。この関数はオート変数によるフル RAM アクセスが十分でないデバイスに使用されます。
57 チップでは、bank は 1～3 で、offset は、0～15 です。
PCM では、bank は 1 で、オフセットは 0 から取ります。
対象デバイス: PCB ではメモリー1Fh 以上, PCM ではメモリーFFh 以上
必要: なし
サンプル：どのようにデータを得るかをみるには write_bank サンプルを見て下さい。

```
データをバッファから LCD に移動します。
i=0;
do {
    c=read_bank(1,i++);
    if(c!=0x13)
        lcd_putc(c);
} while (c!=0x13);
```

サンプル・ファイル：ex_psp.c
参照：write_bank()と FAQ(よくある質問)

READ_CALIBRATION ( )
構文： value=read_calibration***(n)***
パラメータ：***n*** は 0 で始まるカリブレーション・メモリーへのオフセット
戻り値：8bit バイト

機能: 14000 のカリブレーション・メモリの場所 “n”番地を読み出します。
対象デバイス: PIC14000
サンプル： fin=read_calibration(16);
サンプル・ファイル：ex_14kad.c with 14kcal.c

READ_EEPROM ( )
構文： value=read_eeprom***(address)***
パラメータ：***address*** は 8bit 整数
戻り値：8bit 整数
機能: 指定されたデータ EEPROM アドレスからバイトを読み込みます。アドレスは 0 から始まりますが、範囲はチップによります。このコマンドは内蔵 EEPROM を持ったデバイスのみに使用出来ます。
対象デバイス: 内臓 EEPROM を持ったデバイス
必要: なし
サンプル：

```
#define last_volume 10
 volume = read_EEPROM(LAST_VOLUME);
```

サンプル・ファイル：ex_intee.c
参照：write_eeprom()

READ_PROGRAM_MEMORY( )
READ_EXTERNAL_MEMORY( )
構文： read_program_memory***(address, dataptr, count);***
read_external_memory***(address, dataptr, count);***
パラメータ：***address*** は PCM デバイスでは 16bit、PCH デバイスでは 32bit
PCM では最下位 bit はつねに 0。***dataptr*** は１つ又は、それ以上のバイトへのポインター。
count は８ビット整数
戻り値：未定義
機能:
対象デバイス: プログラム・メモリーから読み出しできるデバイスのみ
必要: なし

サンプル：char buffer[64];
read_external_memory(0x40000, buffer, 64);
サンプル・ファイル：なし
参照：write_program_memory()

REALLOC( )
構文：realloc***(ptr, size)***
パラメータ：***ptr*** は calloc, malloc 又は、realloc 関数で返された null ポインター又は、ポインターです。***size*** は割り当てられるべきバイトの数を表す整数
戻り値：割り当てられたメモリーが移動されたポインタを返します。さもなくば null を返します。
機能: realloc 関数はサイズで指定されたサイズが ptr によって割り当てられたオブジェクトのサイズを変更します。
対象デバイス: 全デバイス
必要：STDLIB.H
サンプル: int * iptr;
iptr=malloc(10)
realloc(iptr, 20)
//iptr は可能な場合 20 バイトのメモリーのブロックにポイントされます。
参照: malloc(), free(), calloc()

READ_PROGRAM_EEPROM ( )
構文： value=read_program_eeprom***(address)***
パラメータ：***address*** は PCM では 16bit, PCH では 32bit
戻り値：PCM では 16bit, PCH では 32bit
機能: プログラム・メモリーからデータを読み込みます。
対象デバイス: プログラム・メモリーからデータを読みこむことの出来るデバイス
必要: なし
サンプル：checksum = 0
for(i=0;i<8196;i++)
checksum^=read_program_eeprom(i);
printf(“Checksum is %2X¥r¥n”,checksum);
サンプル・ファイル: なし
参照：write_program_eeprom(), write_eeprom(), read_eeprom()

READ_PROGRAM_MEMORY( )
READ_EXTERNAL_MEMORY( )
構文： read_program_memory(address, dataptr, count);
read_external_memory(address, dataptr, count);

パラメータ：address は PCM デバイスでは 16bit、PCH デバイスでは 32bit
PCM では最下位 bit はつねに 0。dataptr は１つ又は、それ以上のバイトへのポインター。
count は８ビット整数
戻り値：未定義
機能: dataptr から address までの count バイトをプログラム・メモリーから読み出します。
対象デバイス: プログラム・メモリーから読み出しできるデバイスのみ
必要: なし
サンプル：char buffer[64];
read_external_memory(0x40000, buffer, 64);
サンプル・ファイル：なし
参照：write_program_memory()

RESET_CPU( )
構文： reset_cpu()
パラメータ：なし
戻り値：なし
機能: 汎用目的のデバイス・リセットです。PCB, PCM では 0 にジャンプします。PIC18 ではパワーアップ・ステートにレジスターをリセットします。
対象デバイス: 全デバイス
必要: なし
サンプル：if(checksum!=0)
reset_cpu();
サンプル・ファイル: なし
参照: なし

RESTART_CAUSE ( )
構文： value=restart_cause()
パラメータ：なし
戻り値：PCM では 16bit, PCH では 32bit
機能: CPU がリスタートした要因を返します。戻り値は次のうちから１つが要因により返されます。
WDT_FROM_SLEEP ： スリープ・モードから起動した
WDT_TIME_OUT ： WDT タイムアウト割込で起動した
MCLR_FROM_SLEEP ： マスター・リセットで起動した
NORMAL_POWER_UP： 通常のパワーON で起動した
対象デバイス: 全デバイス
必要：定数が各デバイスの .h ファイルで定義されている必要があります。
サンプル： switch (restart_cause()){
case WDT_FROM_SLEEP:
case WDT_TIMEOUT:
handle_error();
}
サンプル・ファイル：ex_wdt.c
参照： restart_wdt(), reset_cpu()

RESTART_WDT ( )
構文： restart_wdt()
パラメータ：なし
戻り値：未定義
機能: ウォッチドッグ・タイマーをリスタートさせます。
ループなどでステータスをポーリングしているときなど、WDT でプロセッサーがリセットしないようにこの関数を使用します。この関数の使用に先立ち、WDT をイネーブルにして、WDT の設定を完了しておかなければこの関数を呼び出しても、その効果を得られない場合があります。

	PCB/PCM	PCH
Enable/Disable	#fuses	set_wdt()
Timeout time	setup_wdt()	#fuses
Restart	restart_wdt()	restart_wdt()

対象デバイス: 全デバイス
必要：#fuses
サンプル：#fuses WDT //PCB/PCM の例
//PIC18 では setup_wdt を見て下さい。
Example
Main() }
setup_wdt(WDT_2304MS);
while (TRUE){

```
                restart_wdt();
                perform_activity();
            }
        }
```

サンプル・ファイル：ex_wdt.c
参照：#fuses, setup_wdt()

ROTATE_LEFT( )
構文： rotate_left***(address, bytes)***
パラメータ：***address*** はメモリーへのポインタ、***bytes*** はバイト数のカウント
戻り値：未定義
機能: 構造体や配列から左ビットローテーションを行ないます。
address には、配列や構造体のアドレスを渡します。（&data と同じ）
bytes には、ローテートに必要なバッファサイズをバイト値で与えます。
また、byte は操作に十分な容量が必要です。
対象デバイス: 全デバイス
必要：なし
サンプル：

```
x = 0x86;
rotate_left( &x, 1 );   // 0x86 が 0x0d となる
```

参照： rotate_right(), shift_left(), shift_right()

ROTATE_RIGHT( )
構文： rotate_right***(address, bytes)***
パラメータ：***address*** はメモリーへのポインタ、***bytes*** はバイト数のカウント
戻り値：未定義
機能: 構造体や配列から右ビット・ローテーションを行ないます。address には配列や構造体のアドレスを渡します。（&data と同じ）bytes には、ローテートに必要なバッファサイズをバイト値で与えます。また、byte は操作に十分な容量が必要です。RAM の最下位バイトのビット 0 を LSB とみなします。
対象デバイス: 全デバイス
必要：なし
サンプル：

```
//cell を順に回転させる  1->2, 2->3, 3->4, 4->1
struct {
int cell_1   : 4;
int cell_2   : 4;
int cell_3   : 4;
int cell_4   : 4; } cells;

rotate_right( &cells, 2 );
rotate_right( &cells, 2 );
rotate_right( &cells, 2 );
rotate_right( &cells, 2 );
```

参照： rotate_right(), shift_left(), shift_right()

SET_ADC_CHANNEL( )
構文： set_adc_channel***(chan)***
パラメータ：***chan*** は選択するチャンネル番号。チャンネル番号は 0 から始まります。そして、データシート AN0, AN1 にラベルされます。
戻り値：未定義
機能: 次に READ_ADC で変換されるチャンネルを指定します。
71 チップは、0～3（4ch）で、74 チップは 0～7（8ch）を指定できます。
72 チップは、0～4（5ch）が指定できます。

チャンネル選択後､変換を行なうまで少しのウェイトが必要です｡また､一度チャンネルを設定し、同じチャンネルを続けて変換する場合、チャンネル指定を行なう必要はありません。
異なったチャンネルを変換するとき指定します。
対象デバイス: A/D ハードウエアを持ったデバイス
必要：なし
サンプル：set_adc_channel(2);
delay_us(10);
value = read_adc();
サンプル・ファイル： ex_admm.c
参照：read_adc(), setup_adc(), setup_adc_ports()

SET_PWM1_DUTY( )
SET_PWM2_DUTY( )
SET_PWM3_DUTY( )
SET_PWM4_DUTY( )
SET_PWM5_DUTY( )
構文： set_pwm1_duty***(value)***
set_pwm2_duty***(value)***
set_pwm3_duty***(value)***
set_pwm4_duty***(value)***
set_pwm5_duty***(value)***
パラメータ：***value*** は 8 又は、16bit の定数又は、変数
戻り値：未定義
機能: 10bit の値で PWM のデューティを下記の様に設定します。0x200 で 50%となります。
最下位ビット(LSB)が必要とされない場合、8bit の値が使用されます。
・value*(1/clock)*t2div
クロックがオシレータ周波数で、t2div が timer 2 プリスケーラ(setup_timer2 にコールをセットします。)
対象デバイス: CCP/PWM ハードウエアを持ったデバイス
必要：なし
サンプル：long duty

duty = 520; // .000416/(16*(1/20000000)
set_pwm1_duty(duty);
サンプル・ファイル： ex_pwm.c
参照：setup_ccpX()

SET_RTCC( )
SET_TIMER0( )
SET_TIMER1( )
SET_TIMER2( )
SET_TIMER3( )
SET_TIMER4( )
構文： set_timer0(value) 又は、set_rtcc(value)
set_timer1(value)
set_timer2(value)
set_timer3(value)
set_timer3(value)
パラメータ：Timer1 と 3 は 16bit
Timer2 は 8bit
Timer0(AKA RTCC)は 8bit 整数 *16bit 整数が必要な PIC18 は除く
戻り値：未定義
機能: リアルタイム・カウンター・タイマーの値を設定します。
TIMER0 と RTCC は同じカウンターで、TIMER1 には、16bit の値を設定します。

それ以外の値は、8bit 値を設定します。
対象デバイス:Timer0 - 全デバイス
Timer1,2 – PCM デバイスのほとんど *全部ではありません。
Timer3 – PIC18XXX
必要：なし
サンプル：//20MHz クロック、プリスケーラなし、セット・タイマー0、35us でオーバーフロー
set_timer0(81): //256-(.000035/(4/20000000))
サンプル・ファイル：ex_patg.c
参照：set_timerX(), get_timerX()

SET_TRIS_A( )
SET_TRIS_B( )
SET_TRIS_C( )
SET_TRIS_D( )
SET_TRIS_E( )
構文: set_tris_a***(value)***
set_tris_b***(value)***
set_tris_c***(value)***
set_tris_d***(value)***
set_tris_e***(value)***
パラメータ：***value*** は I/O ポートのビットを現す各ビットの 8bit の整数
戻り値：未定義
機能: トライステート(3's ステート)レジスタの設定を行ないます。FAST_IO でポートが使用されていることが必要で I/O ポートはメモリー・アクセスのように#BYTE ディレクティブが使用されます。設定される数値はそれぞれのビットに対応した数値で、１を与えると入力、０を与えると出力ポートとなります。
対象デバイス: 全デバイス(すべての I/O ポートを持ったデバイスではありません。)
必要：なし
サンプル：set_tris_b(0x0f); // B7,B6,B5,B4 は出力、B3,B2,B1,B0 は入力
サンプル・ファイル： lcd.c
参照：#user xxxx_io

SET_UART_SPEED( )
構文: set_uart_speed***(baud)***
パラメータ：***baud*** はビット数/秒を現す定数 100～115200value は I/O ポートのビットを現す各ビットの 8bit の整数
戻り値：未定義
機能: 内蔵ハードウエア RS232 のボーレードを変更します。
対象デバイス: 内蔵 UART を持ったデバイス
必要：#use rs232
サンプル：ピン B0 と B1 の設定に基づくボーレートを設定

```
switch( input_b() & 3 ) {
    case0 : set_uart_speed(2400);   break;
    case1 : set_uart_speed(4800);   break;
    case2 : set_uart_speed(9600);   break;
   case3 : set_uart_speed(19200);   break;
}
```

サンプル・ファイル：loader.c
参照：#use rs232, putc(), getc()

SETUP_ADC(mode)
構文: setup_adc***(mode)***

パラメータ：***mode*** – アナログからデジタル・モード。有効なオプションはデバイスに依存します。デバイスの .h ファイルを見てください。いくつかの典型的なオプションは, ADC_OFF 又は、ADC_CLOCK_INTERNAL を含みます。
戻り値：未定義
機能: 内蔵 AD コンバーターの設定を行ないます。14000 チップの場合この関数で充電電流を設定します。14000 以外の設定モードは以下の通りです。

```
ADC_OFF
ADC_CLOCK_DIV_2
ADC_CLOCK_DIV_8
ADC_CLOCK_DIV_32
ADC_CLOCK_INTERNAL
```

14000 の設定モード

```
CURRENT_0, CURRENT_2, CURRENT_4, CURRENT_7, CURRENT_9,
CURRENT_11, CURRENT_13, CURRENT_16, CURRENT_18, CURRENT_20,
CURRENT_22, CURRENT_25, CURRENT_27, CURRENT_29, CURRENT_31
CURRENT_34
```

対象デバイス: 内蔵 AD コンバータを持ったデバイス
必要：デバイスの .h ファイルで定数が定義されている必要があります。
サンプル：

```
setup_adc_ports( ALL_ANALOG );
setup_adc( ADC_CLOCK_INTERNAL );
set_adc_channel( 0 );
value = read_adc();
setup_adc( ADC_OFF );
```

サンプル・ファイル：ex_admm.c
参照：setup_adc_ports, set_adc_channel, read_adc, #device. .h ファイル

SETUP_ADC_PORTS( )
構文: setup_adc_ports***(value)***
パラメータ：***value*** は .h ファイルで定義された定数
戻り値：未定義
機能: ポート A のアナログ／デジタル／混合の各モードに設定いします。
設定は、次の値を取ります。
71 チップ

NO_ANALOGS	：	すべてデジタル・ポート
ALL_ANALOG	：	全てアナログ入力ポート（0～5V）
ANALOG_RA3_REF	：	0,1,2 ポートはアナログ入力、3 はリファレンス
RA0_RA1_ANALOG	：	2,3 ポートはデジタル・ポート（74 は使用できません）

73/74 チップには次の値を設定します。

ALL_ANALOG	：	すべてアナログ
ANALOG_RA3_REF	：	RA3 はリファレンス、その他はアナログ
A_ANALOG	：	RA はアナログ、RE はデジタル
A_ANALOG_RA3_REF	：	RA3 はリファレンス、その他の RA はアナログ、RE はデジタル
RA0_RA1_RA3_ANALOG	：	RA0,1,3 以外はデジタル
RA0_RA1_ANALOG_RA3_REF	：	RA0,1 はアナログ、3 はリファレンス、その他デジタル
NO_ANALOGS	：	すべてデジタル

対象デバイス: 内蔵 AD コンバータを持ったデバイス
必要：デバイスの .h ファイルで定数が定義されている必要があります。
サンプル：

```
setup_adc_ports( ALL_ANALOG );
setup_adc_ports( A0_RA1_ANALOGRA3_REF );
```

サンプル・ファイル：ex_admm.c
参照：setup_adc(), read_adc(), set_adc_channel()

SETUP_CCP1( )
SETUP_CCP2( )
SETUP_CCP3( )
SETUP_CCP4( )
SETUP_CCP5( )
構文: setup_ccp1***(mode)***
setup_ccp2***(mode)***
setup_ccp3***(mode)***
setup_ccp4***(mode)***
setup_ccp5***(mode)***
パラメータ：***mode*** は定数。 .h ファイルで定義された定数
mode に与える値は次の通りです。
CCP_OFF ： CCP を使用しません。
CCP をキャプチャー・モードにセット
CCP_CAPTURE_FE ： キャプチャー・モード 立ち下がりエッジ
CCP_CAPTURE_RE ： 〃 立ち上がりエッジ
CCP_CAPTURE_DIV_4 ： 〃 1/4
CCP_CAPTURE_DIV_16 ： 〃 1/16
CCP をコンペア・モードにセット
CCP_COMPARE_SET_ON_MATCH ： コンペア・モード CCPX ピン'H'
CCP_COMPARE_CLR_ON_MATCH ： 〃 CCPX ピン'L'
CCP_COMPARE_INT ： 〃 割込
CCP_COMPARE_RESET_TIMER ： 〃 タイマー・リセット
CCP を PWM モードにセット
CCP_PWM ： PWM モード
CCP_PWM_PLUS_1 ： PWM モード 8bit デューティ
CCP_PWM_PLUS_2 ： 〃
CCP_PWM_PLUS_3 ： 〃
戻り値：未定義
機能: CCP を初期化します。
CCP カウンタは、次の変数でアクセスされます。
CCP_1_LOW, CCP_1_HIGH
CCP_2_LOW, CCP_2_HIGH
CCP は３つのモードで操作されます。
キャプチャー・モードでは入力ピンにイベントが起こったときタイマー1 のカウント値を CCP_x にコピーします。コンペア・モードではタイマー1 と CCP_x がイコールのとき、トリガーします。
PWM モードでは正弦波を発生します。
コンペア・モードではタイマー1 と CCP_x が等しくなったとき、トリガーします。
PWM モードでは方形波を発生します。
PCW ウィザードは、特定用途のためにモードとタイマを設定するのを助けるでしょう。

対象デバイス: CCP ハードウエアを持ったデバイス
必要：デバイスの .h ファイルで定数が定義されている必要があります。
サンプル：setup_ccp1(CCP_CAPTURE_RE);
サンプル・ファイル：ex_pwm.c, ex_ccpmp.c, ex_ccp1s.c
参照： set_pwmX_duty()

SETUP_COMPARATOR( )
構文: setup_comparator***(mode)***
パラメータ：***mode*** は定数。 .h ファイルで定義された定数
A0_A3_A1_A2

A0_A2_A1_A2
NC_NC_A1_A2
NC_NC_NC_NC
A0_VR_A2_VR
A3_VR_A2_VR
A0_A2_A1_A2_OUT_ON_A3_A4
A3_A2_A1_A2

戻り値：未定義
機能: これらは、C1-、C1+、C2-、C2+へ接続されます。また、A0→AN0、. . . に対応しています。接続の具体的な内容につきましては、マイクロチップのデータブックを参照してください。
対象デバイス: アナログ・コンパレータを持ったデバイス
必要：デバイスの .h ファイルで定数が定義されている必要があります。
サンプル：setup_compararor(A0_A3_A1_A2);　// 2 つのｺﾝﾊﾟﾚｰﾀを使用
サンプル・ファイル：ex_comp.c

SETUP_COUNTERS( )
構文: setup_counters ***(rtcc_state, ps_state)***
パラメータ：***rtcc_state*** は.h ファイルで定義された定数の１つ
rtcc_state に使用できる値
RTCC_INTERNAL　：　内部クロックを利用(fosc/4)
RTCC_EXT_L_TO_H　：　外部クロックの立ち上がりエッジ
RTCC_EXT_H_TO_L　：　外部クロックの立ち下がりエッジ
ps_state は.h ファイルで定義された定数の１つ
ps_state に使用できる値

RTCC_DIV_2	：	プリスケーラ　1/2
RTCC_DIV_4	：	〃　1/4
RTCC_DIV_8	：	〃　1/8
RTCC_DIV_16	：	〃　1/16
RTCC_DIV_32	：	〃　1/32
RTCC_DIV_64	：	〃　1/64
RTCC_DIV_128	：	〃　1/128
RTCC_DIV_256	：	〃　1/256
WDT_18MS	：	WDT プリスケーラ 1/1
WDT_32MS	：	〃　1/2
WDT_72MS	：	〃　1/4
WDT_144MS	：	〃　1/8
WDT_288MS	：	〃　1/16
WDT_576MS	：	〃　1/32
WDT_1152MS	：	〃　1/64
WDT_2304MS	：	〃　1/128

戻り値：未定義
機能: RTCC か WDT のセットアップを行ないます。
rtcc_state は、RTCC 出力の出力先とその駆動方法を決めます。
ps_state は、プリスケーラの設定を行ないます。
RTCC のプリスケーラを設定すると WDT は WDT_18MS に設定され、WDT プリスケーラの設定を行なうと、RTCC
は RTCC_DIV_1 に設定されます。
対象デバイス: 全デバイス
必要：デバイスの .h ファイルで定数が定義されている必要があります。
サンプル：setup_counters(RTCC_INTERNAL, WDT_2304MS);
参照：setup_wdt(), setup_timer(), device .h ファイル

SETUP_EXTERNAL_MEMORY( )
構文： setup_external_memory***(mode)***
パラメータ：***mode*** はデバイスヘッダーファイルで定義されている 1 つ又は複数の定数で、複数指定する場合は | でつなぎます。
戻り値：未定義
機能: 外部メモリー・バスのモードを設定します。
対象デバイス: 外部メモリーが使用できるデバイスのみ
必要：デバイスの .h ファイルで定数が定義されている必要があります。
サンプル： setup_external_memory(EXTMEM_WORD_WRITE
｜EXTMEM_WAIT_0);
setup_external_memory(EXTMEM_DISABLE);
サンプル・ファイル：なし
参照：write_program_eeprom(), write_program_memory()

SETUP_LCD( )
構文: set_lcd***(mode, prescale, segments)***
パラメータ：***mode*** には次の中から 1 つを選択します。
LCD_DISABLE：ディセーブル
LCD_STATIC： スタティック動作
LCD_MUX12： ダイナミック動作
LCD_MUX13： ダイナミック動作
LCD_MUX14： ダイナミック動作
更に次の値を必要に応じて or します。
STOP_ON_SLEEP：停止後スリープとする
USE_TIMER_1： タイマー1 を使用する
Prescale[プリスケール値]は 1～15 でセグメント segments は次の中から選択します。
SEG0_4, SEG5_8, SEG9_11, SEG12_15, SEG16_19, SEG20_26, SEG27_28, SEG29_31, ALL_LCD_PINS
戻り値：未定義
機能: 923/924 チップの LCD コントローラーの設定を行います。
対象デバイス: 内蔵 LCD ドライブ・ハードウエアを持っているデバイス
必要：デバイスの .h ファイルで定数が定義されている必要があります。
サンプル：setup_lcd(LCD_MUX14｜STOP_ON_SLEEP,2,ALL_LCD_PINS);
サンプル・ファイル：ex_92lcd.c
参照： lcd_symbol(), lcd_load()

SETUP_PSP( )
構文： setup_psp***(mode)***
パラメータ：***mode*** は次の 2 通りです。
PSP_ENABLE
PSP_DISABLE
戻り値：未定義
機能: PSP(パラレル・スレーブ・ポート)を初期化します。
SET_TRIS_E(value)関数で、データの入出力方向が決まります。データは、PSP_DATA 変数で読み書きされます。
対象デバイス: PSP ハードウエアを持っているデバイス
必要：デバイスの .h ファイルで定数が定義されている必要があります。
サンプル： setup_psp(PSP_ENABLED);
サンプル・ファイル： ex_psp.c
参照：set_tris_e()

SETUP_SPI( )
構文：setup_spi***(mode)***
パラメータ：***mode*** には、次の値を or で接続して設定します。
SPI_MASTER： マスターモード
SPI_SLAVE ： スレーブモード
SPI_SS_DISABLED： スレーブセレクトピン禁止
SPI_L_TO_H ： 立ち上がりエッジ
SPI_H_TO_L ： 立ち下がりエッジ
SPI_CLK_DIV_4 ： クロック分周 1/4
SPI_CLK_DIV_16 ： 〃 1/16
SPI_CLK_DIV_32 ： 〃 1/32
SPI_CLK_DIV_T2 ： 〃 タイマー2
戻り値：未定義
機能: SPI(シリアル・ポート・インターフェース)を初期化します。共通クロック/データ・プロトコルに準拠した 2 又は、3 線シリアル・デバイスに使用されます。
対象デバイス: SPI ハードウエアを持っているデバイス
必要：デバイスの .h ファイルで定数が定義されている必要があります。
サンプル：setup_spi(spi_master | spi_l_to_h | spi_clk_div_16);
サンプル・ファイル：ex_spi.c
参照： spi_write(), spi_read(), spi_data_in()

SETUP_TIMER_0( )
構文：setup_timer_0***(mode)***
パラメータ：***mode*** は、｜で接続された or の各グループから１つの定数
RTCC_INTERNAL, RTCC_EXT_L_TO_H 又は、RTCC_EXT_H_TO_L
RTCC_DIV_2
RTCC_DIV_4
RTCC_DIV_8
RTCC_DIV_16
RTCC_DIV_32
RTCC_DIV_64
RTCC_DIV_128
RTCC_DIV_256
PIC18 のみ
RTCC_OFF
RTCC_8_BIT
戻り値：未定義
機能: タイマー0(aka RTCC)を設定します。
対象デバイス: 全デバイス
必要：デバイスの .h ファイルで定数が定義されている必要があります。
サンプル： setup_timer_0(RTCC_DIV_2｜RTCC_EXT_L_TO_H);
サンプル・ファイル：ex_stwtc
参照： get_timer0(), setup_timer0(), setup_counters()

SETUP_TIMER_1( )
構文：setup_timer_1***(mode)***
パラメータ：***mode*** には、下記に示される値を設定します。｜で接続された or の異なったグループからの定数
T1_DISABLED ： タイマー1 を使用しない
T1_INTERNAL ： 内部クロックを使用する
T1_EXTERNAL ： 外部クロックを使用する
T1_EXTERNAL_SYNC ： 外部信号に同期
T1_CLK_OUT ： タイマー1 の出力をピンへ

T1_DIV_BY_1　　：　プリスケーラ設定 1/1
T1_DIV_BY_2　　：　〃　1/2
T1_DIV_BY_4　　：　〃　1/4
T1_DIV_BY_8　　：　〃　1/8
戻り値：未定義
機能: タイマー1 を初期化します。タイマー値は GET_TIMER1 や SET_TIMER1 を使って読み書きすることが出来ます。タイマー1 は 16bit タイマーです。タイマー1 ハードウエアを持ったデバイスでのみ使用することが出来ます。
対象デバイス: Timer1 を持ったデバイス
必要：デバイスの .h ファイルで定数が定義されている必要があります。
サンプル：setup_timer_1(T1_DISABLED);
　　setup_timer_1(T1_INTERNAL | T1_DIV_BY_4);
　　setup_timer_1(T1_INTERNAL | T1_DIV_BY_8);
　　// 20MHz クロックの時、1.6μS でカウントされ、104.8576m S で
　　// オーバーフローする（カウンタを 0xffff 設定時）
サンプル・ファイル：ex_patg.c
参照： get_timer1()

SETUP_TIMER_2( )
構文：setup_timer_2***(mode,periold,postscale)***
パラメータ：***mode*** は、下記の値を設定します。
T2_DISABLED　：　タイマー2 を使用しない
T2_DIV_BY_1　：　プリスケーラ設定　1/1
T2_DIV_BY_4　：　〃　1/4
T2_DIV_BY_16　：　〃　1/16
Period[カウンタ]は 8bit で 0～255 までを設定します。リセット時は 0 が設定されています。
postscale は 1～16 を設定します。0 を設定するとオーバーフローで割込がかかり、1 を設定すると 2 回のオーバーフローでかかるようになり、以下、同様に設定されます。
戻り値：未定義
機能: タイマー2 の初期化を行ないます。下記に示される設定内容を設定します。
タイマー値は GET_TIMER2 や SET_TIMER2 を使って読み書きすることが出来ます。タイマー2 は 8bit カウンター/タイマーです。
対象デバイス: Timer2 ハードウエアを持ったデバイス
必要：デバイスの .h ファイルで定数が定義されている必要があります。
サンプル：setup_timer_2(T2_DIV_BY_4, 0xc0, 2);
　　// 20MHz クロックで 800nS でカウントされ、153.6us でカウンタ
　　// がオーバーフローし、割込は 460.3us でかかります。
サンプル・ファイル：ex_pwm.c
参照：get_timer2(), setup_timer2()

SETUP_TIMER_3( )
構文：setup_timer_3***(mode)***
パラメータ：***mode*** は｜で接続された or の異なったグループからの定数の１つ
T3_DISABLED　　：　タイマー3 を使用しない
T3_INTERNAL
T3_EXTERNAL
T3_EXTERNAL_SYNC
T3_DIV_BY_1　　：　プリスケーラ設定 1/1
T3_DIV_BY_2　　：　プリスケーラ設定 1/2
T3_DIV_BY_4　　：　〃　1/4
T3_DIV_BY_8　　：　〃　1/8

戻り値：未定義
機能: タイマー3 の初期化を行ないます。下記に示される設定内容を設定します。
タイマー値は GET_TIMER3()や SET_TIMER3()を使って読み書きすることが出来ます。
タイマー3 は 16bit カウンター/タイマーです。
PIC18 デバイスでのみ使用することが出来ます。
対象デバイス: PIC18
必要：デバイスの .h ファイルで定数が定義されている必要があります。
サンプル：setup_timer_3(T3_INTERNAL｜T3_DIV_BY_2);
参照：get_timer3(), setup_timer3()

SETUP_VREF( )
構文： setup_vref***(mode ／value)***
パラメータ：***mode*** は下記のいずれかの定数です。
FALSE : OFF
VREF_LOW : VDD*VALUE/24
VREF_HIGH : VDD*VALUE/32+VDD/4
VREF_A2 : A2 ピン
Value は整数 0～15 です。
戻り値：未定義
機能: アナログ・コンペアと、又は、ピン A2 上の出力のたに使用される内部リファレンスの電圧(VREF)を設定します。VALUE と組合せるか、VREF_A2 を指定します。
対象デバイス: VREF ハードウエアを持っているデバイス
必要：デバイスの .h ファイルで定数が定義されている必要があります。
サンプル：setup_vref(VREF_HIGH | 6); // VDD=5V のとき VREF=2.19V
サンプル・ファイル：ex_comp.c

SETUP_WDT ( )
構文： setup_wdt***(mode)***
パラメータ：PCB/PCM では WDT_18MS, WDT_36MS, WDT_72MS, WDT_144MS,
WDT_288MS, WDT_576MS, WDT_1152MS, WDT_2304MS
PIC18 では WDT_ON, WDT_OFF
戻り値：未定義
機能: ウォッチドッグ・タイマーを設定します。
ソフトウエアがスタックした場合にハードウエアをリセットするために使用されます。

	PCB/PCM	PCH
Enable/Disable	#fuses	set_wdt()
Timeout time	setup_wdt()	#fuses
Restart	restart_wdt()	restart_wdt()

対象デバイス: 全デバイス
必要：#fuses, デバイスの .h ファイルで定数が定義されている必要があります。
サンプル：#fuses WDT_18MS // PIC18 の例は、restart_wdt を見て下さい。

```
Main() }
     setup_wdt(WDT_ON);
     while (TRUE){
          restart_wdt();
          perform_activity();
    }
}
```

サンプル・ファイル： ex_wdt.c
参照: #fuses, restart_wdt()

SHIFT_LEFT()
構文： shift_left***(address, bytes, value)***

パラメータ：***address*** はメモリーへのポインター、***byte*** はシフトに必要なバイト数のカウント、***value*** には挿入するビット値 0 から 1
戻り値：ビット・シフト出力 0 又は、1
機能: 構造体や配列から左ビットシフトを行ないます。
address には、配列や構造体のアドレスを渡します。（&data と同じ）
bytes にはシフトに必要なバッファ・サイズをバイト値で与えます。bytes は操作に十分な容量が必要です。RAM の最下位バイトのビット 0 は LSB として扱われます。
対象デバイス: 全デバイス
必要：なし
サンプル：byte buffer[3];

```
for(I=i; i<=24; ++i) }
      while (!input(PIN_A2));          //クロック・ハイお待ちます。
      shift_left(buffer,3,input(PIN_A3));
      while (input(PIN_A2));           //クロック・ローを待ちます。
 }
// ピン A1 から 16bit を読み出し、ピン A2 の値が変化（L->H、H->L）したとき
   それぞれの値を読み込みます。
```

サンプル・ファイル： ex_extee.c with 9356.c
参照: shift_right(), rotate_right(), rotate_left(), <<,>>

SHIFT_RIGHT()
構文：shift_right***(address, bytes, value)***
パラメータ：***address*** はメモリーへのポインター、***byte*** はシフトに必要なバイト数のカウント、***value*** には挿入するビット値 0 から 1
戻り値：ビット・シフト出力 0 又は、1
機能: 構造体や配列から右ビットシフトを行ないます。
address には、配列や構造体のアドレスを渡します。（&data と同じ）
bytes にはシフトに必要なバッファ・サイズをバイト値で与えます。bytes は操作に十分な容量が必要です。value には、挿入するビット値を与えます。RAM の最下位バイトのビット 0 は LSB として扱われます。
対象デバイス: 全デバイス
必要：なし
サンプル：

```
// ピン A1 から 16bit を読み出し、ピン A2 の値が変化（L->H、H->L）したとき 各
   値を読み込みます。
 struct { byte time;
          byte command : 4;
          byte source  : 4; } msg;

 for ( i = 0; i < 16; i++ ){
      while ( !input( PIN_A1 ) );
      shift_right( &msg, 3, input( PIN_A2 ) );
      while ( input( PIN_A1 ) );
 }

// ピン A0 に data の下位ビットから出力
 for ( i = 0; i < 8; i++ )
      out_bit( PIN_A0, shift_right( &data, 1, 0 ) );
```

サンプル・ファイル： ex_extee.c with 9356.c
参照: shift_left(), rotate_right(), rotate_left(), <<,>>

SIN ( )
COS ( )
TAN ( )
ASIN ( )
ACOS ( )
ATAN ( )
SINH ( )
COSH()
TANH ( )
ATAN2 ( )

構文：val=sin***(rad)***
val=cos***(rad)***
val=tan***(rad)***
rad=asin***(val)***
rad=acos***(val)***
rad1=atan***(val)***
rad2=atan2***(val, val)***
result=sinh***(value)***
result=cosh***(value)***
result=tanh***(value)***

パラメータ：***rad*** はラジアン –2pi から 2pi の角度を現す float[実数]
val は範囲 –1.0 から 1.0 の float[実数], ***value*** は float

戻り値：rad はラジアン –pi/2 から pi/2 の角度を現す float[実数]
val は範囲 –1.0 から 1.0 の float[実数]
rad1 はラジアン –0 から pi の角度を現す float[実数]
rad2 はラジアン –pi から pi の角度を現す float[実数]
result は float

機能: 基本的な三角関数です。

sin	パラメータ(ラジアン)のサイン値を返します。
cos	パラメータ(ラジアン)のコサイン値を返します。
Tan	パラメータ(ラジアン)のタンジェント値を返します。
asin	範囲[-pi/2,+pi/2]ラジアンでアーク・サイン値を返します。
acos	範囲[0,pi]ラジアンでアーク・コサイン値を返します。
atan	範囲[-pi/2,+pi/2]ラジアンでアーク・タンジェント値を返します。
atan2	範囲[-pi,+pi]ラジアンで y/x のアーク・タンジェント値を返します。
sinh	x の双曲線サインを返します。
cosh	x の双曲線コサインを返します。
tanh	x の双曲線タンジェントを返します。

エラー・ハンドリング：
もし、”errno.h”が含まれますとドメイン[領域]とレンジ[範囲]・エラーが errno 変数にストアされます。ユーザーはエラーが起こったかを errno でチェックすることが出来ます。 Perror 関数を使ってプリントをチェックすることが出来ます。

ドメイン・エラーは下記の場合に起こります。：
・asin: 引数がレンジ[-1,+1]でないとき
・acos: 引数がレンジ[-1,+1]でないとき
・atan2: 両方の因数がゼロのとき

レンジ・エラーは下記の場合に起こります。：
・cosh: 引数が大きすぎるとき
・sinh: 引数が大きすぎるとき

対象デバイス: 全デバイス

必要：math.h が含まれる必要があります。
サンプル： float phases; // 1 つの正弦波を出力
for(phase=; phase<2+3.141596; phase+=0.01)
set_analog_voltage(sin(phase)+1);
サンプル・ファイル：ex_tank.c
参照：log(), log10(), exp(), pow(), sqrt()

SINH ( )
参照：SIN()

SLEEP ( )
構文：sleep()
パラメータ：なし
戻り値：未定義
機能: スリープ命令を実行します。詳細は各デバイスによりことなりますが、一般的にロー・パワー・モードに入り、そして、指定された外部イベントによるウォーケンまでプログラムの実行が中止されます。
対象デバイス: 全デバイス
必要：なし
サンプル：SLEEP();
サンプル・ファイル：ex_wakup.c
参照：reset_cpu()

SPI_DATA_IS_IN ( )
構文： result = spi_data_is_in()
パラメータ：なし
戻り値：0(FALSE)又は、1(TRUE)
機能: SPI からのデータがあれば TRUE を返します。
対象デバイス: SPI ハードウエアを持ったデバイス
必要: なし
サンプル：while(!spi_data_is_in() && input(PIN_B2)) ;
if(spi_data_is_in())
data = spi_read();
参照：spi_read(), spi_write()

SPI_READ ( )
構文：spi_read***(data)***
パラメータ：***data*** はオプションですが、もし、使用する場合は 8bit 整数
戻り値：8bit 整数
機能: SPI のデータを返します。値が SPI_READ に渡されますと、データがクロックを出力し、そして、受け取られたデータを戻します。もし、このデバイスがクロックを供給する場合は、SPI_READ()に続く SPI_WRITE(data)、又は、SPI_READ(data)のどちらでも実行します。これらは同じことをすることになり、クロックを出します。送るデータがないときは、クロックを得るために SPI_READ(0)として下さい。他のデバイスによりクロックを供給する場合は、クロックとデータ待ちのためには SPI_READ(),そして、データを決定するために SPI_DATA_IS_IN()を使います。
対象デバイス: SPI ハードウエアを持ったデバイス
必要: なし
サンプル： in_data = spi_read(out=data);
サンプル・ファイル： ex_spi.c
参照: spi_data_is_in(), spi_write()

SPI_WRITE ( )
構文：spi_write***(value)***
パラメータ：***value*** は 8bit 整数
戻り値：なし
機能: SPI インターフェースにバイトを出力します。これは 8 クロック生成されることになります。SPI に値を出力します。
対象デバイス: SPI ハードウエアを持ったデバイス
必要: #include<math.h>
サンプル：spi_write(data_out);
data_in = spi_read();
サンプル・ファイル： ex_spi.c
参照: spi_read(), spi_data_is_in()

SPRINTF ( )
構文：sprintf***(string, cstring, values..)***
パラメータ：***string*** は文字配列
cstring は constant string 又は, null でターミネートされた文字配列
Value はカンマでセパレートされた変数のリスト
戻り値：なし
機能: sprintf は指定した文字列に出力されたデータを置く点を除けば、printf と同じです。
出力文字列は null でターミネートされます。文字列の大きさがデータを格納するのに十分であるかどうかはチェックしません。出力フォーマットについては printf() を参照ください。
対象デバイス: 全デバイス
必要：なし
サンプル： char mystring[20];
long mylong;

mylong=1234:
sprintf(mystring,"<%1u>",mylong); // mystring は<1 2 3 4> ¥0
サンプル・ファイル: stdlib.h
参照 : printf()

SQRT ( )
構文：result = sqrt***(value)***
パラメータ：***value*** は float[実数]
戻り値：float[実数]
機能: もし、errno.h がインクルードされていればドメイン及びレンジエラーは errno 変数に格納されます。ユーザーは errno をチェックし、エラーが発生した場合に Perror 関数によりエラーを出力することができます。
ドメインエラーは以下の場合に発生する。:
● sqrt: 引数が負のとき
対象デバイス: 全デバイス
必要：#include <math.h>
サンプル：distance = sqrt(sqr(x1-x2) + sqr(y1-y2));
サンプル・ファイル: なし

SRAND( )
構文： srand***(n)***
パラメータ：***n*** は擬似乱数を生成するシード値(整数)です。
戻り値：なし。
機能: srand 関数は擬似乱数の乱数系列を再設定する擬似乱数生成を初期化する関数です。
対象デバイス ： 全デバイス

必要：#include<STDLIB.H>
サンプル：srand(10);
　　I=rand();
サンプル・ファイル: なし
参照：rand()

WRITE_EXTERNAL_MEMORY ( )
構文：write_external_memory(***address, dataptr, count***)
パラメータ：***address*** は PCM デバイスでは 16bit、PCH デバイスでは 32bit、***adataptr*** は ***count*** は 8 ビット整数
戻り値：未定義
機能: dataptr から address までの count バイトをプログラム・メモリーに書き込みます。WRITE_PROGRAM_EEPROM と WRITE_PROGRAM_EEPROM、この関数は特別な EEPROM/FLASH 書き込みアルゴリズムを使用しません。データは単にレジスター・アドレスからプログラム・メモリー・アドレスにコピーされます。
これは、外部 RAM 又は外部のフラッシュのためにアルゴリズムを与えるために役に立ちます。
対象デバイス: PCH デバイスのみ
必要: なし
サンプル: for(i=0x1000;i<=0x1fff;i++) {
　　value=read_adc() ;
　　write_external_memory(I, value, 2) ;
　　dlay_ms(1000) ;
　　}
サンプル・ファイル： loader.c
参照: write_program_memory(), read_external_memory()

WRITE_PROGRAML_MEMORY ( )
構文：write_program_memory(***address, dataptr, count***)
パラメータ：***address*** は PCM デバイスでは 16bit、PCH デバイスでは 32bit、***adataptr*** は ***count*** は 8 ビット整数
戻り値：未定義
機能: dataptr から address までの count バイトをプログラム・メモリーに書き込みます。この関数は count が FLASH_WRITE_SIZE の倍数であるときに効果的です。
この関数が FLASH_ERASE_SIZE の倍数アドレスに書き込まれようとするときはいつも全体のブロックが消去されます。
対象デバイス: プログラム・メモリーに書き込めるデバイスのみ
必要: なし
サンプル: for(i=0x1000;i<=0x1fff;i++) {
　　value=read_adc() ;
　　write_external_memory(I, value, 2) ;
　　dlay_ms(1000) ;
　　}
サンプル・ファイル： loader.c
参照: write_program_eeprom(), erase_program_eeprom()

WRITE_EXTERNAL_MEMORY ()
構文：write_external_memory(***address, dataptr, count***)
パラメータ：***address*** は PCM デバイスでは 16bit、PCH デバイスでは 32bit、PCM では最下位のビットは常に 0 となります。***adataptr*** は単一又は複数バイトのポインター。
count は 8 ビット整数
戻り値：未定義
機能: dataptr から address までの count バイトをプログラム・メモリーに書き込みます。

WRITE_PROGRAM_EEPROM と WRITE_PROGRAM_EEPROM、この関数は特別な EEPROM/FLASH 書き込みアルゴリズムを使用しません。データは単にレジスター・アドレスからプログラム・メモリー・アドレスにコピーされます。
これは、外部 RAM 又は外部のフラッシュのためにアルゴリズムを与えるために役に立ちます。
対象デバイス: PCH デバイスのみ
必要: なし
サンプル: for(i=0x1000;i<=0x1fff;i++) {
value=read_adc() ;
write_external_memory(I, value, 2) ;
dlay_ms(1000) ;
}
サンプル・ファイル： loader.c
参照: write_program_memory(), read_external_memory()

WRITE_PROGRAML_MEMORY ( )
構文：write_program_memory(***address, dataptr, count***)
パラメータ：***address*** は PCM デバイスでは 16bit、PCH デバイスでは 32bit、ＰＣＭでは最下位のビットは常に 0 となります。***adataptr*** は単一又は複数バイトのポインター。
count は 8 ビット整数
戻り値：未定義
機能: dataptr から address までの count バイトをプログラム・メモリーに書き込みます。
この関数はカウントが FLASH_WRITE_SIZE の複数であるとに効果的です。
この関数が FLASH_ERASE_SIZE の倍数アドレスに書き込まれようとするときはいつも全体のブロックが消去されます。

対象デバイス: プログラム・メモリーに書き込めるデバイスのみ
必要: なし
サンプル: for(i=0x1000;i<=0x1fff;i++) {
value=read_adc() ;
write_external_memory(I, value, 2) ;
dlay_ms(1000) ;
}
サンプル・ファイル： loader.c
参照: write_program_eeprom(), erase_program_eeprom()

標準文字列操作関数 – STANDARD STRING
MEMCHP ( )
MEMCMP()
STRCAT ( )
STRCHR ( )
STRCMP ( )
STRCOLL ( )
STRCSPN ( )
STRICMP ( )
STRLEN ( )
STRLWR ( )
STRNCAT ( )
STRNCMP ( )
STRNCPY ( )
STRPBRK ( )
STRRCHR ( )
STRSPN ( )
STRSTR( )
STRXFRM ( )

構文：
ptr=strcat***(s1, s2)*** 文字列 s1 に s2 を連結し、その結果を s1 に格納します。戻り値は s1 の先頭のポインターが返されます。
ptr=strchr***(s1, c)*** 文字列 s に c が含まれるかを s の先頭から検索します。文字 c が見つかったらそのアドレスを返します。
ptr=strrchr***(s1, c)*** 文字列 s に c が含まれるかを s の終わりから検索します。文字 c が見つかるとそのアドレスを返します。
cresult=strcmp***(s1, s2)*** 文字列 s1 と文字列 s2 を比較します。結果が－１のとき s1＜s2、0 のとき s1=s2、1 のとき s1>s2 です。
iresult=strncmp***(s1, s2, n)*** 文字列 s1 と文字列 s2 を最大 n バイトもしくは NULL に達するまで比較します。結果が -1 のとき s1＜s2、0 のとき s1=s2、1 のとき s1>s2 です。
iresult=stricmp***(s1, s2)*** 文字列 s1 と文字列 s2 を大文字、小文字の区別なく比較します。結果が－１のとき s1＜s2、0 のとき s1=s2、1 のとき s1>s2 です。
ptr=strncpy***(s1, s2, n)*** バッファ s1 に s2 を最大 n バイトもしくは NULL が検出されるまでコピーします。
iresult=strcspn***(s1, s2)*** 文字列 s1 をスキャンし s2 に含まれない文字のみからなる最初の部分を返します。
iresult=strspn***(s1, s2)*** 文字列s1をスキャンしs2に含まれる文字のみからなる最初の部分を返します。
iresult=strlen***(s1)*** 文字列 s の長さを返します。
ptr=strlwr***(s1)*** 文字列 s の中の大文字を小文字に変換します。戻り値は s のポインターです。
ptr=strpbrk***(s1, s2)*** 文字列 s1 をスキャンして s2 中の文字が最初に現れた位置を返します。
戻り値は最初に見つかった位置を返します。見つからない場合は NULL が返されます。
ptr=strstr***(s1, s2)*** 文字列 s1 をスキャンして文字列 s2 が現れる位置を返します。見つからない場合は NULL が返されます。
iresult=strcoll***(s1, s2)*** 文字列を現在のローカライズ処理に従っておこなうことを除けば strcmp() と同じです。
res=strxfrm***(s1, s2, n)*** strcoll()による文字列の比較の代わりに strcmp()を使って直接比較が出来るように文字列 s2 を変換し、その結果を s1 がポイントする配列に空き文字を含めて最大だ n 文字格納します。変換された文字列の長さを返します。その値が n 以上の場合、s1 がポイントする配列の内容は不定です。N が()で s1 が空ポインターのとき変換の文字列の長さを返します。
iresult=memcmp***(m1, m2, n)*** m1 と m2 の最初の n バイトを比較します。大きい、等しい、小さい

によって正、0、負の整数を返します。
ptr=memchr***(m1, c, n)*** バイト列 m1 の最初の n バイト中に文字 c があれば、その位置へのポインタを返します。無ければ空ポインターを返します。
パラメータ： ***s1*** と ***s2*** は文字配列へのポインター（又は、配列名）
s1 と s2 は定数であってはいけません。(“hi”の様な)。
n は操作のためのキャラクターの最大数のカウント
c は 8bit キャラクター
m1 と ***m2*** とポインター
戻り値：ptr は s1 ポインターのコピー
iresult は 8bit 整数
cresult は 1(以下), 0(イコール)又は、1(以上)
res は整数
機能: 文字列には、文字列の最後に NULL コードが入ります。この分を含めた十分な長さのバッファを必要に応じて用意してください。
対象デバイス: 全デバイス
必要：#include<string.h>
サンプル： char string1[10], string2[10];

```
strcpy(string1,“hi”);
strcpy(string2, “there”);
strcpy(string2, “there”);

printf(“Length is %u¥r¥n”, strlen(string1) );          // print 8
```

サンプル・ファイル：ex_str.c
参照：strcpy(), strtok()

STRCPY ( )
構文： strcpy***(dest, src)***
パラメータ：***dest*** は RAM 文字配列へのポインター，***src*** は RAM 文字配列へのポインター又は、文字列定数
戻り値：未定義
機能: 文字列定数 source を RAM バッファ dest にコピーします。
対象デバイス: 全デバイス
必要: なし
サンプル： char strings[10], string2[10];

```
.
.
.
strcpy(string, “Hi There”);
strcpy(string2, string);
```

サンプル・ファイル：ex_str.c
参照：strxxxx()

STRTOD()
構文： result=strtod***(nptr, & endptr)***
パラメータ：***nptr*** と ***endptr*** は文字列
戻り値：result は float。
変換すれば変換した値を返します。変換が行わなければ 0 を返します。
機能: strtod 関数は nptr によりポイントされた文字列の最初の部分を float 表現に変換します。変換対象列が空であるかまたは認識可能な形式でないとき、変換は行ないません。このとき、endptr が空ポインタでなければ、nptr の値を endptr が指すオブジェクトに格納します。
対象デバイス: 全デバイス
必要：STDLIB.H が含まれなくてはいけません。

サンプル： float result;
char str[2]=”123.45hello”;
char *ptr;
result=strtod(str, &ptr)
//result は 123.45、ptr は”hello”
参照：strtol(), strtoul()

STRTOK()
構文： ptr=strtok***(s1, s2)***
パラメータ：***s1*** と ***s2*** は文字配列へのポインター（又は、配列名）
s1 と s2 は定数であってはいけません。(“hi”の様な)。s1 は連続操作を示す 0 であっても構いません。
戻り値：ptr は s1 又は、0 のキャラクター
機能: 文字列 s1 から s2 で指定されるトークンにより次々に切り出します。
文字列 s1 には s2 で指定されたトークンが含まれているものとし、最初に STRTOK を呼び出すと s1 中のトークンの位置を返します。そして元の s1 のトークンの直後に NULL を書き込みます。その後、最初の引数に NULL を指定して STRTOK を連続して呼び出すと次々にトークンを切り出して行きます。トークンは呼び出すたびに変化してもかまいません。
対象デバイス: 全デバイス
必要：#include <string.h>
サンプル： char string[30], term[3], *ptr;

strcpy(string,“one,two,three;”);
strcpy(term, “,;”);

ptr = strtok(“a,b.c/d”, “,./”);
while (prt != NULL){
ptr = strtok(NULL, “,./”);
}
//プリント one,two,three
サンプル・ファイル：ex_str.c
参照：strxxxx(), strcpy()

STRTOL()
構文： result=strtol***(nptr, & endptr, base)***
パラメータ：***nptr*** と ***endptr*** は文字列、***base*** は整数
戻り値：result は signed long int
機能: strtol 関数は nptr によりポイントされた文字列の最初の部分を、long 型の表現に変換します。変換対象列が空であるかまたは認識可能な形式でないとき、変換は行いません。このとき、endptr が空ポインタでなければ、nptr の値を endptr が指すオブジェクトに格納します。
対象デバイス: 全デバイス
必要：STDLIB.H が含まれなくてはいけません。
サンプル： signed long result;
char str[2]=”123hello”;
char *ptr;
result=strtol(str, &ptr, 10)
//result は 123、ptr は”hello”
参照：strtod(), strtoul()

STRTOUL()
構文： result=strtoul***(nptr, & endptr, base)***
パラメータ：***nptr*** と ***endptr*** は文字列、***base*** は整数

戻り値：result は unsigned long int。
機能: strtol 関数は nptr によりポイントされた文字列の最初の部分を、long 型の表現に変換します。変換対象列が空であるかまたは認識可能な形式でないとき、変換は行いません。このとき、endptr が空ポインタでなければ、nptr の値を endptr が指すオブジェクトに格納します。
対象デバイス: 全デバイス
必要：STDLIB.H が含まれなければいけません。
サンプル： long result;
```
char str[2]=”123hello”;
char *ptr;
result=strtoul(str, &ptr, 10)
//result は 123、ptr は”hello”
```
参照：strtol(), strtod()

SWAP ( )
構文： swap***(lvalue)***
パラメータ：***ivalue*** はバイト変数
戻り値：未定義　＊注意！: この関数は結果をもどしません。
機能: １バイトの byte の上位 4bit（上位ニブル）と下位 4bit(下位ニブル)を入れ替えます。これは, 次のような演算式と同じです。byte = (byte <<4) | (byte >> 4);
対象デバイス: 全デバイス
必要: なし
サンプル： x = 0x45;
```
swap( x );                    //結果は 0x54 になる
```
参照：rotate_right(), rotate_left()

TAN ( )
参照: sin()

TANH ( )
参照: sin()

TOUPPER ( )
TOLOWER ( )
構文：　result = tolower ***(cvalue)***
　　result = toupper ***(cvalue)***
パラメータ：***cvalue*** はキャラクター
戻り値： 8bit キャラクター
機能: char を変換して変換した結果を返す。変換を受けない文字はそのまま返される。
TOUPPER(x) ： 'a'..'z' → 'A'..'Z'
TOLOWER(x) ： 'A'..'Z' → 'a'..'z'
対象デバイス: 全デバイス
必要: なし
サンプル：
```
switch( toupper(getc()) ) {
    case ‘R’ : read_cmd() ; break;
    case ‘W’ : write_cmd(); break;
    case ‘Q’ : done=TRUE;     break;
  }
```
サンプル・ファイル：ex_str.c
参照 :なし

WRITE_BANK ( )
構文：　write_bank***(bank, offset, value)***
パラメータ：***bank*** は物理的 RAM バンク 1-3(デバイスに依存)、***offset*** は 0 で始まるそのバンクの

ためのユーザーRAM 内にオフセット、***value*** は書き込みのための 8bit データ
戻り値：未定義
機能: 指定されたメモリー・バンクのユーザーRAM エリアにデータ・バイトを書き込みます。57 チップでは、bank は 1～3 で、offset は 0～15 です。PCM コンパイラーでは、bank は１で、オフセットは 0 から取ります。
対象デバイス: PCB ではメモリー 1Fh 以上、PCM ではメモリーFFh 以上
必要: なし
サンプル：
```
i=0;                          //RS232 バッファとしてバンク 1 を使用
do {
    c=getc();
    write_bank(1,i++,c);
} while(c!=0x13);
```
サンプル・ファイル：ex_psp.c
参照：FAQ(よくある質問)を見て下さい。

WRITE_EEPROM ( )
構文： write_eeprom***(address, value)***
パラメータ：***address*** は 8bit 整数、レンジはデバイスに依存。***Value*** は 8bit 整数
戻り値：未定義
機能: 指定されたデータ EEPROM にバイトを書き込みます。この機能は実行に数ミリ秒掛かります。デバイスのコアに内蔵 EEPROM を持ったデバイスにのみ利用出来ます。
対象デバイス: 外部 EEPROM を持ったデバイス、又は、同一パッケージ(12CE671)にセパレートされた EEPROM を持っているデバイス
必要: なし
サンプル：
```
#define     last_volume   10          //EEPROM 内の場所
volume++;
write_eeprom( last_volume, volume );
```
サンプル・ファイル： ex_intee.c
参照: read_eeprom(), write_program_eeprom(),read_program_eeprom(),
EX_EXTEE.c with CE51x.c, CE61x.c 又は、CE67x.c

WRITE_EXTERNAL_MEMORY ( )
構文：write_external_memory***(address, dataptr, count)***
パラメータ：***address*** は PCM デバイスでは 16bit、PCH デバイスでは 32bit、***adataptr*** は ***count*** は 8 ビット整数
戻り値：未定義
機能: ***dataptr*** から ***address*** までのカウント・バイトをプログラム・メモリーに書き込みます。WRITE_PROGRAM_EEPROM と WRITE_PROGRAM_EEPROM、この関数は特別な EEPROM/FLASH 書き込みアルゴリズムを使用しません。データは単にレジスター・アドレスからプログラム・メモリー・アドレスにコピーされます。これは外部 RAM 又は、外部フラッシュのためのアルゴリズムのインプリメントに有用です。
対象デバイス: PCH デバイスのみ
必要: なし
サンプル:
```
for(i=0x1000;i<=0x1fff;i++) {
    value=read_adc() ;
    write_external_memory(I, value, 2) ;
    dlay_ms(1000) ;
    }
```
サンプル・ファイル： loader.c
参照: write_program_memory(), read_external_memory()

WRITE_PROGRAM_EEPROM ( )
構文：write_program_eeprom***(address, data)***
パラメータ：***address*** は、PCM デバイスでは 16bit、PCH デバイスでは 32bit、***data*** は 16bit
戻り値：未定義
機能: 指定されたプログラム EEPROM にデータを書き込みます。
対象デバイス: プログラム・メモリーに書き込むことの出来るデバイスのみ
必要: なし
サンプル：write_program_eeprom(0,0x2800) ;
　　//disables program
サンプル・ファイル： ex_load.c, loader.c
参照: read_program_eeprom(), read_eeprom(), write_eeprom(), write_program_memory(), erase_program_eeprom()

WRITE_PROGRAML_MEMORY ( )
構文：write_program_memory***(address, dataptr, count)***
パラメータ：***address*** は PCM デバイスでは 16bit、PCH デバイスでは 32bit、***adataptr*** は ***count*** は 8 ビット整数
戻り値：未定義
機能: ***dataptr*** から ***address*** までのカウント・バイトをプログラム・メモリーに書き込みます。この関数はカウントが FLASH_WRITE_SIZE の複数であるとに効果的です。
この関数が FLASH_ERASE_SIZE の複数場所に書き込まれようとするときはいつも全ブロックイレースされます。
対象デバイス: プログラム・メモリーに書き込めるデバイスのみ
必要: なし
サンプル: for(i=0x1000;i<=0x1fff;i++) {
　　value=read_adc() ;
　　write_external_memory(I, value, 2) ;
　　dlay_ms(1000) ;
　　}
サンプル・ファイル： loader.c
参照: write_program_eeprom(), erase_program_eeprom()

標準 C 定義

標準 C 定義

標準C定義とマクロのための各種ヘッダー・ファイルが付属しています。下記はヘッダー・ファイルのリストとその定義です。そのファイル自身又は、さらに詳しい情報はC標準を参照して下さい。

errno.h

errno.h

EDOM:	ドメイン[領域]・エラー値
ERANGE:	レンジ[範囲]・エラー値
errno:	エラー値

float.h

float.h

FLT_RADIX:	指数表現の基数
FLT_MANT_DIG:	仮数部のビット数
FLT_DIG:	精度の 10 進桁数
FLT_MIN_EXP:	指数の底
FLT_MIN_10_EXP:	最小の 10 進の指数
FLT_MAX_EXP:	最大の 2 進の指数
FLT_MAX_10_EXP:	最大の 10 進の指数
FLT_MAX:	最大の浮動小数点数
FLT_EPSILON:	1.0+x != 1.0 なる x の値
FLT_MIN:	最小の浮動小数点数
DBL_MANT_DIG:	仮数部のビット数
DBL_DIG:	10 進精度の桁数
FLT_RADIX:	指数表現の基数
FLT_MANT_DIG:	仮数部のビット数
FLT_DIG:	精度の 10 進桁数
FLT_MIN_EXP:	指数の底
FLT_MIN_10_EXP:	最小の 10 進の指数
FLT_MAX_EXP:	最大の 2 進の指数
FLT_MAX_10_EXP:	最大の 10 進の指数
FLT_MAX:	最大の浮動小数点数
FLT_EPSILON:	1.0+x != 1.0 なる x の値
FLT_MIN:	最小の浮動小数点数
DBL_MANT_DIG:	仮数部のビット数
DBL_DIG:	10 進精度の桁数
DBL_MIN_EXP:	最小の 2 進の指数
DBL_MIN_10_EXP:	最小の 10 進の指数
DBL_MAX_EXP:	最大の 2 進の指数
DBL_MAX_10_EXP:	最大の 10 進の指数
DBL_MAX:	最大の浮動小数点数
DBL_EPSILON:	1.0+DBL_EPSILON !=1.0 になるような最小の数
DBL_MIN:	最小の浮動小数点数
LDBL_MANT_DIG:	FLT_RADIX で指定される基数で表現したときの浮動小数点数の有効桁数
LDBL_DIG:	浮動小数点数の桁数
LDBL_MIN_EXP:	浮動小数点数で表すことができる絶対値が最小の数を FLT_RADIX のべき乗表現したときの指数
LDBL_MIN_10_EXP:	浮動小数点数で表すことがで きる絶対値が最小の数を 10 のべき乗表現したときの指数

LDBL_MAX_EXP: 浮動小数点数で表すことができる絶 対値が最大の数を FLT_RADIX のべき乗表現したときの指数
LDBL_MAX_10_EXP: 浮動小数点数で表すことができる絶対 値が最大の数を 10 のべき乗表現したときの指数
LDBL_MAX: 浮動小数点数で表すことができる最大値
LDBL_EPSILON: 1.0+x と 1.0 が異なる値となる最も小さな 正の浮動小数点数
LDBL_MIN: 最小の正の正規化された浮動小数点数

limits.h
CHAR_BIT: ビット・フィールドではない最小オブジェクトのビット数
SCHAR_MIN: signed char の最小値
SCHAR_MAX: signed char の最大値
UCHAR_MAX: unsigned char の最大値
CHAR_MIN: char の最小値
CHAR_MAX: char の最大値
MB_LEN_MAX: 全てのロケールでの１文字の最大マルチバイト長
SHRT_MIN: short の最小値
SHRT_MAX: short の最大値
USHRT_MAX: unsigned short の最大値
INT_MAX: int の最大値
INT_MIN: int の最小値
UINT_MAX: unsigned int の最大値
LONG_MAX: long の最大値
LONG_MIN: long の最小値
ULONG_MAX: unsigned long 型の最大値

locale.h
iocale.h ローカライゼーションはサポートされておりません。
Lconv ローカライゼーション構造
SETLOCALE() null を返します。
LOCALCONV() clocale を返します。

setjmp.h
setjmp.h
jmp_buf: setjmp 及び longjmp の関数で使用される配列
setjmp: 次に発生する longjmp 関数のために現在の位置を保存
longjmp: setjmp で保存した位置にジャンプ

stddef.h
ptrdiff_t: ポインタ同士の減算の結果の型
size_t: sizeof 演算子の結果の型
wchar_t: サポートされている最大文字コード(char)(8 ビット)
NULL: 空(ヌル)ポインター(0)

stdio.h
stderr: 標準エラー出力(ストリームとして指定された USE RS232 又は、最初の USE RS232)
stdout: 標準出力(ストリームとして指定された USE RS232 又は、最後の USE RS232)
stdin: 標準入力(ストリームとして指定された USE RS232 又は、最後の USE RS232)

stdlib.h
Stdlib.h 関数は ANSI C に準拠しています。
div_t 商と余りを格納する 2 つの signed int からなる構造体

idiv_t:	商と余りを格納する 2 つの signed long からなる構造体
EXIT_FAILURE	1 を返します。
EXIT_SUCCESS	0 を返します。
RAND_MAX-	
MBCUR_MAX-	1
SYSTEM()	0 を返します。(サポートされていません。)
Multibyte	マルチバイト・キャラクターはサポートされていません。
MBLEN()	その文字列の長さを返します。
MBTOWC()	1 を返します。
WCTOMB()	1 を返します。
MBSTOWCS()	文字列の長さを返します。
WBSTOMBS()	文字列の長さを返します。

コンパイラー・エラーメッセージ

A #DEVICE required before this line
この行より前に#DEVICE が必要です。コードの生成を伴うようなステートメント又はコンパイラディレクティブがあらわれる前に、コンパイラは#device を必要とします。
一般的に#define は#device より前にありますが多くはありません。

A numeric expression must appear here
ここには式が必要です。ここには C 言語の式(123、A 又は B+C)が必要です。式は評価されて数値になります。

Array dimensions must be specified
配列の次元が指定されていません。[]を使って指定してください。

Arrays of bits are not permitted
配列には SHORT INT 型（bit 型）は使用できません。配列のレコードは次の要素にとられます。

Attempt to create a pointer to a constant
定数のテーブルは関数で実現され、ポインターは関数内で作成されません。
例えば、char const msg[9]={"Hi There" };では&msg での参照は行なえません。msg の参照は msg[i]で行ない、printf や strcpy でも同様です。

Attributes used may only be applied to a function (INLINE or SEPARATE)
関数の呼び出しには、#INLINE か#SEPARATE しか使用できません。

Bad expression syntex
式の文法が不正です。これは一般的なエラーメッセージです。全ての不正な文法に対して表示されます。

Baud rate out of range
指定されたボーレートでコンパイラーは設定することができませんでした。内蔵の UART はクロックから作成されます。UART は与えられたボーレートで３%以上の設定誤差を発生してしまっています。内蔵の UART を使用している場合は、クロックの値を変更してください。
また、高速なボーレートを設定するためには、高速なクロックが必要です。

BIT variable not permitted here
bit 型の変数は使用できません。例えば、X が SHORT INT 型で&X（X のポインタ）は使えません。

Can't change device type this far into the code
#DEVICE でデバイスの指定を行なう場合、コード内部での設定は行なえません。
コードエリアから#DEVICE を外してください。

Character constant constructed incorrectly
基本的に多くのキャラクターはシングルクォート’,’で囲んで指定します。'ab'や'¥nr'はシングル・キャラクターでないのでエラーとなります。'¥n'や'010'はシングル・キャラクターです。

Constant out of the valid range
アセンブラコードにおいて定数の値が範囲を超えるか正しくありません。
例えば、btfsc 3,9 でセカンドオペランドの９はエラーで正しくは、０～7 です。

Constant too large, must be < 65536
定数が大きすぎます。

Define expansion is too large
これらのエラーは、値の範囲がオーバーしています。0xffff を超える値や 255 文字以上の
文字列を指定しています。

Define syntax error
#DEFINE での構文エラーです。()や；などに注意してください。

Different levels of indirection
INLINE 宣言された関数のパラメータに変数を与えてパラメータを指定して呼び出しています。
定数で関数を呼び出してください。

Divide by zero
0（ゼロ）で除算しています。もしくは、式の評価が 0 となる除算を行なっています。

Duplicate case value
Duplicate DEFAULT statements
DEFAULT 文が複数存在します。１つの CASE 文には、DEFAULT 節は１つだけです。

Duplicate #define
Duplicate function
同じ関数が複数あります。関数のオーバーライドはサポートされていません。

Duplicate Interrupt Procedure
同じ割込関数があります。１つの割込要素に対して記述できる関数は１つです。
例えば、#INT_RB は各プログラム内で１度だけ現れます。

Duplicate USE
#USE ディレクティブで同じ指定が行われています。

Element is not a member
配列や構造体、共用体のメンバーが見当たりません。メンバー名を確認してください。

ELSE with no corresponding IF
End of file while within define definition
これらのエラーは、ファイルの終わりまで、対応する文が見つからないときに発生します。
もう一度｛｝や if else、()などの対応をチェックしてください。

End of source file reached without closing comment */ symbol
コメント記号/*に対応する*/がありません。

Error in define syntax
Error text not in file
エラーを定義した"ERROR.TXT"ファイルが見つかりません。バージョン、ディレクトリー、
パスなどを確認してください。

Expect ;
Expect comma

Expect WHILE
Expect }
Expecting :
Expecting =
Expecting a (
Expecting a, or)
Expecting a, or }
Expecting a .
Expecting a ; or ,
Expecting a ; or {
Expecting a basic type
Expecting a close paren
Expecting a declaration
Expecting a structure/union
Expecting a varible
Expecting a]
Expecting a }
Expecting an =
Expecting an array
Expecting an expression
Expecting an identifier
Expecting an opecode mnumonic
これらのエラーは全て対応する記号をチェックするか、配列、型などをチェックしてください。また、インライン・アセンブラーを記述している場合は、ニーモニックが正しいがどうかチェックしてください。

Expecting LVALUE such as a variable name or * expression
定数に代入しようするなど式が期待した記述となっていません。例えば、4=5 など

Expecting a basic type
関数呼び出しを期待していますが、異なった記述がされています。

Expecting Procedure name
Expression must be a constant or simple variable
Expression must evaluate to a constant
変数と定数の関係がおかしくなっています。

Expression too complex
コード生成で式や文が複雑過ぎてコード生成に失敗しています。単純な式や構文に書き直してください。

Extra characters in preprocessor command line
プリプロセッサー・コマンド行に解析できないコマンドや文字列が書かれています。
例えば、

```
#PRAGMA DEVICE <PIC16C74> main(){intx; x=1;}  など
```

プリプロセッサー行には 1 つのプリプロセッサー・コマンド書きます。

File in #INCLUDE can not be opened
インクルードしようとしたファイルがオープンできません。ファイル名、パスなどチェックしてください。

Filename must start with " or <
ファイル名の指定は、"か<で始まります。

Filename must terminate with " or >
ファイル名の指定は、"か>で終わります。

Floating point numbers not supported
浮動小数点演算、浮動小数点型が使用できないところに使っています。
例えば、インクリメント演算子に FLOAT 型を使用する（++F の F が float 型）など

Function definition different from previous definition
関数の定義が、プロトタイプ宣言の定義と異なった形で行なわれています。

Function used but not defined
示された関数はプロトタイプ宣言されています。プログラムでは定義できません。

Indetifier is already used in this scope
すでに同じ定義、名称が使用されています。

Illegal C character in input file
入力ファイルにおかしな文字が挿入されています。もう一度、ファイルをチェックしてください。
（コントロール・コードや、全角文字など）

Improper use of a function identifier
関数名と同じ名称が使用されています。関数なら()が必要です。

Incorrectly constructed label
不適当なラベル名が使用されています。x=5+や MPLAB:はラベル名には使用できません。

Initialization if unions is not permitted
共用体を初期化して宣言することはできません。構造体にしてください。

Internal compiler limit reached
コンパイラーの内部制限に引っかかっています。使用環境やソースファイルとともにご連絡下さい。対応がとれる場合もあります。

Invalid conversion from LONG INT to INT
LONG INT 型は、INT 型へ変換できません。強制的に行なわせるには型キャストしてください。

```
i=(int)(li);        //  i は int 型、li は long int 型
```

Internal Error - Contact CCS
コンパイラー内部のエラーです。ここにあるエラー以外のエラーが発生したり、ソースコードの解析に失敗しています。
ソースコード、コンパイラー環境とともにご連絡下さい。対応がとれる場合もあります。

Invalid parameters to shift function
シフト関数のパラメータが不正です。関数のパラメータをチェックしてください。

Invalid ORG range
ORG レンジが不正です。

Invalid Pre-Processor directive
プリロプロセッサー・ディレクティブが不正です。または、未知のディレクティブです。

Library in USE not found
#USE ディレクティブに指定された内容はライブラリに存在しません。スペルなどチェックしてください。

LVALUE required
構文から判断して変数が必要です。特に式の左辺値をチェックしてください。

Macro identifier requires parameters
#DEFINE で定義されたマクロのパラメータが必要です。
例：#define min(x,y)　((x<y)?x:y)のように定義された min マクロを呼び出すには
　　r=min(VALUE,6)とします。

Missing #ENDIF
呼応する #ENDIF なしで #IF が見つかった。
#endif が間違っているか、不要なところにあります。#if、#ifdef、#ifndef などの関係をチェックしてください。

Missing or invalid .REG file
ユーザー・レジストレーション・ファイルはアップデートの場合のダウンロード・ソフトウエアには含まれません。ソフトウエアを実行するにはこのレジストレーション・ファイルが .exe ファイルと同じディレクトリーになければいけません。これらはオリジナルのディスク又は、CD-ROM にあります。

Must have a #USE DELAY before a #USE RS232
RS232 ライブラリを使用する前には、#use delay でクロック周波数の設定が必要です。

No MAIN() function found
ソースファイルにメイン関数がありません。プログラムには必ず 1 つのメイン関数が必要です。

Not enough RAM for all variables
全ての変数を扱えるＲＡＭ領域が不足しています。変数を減らしてください。
Alt-M でメモリマップを表示して確認するか、大きな関数を小さな関数に分解して記述して RAM を節約してください。ローカル変数をたようしいてる場合は、いくつかの関数に分解して、ローカル変数を減らすと効果的です。

Number of bits is out of range
構造体でのメンバーに bit 型を指定する場合、1～8 を使用してビットを表しますが、
#BIT ディレクティブでは 0～7 を使用します。

Out of ROM, A segment or the program is too large
プログラムが大きすぎて、チップの ROM 領域に収まりません。プログラムを小さくするか、より大きな ROM を持つデバイスに置き換えてください。また、INLINE で宣言されている関数を、SEPARATE で宣言し直すなどしてコードのサイズを小さくしてみてください。
Alt-T で関数のコールツリーを表示してみて、それぞれの関数に重複した箇所や呼び出しのオーバーヘッドを避けて関数を再構成してみる方法もあります。
また、同じような動作をする部分を関数として独立させることもコードを節約できます。

Parameters not permitted
パラメータは許可されません。この関数又はプリプロセッサマクロは()内にパラメータを必要としません。

Pointers to bits are not permitted
Bit 型にはポインタは使用できません。

Pointers to functions are not valid
関数のアドレスは使用できません。

Previous identifier must be a pointer
構造体のメンバーのポインターによる参照は、構造体自身のポインタを宣言した後でしか使用できません。

Printf format type is invalid
printf 関数のフォーマットが不正です。%以降のフォーマット文字に知らない文字が使用されています。

Printf format (%) invalid
printf 関数のフォーマットが不正です。%と組み合わせる文字が不正です。

Printf variable count (%) does not match actual count
printf 関数の%で示された数と渡されるパラメータ数が一致しません。

Recursion not permitted
リンカーは、関数の再帰呼び出しをサポートしていません。再帰呼び出しと再帰的に呼び出されるようなルーチンを書き直してください。

Recursively defined structures not permitted
構造体の中に自身の構造体を入れることはできません。

Reference array are not permitted
配列の参照が不正です。

Return not allowed in void function
void 関数では Return 文は不必要です。関数が void の場合にはリターンするとき戻り値は不要です。

Signed type are not supported
符号付きの型はサポートしていません。全て符号なし(Unsigned）です。

String too long
文字列が長すぎます。255 文字以内としてください。

Structure field name required
Structures and UNIONS can not be parameters (use * or &)
構造体や共用体へのアクセスが不正です。

Subscript out of range
RAM 配列は 128 バイトを超えています。バンクにまたがる RAM の配列は指定できません。
ROM 配列が 256 エレメント以上にならない様にして下さい。

This linker function is not available in this compiler version
このコンパイラーのバージョンではこのリンカー機能は利用出来ません。

This type can not be qualified with this qualifier
許されない型宣言を行なっています。

This version does not support LONG INTs
LONG INT 型がサポートされていないバージョンです。
LONG INT 型がサポートされているバージョンにバージョンアップしてください。

Too many #DEFINE statements
#define ディレクティブが多すぎます。

Too many array subscripts
配列の次元が高すぎます。配列は５次元までです。

Too many constant structures to fit into available space
定数が多すぎてスペース内に展開できません。コールツリー・リストを見てスペースを評価して下さい。

Too many elements in an ENUM
ENUM で使用できるのは最大 256 エレメントです。

Too many identifiers have been defined
コンパイラーの内部で必要な変数が確保できませんでした。
ソースコード、コンパイラー環境とともにご連絡下さい。対応がとれる場合もあります。

Too many identifiers in program
コンパイラーの制限でプログラム領域が確保できませんでした。
ソースコード、コンパイラー環境とともにご連絡下さい。対応がとれる場合もあります。

Too many nested #INCLUDEs
インクルード・ファイルのネストが複雑過ぎます。ネストを浅くしてください。

Too many parameters
関数に宣言されているパラメータが多すぎます。パラメータを減らすか、関数を分けてください。

Too many subscripts
宣言された配列の要素を超えています。

Type is not defined
型が定義されていません。ほどんどがスペルのミスか他のコンパイラーの移植のミスです　。

Type specification not valid for a function

不正な関数宣言です。宣言内容をチェックしてください。

Undefined identifier
未定義な式、構文、関数などです。スペルをチェックしてみてください。

Undefined label that was used in a GOTO
ラベルが未定義か GOTO 文が不正です。

Unknown device type
未知のデバイス・タイプです。

Unknown keyword in #FUSES
#FUSE ディレクティブの要素に未知の要素が含まれています。キーワードを確認してください。

Unknown type
未知の型です。スペルをチェックしてください。

USE parameter invalid
不正なパラメータが使用されています。パラメータをチェックしてください。

USE parameter value is out of range
パラメータの値が範囲をを超えています。

コンパイラ警告メッセージ
コンパイラの警告メッセージには次の様なものがあります。
Assignment inside relational expression(内部関係式に関わる警告)
if(a==b)を意図したときに f(a=b)のような一般的なエラーです。

この警告は行内に入力ミスと同様の表示がされます。

Assignment to enum is not of the correct type(ENUM 変数の不正使用)
もし、変数が ENUM として宣言された場合、enum の変数としてのみ割り当てられるのがベストです。例:

```
enum colors {RED,GREEN,BLUE] color;
...
color = GREEN;     // OK
color = 1;         //  警告  209
color = (colors)1;     //OK
```

Code has no effect(無効なコード)
コンパイラーはこのソース・コードが生成されたコードが持ついかなる影響も識別することができません。例:

```
1;
a==b;
1,2,3;
```

Condition always FALSE(常に FALSE となる条件)
関係式で決して FALSE とにならないとコンパイラが認識したエラー。例

```
int x;
if( x>>9 )
```

Condition always TRUE(常に TRUE となる条件)
関係式で決して TRUE とにならないとコンパイラが認識したエラー。例

```
#define PIN_A1 41
...
if( PIN_A1 )//  意図したのは:
// if( input(PIN_A1) )
```

Function not void and does not return a value(非 void 関数関数であるにかかわらず戻り値がない場合)
戻り値として宣言された関数は戻るべき値を持ったリターン・ステートメントを持ってなければいけません。C の関数だけが戻り値の意図しない場合は void を宣言することを理解すべきです。もし、何の指定もされなければ戻り値は int 型とみなされます。

Operator precedence rules may not be as intended, use( ) to clarify(演算子の優先順位明示的に使用するために ( )を使用)

式において複数の演算子による結合は、プログラマを惑わせることがあります。この警告メッセージは、式の意味を明示するようにどこに（）を挿入するかを助ける為に出力されます。例

```
if( x << n + 1 )
```

これは次のように表した場合より一般的に理解されるでしょう。

```
if( x << (n + 1) )
```

Structure passed by value(値参照の構造体)

構造体は、いつも参照によって関数に引き渡される。この警告メッセージは構造体が値によってされた場合に現れる。構造体のサイズが 5 バイトに満たない場合は現れません。例

```
void myfunct( mystruct s1 )     // 値による引渡し - 警告
myfunct( s2 );
void myfunct( mystruct * s1 ) // 参照による引渡し- OK
myfunct( &s2 );
void myfunct( mystruct & s1 ) // 参照による引渡し- OK
myfunct( s2 )
```

Unreachable code(実行されないコード)
プログラムに含まれたコードはけっして実行されません。例:

```
if(n==5)
goto do5;
goto exit;
if(n==20) //この行にはいく方法はありません。
return;
```

Unsigned variable is never less than zero(Unsigned 変数は負の値をとりません。)
Unsigned 変数は決して負の値をとりません。この警告メッセージは、もし Unsigned 変数が負の値を取り得る場合にチェックを促しています。例

```
int i;
for(i=10); i>=0; i--)
```

Variable never used(使用されない変数)
変数が宣言されていて、コード内で参照されていません。

Variable of this data type is never greater than this constant(変数の型によって、変数の取り得る値の範囲を超える定数が記述されています。)
変数が定数と照合されます。変数の最大値は定数より大きくならない場合
次の例ではけっして TRUE にならない。

```
int x;       // 8 bits, 0-255
if ( x>300)
```

Q＆A、ちょっとしたテクニック

Q: RS232C ポートが思ったように動いてくれません。

A: 問題はいくつか考えられます。順に解説します。

1. PIC がゴミの様なデータを送信している。
   A. クロックをチェックしてください。クリスタル発振を用いている場合であればまず間違いはありませんが、RC オシレーターを使っている場合は発振周波数をチェックしてください。もし周波数がずれていたら、#use delay ディレクティブで周波数をを調整してください。
   B. ターミナルのボーレートやパリティをチェックしてください。
   C. RS232C のレベル・コンバーターと出力レベルをチェックしてください。
   MAX232 などを使用している場合は、INVERT は不必要です。直接抵抗やダイオードを通じて出力している場合は、INVERT が必要です。INVERT オプションは、#use rs232 で指定します。
   D. putc(6)は ASCII で 6(ACK)を出力します。表示できる文字とするためには、文字コードを与えます。A なら puts('A')となります。

2．PIC がゴミの様なデータを受信している。
   A. 1の内容（クロック、ターミナル設定、レベル）をチェックしてください。
   送信中など他のタスクを行なっているときに受信データが入ってきて、データが失われてはいませんか。 getc()関数をもっとも高い位置(優先順位の高いところ)においてデータを落とさないようにプログラムを再構成します。
   もし、ポーリングでデータを得てる場合、kbhit()を使うと 1/3 の時間で受信データのチェックが行なえます。また、9600bps のとき、kbhit()関数をおおよそ 35μS で呼び出して受信をチェックしないとデータを落とします。

3．何も送られていない場合
   A. トライステート・レジスターの設定はどうなっていますか。また、ポートのモードも #use rs232 り先に定義されていると安定します。なお、ポートのモードは standard が安定しています。
   B. 次のプログラムをコンパイルして実行してみてください。

```
main(){
        while(TRUE)
              putc('U');
}
```

このプログラムを実行させると、XMIT ピンから U が出力され続けます。U は 0x55 ですから、設定したボーレート分の１の方形波が出力される様子をオシロスコープなどで確認してください。また、レベルコンバータがある場合は、コンバーターの出力側もチェックしてみてください。

4．何も受信していない様子です。
最初に、PIC から確実に送信できることを 1～3 の内容を見て確認してください。
そして、次のプログラムを実行してみてください。

```
main(){
      printf( "start " );
      while(TRUE)
            putc( getc() + 1 );
}
```

これでターミナルから A を入力すると B が返されるはずです。
これでも受信できないようなら、ハードウェアのチェックをおこないます。RCV ピンに信号が入っているかなどターミナルから入力を行なって確認します。（一般的には、なにも信号が入力されない状態で RCV ピンは'H'で、ターミナルのキーを押すと'L'のパルスが入ってきます）

5．PIC の受信データや送信が全くできないわけではなく、でも動作はおかしいようです。

A. INVERT オプションがハードウェアと合致していますか。信号がなにも入力されない状態で RCV ピンが'H'なら INVERT は不要です。'L'なら INVERT が必要です。また、送信と受信でレベルが異なっているハードウェアならどちらかにレベルを合わせて、後はハードウェアを改良します。
B. 何もデータがないとき、通信に使用しているピンが'H'か'L'のレベルになっていますか。
C. ポート A を使っている場合、SETUP_PORT_A()でポート設定が行なわれていますか。
ポート A にアナログポートが割り当てられているデバイスでは、デジタル I/O はデフォルト設定ではありません。デバイスによってアナログポートやコンパレータポートとなっている場合があります。

6．コンパイラーが INVALID BAUD RATE エラーを出力する。
A. これは内蔵 UART を使用しない場合は、クロックとボーレートの関係で誤差が大きかったり、設定できない組合せを行なっているためです。大抵、クロックに対してボーレートが速すぎます。
B. 内蔵の UART を使っている場合も 3%以内の誤差でボーレートが設定できない状態がほとんどありませんが、内部の BRGH のバグで BRGH に１を設定したとき、同じ状態になります。これを回避するには、BRGK10K を#USE RS232 ディレクティブに指定してください。

Q: １つの PIC デバイスに 2 チャンネル以上の RS232 ポートは設定できますか？

A: #use rs232 ディレクティブ(I2C もほぼ同じです）は、GETC、PUTC、PRINTF、KBHIT の各関数を#use rs232 が見つかった時点で用意します。
#use rs232 が定義される行は、実行される（プログラムコードになる）行ではなくどちらかというと#define に似た動きをします。

次の参考プログラムを見てください。このプログラムは、RS-232 ポート A で受信した内容を、RS232 ポート A とポート B にエコーします。

```
#use rs232( BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1 )
void    put_to_a( char c ){
      putc(c);
}
char    get_from_a( ){
                  return( getc() ); }
#use rs232( BAUD=9600, XMIT=PIN_B2, RCV=PIN_B3 )
void    put_to_b( char c ){
      putc(c);
}
main() {
      char c;
      put_to_a( "Online¥n¥r" );
      put_to_b( "Online¥n¥r" );
      while(1){
            c=get_from_a();
            put_to_a( c );
            put_to_b( c );
      }
}
```

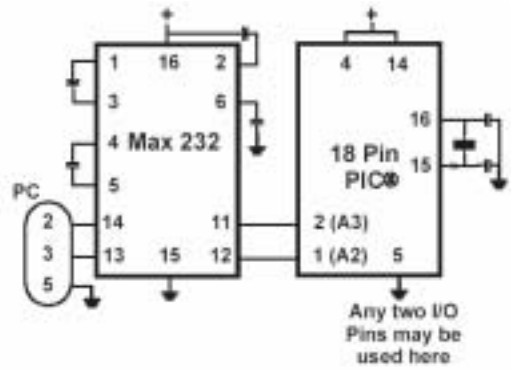
このプログラムと同じ動作するプログラムは次のようにもかけますが、非常に見にくいプログラムとなります。

```
main() {
char c;
#USE RS232( BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1 )
      printf( "Online¥n¥r" );
#USE RS232( BAUD=9600, XMIT=PIN_B2, RCV=PIN_B3 )
```

```
        printf( "Online¥n¥r" );
        while(TRUE){
#USE RS232( BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1 )
        c= getc();
#USE RS232( BAUD=9600, XMIT=PIN_B2, RCV=PIN_B3 )
        putc( c );
#USE RS232( BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1 )
        putc( c );
}
```

Q: PIC と PC（ターミナル）との接続方法は？

A: PIC のデジタル I/O のレベルは 0〜5V です。また、RS232C に用いられるレベルは±3〜12V で全くレベルが合いません。勿論、直接接続できません。そこで一般的には、MAX232 などのレベルコンバータを使用します。続例を以下に示します。参考にしてください。



Q: ROM が残っているようですがどうして ROM 不足のエラーが発生するのですか？

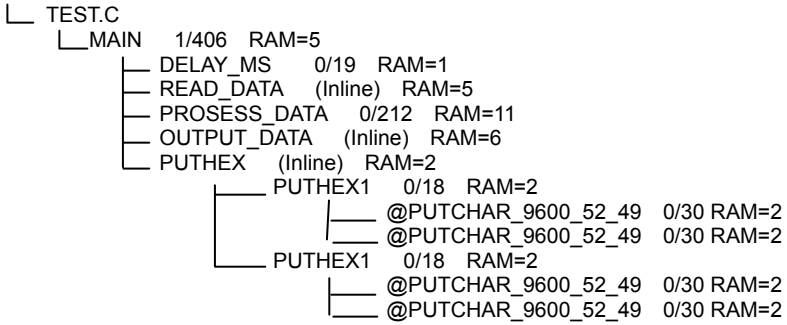
A: ROM エリアの不足は関数が１つのセグメントに割り当てられない場合に起こります。関数とインライン関数の全ては同じ１つのハードウェアページに割り当てられます。これらはリンカーが自動的に割り当てます。#separate で宣言された関数のみ分離されます。

次の例で見てみましょう。

```
TEST.C
  MAIN   ?/614   RAM=5
    DELAY_MS      0/19   RAM=1
    READ_DATA     (Inline)   RAM=5
    PROSESS_DATA     (Inline)   RAM=11
    OUTPUT_DATA     (Inline)   RAM=6
    PUTHEX     (Inline)   RAM=2
      PUTHEX1 0/18   RAM=2
        @PUTCHAR_9600_52_49   0/30 RAM=2
        @PUTCHAR_9600_52_49   0/30 RAM=2
      PUTHEX1 0/18   RAM=2
        @PUTCHAR_9600_52_49   0/30 RAM=2
        @PUTCHAR_9600_52_49   0/30 RAM=2
```

この例ではメインプログラムがほとんどインライン関数を呼び出しています。main()のサイズは 614 ワードあり`56 デバイスの 512 ワードを超えていますから割り当てることが出来ません。リンカーが出す？はセグメントに割り当てようとして割り当てられない場合に表示します。x/y の表示は x がページセグメント、y がコードサイズを表します。ここでリンカーの規則としてインライン関数はスタックを節約して呼び出せますが、その代わり関数を記述するコードサイズは大きくなります。

この例で、比較的大きな関数 PROCESS_DATA と MAIN の 2 つの関数を#separate で宣言して結果を見ると次の様になります。



Q: RAM 不足となりますが？ (OUT OF RAM)

A: コンパイラーは、多くの RAM を使用しないように努力します。ぜひとも理解してほしいことは、プログラムのデザイン（書き方の方法でなくアルゴリズムとでも言った方がいいのでしょうか）により RAM の使用量は大きく変化します。最も良い方法は、全て変数をローカルに使用することです。ローカル変数とすることにより RAM は一時的に使用されますが、関数を抜けた時点で解放されるからです。ここにあるサンプルプログラムでは、勿論、RAM 不足のエラーは発生しません。

RAM は、複雑な構文や式を評価、実行するとき比較的多く使用されます。また、コンパイラーは複雑な式や構文を展開するとき、出力コードの品質が低下するかもしれません。
RAM の割り当ては、関数で宣言された要素だけでなく、式の展開や評価の際も行なわれます。関数で必要とする RAM の合計は、パラメーター、ローカル変数、関数内の大きな式の展開、評価を行なうために必要な RAM の和となります。RAM は、コールツリー表示の RAM=のところに必要な量が計算されて表示されます。
また、新しい関数をコールするために新しい RAM が必要となる場合もあります。
関数が return で戻るとき、呼び出した関数の RAM は開放されますが、シーケンシャルにいつもコールとリターンを行なっているわけではないので、大きな関数を使用する場合は、効率よく RAM を利用することが難しくなります。

SHORT INT（1bit）型の変数をフラグや論理型(Boolean)に使用することは RAM の節約につながります。コンパイラーは、これらのビットをまとめて１バイトの RAM に割り当てようとします。HORT INT と宣言して使う変数は、自動的にコンパイラーが割り当てます。これは RAM サイズと ROM サイズの圧縮につながり、大変効果的です。

最後に外部にメモリ・デバイスを接続したときのことを述べます。
外部に 8 ピンの EEPROM や RAM を接続するためには、最低 2 本の信号線が必要です。そして、データ容量が増えることになります。コンパイラーのパッケージにはこれらのデバイスのアクセスに必要なサンプルプログラムが入っています。
これらのメモリの第一の欠点はリード/ライトのアクセスタイムが遅いことです。（PIC の内蔵の RAM に対して）S-RAM を使うとリード／ライトアクセスは比較的高速にできますが、電源がなくなると内容も消えます。（バッテリー・バックアップで補えますが、ハードウェアに負担が必要）また、EEPROM は電源を切っても内容は保持されますが書き込み時間が大変多く必要とします。（10mS 程度）

Q: なぜ．LST ファイルはこんなに乱雑なのですか？

A: “．LST”ファイル（リスト・ファイル）は、C ソースコードからアセンブラー・コードを生成

しその結果をファイルにしています。それぞれの C ソース行ごとにコンパイラーが生成したアセンブラー・コードが書き込まれます。但し、以下の３つの場合には注意が必要です。
これらは特別な場合で、リスト・ファイルは、本来、大変使いやすく有効なファイルです。

1. プログラムの先頭近くに実行可能なソースが無いのにアセンブラ・コードが書き込まれることがあります。

コンパイラーにより作成されたコードのあるものは特定のソース行には対応しません。
これは、コンパイラーにより作成されたコードのある種のものが特定のソース行に対応しないためです。コンパイラーはこのようなコードをプログラムの先頭近くに置くか、#USE 行に対応するサブルーチンを置きます。

2．アドレスが範囲をオーバーした場合

コンパイラーは C ソース・コードの順番で .LST ファイルを作成します。
リンカーは、関数がコードページ内又はコードページの半分に入るようにコードを再配置します。結果としてコードはソース順とはなりません。コンパイラは .LST ファイル中に不連続があるときはファイルの中に＊行を置きます。これは関数及びインライン関数がコールされる場所でしばしば見られます。インライン関数の場合、アドレスはインライン関数のソースが置かれた順序となります。

3．コンパイラが再三同じインストラクションを生成する場合。

```
.....A=0;
03F:    CLRF   15
*
046:    CLRF   15
*
051:    CLRF   15
*
113:    CLRF   15
```

このようなコードは、関数が一つのインライン関数であるときに一つ以上の場所からコールされるときにみられます。上記のケースでは、A=0 の行は一つのインライン関数で、４つの場所からコールされています。関数がコールされるたびに、コードのコピーが作成されます。コードの各インスタンスは元のソース行に沿って示され、結果としてアドレスまで異常に見えるかもしれないので、* が付けられます。

Q: TIMER0 割込の使い方とその周期の設定方法は？

A: 次の例は、固定した周期でパルスを出力するプログラムです。

```
#include <16Cxx.H>
#use delay( clock=15000000)
#define   high_start 114
byte   seconds, high_count;
#INT_RTCC
clock_isr() {
    if(--high_count == 0 ){
          output_high(pin_b0);
          delay_us(5);
          output_low(pin_b0);
          high_count=high_start;
    }
}
main(){
      high_count = high_start;
      set_rtcc(0);
      setup_counters( RTCC_INTERNAL, RTCC_DIV_128 );
      enable_interrupts( RTCC_ZERO );
```

```
        enable_interrupts( BLOBAL );
        while(TRUE);
    }
```

このプログラムでは、約１秒でパルスを出力します。この計算方法を以下に示します。

タイマーのカウンター・インクリメントは(CLOCK/4)/RTCC_DIV。
この例では、15MHz/4/128=29297Hz(34Ms)

タイマーは 256 カウントするとオーバーフローし割込かかかります。
この例では、29297/256=114(8.77mS)
プログラムで high_start を 114 にしていますので、１秒でパルスが出力されます。

Q: バイトとワードの変換はコンパイラーではどのように行なわれますか？

A: 例えは次の式を見てみます。

```
bytevar = wordvar;
```

この式では、wordvar の上位バイトが失われます。つまり、bytevar = wordvar & 0xff; と同じことです。上位バイトを取り出すには次の式を用います。

```
bytevar = wordvar >> 8;
```

では、ワードとバイトの比較を行なってみるとどうでしょうか。次の例を見てください。

```
wordvar = 0x1234;
bytevar = 0x34;

if( wordvar == bytevar )      この結果は
```

結果は、FALSE です。比較をする場合、bytevar は 0x34 ではなく、0x0034 となります。
このため 0x1234 != 0x0034 で FALSE。

次に、byte 値の演算で次の例を見てください。

```
bytevar1 = 0x80;
bytevar2 = 0x04;

wordvar = bytevar1 * bytevar2;
```

結果は、0 です。0x200 とはなりません。
そこで正しく演算結果を得るために次のように書き直します。

```
wordvar = (long)bytevar1 * (long)bytevar2;
```

これで結果をワードで得ることができます。このように、他の MS-DOS 上の C コンパイラーなどのように符号やデータ長の拡張は自動的には行なわれません。

Q: TRUE と FALSE、コンパイラーはどのように決定するのでしょうか？

A: 次の式を見てください。バイトやワードの評価で 0 か 1 が返されます。

```
bytevar = 5>0;      // bytevar は 1 です。
bytevar = 0>5;      // bytevar は 0 です。
```

同様に次の式の評価はどうなるでしょう。

```
bytevar = (x > Y) * 4;
```

これは次のプログラムと同じ結果を得ます。

```
if( x > y )
    bytevar = 4;
else
    bytebar = 0;
```

SHORT INT 型も 0 か 1 をとりますので、式の中で同じように使用できます。
次の例では、bit 型の変数が変化します。

```
bytevar = 54;
bitvar = bytevar;          // bytevar は 54 なので bitvar は 1

if( bytevar )               // 勿論、条件は TRUE
    bytevar = 0;
bitvar = bytevar;          // bytevar が 0 になったので bitvar も 0
```

Q: 割り込みから呼び出す関数には何か制限があるのでしょうか？

A: 割り込みプログラムがきちんと動作するには、十分なスタックが確保されていることです。コンパイラーは main()関数でスタックのチェックを行いそれ以外のチェックは行われません。割り込み処理があるとスタックが追加されます。1 つの割り込みルーチンで割り込み関数からは SEAPRATE コールで呼び出されます。これらの様子は、コールツリーを表示してみると良く分かります。main 関数でのスタックの最低残量が 9 以上ある場合、割り込みが有効にできます。

コンパイラーは再帰呼び出しをサポートしていません。これはプロセッサーが RISC 命令を使っているためで旧態的な C 言語のスタックではうまくインプリメントできないからです。関数で必要とされるすべての RAM のロケーションは、関数がアクティブでなくてもリンク時には決定されていることが必要となります。例えば、main 関数が関数 A()をコールし、A()が関数 B()をコールすると B()は main 関数から呼ばれていないことになります。

割り込みが特別な問題を引き起こす状況を良く考えてみると、割り込み関数 ISR()が関数 A()をコールするということは、main 関数が A()をコールすることと変わりはありません。main 関数が A()を呼び出し、A()が実行中に割り込み関数 ISR()がイベントによりコールされると再帰的な呼び出し状態が発生します。
ここで、関数 A()の呼び出しにプロテクトを行うようにコンパイラーが処理をした場合は、main 関数は A()が呼び出される前にすべての割り込みをディセーブルとして、A()の処理が終了したときに再度割り込みをイネーブルとします。
割り込み関数とメイン・プログラムとの間に次に示されるような関係を保つことで割り込みから呼びだせるようになります。このことは、MS-DOS のシステム・コールと割り込みの関係に似ているかもしれません。

1. 先ほど考察したように関数 A()が実行する前に割り込みでこの関数が呼び出されないように設定することです。
2. 関数 A()を ENABLE/DISABLE_INTERRUPT で決して再帰的な呼び出しがなされないよう INTCON レジスターを操作します。さらにグローバルインタラプトフラグがイネーブルだと、関数 A()が間違って実行されることがなくなります。
3. プログラムは割り込みがディセーブルな状態ならどのような状態であっても依存しなくなります。コンパイラーは割り込みがディセーブルでないなら関数の呼び出しや関数内でローカルな RAM を必要とします。
4. 割り込みがディセーブルならコンパイラー内部の関数を呼び出しても差し支えありません。例えば乗算（*）を行う場合でも当てはまります。

Q: なぜコンパイラーは現在使われない TRIS を使用するのでしょうか？

A: TRIS の扱いはどのユーザーでも関心があることです。マイクロチップ社のデータシートによりますと TRIS 命令を使用しないコンパチブルな内容があります。アセンブラー・コードに依存し、新しいマイクロチップ社の TRIS を使わないポートの設定は大変困難を伴います。

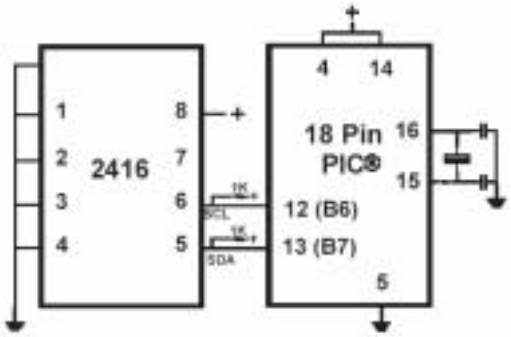
C ではこの実装は今のところ考えられていません。コンパイラーはそれぞれのデバイスにより設定されたデータベースを持っています。TRIS だけでなくいろいろな部分のデバイスの違いをこれらのデータベースで吸収しています。この質問には、デバイス・データシートの誤植をご覧にな

られたのだと思われます。

CCS では新しいデバイスを追加していますのでデバイスのデータの改訂や誤植のあるデータシートを見たことがあります。PCW ユーザーはデバイス・データベースを編集することができます。TRIS の使い方は単にデータベース・エントリーを交換するだけでコンパイラーに関わることではありません。

Q: I²C デバイスに PIC を用いるには？

A: 2 線式で接続できる I²C は、それぞれのラインをプルアップしておきます。大抵の I²C デバイスはハードウェアかスイッチでアドレスを選択しておきます。そして、参考図に示すような形で接続します。



Q: なぜコンパイラーが 0 番地をコールするのですか？

A: PIC の ROM アドレスフィールドの 8~10bit はオペコードとチップに依存します。アドレス・ビットの残りはレジスタにより決定されます。例えば、74 チップでアドレス 800h を 1 ページ目からコールすると

```
BSF   0A,3
CALL 0
```

この様に 800H のコールなのに 000h が呼び出されているように見えますが、ビット 11 が PCLATH レジスタ 0Ah のビット 3 をセットすることで 800H を指定しています。

Q: なぜコンパイラーが A0 の代わりに 20 番地をアクセスするのですか？

A: PIC の RAM アドレスフィールドの 5～7bit はオペコードとチップに依存します。アドレス・ビットの残りはレジスタにより決定されます。例えば、74 チップでアドレス A0h を W レジスタにロードすると

```
BSF      3,5
MOVFW    20
```

コンパイラーの最適化はアクセスする前のバンクスイッチ BSF を使う冗長なコードを除外するかもしれません。

Q: 直接レジスターをアクセスする方法はありますか？

A: ハードウェアレジスタは、C の変数として割り当てることにより直接リード/ライトすることができます。タイマー0 レジスタをアクセスする例を示します。

```
#BYTE   timer0 = 0x01
timer0 = 128;                  // タイマー0 レジスターに 128 をセット
while ( timer0 = 200 );        // タイマー0 が 200 になるのを待つ
```

また、レジスタの特定のビットを割り当てるには次のようにします。

```
    #BIT   T0IF = 0x0b.1              // タイマー0 の割り込みビット
…
    while( !T0IF );          // タイマー0 割り込みを待つ
```

レジスターは間接アドレスによってアクセスすることもできます。次にそのような例を示します。

```
printf ( “enter address:” );
a = gethex();
printf( “¥r¥n value is %x¥r¥n”, *a );
```

コンパイラはたくさんの組み込み関数内で C 関数コールを多くの共通した手段により行っています。そこでは可能な限り直接レジスタのアクセスを行っています。レジスタのロケーションはチップやおなじレジスタを操作するときでも関連するレジスタの値を変更することにより変化します。コンパイラーはこれらの違いを組み込み関数のインプリメンテーションに影響を与えないように割り当てます。
例えば、*0x85=0;とするより set_tris_a(0)という関数を使った方がこれらのトラブルを回避することができます。

Q: ROM エリアに定数データテーブルを置きたいのですが？

A: コンパイラーは ROM エリアに読み出し専用のデータ構造を置くことをサポートしています。PIC デバイスは、ROM と RAM のデータは完全に分離されています。そしてこれらのアクセスに対して制限を加えています。例えば 10 個の要素を持つバイト型の配列を ROM に取るときは次のようにします。

```
BYTE CONST TABLE = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
```

この配列 TABLE のアクセスは

```
x = TABLE[i ];
```

または、

```
x = TABLE[5];
```

とはできますが、次のようにはできません。

```
ptr = &TABLE[ i ];
```

この場合、テーブルのポインターは作成できません。

同様に CONST を使用することにより LONG 型や FLOAT 型のデータ型を持つ構造も使用することができます。上記のようなテーブルの添え字を使ったアクセスのインプリメンテーションはコンパイル時間が掛かります。

Q: RB ポートの割り込みでボタンのプッシュを検出したいのですが？

A: RB ポートの割り込みは B4~7 のピンが入出力で変化したとき発生します。よって、割り込みがかけられたとき、どこは変化したかは分かりません。そこで、プログラマーは変化する前のポートの状態を保持しておき、割り込みが掛かったときどこがどのように変化したかを知ることができます。さらに、ボタンがただ１回押されたときでも、スイッチ接点のチャタリングにより予期しない割り込みか掛かってしまう場合もあります。これは機械式の接点では避けられないものでこれに対応したプログラムが要求されます。そこで、簡単なサンプルを示します。

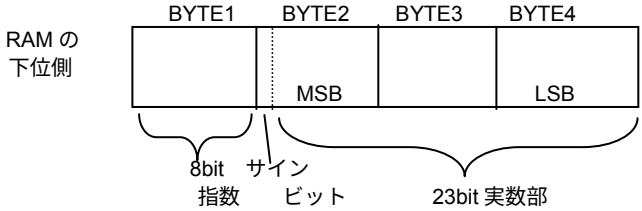
```
#int_rb
rb_isr ( ){
    byte changes;
    changes = last_b ^ port_b;
    last_b = port_b;
    if( bit_test ( changes, 4 ) && !( bit_test( last_b, 4 ) ){
                                // RB4  が Low に
    }
    if( bit_test ( changes, 5 ) && !( bit_test( last_b, 5 ) ){
```

```
 …                    // RB5 が Low に
delay_ms( 100 );                // チャタリング・キャンセル
 }
```

ディレイに 100mS を与えているのは、瞬間的に押された場合や、チャタリングをキャンセルするためです。また、これにより 100mS 以上でないと割り込み関数 ISR は呼ばれません。もう少しエレガントな方法は、キーが押されたときの割り込みでタイマーを使用してタイマーがオーバーフローしたときにポートが変化していなければピンの変化があったとすれば良いと思われます。

Q: 浮動小数点のフォーマットはどのようになっていますか？

A: CCS が採用しているフォーマットは、マイクロチップ社が 14000 デバイスで使用しているフォーマットと基本的に同じです。PCW のユーザーなら PCONVERT を実行させてみることによりフォーマットが分かります。サンプル・ファイルに“EX_FLOAT.C”がありますのでこれも大変良い参考となります。フォーマットは次のようになっています。



（7F のバイアスがのっています)

参考値：

0	00 00 00 00
1	7F 00 00 00
-1	7F 80 00 00
10	82 20 00 00
100	85 48 00 00
123.45	85 76 E6 66
12.45E20	C8 27 4E 53
123.45E-20	43 36 2E 17

RAM の下位バイト

Q: コンパイラーが RAM 不足をいってきていますが本当でしょうか？

A: いくつかのデバイスでは標準的な RAM のアクセスを行っていると RAM 不足な場合があります。これらのデバイスには PIC16C509,57,66,67,76,77 などがあります。デフォルトではコンパイラーは自動的に変数を RAM に割り当て、RAM の下位アドレスから利用できるように思われます。

3 つの RAM の使い方を示します。

1．#BYTE、＃BIT ディレクティブで RAM に変数を割り当てます。これらの変数にはポインターは作成できません。

例: ＃BYTE counter = 0x30

2．READ_BANK,WRITE_BANK 関数で配列のように RAM をアクセスします。このアクセスでは、この RAM に配列を割り当てたようなものです。

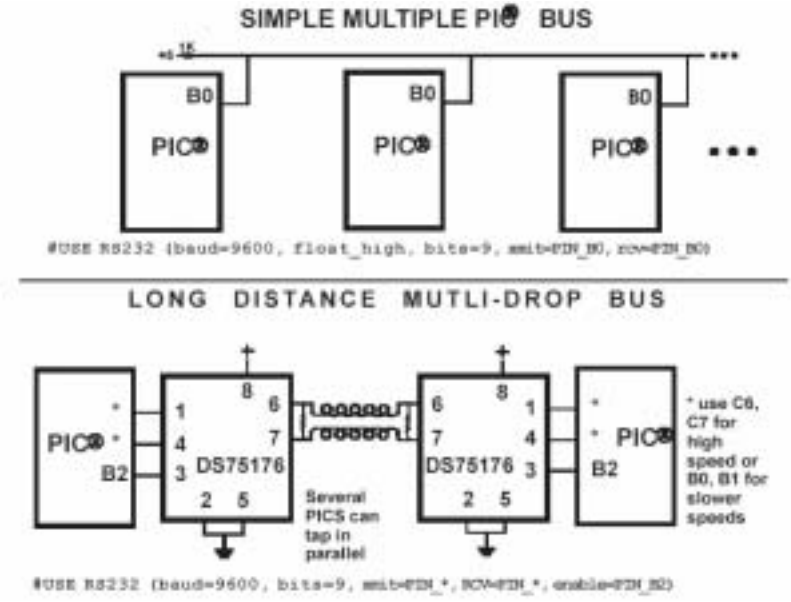
例: for(i = 0; i < 15; i++)

```
    write_bank( 1,i, getc() );
for( i = 0; i < 15; i++ )
    putc( read_bank( 1, i ) );
```

3. PCM ユーザーはフル RAM アクセスのために 16bit ポインターに切替えることが出来ます。
これはより多くの ROM を必要とします。#DEVICE に*=16 を追加して下さい。
例： #DEVICE PIC16C77 *=16

Q: 2 個もしくは複数個の PIC 同士のコミュニケーションを取りたいのですが？

A: 2 本の参考プログラム“EX_PBUSM.C”、“EX_PBUSR.C”は単に 1 本の配線で PIC 関の通信を行わせるものです。スローデータ転送を RB0 ポートと外部割り込みを使って実現させています。内蔵の UART を使用して高速で長距離のデータ転送を行えます。RS485 ドライバー/レシーバーは高速の UART を 2 線で接続するものです。それぞれのハードウェアの概略を以下に示します。



Q: バイトでない EEPROM にどのように変数を書いたらよいですか？

A: 下記が EEPROM に浮動点を読み書きする方法の例です。同様のコンセプトをストラクチャー、アレイ又は、その他のタイプに使用することが出来ます。

- n は eeprom へのオフセットです。
- フロートは 4 でインクレメントしなければいけません。
例えば、最初のフロートが 0 だとしますと、２番目は 4，３番目は 8 です。

```
WRITE_FLOAT_EXT_EEPROM(long int n, float data) {
  int i;

  for (i = 0; i < 4; i++)
    write_ext_eeprom(i + n, *(&data + i) ) ;
  {
  float REAT_FLOAT_EXT_EEPROM(long int n) {
     int i;
      float data;
```

```
        for (i = 0; i < 4; i++)
          * (&data + i) = read_ext_eeprom(i + n, *) ;
        return(data);
    {
```

Q: 指定された時間後にタイムアウトするような getc()は作れますか?

A: getc()は常にキャラクタを取得するまで待ちます。
キャラクタの準備ができるまで、getc()をコールしないことが賢明です。kbhit()によりキャラクタの準備が完了したことを調べることができます。以下は、RS232 からキャラクタを受信するときのタイムアウト処理例です。ハードウェア UART を使わない場合は、delay_us は 1bit 分のパルス幅の 1/10 より小さくしてください(9600 ボーで 10μs)。ハードウェア UART を使う場合は 1bit 分のパルス幅の 10 倍まで設定することが可能です(9600 ボーで 1000μs)。
もし、65535 以上のタイムアウト値が必要な場合は、二つのカウンタを使用します。

```
short timeout_error;

char timed_getc() {
    long timeout;

  timeout_error=FALSE;
  timeout=0;
  while(!kbhit&&(++timeout<50000))                // 1/2 秒
      delay_us(10);
  if(kbhit() )
      return(getc());
  else {
      timeout_error=TRUE;
      return(0);
  }
}
```

Q: どうしたら変数を OUTPUT_HIGH()のような関数に渡すことができますか？

A: OUTPUT_HIGH のような組込み関数の引数である pin はコンパイル時にポート及びビットが確定していなければなりません。アプリケーションでコンパイル時に異なった幾つかのピンのどれを使うか決定できない場合は以下のようにします。

```
Switch(pin_to_use) {
        case PIN_B3   :   output_high(PIN_B3); break;
        case PIN_B4   :   output_high(PIN_B4); break;
        case PIN_B5   :   output_high(PIN_B5); break;
        case PIN_A1   :   output_high(PIN_A1); break;
        }
```

決まったポートの任意ビットを使う場合は：

```
#byte portb = 6
#byte portb_tris = 0x86  // **

port_tris &=  ～(1<<bit_to_use);       // **
portb  ｜= (1<<bit_to_use);             //bit_to_use is 0-7
```

任意ポートの任意ビットを使う場合は：

```
*(pin_to_use/8｜0x80) &=  ～(1<<(pin_to_use&7)): file://**
 *(pin_to_use/8)  ｜= (1<<(pin_to_use&7)):
```

この場合は、pin_to_use で使う全てのピン(PIN_A0...)を上記の case 文で定義する必要があります。
上記のコードで**行は、ピンの入出力方向を変更する場合のみ必要となります。

Q: インサーキットデバッガ(IDC)を使うために、アドレス 0 に NOP 命令を置かなければいけませんか？

A: CCS のコンパイラは MPLAB を使ったマイクロチップの ICD デバッガー互換です。ICD デバッグ(アドレス 0 に NOP 命令を置く等)のためにプログラムで準備することは、他に定義した #DEVICE 命令の後に、#DEVICE ICD=TRUE を追加することです。

例：#INCLUDE <16F877.h>
#DEVICE ICD=TRUE

Q: printf の出力を文字配列に格納するには？

A: 以下は printf の出力を文字配列に格納する例です。ここで、¥f は文字列への格納を開始するためのものです。このサンプルは float[実数]を文字配列に格納する例です。

```
char string[20];
byte stringptr=0;

tostring(char c) {
   if(c=='¥f')
      stringptr=0;
   else
      string[stringptr++]=c;
   string[stringptr]=0;
}

main() {
    float f;

    f=12.345;

    printf(tostring,"¥f%6.3f" ,f);
}
```

Q: どのように関数へのポインタをつくりますか？

A: コンパイラは、コンパイラがコンパイル時に完全なコールツリーを知る必要があるため、関数へのポインタを許可しません。コールツリーは完全な RAM 再使用のためのメモリ割り当てに使用します。同時に実行されない関数では、関数は同じ RAM 領域を使用することがあります。加えて PIC にはデータスタックがありません。関数のパラメータは、どのような関数がコールされるのかをコンパイル時に調べて特別な方法で渡されます。ポインタにより関数をコールすることは、コンパイル時にこれら両方のことを知ることを妨げます。ステートマシンを作成するためには関数のポインタが欲しくなりますが、以下はポインタを使わないで如何にこれをおこなうかの例です。

```
enum tasks {taskA, taskB, taskC};

run_task(tasks task_to_run) {

        switch(task_to_run) {
        case taskA : taskA_main(); break;
        case taskB : taskB_main(); break;
        case taskC : taskC_main(); break;
        }
```

```
}
```

Q: 演算操作にどれくらいを要しますか？

A: 符号なし 8 ビット操作では非常に高速ですが、浮動小数点では大変遅くなります。もし、よろしければ浮動小数点の代わりに固定点を使用してみて下さい。下記は 20mhz, 14bit PIC マイコンのおよその時間です。時間は使用されているメモリー・バンクに著しく依存します。

8bit add	<1us
8bit multiply	9us
8bit devide	20us
16bit add	2us
16bit multiply	48us
16bit devide	65us
32bit add	5us
32bit multiply	138us
32bit devide	162us
float add	32us
float multiply	147us
float devide	274us
exp()	1653us
in()	2676us
sin()	3535us

Q: I/O ピンのバイト幅での入出力の方法は？

A: 組み込み関数の INPUT、OUTPUT_HIGH、OUTPUT_LOW は、すべてピンに対応するものです。つまり 1bit の I/O を行なう関数です。
バイト幅で I/O を行なうための例を次に示します。

```
#byte        port_b = 6;
#define     all_out   0
#define     all_in     0xff
main(){
    int   i;

    set_tris_b( all_out );              // B ポートを出力に設定
    portb = 0;                          //  すべて’L’を出力
    for ( i = 0; i <=127; i++ )
         port_b = i;                    // 0～127 を出力
    set_tris_b( all_in );               //  入力ポートに設定
    i = port_b;                         // B ポートのデータ入力
}
```

#BYTE でポートレジスタを指定します。（メモリー・マップド I/O と同じ）トライステート・レジスター設定を入力（0）出力（1）に指定し、あとはポートに書き込むことで出力。ポートから読み込むと入力となります。

次に構造体を用いた I/O アクセスの例を示します。

```
struct   port_b_layout
                        {int   data   : 4;          // data を 4bit
                         int   rw     : 1;          // rw を 1bit
                         int   cd     : 1;          // cd を 1bit
                         int   enable: 1;           // enable を 1bit
                         int   reset  : 1; };       // reset を 1bit

struct port_b_layout   port_b;          // port_b を構造体 port_b_layout 型で宣言
```

```
#byte port_b = 6;                       // port_b をポートに関連付ける
                                        // data,re,cd,enable,reset メンバーの初期値
struct port_b_layout   const   INIT_1 = {0,1,1,1,1};
struct port_b_layout   const   INIT_2 = {3,1,1,1,0};
struct port_b_layout   const   INIT_3 = {0,0,0,0,0};
struct port_b_layout   const   FOR_SEND = {0,0,0,0,0};      //全て出力
struct port_b_layout   const   FOR_READ = {15,0,0,0,0};     //データは入力
main() {
    int   x;
    set_tris_b( (int)FOR_SEND );   // 0x00 で出力設定

    port_b = INIT_1;                    // 初期化構造体を使って順に初期化
    delay_us(25);
    port_b = INIT_2;
    delay_us(25);
            port_b = INIT_3;

    set_tris_b( (int)FOR_READ );   // 入力設定
    port_b.rw = 0;

    port_b.cd = 1;
    port_b.enable = 1;
    x = port_b.data;
    port_b.enable = 0;
}
```

Q: FAST I/O と STANDARD I/O の違いは？

A: 最初に覚えてほしいことは、fastI/O モードは、I/O を操作することではないということです。プログラマーは、I/O ポートの使用に先立ち SET_TRIS_x()関数でポートを入力か出力ポートかに設定します。SET_TRIS_x()関数は、バイト値を与え、'1'ならば入力、'0'ならば出力として値を構成します。例えば次の例ではポートＢの RB7 ポートを入力、他のポートを出力とするものです。

```
set_tris_b(0x80);
```

次に、fastI/O モードでは、大変高速で次のようなコードで

```
output_high(PIN_B0);
output_low(PIN_B1);
```

この場合、BSF6,0、BCF6,1 というアセンブリ言語に変換します。マイクロ・プロセッサーはこれらの BSF、BCF のコードを実行する場合、ポートを読み込み、ビットを変更し、そして書き戻します。

この例では、BCF を実行した後、Ｂ０ピンの状態が保持されるとは限りません。bsf、bcf 命令では入力バッファからの読み込み後ビットの演算を行ないその後、出力ラッチへの出力と移行します。

よって参照される入力バッファの値により設定した通りの出力が得られない場合があります。standard や fixedI/O ではこのような連続アクセスを行なわず､ピンの状態が安定するように次の様な命令に展開されます。

```
output_high(PIN_B0);
delay_cycles(1);
output_low(PIN_B1);
```

間に挿入されたディレイは、NOP と等しく、20MHz のクロックで 0.2μS となります。ポート外部に接続する回路によってはもう少しリカバリを取らないといけない場合があるかもしれません。

Q: BIT 型変数はどうして使用されるのでしょうか？ ソースファイルの可読性が悪くなるように思えます。

A: bit 変数(SHORT INT)は大幅に RAM エリアを節約する効果があります。
次の例を見てください

```
int   x,y;
short int bx,by;
x=5;
y=10;
bx=0;
by=1;
x=(x+by)-bx*by+(y-by);
```

ここに示された例で bx と by に使用されている算術演算子（＋と－,＊）がある場合、最初に bx と by は byte(int)に拡張され、これはまずいことです。実行スピードとコードサイズを節約するためには次のように書き換えてみます。

```
x=by;
x=(x+z)-bx*z+(y-z);
```

多少はましになりました（型変換の数が少なくなっている）が、やはりあまりよい形とはいえません。
このような演算を行なわせるには、IF 文でまずビット操作を行なった後、byte 演算を行なわせます。
勿論、IF 文で使用される演算子は、&&、||、!が効果的です。
例を見てみます。

```
if (by || (bx && bz) || !bw)
z=0;
```

!と~、&&と&、||と|の違いを思い出してください。
同様に、input()関数も返される値は、bit 型です。これを if 文で判定するには次の例を使用します。

```
if ( !inout( PIN_B0) )          // 1
if ( inout( PIN_B0) == 0 )      // 2
```

１の例はビット演算で処理され、２の例では、byte 型へキャストされてから演算されますので悪い例です。このように、プロセッサーの特色をいかしたコードを出力するように作成されたコンパイラーですから、なるべくコンパクトで実行スピードの速いソースコードを書くべきです。可読性はコードの書き方とコメント文でかなりカバーできます。

サンプル・プログラム

サンプルプログラムは、全て付属のディスクに収められています。
EXAMPLES ディレクトリーの EXAMPLES.TXT ファイルを参照してください。このファイルに収めてあるファイルのリストがあります。また、多くのプログラムは、デバイス定義やインクルード・ファイルの定義がされていません。このマニュアルをお読みになられた方であればコメントヘッダーにある程度の情報が記入されていますので参考にしてプログラムをコンパイルできると思います。

また、“SIO.EXE”というユティリティがあり、このプログラムを PC（IBM-PC、または互換機）で動作させるとかんたんなターミナルとして PC が使用できますのでターゲットとの通信を試してみることができます。

ジェネリックヘッダー・ファイルは PIC デバイスの各部を定義したファイルで、DEVICES ディレクトリにあります。チップのピンは PIN_B2 などのようにこれらのファイルで定義してあります。プログラムを新しいプロジェクトとして作成するとき、プロジェクト・ヘッダーにこれらの情報が取り込まれます。勿論、ピン定義など全ての情報を編集することができますが、元のヘッダー・ファイルは直接編集を行なわない方がいいでしょう。
プロジェクトに取り込むには、C ソースファイルに次の１行を加えます。
#include <16c74.h>
ヘッダ・ファイルのチップ名はチップのおおよその型名を表し、同じタイプのチップで異なったデバイスを使用するときは#DEVICE ディレクティブで再指定します。

```
#include <16C74.H>
#device PIC16C74B
```

次の各サンプルソースの簡単な説明を行なっておきます。

EX_14KAD
PIC14000 と 14KCAL.C を使って AD コンバータを動作させます。
EX_1920
Dallas DS1920 ボタンを使用し温度の読み出します。
EX_8PIN
特別な I/O で 8 ピン PIC ピンお使用をデモしています。
EX_92LCD
923/924 デバイスで LCD の使い方を示しています。
EX_AD12
外部に設置した 12bit の A D コンバーターを動作させます。LTC1298.C ドライバーを利用します。
EX_ADMM
30 回の AD コンバータのサンプリングを行ない、最大値と最小値を RS232C から出力します。この処理を繰り返します。
EX_CCP1S
ハードウェア CCP でワンショット・パルスを生成します。
EX_CCPMP
ハードウェア CCP を使ってパルスの“H”の時間を測定します。
EX_COMP
ある種の PIC で利用できるアナログ・コンパレータと電圧リファレンスを使用しています。
EX_CRC
高速でパワフルなビット操作を示しながらメッセージ上の CRC を計算します。
EX_CUST
特別なプリプロセッサーを使ってコンパイラーを変更します。
EX_DPOT
このプログラムは DS1868.C を使ってデジタル POT をインプリメントする方法を示します。
EX_DTMF
DTMF トーンを生成します。
EX_ENCOD
光学エンコーダーに方向と速度を決定させるインターフェースです。

EX_EXPIO
74165.C と 74595.C を使ってさらに入力と出力ピンを作成するのに外部チップをどの様に使うかを示すプログラムです。
EX_EXTEE
EEPROM のリード／ライトのプログラムです。プログラムによっては、わりと大きな型番の PIC デバイス(6x や 74 など)が必要ですが、次に示すように多くの EEPROM をテストできます。
2401.C, 2402.C, 2404.C, 2408.C, 2416.C, 2432.C, 2465.C, 9346.C, 9356.C, 9366.C, 9356SPI.C
24xx.C では I^2C ポートのデモンストレーション・プログラムでそれ以外のプログラムは 2-3 本の配線で EEPROM とシリアル接続します。SPI のつくファイルは、内蔵の SPI ハードウェアポートを利用したプログラムとなっています。
EX_FLOAT
コンパイラーの浮動小数点のデモです。
EX_FREQC
50MHz 周波数・カウンター
EX_GLINT
高速インターラプトのためのカスタム・グローバル・インターラプト・ハンドラーの定義を示します。
EX_INTEE
チップ内部に EEPROM を持っているチップの EEPROM アクセスプログラムです。
EX_LCDKB
16x2 の LCD と 3x4 のキーボードを LCD.C、KBD.C を使ってデモします。
プログラムの実行には、LCD 表示器とキーパッドが必要です。
EX_LCDTH
現在、最小、最大温度を LCD に表示します。
EX_LED
2 デジット 7 セグメント LED を直接ドライブするプログラムです。
EX_LOAD
16F877 のようなチップに対するシリアル・ブート・ローダー・プログラム
EX_MACRO
Ｃでのパワフルなマクロの例
EX_PATG
異なった周波数で 8 方形波を発生します。
EX_PBUSM
１線式の PIC 通信プログラムです。
EX_PBUSR
１線式で PIC とシェアード RAM とした PIC との通信を行います。いくつかの番号を持つ PIC を１本の通信線に接続しそれぞれ RAM ブロックのように扱います。１つの PIC の RAM が変化するとすべての PIC の RAM が同じように変化します。
EX_PBUTT
プッシュ・ボタンを検出するために B ポートのインターラプトの変更の使用方法を示します。
EX_PGEN
周期とデューティー・スイッチ選択でパルスを発生
EX_PLL
ラジオをチューンするための外部周波数・シンセサイザーへのインターフェース
EX_PSP
プリンター・パラレルからシリアル・コンバーターのための PIC PSP の使用を示します。
EX_PULSE
Timer0 を使ってパルス幅を測定します。
このプログラムは、RTCC(Timer0)を用いたシングルパルスを PIC 入力すると RTCC の値を 10 進で出力します。RTCC の基本的な使用方法が分かります。
EX_PWM
パルス・ストリームを発生するために PIC CCP を使用します。

EX_REACT
CCP モジュールを使ってリレイ・クロージングのリアクション時間
EX_RMSDB
RMS ボルテージと AC シグナルのデシベル・レベルを計算
EX_RTC
RS232C ポートを通じて、外部に設けた RTC（リアルタイムクロック）を設定するプログラムです。RTC によって次のプログラムをインクルードしてください。
DS1302.C、 NJU6355.C
EX_RTCLK
LCD とキーバッドを使って外部リアル・タイム・クロックをセットし、読みます。
EX_SINE
D/A コンバーターを使って正弦波を発生
EX_SISR
RS232C レシーバーを割り込み駆動としたプログラムです。
EX_SLAVE
240LC01 EEPROM をエミュレートするために PIC を I2C スレーブ・モードで使用したプログラムです。
EX_SPEED
モデル・カーのような外部オブジェクトの速度を測定
EX_SPI
H/W SPI モジュールを使ってシリアル EEPROM との通信
EX_SQW
小さなプログラムで、RS232C よりプログラムの動作開始メッセージを出力した後、１KHz の方形波を出力します。基本的な組み込み関数だけで作成されています。
EX_SRAM
DS2223.C、PCF8570.C ドライバーを使って RAM チップをアクセスします。メモリ増設の参考にしてください。
EX_STEP
ステッピング・モータードライブのプログラムです。
EX_STR
関数を扱うベーシックＣストリングをどのように使うかを示します。
EX_STWT
このプログラムは、割込で１秒のカウンタを持っていて、RS232C を使ってストップウォッチのように動作します。割込のかんたんなサンプルです。
EX_TANK
特殊な形状のタンク内の液体を計算するためにトリグ関数を使用します。
EX_TEMP
デジタル・センサーから温度を表示(RS232 経由)
EX_TGETC
RS232 データ待ちを如何にタイムアウトさせるかをデモ
EX_TONES
“HAPPY BIRTHDAY”を演奏するトーンをどのように発生させるか示します。
EX_TOUCH
DALLAS セミコンダクターのタッチ・デバイスを使用したサンプル・プログラムです。
TOUCH.C ドライバーが必要です。
EX_USB
PIC16C765 での USB デバイスを実行
EX_VOICE
ボイス・プログラムのためのセルフ・ラーニング・テキスト
EX_WDT
PIC のウォッチ・ドッグ・タイマーの使用方法
EX_X10
X10.C ドライバー・プログラムを利用して RS232C と X10 をインターフェースします。

インクルード・ファイル・リスト
14KCAL.C
PIC14000 A/D コンバーターのためのカリブレーション関数
2401.C
シリアル EEPROM 関数
2402.C
シリアル EEPROM 関数
2404.C
シリアル EEPROM 関数
2408.C
シリアル EEPROM 関数
4128.C
シリアル EEPROM 関数
2416.C
シリアル EEPROM 関数
24256.C
シリアル EEPROM 関数
2432.C
シリアル EEPROM 関数
2465.C
シリアル EEPROM 関数
25160.C
シリアル EEPROM 関数
25320.C
シリアル EEPROM 関数
25640.C
シリアル EEPROM 関数
25C080.C
シリアル EEPROM 関数
68HC68R1.C
シリアル RAM 関数
68HC68R2.C
シリアル RAM 関数
74165.C
拡張入力関数
74595.C
拡張入力関数
9346.C
シリアル EEPROM 関数
9356.C
シリアル EEPROM 関数
9356SPI.C
シリアル EEPROM 関数(H/W SPI 使用)
9366.C
シリアル EEPROM 関数
AD7715.C
A/D コンバータ関数
AD8400.C
デジタル POT 関数
AT25256.C
シリアル EEPROM 関数
CE51X.C
12CE51x EEPROM へのアクセス関数

CE62X.C
12CE62x EEPROM へのアクセス関数
CE67X.C
12CE67x EEPROM へのアクセス関数
CTYPE.H
関数を扱う各種特性の定義
DS1302.C
リアル・タイム・クロック関数
DS1621.C
温度関数
DS1868.C
デジタル POT 関数
FLOATEE.C
EEPROM へ浮動小数点を読み・書きするための関数
INPUT.C
RS232 経由でストリングスと番号を読むための関数
KBD.C
キーパッドを読むための関数
LCD.C
LCD モジュール関数
LOADER.C
簡単な RS232 プログラム・ローダー
LTC1298.C
12bit A/D コンバーター関数
MATH.H
各種スタンダード・トリグ関数
MAX517.C
D/A コンバータ関数
MCP3208.C
A/D コンバータ関数
NJU6355.C
リアル・タイム・クロック関数
PCF8570.C
シリアル RAM 関数
STDIO.H
ここでは多くはありません。- スタンダード C 互換のために用意されています。
STDLIB.H
番号関数へのストリング
STRING.H
各種スタンダード・ストリング関数
TONES.C
トーン発生のための関数
TOUCH.C
各種スタンダード・ストリング関数
X10.C
X10 コードを読み/書きするための関数

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///                                                EX_SQW.C                                                      ///
/// This program displays a message over the RS-232 and waits for                                               ///
/// any keypress to continue. The program will then begin a 1khz                                                ///
/// square wave over I/O pin B0.                                                                                ///
/// Change both delay_us to delay_ms to make the frequency 1 hz.                                                ///
/// This will be more visible on a LED.                                                                         ///
/// Configure the CCS prototype card as follows:                                                                ///
/// insert jumpers from: 11 to 17 and 12 to 18 and 42 to 47. :                                                  ///
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#ifdef __PCB__
#include <16C56.H>
#else
#include <16C84.H>
#endif

#use delay(clock=20000000)
#use rs232(baud=9600, xmit=PIN_A3, rcv=PIN_A2)

main()   {
        printf("Press any key to begin¥n¥r");
        getc() ;
        printf("1 khz signal activated¥n¥r");
        while (TRUE)   {
        output_high (PIN_B0);
        delay_us(500);
        output_low (PIN_B0);
        delay_us(500);
        }
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////                                   EX_STWT.C                                                                ////
////                                                                                                            ////
//// This program uses the RTCC (timer0) and interrupts to keep a                                               ////
//// real time seconds counter.    A simple stop watch function is                                              ////
//// then implemented.                                                                                          ////
////                                                                                                            ////
//// Configure the CCS prototype card as follows:                                                               ////
////         Insert jumpers from: 11 to 17 and 12 to 18.                                                        ////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#include <16C84.H>

#fuses HS,NOWDT,NOPROTECT

#use delay(clock=20000000)
#use rs232(baud=9600, xmit=PIN_A3, rcv=PIN_A2)

#define INTS_PER_SECOND 76        // (20000000/(4*256*256))

byte seconds;          // A running seconds counter
byte int_count;       // Number of interrupts left before a
                              // second has elapsed

#int_rtcc                 // This function is called every time
clock_isr() {          // the RTCC (timer0) overflows (255->0).
                           // For this program this is apx 76 times
      if(--int_count==0) {                 // per second.
         ++seconds;
         int_count=INTS_PER_SECOND;
      }

}

main() {

    byte start;

    int_count=INTS_PER_SECOND;
    set_rtcc(0);
    setup_counters( RTCC_INTERNAL, RTCC_DIV_256);
    enable_interrupts(RTCC_ZERO);
    enable_interrupts(GLOBAL);

    do {

        printf("Press any key to begin.¥n¥r");
        getc();
        start=seconds;
        printf("Press any key to stop.¥n¥r");
        getc();
        printf("%u seconds.¥n¥r",seconds-start);

    } while (TRUE);

}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////
///                                        EX_INTEE.C                                            ///
///   This program will read and write to the 83 or 84 internal EEPROM                           ///
///              Configure the CCS prototype card as follows:                                    ///
///             Insert jumpers from: 11 to 17 and 12 to 18                                       ///
////////////////////////////////////////////////////////////////////////////////////////////////////
#include<16C84.H>

#use delay(clock=20000000)
#use rs232(baud=9600, xmit=PIN_A3,rcv=PIN_A2)

#include<HEX.C>

main()   {
            byte I, j, address, value ;

            do {
                        printf("¥r¥n¥nEEPROM:¥r¥n) ;    //Display contents
                        for (I=0 ; I<=3;    ++j)   {
                              for (j=0 ; j<=15;    ++j)   {   //in hex
                                    printf("%2x", read_eeprom(I*16+j ) ) ;
                              )
                              printf("¥n¥r") ;
                        }
                        printf("¥r¥nLocation to change: ") ;
                        address = gethex()   ;
                        printf("¥r¥nNew value: ") ;
                        value = gethex() ;

                        write_eeprom( address, value) ;
            } while (TRUE)
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////
//// Library for a Microchip 93C56 configured for a x8                                          ////
//// org init_ext_eeprom(); Call before the other                                               ////
//// functions are used write_ext_eeprom(a,d); Write                                            ////
//// the byte d to the address a d=read_ext_eeprom (a);                                         ////
//// Read the byte d from the address a. The main program                                       ////
//// may define eeprom_select, eeprom_di, eeprom_do and eeprom_clk                              ////
//// to override the defaults below.                                                            ////
////////////////////////////////////////////////////////////////////////////////////////////////////

#ifndef EEPROM_SELECT

#define EEPROM_SELECT               PIN_B7
#define EEPROM_CLK          PIN_B6
#define EEPROM_DI           PIN_B5
#define EEPROM_DO           PIN_B4

#endif

#define EEPROM_ADDRESS byte
#define EEPROM_SIZE                 256

void init_ext_eeprom () {
        byte cmd[2];
        byte I;

        output_low(EEPROM_DI);
        output_low(EEPROM_CLK);
        output_low(EEPROM_SELECT);

        cmd[0]=0x80;
        cmd[0]=0x9;

        for (i=1; i<=4; ++i)
                shift_left(cmd, 2,0);
        output_high (EEPROM_SELECT);
                for (i=1; i<=12; ++i) {
                        output_bit (EEPROM_DI, shift_left(cmd, 2,0));
                        output_high (EEPROM_CLK);
                        output_low(EEPROM_CLK);
}
                        output_low(EEPROM_DI);
                        output_low(EEPROM_SELECT);
}

void write_ext_eeprom (EEPROM_ADDRESS address, byte data)   {
        byte cmd[3];
        byte i;
                         cmd[0]=data;
                         cmd[1]=address;
                         cmd[2]=0xa;

                         for(i=1;i<=4;++i)
                                 shift_left(cmd,3,0);
                         output_high(EEPROM_SELECT);
                         for(i=1;i<=20;++i) {
```

```
                                output_bit (EEPROM_DI, shift_left(cmd,3,0));
                                output_high (EEPROM_CLK);
                                output_low(EEPROM_CLK);
                        }
                        output_low (EEPROM_DI);
                        output_low (EEPROM_SELECT);
                        delay_ms(11);
}

byte read_ext_eeprom(EEPROM_ADDRESS address) {
        byte cmd[3];
        byte i, data;

        cmd[0]=0;
        cmd[1]=address;
        cmd[2]=0xc;

         for(i=1;i<=4;++i)
                shift_left(cmd,3,0);
         output_high(EEPROM_SELECT);
         for(i=1;i<=20;++i) {
                output_bit (EEPROM_DI, shift_left(cmd,3,0));
                output_high (EEPROM_CLK);
                output_low(EEPROM_CLK);
                if (i12)
                        shift_left (&data, 1, input (EEPROM_DO));
          }
          output_low (EEPROM_SELECT);
          return(data);
}
```

ソフトウエア使用許諾合意及び、著作権

このソフトウエアのパッケージを開けることで次の条件に同意したことになります。同意しない場合はパッケージを未開風のままご返却下さい。代金はお返し致します。

1.ライセンス: Custom Computer Services(“CCS”)は 1 つのコンピューターでのこのソフトウエア・プログラムの使用のライセンスの許可を与えます。ネットワーク及び、複数での使用は追加料金で行えます。

2.アプリケーション・ソフトウエア: ユーザーがライセンスを受けたこのソフトウエアを使って作成したデリバティブ・プログラム(派生的なプログラム)をここではアプリケーション・プログラムと呼びますが、アプリケーション・プログラムはこの合意の対象外となります。

3. 保証: CCS はメディアとそのワークマンシップの欠陥は保証します。ソフトウェアはユーザー購入の日付の数日後にユーザー登録をすることで 30 日の無償アップデートが出来ますが、CCS は、このアップデートによるライセンス・マテリアルの欠陥の解決と特定の必要条件を満たすことは保証しません。

4.制限: CCS はライセンス・マテリアルに関して、特定の目的に合致していることをいかなる条件のもとでも保証致しておりません。
マニュアルの内容及び、使用に於いて発生した如何なる物品、人体に対する障害や損害を保証するものではありません。

5. 転売 : ライセンス同意者は、CCS が最初に販売した国以外にライセンス・マテリアルを転売、輸出しないことに同意します。

この日本語マニュアルの著作権は有限会社データ ダイナミクスが所有しております。
このマニュアルを有限会社データ ダイナミクス社の許可なく無断で複写、転載、翻訳することは禁じられております。また、有限会社データ ダイナミクス及び原文のマニュアルの発行もとである CCS 社、及び、有限会社データ ダイナミクスはそのマニュアルの内容及び、使用に於いて発生した如何なる物品、人体に対する障害や損害を保証するものではありません。

ライセンス・マテリアル著作権
1994, 2002 Custom Computer Services Incorporated
All Rights Reserved Worldwide
P.O.Box 2452
Brookfield, WI 53008, U.S.A.

有限会社 データダイナミクス
〒579-8062 東大阪市上六万寺町 13-10
TEL: 0729-81-6332 FAX: 0729-81-6085
http://www.datadynamics.co.jp
Info@datadynamics.co.jp