16 ビット言語ツール ライブラリ

マイクロチップ・テクノロジー社 (以下、マイクロチップ社) デバイスのプログラム保護機能に関して、以下の点にご注意ください。

- マイクロチップ社製品は、該当する「マイクロチップ社データシート」に記載の仕様を満たしています。
- マイクロチップ社では、通常の条件ならびに仕様どおりの方法で使用した場合、マイクロチップ社製品は現在市場に流通している同種製品としては最もセキュリティの高い部類に入る製品であると考えております。
- プログラム保護機能を解除するための不正かつ違法な方法が存在します。マイクロチップ社の確認している範囲では、このような方法のいずれにおいても、マイクロチップ社製品を「マイクロチップ社データシート」の動作仕様外の方法で使用する必要があります。このような行為は、知的所有権の侵害に該当する可能性が非常に高いと言えます。
- マイクロチップ社は、コードの保全について懸念を抱いているお客様と連携し、対応策に取り組んでいきます。
- マイクロチップ社を含むすべての半導体メーカーの中で、自社のコードのセキュリティを完全に保証できる企業はありません。プログラム保護機能とは、マイクロチップ社が製品を「解読不能」として保証しているものではありません。

プログラム保護機能は常に進歩しています。マイクロチップ社では、製品のプログラム保護機能の改善に継続的に取り組んでいます。マイクロチップ社のプログラム保護機能を解除しようとする行為は、デジタルミレニアム著作権法に抵触する可能性があります。そのような行為によってソフトウェアまたはその他の著作物に不正なアクセスを受けた場合は、デジタルミレニアム著作権法の定めるところにより損害賠償訴訟を起こす権利があります。

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

商標

再生紙を使用しています。

マイクロチップ社では、Chandler および Tempe (アリゾナ州)、Gresham (オレゴン州)、Mountain View (カリフォルニア州) の本部、設計部およびウエハ製造工場が ISO/TS-16949:2002 認証を取得しています。マイクロチップ社の品質システムプロセスおよび手順は、PIC® MCU および dsPIC® DSC、KEELOQ® コードホッピングデバイス、シリアル EEPROM、マイクロペリフェラル、不揮発性メモリ、アナログ製品に採用されています。また、マイクロチップ社の開発システムの設計および製造に関する品質システムは、ISO 9001:2000 の認証を受けています。

目次

16 ビット言語ツール ライブラリ

はじめに

顧客の皆様への注意

すべての文書には日付が記載されており、このマニュアルも例外ではありません。マイクロチップ社のツールと文書は顧客の皆様のニーズに応えるため日々進化を続けており、このマニュアル内のダイアログおよび / またはツールの説明も変更される可能性があります。最新のマニュアルは当社の Web サイト　www.microchip.com から入手してください。

マニュアルは "DS " 番号で識別されます。この番号は各ページ下のページ数の下欄に記載されています。DS 番号は "DSXXXXXA " となっており、"XXXXX XXXXX" は文書番号、"A" は改訂番号となっています。

開発ツールの最新情報は、MPLAB IDE のオンラインヘルプを参照してください。「ヘルプ」メニューを選択し、「トピックス」をクリックすると、入手可能なオンラインヘルプファイルのリストが開きます。

序論

このドキュメントは、GCC (GNU コンパイラコレクション) 技術を採用したマイクロチップ社の 16 ビット言語ツールに使うことができるライブラリを定義し、説明することを目的としています。次の言語ツールと関連します。

- MPLAB® ASM30 アセンブラ
- MPLAB C30 C コンパイラ
- MPLAB LINK30 リンカー
- MPLAB LIB30 アーカイバ / ライブラリアン
- その他のユーティリティ

本章で説明する内容は :

- 本ガイドについて
- 推奨文献
- トラブルシューティング
- マイクロチップ・ウエブサイト
- 開発システム変更の顧客通知サービス
- カスタマーサポート

本ガイドについて

ドキュメント構成

本ドキュメントでは、16 ビット・アプリケーションのコードを作成する際の GNU 言語ツールの使い方を説明します。ドキュメントは次のように構成されています。

- **第 1 章 : ライブラリの概要** — ライブラリの概要を説明します。
- **第 2 章 : DSP ライブラリ** — DSP 動作のライブラリ関数をリストアップします。
- **第 3 章 : 16 ビット・ペリフェラル・ライブラリ** — 16 ビット・デバイスのソフトウェアとハードウェア・ペリフェラル動作を対象とするライブラリ関数とマクロをリストアップします。
- **第 4 章 : 標準 C ライブラリ (算術関数付き)** — 標準 C 動作のライブラリ関数とマクロをリストアップします。
- **第 5 章 : MPLAB C30 組込関数** — C コンパイラの組込関数 MPLAB C30 をリストアップします。

本ガイドで使用する表記法

本マニュアルでは次のドキュメント表記法を使います。

ドキュメント表記法

表示形式	表示内容	使用例
明朝フォント:		
イタリック文字	参考文献	*MPLAB®IDE ユーザーズ・ガイド*
	強調テキスト	*.. はコンパイラのみ ...*
先頭が大文字	ウインドウ	Output ウインドウ
	ダイアログ	Settings ダイアログ
	メニュー選択	Enable Programmer を選択
引用	ウインドウまたはダイアログ内のフィールド名	"save project before build"
右かぎ括弧と下線付きイタリック・テキスト	メニューパス	*File>Save*
太文字	ダイアログボタン	**OK** をクリック
	タブ	**Power** タブをクリック
'*bnnnn*	バイナリ値、n は桁	'b00100, 'b10
かぎ括弧 <> で囲んだテキスト	キーボードのキー	<Enter>、<F1> を押します
クーリエ・フォント:		
通常クーリエ	サンプルソースコード	`#define START`
	ファイル名	`autoexec.bat`
	ファイルパス	`c:¥mcc18¥h`
	キーワード	`_asm, _endasm, static`
	コマンドライン・オプション	`-Opa+, -Opa-`
	ビット値	`0, 1`
イタリック・クーリエ	変数の引数	*`file`*`.o`、ここで、*`file`* は有効なファイル名
`0`*`xnnnn`*	16 進数、*`n`* は 16 進数桁	`0xFFFF, 0x007A`
角括弧 [ ]	オプションの引数	`mcc18 [options]` *`file`* `[options]`
中括弧とパイプ文字 : {|}	互いに排他的な引数の選択肢; OR 選択	`errorlevel {0|1}`
省略 ...	テキストの繰り返し	`var_name [, var_name...]`
	ユーザーが入力するコード	`void main (void) { ... }`

推奨文献

本ドキュメントでは、16 ビット・ライブラリ関数とマクロについて説明します。16 ビット言語ツールの詳細とその他のツールの使い方については、次の推奨文献を参照してください。

README ファイル

マイクロチップ・ツールの最新情報については、ソフトウェアに添付されている README ファイル (ASCII テキスト・ファイル) をご覧ください。

dsPIC® 言語ツールの入門 (DS70094)

16 ビット・デバイス用マイクロチップ言語ツール (MPLAB ASM30、MPLAB LINK30、MPLAB C30) のインストールと使い方を説明しています。16 ビット・シミュレータ MPLAB SIM30 の使用例も記載してあります。

MPLAB® ASM30、MPLAB® LINK30、ユーティリティのユーザーズ・ガイド (DS51317)

16 ビット・アセンブラ、MPLAB ASM30、16 ビット・リンカー、MPLAB LINK30、さらに MPLAB LIB30 アーカイバ / ライブラリアンなどの種々の 16 ビット・ユーティリティの使い方を説明しています。

MPLAB® C30 C コンパイラ・ユーザーズ・ガイド (DS51284)

16 ビット C コンパイラの使い方を説明しています。MPLAB LINK30 は本ツールと組み合わせて使います。

dsPIC30F ファミリの概要 (DS70043)

dsPIC30F デバイスとアーキテクチャの概要を説明しています。

dsPIC30F/33F プログラマズ・リファレンス・マニュアル (DS70157)

dsPIC30F/33F デバイスのプログラマズ・ガイドプログラマ・モデルと命令セットが記載してあります。

マイクロチップ・ウエブサイト

マイクロチップ・ウエブサイト (http://www. microchip.com) から多くのドキュメントを提供しています。Individual データシート、アプリケーション・ノート、チュートリアル、ユーザーズ・ガイドは、すべてダウンロードできます。ドキュメントはすべて Adobe Acrobat (PDF) です。

トラブルシューティング

本ドキュメントに記載していない一般的な問題については、`README` ファイルをご覧ください。

マイクロチップ・ウェブ・サイト

マイクロチップは WWW サイト www.microchip.com からオンライン・サポートを行っています。本ウェブ・サイトはファイルや情報をいち早くお客様に提供する手段として使っています。ご使用のブラウザでアクセスでき、ウェブ・サイトには下記情報が含まれます。

- **製品サポート**—データシートとエラッタ、アプリケーションノートとサンプルプログラム、設計リソース、ユーザーガイド、ハードウェア サポート文書、最新リリース ソフトウェア、保管ソフトウェア
- **一般的技術サポート**—頻繁な質問と回答 (FAQ)、技術支援要請、オンライン ディスカッション グループ、マイクロチップ コンサルタント プログラム メンバーリスト
- **マイクロチップのビジネス**—製品選択と注文ガイド、最新マイクロチップ プレスリリース、セミナとイベントのリスト、マイクロチップの営業オフィス、代理店、工場代理人のリスト

開発システム変更の顧客への通知サービス

マイクロチップは、お客様が最小の努力で現在のマイクロチップ製品について最新情報を入手できることをお手伝いできるように、顧客通知サービスを継続して行っています。一度ご登録いただければ、ご指定の製品ファミリもしくはご興味のある開発ツールに関して、変更、更新、改定もしくは正誤表が発行される毎に電子メールで通知を受けることができます。

登録するには、マイクロチップのウエブサイトにアクセスし、Customer Change Notification をクリックし、指示に従ってご登録ください。

開発システム製品は下記の通り分類されます。

- **コンパイラ**ーマイクロチップ C コンパイラとその他の言語ツールに関する最新情報で、以下を含みます。MPLAB C17, MPLAB C18 もしくは MPLAB C30 C コンパイラ ; MPASM™、MPLAB ASM30 アセンブラ ; MPLINK™、MPLAB LINK30 オブジェクトリンカー ; MPLIB™、MPLAB LIB30 オブジェクトライブラリアン。
- **エミュレータ**ーマイクロチップ・インサーキット・エミュレータの最新情報。これには MPLAB ICE 2000 と MPLAB ICE 4000 が含まれます。
- インサーキットデバッガーマイクロチップインサーキットデバッガ MPLABICD 2 の最新情報。
- **MPLAB IDE**—開発システム・ツール用 Windows® 統合開発環境であるマイクロチップ MPLAB™ IDE に関する最新情報です。MPLAB IDE、MPLAB SIM、MPLAB SIM30 シミュレータ、MPLAB IDE プロジェクト・マネージャ、全体的な編集機能とデバッグ機能を中心に説明しています。
- **プログラマ**—マイクロチップ社の書込器の最新情報。MPLAB PM3 デバイス・プログラマと PRO MATE® II デバイス・プログラマ、PICSTART® Plus 開発プログラマについて記載しています。

カスタマサポート

マイクロチップ製品のユーザーは、次のチャネルを介してサポートを受けることができます。

- 代理店または代理人
- 地域の営業オフィス
- フィールド アプリケーション エンジニア (FAE)
- 技術支援

サポートが必要な場合は、ディストリビュータ、販売特約店もしくは現地アプリケーションエンジニア (FAE) にお電話ください。地域販売オフィスでも顧客のサポートを行っています。販売店とその所在地については本ドキュメントの最後のページをご覧ください。

テクニカルサポートはウエブサイト http://support.microchip.com からご利用いただけます。

第 1 章 . ライブラリの概要

1.1 序論

ライブラリとは、参照とリンクを容易にするためにグループ化した関数の集まりです。ライブラリの作成と使い方の詳細については、*"MPLAB ASM30, MPLAB LINK30 and Utilities User's Guide"* (DS51317) を参照してください。

1.1.1 アセンブリ・コード・アプリケーション

16 ビット言語ツール・ライブラリの無償バージョンは、マイクロチップのウエブサイトから提供しています。DSP と 16 ビット・ペリフェラルのライブラリには、オブジェクト・ファイルとソース・コードが含まれています。標準 C ヘッダー・ファイル <math.h> の関数を含む算術ライブラリは、オブジェクト・ファイルとしてのみ提供しています。完全な標準 C ライブラリは MPLAB C30 C コンパイラと一緒に提供しています。

1.1.2 C コード・アプリケーション

16 ビット言語ツール・ライブラリは、c:¥Program Files¥Microchip¥MPLAB C30¥lib ディレクトリ内にあります。ここで、c:¥Program Files¥Microchip¥MPLAB C30 は、MPLAB C30 C コンパイラのインストール・ディレクトリです。これらは、MPLAB LINK 30 を使って直接アプリケーションにリンクすることができます。

1.1.3 本章の構成

本章は次のように構成されています。

- OMF 固有のライブラリ / スタートアップモジュール
- スタートアップ・コード
- DSP ライブラリ
- 16 ビット・ペリフェラル・ライブラリ
- 標準 C ライブラリ (Math 関数付き)
- MPLAB C30 組込関数

1.2 OMF 固有のライブラリ / スタートアップ・モジュール

ライブラリ・ファイルとスタートアップ・モジュールは OMF (オブジェクト・モジュール・フォーマット) 専用です。OMF は次のいずれかになっています。

- COFF—これがデフォルトです。
- ELF—ELF オブジェクト・ファイルに使われるデバッグ・フォーマットは DWARF 2.0 です。

OMF は次の 2 つの方法で選択できます。

1. PIC30_OMF という名前の環境変数を全ツールに対して設定する。
2. ツールを起動するときコマンドライン上で OMF を選択する。すなわち、-omf=*omf* または -momf=*omf*。

ユーザーのアプリケーション (非 OMF 仕様) をビルトする際に 16 ビット・ツールは一般ライブラリ・ファイルを探します。これらが見つからない場合、ツールは OMF 仕様を調べて、使用するライブラリ・ファイルを決定します。

一例として、libdsp.a が見つからず、かつ環境変数もコマンドライン・オプションも設定されていない場合、ファイル libdsp-coff.a がデフォルトとして使われます。

1.3 スタートアップ・コード

データ・メモリ内の変数を初期化するため、リンカーはデータ初期化テンプレートを生成します。このテンプレートをスタートアップ時に処理した後に、アプリケーションが制御を開始します。C プログラムの場合、この関数は libpic30-coff.a (crt0.o または crt1.o) または libpic30-elf.a (crt0.eo または crt1.eo) 内でスタートアップ・モジュールによって実行されます。アセンブリ言語プログラムは、スタートアップ・モジュール・ファイルと直接リンクしてこれらのモジュールを使います。スタートアップ・モジュールのソース・コードは、対応する .s ファイル内に用意されています。

プライマリ・スタートアップ・モジュール (crt0) は、固定データ・セクション内の変数を除くすべての変数を初期化します (イニシャライザのない変数は ANSI 標準に従ってゼロに設定)。もう 1 つのスタートアップ・モジュール (crt1) は、データの初期化を行いません。

スタートアップ・コードの詳細については、*"MPLAB ASM30, MPLAB LINK30 and Utilities User's Guide"* (DS51317) を、C アプリケーションについては、*"MPLAB C30 C Compiler User's Guide"*(DS51284) を、それぞれ参照してください。

1.4 DSP ライブラリ

DSP ライブラリ (libdsp-*omf*.a) は、dsPIC30F デジタル信号コントローラ (DSC) 上での実行を対象とするプログラムに対してデジタル信号処理動作のセットを提供します。DSP ライブラリは、合計 49 の関数をサポートしています。

1.5 16 ビット・ペリフェラル・ライブラリ

16 ビット (ソフトウェアとハードウェア) ペリフェラル・ライブラリは、16 ビット・ペリフェラルの設定と制御を行う関数とマクロを提供します。用例も本資料の関連する各章に示してあります。

これらのライブラリはプロセッサに固有であり、libp*Device*-*omf*.a の形式で表わされます。ここで、Device は 16 ビット・デバイス番号です (例えば、dsPIC30F6014 デバイスの場合 libp30F6014-coff.a となります)。

1.6 標準 C ライブラリ (算術関数付き)

ANSI-89 に準拠したライブラリの完全なセットを提供します。標準 C ライブラリ・ファイルは、libc-*omf*.a (業界のリーダである Dinkumware が作成) および libm-*omf*.a (マイクロチップ作成の算術関数) です。

さらに、いくつかの 16 ビット標準 C ライブラリ・ヘルパー関数と 16 ビット・デバイス用に変更する必要のある標準関数が libpic30-*omf*.a の中にあります。

一般的な C アプリケーションでは、3 つのライブラリすべてが必要です。

1.7 MPLAB C30 組込関数

MPLAB C30 C コンパイラには、開発者にとってライブラリ関数のように動作する組込関数が含まれています。

第 2 章 . DSP ライブラリ

2.1 序論

DSP ライブラリは、dsPIC30/F33F デジタル信号コントローラ上での実行を対象とするプログラムに対してデジタル信号処理演算のセットを提供します。このライブラリは、最も一般的な信号処理関数の効率良い組込み方法を C ソフトウェア開発者に手供するようにデザインされています。DSP ライブラリは、合計 52 の関数をサポートしています。

ライブラリの基本的な目標は、各関数の実行時間を短縮することです。DSP ライブラリはこの目標を実現するため、大部分が最適化されたアセンブリ言語で書かれています。DSP ライブラリを使用すると、ANSI C で書かれた等価なコードに比べて実行速度が大幅に向上します。さらに、DSP ライブラリは厳格にテストされているため、DSP ライブラリを使うと、アプリケーションの開発時間を短縮することができます。

2.1.1 アセンブリ・コード・アプリケーション

このライブラリの無償バージョンと対応するヘッダー・ファイルは、マイクロチップのウエブサイトから提供しています。ソース・コードも添付されています。

2.1.2 C コード・アプリケーション

MPLAB C30 C コンパイラのインストール・ディレクトリ (`c:¥programfiles¥microchip¥mplab c30`) には、ライブラリ関連ファイルの次のサブディレクトリが含まれています。

- `lib`—DSP ライブラリ / アーカイブ・ファイル
- `src¥dsp`—ライブラリ関数のソース・コードとライブラリを再ビルドするためのバッチ・ファイルのソース・コード
- `support ¥h`—DSP ライブラリのヘッダー・ファイル

2.1.3 本章の構成

本章は次のように構成されています。

- DSP ライブラリの使い方
- ベクタ関数
- ウインドウ関数
- 行列関数
- フィルタ関数
- 変換関数
- 制御関数
- その他の関数

2.2 DSP ライブラリの使い方

2.2.1 DSP ライブラリによるビルド

DSP ライブラリを使ったアプリケーションをビルドする際は、dsp.h と libdsp-*omf*.a の 2 つのファイルだけが必要です。dsp.h は、ライブラリで使用する関数プロトタイプ、#define、typedef のすべてを提供するヘッダー・ファイルです。libdsp-*omf*.a はアーカイブされたライブラリ・ファイルであり、個々のライブラリ関数の各オブジェクト・ファイルをすべて含んでいます (OMF-specific ライブラリの詳細については、**セクション 1.2「OMF 固有のライブラリ / スタートアップ・モジュール」**を参照してください)。

アプリケーションをコンパイルするときは、DSP ライブラリの関数をコールしているすべてのソース・ファイル、またはそのシンボルまたは typedef を使用しているすべてのソース・ファイルがこの dsp.h を参照します。アプリケーションをリンクするときは、libdsp-omf.a をリンカーに対する入力として使って (-- library または -l linker スイッチを使用)、アプリケーションで使われている関数がアプリケーションにリンクできるようにする必要があります。

リンカーは、DSP ライブラリの関数を .libdsp という名前の特別なテキスト・セクションに配置します。リンカーが生成する map ファイルを見ると、このことが判ります。

2.2.2 メモリ・モジュール

この DSP ライブラリは、最小のライブラリを生成する " スモール・コード " と " スモール・データ " のメモリ・モデルを使ってビルドしてあります。いくつかの DSP ライブラリ関数は C で書かれていて、コンパイラの浮動小数ライブラリを使っているため、.libm テキスト・セクションと .libdsp テキスト・セクションは、MPLAB C30 リンカーのスクリプト・ファイルにより、互いに隣り合うように配置されます。これにより、浮動小数ライブラリ内にある必要な浮動小数ルーチンをコールする RCALL 命令を DSP ライブラリが安全に使えるようになっています。

2.2.3 DSP ライブラリ関数のコール規則

DSP ライブラリ内の全オブジェクト・モジュールは dsPIC30F/33F DSC に対する C 互換性ガイドラインに準拠し、マイクロチップの "*MPLAB® C30 C コンパイラ・ユーザーズ・ガイド*" (DS51284) に記載する関数コール基準に従っています。特に、関数は最初の 8 個のワーキング・レジスタ (W0 ～ W7) を関数の引数として使います。その他の関数引数はスタックを介して渡されます。

ワーキング・レジスタ W0 ～ W7 はスクラッチ・メモリとして扱われ、これらの値は関数コールの後保存されません。一方、ワーキング・レジスタ W8 ～ W13 は関数によって使用される場合、ワーキング・レジスタは最初に待避され、レジスタの使用が終わった後に、関数が戻るときに元の値に復元されます。関数の戻り値 (非 void) は、ワーキング・レジスタ W0 (WREG とも呼ばれます) で使用可能です。必要に応じて、"*MPLAB® C30 コンパイラ・ユーザーズ・ガイド*" に記載する C システム・スタック規則に従ってランタイム・ソフトウェア・スタックが使用されます。これらのガイドラインに基づいて、DSP ライブラリのオブジェクト・モジュールは、C プログラム、アセンブリ・プログラム、または両言語のコードを使うプログラムへリンクすることができます。

2.2.4 データ型

DSP ライブラリが提供する演算は、dsPIC30F/33F DSC の DSP 命令セットとアーキテクチャ機能を利用するようにデザインされています。この意味で、大部分の動作は小数演算を使って計算されます。

DSP ライブラリでは、整数型を基に小数型を次のように定義しています :

```
#ifndef fractional
typedef int fractional;
#endif
```

fractional データ型は、1 ビットの符号と 15 ビットの小数を持つデータを表わすときに使います。このフォーマットを使うデータは、一般に "1.15" データと呼ばれています。

乗算器を使う関数の場合、結果は 40 ビットのアキュムレータを使って計算され、"9.31" 演算が使われます。このデータ・フォーマットは符号 / 振幅の 9 ビットと小数部の 31 ビットから構成されており、1.15 フォーマットの範囲 (-1.00 ～ ~+1.00) より演算範囲余裕が広くなっています。これらの関数の演算結果は、1.15 フォーマットの小数データ型に変換されます。

小数演算の使用により、特定の関数へ入力できる値のセットに制約が発生します。これらの制約を守ると、DSP ライブラリが提供する演算は、14 ビットの正しい数値を出力します。ただし、関数によっては入力データおよび / または出力結果に対して暗黙的なスケーリングを行うものもあり、これにより出力値の分解能が低下することがあります (浮動小数と比べて)。

DSP ライブラリ内の一部の演算で高い数値分解能を必要とするものは、浮動小数演算を使います。しかし、これらの演算の結果は、アプリケーションと統合するために小数値に変換されます。これに対する唯一の例外は MatrixInvert 関数であり、この関数は浮動小数行列の逆行列を浮動小数で計算して、結果を浮動小数フォーマットで出力します。

2.2.5 データ・メモリの使い方

DSP ライブラリでは RAM の割り当てを行わず、ユーザーがこれを行うことになっています。適切なメモリ量を割り当てず、かつデータを正しく揃えないと、関数の実行時に望ましくない結果が発生します。さらに、実行時間を短縮するため、関数の引数 (データ・メモリを指すポインタも含む) の有効 / 無効を DSP ライブラリはチェックしていません。DSP ライブラリ関数を使ったプロジェクト例を用意してありますので、これを参照して関数の正しい使い方を確認してください。MPLAB IDE を採用したプロジェクト / ワークスペース例が MPLAB C30 ツールスイートのインストレーション・フォルダに用意してあります。

大部分の関数はデータ・ポインタを関数の引数として受け取ります。このデータ・ポインタは演算対象となるデータと、さらに演算結果を保存するロケーションも含みます。使い易くするために、DSP ライブラリの大部分の関数では、入力引数がデフォルトの RAM メモリ空間 (X-Data または Y-Data) に配置されていること、さらに出力がデフォルトの RAM メモリ空間へ格納されることを想定しています。ただし、計算を多用する関数では、16 ビット・アーキテクチャのデュアル・データ・フェッチ機能を使用できるようにするため、いくつかのオペランドを X-Data と Y-Data (またはプログラム・メモリと Y-Data) に配置することが必要です。

2.2.6 CORCON レジスタの使い方

DSP ライブラリの多くの関数は、CORCON レジスタ値を変更することにより、dsPIC30F/33F デバイスを特別な動作モードに設定します。これらの関数が起動されると、CORCON レジスタがスタックにプッシュされます。次に、所望の動作を行うように変更され、最後に CORCON レジスタがスタックからポップされて元の値が回復されます。このメカニズムを使うと、CORCON の設定を変更することなく、正しくライブラリを実行することができます。

CORCON レジスタを変更するときは、このレジスタには一般に 0x00F0 が設定されます。この変更により、dsPIC30F/33F デバイスは次の動作モードになります :

- DSP 乗算では符号付き小数データを使用
- アキュムレータ A とアキュムレータ B でアキュムレータの飽和を許容

- 飽和モードを 9.31 飽和 (超飽和) に設定
- データ空間書き込み飽和をイネーブル
- プログラム空間可視化をディスエーブル
- 不偏向のまるめ処理 (バイアスなし) をイネーブル

CORCON レジスタとその機能の詳細については、"*dsPIC30F ファミリ・リファレンス・マニュアル*" (DS70046) を参照してください。

2.2.7 オーバーフローと飽和の処理

DSP ライブラリは大部分の計算を 9.31 飽和を使って行いますが、関数の出力を 1.15 フォーマットで格納する必要があります。演算中に、使用中のアキュムレータが飽和した場合 (0x7F FFFF FFFF を超えた場合または 0x80 0000 0000 を下回った場合)、STATUS レジスタ内の該当する飽和ビット (SA または SB) がセットされます。このビットはクリアされるまでセットされたままになります。この機能を使うと、関数の実行後に SA または SB を調べて、関数の入力データをスケールしたか否かを知ることができます。

同様に、アキュムレータを使った計算でオーバーフローが発生した場合 (アキュムレータが 0x00 7FFF FFFF を超えた場合または 0xFF 8000 0000 を下回った場合)、STATUS レジスタの該当するオーバーフロー・ビット (OA または OB) がセットされます。SA と SB の各ステータス・ビットとは異なり、OA と OB はクリアされるまでセット状態を維持しません。両ビットは、アキュムレータを使う演算が実行される毎に更新されます。この指定された範囲を超えることが重大なイベントである場合には、INTCON1 レジスタの OVATE ビット、OVBTE ビット、COVTE ビットを使ってアキュムレータ・オーバーフロー・トラップを許可することができます。これにより、オーバーフロー状態が発生すると直ちに Arithmetic Error トラップが発生することになり、これに対して必要なアクションをとることができます。

2.2.8 割り込みと RTOS の組み込み

DSP ライブラリは、割り込みまたは RTOS を使うアプリケーションに容易に組込むことができますが、ガイドラインに従う必要があります。実行時間を短縮するため、DSP ライブラリは DO ループ、REPEAT ループ、モジュロ・アドレッシング、ビット反転アドレッシングを使っています。これらの各コンポーネントは 16 ビット・デバイス上の有限なハードウェア・リソースであるため、バックグランド・コードが DSP ライブラリ関数の実行を阻害する場合には各リソースの使用について考慮する必要があります。

DSP ライブラリを組込むとき、各関数説明の関数プロファイルを調べて、使用しているリソースを知る必要があります。ライブラリ関数が割り込み可能な場合には、DO、REPEAT、特別なアドレッシング・ハードウェアの各状態を含み、関数が使用するすべてのレジスタの値の待避と復元はユーザーの責任で行う必要があります。また、CORCON 値とステータス・レジスタ値の待避と復元も当然含まれます。

2.2.9 DSP ライブラリの再ビルド

makedsplib.bat という名前のバッチ・ファイルが、DSP ライブラリの再ビルドのために提供されています。MPLAB C30 コンパイラは、DSP ライブラリを再ビルドすることを必要とし、バッチ・ファイルはデフォルト・ディレクトリ c:¥Program Files¥Microchip¥MPLAB C30¥ にインストールされていることを想定しています。言語ツールが別のディレクトリにインストールされている場合は、バッチ・ファイル内のディレクトリを変更して、言語ツールのロケーションに一致させる必要があります。

2.3 ベクタ関数

このセクションでは、DSP ライブラリで使う小数ベクタの概念を説明し、ベクタ演算を行う個々の関数について説明します。

2.3.1 小数ベクタ演算

小数ベクタとは、先頭要素を最下位メモリ・アドレスとしてメモリ内に連続して配置された数値 (ベクタ要素) の集まりを意味します。メモリの 1 ワード (2 バイト) を使って各要素の値を保存し、この数値は 1.15 データ・フォーマットで表わされた小数値と解釈されます。

ベクタの先頭要素をアドレス指定するポインタは、各ベクタ値に対するアクセスを提供するハンドルとして使用されます。先頭要素のアドレスは、ベクタのベース・アドレスと呼ばれます。ベクタの各要素は 16 ビットであるため、ベース・アドレスは偶数アドレスである必要があります。

ベクタの一次元配置はデバイスのメモリ・モデルに即し、N 個の要素を持つベクタの n 番目の要素はベクタのベース・アドレス BA から次のようにアクセスできるようになっています :

BA + 2(n - 1)、ただし 1 ≦ n ≦ N。

係数 2 は、16 ビット・デバイスのバイト・アドレッシング機能のために使われています。

単項と 2 項の小数ベクタ演算がこのライブラリに組み込まれています。単項演算のオペランド・ベクタは、ソース・ベクタと呼ばれます。2 項演算では、1 つ目のオペランドはソース 1 ベクタと呼ばれ、2 番目はソース 2 ベクタと呼ばれます。各演算では、計算をソース・ベクタの 1 個または複数の要素に対して行います。ある演算ではスカラー値 (1.15 フォーマットの小数値) が、他の演算ではベクタが、それぞれ結果として発生されます。結果もベクタの場合には、ディステネーション・ベクタと呼ばれます。

1 つのベクタを発生するいくつかの演算では、イン・プレイス計算が可能です。これは、演算結果がソース・ベクタ (2 項演算の場合はソース 1 ベクタ) に書き込まれることを意味します。この場合、ディステネーション・ベクタがソース・ベクタ (ソース 1 ベクタ) を (物理的に) 置き換えたと言われます。演算でイン・プレイス計算が可能な場合には、関数説明内のコメントに、その旨表示してあります。

いくつかの 2 項演算では、2 つのオペランドが同一のソース・ベクタ (物理的に 1 つ) である場合があります。これは、演算が同じソース・ベクタに対して行われることを意味します。与えられた演算に対してこのタイプの計算が可能な場合には、関数説明内のコメントにより、その旨表示されています。

演算によっては同一適用可能であり、かつイン・プレイス計算可能なものもあります。

このライブラリ内の小数ベクタ演算はすべて、引数としてオペランド・ベクタの次元数 (要素数) を受け取ります。この引数の値に基づいて、次のように仮定します :

a) 特定の演算に関係するすべてのベクタのサイズの総和は、ターゲット・デバイスの使用可能なデータ・メモリの範囲内にある
b) 2 項演算の場合、両オペランド・ベクタの次元数はベクタ代数の規則に従う (特に、`VectorConvolve` 関数と `VectorCorrelate` 関数の備考を参照)
c) ディステネーション・ベクタは、演算結果を受け取れるように十分大きい

2.3.2 ユーザーの考慮事項

a) これらの関数は境界チェックを行いません。範囲外の次元数 (長さゼロのベクタも含む) を使用したり、2 項演算で違法なソース・ベクタ・サイズを使用すると、予期しない結果が発生します。

b) ベクタの加算と減算では、ソース・ベクタ内の対応する複数の要素の総和が $1-2^{-15}$ を超える場合または -1.0 を下回る場合、飽和が発生します。同様に、ベクタのドット積と累乗でも、積の和が $1-2^{-15}$ を超える場合または -1.0 を下回る場合には、飽和が発生します。

c) 各関数コールが完了した後に、ステータス・レジスタ (SR) を調べることが推奨されます。特に、関数のリターン後に SA、SB、SAB の各フラグを調べると、飽和の有無を知ることができます。

d) 関数はすべて、デフォルトの RAM メモリ空間 (X-Data または Y-Data) に配置された小数ベクタに対して演算するようにデザインされています。

e) ディステネーション・ベクタを返す演算をネストすることができます。例えば、次のようになります :

a = Op1 (b, c)、ここで b = Op2 (d)、かつ c = Op3 (e, f) とすると、

a = Op1 (Op2 (d)、Op3 (e, f))

2.3.3 その他の注意

関数の説明では、演算の通常の使い方と見なされる範囲にその説明の適用範囲を限定しています。ただし、これら関数の計算中に境界のチェックを行っていないため、演算とその結果が特定の要求を満たすか否かの判断はユーザーに任されています。

例えば、VectorMax 関数の計算中に、ソース・ベクタの長さが numElems を超えることがあります。この場合、関数はソース・ベクタの最初の numElems 個の要素の中で最大値を求めることにしか使うことができません。

別の例としては、N 〜 N+numElems-1 の範囲に配置されているディステネーション・ベクタの numElems 個の要素を、M 〜 M+numElems-1 の範囲に配置されているソース・ベクタの numElems 個の要素で置き換えるとします。そうすると、VectorCopy 関数を次のように使うことができます :

```
fractional* dstV[DST_ELEMS] = {...};
fractional* srcV[SRC_ELEMS] = {...};
int n = NUM_ELEMS;
int N = N_PLACE;   /* NUM_ELEMS+N ≦ DST_ELEMS */
int M = M_PLACE;   /* NUM_ELEMS+M ≦ SRC_ELEMS */
fractional* dstVector = dstV+N;
fractional* srcVector = srcV+M;

dstVector = VectorCopy (n, dstVector, srcVector);
```

また、この例では、VectorZeroPad 関数は dstV = srcV となったため、イン・プレイス計算が可能です。numElems はソース・ベクタの先頭にある保持する要素数で、numZeros はベクタ後尾のゼロに設定される要素数です。

境界がチェックされていないことから、他の可能性を利用することもできます。

2.3.4 個別関数

以下に、ベクタ演算を行う個別関数について説明します。

VectorAdd

説明 : VectorAdd は、ソース 1 ベクタ内の各要素の値をソース 2 ベクタ内の対応する各要素の値に加算して、結果をディステネーション・ベクタに格納します。

インクルード : dsp.h

プロトタイプ :

```
extern fractional* VectorAdd (
    int numElems,
    fractional* dstV,
    fractional* srcV1,
    fractional* srcV2
);
```

引数 :

numElems	ソース・ベクタ内の要素数
dstV	ディステネーション・ベクタを指すポインタ
srcV1	ソース 1 ベクタを指すポインタ
srcV2	ソース 2 ベクタを指すポインタ

戻り値 : ディステネーション・ベクタのベース・アドレスを指すポインタ

備考 : *srcV1*[n] + *srcV2*[n] の絶対値が $1\text{-}2^{-15}$ より大きい場合、この演算は n 番目の要素に対して飽和します。
この関数はイン・プレイス計算が可能です。
この関数は同一適用可能です。

ソース・ファイル : vadd.s

関数プロファイル :

システム・リソースの使用 :

W0..W4	使用、復旧なし
ACCA	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
13

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
17 + 3(*numElems*)

VectorConvolve

説明 : VectorConvolve は、2 つのソース・ベクタについてコンボリューションを計算して、結果をディステネーション・ベクタへ格納します、結果は次のように計算されます :

$$y(n) = \sum_{k=0}^{n} x(k)h(n-k), \text{ for } 0 \leqq n < M$$

$$y(n) = \sum_{k=n-M+1}^{n} x(k)h(n-k), \text{ for } M \leqq n < N$$

$$y(n) = \sum_{k=n-M+1}^{N-1} x(k)h(n-k), \text{ for } N \leqq n < N+M-1$$

ここで、x(k) = サイズ N のソース 1 ベクタ、h(k) = サイズ M のソース 2 ベクタ (M ≦ N)。

インクルード : dsp.h

プロトタイプ :

```
extern fractional* VectorConvolve (
   int numElems1,
   int numElems2,
   fractional* dstV,
   fractional* srcV1,
   fractional* srcV2
);
```

引数 :

numElems1	ソース 1 ベクタ内の要素数
numElems2	ソース 2 ベクタ内の要素数
dstV	ディステネーション・ベクタを指すポインタ
srcV1	ソース 1 ベクタを指すポインタ
srcV2	ソース 2 ベクタを指すポインタ

戻り値 : ディステネーション・ベクタのベース・アドレスを指すポインタ

備考 : ソース 2 ベクタ内の要素数は、ソース 1 ベクタ内の要素数以下である必要があります。
ディステネーション・ベクタは、*numElems1*+*numElems2*-1 の要素数で既に存在している必要があります。
この関数は同一適用可能です。

ソース・ファイル : vcon.s

VectorConvolve (続き)

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W10	待避、使用、復旧
ACCA	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 2 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
58

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
For N = *numElems1*, and M = *numElems2*,

$$28 + 13M + 6\sum_{m=1}^{M} m + (N - M)(7 + 3M), \text{ for } M \neq N$$

$$28 + 13M + 6\sum_{m=1}^{M} m, \text{ for } M = N$$

VectorCopy

説明 : VectorCopy は、ソース・ベクタの各要素を (既に存在している) ディステネーション・ベクタの先頭にコピーして次のようにします :
dstV[n] = *srcV*[n], 0 ≦ n < *numElems*

インクルード : dsp.h

プロトタイプ :

```
extern fractional* VectorCopy (
    int numElems,
    fractional* dstV,
    fractional* srcV
);
```

引数 :

numElems	ソース・ベクタ内の要素数
dstV	ディステネーション・ベクタを指すポインタ
srcV	ソース・ベクタを指すポインタ

戻り値 : ディステネーション・ベクタのベース・アドレスを指すポインタ

備考 : ディステネーション・ベクタは存在している*必要があります*。ディステネーション・ベクタは、*numElems* 個以上の要素を持つ*必要があります*。
この関数はイン・プレイス計算が可能です。この動作モードについては、このセクションの終わりにある「その他の注意」も参照してください。

ソース・ファイル : vcopy.s

VectorCopy (続き)

関数プロファイル : システム・リソースの使用 :
W0..W3 使用、復旧なし

DO 命令と REPEAT 命令の使用 :
DO 命令 : なし
REPEAT 命令 : 1 レベル

プログラム・ワード数 (24 ビット命令):
6

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
12 + *numElems*

VectorCorrelate

説明 : VectorCorrelate は、2 つのソース・ベクタの間の相関を計算して、結果をディステネーション・ベクタへ格納します、結果は次のように計算されます :

$$r(n) = \sum_{k=0}^{N-1} x(k)y(k+n), \text{ for } 0 \leqq n < N+M-1$$

ここで、x(k) = サイズ N のソース 1 ベクタ、y(k) = サイズ M のソース 2 ベクタ (M ≦ N)。

インクルード : dsp.h

プロトタイプ :

```
extern fractional* VectorCorrelate (
    int numElems1,
    int numElems2,
    fractional* dstV,
    fractional* srcV1,
    fractional* srcV2
);
```

引数 :

numElems1	ソース 1 ベクタ内の要素数
numElems2	ソース 2 ベクタ内の要素数
dstV	ディステネーション・ベクタを指すポインタ
srcV1	ソース 1 ベクタを指すポインタ
srcV2	ソース 2 ベクタを指すポインタ

戻り値 : ディステネーション・ベクタのベース・アドレスを指すポインタ

備考 : ソース 2 ベクタ内の要素数は、ソース 1 ベクタ内の要素数以下である *必要があります*。
ディステネーション・ベクタは、*numElems1*+*numElems2*-1 の要素数で既に存在している *必要があります*。
この関数は同一適用可能です。
この関数は VectorConvolve を使います。

ソース・ファイル : vcor.s.s

VectorCorrelate (続き)

関数プロファイル : システム・リソースの使用 :
W0..W7 使用、復旧なし ,
さらに VectorConvolve からのリソース

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : なし ,
さらに、VectorConvolve からの DO/REPEAT 命令

プログラム・ワード数 (24 ビット命令):
14,
さらに、VectorConvolve からのプログラム・ワード

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
19 + floor(M / 2) * 3、M = *numElems2*,
さらに、VectorConvolve のサイクル数

注 : VectorConvolve の説明で、報告されるサイクル数には 4 サイクルの C 関数コールのオーバーヘッドが含まれます。したがって、VectorConvolve から VectorCorrelate へ加算される実際のサイクル数は、VectorConvolve 単体に対して報告される数値より小さい 4 になります。

VectorDotProduct

説明 : VectorDotProduct は、ソース 1 ベクタとソース 2 ベクタの各対応する要素間で積とり、各積の総和を計算します。

インクルード : dsp.h

プロトタイプ :
```
extern fractional VectorDotProduct (
    int numElems,
    fractional* srcV1,
    fractional* srcV2
);
```

引数 :

numElems	ソース・ベクタ内の要素数
srcV1	ソース 1 ベクタを指すポインタ
srcV2	ソース 2 ベクタを指すポインタ

戻り値 : 各積の総和。

備考 : 各積の総和が $1-2^{-15}$ より大きい場合、この演算は飽和します。
この関数は同一適用可能です。

ソース・ファイル : vdot.s

VectorDotProduct (続き)

関数プロファイル : システム・リソースの使用 :

W0..W2	使用、復旧なし
W4..W5	使用、復旧なし
ACCA	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
13

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
17 + 3(*numElems*)

VectorMax

説明 : VectorMax は、ソース・ベクタ内で前のベクタ要素の値以上の値を持つ最後の要素を求めます。したがって、最大値と最大要素のインデックスを出力します。

インクルード : dsp.h

プロトタイプ :

```
extern fractional VectorMax (
    int numElems,
    fractional* srcV,
    int* maxIndex
);
```

引数 :

numElems	ソース・ベクタ内の要素数
srcV	ソース・ベクタを指すポインタ
maxIndex	最大 (最終) 要素のインデックスのホルダーを指すポインタ

戻り値 : ベクタ内の最大値。

備考 : If *srcV*[i] = *srcV*[j] = maxVal, and i < j, then**maxIndex* = j.

ソース・ファイル : vmax.s

関数プロファイル : システム・リソースの使用 :

W0..W5	使用、復旧なし

DO 命令と REPEAT 命令の使用 :
DO 命令 : なし
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
13

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
14
if *numElems* = 1
20 + 8(*numElems* - 2)
if *srcV*[n] ≦ *srcV*[n + 1], 0 ≦ n < *numElems* - 1
19 + 7(*numElems* - 2)
if *srcV*[n] > *srcV*[n + 1], 0 ≦ n < *numElems* - 1

VectorMin

説明 : VectorMin は、ソース・ベクタ内で前のベクタ要素の値以下の値を持つ最後の要素を求めます。したがって、最小値と最小要素のインデックスを出力します。

インクルード : dsp.h

プロトタイプ :
```
extern fractional VectorMin (
    int numElems,
    fractional* srcV,
    int* minIndex
);
```

引数 :

numElems	ソース・ベクタ内の要素数
srcV	ソース・ベクタを指すポインタ
minIndex	最小 (最終) 要素のインデックスのホルダーを指すポインタ

戻り値 : ベクタ内の最小値。

備考 : If *srcV*[i] = *srcV*[j] = minVal, and i < j, then **minIndex* = j.

ソース・ファイル : vmin.s

関数プロファイル : システム・リソースの使用 :

W0..W5 使用、復旧なし

DO 命令と REPEAT 命令の使用 :

DO 命令 : なし

REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):

13

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):

14

if *numElems* = 1

20 + 8(*numElems* - 2)

if *srcV*[n] ≧ *srcV*[n + 1], 0 ≦ n < *numElems* - 1

19 + 7(*numElems* - 2)

if *srcV*[n] < *srcV*[n + 1], 0 ≦ n < *numElems* - 1

VectorMultiply

説明 : VectorMultiply は、ソース 1 ベクタ内の各要素値とソース 2 ベクタ内の対応する各要素値の積をとり、結果をディスステネーション・ベクタの対応する各要素に格納します。

インクルード : dsp.h

プロトタイプ :
```
extern fractional* VectorMultiply (
    int numElems,
    fractional* dstV,
    fractional* srcV1,
    fractional* srcV2
);
```

引数 :

numElems	ソース・ベクタ内の要素数
dstV	ディステネーション・ベクタを指すポインタ
srcV1	ソース 1 ベクタを指すポインタ
srcV2	ソース 2 ベクタを指すポインタ

戻り値 : ディステネーション・ベクタのベース・アドレスを指すポインタ

備考 : この演算は、ベクタ要素毎の乗算と呼ばれています。
この関数はイン・プレイス計算が可能です。
この関数は同一適用可能です。

VectorMultiply (続き)

ソース・ファイル : vmul.s

関数プロファイル : システム・リソースの使用 :

W0..W5	使用、復旧なし
ACCA	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
14

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
17 + 4(*numElems*)

VectorNegate

説明 : VectorNegate は、ソース・ベクタ内の各要素の値を反転して (すなわち符号を変えて)、結果をディステネーション・ベクタへ格納します。

インクルード : dsp.h

プロトタイプ :

```
extern fractional* VectorNeg (
    int numElems,
    fractional* dstV,
    fractional* srcV
);
```

引数 :

numElems	ソース・ベクタ内の要素数
dstV	ディステネーション・ベクタを指すポインタ
srcV	ソース・ベクタを指すポインタ

戻り値 : ディステネーション・ベクタのベース・アドレスを指すポインタ

備考 : 値 0x8000 の反転は 0x7FFF。
この関数はイン・プレイス計算が可能です。

ソース・ファイル : vneg.s

関数プロファイル : システム・リソースの使用 :

W0..W5	使用、復旧なし
ACCA	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
16

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
19 + 4(*numElems*)

VectorPower

説明 : VectorPower は、ソース・ベクタの累乗を各要素の 2 乗の和として計算します。

インクルード : dsp.h

プロトタイプ :
```
extern fractional VectorPower (
    int numElems,
    fractional* srcV
);
```

引数 :

numElems	ソース・ベクタ内の要素数
srcV	ソース・ベクタを指すポインタ

戻り値 : ベクタの累乗値 (2 乗の和)。

備考 : 2 乗の和の絶対値が $1\text{-}2^{-15}$ より大きい場合、この演算は飽和します。この関数は同一適用可能です。

ソース・ファイル : vpow.s

関数プロファイル : システム・リソースの使用 :

W0..W2	使用、復旧なし
W4	使用、復旧なし
ACCA	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : なし
REPEAT 命令 : 1 レベル

プログラム・ワード数 (24 ビット命令):
12

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
16 + 2(*numElems*)

VectorScale

説明 : VectorScale は、ソース・ベクタ内のすべての要素値にスケール値を乗算し、結果をディステネーション・ベクタに格納します。

インクルード : dsp.h

プロトタイプ :
```
extern fractional* VectorScale (
    int numElems,
    fractional* dstV,
    fractional* srcV,
    fractional sclVal
);
```

引数 :

numElems	ソース・ベクタ内の要素数
dstV	ディステネーション・ベクタを指すポインタ
srcV	ソース・ベクタを指すポインタ
sclVal	ベクタ要素をスケールする値

戻り値 : ディステネーション・ベクタのベース・アドレスを指すポインタ

備考 : sclVal は、1.15 フォーマットの小数値である必要があります。この関数はイン・プレイス計算が可能です。

ソース・ファイル : vscl.s

VectorScale (続き)

関数プロファイル : システム・リソースの使用 :

W0..W5	使用、復旧なし
ACCA	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
14

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
18 + 3(*numElems*)

VectorSubtract

説明 : VectorSubtract は、ソース 2 ベクタ内の各要素の値をソース 1 ベクタ内の対応する各要素の値から減算して、結果をディステネーション・ベクタに格納します。

インクルード : dsp.h

プロトタイプ :

```
extern fractional* VectorSubtract (
    int numElems,
    fractional* dstV,
    fractional* srcV1,
    fractional* srcV2
);
```

引数 :

numElems	ソース・ベクタ内の要素数
dstV	ディステネーション・ベクタを指すポインタ
srcV1	ソース 1 ベクタを指すポインタ (被減数)
srcV2	ソース 2 ベクタを指すポインタ (減数)

戻り値 : ディステネーション・ベクタのベース・アドレスを指すポインタ

備考 : *srcV1*[n] - *srcV2*[n] の絶対値が $1\text{-}2^{-15}$ より大きい場合、この演算は n 番目の要素に対して飽和します。
この関数はイン・プレイス計算が可能です。
この関数は同一適用可能です。

ソース・ファイル : vsub.s

関数プロファイル : システム・リソースの使用 :

W0..W4	使用、復旧なし
ACCA	使用、復旧なし
ACCB	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
14

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
17 + 4(*numElems*)

VectorZeroPad

説明 : VectorZeroPad は、ソース・ベクタを (既に存在している) ディステネーション・ベクタの先頭にコピーし、次のようにディステネーション・ベクタの残りの numZeros 要素にゼロを書き込みます :

dstV[n] = *srcV*[n], 0 ≦ n < *numElems*

dstV[n] = 0, *numElems* ≦ *nnumElems*+*numZeros*

インクルード : dsp.h

プロトタイプ :

```
extern fractional* VectorZeroPad (
   int numElems,
   int numZeros,
   fractional* dstV,
   fractional* srcV
);
```

引数 :

numElems	ソース・ベクタ内の要素数
numZeros	ディステネーション・ベクタの終わりでゼロを書き込む要素数
dstV	ディステネーション・ベクタを指すポインタ
srcV	ソース・ベクタを指すポインタ

戻り値 : ディステネーション・ベクタのベース・アドレスを指すポインタ

備考 : ディステネーション・ベクタは、*numElems* + *numZeros* の要素数で既に存在している必要があります。
この関数はイン・プレイス計算が可能です。この動作モードについては、このセクションの始めにある「その他の注意」も参照してください。
この関数は、VectorCopy を使います。

ソース・ファイル : vzpad.s

関数プロファイル : システム・リソースの使用 :

W0..W6 使用、復旧なし
さらに VectorCopy からのリソース

DO 命令と REPEAT 命令の使用 :
DO 命令 : なし
REPEAT 命令 : 1 レベル
さらに、VectorCopy からの DO/REPEAT 命令

プログラム・ワード数 (24 ビット命令):
13,
さらに、VectorCopy からのプログラム・ワード

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
18 + *numZeros*
さらに、VectorCopy のサイクル数

注 : VectorCopy の説明で、報告されるサイクル数には 3 サイクルの C 関数コール・オーバーヘッドが含まれます。したがって、VectorCopy から VectorCorrelate へ加算される実際のサイクル数は、VectorCopy 単体に対して報告される数値より小さい 3 になります。

2.4 ウインドウ関数

ウインドウとは、領域内 (0 ≦ n ＜ *numElems*) で特定の値分布を持つベクタを意味します。特定の値分布は、生成されるウインドウの特性に依存します。

ベクタが与えられたとき、その値分布はウインドウを適用して変更されます。この場合、ウインドウは変更されるベクタと同じ要素数を持つ*必要があります*。

ベクタをウインドウ化する前に、ウインドウをクリアしておく必要があります。ウインドウの初期化演算が提供されています。この演算は、ウインドウ要素の値を発生します。数値精度を高くする場合、これらの値は浮動小数演算で計算され、結果は 1.15 フォーマットの小数で格納されます。

ウインドウ演算を適用する際に余分なオーバーヘッドをなくするため、特定のウインドウを 1 回だけ生成して、プログラムの実行時に何回も使用します。したがって、初期化演算から返されたウインドウを固定 (静的) ベクタとして保存することをお薦めします。

2.4.1 ユーザーの考慮事項

a) ウインドウ初期化関数はすべて、デフォルトの RAM メモリ空間 (X-Data または Y-Data) 内に割り当てられたウインドウ・ベクタを生成するようにデザインされています。
b) ウインドウ化関数は、デフォルトの RAM メモリ空間 (X-Data または Y-Data) 内に割り当てられたベクタに対して演算するようにデザインされています。
c) 各関数コールが完了した後に、ステータス・レジスタ (SR) を調べることが推奨されます。
d) ウインドウ初期化関数は C を使って実装されているため、最新サイクル・カウント情報のリリースに含まれている電子ドキュメントを参照してください。

2.4.2 個別関数

以下に、ウインドウ演算を行う個別関数について説明します。

BartlettInit

説明 : `BartlettInit` は、長さ *numElems* の Barlett ウインドウを初期化します。

インクルード : `dsp.h`

プロトタイプ :
```
extern fractional* BartlettInit (
   int numElems,
   fractional* window
);
```

引数 :
numElems ウインドウ内の要素数
window 初期化するウインドウを指すポインタ

戻り値 : 初期化するウインドウのベース・アドレスを指すポインタ

備考 : ウインドウ・ベクタは、*numElems* の要素数で既に存在している必要があります。

ソース・ファイル : `initbart.c`

BartlettInit (続き)

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W14	待避、使用、復旧なし

DO 命令と REPEAT 命令の使用 :
なし

プログラム・ワード数 (24 ビット命令):
詳細については、"Readme for dsPIC Language Tools Libraries.txt" を参照してください。

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
詳細については、"Readme for dsPIC Language Tools Libraries.txt" を参照してください。

BlackmanInit

説明 : BlackmanInit は、長さ *numElems* の Blackman (3 項) ウインドウを初期化します。

インクルード : dsp.h

プロトタイプ :

```
extern fractional* BlackmanInit (
    int numElems,
    fractional* window
);
```

引数 :

numElems	ウインドウ内の要素数
window	初期化するウインドウを指すポインタ

戻り値 : 初期化するウインドウのベース・アドレスを指すポインタ

備考 : ウインドウ・ベクタは、*numElems* の要素数で既に存在している必要があります。

ソース・ファイル : initblck.c

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W14	待避、使用、復旧なし

DO 命令と REPEAT 命令の使用 :
なし

プログラム・ワード数 (24 ビット命令):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

HammingInit

説明 : HammingInit は、長さ *numElems* の Hamming ウインドウを初期化します。

インクルード : dsp.h

プロトタイプ :
```
extern fractional* HammingInit (
   int numElems,
   fractional* window
);
```

引数 :
numElems ウインドウ内の要素数
window 初期化するウインドウを指すポインタ

戻り値 : 初期化するウインドウのベース・アドレスを指すポインタ

備考 : ウインドウ・ベクタは、*numElems* の要素数で既に存在している*必要があります。*

ソース・ファイル : inithamm.c

関数プロファイル : システム・リソースの使用 :
W0..W7 使用、復旧なし
W8..W14 待避、使用、復旧なし

DO 命令と REPEAT 命令の使用 :
なし

プログラム・ワード数 (24 ビット命令):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

HanningInit

説明 : HanningInit は、長さ *numElems* の Hanning ウインドウを初期化します。

インクルード : dsp.h

プロトタイプ :
```
extern fractional* HanningInit (
   int numElems,
   fractional* window
);
```

引数 :
numElems ウインドウ内の要素数
window 初期化するウインドウを指すポインタ

戻り値 : 初期化するウインドウのベース・アドレスを指すポインタ

備考 : ウインドウ・ベクタは、*numElems* の要素数で既に存在している*必要があります。*

ソース・ファイル : inithann.c

HanningInit (続き)

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W14	待避、使用、復旧なし

DO 命令と REPEAT 命令の使用 :
なし

プログラム・ワード数 (24 ビット命令):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

KaiserInit

説明 : KaiserInit は、引数 *betaVal* と長さ *numElems* で決定される形状を持つ Kaiser ウインドウを初期化します。

インクルード : dsp.h

プロトタイプ :

```
extern fractional* KaiserInit (
    int numElems,
    fractional* window,
    float betaVal
);
```

引数 :

numElems	ウインドウ内の要素数
window	初期化するウインドウを指すポインタ
betaVal	ウインドウ形状を指定するパラメータ

戻り値 : 初期化するウインドウのベース・アドレスを指すポインタ

備考 : ウインドウ・ベクタは、*numElems* の要素数で既に存在している*必要があります*。

ソース・ファイル : initkais.c

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W14	待避、使用、復旧なし

DO 命令と REPEAT 命令の使用 :
なし

プログラム・ワード数 (24 ビット命令):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

VectorWindow

説明 : VectorWindow は、ウインドウを指定されたソース・ベクタに適用し、ウインドウ化された結果のベクタをディステネーション・ベクタへ格納します。

インクルード : dsp.h

プロトタイプ :

```
extern fractional* VectorWindow (
    int numElems,
    fractional* dstV,
    fractional* srcV,
    fractional* window
);
```

引数 :

numElems	ソース・ベクタ内の要素数
dstV	ディステネーション・ベクタを指すポインタ
srcV	ソース・ベクタを指すポインタ
window	初期化されたウインドウを指すポインタ

戻り値 : ディステネーション・ベクタのベース・アドレスを指すポインタ

備考 : ウインドウ・ベクタは初期化済みで、*numElems* の要素数で既に存在している*必要があります*。
この関数はイン・プレイス計算が可能です。
この関数は同一適用可能です。
この関数は VectorMultiply を使います。

ソース・ファイル : dowindow.s

関数プロファイル : システム・リソースの使用 :
VectorMultiply からのリソース

DO 命令と REPEAT 命令の使用 :
DO 命令 : なし
REPEAT 命令 : なし ,
さらに、VectorMultiply からの DO/REPEAT

プログラム・ワード数 (24 ビット命令):
3,
さらに、VectorMultiply からのプログラム・ワード

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
9,
さらに、VectorMultiply のサイクル数

注 : VectorMultiply の説明で、報告されるサイクル数には 3 サイクルの C 関数コール・オーバーヘッドが含まれます。したがって、VectorMultiply から VectorWindow へ加算される実際のサイクル数は、VectorMultiply 単体に対して報告される数値より小さい 3 になります。

2.5 行列関数

このセクションでは、DSP ライブラリで使う小数行列の概念を説明し、行列演算を行う個々の関数について説明します。

2.5.1 小数行列演算

小数行列とは、先頭要素を最下位メモリ・アドレスとしてメモリ内に連続して配置された数値 (行列要素) の集まりを意味します。メモリの 1 ワード (2 バイト) を使って各要素の値を保存し、この数値は 1.15 データ・フォーマットで表わされた小数値と解釈されます。

行列の先頭要素をアドレス指定するポインタは、各行列値に対するアクセスを提供するハンドルとして使用されます。先頭要素のアドレスは、行列のベース・アドレスと呼ばれます。行列の各要素は 16 ビットであるため、ベース・アドレスは偶数アドレスである必要があります。

行列の 2 次元配置は、メモリ領域内では行毎に要素を配列してエミュレートします。したがって、メモリ内の先頭値は先頭行の先頭要素になります。先頭行の残りの要素がこの後に続きます。その後に、2 行目の要素が格納され、すべての行がメモリに収まるまで以下同様に続きます。この方法では、R 個の行と C 個の列から構成されている行列の r 行 c 列要素の位置は、行列ベース・アドレス BA から次のように計算されます :

$BA + 2(C(r - 1) + c - 1)$、ただし $1 \leqq r \leqq R$、$1 \leqq c \leqq C$。

係数 2 は、16 ビット・デバイスのバイト・アドレッシング機能のために使われていることに注意してください。

単項と 2 項の小数行列演算がこのライブラリに組み込まれています。単項演算のオペランド行列は、ソース行列と呼ばれます。2 項演算では、1 つ目のオペランドはソース 1 行列と呼ばれ、2 番目はソース 2 行列と呼ばれます。各演算は、ある計算をソース行列の 1 個または複数の要素に対して行います。演算結果は行列となり、ディステネーション行列と呼ばれます。

1 つの行列を発生するいくつかの演算では、イン・プレイス計算が可能です。これは、演算から得られる各結果がソース行列 (2 項演算の場合はソース 1 行列) に書き込まれることを意味します。この場合、ディステネーション行列がソース行列 (ソース 1 行列) を (物理的に) 置き換えたと言われます。演算でイン・プレイス計算が可能な場合には、関数説明内のコメントに、その旨表示してあります。

2 項演算によっては、2 つのオペランドが同一のソース行列 (物理的に 1 つ) である場合があります。これは、演算が同一のソース行列に対して行われることを意味します。与えられた演算に対してこのタイプの計算が可能な場合には、関数説明内のコメントにより、その旨表示されています。

演算によっては同一適用可能であり、かつイン・プレイス計算可能なものもあります。

このライブラリ内の小数行列演算はすべて、引数としてオペランド行列の行数と列数を受け取ります。これらの引数の値に基づいて、次のように仮定します :

a) 特定の演算に関係するすべての行列のサイズの総和は、ターゲット・デバイスの使用可能なデータ・メモリの範囲内にある。

b) 2 項演算の場合、オペランド行列の行数と列数は、ベクタ代数の規則に従う。すなわち、行列の加算と減算の場合は、2 つの行列は同じ行数と列数を持ち、行列の乗算の場合は、最初のオペランドの列数は 2 つ目のオペランドの行数に一致する必要があります。逆行列演算でのソース行列は正方行列 (行数と列数が一致) であり、かつ非特異行列 (行列式が非ゼロ) である必要があります。

c) ディステネーション行列は、演算結果を受け取れるように十分大きい。

2.5.2 ユーザーの考慮事項

a) これらの関数は境界チェックを行いません。範囲外の次元数 (ゼロ行および / またはゼロ列の行列も含む) を使用したり、2 項演算で違法なソース行列サイズを使用すると、予期しない結果が発生します。

b) 行列の加算と減算では、ソース行列内の対応する複数の要素の総和が $1-2^{-15}$ を超える場合または -1 を下回る場合、飽和が発生します。

c) 行列の乗算では、対応する行と列との各積の和が $1-2^{-15}$ を超える場合または -1 を下回る場合、飽和が発生します。

d) 各関数コールが完了した後に、STATUS レジスタ (SR) を調べることが推奨されます。特に、関数のリターン後に SA、SB、SAB の各フラグを調べると、飽和の有無を知ることができます。

e) 関数はすべて、デフォルトの RAM メモリ空間 (X-Data または Y-Data) に配置された小数行列に対して演算するようにデザインされています。

f) ディステネーション行列を返す演算をネストすることができます。例えば、次のようになります :

a = Op1 (b, c)、ここで b = Op2 (d)、かつ c = Op3 (e, f) とすると、

a = Op1 (Op2 (d)、Op3 (e, f))

2.5.3 その他の注意

関数の説明では、演算の通常の使い方と見なされる範囲にその説明の適用範囲を限定しています。ただし、これら関数の計算中に境界のチェックを行っていないため、演算とその結果が特定の要求を満たすか否かの判断はユーザーに任されています。

例えば、`MatrixMultiply` 関数の計算中に、関係する行列の次元は必ずしも、ソース 1 行列に対して {*numRows1, numCos1Rows2*} に、ソース 2 行列に対して {*numCols1Rows2, numCols2*} に、ディステネーション行列に対して {*numRows1, numCols2*} に、それぞれなるとは限りません。実際に必要なことは、計算中にポインタがメモリ範囲を超えないようにするため、それぞれのサイズを十分大きくすることです。

もう一つの例は、次元 {*numRows, numCols*} のソース行列を転置する場合、ディステネーション行列の次元は {*numCols, numRows*} になります。したがって、ソース行列が正方行列である場合*のみ*、演算はイン・プレイス計算可能です。それでも、非正方行列に対する演算は正常にイン・プレイス計算で適用できます。注意すべきことは、次元の*暗黙的な*変更です。

境界がチェックされていないことから、他の可能性を利用することもできます。

2.5.4 個別関数

以下に、行列演算を行う個別関数について説明します。

MatrixAdd

説明 : MatrixAdd は、ソース 1 行列内の各要素の値をソース 2 行列内の対応する各要素の値に加算して、結果をディステネーション行列に格納します。

インクルード : dsp.h

プロトタイプ :

```
extern fractional* MatrixAdd (
    int numRows,
    int numCols,
    fractional* dstM,
    fractional* srcM1,
    fractional* srcM2
);
```

引数 :

numRows	ソース行列内の行数
numCols	*ソース行列内の列数*
dstM	ディステネーション行列を指すポインタ
srcM1	ソース 1 行列を指すポインタ
srcM2	ソース 2 行列を指すポインタ

戻り値 : ディステネーション行列のベース・アドレスを指すポインタ

備考 : *srcM1*[r][c]+*srcM2*[r][c] の絶対値が $1\text{-}2^{-15}$ より大きい場合、この演算は (r,c) 要素に対して飽和します。
この関数はイン・プレイス計算が可能です。
この関数は同一適用可能です。

ソース・ファイル : madd.s

関数プロファイル :

システム・リソースの使用 :

W0..W4	使用、復旧なし
ACCA	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
14

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
20 + 3(*numRows * numCols*)

MatrixMultiply

説明 : MatrixMultiply はソース 1 行列とソース 2 行列との間で行列乗算を行い、結果をディステネーション行列に格納します。記号表示すると :

$$dstM[i][j] = \sum_{k}(srcM1[i][k])(srcM2i[k][j])$$

ここで :
$0 \le i < numRows1$
$0 \le j < numCols2$
$0 \le k < numCols1Rows2$

インクルード : dsp.h

プロトタイプ :

```
extern fractional* MatrixMultiply (
    int numRows1,
    int numCols1Rows2,
    int numCols2,
    fractional* dstM,
    fractional* srcM1,
    fractional* srcM2
);
```

引数 :

numRows1	ソース 1 行列内の行数
numCols1Rows2	ソース 1 行列内の列数。これはソース 2 行列内の行数に一致する必要があります。
numCols2	ソース 2 行列内の列数
dstM	ディステネーション行列を指すポインタ
srcM1	ソース 1 行列を指すポインタ
srcM2	ソース 2 行列を指すポインタ

戻り値 : ディステネーション行列のベース・アドレスを指すポインタ

備考 : 次の絶対値が

$$\sum_{k}(srcM1[i][k])(srcM2i[k][j])$$

$1\text{-}2^{-15}$ より大きい場合、演算は (i,j) 要素に対して飽和します。
ソース 1 行列が正方行列である場合にのみ、この関数はイン・プレイス計算可能で、かつ同一適用可能です。この動作モードについては、このセクションの始めにある「その他の注意」も参照してください。

ソース・ファイル : mmul.s

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W13	待避、使用、復旧
ACCA	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 2 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
35

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
36 + *numRows1* * (8 + *numCols2* * (7 + 4 * *numCols1Rows2*))

MatrixScale

説明 : MatrixScale は、ソース行列内のすべての要素値にスケール値を乗算し、結果をディステネーション行列に格納します。

インクルード : dsp.h

プロトタイプ :
```
extern fractional* MatrixScale (
    int numRows,
    int numCols,
    fractional* dstM,
    fractional* srcM,
    fractional sclVal
);
```

引数 :

numRows	ソース行列内の行数
numCols	ソース行列内の列数
dstM	ディステネーション行列を指すポインタ
srcM	ソース行列を指すポインタ
sclVal	行列要素をスケールする値

戻り値 : ディステネーション行列のベース・アドレスを指すポインタ

備考 : この関数はイン・プレイス計算が可能です。

ソース・ファイル : mscl.s

関数プロファイル : システム・リソースの使用 :

W0..W5	使用、復旧なし
ACCA	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
14

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
20 + 3(*numRows* * *numCols*)

MatrixSubtract

説明 : MatrixSubtract は、ソース 2 行列内の各要素の値をソース 1 行列内の対応する各要素の値から減算して、結果をディステネーション行列に格納します。

インクルード : dsp.h

プロトタイプ :
```
extern fractional* MatrixSubtract (
    int numRows,
    int numCols,
    fractional* dstM,
    fractional* srcM1,
    fractional* srcM2
);
```

引数 :

numRows	ソース行列内の行数
numCols	ソース行列内の列数
dstM	ディステネーション行列を指すポインタ
srcM1	ソース 1 行列を指すポインタ (被減数)
srcM2	ソース 2 行列を指すポインタ (減数)

戻り値 : ディステネーション行列のベース・アドレスを指すポインタ

MatrixSubtract (続き)

備考 : *srcM1*[r][c]-*srcM2*[r][c] の絶対値が $1\text{-}2^{-15}$ より大きい場合、この演算は (r,c) 要素に対して飽和します。
この関数はイン・プレイス計算が可能です。
この関数は同一適用可能です。

ソース・ファイル : msub.s

関数プロファイル : システム・リソースの使用 :

W0..W4	使用、復旧なし
ACCA	使用、復旧なし
ACCB	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
15

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
20 + 4(*numRows* * *numCols*)

MatrixTranspose

説明 : MatrixTranspose はソース行列内で行と列を入れ換えて、結果をディステネーション行列に格納します。次のようになります :
dstM[i][j] = *srcM*[j][i],
$0 \le i$ *numRows*, $0 \le j$ *numCols*.

インクルード : dsp.h

プロトタイプ :

```
extern fractional* MatrixTranspose (
    int numRows,
    int numCols,
    fractional* dstM,
    fractional* srcM
);
```

引数 :

numRows	ソース行列内の行数
numCols	ソース行列内の列数
dstM	ディステネーション行列を指すポインタ
srcM	ソース行列を指すポインタ

戻り値 : ディステネーション行列のベース・アドレスを指すポインタ

備考 : ソース行列が正方行列である場合、この関数はイン・プレイス計算が可能です。この動作モードについては、このセクションの始めにある「その他の注意」も参照してください。

ソース・ファイル : mtrp.s

関数プロファイル : システム・リソースの使用 :

W0..W5	使用、復旧なし

DO 命令と REPEAT 命令の使用 :
DO 命令 : 2 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
14

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
16 + *numCols* * (6 + (*numRows*-1) * 3)

2.5.5 逆行列

非特異の正方小数行列の逆行列は正方行列 (次元が一致) になり、各要素値は離散的な小数の集合 {-1, ..., $1-2^{-15}$} になるとは限りません。したがって、小数行列に対する逆行列演算は提供されていません。

ただし、逆行列演算は非常に有用な演算であるため、浮動小数を採用した演算を DSP ライブラリ内に用意してあります。以下に、それを説明します。

MatrixInvert

説明 : MatrixInvert はソース行列の逆行列を計算して、結果をディステネーション行列に格納します。

インクルード : dsp.h

プロトタイプ :

```
extern float* MatrixInvert (
    int numRowsCols,
    float* dstM,
    float* srcM,
    float* pivotFlag,
    int* swappedRows,
    int* swappedCols
);
```

引数 :

numRowCols	ソース行列 (正方行列) の行数と列数
dstM	ディステネーション行列を指すポインタ
srcM	ソース行列を指すポインタ
次を内部で使用 :	
pivotFlag	長さ numRowsCols のベクタを指すポインタ
swappedRows	長さ numRowsCols のベクタを指すポインタ
swappedCols	長さ numRowsCols のベクタを指すポインタ

戻り値 : ディステネーション行列のベース・アドレスを指すポインタ、またはソース行列が特異行列の場合に NULL

備考 : *pivotFlag*、*swappedRows*、*swappedCols* の各ベクタは内部で使用されますが、この関数をコールする前にこれらのベクタを割り当てておく必要があります。
ソース行列が特異である場合 (行列式がゼロ)、この行列には逆行列は存在しません。この場合、関数は NULL を返します。
この関数はイン・プレイス計算が可能です。

ソース・ファイル : minv.s (C コードからアセンブル)

関数プロファイル :

システム・リソースの使用 :

W0..W7	使用、復旧なし
W8、W14	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
なし

プログラム・ワード数 (24 ビット命令):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

2.6 フィルタ処理関数

このセクションでは、DSP ライブラリで使う小数フィルタの概念を説明し、フィルタ演算を行う個々の関数について説明します。DSP ライブラリ・フィルタ関数を使ったプロジェクト例を用意してありますので、これを参照して関数の正しい使い方を確認してください。MPLAB IDE を採用したプロジェクト / ワークスペース例が MPLAB C30 ツールスイートのインストレーション・フォルダに用意してあります。

2.6.1 小数フィルタ演算

小数ベクタ x[n] ($0 \leqq n < N$) を使って表現したデータのフィルタ処理シーケンスは、差分方程式を解く過程と等価です。

$$y[n] + \sum_{p=1}^{P-1} (-a[p])(y[n-p]) = \sum_{m=0}^{M-1} (b[m])(x[n-m])$$

これを各 n 番毎のサンプルに対して計算すると、フィルタされたデータ・シーケンス y[n] が得られます。この意味で、小数フィルタは小数ベクタ a[p] ($0 \leqq p < P$) と b[m] ($0 \leqq m < M$) により決定されると言えます。これらのベクタはフィルタ係数のセットと呼ばれ、入力データ・シーケンスで表わされた信号に所定の変更を加えるためにデザインされています。

フィルタ処理では、入力と出力のデータ・シーケンス (x[n]、$-M + 1 \leqq n < 0$、および y[n]、$-P + 1 \leqq n < 0$) の過去の状態を知り管理することが重要です。こ過去の状態が、フィルタ演算の初期条件になります。また、入力データ・シーケンスの隣接セクションにフィルタを繰り返し適用する際、直前のフィルタ演算の最終状態 (x[n]、$N - M + 1 \leqq n < N - 1$、および y[n]、$N - P + 1 \leqq n < N - 1$) を記憶しておくことが必要です。この最終状態は、次のフィルタ処理ステージの計算で使用されます。過去の状態と現在の状態を考慮することは、フィルタ処理演算に必要なことです。

フィルタ処理演算の過去の状態と現在の状態の管理は一般に、フィルタ遅延線と呼ばれる別のシーケンス (これも小数ベクタ) を使って行われます。遅延は、フィルタ演算を適用する前のフィルタの過去の状態を記述しています。フィルタ処理演算を実行した後の遅延には、直前にフィルタ処理されたデータ・サンプルと直前の出力サンプルのセットが含まれています (特定のフィルタの正しい演算を行うためには、対応する初期化関数をコールして、遅延の値をゼロに初期化しておくことが推奨されます)。

DSP ライブラリから提供するフィルタでは、入力データ・シーケンスはソース・サンプルのシーケンスと呼ばれ、一方フィルタされたシーケンスはディステネーション・サンプルと呼ばれます。一般に、フィルタ係数 (a,b) と遅延が、フィルタ構造を構成していると見なすことができます。すべてのフィルタで、入力と出力のデータ・サンプルは、デフォルトの RAM メモリ空間 (X-Data または Y-Data) に配置されます。フィルタ係数は X-Data メモリまたはプログラム・メモリに配置することができ、フィルタ遅延の値は Y-Data からのみアクセスされます。

2.6.2 FIR フィルタと IIR フィルタ

フィルタの特性は、係数の値分布に依存します。特に、次の 2 種類のフィルタは重要です。有限インパルス応答 (FIR) フィルタは、$1 \leqq m < M$ に対して $a[m] = 0$ であり、無限インパルス応答 (IIR) フィルタは、{1, ..., M} 内の m に対して $a[0] \neq 0$、かつ $a[m] \neq 0$ です。FIR フィルタと IIR フィルタのファミリに属する他の分類では、演算が入力データ・シーケンスに与える効果を考慮しています。

さらに、フィルタ処理は前述の差分方程式の解法から構成されているとしても、差分方程式を直接計算するよりは効率良い方法がいくつかあります。また、小数演算の制約の下にフィルタ処理演算を実行するようにデザインされた他のフィルタ処理もあります。

これらの多くを考慮すると多くのフィルタ処理演算が存在しますが、DSP ライブラリはこれらの一部を提供します。

2.6.3 シングル・サンプル・フィルタ処理

DSP ライブラリで提供しているフィルタ処理関数は、ブロック処理用にデザインされています。各フィルタ関数は *numSamps* という名前の引数を受け取ります。この引数は、演算対象となる入力データのワード数 (ブロック・サイズ) を表わします。シングル・サンプル・フィルタ処理が必要な場合、*numSamps* に 1 を設定することができます。これにより、1 入力サンプルのフィルタ処理が可能になり、関数はフィルタの 1 出力サンプルを計算します。

2.6.4 ユーザーの考慮事項

このライブラリ内のすべての小数フィルタ処理演算は、処理するサンプル数、係数のサイズ、遅延ベクタを指定する際に入力パラメータまたはデータ構造体要素の値を使います。これらの値に基づいて、次のように仮定します :

a) 特定の演算に関係するすべてのベクタ (ベクタ・シーケンス) のサイズの総和は、ターゲット・デバイスの使用可能なデータ・メモリの範囲内にある
b) ディステネーション・ベクタは、演算結果を受け取れるように十分大きい
c) これらの関数は境界チェックを行いません。範囲外のサイズ (長さゼロのベクタも含む) を使用したり、違法なソース・ベクタ・セットと係数セットを使用すると、予期しない結果が発生します。
d) 各関数コールが完了した後に、STATUS レジスタ (SR) を調べることが推奨されます。特に、関数のリターン後に SA、SB、SAB の各フラグを調べると、飽和の有無を知ることができます。
e) ディステネーション・ベクタを返す演算をネストすることができます。例えば、次のようになります :

   a = Op1 (b, c)、ここで b = Op2 (d)、かつ c = Op3 (e, f) とすると、

   a = Op1 (Op2 (d)、Op3 (e, f))

2.6.5 個別関数

以下に、フィルタ演算を行う個別関数について説明します。デジタル・フィルタの詳細については、Alan Oppenheim および Ronald Schafer 著の「*Discrete-Time Signal Processing*」、Prentice Hall、1989 年を参照してください。最小 2 乗平均 FIR フィルタの詳細については、T. Hsia 執筆の「*Convergence Analysis of LMS and NLMS Adaptive Algorithms*」、Proc.ICASSP、pp.667-670、1983 年、さらに Sangil Park および Garth Hillman 執筆の「*On Acoustic-Echo Cancellation Implementation with Multiple Cascadable Adaptive FIR Filter Chips*」、Proc.ICASSP、1989 年を参照してください。

FIRStruct

構造体 : FIRStruct は、任意の FIR フィルタのフィルタ構造体を記述します。

インクルード : dsp.h

宣言 :

```
typedef struct {
  int numCoeffs;
  fractional* coeffsBase;
  fractional* coeffsEnd;
  int coeffsPage;
  fractional* delayBase;
  fractional* delayEnd;
  fractional* delay;
} FIRStruct;
```

パラメータ :

numCoeffs	フィルタ内の係数の数 (= M)
coeffsBase	フィルタ係数のベース・アドレス (= h)
coeffsEnd	フィルタ係数の最終アドレス
coeffsPage	係数バッファ・ページ番号
delayBase	遅延バッファのベース・アドレス
delayEnd	遅延バッファの最終アドレス
delay	遅延ポインタの現在値 (= d)

備考 :

フィルタ内の係数の数は M。
係数 h[m] は、0 ≦ m ＜ M で X-Data またはプログラム・メモリ内に定義。

遅延バッファ d[m] は、0 ≦ m ＜ M で Y-Data 内にのみ定義。

係数が X-Data 空間に保存された場合、*coeffsBase* は係数が配置されている実際のアドレスを指します。係数がプログラム・メモリに保存された場合、*coeffsBase* は係数が含まれているプログラム・ページ境界から係数が配置されているページ内のアドレスまでのオフセットになります。後者の値は、インライン・アセンブリ・オペレータ psvoffset() を使って計算することができます。

coeffsEnd は、フィルタ係数バッファの最終バイトの X-Data 空間 (プログラム・メモリの場合はオフセット) 内のアドレスです。

係数が X-Data 空間に保存された場合、*coeffsPage* には 0xFF00 (固定値 COEFFS_IN_DATA) を設定する必要があります。係数がプログラム・メモリ内に保存された場合、これは係数を含むプログラム・ページ番号になります。後者の値は、インライン・アセンブリ・オペレータ psvpage() を使って計算することができます。

delayBase は、遅延バッファが配置されている実際のアドレスを指します。

delayEnd は、フィルタ遅延バッファの最終バイトのアドレスです。

FIRStruct (続き)

係数バッファと遅延バッファが循環増加モジュロ・バッファとして構成された場合は、*coeffsBase* と *delayBase* は 2 つのアドレスの ' ゼロ ' 乗位置に配置される*必要があります*(*coeffsEnd* と *delayEnd* は奇数アドレス)。これらのバッファが巡回増加モジュロ・バッファとして構成されているか否かは、各 FIR フィルタ関数説明の備考に示してあります。

係数バッファと遅延バッファが巡回 (増加) モジュロ・バッファとして構成されていない場合は、*coeffsBase* と *delayBase* は 2 つのアドレスの ' ゼロ ' 乗に配置される*必要はなく*、*coeffsEnd* と *delayEnd* の値は特定の FIR フィルタ関数内で無視されます。

FIR

説明 : FIR は、ソース・サンプルのシーケンスに FIR フィルタを適用して、結果をディステネーション・サンプルのシーケンスへ格納して、遅延値を更新します。

インクルード : dsp.h

プロトタイプ :

```
extern fractional* FIR (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   FIRStruct* filter
);
```

引数 :

numSamps	フィルタ処理する入力サンプル数 (= N)
dstSamps	ディステネーション・サンプルを指すポインタ (= y)
srcSamps	ソース・サンプルを指すポインタ (= x)
filter	FIRStruct フィルタ構造体を指すポインタ

戻り値 : ディステネーション・サンプルのベース・アドレスを指すポインタ

備考 : フィルタ内の係数の数は M。
係数 h[m] は、0 ≦ m ＜ M で、巡回増加モジュロ・バッファとして構成。
遅延 d[m] は、0 ≦ m ＜ M で、巡回増加モジュロ・バッファとして構成。
ソース・サンプル x[n] は、0 ≦ n ＜ N で定義。
ディステネーション・サンプル y[n] は、0 ≦ n ＜ N で定義。
(FIRStruct、FIRStructInit、FIR DelayInit も参照してください)

ソース・ファイル : fir.s

FIR (続き)

関数プロファイル : システム・リソースの使用 :

W0..W6	使用、復旧なし
W8、W10	待避、使用、復旧
ACCA	使用、復旧なし
CORCON	待避、使用、復旧
MODCON	待避、使用、復旧
XMODSTRT	待避、使用、復旧
XMODEND	待避、使用、復旧
YMODSTRT	待避、使用、復旧
PSVPAG	待避、使用、復旧 (

係数が P メモリ内の場合)

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : 1 レベル

プログラム・ワード数 (24 ビット命令):
55

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
53 + N(4+M)、または
係数が P メモリの場合 56 + N(8+M)

例 : この関数の使い方を説明するサンプル・プロジェクトについては、MPLAB C30 インストレーション・フォルダを参照してください。

FIRDecimate

説明 : FIRDecimate は、ソース・サンプルのシーケンスを R : 1 またはこれと同等の割合で間引きして、信号のサンプル・レートを 1/R にします。次のようになります :
y[n] = x[Rn]
折り返しノイズの影響を軽減するため、ソース・サンプルをフィルタ処理した後にダウンサンプルします。間引きされた結果はディステネーション・サンプルのシーケンスに格納され、遅延値が更新されます。

インクルード : dsp.h

プロトタイプ :

```
extern fractional* FIRDecimate (
    int numSamps,
    fractional* dstSamps,
    fractional* srcSamps,
    FIRStruct* filter,
    int rate
);
```

引数 :

numSamps	出力サンプル数 (= N、N = Rp、p は整数)
dstSamp	ディステネーション・サンプルを指すポインタ (= y)
srcSamps	ソース・サンプルを指すポインタ (= x)
filter	FIRStruct フィルタ構造体を指すポインタ
rate	デシメーション率 (ダウンサンプリング係数 R)

戻り値 : ディステネーション・サンプルのベース・アドレスを指すポインタ

FIRDecimate (続き)

備考 : フィルタ内の係数の数は M で、M は R の整数倍。
係数 h[m] は 0 ≦ m ＜ M で、巡回モジュロ・バッファとして構成されていません。
遅延 d [m] は 0 ≦ m ＜ M で、巡回モジュロ・バッファとして構成されていません。
ソース・サンプル x[n] は、0 ≦ n ＜ NR で定義。
ディステネーション・サンプル y[n] は、0 ≦ n ＜ N で定義。
(FIRStruct、FIRStructInit、FIRDelayInit も参照してください)

ソース・ファイル : firdecim.s

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W12	待避、使用、復旧
ACCA	使用、復旧なし
CORCON	待避、使用、復旧
PSVPAG	待避、使用、復旧 (

係数が P メモリ内の場合)

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : 1 レベル

プログラム・ワード数 (24 ビット命令):
48

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
45 + N(10 + 2M)、または
係数が P メモリの場合 48 + N(13+2M)

FIRDelayInit

説明 : FIRDelayInit は、FIRStruct フィルタ構造体内の遅延値をゼロに初期化します。

インクルード : dsp.h

プロトタイプ :

```
extern void FIRDelayInit (
    FIRStruct* filter
);
```

引数 : *filter* FIRStruct フィルタ構造体を指すポインタ

備考 : FIRStruct 構造体の説明を参照してください。
注 : FIR インターポレータの遅延は、関数 FIRInterpDelayInit により初期化されます。

ソース・ファイル : firdelay.s

関数プロファイル : システム・リソースの使用 :

W0..W2	使用、復旧なし

DO 命令と REPEAT 命令の使用 :
DO 命令 : なし
REPEAT 命令 : 1 レベル

プログラム・ワード数 (24 ビット命令):
7

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
11 + M

FIRInterpolate

説明 : FIRInterpolate は、ソース・サンプルのシーケンスを比率 1 : R またはこれと同等の割合で補間して、信号のサンプル・レートを R 倍にします。次のようになります :
y[n] = x[n/R].
折り返しノイズの影響を軽減するため、ソース・サンプルをアップサンプルした後にフィルタ処理します。補間された結果はディステネーション・サンプルのシーケンスに格納され、遅延値が更新されます。

インクルード : dsp.h

プロトタイプ :
```
extern fractional* FIRInterpolate (
    int numSamps,
    fractional* dstSamps,
    fractional* srcSamps,
    FIRStruct* filter,
    int rate
);
```

引数 :

numSamps	入力サンプル数 (= N、N = Rp、p は整数)
dstSamps	ディステネーション・サンプルを指すポインタ (= y)
srcSamps	ソース・サンプルを指すポインタ (= x)
filter	FIRStruct フィルタ構造体を指すポインタ
rate	補間率 (アップサンプリング係数 R)

戻り値 : ディステネーション・サンプルのベース・アドレスを指すポインタ

備考 : フィルタ内の係数の数は M で、M は R の整数倍。
係数 h[m] は 0 ≦ m M で、巡回モジュロ・バッファとして構成されていません。
係数 d [m] は 0 ≦ m ＜ M で、巡回モジュロ・バッファとして構成されていません。
ソース・サンプル x[n] は、0 ≦ n ＜ N で定義。
ディステネーション・サンプル y[n] は、0 ≦ n ＜ NR で定義。
(FIRStruct、FIRStructInit、FIRInterDelayInit も参照してください)

ソース・ファイル : firinter.s

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W13	待避、使用、復旧
ACCA	使用、復旧なし
CORCON	待避、使用、復旧
PSVPAG	待避、使用、復旧 (

係数が P メモリ内の場合)

DO 命令と REPEAT 命令の使用 :
DO 命令 : 2 レベル
REPEAT 命令 : 1 レベル

プログラム・ワード数 (24 ビット命令):
63

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
45 + 6(M / R) + N(14 + M / R + 3M + 5R)、または
係数が P メモリにある場合、48 + 6(M / R) + N(14 + M / R + 4M + 5R)。

FIRInterpDelayInit

説明 : FIRInterpDelayInit は、FIR インターポレーション・フィルタ用に最適化された FIRStruct フィルタ構造体内の遅延値をゼロに初期化します。

インクルード : dsp.h

プロトタイプ :
```
extern void FIRDelayInit (
    FIRStruct* filter,
    int rate
);
```

引数 :

filter	FIRStruct フィルタ構造体を指すポインタ
rate	補間率 (アップサンプリング係数 R)

備考 : 遅延 d[m] は、0 ≦ m ＜ M/R で定義され、M はインターポレータ内のフィルタ係数の数。
FIRStruct 構造体の説明を参照してください。

ソース・ファイル : firintdl.s

関数プロファイル : システム・リソースの使用 :
W0..W4 使用、復旧なし

DO 命令と REPEAT 命令の使用 :
DO 命令 : なし
REPEAT 命令 : 1 レベル

プログラム・ワード数 (24 ビット命令):
13

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
10 + 7M/R

FIRLattice

説明 : FIRLattice は、ラティス構造体を使ってソース・サンプルのシーケンスへ FIR フィルタを適用します。次に、結果をディステネーション・サンプルのシーケンスに格納して、遅延値を更新します。

インクルード : dsp.h

プロトタイプ :
```
extern fractional* FIRLattice (
    int numSamps,
    fractional* dstSamps,
    fractional* srcSamps,
    FIRStruct* filter
);
```

引数 :

numSamps	フィルタ処理する入力サンプル数 (= N)
dstSamps	ディステネーション・サンプルを指すポインタ (= y)
srcSamps	ソース・サンプルを指すポインタ (= x)
filter	FIRStruct フィルタ構造体を指すポインタ

戻り値 : ディステネーション・サンプルのベース・アドレスを指すポインタ

備考 : フィルタ内の係数の数は M。
ラティス係数 h[m] は 0 ≦ m ＜ M で、巡回モジュロ・バッファとして構成されていません。
遅延 d [m] は 0 ≦ m ＜ M で、巡回モジュロ・バッファとして構成されていません。
ソース・サンプル x[n] は、0 ≦ n ＜ N で定義。
ディステネーション・サンプル y[n] は、0 ≦ n ＜ N で定義。
(FIRStruct、FIRStructInit、FIRDelayInit も参照してください)

ソース・ファイル : firlatt.s

FIRLattice (続き)

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W12	待避、使用、復旧
ACCA	使用、復旧なし
ACCB	使用、復旧なし
CORCON	待避、使用、復旧
PSVPAG	待避、使用、復旧 (

係数が P メモリ内の場合)

DO 命令と REPEAT 命令の使用 :
DO 命令 : 2 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
50

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
41 + N(4 + 7M)
係数が P メモリの場合 44 + N(4+8M)

FIRLMS

説明 : FIRLMS は、ソース・サンプルのシーケンスに適応型 FIR フィルタを適用し、結果をディステネーション・サンプルのシーケンスに格納して、遅延値を更新します。
フィルタ係数も、リファレンス・サンプルの値に従って適用される最小 2 乗平均アルゴリズムを使ってサンプル毎に更新されます。

インクルード : dsp.h

プロトタイプ :

```
extern fractional* FIRLMS (
    int numSamps,
    fractional* dstSamps,
    fractional* srcSamps,
    FIRStruct* filter,
    fractional* refSamps,
    fractional muVal
);
```

引数 :

numSamps	入力サンプル数 (= N)
dstSamps	ディステネーション・サンプルを指すポインタ (= y)
srcSamps	ソース・サンプルを指すポインタ (= x)
filter	FIRStruct フィルタ構造体を指すポインタ
refSamps	リファレンス・サンプルを指すポインタ (= r)
muVal	適応係数 (= mu)

戻り値 : ディステネーション・サンプルのベース・アドレスを指すポインタ

FIRLMS (続き)

備考 : フィルタ内の係数の数は M。
係数 h[m] は、0 ≦ m ＜ M で、巡回増加モジュロ・バッファとして構成。
遅延 d[m] は、0 ≦ m ＜ M-1 で、巡回増加モジュロ・バッファとして構成。
ソース・サンプル x[n] は、0 ≦ n ＜ N で定義。
リファレンス・サンプル r[n] は、0 ≦ n ＜ N で定義。
ディステネーション・サンプル y[n] は、0 ≦ n ＜ N で定義。
適応 :

h_m[n] = h_m[n - 1] + mu * (r[n] - y[n]) * x[n - m]、
ただし、0 ≦ n ＜ N、0 ≦ m ＜ M。

(r[n] - y[n]) の絶対値が 1 以上の場合、演算は飽和することがあります。
フィルタ係数はプログラム・メモリへ配置すると値を変更できないので、プログラム・メモリへ配置しないようにする*必要があります*。
フィルタ係数がプログラム・メモリに配置されていることを検出すると、関数は NULL を返します。
(`FIRStruct`、`FIRStructInit`、`FIRDelayInit` も参照してください)

ソース・ファイル : `firlms.s`

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W12	待避、使用、復旧
ACCA	使用、復旧なし
ACCB	使用、復旧なし
CORCON	待避、使用、復旧
MODCON	待避、使用、復旧
XMODSTRT	待避、使用、復旧
XMODEND	待避、使用、復旧
YMODSTRT	待避、使用、復旧

`DO` 命令と `REPEAT` 命令の使用 :
`DO` 命令 : 2 レベル
`REPEAT` 命令 : 1 レベル

プログラム・ワード数 (24 ビット命令):
76

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
61 + N(13 + 5M)

FIRLMSNorm

説明 : `FIRLMSNorm` は、ソース・サンプルのシーケンスに適応型 FIR フィルタを適用し、結果をディステネーション・サンプルのシーケンスに格納して、遅延値を更新します。
フィルタ係数も、リファレンス・サンプルの値に従って適用される正規化最小 2 乗平均アルゴリズムを使ってサンプル毎に更新されます。

インクルード : `dsp.h`

FIRLMSNorm (続き)

プロトタイプ :

```
extern fractional* FIRLMSNorm (
    int numSamps,
    fractional* dstSamps,
    fractional* srcSamps,
    FIRStruct* filter,
    fractional* refSamps,
    fractional muVal,
    fractional* energyEstimate
);
```

引数 :

numSamps	入力サンプル数 (= N)
dstSamps	ディステネーション・サンプルを指すポインタ (= y)
srcSamps	ソース・サンプルを指すポインタ (= x)
filter	FIRStruct フィルタ構造体を指すポインタ
refSamps	リファレンス・サンプルを指すポインタ (= r)
muVal	適応係数 (= mu)
energyEstimate	直前の M 入力信号サンプルの予測エネルギ値、M はフィルタ係数の数

戻り値 : ディステネーション・サンプルのベース・アドレスを指すポインタ

備考 : フィルタ内の係数の数は M。
係数 h[m] は、0 ≦ m ＜ M で、巡回増加モジュロ・バッファとして構成。
遅延 d[m] は、0 ≦ m ＜ M で、巡回増加モジュロ・バッファとして構成。
ソース・サンプル x[n] は、0 ≦ n ＜ N で定義。
リファレンス・サンプル r[n] は、0 ≦ n ＜ N で定義。
ディステネーション・サンプル y[n] は、0 ≦ n ＜ N で定義。
適応 :

h_m[n] = h_m[n - 1] + nu[n] * (r[n] - y[n]) * x[n - m]、
ただし、0 ≦ n ＜ N、0 ≦ m ＜ M。
ここで、nu[n] = mu / (mu + E[n])、
$E[n]=E[n-1] + (x[n])^2 - (x[n-M+1])^2$ は入力信号エネルギの予測値。

起動時、energyEstimate を E[-1] 値 (フィルタが最初に起動されたときはゼロ) に初期化する必要があります。リターン時、energyEstimate は値 E[N - 1] に更新されます (これは、入力信号の拡張をフィルタ処理する場合、後続の関数コールで起動値として使用可能)。
(r[n] - y[n]) の絶対値が 1 以上の場合、演算は飽和することがあります。
注 : エネルギ予測値のもう一つの式 :
$E[n] = (x[n])^2 + (x[N-1])^2 + ... + (x[N-M+2])^2$
したがって、予測値の計算で飽和を防止するためには、入力サンプル値が丸め処理して次を満たすようにする必要があります。

$$\sum_{m=0}^{-M+2} (x[n+m])^2 < 1 \text{ , for } 0 \leqq n < N.$$

フィルタ係数はプログラム・メモリへ配置すると値を変更できないので、プログラム・メモリへ配置しないようにする必要があります。
フィルタ係数がプログラム・メモリに配置されていることを検出すると、関数は NULL を返します。
(FIRStruct、FIRStructInit、FIRDelayInit も参照してください)

ソース・ファイル : firlmsn.s

FIRLMSNorm (続き)

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W13	待避、使用、復旧
ACCA	使用、復旧なし
ACCB	使用、復旧なし
CORCON	待避、使用、復旧
MODCON	待避、使用、復旧
XMODSTRT	待避、使用、復旧
XMODEND	待避、使用、復旧
YMODSTRT	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 2 レベル
REPEAT 命令 : 1 レベル

プログラム・ワード数 (24 ビット命令):
91

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
66 + N(49 + 5M)

FIRStructInit

説明 : FIRStructInit は、FIRStruct FIR フィルタ構造体内のパラメータ値を初期化します。

インクルード : dsp.h

プロトタイプ :

```
extern void FIRStructInit (
  FIRStruct* filter,
  int numCoeffs,
  fractional* coeffsBase,
  int coeffsPage,
  fractional* delayBase
);
```

引数 :

filter	FIRStruct フィルタ構造体を指すポインタ
numCoeffs	フィルタ内の係数の数 (= M)
coeffsBase	フィルタ係数のベース・アドレス (= h)
coeffsPage	係数バッファ・ページ番号
delayBase	遅延バッファのベース・アドレス

備考 : FIRStruct 構造体の説明を参照してください。
FIRStructInit は終了時に、coeffsEnd ポインタと delayEnd ポインタを初期化します。また、遅延も delayBase に一致させられます。

ソース・ファイル : firinit.s

関数プロファイル : システム・リソースの使用 :

W0..W5	使用、復旧なし

DO 命令と REPEAT 命令の使用 :
DO 命令 : なし
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
10

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
19

IIRCanonic

説明 : IIRCanonic は、標準 (ダイレクト・フォーム II) 双二次セクションのカスケード接続を使って、ソース・サンプルのシーケンスに IIR フィルタを適用します。次に、結果をディステネーション・サンプルのシーケンスに格納して、遅延値を更新します。

インクルード : dsp.h

プロトタイプ :

```
typedef struct {
  int numSectionsLess1;
  fractional* coeffsBase;
  int coeffsPage;
  fractional* delayBase;
  int initialGain;
  int finalShift;
} IIRCanonicStruct;

extern fractional* IIRCanonic (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   IIRCanonicStruct* filter
);
```

引数 : フィルタ構造体 :

numSectionsLess1	カスケード接続された二次 (双二次) セクション数より 1 だけ小さい数 (= S-1)
coeffsBase	フィルタ係数を指すポインタ (= {a、b})、X-Data またはプログラム・メモリ内
coeffsPage	係数バッファ・ページ番号、または係数がデータ空間にある場合は 0xFF00 (固定値 COEFFS_IN_DATA)
delayBase	フィルタ遅延を指すポインタ (= d)、Y-Data の場合
initialGain	初期ゲイン値
finalShift	出力スケーリング (左シフト)

フィルタ記述 :

numSamps	フィルタ処理する入力サンプル数 (= N)
dstSamps	ディステネーション・サンプルを指すポインタ (= y)
srcSamps	ソース・サンプルを指すポインタ (= x)
filter	IIRCanonicStruct フィルタ構造体を指すポインタ

戻り値 : ディステネーション・サンプルのベース・アドレスを指すポインタ

備考 : 順序付けられた集合 {a2[s], a1[s], b2[s], b1[s], b0[s]}、0 ≦ s ＜ S 内に配置された二次 (双二次) セクション当たり 5 個の係数が存在します。係数値は、Momentum Data Systems 社の dsPICFD フィルタ・デザイン・パッケージまたは同等のツールを使って生成する必要があります。
遅延は、セクション {d1[s], d2[s]}、0 ≦ s ＜ S 当たり 2 ワードのフィルタ状態で構成されます。
ソース・サンプル x[n] は、0 ≦ n ＜ N で定義。
ディステネーション・サンプル y[n] は、0 ≦ n ＜ N で定義。
初期ゲイン値は、フィルタ構造体に入る前に各入力サンプルに適用されます。
出力スケールは、結果を出力シーケンスに格納する前に、フィルタ構造体出力に対するシフトとして適用されます。フィルタ・ゲインを 0 dB に戻すために使われます。シフト数はゼロであることもできます。ゼロでない場合は、シフトするビット数を表わします。負の値は左シフトを、正の値は右シフトをそれぞれ表わします。

ソース・ファイル : iircan.s

IIRCanonic (続き)

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W11	待避、使用、復旧
ACCA	使用、復旧なし
CORCON	待避、使用、復旧
PSVPAG	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 2 レベル
REPEAT 命令 : 1 レベル

プログラム・ワード数 (24 ビット命令):
42

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
36 + N(8 + 7S)、または
係数が P メモリの場合 39 + N(9+12S)

IIRCanonicInit

説明 : IIRCanonicInit は、IIRCanonicStruct フィルタ構造体内の遅延値をゼロに初期化します。

インクルード : dsp.h

プロトタイプ :

```
extern void IIRCanonicInit (
    IIRCanonicStruct* filter
);
```

引数 : フィルタ構造体 :
(IIRCanonic 関数の説明を参照してください)

初期化記述 :

filter	IIRCanonicStruct フィルタ構造体を指すポインタ

備考 : 二次セクション {d1[s], d2[s]}、0 ≦ s ＜ S 当たり 2 ワードのフィルタ状態。

ソース・ファイル : iircan.s

関数プロファイル : システム・リソースの使用 :

W0、W1	使用、復旧なし

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
7

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
10 + S2.

IIRLattice

説明 : IIRLattice は、ラティス構造体を使ってソース・サンプルのシーケンスへ IIR フィルタを適用します。次に、結果をディステネーション・サンプルのシーケンスに格納して、遅延値を更新します。

インクルード : dsp.h

プロトタイプ :

```
typedef struct {
  int order;
  fractional* kappaVals;
  fractional* gammaVals;
  int coeffsPage;
  fractional* delay;
} IIRLatticeStruct;

extern fractional* IIRLattice (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   IIRLatticeStruct* filter
);
```

引数 :

フィルタ構造体 :

order	フィルタ次数 (= M、M ≦ N; N については FIRLattice を参照)
kappaVals	ラティス係数のベース・アドレス (= k)、X-Data またはプログラム・メモリ内
gammaVals	ラダー係数のベース・アドレス (= g)、X-Data またはプログラム・メモリ内 NULL の場合、関数は全極フィルタを構成します。
coeffsPage	係数バッファ・ページ番号、または係数がデータ空間にある場合は 0xFF00 (固定値 COEFFS_IN_DATA)
delay	遅延のベース・アドレス (= d)、Y-Data のみ

フィルタ記述 :

numSamps	フィルタ処理する入力サンプル数 (= N、N ≧ M; M については IIRLatticeStruct を参照)
dstSamps	ディステネーション・サンプルを指すポインタ (= y)
srcSamps	ソース・サンプルを指すポインタ (= x)
filter	IIRLatticeStruct フィルタ構造体を指すポインタ

戻り値 : ディステネーション・サンプルのベース・アドレスを指すポインタ

備考 : ラティス係数 k[m] は 0 ≦ m ≦ M で定義。
ラダー係数 g[m] は、0 ≦ m ≦ M 定義 (全極フィルタを構成する限り)。
遅延 d[m] は、0 ≦ m ≦ M で定義。
ソース・サンプル x[n] は、0 ≦ n < N で定義。
ディステネーション・サンプル y[n] は、0 ≦ n < N で定義。
注 : このライブラリで提供する小数による構成では、飽和する傾向があります。このセクションの終わりに記載する OCTAVE モデルなどのような浮動小数による構成を使って " オフライン " でフィルタをデザイン / テストしてください。
そして、浮動小数の実行時に順方向と逆方向の中間値を監視して、[-1、1] 範囲の外側のレベルを探ってください。中間値がこの範囲を超える場合、最大絶対値を使ってリアルタイムの小数フィルタを適用する前に入力信号をスケールします。すなわち、最大値の逆数を信号に乗算します。このスケーリングにより、小数構成での飽和を防止することができます。

ソース・ファイル : iirlatt.s

IIRLattice (続き)

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W13	待避、使用、復旧
ACCA	使用、復旧なし
ACCB	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 2 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
76

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
46 + N(16 + 7M)、または
係数がプログラム・メモリの場合 49 + N(20+8M)

全極フィルタ構成の場合 :
46 + N(16 + 6M)、または
係数がプログラム・メモリの場合 49 + N(16+7M)

IIRLatticeInit

説明 : IIRLatticeInit は、IIRLatticeStruct フィルタ構造体内の遅延値をゼロに初期化します。

インクルード : dsp.h

プロトタイプ :

```
extern void IIRLatticeInit (
    IIRLatticeStruct* filter
);
```

引数 : フィルタ構造体 :
(IIRLattice 関数の説明を参照してください)

初期化記述 :

filter	IIRLatticeStruct フィルタ構造体を指すポインタ

ソース・ファイル : iirlattd.s

関数プロファイル : システム・リソースの使用 :

W0..W2	使用、復旧なし

DO 命令と REPEAT 命令の使用 :
DO 命令 : なし
REPEAT 命令 : 1 レベル

プログラム・ワード数 (24 ビット命令):
6

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
10 + M

IIRTransposed

説明 : IIRTransposed は、転置 (ダイレクト・フォーム II) 双二次セクションのカスケード接続を使って、ソース・サンプルのシーケンスに IIR フィルタを適用します。次に、結果をディステネーション・サンプルのシーケンスに格納して、遅延値を更新します。

インクルード : dsp.h

プロトタイプ :

```
typedef struct {
  int numSectionsLess1;
  fractional* coeffsBase;
  int coeffsPage;
  fractional* delayBase1;
  fractional* delayBase2;
  int finalShift;
} IIRTransposedStruct;

extern fractional* IIRTransposed (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   IIRTransposedStruct* filter
);
```

引数 : フィルタ構造体 :

numSectionsLess1	カスケード接続された二次 (双二次) セクション数より 1 だけ小さい数 (= S-1)
coeffsBase	フィルタ係数を指すポインタ (= {a、b})、X-Data またはプログラム・メモリ内
coeffsPage	係数バッファ・ページ番号、または係数がデータ空間にある場合は 0xFF00 (固定値 COEFFS_IN_DATA)
delayBase1	フィルタ状態 1 を指すポインタ、二次セクション当たり 1 ワードの遅延 (= d1)、Y-Data のみ
delayBase2	フィルタ状態 2 を指すポインタ、二次セクション当たり 1 ワードの遅延 (= d2)、Y-Data のみ
finalShift	出力スケーリング (左シフト)

フィルタ記述 :

numSamps	フィルタ処理する入力サンプル数 (= N)
dstSamps	ディステネーション・サンプルを指すポインタ (= y)
srcSamps	ソース・サンプルを指すポインタ (= x)
filter	IIRTransposedStruct フィルタ構造体を指すポインタ

戻り値 : ディステネーション・サンプルのベース・アドレスを指すポインタ

備考 : 順序付けられた集合 {a2[s], a1[s], b2[s], b1[s], b0[s]}、0 ≦ s ＜ S 内に配置された二次 (双二次) セクション当たり 5 個の係数が存在します。係数値は、Momentum Data Systems 社の dsPICFD フィルタ・デザイン・パッケージまたは同等のツールを使って生成する必要があります。
遅延は 2 つの独立なバッファから構成され、各バッファにはセクション {d2[s], d1[s]}、0 ≦ s ＜ S 当たり 1 ワードのフィルタ状態が含まれます。
ソース・サンプル x[n] は、0 ≦ n ＜ N で定義。
ディステネーション・サンプル y[n] は、0 ≦ n ＜ N で定義。
出力スケールは、結果を出力シーケンスに格納する前に、フィルタ構造体出力に対するシフトとして適用されます。フィルタ・ゲインを 0 dB に戻すために使われます。シフト数はゼロであることもできます。ゼロでない場合は、シフトするビット数を表わします。負の値は左シフトを、正の値は右シフトをそれぞれ表わします。

ソース・ファイル : iirtrans.s

IIRTransposed (続き)

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W11	待避、使用、復旧
ACCA	使用、復旧なし
ACCB	使用、復旧なし
CORCON	待避、使用、復旧
PSVPAG	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 2 レベル
REPEAT 命令 : 1 レベル

プログラム・ワード数 (24 ビット命令):
48

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
35 + N(11 + 11S)、または
係数が P メモリの場合 38 + N(9+17S)
S は二次セクション数。

例 この関数の使い方を説明するサンプル・プロジェクトについては、MPLAB C30 インストレーション・フォルダを参照してください。

IIRTransposedInit

説明 : IIRTransposedInit は、IIRTransposedStruct フィルタ構造体内の遅延値をゼロに初期化します。

インクルード : dsp.h

プロトタイプ :
```
extern void IIRTransposedInit (
   IIRTransposedStruct* filter
);
```

引数 : フィルタ構造体 :
(IIRTransposed 関数の説明を参照してください)

初期化記述 :

filter	IIRTransposedStruct フィルタ構造体を指すポインタ

備考 : 遅延は 2 つの独立なバッファから構成され、各バッファにはセクション {d2[s], d1[s]}、0 ≦ s ＜ S 当たり 1 ワードのフィルタ状態が含まれます。

ソース・ファイル : iirtrans.s

関数プロファイル : システム・リソースの使用 :

W0..W2	使用、復旧なし

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
8

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
11 + 2S、
S は二次セクション数。

例 : この関数の使い方を説明するサンプル・プロジェクトについては、MPLAB C30 インストレーション・フォルダを参照してください。

2.6.6 IIRLattice フィルタ解析用 OCTAVE モデル

次の OCTAVE モデルは、IIRLattice 関数で提供される小数演算によるフィルタを使う前に IIR ラティス・フィルタの性能を調べるときに使うことができます。

IIRLattice OCTAVE モデル

```
function [out, del, forward, backward] = iirlatt (in, kappas, gammas, delay)
## FUNCTION.-
## IIRLATT: IIR Fileter Lattice implementation.
##
##     [out, del, forward, backward] = iirlatt (in, kappas, gammas, delay)
##
##     forward: records intermediate forward values.
##     backward: records intermediate backward values.

#.....................................................................

## Get implicit parameters.
numSamps = length(in); numKapps = length(kappas);
if (gammas != 0)
   numGamms = length(gammas);
else
   numGamms = 0;
endif
numDels = length(delay); filtorder = numDels-1;

## Error check.
if (numGamms != 0)
   if (numGamms != numKapps)
      fprintf ("ERROR!%d should be equal to %d.¥n", numGamms, numKapps);
      return;
   endif
endif
if (numDels != numKapps)
   fprintf ("ERROR!%d should equal to %d.¥n", numDels, numKapps);
   return;
endif

## Initialize.
M = filtorder; out = zeros(numSamps,1); del = delay;
forward = zeros(numSamps*M,1); backward = forward; i = 0;

## Filter samples.
for n = 1:numSamps
   ## Get new sample.
   current = in(n);
```

```
   ## Lattice structure.
   for m = 1:M
      after     = current  - kappas(M+1-m) * del(m+1);
      del(m)    = del(m+1) + kappas(M+1-m) * after;
      i = i+1;
      forward(i) = current;
      backward(i) = after;
      current   = after;
   end
   del(M+1) = after;

   ## Ladder structure (computes output).
   if (gammas == 0)
      out(n) = del(M+1);
   else
      for m = 1:M+1
         out(n) = out(n) + gammas(M+2-m)*del(m);
      endfor
   endif
endfor

## Return.
return;

#.....................................................................

endfunction
```

2.7 変換関数

このセクションでは、DSP ライブラリで使う小数変換の概念を説明し、変換演算を行う個々の関数について説明します。DSP ライブラリ変換関数を使ったプロジェクト例を用意してありますので、これを参照して関数の正しい使い方を確認してください。MPLAB IDE を採用したプロジェクト / ワークスペース例が MPLAB C30 ツールスイートのインストレーション・フォルダに用意してあります。

2.7.1 小数変換演算

小数変換は、時間領域の小数サンプル・シーケンスに適用すると、このサンプル・シーケンスを周波数領域内の小数周波数に変換する時間不変の離散的な線形演算です。逆に、逆小数変換演算は、周波数領域のデータに適用すると、これを時間領域表現に変換します。

DSP ライブラリは、変換のセット (および逆変換のサブセット) を提供しています。最初のセットは、離散的フーリエ変換 (またはその逆変換) を複素数データ・セットに適用します (下記の小数複素数値の説明を参照)。2 番目のセットは、タイプ II 離散的コサイン変換 (DCT) を実数シーケンスに適用します。これらの変換は、アウト・オブ・プレイスまたはイン・プレイスで演算するようにデザインされています。前者のタイプでは、出力シーケンスに変換結果が設定されます。後者では、入力シーケンスが変換されたシーケンスで物理的に置換えられます。アウト・オブ・プレイス演算の場合、計算結果を保持する十分なメモリを用意する必要があります。

変換では変換係数 (定数) を使用します。この変換係数は、起動時に変換関数に渡す必要があります。複素数データ・セットのこれらの係数は、浮動小数演算で計算された後に、演算で使用するために小数に変換されます。変換を適用する際に余分なオーバーヘッドをなくするため、変換係数の特定のセットを 1 回だけ生成して、プログラムの実行時に何回も使用することができます。したがって、初期化演算から返された係数を固定 (静的) ベクタとして保存することをお薦めします。係数を " オフライン " で生成して、プログラム・メモリ内に配置し、後でプログラムを実行する際に使うことも有効です。この方法では、サイクル数だけでなく、変換を使うアプリケーションをデザインする際に RAM メモリも節約できます。

2.7.2 小数複素数ベクタ

複素数データ・ベクタは、各ベクタ要素を表わす値の対で構成されたデータ・セットで表わされます。対の最初の値は要素の実数部で、2 番目の値は虚数部です。実数部と虚数部は、メモリの 1 ワード (2 バイト) を使って格納され、1.15 形式の小数として解釈されます。小数複素数ベクタの要素は、小数ベクタの場合と同様に、メモリ内に連続して格納されます。

小数複素数ベクタ形式のデータ構造は、次のデータ構造体を使ってアドレス指定することができます。

```
#ifdef fractional
#ifndef fractcomplex
typedef struct {
  fractional real;
  fractional imag;
} fractcomplex;
#endif
#endif
```

2.7.3 ユーザーの考慮事項

a) これらの関数は境界チェックを行いません。範囲外のサイズ (長さゼロのベクタも含む) を使用したり、違法なソース複素数ベクタ・セットと係数セットを使用すると、予期しない結果が発生します。

b) 各関数コールが完了した後に、STATUS レジスタ (SR) を調べることが推奨されます。特に、関数のリターン後に SA、SB、SAB の各フラグを調べると、飽和の有無を知ることができます。

c) 変換ファミリで使われる入力と出力の複素数ベクタは、Y-Data メモリに配置する必要があります。変換係数は、X-Data またはプログラム・メモリに配置することができます。

d) ビット反転アドレッシングではベクタ・セットをモジュロ整列させる必要があるため、明示的または暗黙的に BitReverseComplex 関数を使う演算内の入力と出力の複素数ベクタは、正しく配置する必要があります。

e) ディステネーション複素数ベクタを返す演算をネストすることができます。例えば、次のようになります :

a = Op1 (b, c)、ここで b = Op2 (d)、かつ c = Op3 (e, f) とすると、

a = Op1 (Op2 (d)、Op3 (e, f))

以下に、変換演算と逆変換演算を行う個別関数について説明します。

BitReverseComplex

説明 : `BitReverseComplex` は、複素数ベクタの要素のビット順を逆にします。

インクルード : `dsp.h`

プロトタイプ :

```
extern fractcomplex* BitReverseComplex (
    int log2N,
    fractcomplex* srcCV
);
```

引数 :

log2N N の 2 を基底とする対数 (ソース・ベクタ内の複素要素数)
srcCV ソース複素数ベクタを指すポインタ

戻り値 : ソース複素数ベクタのベース・アドレスを指すポインタ

備考 : N は、2 の整数乗である*必要があります*。
srcCV ベクタは、N のモジュロ整列で配置される必要があります。
この関数はイン・プレイス計算が可能です。

ソース・ファイル : `bitrev.s`

関数プロファイル :

システム・リソースの使用 :

W0..W7	使用、復旧なし
MODCON	待避、使用、復旧
XBREV	待避、使用、復旧

`DO` 命令と `REPEAT` 命令の使用 :
`DO` 命令 : 1 レベル
`REPEAT` 命令 : なし

プログラム・ワード数 (24 ビット命令):
27

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
下に記載 :

例 : この関数の使い方を説明するサンプル・プロジェクトについては、MPLAB C30 インストレーション・フォルダを参照してください。

変換サイズ	複素数要素の数	サイクル数
32 ポイント	32	245
64 ポイント	64	485
128 ポイント	128	945
256 ポイント	256	1905

CosFactorInit

説明 : CosFactorInit は、タイプ II 離散的コサイン変換で必要なコサイン係数の最初の半分を生成して、結果を複素数ディステネーション・ベクタへ格納します。セットには次の値が格納されます :

$CN(k) = e^{j\frac{\pi k}{2N}}$、ここで 0 ≦ k N/2。

インクルード : dsp.h

プロトタイプ :

```
extern fractcomplex* CosFactorInit (
   int log2N,
   fractcomplex* cosFactors
);
```

引数 :

log2N	N の 2 を基底とする対数 (DCT で必要とされる複素数係数の数)
cosFactors	複素数コサイン係数を指すポインタ

戻り値 : コサイン係数のベース・アドレスを指すポインタ

備考 : N は、2 の整数乗である*必要があります*。
最初の半分の N/2 個のコサイン係数のみを発生。
関数のコール前に、サイズ N/2 の複素数ベクタは既に配置済みであり、cosFactors に割り当て済みである*必要があります*。複素数ベクタは X-Data メモリに配置する*必要があります*。
係数は浮動小数演算で計算され、1.15 複素数小数に変換されます。

ソース・ファイル : initcosf.c

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W14	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
なし

プログラム・ワード数 (24 ビット命令):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

DCT

説明 : DCT は、ソース・ベクタの離散的コサイン変換を計算し、結果をディステネーション・ベクタへ格納します。

インクルード : dsp.h

プロトタイプ :

```
extern fractional* DCT (
   int log2N,
   fractional* dstV,
   fractional* srcV,
   fractcomplex* cosFactors,
   fractcomplex* twidFactors,
   int factPage
);
```

引数 :

log2N	N の 2 を基底とする対数 (ソース・ベクタ内の複素要素数)
dstCV	ディステネーション・ベクタを指すポインタ
srcCV	ソース・ベクタを指すポインタ
cosFactors	コサイン係数を指すポインタ
twidFactors	調整係数を指すポインタ
factPage	変換係数のメモリ・ページ

戻り値 : ディステネーション・ベクタのベース・アドレスを指すポインタ

備考 : N は、2 の整数乗である*必要があります*。
この関数はアウト・オブ・プレイス計算が可能です。サイズ 2N のベクタは配置済みで、dstV に割り当て済みである必要があります。
dstV ベクタは、N のモジュロ整列で配置される必要があります。
計算結果は、ディステネーション・ベクタの最初の N 個の要素に格納されます。
計算中の飽和 (オーバーフロー) を防止するため、ソース・ベクタの値は [-0.5、0.5] の範囲内である*必要があります*。
最初の半分の N/2 個のコサイン係数のみ必要。
最初の半分の N/2 個の調整係数のみ必要。
変換係数が X-Data 空間に格納されている場合は、cosFactors と twidFactors は、係数が配置されている実際のアドレスを指します。変換係数がプログラム・メモリに格納されている場合は、cosFactors と twidFactors は、係数が配置されているプログラム・ページ境界からのオフセットになります。後者の値は、インライン・アセンブリ・オペレータ psvoffset () を使って計算することができます。
変換係数が X-Data 空間に配置されている場合は、factPage には 0xFF00 (固定値 COEFFS_IN_DATA) を設定する必要があります。係数がプログラム・メモリ内に配置された場合、factPage は係数を含むプログラム・ページ番号になります。後者の値は、インライン・アセンブリ・オペレータ psvpage () を使って計算することができます。
調整係数は、conjFlag をゼロ以外の値設定して初期化する*必要があります*。
最初の半分の N/2 個のコサイン係数のみ必要。
出力は、係数によりスケールされます。$1/(\sqrt{2N})$

ソース・ファイル : dctoop.s

DCT (続き)

関数プロファイル : システム・リソースの使用 :
W0..W5 使用、復旧なし
さらに、VectorZeroPad と DCTIP からのシステム・リソース

DO 命令と REPEAT 命令の使用 :
DO 命令 : なし
REPEAT 命令 : なし
さらに、VectorZeroPad と DCTIP からの DO/REPEAT 命令

プログラム・ワード数 (24 ビット命令):
16
さらに、VectorZeroPad と DCTIP からのプログラム・ワード数

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
22
さらに、VectorZeroPad と DCTIP からのサイクル数

注 : VectorZeroPad の説明で、報告されるサイクル数には 4 サイクルの C 関数コール・オーバーヘッドが含まれます。したがって、VectorZeroPad から DCT へ加算される実際のサイクル数は、VectorZeroPad 単体に対して報告される数値より小さい 4 になります。同様に、DCTIP から DCT へ加算される実際のサイクル数は、DCTIP 単体に対して報告される数値より小さい 3 になります。

DCTIP

説明 : DCTIP は、ソース・ベクタの離散的コサイン変換をイン・プレイス計算します。

インクルード : dsp.h

プロトタイプ :
```
extern fractional* DCTIP (
    int log2N,
    fractional* srcV,
    fractcomplex* cosFactors,
    fractcomplex* twidFactors,
    int factPage
);
```

引数 :

log2N	N の 2 を基底とする対数 (ソース・ベクタ内の複素要素数)
srcCV	ソース・ベクタを指すポインタ
cosFactors	コサイン係数を指すポインタ
twidFactors	調整係数を指すポインタ
factPage	変換係数のメモリ・ページ

戻り値 : ディステネーション・ベクタのベース・アドレスを指すポインタ

DCTIP (続き)

備考 : N は、2 の整数乗である*必要があります*。
この関数は、ソース・ベクタには長さ 2N までゼロが詰められているものと想定します。
srcV ベクタは、N のモジュロ整列で配置される必要があります。
計算結果は、ソース・ベクタの最初の N 個の要素に格納されます。
計算中の飽和 (オーバーフロー) を防止するため、ソース・ベクタの値は [-0.5、0.5] の範囲内である*必要があります*。
最初の半分の N/2 個のコサイン係数のみ必要。
最初の半分の N/2 個の調整係数のみ必要。
変換係数が X-Data 空間に格納されている場合は、cosFactors と twidFactors は、係数が配置されている実際のアドレスを指します。変換係数がプログラム・メモリに格納されている場合は、cosFactors と twidFactors は、係数が配置されているプログラム・ページ境界からのオフセットになります。後者の値は、インライン・アセンブリ・オペレータ psvoffset() を使って計算することができます。
変換係数が X-Data 空間に配置されている場合は、factPage には 0xFF00 (固定値 COEFFS_IN_DATA) を設定する必要があります。係数がプログラム・メモリ内に配置された場合、factPage は係数を含むプログラム・ページ番号になります。後者の値は、インライン・アセンブリ・オペレータ psvpage() を使って計算することができます。
調整係数は、conjFlag をゼロ以外の値設定して初期化する*必要があります*。
出力は、係数によりスケールされます。$1/(\sqrt{2N})$.

ソース・ファイル : dctoop.s

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W13	待避、使用、復旧
ACCA	使用、復旧なし
CORCON	待避、使用、復旧
PSVPAG	待避、使用、復旧 (係数が P メモリ内の場合)

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : 1 レベル
さらに、IFFTComplexIP からの DO/REPEAT 命令

プログラム・ワード数 (24 ビット命令):
92
さらに、IFFTComplexIP からのプログラム・ワード数
サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
71 + 10N、または
係数がプログラム・メモリに格納されている場合は 73 + 11N、
さらに、IFFTComplexIP からのサイクル数

注 : IFFTComplexIP の説明で、報告されるサイクル数には 4 サイクルの C 関数コールのオーバーヘッドが含まれます。したがって、IFFTComplexIP から DCTIP へ加算される実際のサイクル数は、IFFTComplexIP 単体に対して報告される数値より小さい 4 になります。

FFTComplex

説明 : FFTComplex は、ソース複素数ベクタの離散的フーリエ変換を計算し、結果をディステネーション複素数ベクタへ格納します。

インクルード : dsp.h

プロトタイプ :

```
extern fractcomplex* FFTComplex (
   int log2N,
   fractcomplex* dstCV,
   fractcomplex* srcCV,
   fractcomplex* twidFactors,
   int factPage
);
```

引数 :

log2N	N の 2 を基底とする対数 (ソース・ベクタ内の複素要素数)
dstCV	ディステネーション複素数ベクタを指すポインタ
srcCV	ソース複素数ベクタを指すポインタ
twidFactors	調整係数のベース・アドレス
factPage	変換係数のメモリ・ページ

戻り値 : ディステネーション・ベクタのベース・アドレスを指すポインタ

備考 : N は、2 の整数乗である*必要があります*。
この関数はアウト・オブ・プレイス計算が可能です。演算結果を受け取る十分大きい複素数ベクタを配置し、dstCV へ割り当てておく必要があります。
dstCV ベクタは、N のモジュロ整列で配置される必要があります。
ソース複素数ベクタの要素は、自然順であると想定されています。
ディステネーション複素数ベクタの要素は、自然順で生成されます。
計算中の飽和 (オーバーフロー) を防止するため、ソース複素数ベクタの値は [-0.5、0.5] の範囲内である*必要があります*。
最初の半分の N/2 個の調整係数のみ必要。
調整係数が X-Data 空間に格納されている場合は、twidFactors は、係数が配置されている実際のアドレスを指します。調整係数がプログラム・メモリに格納されている場合は、twidFactors は、係数が配置されているプログラム・ページ境界からのオフセットになります。後者の値は、インライン・アセンブリ・オペレータ psvoffset () を使って計算することができます。
調整係数が X-Data 空間に配置されている場合は、factPage には 0xFF00 (固定値 COEFFS_IN_DATA) を設定する必要があります。係数がプログラム・メモリ内に配置された場合、factPage は係数を含むプログラム・ページ番号になります。後者の値は、インライン・アセンブリ・オペレータ psvpage () を使って計算することができます。
調整係数は、conjFlag をゼロに設定して初期化する*必要があります*。
出力は、係数 1/N によりスケールされます。

ソース・ファイル : fftoop.s

FFTComplex (続き)

関数プロファイル : システム・リソースの使用 :
W0..W4 使用、復旧なし
さらに、VectorCopy、FFTComplexIP、BitReverseComplex からのシステム・リソース。

DO 命令と REPEAT 命令の使用 :
DO 命令 : なし
REPEAT 命令 : なし
さらに、VectorCopy、FFTComplexIP、BitReverseComplex からの DO/REPEAT 命令。

プログラム・ワード数 (24 ビット命令):
17
さらに、VectorCopy、FFTComplexIP、BitReverseComplex からのプログラム・ワード数。

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
23
さらに、VectorCopy、FFTComplexIP、BitReverseComplex からのサイクル数。

注 : VectorCopy の説明で、報告されるサイクル数には 3 サイクルの C 関数コール・オーバーヘッドが含まれます。したがって、VectorCopy から FFTComplex へ加算される実際のサイクル数は、VectorCopy 単体に対して報告される数値より小さい 3 になります。同様に、FFTComplexIP から FFTComplex へ加算される実際のサイクル数は、FFTComplexIP 単体に対して報告される数値より小さい 4 になります。また、BitReverseComplex からのサイクル数は、FFTComplex 単体に対して報告される値より小さい 2 になります。

FFTComplexIP

説明 : FFTComplexIP は、ソース複素数ベクタの離散的フーリェ変換をイン・プレイス計算します。

インクルード : dsp.h

プロトタイプ

```
extern fractcomplex* FFTComplexIP (
   int log2N,
   fractcomplex* srcCV,
   fractcomplex* twidFactors,
   int factPage
);
```

引数 :

log2N	N の 2 を基底とする対数 (ソース・ベクタ内の複素要素数)
srcCV	ソース複素数ベクタを指すポインタ
twidFactors	調整係数のベース・アドレス
factPage	変換係数のメモリ・ページ

戻り値 : ソース複素数ベクタのベース・アドレスを指すポインタ

FFTComplexIP (続き)

備考 : N は、2 の整数乗である*必要があります*。
ソース複素数ベクタの要素は、自然順であると想定されています。変換結果は、逆ビット順で格納されます。
計算中の飽和 (オーバーフロー) を防止するため、ソース複素数ベクタの値は [-0.5、0.5] の範囲内である必要があります。
最初の半分の N/2 個の調整係数のみ必要。
調整係数が X-Data 空間に格納されている場合は、`twidFactors` は、係数が配置されている実際のアドレスを指します。調整係数がプログラム・メモリに格納されている場合は、`twidFactors` は、係数が配置されているプログラム・ページ境界からのオフセットになります。後者の値は、インライン・アセンブリ・オペレータ `psvoffset ()` を使って計算することができます。
調整係数が X-Data 空間に配置されている場合は、`factPage` には 0xFF00 (固定値 `COEFFS_IN_DATA`) を設定する必要があります。係数がプログラム・メモリ内に配置された場合、`factPage` は係数を含むプログラム・ページ番号になります。後者の値は、インライン・アセンブリ・オペレータ `psvpage ()` を使って計算することができます。
調整係数は、`conjFlag` をゼロに設定して初期化する*必要があります*。
出力は、係数 1/N によりスケールされます。

ソース・ファイル : `fft.s`

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W13	待避、使用、復旧
ACCA	使用、復旧なし
ACCB	使用、復旧なし
CORCON	待避、使用、復旧
PSVPAG	待避、使用、復旧 (係数が P メモリ内の場合)

`DO` 命令と `REPEAT` 命令の使用 :
`DO` 命令 : 2 レベル
`REPEAT` 命令 : なし

プログラム・ワード数 (24 ビット命令):
59

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
下表に記載 :

例 : この関数の使い方を説明するサンプル・プロジェクトについては、MPLAB C30 インストレーション・フォルダを参照してください。

変換サイズ	調整係数が X- メモリの場合のサイクル数	調整係数が P- メモリの場合のサイクル数
32 ポイント	1,633	1,795
64 ポイント	3,739	4,125
128 ポイント	8,485	9,383
256 ポイント	19,055	21,105

IFFTComplex

説明 : IFFTComplex は、ソース複素数ベクタの逆離散的フーリエ変換を計算し、結果をディステネーション複素数ベクタへ格納します。

インクルード : dsp.h

プロトタイプ :

```
extern fractcomplex* IFFTComplex (
   int log2N,
   fractcomplex* dstCV,
   fractcomplex* srcCV,
   fractcomplex* twidFactors,
   int factPage
);
```

引数 :

log2N	N の 2 を基底とする対数 (ソース・ベクタ内の複素要素数)
dstCV	ディステネーション複素数ベクタを指すポインタ
srcCV	ソース複素数ベクタを指すポインタ
twidFactors	調整係数のベース・アドレス
factPage	変換係数のメモリ・ページ

戻り値 : ディステネーション・ベクタのベース・アドレスを指すポインタ

備考 : N は、2 の整数乗である*必要があります*。
この関数はアウト・オブ・プレイス計算が可能です。演算結果を受け取る十分大きい複素数ベクタを配置し、dstCV へ割り当てておく*必要があります*。
dstCV ベクタは、N のモジュロ整列で配置される必要があります。
ソース複素数ベクタの要素は、自然順であると想定されています。
ディステネーション複素数ベクタの要素は、自然順で生成されます。
計算中の飽和 (オーバーフロー) を防止するため、ソース複素数ベクタの値は [-0.5、0.5] の範囲内である必要があります。
調整係数が X-Data 空間に格納されている場合は、twidFactors は、係数が配置されている実際のアドレスを指します。調整係数がプログラム・メモリに格納されている場合は、twidFactors は、係数が配置されているプログラム・ページ境界からのオフセットになります。後者の値は、インライン・アセンブリ・オペレータ psvoffset () を使って計算することができます。
調整係数が X-Data 空間に配置されている場合は、factPage には 0xFF00 (固定値 COEFFS_IN_DATA) を設定する必要があります。係数がプログラム・メモリ内に配置された場合、factPage は係数を含むプログラム・ページ番号になります。後者の値は、インライン・アセンブリ・オペレータ psvpage () を使って計算することができます。
調整係数は、conjFlag をゼロ以外の値設定して初期化する*必要があります*。
最初の半分の N/2 個の調整係数のみ必要。

ソース・ファイル : ifftoop.s

IFFTComplex (続き)

関数プロファイル : システム・リソースの使用 :
W0..W4 使用、復旧なし
さらに、VectorCopy と IFFTComplexIP からのシステム・リソース
DO 命令と REPEAT 命令の使用 :
DO 命令 : なし
REPEAT 命令 : なし
さらに、VectorCopy と IFFTComplexIP からの DO/REPEAT 命令
プログラム・ワード数 (24 ビット命令):
12
さらに、VectorCopy と IFFTComplexIP からのプログラム・ワード数
サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
15
さらに、VectorCopy と IFFTComplexIP からのサイクル数

注 : VectorCopy の説明で、報告されるサイクル数には 3 サイクルの C 関数コール・オーバーヘッドが含まれます。したがって、VectorCopy から IFFTComplex へ加算される実際のサイクル数は、VectorCopy 単体に対して報告される数値より小さい 3 になります。同様に、IFFTComplexIP から IFFTComplex へ加算される実際のサイクル数は、IFFTComplexIP 単体に対して報告される数値より小さい 4 になります。

IFFTComplexIP

説明 : IFFTComplexIP は、ソース複素数ベクタの逆離散的フーリェ変換をイン・プレイス計算します。

インクルード : dsp.h

プロトタイプ :
```
extern fractcomplex* IFFTComplexIP (
    int log2N,
    fractcomplex* srcCV,
    fractcomplex* twidFactors,
    int factPage
);
```

引数 :

log2N	N の 2 を基底とする対数 (ソース・ベクタ内の複素要素数)
srcCV	ソース複素数ベクタを指すポインタ
twidFactors	調整係数のベース・アドレス
factPage	変換係数のメモリ・ページ

戻り値 : ソース複素数ベクタのベース・アドレスを指すポインタ

IFFTComplexIP (続き)

備考 : N は、2 の整数乗である*必要があります*。
ソース複素数ベクタの要素は、逆順であると想定されています。変換結果は、自然順で格納されます。
srcCV ベクタは、N のモジュロ整列で配置される必要があります。
計算中の飽和 (オーバーフロー) を防止するため、ソース複素数ベクタの値は [-0.5、0.5] の範囲内である*必要があります*。
調整係数が X-Data 空間に格納されている場合は、twidFactors は、係数が配置されている実際のアドレスを指します。調整係数がプログラム・メモリに格納されている場合は、twidFactors は、係数が配置されているプログラム・ページ境界からのオフセットになります。後者の値は、インライン・アセンブリ・オペレータ psvoffset () を使って計算することができます。
調整係数が X-Data 空間に配置されている場合は、factPage には 0xFF00 (固定値 COEFFS_IN_DATA) を設定する必要があります。係数がプログラム・メモリ内に配置された場合、factPage は係数を含むプログラム・ページ番号になります。後者の値は、インライン・アセンブリ・オペレータ psvpage () を使って計算することができます。
調整係数は、conjFlag をゼロ以外の値設定して初期化する*必要があります*。
最初の半分の N/2 個の調整係数のみ必要。

ソース・ファイル : ifft.s

関数プロファイル : システム・リソースの使用 :
W0..W3 使用、復旧なし
さらに、FFTComplexIP と BitReverseComplex からのシステム・リソース。

DO 命令と REPEAT 命令の使用 :
DO 命令 : なし
REPEAT 命令 : なし
さらに、FFTComplexIP と BitReverseComplex からの DO/REPEAT 命令。

プログラム・ワード数 (24 ビット命令):
11
さらに、FFTComplexIP と BitReverseComplex からのプログラム・ワード数。

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
15
さらに、FFTComplexIP と BitReverseComplex からのサイクル数。

注 : FFTComplexIP の説明で、報告されるサイクル数には 3 サイクルの C 関数コールのオーバーヘッドが含まれます。したがって、FFTComplexIP から IFFTComplexIP へ加算される実際のサイクル数は、FFTComplexIP 単体に対して報告される数値より小さい 3 になります。同様に、BitReverseComplex から IFFTComplexIP へ加算される実際のサイクル数は、BitReverseComplex 単体に対して報告される数値より小さい 2 になります。

SquareMagnitudeCplx

説明 : SquareMagnitudeCplx は、複素数ソース・ベクタ内の各要素の大きさの 2 乗を計算します。

インクルード : dsp.h

プロトタイプ :
```
extern fractional* SquareMagnitudeCplx (
    int numelems,
    fractcomplex* srcV,
    fractional* dstV
);
```

引数 :

numElems	複素数ソース・ベクタ内の要素数
srcV	複素数ソース・ベクタを指すポインタ
dstV	実数ディステネーション・ベクタを指すポインタ

戻り値 : ディステネーション・ベクタのベース・アドレスを指すポインタ

備考 : ソース・ベクタ内の複素数要素の実数部と虚数部の 2 乗の和が $1\text{-}2^{-15}$ より大きい場合、この演算は飽和します。
この関数を使って、ソース・データ・セットに対してイン・プレイス計算することができます。

ソース・ファイル : cplxsqrmag.s

関数プロファイル : システム・リソースの使用 :

W0..W2	使用、復旧なし
W4、W5、W10	待避、使用、復旧
ACCA	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 1 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
19

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
20 + 3(*numElems*)

例 : この関数の使い方を説明するサンプル・プロジェクトについては、MPLAB C30 インストレーション・フォルダを参照してください。

TwidFactorInit

説明 : TwidFactorInit は、離散的フーリェ変換または離散的コサイン変換で必要な調整係数の最初の半分を生成して、結果を複素数ディステネーション・ベクタへ格納します。セットには次の値が格納されます :

$WN(k) = e^{-j\frac{2\pi k}{N}}$ 、ここで $0 \leqq k \leqq N/2$、*conjFlag* = 0

$WN(k) = e^{j\frac{2\pi k}{N}}$ 、ここで $0 \leqq k \leqq N/2$、*conjFlag* ! = 0

インクルード : dsp.h

プロトタイプ :
```
extern fractcomplex* TwidFactorInit (
    int log2N,
    fractcomplex* twidFactors,
    int conjFlag
);
```

TwidFactorInit (続き)

引数 :

log2N	N の 2 を基底とする対数 (DFT で必要とされる複素数係数の数)
twidFactors	複素数の調整係数を指すポインタ
conjFlag	共役値生成の有無を表示するフラグ

戻り値 : 調整係数のベース・アドレスを指すポインタ

備考 : N は、2 の整数乗である*必要があります*。
最初の半分の N/2 個の調整係数のみ生成。
conjFlag 値が、指数関数の引数の符号を決定します。フーリエ変換の場合、conjFlag に '0' を設定します。逆フーリエ変換と離散的コサイン変換の場合は、conjFlag に '1' を設定します。
関数のコール前に、サイズ N/2 の複素数ベクタは既に配置済みであり、twidFactors に割り当て済みである必要があります。複素数ベクタは X-Data メモリに配置する必要があります。
係数は浮動小数演算で計算され、1.15 複素数小数に変換されます。

ソース・ファイル : inittwid.c

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W14	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
なし

プログラム・ワード数 (24 ビット命令):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

例 : この関数の使い方を説明するサンプル・プロジェクトについては、MPLAB C30 インストレーション・フォルダを参照してください。

2.8 制御関数

このセクションでは、閉ループ制御システムの実現を支援する、DSP ライブラリ提供の関数について説明します。

2.8.1 PID 制御

比例積分微分 (PID) コントローラの詳しい説明は、ここでの説明の範囲を超えますが、このセクションでは PID コントローラ調整のガイドラインを提供します。

2.8.1.1 PID コントローラの背景

PID コントローラは閉制御ループ内で誤差信号に応答し、所定のシステム応答を実現するように制御量を調節します。制御パラメータとしては、速度、電圧または電流などの任意の測定可能なシステム量が可能です。PID コントローラの出力は、制御システム量に影響を与える 1 個または複数のシステム・パラメータを制御することができます。例えば、センサレス・ブラシレス DC モータ・アプリケーション内の速度制御ループは、PWM デューティ・サイクルを直接制御することができます。あるいは、モータ電流を安定化する内側制御ループに対する電流要求を設定することができます。PID コントローラの利点は、1 個または複数のゲイン値を調節して、システム応答の変化を観測することにより、経験的に調整できることです。

デジタル PID コントローラは周期的なサンプリング間隔で実行されるため、コントローラはシステムが正常に制御されるように十分な頻度で実行されるものとします。例えば、センサレス・ブラシレス DC モータ・アプリケーションの電流コントローラは、モータが非常に高速に変化できるため各 PWM サイクル毎に実行されます。このようなアプリケーションでの速度コントローラは、モータ速度変化は機械的時定数のため比較的低速で発生するので、中程度のイベント・レート (100 Hz) で実行されます。

誤差信号は、制御対象パラメータの所望の設定をパラメータの実際の測定値から減算してつくります。誤差の符号は、制御入力が必要とする変化の方向を表わします。

コントローラの比例項 (P) は、誤差信号に P ゲインを乗算してつくります。これにより、PID コントローラは誤差の大きさの関数としての制御応答を発生します。誤差信号が大きくなると、さらに大きな補正を加えるためにコントローラの P 項が大きくなります。

P 項の効果には、時間の経過とともに誤差全体を減らす傾向があります。ただし、P 項の効果は誤差がゼロに近付くに従って小さくなります。大部分のシステムでは、制御されるパラメータの誤差は非常にゼロに近い値ですが、収束しません。結果的に小さい定常状態誤差が残ります。コントローラの積分項 (I) は、定常状態誤差を小さく固定するために使用されます。I 項は、誤差信号の連続的な変化の合計値になります。このため、小さい定常状態誤差は時間の経過とともに累積されて大きな誤差値になります。この累積された誤差信号に I ゲイン係数を乗算すると、PID コントローラの I 出力項になります。

PID コントローラの微分項 (D) は、コントローラの速度を加速するときに使用され、誤差信号の変化率に応答します。D 項入力は、前の値から予め設定された誤差値を減算して計算されます。この差分誤差値に D ゲイン係数が乗算されて、PID コントローラの D 出力項になります。コントローラの D 項は、システム誤差が速く変化するほど大きな制御出力を発生します。

すべての PID コントローラが D 項を組込むわけではなく、I 項もそれほど使用されるわけではないことに注意してください。例えば、マイクロチップ・アプリケーション・ノート AN901 で説明したブラシレス DC モータ・アプリケーションの速度コントローラでは、モータ速度変化の応答が比較的低速であるため D 項を使っていません。この場合、D 項を使うと、PWM デューティ・サイクルの変化を過大にして、センサレス・アルゴリズムの動作に影響を与えて過大な電流トリップを発生するほどになります。

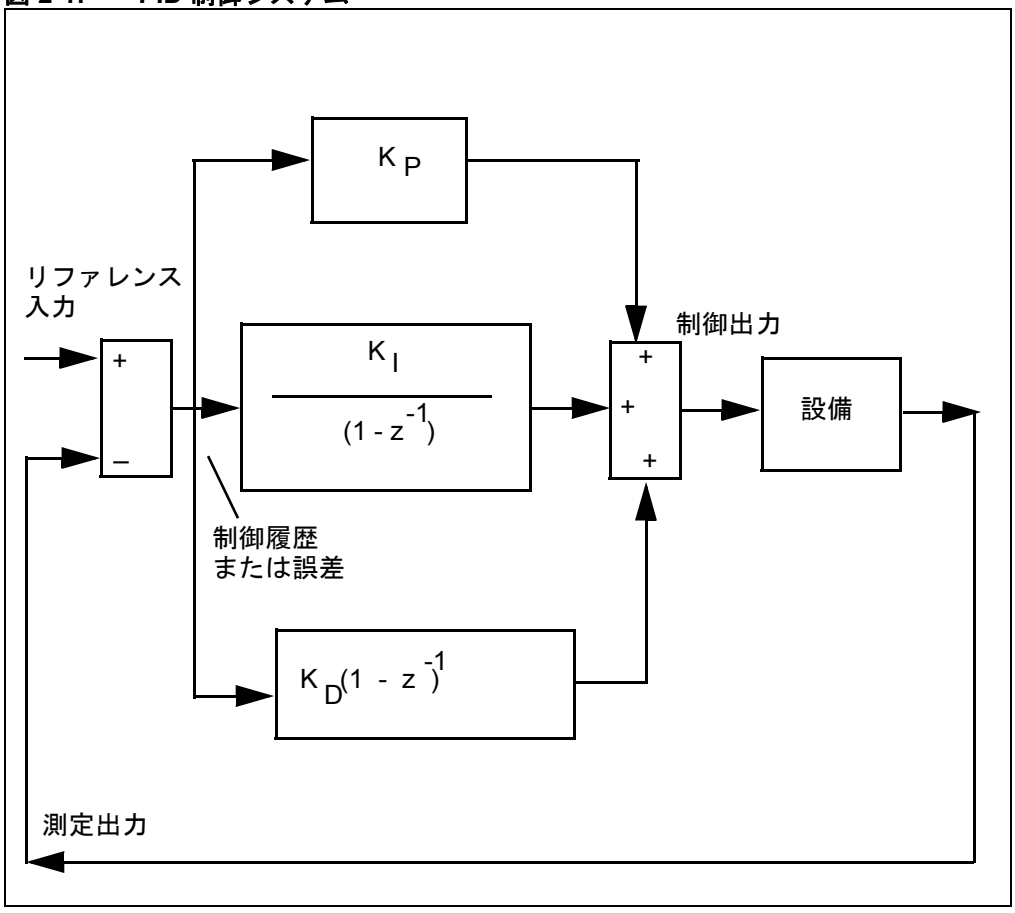
2.8.1.2 PID ゲインの調節

PID コントローラの P ゲインは、システム全体の応答を設定します。コントローラを最初に調整するとき、I ゲインと D ゲインをゼロに設定します。そして、大きなオーバーシュートまたは発振なしに、設定ポイントの変化にシステムが応答するまで P ゲインを大きくすることができます。P ゲインに小さい値を使用するとシステムの 'ルーズ' な制御になり、大きな値を使うと 'タイト' な制御になります。このポイントでは、システムは設定ポイントに収束しない可能性があります。

適切な P ゲインを選択した後、I ゲインをゆっくり増加させてシステム誤差をゼロにします。大部分のシステムでは、小さい I ゲインで済みます。I ゲインが十分な場合その効果により、P 項の動作が打ち負かされ、全体の制御応答が低速になり、設定ポイント付近でシステムが発振することがあることに注意してください。このようなことが発生した場合、I ゲインを減らし、P ゲインを増やすと、通常問題を解決することができます。

P ゲインと I ゲインを設定した後、D ゲインを設定することができます。D 項は制御変化の応答を加速しますが、コントローラ出力の非常に高速な変化を可能にするため、D 項の使用は控え目にする必要があります。この動作は、'セット・ポイント・キック' と呼ばれています。セット・ポイント・キックは、制御設定ポイントを変えたとき、システム誤差の差が瞬時に非常に大きくなるために発生します。場合によっては、システム・ハードウェアを損傷することがあります。D ゲインがゼロでシステム応答が許容できる場合には、D 項を使わないで済むと思われます。

図 2-1: PID 制御システム



2.8.1.3 PID ライブラリ関数とデータ構造体

DSP ライブラリは、PID コントローラ関数、PID (tPID*) を提供して、PID 演算を実行します。関数はヘッダー・ファイル dsp.h 内に定義されたデータ構造体を使い、この構造体は次のようになっています :

```
typedef struct {
fractional* abcCoefficients;
fractional* controlHistory;
fractional controlOutput;
fractional measuredOutput;
fractional controlReference;
} tPID;
```

PID() 関数を起動する前に、アプリケーションがタイプ tPID のデータ構造体を初期化する必要があります。次のステップで行います :

1. PID の各ゲイン値の係数の計算

   タイプ tPID のデータ構造体内の要素 abcCoefficients は、X-data 空間に配置された A、B、C の各係数を指すポインタです。これらの係数は、図 2-1 に示す PID ゲイン値 Kp、Ki、Kd から次のように導出されます :

   ```
   A = Kp + Ki + Kd
   B = -(Kp + 2*Kd)
   C = Kd
   ```

   A、B、C の各係数を導出するため、DSP ライブラリは関数 PIDCoeffCalc を提供しています。

2. PID 状態変数のクリア

   構造体の要素 controlHistory は、先頭サンプルを最新 (現在) として Y- 空間に配置された 3 個のサンプル履歴を指すポインタです。これらのサンプルは、リファレンス入力と設備機能の測定出力との間の現在と過去の差を構成します。PIDInit 関数は、controlHistory が指す要素をクリアします。また、tPID データ構造体内の controlOutput 要素もクリアします。

2.8.2 個別関数

PIDInit

説明 :	このルーチンは、Y- 空間内に配置され、かつ controlHistory により指定される 3 要素配列内の遅延線要素をクリアします。また、現在の PID 出力要素 controlOutput もクリアします。
インクルード :	dsp.h
プロトタイプ :	void PIDInit (tPID *fooPIDStruct);
引数 :	*fooPIDStruct* は、タイプ tPID の PID データ構造体を指すポインタです。
戻り値 :	void.
備考 :	
ソース・ファイル :	pid.s

PIDInit (続き)

関数プロファイル : システム・リソースの使用 :

W0..W4	使用、復旧なし
ACCA、ACCB	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 0 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
11

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
13

PIDCoeffCalc

説明 : PIDInit は、ユーザー入力の値 Kp、Ki、Kd に基づいて各 PID 係数を次のように計算します。

```
abcCoefficients[0] = Kp + Ki + Kd
abcCoefficients[1] = -(Kp + 2*Kd)
abcCoefficients[2] = Kd
```

また、このルーチンは、配列 ControlDifference 内の遅延線要素と現在の PID 出力要素 ControlOutput もクリアします。

インクルード : dsp.h

プロトタイプ :

```
void PIDCoeffCalc ( fractional *fooPIDGainCoeff,
tPID *fooPIDStruct )
```

引数 : *fooPIDGainCoeff* は、Kp、Ki、Kd 係数を [Kp、Ki、Kd] の順に含む入力配列を指すポインタ
fooPIDStruct は、タイプ tPID の PID データ構造体を指すポインタです。

戻り値 : Void.

備考 : PIDCoefficient 配列の要素は、Kp、Ki、Kd の値に応じて飽和することがあります。

ソース・ファイル : pid.s

関数プロファイル : システム・リソースの使用 :

W0..W2	使用、復旧なし
ACCA、ACCB	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
DO 命令 : 0 レベル
REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):
18

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
20

PID

説明 : PID は、データ構造体 tPID の controlOutput 要素を次のように計算します。

```
controlOutput[n] = controlOutput[N - 1]
   + controlHistory[n] * abcCoefficient[0]
   + controlHistory[N - 1] * abcCoefficient[1]
   + controlHistory[N - 2] * abcCoefficient[2]
```

ここで、

```
abcCoefficient[0] = Kp + Ki + Kd
abcCoefficient[1] = -(Kp + 2*Kd)
abcCoefficient[2] = Kd
ControlHistory[n] =
MeasuredOutput[n] - ReferenceInput[n]
```

インクルード : dsp.h

プロトタイプ :

```
extern void PID ( tPID* ooPIDStruct );
```

引数 : *fooPIDStruct* は、タイプ tPID の PID データ構造体を指すポインタです。

戻り値 : *fooPIDStruct* を指すポインタ

備考 : controlOutput 要素は、PID() ルーチンにより更新されます。controlOutput は飽和することがあります。

ソース・ファイル : pid.s

関数プロファイル : システム・リソースの使用 :

W0..W5	使用、復旧なし
W8,W10	待避、使用、復旧
ACCA	使用、復旧なし
CORCON	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :

DO 命令 : 0 レベル

REPEAT 命令 : なし

プログラム・ワード数 (24 ビット命令):

28

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):

30

2.9 その他の関数

このセクションでは、DSP ライブラリが提供するその他の有用な関数について説明します。

2.9.1 個別関数

Fract2Float

説明 : Fract2Float は、1.15 小数値を IEEE 浮動小数値へ変換します。

インクルード : dsp.h

プロトタイプ :
```
extern float Fract2Float (
    fractional aVal
);
```

引数 : *aVal* 暗黙的な範囲 [-1,(+ 1 - 2^{-15})] の 1.15 小数値

戻り値 : 範囲 [-1, (+ 1 - 2^{-15})] の IEEE 浮動小数値

備考 : なし

ソース・ファイル : flt2frct.c

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W14	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
なし

プログラム・ワード数 (24 ビット命令):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

Float2Fract

説明 : Float2Fract は、IEEE 浮動小数値を 1.15 小数値へ変換します。

インクルード : dsp.h

プロトタイプ :
```
extern fractional Float2Fract (
    float aVal
);
```

引数 : *aVal* 範囲 [-1,(+ 1 - 2^{-15})] の浮動小数値

戻り値 : 範囲 [-1, (+ 1 - 2^{-15})] の 1.15 小数値

備考 : 変換は、不偏向のまるめ処理と飽和処理を使って実行します。

ソース・ファイル : flt2frct.c

Float2Fract (続き)

関数プロファイル : システム・リソースの使用 :

W0..W7	使用、復旧なし
W8..W14	待避、使用、復旧

DO 命令と REPEAT 命令の使用 :
なし

プログラム・ワード数 (24 ビット命令):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

サイクル数 (C 関数コールとリターン・オーバーヘッドを含む):
詳細については、pic30_tools¥src¥dsp 内のファイル "readme.txt" を参照してください。

第 3 章 . 16 ビット・ペリフェラル・ライブラリ

3.1 序論

この章は、16 ビットペリフェラル・ライブラリに含まれている関数とマクロをドキュメント化したものです。使用例も記載してあります。

各ライブラリ関数またはマクロのコード・サイズは、Program Files¥Microchip¥MPLAB C30¥src¥peripheral 内のファイル readme.txt に記載してあります。

3.1.1 アセンブリ・コード・アプリケーション

これらのライブラリと関連ヘッダー・ファイルの無償バージョンは、マイクロチップ社のウエブサイトから提供しています。ソース・コードも添付されています。

3.1.2 C コード・アプリケーション

MPLAB C30 C コンパイラのインストール・ディレクトリ (c:¥Program Files¥Microchip¥MPLAB C30) には、ライブラリ関連ファイルの次のサブディレクトリが含まれています :

- lib—16 ビット・ペリフェラル・ライブラリ・ファイル
- src¥ peripheral—ライブラリを再ビルドする際に使うライブラリ関数のソース・コードとバッチ・ファイル
- support ¥h—ライブラリのヘッダー・ファイル

3.1.3 本章の構成

本章は次のように構成されています。

- 16 ビット・ペリフェラル・ライブラリの使い方

ソフトウェア関数

- 外部 LCD 関数

ハードウェア関数

- CAN 関数
- ADC12 関数
- ADC10 関数
- タイマ関数
- リセット / 制御関数
- I/O ポート関数
- 入力キャプチャ関数
- 出力コンペア関数
- UART 関数
- DCI 関数
- SPI 関数
- QEI 関数
- PWM 関数
- I2C™ 関数

3.2 16 ビット・ペリフェラル・ライブラリの使い方

各ペリフェラル・モジュールに対するプロセッサ固有のライブラリ・ファイルとヘッダー・ファイルを必要とする 16 ビット・ペリフェラル・ライブラリを使うアプリケーションのビルド

各ペリフェラルに対して、対応するヘッダー・ファイルは、ライブラリが使用するすべての関数プロトタイプ #defines と typedefs を提供します。アーカイブされたライブラリ・ファイルには、各ライブラリ関数の個々のオブジェクト・ファイルがすべて含まれています。

ヘッダー・ファイルは、*peripheral.h* の形式を持ちます。ここで、*peripheral* = 使用する特定のペリフェラルの名前です (例えば、CAN の場合は can.h)。

ライブラリ・ファイルは libp*Device-omf*.a の形式を持ちます。ここで、*Device* = 16 ビット・デバイスの名前です (例えば、dsPIC30F6014 デバイスの場合は libp30F6014-coff.a)。OMF 固有のライブラリの詳細については、**セクション 1.2「OMF 固有のライブラリ / スタートアップ・モジュール」**を参照してください。

アプリケーションをコンパイルするときは、ライブラリの関数をコールしているすべてのソース・ファイル、またはそのシンボルまたは typedef を使用しているすべてのソース・ファイルがこのヘッダー・ファイルを参照します (#include を使用)。アプリケーションをリンクするときは、ライブラリ・ファイルをリンカーに対する入力として使って (-- library または -l linker スイッチを使用)、アプリケーションで使われている関数がアプリケーションにリンクできるようにする必要があります。

バッチ・ファイル makeplib.bat は、ライブラリをリメークするときに使用できます。デフォルト動作は、サポートされているすべてのターゲット・プロセッサを対象にペリフェラル・ライブラリをビルドするためのものですが、コマンドラインから名前を指定してビルドする特定のプロセッサを選択することができます。例えば :

```
makeplib.bat 30f6014
```

または

```
makeplib.bat 30F6014
```

これにより、dsPIC30F6014 デバイス用のライブラリが再ビルドされます。

3.3 外部 LCD 関数

このセクションには、P-tec PCOG1602B LCD コントローラとのインターフェースに使う個別関数の一覧と、このセクション内の各関数の使用例を記載します。関数はマクロとして構成可能です。

外部 LCD 関数は、次のデバイスのみをサポートしています :

- dsPIC30F5011
- dsPIC30F5013
- dsPIC30F6010
- dsPIC30F6011
- dsPIC30F6012
- dsPIC30F6013
- dsPIC30F6014

3.3.1 個別関数

BusyXLCD

説明 : この関数は、P-tec PCOG1602B LCD コントローラのビジー・フラグを調べます。

インクルード : `xlcd.h`

プロトタイプ :
```
char BusyXLCD(void);
```

引数 : なし

戻り値 : LCD コントローラがビジーでコマンドを受け付けられないとき、'1' が返されます。
LCD が次のコマンドを受け付け可能なとき、'0' が返されます。

備考 : この関数は、P-tec PCOG1602B LCD コントローラのビジー・フラグのステータスを返します。

ソース・ファイル : `BusyXLCD.c`

コード例 :
```
while(BusyXLCD());
```

OpenXLCD

説明 : この関数は I/O ピンを設定し、P-tec PCOG1602B LCD コントローラを初期化します。

インクルード : `xlcd.h`

プロトタイプ :
```
void OpenXLCD (unsigned char lcdtype);
```

引数 :

lcdtype 次のように設定される LCD コントローラのパラメータを含みます :

インターフェースのタイプ
```
FOUR_BIT
EIGHT_BIT
```

ライン数
```
SINGLE_LINE
TWO_LINE
```

セグメント・データの転送方向
```
SEG1_50_SEG51_100
SEG1_50_SEG100_51
SEG100_51_SEG50_1
SEG100_51_SEG1_50
```

COM データの転送方向
```
COM1_COM16
COM16_COM1
```

戻り値 : なし

備考 : この関数は、P-tec PCOG1602B LCD コントローラの制御に使用する I/O ピンを設定します。また、LCD コントローラの初期化も行います。外部 LCD が正しく動作するために設定する必要のある I/O ピン定義を次に示します :

OpenXLCD (続き)

コントロール I/O ピンの定義

RW_PIN	PORTxbits.Rx?
TRIS_RW	TRISxbits.Rx?
RS_PIN	PORTxbits.Rx?
TRIS_RS	TRISxbits.Rx?
E_PIN	PORTxbits.Rx?
TRIS_E	TRISxbits.Rx?

ここで、x は PORT、? はピン番号です。

データ・ピンの定義

DATA_PIN_?	PORTxbits.RD?
TRIS_DATA_PIN_?	TRISxbits.TRISD?

ここで、x は PORT、? はピン番号です。

データ・ピンとしては、1 つのポートまたは複数のポートのピンを指定することができます。

コントロール・ピンとしては、任意のポートのピンが指定でき、同じポートである必要はありません。データ・インターフェースは、4 ビットまたは 8 ビットとして定義する必要があります。ヘッダー・ファイル xlcd.h 内に #define EIGHT_BIT_INTERFACE が含まれている場合に、8 ビット・インターフェースが定義され、この define が含まれていない場合には、4 ビット・インターフェースがインクルードされます。

ユーザーはこれらの定義を行った後に、アプリケーション・コードをコンパイルして、リンクに使うオブジェクトを生成する必要があります。

また、この関数は特定の遅延に対して次の 3 つの外部ルーチンも必要とします :

DelayFor18TCY()	18 Tcy 遅延
DelayPORXLCD()	15ms 遅延
DelayXLCD()	5ms 遅延
Delay100XLCD()	100Tcy 遅延

ソース・ファイル : openXLCD.c

コード例 :
```
OpenXLCD(EIGHT_BIT & TWO_LINE
        & SEG1_50_SEG51_100 &COM1_COM16);
```

putsXLCD
putrsLCD

説明 : この関数は、文字列を P-tec PCOG1602B LCD コントローラに書き込みます。

インクルード : xlcd.h

プロトタイプ :
```
void putsXLCD (char *buffer);
void putrsXLCD (const rom char *buffer);
```

引数 : *buffer* LCD コントローラへ書き込む文字を指すポインタ

戻り値 : なし

備考 : これらの関数は、文字列内で NULL 文字に遭遇するまでバッファ内の文字列を P-tec PCOG1602B LCD コントローラへ書き込みます。
P-tec PCOG1602B LCD コントローラへ書き込まれるデータを連続表示するときは、ディスプレイを Shift モードに設定する必要があります。

ソース・ファイル : PutsXLCD.c
PutrsXLCD.c

コード例 :
```
char display_char[13];
putsXLCD(display_char);
```

ReadAddrXLCD

説明 :	この関数は、P-tec PCOG1602B LCD コントローラからアドレス・バイトを読み出します。
インクルード :	xlcd.h
プロトタイプ :	unsigned char ReadAddrXLCD (void);
引数 :	なし
戻り値 :	この関数は、バイトの下位 7 ビットに配置されている 7 ビット・アドレスと 8 番目のビットであるビジー・ステータス・フラグで構成される 8 ビットを返します。
備考 :	この関数は、P-tec PCOG1602B LCD コントローラからアドレス・バイトを読み出します。ユーザーは最初に BusyXLCD() 関数を呼び出して、LCD コントローラのビジーをチェックする必要があります。コントローラから読み出されたアドレスは、呼び出した前の Set??RamAddr() 関数に応じて、キャラクタ・ジェネレータ RAM またはディスプレイ・データ RAM 用です。ここで、?? としては CG または DD が指定可能です。
ソース・ファイル :	ReadAddrXLCD.c

コード例 :	char address;
	while(BusyXLCD());
	address = ReadAddrXLCD();

ReadDataXLCD

説明 :	この関数は、P-tec PCOG1602B LCD コントローラからデータ・バイトを読み出します。
インクルード :	xlcd.h
プロトタイプ :	char ReadDataXLCD (void);
引数 :	なし
備考 :	この関数は、P-tec PCOG1602B LCD コントローラからデータ・バイトを読み出します。ユーザーは最初に BusyXLCD() 関数を呼び出して、LCD コントローラのビジーをチェックする必要があります。コントローラから読み出されたデータは、呼び出した前の Set??RamAddr() 関数に応じて、キャラクタ・ジェネレータ RAM またはディスプレイ・データ RAM 用です。ここで、?? としては CG または DD が指定可能です。
戻り値 :	この関数は、アドレスで指定された 8 ビット・データ値を返します。
ソース・ファイル :	ReadDataXLCD.c

コード例 :	char data;
	while (BusyXLCD());
	data = ReadDataXLCD();

SetCGRamAddr

説明: この関数は、キャラクタ・ジェネレータのアドレスを設定します。

インクルード: xlcd.h

プロトタイプ: void SetCGRamAddr (unsigned char *CGaddr*);

引数: *CGaddr* キャラクタ・ジェネレータ・アドレス

戻り値: なし

備考: この関数は、P-tec PCOG1602B LCD コントローラのキャラクタ・ジェネレータ・アドレスを設定します。ユーザーは最初に BusyXLCD() 関数を呼び出して、コントローラのビジーをチェックする必要があります。

ソース・ファイル: SetCGRamAddr.c

コード例:

```
char cgaddr = 0x1F;
while (BusyXLCD());
SetCGRamAddr(cgaddr);
```

SetDDRamAddr

説明: この関数は、ディスプレイ・データ・アドレスを設定します。

インクルード: xlcd.h

プロトタイプ: void SetDDRamAddr (unsigned char *DDaddr*);

引数: *DDaddr* ディスプレイ・データ・アドレス

戻り値: なし

備考: この関数は、P-tec PCOG1602B LCD コントローラのディスプレイ・データ・アドレスを設定します。ユーザーは最初に BusyXLCD() 関数を呼び出して、コントローラのビジーをチェックする必要があります。

ソース・ファイル: SetDDRamAddr.c

コード例:

```
char ddaddr = 0x10;
while (BusyXLCD());
SetDDRamAddr(ddaddr);
```

WriteDataXLCD

説明: この関数は、P-tec PCOG1602B LCD コントローラへデータ・バイト (1 文字) を書き込みます。

インクルード: xlcd.h

プロトタイプ: void WriteDataXLCD (char *data*);

引数: *data* データ値としては任意の 8 ビット値が可能ですが、P-tec PCOG1602B LCD コントローラの文字 RAM テーブルに対応している必要があります。

戻り値: なし

備考: この関数は、データ・バイトを P-tec PCOG1602B LCD コントローラへ書き込みます。ユーザーは最初に BusyXLCD() 関数を呼び出して、LCD コントローラのビジーをチェックする必要があります。
コントローラから読み出されたデータは、呼び出した前の Set??RamAddr() 関数に応じて、キャラクタ・ジェネレータ RAM またはディスプレイ・データ RAM 用です。ここで、?? としては CG または DD が指定可能です。

ソース・ファイル: WriteDataXLCD.c

コード例: WriteDataXLCD(0x30);

WriteCmdXLCD

説明 : この関数は、コマンドを P-tec PCOG1602B LCD コントローラへ書き込みます。

インクルード : xlcd.h

プロトタイプ : void WriteCmdXLCD (unsigned char *cmd*);

引数 : *cmd* 次のように設定される LCD コントローラのパラメータを含みます :

インターフェースのタイプ
FOUR_BIT
EIGHT_BIT

ライン数
SINLE_LINE
TWO_LINE

セグメント・データの転送方向
SEG1_50_SEG51_100
SEG1_50_SEG100_51
SEG100_51_SEG50_1
SEG100_51_SEG1_50

COM データの転送方向
COM1_COM16
COM16_COM1

ディスプレイの On/Off 制御
DON
DOFF
CURSOR_ON
CURSOR_OFF
BLINK_ON
BLINK_OFF

カーソルまたはディスプレイのシフト定義
SHIFT_CUR_LEFT
SHIFT_CUR_RIGHT
SHIFT_DISP_LEFT
SHIFT_DISP_RIGHT

戻り値 : なし

備考 : この関数は、コマンド・バイトを P-tec PCOG1602B LCD コントローラへ書き込みます。ユーザーは最初に BusyXLCD() 関数を呼び出して、LCD コントローラのビジーをチェックする必要があります。

ソース・ファイル : WriteCmdXLCD.c

コード例 :

```
while(BusyXLCD());
WriteCmdXLCD(EIGHT_BIT & TWO_LINE);
WriteCmdXLCD(DON);
WriteCmdXLCD(SHIFT_DISP_LEFT);
```

3.3.2 使用例

```
#define _ _dsPIC30F6014_ _
#include <p30fxxxx.h>
#include <xlcd.h>
/* holds the address of message */
char * buffer;
char data ;
char mesg1[] = {'H','A','R','D','W','A','R','E','¥0'};
char mesg2[] = {'P','E','R','I','P','H','E','R','A','L'
                ' ', 'L','I','B',' ','¥0'};
int main(void)
{
/* Set 8bit interface and two line display */
    OpenXLCD(EIGHT_BIT & TWO_LINE & SEG1_50_SEG51_100
             & COM1_COM16);
/* Wait till LCD controller is busy */
    while(BusyXLCD());
/* Turn on the display */
    WriteCmdXLCD(DON & CURSOR_ON & BLINK_OFF);
    buffer = mesg1;
    PutsXLCD(buffer);
    while(BusyXLCD());
/* Set DDRam address to 0x40 to dispaly data in the second line */
    SetDDRamAddr(0x40);
    while(BusyXLCD());
    buffer = mesg2;
    PutsXLCD(buffer);
    while(BusyXLCD());
    return 0;
}
```

3.4 CAN 関数

このセクションには、CAN 用個別関数の一覧と、各関数の使用例を記載します。関数はマクロとして構成可能です。

3.4.1 個別関数

CAN1AbortAll
CAN2AbortAll

説明 :	この関数は、すべての待ち状態の転送の中止を初期化します。
インクルード :	`can.h`
プロトタイプ :	`void CAN1AbortAll(void);` `void CAN2AbortAll(void);`
引数 :	なし
戻り値 :	なし
備考 :	この関数は、CiCTRL レジスタ内の ABAT ビットをセットします。したがって、すべての待ち状態の転送の中止を初期化します。ただし、進行中の転送は中止されません。メッセージ転送が正常に中止された場合、このビットはハードウェアによりクリアされます。
ソース・ファイル :	`CAN1AbortAll.c` `CAN2AbortAll.c`
コード例 :	`CAN1AbortAll();`

CAN1GetRXErrorCount
CAN2GetRXErrorCount

説明 :	この関数は、受信エラー・カウント値を返します。
インクルード :	`can.h`
プロトタイプ :	`unsigned char CAN1GetRXErrorCount(void);` `unsigned char CAN2GetRXErrorCount(void);`
引数 :	なし
戻り値 :	8 ビットの CiRERRCNT 値
備考 :	この関数は、受信エラー・カウントを表わす CiRERRCNT 値 (CiEC レジスタの下位バイト) を返します。
ソース・ファイル :	`CAN1GetRXErrorCount.c` `CAN2GetRXErrorCount.c`
コード例 :	`unsigned char rx_error_count;` `rx_error_count = CAN1GetRXErrorCount();`

CAN1GetTXErrorCount
CAN2GetTXErrorCount

説明 :	この関は、送信エラー・カウント値を数返します。
インクルード :	`can.h`
プロトタイプ :	`unsigned char CAN1GetTXErrorCount(void);` `unsigned char CAN2GetTXErrorCount(void);`
引数 :	なし
戻り値 :	8 ビットの CiTERRCNT 値

CAN1GetTXErrorCount
CAN2GetTXErrorCount (続き)

備考 : この関数は、送信エラー・カウントを表わす CiTERRCNT 値 (CiEC レジスタの下位バイト) を返します。

ソース・ファイル :
```
CAN1GetTXErrorCount.c
CAN2GetTXErrorCount.c
```

コード例 :
```
unsigned char tx_error_count;
tx_error_count = CAN1GetTXErrorCount();
```

CAN1IsBusOff
CAN2IsBusOff

説明 : この関数は、CAN ノードが BusOff モードであるか否かを調べます。

インクルード : `can.h`

プロトタイプ :
```
char CAN1IsBusOff(void);
char CAN2IsBusOff(void);
```

引数 : なし

戻り値 : TXBO 値が '1' の場合、'1' を返して、転送エラーのためバスがターンオフされていることを表示します。
TXBO 値が '0' の場合、'0' を返して、バスがターンオフされていないことを表示します。

備考 : この関数は、CiINTF レジスタ内の TXBO ビットのステータスを返します

ソース・ファイル :
```
CAN1IsBusOff.c
CAN2IsBusOff.c
```

コード例 :
```
while(CAN1IsBusOff());
```

CAN1IsRXReady
CAN2IsRXReady

説明 : この関数は、受信バッファ・フル・ステータスを返します。

インクルード : `can.h`

プロトタイプ :
```
char CAN1IsRXReady(char);
char CAN2IsRXReady(char);
```

引数 : *buffno*　ステータスを要求する受信バッファを指定します。

戻り値 : RXFUL が 1 の場合、受信バッファには受信メッセージが存在していることを示します。
RXFUL が 0 の場合、受信バッファが空いていて、新しいメッセージが受信できることを示します。

備考 : この関数は、受信コントロール・レジスタの RXFUL ビットのステータスを返します。

ソース・ファイル :
```
CAN1IsRXReady.c
CAN2IsRXReady.c
```

コード例 :
```
char rx_1_status;
rx_1_status = CAN1IsRXReady(1);
```

CAN1IsRXPassive
CAN2IsRXPassive

説明 :	この関数は、レシーバがエラー・パッシブ状態にあるか否かを調べます。
インクルード :	can.h
プロトタイプ :	char CAN1IsRXPassive(void); char CAN2IsRXPassive(void);
引数 :	なし
戻り値 :	RXEP 値が '1' の場合、'1' を返して、ノードが受信エラーのためパッシブになることを表示します。 RXEP 値が '0' の場合、'0' を返して、バスにエラーないことを表示します。
備考 :	この関数は、CiINTF レジスタ内の RXEP ビットのステータスを返します
ソース・ファイル :	CAN1IsRXPassive.c CAN2IsRXPassive.c
コード例 :	char rx_bus_status; rx_bus_status = CAN1IsRXPassive();

CAN1IsTXPassive
CAN2IsTXPassive

説明 :	この関数は、トランスミッタがエラー・パッシブ状態にあるか否かを調べます。
インクルード :	can.h
プロトタイプ :	char CAN1IsTXPassive(void); char CAN2IsTXPassive(void);
引数 :	なし
戻り値 :	TXEP 値が '1' の場合、'1' を返して、送信バスがエラーのためバスがパッシブになることを表示します。 TXEP 値が '0' の場合、'0' を返して、送信バスにエラーないことを表示します。
備考 :	この関数は、CiINTF レジスタ内の TXEP ビットのステータスを返します
ソース・ファイル :	CAN1IsTXPassive.c CAN2IsTXPassive.c
コード例 :	char tx_bus_status; tx_bus_status = CAN1IsTXPassive();

CAN1IsTXReady
CAN2IsTXReady

説明 :	この関数はトランスミッタ・ステータス返し、CAN ノードが次の転送の準備ができているか否かを表示します。
インクルード :	can.h
プロトタイプ :	char CAN1IsTXReady(*char*); char CAN2IsTXReady(*char*);

CAN1IsTXReady
CAN2IsTXReady (続き)

引数 : *buffno* ステータスを要求する送信バッファを指定します。

戻り値 : TXREQ が '1' の場合に '0' を返して、送信バッファがエンプティでないことを表示します。
TXREQ が '0' の場合に '1' を返して、送信バッファがエンプティであり、トランスミッタが次の転送の準備ができていることを表示します。

備考 : この関数は、送信コントロール・レジスタの TXREQ Status ビットの反転を返します。

ソース・ファイル :
```
CAN1IsTXReady.c
CAN2IsTXReady.c
```

コード例 :
```
char tx_2_status;
tx_2_status = CAN1IsTXReady(2);
```

CAN1ReceiveMessage
CAN2ReceiveMessage

説明 : この関数は、受信バッファからデータを読み出します。

インクルード :
```
can.h
```

プロトタイプ :
```
void CAN1ReceiveMessage(unsigned char *
 data, unsigned char datalen, char MsgFlag);
void CAN2ReceiveMessage(unsigned char *
 data, unsigned char datalen,char MsgFlag);
```

引数 :

data 受信したデータを格納するロケーションを指すポインタ

datalen データのバイト数

MsgFlag データが受信されるバッファ番号
'1' の場合、データを CiRX1B1 から CiRX1B4 へ読み込み
'0' またはその他の場合、データを CiRX0B1 から CiRX0B4 へ読み込み

備考 : この関数は、受信したデータを入力パラメータ・データで指定するロケーションへ読み込みます。

戻り値 : なし

ソース・ファイル :
```
CAN1ReceiveMessage.c
CAN2ReceiveMessage.c
```

コード例 :
```
unsigned char*rx_data;
CAN1ReceiveMessage(rx_data, 5, 0);
```

CAN1SendMessage
CAN2SendMessage

説明 : この関数は、送信するデータを TX レジスタへ書き込み、データ長を設定し転送を開始させます。

インクルード : can.h

プロトタイプ :
```
void CAN1SendMessage(unsigned int sid,
 unsigned long eid, unsigned char *data,
 unsigned char datalen, char MsgFlag);
void CAN2SendMessage(unsigned int sid,
 unsigned long eid, unsigned char *data,
 unsigned char datalen, char MsgFlag);
```

引数 :

sid	CiTXnSID レジスタへ書き込む 16 ビット値 CAN_TX_SID(x)　x は、必要な SID 値。 リモート要求の置き換え CAN_SUB_REM_TX_REQ CAN_SUB_NOR_TX_REQ メッセージ ID タイプ CAN_TX_EID_EN CAN_TX_EID_DIS
eid	CiTXnEID レジスタと CiTXnDLC レジスタに書き込まれる 32 ビット値 CAN_TX_EID(x)　x は必要な EID 値。 リモート要求の置き換え CAN_REM_TX_REQ CAN_NOR_TX_REQ
data	送信データが格納されるロケーションを指すポインタ
datalen	送信データのバイト数。
MsgFlag	データ送信元のバッファ番号 ('0'、'1' または '2') '1' の場合、データは CiTX1B1 ～ CiTX1B4 へ書き込まれます。 '2' の場合、データは CiTX2B1 ～ CiTX2B4 へ書き込まれます。 '0' またはその他の場合、データは CiTX0B1 ～ CiTX0B4 へ書き込まれます。

戻り値 : なし

備考 : この関数は、識別子の値を SID レジスタと EID レジスタへ書き込み、送信データを TX レジスタへ設定し、データ長を設定し、TXREQ ビットをセットすることにより転送を開始させます。

ソース・ファイル :
CAN1SendMessage.c
CAN2SendMessage.c

コード例 :
```
CAN1SendMessage((CAN_TX_SID(1920)) &
(CAN_TX_EID_EN) & (CAN_SUB_NOR_TX_REQ),
(CAN_TX_EID(12344)) & (CAN_NOR_TX_REQ),
Txdata, datalen, tx_rx_no);
```

CAN1SetFilter
CAN2SetFilter

説明: この関数は、指定されたフィルタに対するアクセプタンス・フィルタ値 (SID と EID) を設定します。

インクルード: can.h

プロトタイプ:
```
void CAN1SetFilter(char filter_no, unsigned int sid,
 unsigned long eid);
void CAN2SetFilter(char filter_no, unsigned int sid,
 unsigned long eid);
```

引数:

filter_no	新しいフィルタ値を設定するフィルタ (0、1、2、3、4 または 5)
sid	CiRXFnSID レジスタに書き込む 16 ビット値 CAN_FILTER_SID(x)　x は必要な SID 値。 受信メッセージのタイプ CAN_RX_EID_EN CAN_RX_EID_DIS
eid	CiRXFnEIDH レジスタと CiRXFnEIDL レジスタへ書き込む 32 ビット値。 CAN_FILTER_EID(x)　x は必要な EID 値。

戻り値: なし

備考: この関数は、*filter_no* により指定されたフィルタに応じて、*sid* の 16 ビット値を CiRXFnSID レジスタへ、または *eid* の 32 ビット値を CiRXFnEIDH レジスタと CiRXFnEIDL レジスタへそれぞれ書き込みます。
デフォルトは、フィルタ 0 です。

ソース・ファイル: CAN1SetFilter.c
CAN2SetFilter.c

コード例:
```
CAN1SetFilter(1, CAN_FILTER_SID(7) &
CAN_RX_EID_EN, CAN_FILTER_EID(3));
```

CAN1SetMask
CAN2SetMask

説明: この関数は、指定されたマスクに対するアクセプタンス・マスク値 (SID と EID) を設定します。

インクルード: can.h

プロトタイプ:
```
void CAN1SetMask(char mask_no, unsigned int sid,
 unsigned long eid);
void CAN2SetMask(char mask_no, unsigned int sid,
 unsigned long eid);
```

引数:

mask_no	マスク値を設定するマスク ('0' または '1')。
sid	CiRXMnSID レジスタに書き込む 16 ビット値 CAN_MASK_SID(x)　x は必要な SID 値。 フィルタ内で指定される一致 / 無視メッセージ・タイプ CAN_MATCH_FILTER_TYPE CAN_IGNORE_FILTER_TYPE
eid	CiRXMnEIDH レジスタと CiRXMnEIDL レジスタへ書き込む 32 ビット値 CAN_MASK_EID(x)　x は必要な EID 値。

戻り値: なし

CAN1SetMask
CAN2SetMask (続き)

備考 : この関数は、*mask_no* により指定されたマスクに応じて、*sid* の 16 ビット値を CiRXFnSID レジスタへ、または *eid* の 32 ビット値を CiRXFnEIDH レジスタと CiRXFnEIDL レジスタへそれぞれ書き込みます。
デフォルトは、フィルタ 0 です。

ソース・ファイル : CAN1SetMask.c
CAN2SetMask.c

コード例 :
```
CAN1SetMask(1, CAN_MASK_SID(7) &
CAN_MATCH_FILTER_TYPE, CAN_MASK_EID(3));
```

CAN1SetOperationMode
CAN2SetOperationMode

説明 : この関数は、CAN モジュールを設定します。

インクルード : can.h

プロトタイプ :
```
void CAN1SetOperationMode(unsigned int config);
void CAN2SetOperationMode(unsigned int config);
```

引数 : *config* CiCTRL レジスタへ書き込まれる 16 ビット値、次の定義の組み合わせ :

CAN_IDLE_CON　アイドル・モードでの CAN On
CAN_IDLE_STOP　アイドル・モードでの CAN Stop

CAN_MASTER クロック _1　FCAN は 1FCY
CAN_MASTER クロック _0　FCAN は 4 FCY

CAN 動作モード
CAN_REQ_OPERMODE_NOR
CAN_REQ_OPERMODE_DIS
CAN_REQ_OPERMODE_LOOPBK
CAN_REQ_OPERMODE_LISTENONLY
CAN_REQ_OPERMODE_CONFIG
CAN_REQ_OPERMODE_LISTENALL

CAN キャプチャのイネーブル / ディスエーブル
CAN_CAPTURE_EN
CAN_CAPTURE_DIS

戻り値 : なし

備考 : この関数は、CiCTRL の -CSIDL ビット、REQOP<2:0> ビット、CANCKS ビットを設定します。

ソース・ファイル : CAN1SetOperationMode.c
CAN2SetOperationMode.c

コード例 :
```
CAN1SetOperationMode(CAN_IDLE_STOP &
CAN_MASTERCLOCK_0 & CAN_REQ_OPERMODE_DIS &
CAN_CAPTURE_DIS);
```

CAN1SetOperationModeNoWait
CAN2SetOperationModeNoWait

説明 : この関数は、待ち状態の転送を中止し、CAN モジュールを設定します。

インクルード : can.h

プロトタイプ :
```
void CAN1SetOperationModeNoWait(
 unsigned int config);
void CAN2SetOperationModeNoWait(
 unsigned int config);
```

引数 : *config* CiCTRL レジスタへ書き込まれる 16 ビット値、次の定義の組み合わせ :

CAN_IDLE_CON_NO_WAIT アイドル・モードでの CAN On
CAN_IDLE_STOP_NO_WAIT アイドル・モードでの CAN Stop

CAN_MASTER クロック _1_NO_WAIT FCAN は 1FCY
CAN_MASTER クロック _0_NO_WAIT FCAN は 4 FCY

CAN 動作モード
CAN_REQ_OPERMODE_NOR_NO_WAIT
CAN_REQ_OPERMODE_DIS_NO_WAIT
CAN_REQ_OPERMODE_LOOPBK_NO_WAIT
CAN_REQ_OPERMODE_LISTENONLY_NO_WAIT
CAN_REQ_OPERMODE_CONFIG_NO_WAIT
CAN_REQ_OPERMODE_LISTENALL_NO_WAIT

CAN キャプチャのイネーブル / ディスエーブル
CAN_CAPTURE_EN_NO_WAIT
CAN_CAPTURE_DIS_NO_WAIT

戻り値 : なし

備考 : この関数は、アボート・ビットをセットし、すべての待ち状態の転送を中止させて、CiCTRL の CSIDL ビット、REQOP<2:0> ビット、CANCKS ビットを設定します。

ソース・ファイル : CAN1SetOperationModeNoWait.c
CAN2SetOperationModeNoWait.c

コード例 :
```
CAN1SetOperationModeNoWait(CAN_IDLE_CON &
CAN_MASTERCLOCK_1 & CAN_REQ_OPERMODE_LISTEN &
CAN_CAPTURE_DIS_NO_WAIT);
```

CAN1SetRXMode
CAN2SetRXMode

説明 : この関数は、CAN レシーバを設定します。

インクルード : can.h

プロトタイプ :
```
void CAN1SetRXMode(char buffno, unsigned int
 config);
void CAN2SetRXMode(char buffno, unsigned int
 config);
```

引数 : *buffno* 設定するコントロール・レジスタを指定します。

config CiRXnCON レジスタに書き込む値。次の定義の組み合わせ :

RXFUL ビットのクリア
CAN_RXFUL_CLEAR

ダブル・バッファのイネーブル / ディスエーブル
CAN_BUF0_DBLBUFFER_EN
CAN_BUF0_DBLBUFFER_DIS

CAN1SetRXMode
CAN2SetRXMode (続き)

戻り値 : なし

備考 : この関数は、CiRXnCON レジスタの次のビットを設定します : RXRTR、RXFUL (only 0)、RXM<1:0>、DBEN

ソース・ファイル :
```
CAN1SetRXMode.c
CAN2SetRXMode.c
```

コード例 :
```
CAN1SetRXMode(0,CAN_RXFUL_CLEAR &
CAN_BUF0_DBLBUFFER_EN);
```

CAN1SetTXMode (function)
CAN2SetTXMode

説明 : この関数は、CAN トランスミッタ・モジュールを設定します。

インクルード : can.h

プロトタイプ :
```
void CAN1SetTXMode(char buffno, unsigned int
 config);
void CAN2SetTXMode(char buffno, unsigned int
 config);
```

引数 :

buffno 設定するコントロール・レジスタを指定します。

config CiTXnCON レジスタに書き込む値。次の定義の組み合わせ :

メッセージ送信要求
```
CAN_TX_REQ
CAN_TX_STOP_REQ
```
メッセージ転送優先順位
```
CAN_TX_PRIORITY_HIGH
CAN_TX_PRIORITY_HIGH_INTER
CAN_TX_PRIORITY_LOW_INTER
CAN_TX_PRIORITY_LOW
```

戻り値 : なし

備考 : この関数は、CiTXnCON レジスタの次のビットを設定します : TXRTR、TXREQ、DLC、TXPRI<1:0>

ソース・ファイル :
```
CAN1SetTXMode.c
CAN2SetTXMode.c
```

コード例 :
```
CAN1SetTXMode(1, CAN_TX_STOP_REQ &
CAN_TX_PRIORITY_HIGH);
```

CAN1Initialize
CAN2Initialize

説明 : この関数は、CAN モジュールを設定します。

インクルード : can.h

プロトタイプ :
```
void CAN1Initialize(unsigned int config1,
 unsigned int config2);
void CAN2Initialize(unsigned int config1,
 unsigned int config2);
```

引数 :

config1 CiCFG1 レジスタに書き込む値。次の定義の組み合わせ :

同期ジャンプ幅
CAN_SYNC_JUMP_WIDTH1
CAN_SYNC_JUMP_WIDTH2
CAN_SYNC_JUMP_WIDTH3
CAN_SYNC_JUMP_WIDTH4

ボーレート・プリスケーラ
CAN_BAUD_PRE_SCALE(x) (((x – 1) & 0x3f) | 0xC0)

config2 CiCFG2 レジスタに書き込む値。次の定義の組み合わせ :

ウェイクアップ用の CAN バス・ライン・フィルタ選択
CAN_WAKEUP_BY_FILTER_EN
CAN_WAKEUP_BY_FILTER_DIS

CAN 伝搬セグメントの長さ
CAN_PROPAGATIONTIME_SEG_TQ(x)
(((x – 1) & 0x7) | 0xC7F8)

CAN フェーズ・セグメント 1 の長さ
CAN_PHASE_SEG1_TQ(x)
((((x – 1) & 0x7) * 0x8) | 0xC7C7)

CAN フェーズ・セグメント 2 の長さ
CAN_PHASE_SEG2_TQ(x)
((((x – 1) & 0x7) * 0x100) | 0xC0FF)

CAN フェーズ・セグメント 2 モード
CAN_SEG2_FREE_PROG
CAN_SEG2_TIME_LIMIT_SET

CAN バス・ラインのサンプル
CAN_SAMPLE3TIMES
CAN_SAMPLE1TIME

戻り値 : なし

備考 : この関数は、CiCFG1 レジスタと CiCFG2 レジスタの次のビットを設定します :
SJW<1:0>、BRP<5:0>、CANCAP、WAKEFIL、SEG2PH<2:0>、SEGPHTS、SAM、SEG1PH<2:0>、PRSEG<2:0>

ソース・ファイル : CAN1Initialize.c
CAN2Initialize.c

コード例 :
```
CAN1Initialize(CAN_SYNC_JUMP_WIDTH2 &
CAN_BAUD_PRE_SCALE(2),
CAN_WAKEUP_BY_FILTER_DIS &
CAN_PHASE_SEG2_TQ(5) &
CAN_PHASE_SEG1_TQ(4) &
CAN_PROPAGATIONTIME_SEG_TQ(4) &
CAN_SEG2_FREE_PROG &
CAN_SAMPLE1TIME);
```

ConfigIntCAN1
ConfigIntCAN2

説明 : この関数は、CAN の割り込みを設定します。

インクルード : can.h

プロトタイプ :
```
void ConfigIntCAN1(unsigned int config1,
unsigned int config2);
void ConfigIntCAN2(unsigned int config1,
unsigned int config2);
```

引数 :

config1 次に定義される個別割り込みイネーブル / ディスエーブル情報 :
ユーザーは、すべての個別割り込みに対してイネーブルまたはディスエーブルを指定する必要があります。

割り込みイネーブル
```
CAN_INDI_INVMESS_EN
CAN_INDI_WAK_EN
CAN_INDI_ERR_EN
CAN_INDI_TXB2_EN
CAN_INDI_TXB1_EN
CAN_INDI_TXB0_EN
CAN_INDI_RXB1_EN
CAN_INDI_RXB0_EN
```

割り込みディスエーブル
```
CAN_INDI_INVMESS_DIS
CAN_INDI_WAK_DIS
CAN_INDI_ERR_DIS
CAN_INDI_TXB2_DIS
CAN_INDI_TXB1_DIS
CAN_INDI_TXB0_DIS
CAN_INDI_RXB1_DIS
CAN_INDI_RXB0_DIS
```

config2 次のように定義される CAN 割り込みの優先順位とイネーブル / ディスエーブル情報 :

CAN 割り込みイネーブル / ディスエーブル
```
CAN_INT_ENABLE
CAN_INT_DISABLE
```

CAN 割り込み優先順位
```
CAN_INT_PRI_0
CAN_INT_PRI_1
CAN_INT_PRI_2
CAN_INT_PRI_3
CAN_INT_PRI_4
CAN_INT_PRI_5
CAN_INT_PRI_6
CAN_INT_PRI_7
```

戻り値 : なし

備考 : この関数は、CAN の割り込みを設定します。個別 CAN 割り込みをイネーブル / ディスエーブルします。また、CAN 割り込みをイネーブル / ディスエーブルして優先順位も設定します。

ソース・ファイル :
```
ConfigIntCAN1.c
ConfigIntCAN2.c
```

ConfigIntCAN1
ConfigIntCAN2 (続き)

コード例 :
```
ConfigIntCAN1(CAN_INDI_INVMESS_EN &
              CAN_INDI_WAK_DIS &
              CAN_INDI_ERR_DIS &
              CAN_INDI_TXB2_DIS  &
              CAN_INDI_TXB1_DIS  &
              CAN_INDI_TXB0_DIS &
              CAN_INDI_RXB1_DIS &
              CAN_INDI_RXB0_DIS ,
              CAN_INT_PRI_3 &
              CAN_INT_ENABLE);
```

3.4.2 個別マクロ

EnableIntCAN1
EnableIntCAN2

説明 : このマクロは、CAN 割り込みをイネーブルします。

インクルード : can.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタの CAN 割り込みイネーブル・ビットをセットします。

コード例 :
```
EnableIntCAN1;
```

DisableIntCAN1
DisableIntCAN2

説明 : このマクロは、CAN 割り込みをディスエーブルします。

インクルード : can.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタの CAN 割り込みイネーブル・ビットをクリアします。

コード例 :
```
DisableIntCAN2;
```

SetPriorityIntCAN1
SetPriorityIntCAN2

説明 : このマクロは、CAN 割り込みの優先順位をセットします。

インクルード : can.h

引数 : *priority*

備考 : このマクロは、割り込み優先順位コントロール・レジスタの CAN 割り込み優先順位ビットをセットします。

コード例 :
```
SetPriorityIntCAN1(2);
```

3.4.3 使用例

```
#define _ _dsPIC30F6014_ _
#include<p30fxxxx.h>
#include<can.h>
#define  dataarray 0x1820
int main(void)
{
    /* Length of data to be transmitted/read  */
    unsigned char datalen;
    unsigned char Txdata[] =
     {'M','I','C','R','O','C','H','I','P','¥0'};
    unsigned int TXConfig, RXConfig;
    unsigned long MaskID,MessageID;
    char FilterNo,tx_rx_no;
    unsigned char * datareceived = (unsigned char *)
        dataarray;  /* Holds the data received */
    /* Set request for configuration mode */
    CAN1SetOperationMode(CAN_IDLE_CON  &
                         CAN_MASTERCLOCK_1 &
                         CAN_REQ_OPERMODE_CONFIG &
                         CAN_CAPTURE_DIS);
    while(C1CTRLbits.OPMODE <=3);
    /* Load configuration register */
    CAN1Initialize(CAN_SYNC_JUMP_WIDTH2 &
                   CAN_BAUD_PRE_SCALE(2),
                   CAN_WAKEUP_BY_FILTER_DIS  &
                   CAN_PHASE_SEG2_TQ(5) &
                   CAN_PHASE_SEG1_TQ(4) &
                   CAN_PROPAGATIONTIME_SEG_TQ(4) &
                   CAN_SEG2_FREE_PROG &
                   CAN_SAMPLE1TIME);
    /* Load Acceptance filter register */
    FilterNo = 0;
    CAN1SetFilter(FilterNo, CAN_FILTER_SID(1920) &
                  CAN_RX_EID_EN, CAN_FILTER_EID(12345));
    /* Load mask filter register */
    CAN1SetMask(FilterNo, CAN_MASK_SID(1920) &
                CAN_MATCH_FILTER_TYPE, CAN_MASK_EID(12344));
    /* Set transmitter and receiver mode */
    tx_rx_no = 0;
    CAN1SetTXMode(tx_rx_no,
                  CAN_TX_STOP_REQ &
                  CAN_TX_PRIORITY_HIGH );
    CAN1SetRXMode(tx_rx_no,
                  CAN_RXFUL_CLEAR &
                  CAN_BUF0_DBLBUFFER_EN);
    /* Load message ID , Data into transmit buffer and set
       transmit request bit */
    datalen = 8;
    CAN1SendMessage((CAN_TX_SID(1920)) & CAN_TX_EID_EN &
                    CAN_SUB_NOR_TX_REQ,
                   (CAN_TX_EID(12344)) & CAN_NOR_TX_REQ,
                    Txdata,datalen,tx_rx_no);
```

```
    /* Set request for Loopback mode */
    CAN1SetOperationMode(CAN_IDLE_CON & CAN_CAPTURE_DIS &
                         CAN_MASTERCLOCK_1 &
                         CAN_REQ_OPERMODE_LOOPBK);
    while(C1CTRLbits.OPMODE !=2);
    /* Wait till message is transmitted completely */
    while(!CAN1IsTXReady(0))
    /* Wait till receive buffer contain valid  message */
    while(!CAN1IsRXReady(0));
    /* Read received data from receive buffer and store it into
user defined dataarray */
    CAN1ReceiveMessage(datareceived, datalen, tx_rx_no);
    while(1);
    return 0;
}
```

3.5 ADC12 関数

このセクションには、12 ADC 用個別関数の一覧と、各関数の使用例を記載します。関数はマクロとして構成可能です。

3.5.1 個別関数

BusyADC12

説明 :	この関数は、ADC 変換ステータスを返します。
インクルード :	adc12.h
プロトタイプ :	char BusyADC12(void);
引数 :	なし
戻り値 :	DONE 値が '0' の場合、'1' を返して、ADC が変換動作中でビジーであることを表示します。 DONE 値が '1' の場合、'0' を返して、ADC の変換動作が完了したことを表示します。
備考 :	この関数は ADCON1 <DONE> ビットのステータスの反転を返し、ADC が変換中か否かを表示します。
ソース・ファイル :	BusyADC12.c
コード例 :	while(BusyADC12());

CloseADC12

説明 :	この関数は、ADC モジュールをターンオフし、ADC 割り込みをディスエーブルします。
インクルード :	adc12.h
プロトタイプ :	void CloseADC12(void);
引数 :	なし
戻り値 :	なし
備考 :	この関数は ADC 割り込みをディスエーブルした後に、ADC モジュールをターンオフします。割り込みフラグ・ビット (ADIF) もクリアします。
ソース・ファイル :	CloseADC12.c
コード例 :	CloseADC12();

ConfigIntADC12

説明 :	この関数は、ADC 割り込みを設定します。	
インクルード :	adc12.h	
プロトタイプ :	void ConfigIntADC12(unsigned int *config*);	
引数 :	*config*	次のように定義される ADC 割り込み優先順位とイネーブル / ディスエーブル情報 : ADC 割り込みのイネーブル / ディスエーブル ADC_INT_ENABLE ADC_INT_DISABLE

ConfigIntADC12 (続き)

ADC 割り込みの優先順位
ADC_INT_PRI_0
ADC_INT_PRI_1
ADC_INT_PRI_2
ADC_INT_PRI_3
ADC_INT_PRI_4
ADC_INT_PRI_5
ADC_INT_PRI_6
ADC_INT_PRI_7

戻り値 : なし

備考 : この関数は、割り込みフラグ (ADIF) ビットをクリアした後に、割り込み優先順位を設定して、割り込みをイネーブル / ディスエーブルします。

ソース・ファイル : ConfigIntADC12.c

コード例 :
```
ConfigIntADC12(ADC_INT_PRI_6 &
               ADC_INT_ENABLE);
```

ConvertADC12

説明 : この関数は、A/D 変換を開始させます。

インクルード : adc12.h

プロトタイプ :
```
void ConvertADC12(void);
```

引数 : なし

戻り値 : なし

備考 : この関数は ADCON1<SAMP> ビットをクリアしてサンプリングを停止させて、変換を開始させます。
これは、ADCON1 <SSRC> ビットをクリアすることにより A/D 変換のトリガ・ソースを手動で選択した場合にのみ可能です。

ソース・ファイル : ConvertADC12.c

コード例 :
```
ConvertADC12();
```

OpenADC12

説明 : この関数は ADC を設定します。

インクルード : adc12.h

プロトタイプ :
```
void OpenADC12(unsigned int config1,
               unsigned int config2,
               unsigned int config3,
               unsigned int configport,
               unsigned int configscan)
```

引数 : *config1* ADCON1 レジスタに設定される次のように定義されたパラメータ :

モジュールの On/Off
ADC_MODULE_ON
ADC_MODULE_OFF

アイドル・モード動作
ADC_IDLE_CONTINUE
ADC_IDLE_STOP

OpenADC12 (続き)

	変換結果の出力フォーマット ADC_FORMAT_SIGN_FRACT ADC_FORMAT_FRACT ADC_FORMAT_SIGN_INT ADC_FORMAT_INTG 変換トリガ・ソース ADC_CLK_AUTO ADC_CLK_TMR ADC_CLK_INT0 ADC_CLK_MANUAL 自動サンプリング選択 ADC_AUTO_SAMPLING_ON ADC_AUTO_SAMPLING_OFF サンプルのイネーブル ADC_SAMP_ON ADC_SAMP_OFF
config2	ADCON2 レジスタに設定される次のように定義されたパラメータ : リファレンス電圧 ADC_VREF_AVDD_AVSS ADC_VREF_EXT_AVSS ADC_VREF_AVDD_EXT ADC_VREF_EXT_EXT スキャン選択 ADC_SCAN_ON ADC_SCAN_OFF 割り込み相互間のサンプル数 ADC_SAMPLES_PER_INT_1 ADC_SAMPLES_PER_INT_2 ..... ADC_SAMPLES_PER_INT_15 ADC_SAMPLES_PER_INT_16 バッファモード選択 ADC_ALT_BUF_ON ADC_ALT_BUF_OFF 代替入力サンプル・モードの選択 ADC_ALT_INPUT_ON ADC_ALT_INPUT_OFF
config3	ADCON3 レジスタに設定される次のように定義されたパラメータ : 自動サンプル時間ビット ADC_SAMPLE_TIME_0 ADC_SAMPLE_TIME_1 ..... ADC_SAMPLE_TIME_30 ADC_SAMPLE_TIME_31 変換クロック・ソースの選択 ADC_CONV_CLK_INTERNAL_RC ADC_CONV_CLK_SYSTEM

OpenADC12 (続き)

変換クロックの選択
ADC_CONV_CLK_Tcy2
ADC_CONV_CLK_Tcy
ADC_CONV_CLK_3Tcy2
.....
ADC_CONV_CLK_32Tcy

configport ADPCFG レジスタに設定される次のように定義されたピン選択 :

ENABLE_ALL_ANA
ENABLE_ALL_DIG
ENABLE_AN0_ANA
ENABLE_AN1_ANA
ENABLE_AN2_ANA
.....
ENABLE_AN15_ANA

configscan ADCSSL レジスタに設定される次のように定義されたスキャン選択パラメータ :

SCAN_NONE
SCAN_ALL
SKIP_SCAN_AN0
SKIP_SCAN_AN1
.....
SKIP_SCAN_AN15

戻り値 : なし

備考 : この関数は、ADC の次のパラメータを設定します :
動作モード、スリープ・モード動作、データ出力フォーマット、サンプル・クロック・ソース、Vref ソース、サンプル数 /int、バッファ・フィル・モード、代替入力サンプル・モード、自動サンプル時間、変換クロック・ソース、変換クロック選択ビット、ポート設定コントロール・ビット。

ソース・ファイル : OpenADC12.c

コード例 :

```
OpenADC12(ADC_MODULE_OFF &
          ADC_IDLE_CONTINUE &
          ADC_FORMAT_INTG &
          ADC_AUTO_SAMPLING_ON,
          ADC_VREF_AVDD_AVSS &
          ADC_SCAN_OFF &
          ADC_BUF_MODE_OFF &
          ADC_ALT_INPUT_ON &
          ADC_SAMPLES_PER_INT_15,
          ADC_SAMPLE_TIME_4 &
          ADC_CONV_CLK_SYSTEM &
          ADC_CONV_CLK_Tcy,
          ENABLE_AN0_ANA,
          SKIP_SCAN_AN1 &
          SKIP_SCAN_AN2 &
          SKIP_SCAN_AN5 &
          SKIP_SCAN_AN7);
```

ReadADC12

説明: この関数は、ADC バッファ・レジスタから変換値を読み出します。

インクルード: adc12.h

プロトタイプ: unsigned int ReadADC12(unsigned char *bufIndex*);

引数: *bufIndex* 読み出す ADC バッファ番号。

戻り値: なし

備考: この関数は、ADC バッファレジスタの値を返します。ADCBUF0 ～ ADCBUFF のレジスタを正しく読み出すために、0 ～ 15 の *bufIndex* 値を指定する必要があります。

ソース・ファイル: ReadADC12.c

コード例:
```
unsigned int result;
result = ReadADC12(5);
```

StopSampADC12

説明: この関数は、ConvertADC12 と同じです。

ソース・ファイル: #define to ConvertADC12 in adc12.h

SetChanADC12

説明: この関数は、サンプル・マルチプレクサ A と B の正入力と負入力を設定します。

インクルード: adc12.h

プロトタイプ: void SetChanADC12(unsigned int *channel*);

引数: *channel* ADCHS レジスタに設定される次のように定義された入力選択パラメータ :

SAMPLE A に対する A/D チャンネル 0 の正入力選択
ADC_CH0_POS_SAMPLEA_AN0
ADC_CH0_POS_SAMPLEA_AN1
.....
ADC_CH0_POS_SAMPLEA_AN15

SAMPLE A に対する A/D チャンネル 0 の負入力選択
ADC_CH0_NEG_SAMPLEA_AN1
ADC_CH0_NEG_SAMPLEA_NVREF

SAMPLE B に対する A/D チャンネル 0 の正入力選択
ADC_CH0_POS_SAMPLEB_AN0
ADC_CH0_POS_SAMPLEB_AN1
.....
ADC_CH0_POS_SAMPLEB_AN15

SAMPLE B に対する A/D チャンネル 0 の負入力選択
ADC_CH0_NEG_SAMPLEB_AN1
ADC_CH0_NEG_SAMPLEB_NVREF

戻り値: なし

備考: この関数は、ADCHS レジスタへ書き込みを行うことにより、サンプル・マルチプレクサ A と B に対する入力を設定します。

ソース・ファイル: SetChanADC12.c

コード例:
```
SetChanADC12(ADC_CH0_POS_SAMPLEA_AN4 &
             ADC_CH0_NEG_SAMPLEA_NVREF);
```

3.5.2 個別マクロ

EnableIntADC

説明 : このマクロは、ADC 割り込みをイネーブルします。

インクルード : adc12.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタの ADC 割り込みイネーブル・ビットをセットします。

コード例 :
```
EnableIntADC;
```

DisableIntADC

説明 : このマクロは、ADC 割り込みをディスエーブルします。

インクルード : adc12.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタの ADC 割り込みイネーブル・ビットをクリアします。

コード例 :
```
DisableIntADC;
```

SetPriorityIntADC

説明 : このマクロは、ADC 割り込みの優先順位をセットします。

インクルード : adc12.h

引数 : *priority*

備考 : このマクロは、割り込み優先順位コントロール・レジスタの ADC 割り込み優先順位ビットをセットします。

コード例 :
```
SetPriorityIntADC(6);
```

3.5.3 使用例

```
#define _ _dsPIC30F6014_ _
#include <p30fxxxx.h>
#include<adc12.h>
unsigned int Channel, PinConfig, Scanselect, Adcon3_reg, Adcon2_reg,
Adcon1_reg;
int  main(void)
{
    unsigned int result[20], i;
    ADCON1bits.ADON = 0;         /* turn off ADC */
    Channel = ADC_CH0_POS_SAMPLEA_AN4 &
              ADC_CH0_NEG_SAMPLEA_NVREF &
              ADC_CH0_POS_SAMPLEB_AN2&
              ADC_CH0_NEG_SAMPLEB_AN1;
    SetChanADC12(Channel);
    ConfigIntADC12(ADC_INT_DISABLE);
    PinConfig  = ENABLE_AN4_ANA;
    Scanselect = SKIP_SCAN_AN2 & SKIP_SCAN_AN5 &
                 SKIP_SCAN_AN9 & SKIP_SCAN_AN10 &
                 SKIP_SCAN_AN14 & SKIP_SCAN_AN15 ;

    Adcon3_reg = ADC_SAMPLE_TIME_10 &
                 ADC_CONV_CLK_SYSTEM &
                 ADC_CONV_CLK_13Tcy;

    Adcon2_reg = ADC_VREF_AVDD_AVSS &
                 ADC_SCAN_OFF &
                 ADC_ALT_BUF_OFF &
                 ADC_ALT_INPUT_OFF &
                 ADC_SAMPLES_PER_INT_16;
    Adcon1_reg = ADC_MODULE_ON &
                 ADC_IDLE_CONTINUE &
                 ADC_FORMAT_INTG &
                 ADC_CLK_MANUAL &
                 ADC_AUTO_SAMPLING_OFF;
    OpenADC12(Adcon1_reg, Adcon2_reg,
              Adcon3_reg,PinConfig, Scanselect);
    i = 0;
    while( i <16 )
    {
        ADCON1bits.SAMP = 1;
        while(!ADCON1bits.SAMP);
        ConvertADC12();
        while(ADCON1bits.SAMP);
        while(!BusyADC12());
        while(BusyADC12());
        result[i] = ReadADC12(i);
        i++;
    }
}
```

3.6 ADC10 関数

このセクションには、10 ADC 用個別関数の一覧と、各関数の使用例を記載します。関数はマクロとして構成可能です。

3.6.1 個別関数

BusyADC10

説明 : この関数は、ADC 変換ステータスを返します。

インクルード : `adc10.h`

プロトタイプ :
```
char BusyADC10(void);
```

引数 : なし

戻り値 : DONE 値が '0' の場合、'1' を返して、ADC が変換動作中でビジーであることを表示します。
DONE 値が '1' の場合、'0' を返して、ADC の変換動作が完了したことを表示します。

備考 : この関数は ADCON1 <DONE> ビットのステータスの反転を返し、ADC が変換中か否かを表示します。

ソース・ファイル : `BusyADC10.c`

コード例 :
```
while(BusyADC10());
```

CloseADC10

説明 : この関数は、ADC モジュールをターンオフし、ADC 割り込みをディスエーブルします。

インクルード : `adc10.h`

プロトタイプ :
```
void CloseADC10(void);
```

引数 : なし

戻り値 : なし

備考 : この関数は ADC 割り込みをディスエーブルした後に、ADC モジュールをターンオフします。割り込みフラグ・ビット (ADIF) もクリアします。

ソース・ファイル : `CloseADC10.c`

コード例 :
```
CloseADC10();
```

ConfigIntADC10

説明 : この関数は、ADC 割り込みを設定します。

インクルード : `adc10.h`

プロトタイプ :
```
void ConfigIntADC10(unsigned int config);
```

引数 :

config	次のように定義される ADC 割り込み優先順位とイネーブル / ディスエーブル情報 :

ADC 割り込みのイネーブル / ディスエーブル
ADC_INT_ENABLE
ADC_INT_DISABLE

ConfigIntADC10 (続き)

ADC 割り込みの優先順位
ADC_INT_PRI_0
ADC_INT_PRI_1
ADC_INT_PRI_2
ADC_INT_PRI_3
ADC_INT_PRI_4
ADC_INT_PRI_5
ADC_INT_PRI_6
ADC_INT_PRI_7

戻り値 : なし

備考 : この関数は、割り込みフラグ (ADIF) ビットをクリアした後に、割り込み優先順位を設定して、割り込みをイネーブル / ディスエーブルします。

ソース・ファイル : ConfigIntADC10.c

コード例 :
```
ConfigIntADC10(ADC_INT_PRI_3 &
               ADC_INT_DISABLE);
```

ConvertADC10

説明 : この関数は、A/D 変換を開始させます。

インクルード : adc10.h

プロトタイプ :
```
void ConvertADC10(void);
```

引数 : なし

戻り値 : なし

備考 : この関数は ADCON1<SAMP> ビットをクリアしてサンプリングを停止させて、変換を開始させます。
これは、ADCON1 <SSRC> ビットをクリアすることにより A/D 変換のトリガ・ソースを手動で選択した場合にのみ可能です。

ソース・ファイル : ConvertADC10.c

コード例 :
```
ConvertADC10();
```

OpenADC10

説明 : この関数は ADC を設定します。

インクルード : adc10.h

プロトタイプ :
```
void OpenADC10(unsigned int config1,
               unsigned int config2,
               unsigned int config3,
               unsigned int configport,
               unsigned int configscan)
```

引数 :

config1 ADCON1 レジスタに設定される次のように定義されたパラメータ :

モジュールの On/Off
ADC_MODULE_ON
ADC_MODULE_OFF

アイドル・モード動作
ADC_IDLE_CONTINUE
ADC_IDLE_STOP

OpenADC10 (続き)

変換結果の出力フォーマット
ADC_FORMAT_SIGN_FRACT
ADC_FORMAT_FRACT
ADC_FORMAT_SIGN_INT
ADC_FORMAT_INTG

変換トリガ・ソース
ADC_CLK_AUTO
ADC_CLK_MPWM
ADC_CLK_TMR
ADC_CLK_INT0
ADC_CLK_MANUAL

自動サンプリング選択
ADC_AUTO_SAMPLING_ON
ADC_AUTO_SAMPLING_OFF

同時サンプリング
ADC_SAMPLE_SIMULTANEOUS
ADC_SAMPLE_INDIVIDUAL

サンプルのイネーブル
ADC_SAMP_ON
ADC_SAMP_OFF

config2 ADCON2 レジスタに設定される次のように定義されたパラメータ :

リファレンス電圧
ADC_VREF_AVDD_AVSS
ADC_VREF_EXT_AVSS
ADC_VREF_AVDD_EXT
ADC_VREF_EXT_EXT

スキャン選択
ADC_SCAN_ON
ADC_SCAN_OFF

使用 A/D チャンネル
ADC_CONVERT_CH0123
ADC_CONVERT_CH01
ADC_CONVERT_CH0

割り込み相互間のサンプル数
ADC_SAMPLES_PER_INT_1
ADC_SAMPLES_PER_INT_2
.....
ADC_SAMPLES_PER_INT_15
ADC_SAMPLES_PER_INT_16

バッファモード選択
ADC_ALT_BUF_ON
ADC_ALT_BUF_OFF

代替入力サンプル・モードの選択
ADC_ALT_INPUT_ON
ADC_ALT_INPUT_OFF

OpenADC10 (続き)

	config3	ADCON3 レジスタに設定される次のように定義されたパラメータ : 自動サンプル時間ビット ADC_SAMPLE_TIME_0 ADC_SAMPLE_TIME_1 ..... ADC_SAMPLE_TIME_30 ADC_SAMPLE_TIME_31 変換クロック・ソースの選択 ADC_CONV_CLK_INTERNAL_RC ADC_CONV_CLK_SYSTEM 変換クロックの選択 ADC_CONV_CLK_Tcy2 ADC_CONV_CLK_Tcy ADC_CONV_CLK_3Tcy2 ..... ADC_CONV_CLK_32Tcy
	configport	ADPCFG レジスタに設定される次のように定義されたピン選択 : ENABLE_ALL_ANA ENABLE_ALL_DIG ENABLE_AN0_ANA ENABLE_AN1_ANA ENABLE_AN2_ANA ..... ENABLE_AN15_ANA
	configscan	ADCSSL レジスタに設定される次のように定義されたスキャン選択パラメータ : SCAN_NONE SCAN_ALL SKIP_SCAN_AN0 SKIP_SCAN_AN1 ..... SKIP_SCAN_AN15
戻り値 :	なし	
備考 :	この関数は、ADC の次のパラメータを設定します : 動作モード、スリープ・モード動作、データ出力フォーマット、サンプル・クロック・ソース、Vref ソース、サンプル数 /int、バッファ・フィル・モード、代替入力サンプル・モード、自動サンプル時間、変換クロック・ソース、変換クロック選択ビット、ポート設定コントロール・ビット。	
ソース・ファイル :	OpenADC10.c	

OpenADC10 (続き)

コード例 :

```
OpenADC10(ADC_MODULE_OFF &
          ADC_IDLE_STOP &
          ADC_FORMAT_SIGN_FRACT &
          ADC_CLK_INT0 &
          ADC_SAMPLE_INDIVIDUAL &
          ADC_AUTO_SAMPLING_ON,
          ADC_VREF_AVDD_AVSS &
          ADC_SCAN_OFF &
          ADC_BUF_MODE_OFF &
          ADC_ALT_INPUT_ON &
          ADC_CONVERT_CH0 &
          ADC_SAMPLES_PER_INT_10,
          ADC_SAMPLE_TIME_4 &
          ADC_CONV_CLK_SYSTEM &
          ADC_CONV_CLK_Tcy,
          ENABLE_AN1_ANA,
          SKIP_SCAN_AN0 &
          SKIP_SCAN_AN3 &
          SKIP_SCAN_AN4 &
          SKIP_SCAN_AN5);
```

ReadADC10

説明 : この関数は、ADC バッファ・レジスタから変換値を読み出します。

インクルード : adc10.h

プロトタイプ : `unsigned int ReadADC10(unsigned char bufIndex);`

引数 : *bufIndex* 読み出す ADC バッファ番号。

戻り値 : なし

備考 : この関数は、ADC バッファレジスタの値を返します。ADCBUF0 ～ ADCBUFF のレジスタを正しく読み出すために、0 ～ 15 の *bufIndex* 値を指定する必要があります。

ソース・ファイル : ReadADC10.c

コード例 :

```
unsigned int result;
result = ReadADC10(3);
```

StopSampADC10

説明 : この関数は、ConvertADC10 と同じです。

ソース・ファイル : #define to ConvertADC10 in adc10.h

SetChanADC10

説明 : この関数は、サンプル・マルチプレクサ A と B の正入力と負入力を設定します。

インクルード : adc10.h

プロトタイプ : `void SetChanADC10(unsigned int channel);`

SetChanADC10 (続き)

引数 : *channel* ADCHS レジスタに設定される次のように定義された入力選択パラメータ :

サンプル A に対する A/D チャンネル 1、2、3 の負入力
ADC_CHX_NEG_SAMPLEA_AN9AN10AN11
ADC_CHX_NEG_SAMPLEA_AN6AN7AN8
ADC_CHX_NEG_SAMPLEA_NVREF

サンプル B に対する A/D チャンネル 1、2、3 の負入力
ADC_CHX_NEG_SAMPLEB_AN9AN10AN11
ADC_CHX_NEG_SAMPLEB_AN6AN7AN8
ADC_CHX_NEG_SAMPLEB_NVREF

サンプル A に対する A/D チャンネル 1、2、3 の正入力
ADC_CHX_POS_SAMPLEA_AN3AN4AN5
ADC_CHX_POS_SAMPLEA_AN0AN1AN2

サンプル B に対する A/D チャンネル 1、2、3 の正入力
ADC_CHX_POS_SAMPLEA_AN3AN4AN5
ADC_CHX_POS_SAMPLEB_AN0AN1AN2

サンプル A に対する A/D チャンネル 0 の正入力選択
ADC_CH0_POS_SAMPLEA_AN0
ADC_CH0_POS_SAMPLEA_AN1
.....
ADC_CH0_POS_SAMPLEA_AN15

サンプル A に対する A/D チャンネル 0 の負入力選択
ADC_CH0_NEG_SAMPLEA_AN1
ADC_CH0_NEG_SAMPLEA_NVREF

サンプル B に対する A/D チャンネル 0 の正入力選択
ADC_CH0_POS_SAMPLEB_AN0
ADC_CH0_POS_SAMPLEB_AN1
.....
ADC_CH0_POS_SAMPLEB_AN15

サンプル B に対する A/D チャンネル 0 の負入力選択
ADC_CH0_NEG_SAMPLEB_AN1
ADC_CH0_NEG_SAMPLEB_NVREF

戻り値 : なし

備考 : この関数は、ADCHS レジスタへ書き込みを行うことにより、サンプル・マルチプレクサ A と B に対する入力を設定します。

ソース・ファイル : SetChanADC10.c

コード例 :
```
SetChanADC10(ADC_CH0_POS_SAMPLEA_AN0 &
             ADC_CH0_NEG_SAMPLEA_NVREF);
```

3.6.2 個別マクロ

EnableIntADC

説明 : このマクロは、ADC 割り込みをイネーブルします。

インクルード : adc10.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタの ADC 割り込みイネーブル・ビットをセットします。

コード例 :
```
EnableIntADC;
```

DisableIntADC

説明 : このマクロは、ADC 割り込みをディスエーブルします。

インクルード : adc10.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタの ADC 割り込みイネーブル・ビットをクリアします。

コード例 :
```
DisableIntADC;
```

SetPriorityIntADC

説明 : このマクロは、ADC 割り込みの優先順位をセットします。

インクルード : adc10.h

引数 : *priority*

備考 : このマクロは、割り込み優先順位コントロール・レジスタの ADC 割り込み優先順位ビットをセットします。

コード例 :
```
SetPriorityIntADC(2);
```

3.6.3 使用例

```
#define _ _dsPIC30F6010_ _
#include <p30fxxxx.h>
#include<adc10.h>
unsigned int Channel, PinConfig, Scanselect, Adcon3_reg, Adcon2_reg,
Adcon1_reg;
int  main(void)
{
    unsigned int result[20], i;
    ADCON1bits.ADON = 0;         /* turn off ADC */
    Channel = ADC_CH0_POS_SAMPLEA_AN4 &
              ADC_CH0_NEG_SAMPLEA_NVREF &
              ADC_CH0_POS_SAMPLEB_AN2 &
              ADC_CH0_NEG_SAMPLEB_AN1;
    SetChanADC1(Channel);

    ConfigIntADC10(ADC_INT_DISABLE);
    PinConfig  = ENABLE_AN4_ANA;
    Scanselect = SKIP_SCAN_AN2 & SKIP_SCAN_AN5 &
                 SKIP_SCAN_AN9 & SKIP_SCAN_AN10 &
                 SKIP_SCAN_AN14 & SKIP_SCAN_AN15;

    Adcon3_reg = ADC_SAMPLE_TIME_10 &
                 ADC_CONV_CLK_SYSTEM &
                 ADC_CONV_CLK_13Tcy;

    Adcon2_reg = ADC_VREF_AVDD_AVSS &
                 ADC_SCAN_OFF &
                 ADC_ALT_BUF_OFF &
                 ADC_ALT_INPUT_OFF &
                 ADC_CONVERT_CH0123 &
                 ADC_SAMPLES_PER_INT_16;
    Adcon1_reg = ADC_MODULE_ON &
                 ADC_IDLE_CONTINUE &
                 ADC_FORMAT_INTG &
                 ADC_CLK_MANUAL &
                 ADC_SAMPLE_SIMULTANEOUS &
                 ADC_AUTO_SAMPLING_OFF;
    OpenADC10(Adcon1_reg, Adcon2_reg,
              Adcon3_reg,PinConfig, Scanselect);
    i = 0;
    while(i <16 )
    {
        ADCON1bits.SAMP = 1;
        while(!ADCON1bits.SAMP);
        ConvertADC10();
        while(ADCON1bits.SAMP);
        while(!BusyADC10());
        while(BusyADC10());
        result[i] = ReadADC10(i);
        i++;
    }
}
```

3.7 タイマ関数

このセクションには、タイマ用個別関数の一覧と、各関数の使用例を記載します。関数はマクロとして構成可能です。

3.7.1 個別関数

CloseTimer1
CloseTimer2
CloseTimer3
CloseTimer4
CloseTimer5

説明 :	この関数は、16 ビット・タイマ・モジュールをターンオフします。
インクルード :	`timer.h`

プロトタイプ :	`void CloseTimer1(void);`
	`void CloseTimer2(void);`
	`void CloseTimer3(void);`
	`void CloseTimer4(void);`
	`void CloseTimer5(void);`

引数 :	なし
戻り値 :	なし
備考 :	この関数は、16 ビット・タイマ割り込みをディスエーブルした後、タイマ・モジュールをターンオフします。割り込みフラグ・ビット (TxIF) もクリアします。

ソース・ファイル :	`CloseTimer1.c`
	`CloseTimer2.c`
	`CloseTimer3.c`
	`CloseTimer4.c`
	`CloseTimer5.c`

コード例 :	`CloseTimer1();`

CloseTimer23
CloseTimer45

説明 :	この関数は、32 ビット・タイマ・モジュールをターンオフします。
インクルード :	`timer.h`
プロトタイプ :	`void CloseTimer23 (void)` `void CloseTimer45 (void)`
引数 :	なし
戻り値 :	なし
備考 :	この関数はタイマ割り込みをディスエーブルした後に、タイマ・モジュールをターンオフします。割り込みフラグ・ビット (TxIF) もクリアします。 `CloseTimer23` は Timer2 をターンオフし、Timer3 割り込みをディスエーブルします。 `CloseTimer45` は Timer4 をターンオフし、Timer5 割り込みをディスエーブルします。
ソース・ファイル :	`CloseTimer23.c` `CloseTimer45.c`
コード例 :	`CloseTimer23();`

ConfigIntTimer1
ConfigIntTimer2
ConfigIntTimer3
ConfigIntTimer4
ConfigIntTimer5

説明 : この関数は、16 ビット・タイマ割り込みを設定します。

インクルード : timer.h

プロトタイプ :
```
void ConfigIntTimer1(unsigned int config);
void ConfigIntTimer2(unsigned int config);
void ConfigIntTimer3(unsigned int config);
void ConfigIntTimer4(unsigned int config);
void ConfigIntTimer5(unsigned int config);
```

引数 : *config* 次のように定義されるタイマ割り込み優先順位とイネーブル / ディスエーブル情報 :

```
Tx_INT_PRIOR_7
Tx_INT_PRIOR_6
Tx_INT_PRIOR_5
Tx_INT_PRIOR_4
Tx_INT_PRIOR_3
Tx_INT_PRIOR_2
Tx_INT_PRIOR_1
Tx_INT_PRIOR_0

Tx_INT_ON
Tx_INT_OFF
```

戻り値 : なし

備考 : この関数は、16 ビット割り込みフラグ (TxIF) ビットをクリアした後に、割り込み優先順位を設定して、割り込みをイネーブル / ディスエーブルします。

ソース・ファイル :
```
ConfigIntTimer1.c
ConfigIntTimer2.c
ConfigIntTimer3.c
ConfigIntTimer4.c
ConfigIntTimer5.c
```

コード例 :
```
ConfigIntTimer1(T1_INT_PRIOR_3 & T1_INT_ON);
```

ConfigIntTimer23
ConfigIntTimer45

説明 : この関数は、32 ビット・タイマ割り込みを設定します。

インクルード : timer.h

プロトタイプ :
```
void ConfigIntTimer23(unsigned int config);
void ConfigIntTimer45(unsigned int config);
```

引数 : *config* 次のように定義されるタイマ割り込み優先順位とイネーブル / ディスエーブル情報 :

ConfigIntTimer23
ConfigIntTimer45 (続き)

```
                    Tx_INT_PRIOR_7
                    Tx_INT_PRIOR_6
                    Tx_INT_PRIOR_5
                    Tx_INT_PRIOR_4
                    Tx_INT_PRIOR_3
                    Tx_INT_PRIOR_2
                    Tx_INT_PRIOR_1
                    Tx_INT_PRIOR_0

                    Tx_INT_ON
                    Tx_INT_OFF
```

戻り値 : なし

備考 : この関数は、32 ビット割り込みフラグ (TxIF) ビットをクリアした後に、割り込み優先順位を設定して、割り込みをイネーブル / ディスエーブルします。

ソース・ファイル : ConfigIntTimer23.c
ConfigIntTimer45.c

コード例 :
```
ConfigIntTimer23(T3_INT_PRIOR_5 & T3_INT_ON);
```

OpenTimer1
OpenTimer2
OpenTimer3
OpenTimer4
OpenTimer5

説明 : この関数は、16 ビット・タイマ・モジュールを設定します。

インクルード : timer.h

プロトタイプ :
```
void OpenTimer1(unsigned int config,
                unsigned int period)
void OpenTimer2(unsigned int config,
                unsigned int period)
void OpenTimer3(unsigned int config,
                unsigned int period)
void OpenTimer4(unsigned int config,
                unsigned int period)
void OpenTimer5(unsigned int config,
                unsigned int period)
```

引数 :

config TxCON レジスタに設定される次のように定義されたパラメータ :

タイマ・モジュールの On/Off
```
Tx_ON
Tx_OFF
```
タイマ・モジュール・アイドル・モード On/Off
```
Tx_IDLE_CON
Tx_IDLE_STOP
```
タイマ・ゲーティング時間の積算イネーブル
```
Tx_GATE_ON
Tx_GATE_OFF
```

OpenTimer1
OpenTimer2
OpenTimer3
OpenTimer4
OpenTimer5 (続き)

タイマ・プリスケーラ
Tx_PS_1_1
Tx_PS_1_8
Tx_PS_1_64
Tx_PS_1_128

タイマ同期クロック・イネーブル
Tx_SYNC_EXT_ON
Tx_SYNC_EXT_OFF

タイマ・クロック・ソース
Tx_SOURCE_EXT
Tx_SOURCE_INT

period PR レジスタに格納する一致周期値

戻り値 : なし

備考 : この関数は、16 ビット・タイマ・コントロール・レジスタを設定し、PR レジスタへ一致周期値を設定します。

ソース・ファイル :
```
OpenTimer1.c
OpenTimer2.c
OpenTimer3.c
OpenTimer4.c
OpenTimer5.c
```

コード例 :
```
OpenTimer1(T1_ON & T1_GATE_OFF &
           T1_PS_1_8 & T1_SYNC_EXT_OFF &
           T1_SOURCE_INT, 0xFF);
```

OpenTimer23
OpenTimer45

説明 : この関数は、32 ビット・タイマ・モジュールを設定します。

インクルード : timer.h

プロトタイプ :
```
void OpenTimer23(unsigned int config,
                 unsigned long period);
void OpenTimer45(unsigned int config,
                 unsigned long period);
```

引数 :

config TxCON レジスタに設定される次のように定義されたパラメータ :

タイマ・モジュールの On/Off
Tx_ON
Tx_OFF

タイマ・モジュール・アイドル・モード On/Off
Tx_IDLE_CON
Tx_IDLE_STOP

タイマ・ゲーティング時間の積算イネーブル
Tx_GATE_ON
Tx_GATE_OFF

OpenTimer23
OpenTimer45 (続き)

タイマ・プリスケーラ
Tx_PS_1_1
Tx_PS_1_8
Tx_PS_1_64
Tx_PS_1_128

タイマ同期クロック・イネーブル
Tx_SYNC_EXT_ON
Tx_SYNC_EXT_OFF

タイマ・クロック・ソース
Tx_SOURCE_EXT
Tx_SOURCE_INT

period 32 ビット PR レジスタに格納する一致周期値

戻り値 : なし

備考 : この関数は、32 ビット・タイマ・コントロール・レジスタを設定し、PR レジスタへ一致周期値を設定します。

ソース・ファイル :
```
OpenTimer23.c
OpenTimer45.c
```

コード例 :
```
OpenTimer23(T2_ON & T2_GATE_OFF &
            T2_PS_1_8 & T2_32BIT_MODE_ON &
            T2_SYNC_EXT_OFF &
            T2_SOURCE_INT, 0xFFFF);
```

ReadTimer1
ReadTimer2
ReadTimer3
ReadTimer4
ReadTimer5

説明 : この関数は、16 ビット・タイマ・レジスタの値を読み出します。

インクルード :
```
timer.h
```

プロトタイプ :
```
unsigned int ReadTimer1(void);
unsigned int ReadTimer2(void);
unsigned int ReadTimer3(void);
unsigned int ReadTimer4(void);
unsigned int ReadTimer5(void);
```

引数 : なし

戻り値 : なし

備考 : この関数は、16 ビット TMR レジスタの値を返します。

ソース・ファイル :
```
ReadTimer1.c
ReadTimer2.c
ReadTimer3.c
ReadTimer4.c
ReadTimer5.c
```

コード例 :
```
unsigned int timer1_value;
timer1_value = ReadTimer1();
```

ReadTimer23
ReadTimer45

説明: この関数は、32 ビット・タイマ・レジスタの値を読み出します。

インクルード: timer.h

プロトタイプ:
```
unsigned long ReadTimer23(void);
unsigned long ReadTimer45(void);
```

引数: なし

戻り値: なし

備考: この関数は、32 ビット TMR レジスタの値を返します。

ソース・ファイル:
```
ReadTimer23.c
ReadTimer45.c
```

コード例:
```
unsigned long timer23_value;
timer23_value = ReadTimer23();
```

WriteTimer1
WriteTimer2
WriteTimer3
WriteTimer4
WriteTimer5

説明: この関数は、16 ビット値をタイマ・レジスタへ書き込みます。

インクルード: timer.h

プロトタイプ:
```
void WriteTimer1(unsigned int timer);
void WriteTimer2(unsigned int timer);
void WriteTimer3(unsigned int timer);
void WriteTimer4(unsigned int timer);
void WriteTimer5(unsigned int timer);
```

引数: *timer* TMR レジスタへ書き込む 16 ビット値

戻り値: なし

備考: なし

ソース・ファイル:
```
WriteTimer1.c
WriteTimer2.c
WriteTimer3.c
WriteTimer4.c
WriteTimer5.c
```

コード例:
```
unsigned int timer_init = 0xAB;
WriteTimer1(timer_init);
```

WriteTimer23
WriteTimer45

説明 :	この関数は、32 ビット値をタイマ・レジスタへ書き込みます。
インクルード :	timer.h
プロトタイプ :	void WriteTimer23(unsigned long *timer*); void WriteTimer45(unsigned long *timer*);
引数 :	*timer*　TMR レジスタへ書き込む 32 ビット値
戻り値 :	なし
備考 :	なし
ソース・ファイル :	WriteTimer23.c WriteTimer45.c
コード例 :	unsigned long timer23_init = 0xABCD; WriteTimer23(timer23_init);

3.7.2　個別マクロ

EnableIntT1
EnableIntT2
EnableIntT3
EnableIntT4
EnableIntT5

説明 :	このマクロは、タイマ割り込みをイネーブルします。
インクルード :	timer.h
引数 :	なし
備考 :	このマクロは、割り込みイネーブル・コントロール・レジスタのタイマ割り込みイネーブル・ビットをセットします。
コード例 :	EnableIntT1;

DisableIntT1
DisableIntT2
DisableIntT3
DisableIntT4
DisableIntT5

説明 :	このマクロは、タイマ割り込みをディスエーブルします。
インクルード :	timer.h
引数 :	なし
備考 :	このマクロは、割り込みイネーブル・コントロール・レジスタのタイマ割り込みイネーブル・ビットをクリアします。
コード例 :	DisableIntT2;

SetPriorityIntT1
SetPriorityIntT2
SetPriorityIntT3
SetPriorityIntT4
SetPriorityIntT5

説明 :	このマクロは、タイマ割り込みの優先順位をセットします。
インクルード :	timer.h
引数 :	*priority*
備考 :	このマクロは、割り込み優先順位コントロール・レジスタのタイマ割り込み優先順位ビットをセットします。
コード例 :	SetPriorityIntT4(7);

3.7.3 使用例

```
#define _ _dsPIC30F6014_ _
#include <p30fxxxx.h>
#include<timer.h>
unsigned int timer_value;
void _ _attribute_ _((_ _interrupt_ _)) _T1Interrupt(void)
{
    PORTDbits.RD1 = 1;    /* turn off LED on RD1 */
    WriteTimer1(0);
    IFS0bits.T1IF = 0;    /* Clear Timer interrupt flag */
}
int main(void)
{
    unsigned int match_value;
    TRISDbits.TRISD1 = 0;
    PORTDbits.RD1 = 1;    /* turn off LED on RD1 */
/* Enable Timer1 Interrupt and Priority to "1" */
    ConfigIntTimer1(T1_INT_PRIOR_1 & T1_INT_ON);
    WriteTimer1(0);
    match_value = 0xFFF;
    OpenTimer1(T1_ON & T1_GATE_OFF & T1_IDLE_STOP &
               T1_PS_1_1 & T1_SYNC_EXT_OFF &
               T1_SOURCE_INT, match_value);
/* Wait till the timer matches with the period value */
    while(1)
    {
        timer_value = ReadTimer1();
        if(timer_value >= 0x7FF)
        {
            PORTDbits.RD1 = 0; /* turn on LED on RD1 */
        }
    }
    CloseTimer1();
}
```

3.8 リセット / 制御関数

このセクションには、リセット / 制御の個別関数の一覧を記載します。関数はマクロとして構成可能です。

3.8.1 個別関数

isBOR

説明 :	この関数は、リセットがブラウンアウト・リセットによるものか否かを調べます。
インクルード :	reset.h
プロトタイプ :	char isBOR(void);
引数 :	なし
戻り値 :	この関数は、RCON<BOR> ビットのステータスを返します。 戻り値が '1' の場合、リセットはブラウンアウトによるものであることを示します。 戻り値が '0' の場合、ブラウンアウトは発生していないことを示します。
備考 :	なし
ソース・ファイル :	isBOR.c
コード例 :	char reset_state; reset_state = isBOR();

isPOR

説明 :	この関数は、リセットがパワーオン・リセットによるものか否かを調べます。
インクルード :	reset.h
プロトタイプ :	char isPOR(void);
引数 :	なし
戻り値 :	この関数は、RCON<POR> ビットのステータスを返します。 戻り値が '1' の場合、リセットはパワーオンによるものであることを示します。 戻り値が '0' の場合、パワーオンは発生していないことを示します。
備考 :	なし
ソース・ファイル :	isPOR.c
コード例 :	char reset_state; reset_state = isPOR();

isLVD

説明 :	この関数は、低電圧検出割り込みフラグがセットされているか否かを調べます。
インクルード :	reset.h
プロトタイプ :	char isLVD(void);
引数 :	なし

isLVD (続き)

戻り値 : この関数は、IFS2<LVDIF> ビットのステータスを返します。
戻り値が '1' の場合、低電圧検出割り込みが発生したことを示します。
戻り値が '0' の場合、低電圧検出割り込みが発生していないことを示します。

備考 : なし

ソース・ファイル : isLVD.c

コード例 :
```
char lvd;
lvd = isLVD();
```

isMCLR

説明 : この関数は、リセット条件が $\overline{\text{MCLR}}$ ピンが low になったことによるものか否かを調べます。

インクルード : reset.h

プロトタイプ :
```
char isMCLR(void);
```

引数 : なし

戻り値 : この関数は、RCON<EXTR> ビットのステータスを返します。
戻り値が '1' の場合、リセットは $\overline{\text{MCLR}}$ ピンが low になったことによるものであることを示します。
戻り値が '0' の場合、リセットは $\overline{\text{MCLR}}$ ピンが low になったことによるものでないことを示します。

備考 : なし

ソース・ファイル : isMCLR.c

コード例 :
```
char reset_state;
reset_state = isMCLR();
```

isWDTTO

説明 : この関数は、リセットが WDT タイムアウトによるものか否かを調べます。

インクルード : reset.h

プロトタイプ :
```
char isWDTTO(void);
```

引数 : なし

戻り値 : この関数は、RCON<WDTO> ビットのステータスを返します。
戻り値が '1' の場合、リセットは WDT タイムアウトによるものであることを示します。
戻り値が '0' の場合、リセットは WDT タイムアウトによるものでないことを示します。

備考 : なし

ソース・ファイル : isWDTTO.c

コード例 :
```
char reset_state;
reset_state = isWDTTO();
```

isWDTWU

説明 :	この関数は、スリープからのウェイクアップが WDT ウェイクアップによるものか否かを調べます。
インクルード :	reset.h
プロトタイプ :	char isWDTWU(void);
引数 :	なし
戻り値 :	この関数は、RCON<WDTO> ビットと RCON<SLEEP> ビットのステータスを返します。 戻り値が '1' の場合、スリープからのウェイクアップは WDT タイムアウトによるものであることを示します。 戻り値が '0' の場合、スリープからのウェイクアップは WDT タイムアウトによるものでないことを示します。
備考 :	なし
ソース・ファイル :	isWDTWU.c
コード例 :	char reset_state; reset_state = isWDTWU();

isWU

説明 :	この関数は、スリープからのウェイクアップが $\overline{\text{MCLR}}$、POR、BOR または他の割り込みによるものか否かを調べます。
インクルード :	reset.h
プロトタイプ :	char isWU(void);
引数 :	なし
戻り値 :	この関数は、スリープからのウェイクアップが発生したか否かを調べます。 発生した場合、ウェイクアップの原因を調べます。 '1' の場合、ウェイクアップは割り込みの発生が原因です。 '2' の場合、ウェイクアップは $\overline{\text{MCLR}}$ が原因です。 '3' の場合、ウェイクアップは POR が原因です。 '4' の場合、ウェイクアップは BOR が原因です。 スリープからのウェイクアップが発生しなかった場合、値 '0' が返されます。
備考 :	なし
ソース・ファイル :	isWU.c
コード例 :	char reset_state; reset_state = isWU();

3.8.2 個別マクロ

DisableInterrupts

説明 :	このマクロは、指定された命令サイクル数の間すべてのペリフェラル割り込みをディスエーブルします。
インクルード :	reset.h
引数 :	サイクル数
備考 :	このマクロは DISI 命令を実行して、指定された命令サイクル数の間すべてのペリフェラル割り込みをディスエーブルします。
コード例 :	DisableInterrupts(15);

PORStatReset

説明 : このマクロは、RCON レジスタの POR ビットをリセット状態に設定します。

インクルード : `reset.h`

引数 : なし

備考 : なし

コード例 : `PORStatReset;`

BORStatReset

説明 : このマクロは、RCON レジスタの BOR ビットをリセット状態に設定します。

インクルード : `reset.h`

引数 : なし

備考 : なし

コード例 : `BORStatReset;`

WDTSWEnable

説明 : このマクロは、ウォッチドッグ・タイマをターンオンします。

インクルード : `reset.h`

引数 : なし

備考 : このマクロは、RCON レジスタのソフトウェア WDT イネーブル (SWDTEN) ビットをセットします。

コード例 : `WDTSWEnable;`

WDTSWDisable

説明 : このマクロは、RCON レジスタのソフトウェア WDT イネーブル (SWDTEN) ビットをクリアします。

インクルード : `reset.h`

引数 : なし

備考 : このマクロは、FWDTEN Fuse ビットが '0' の場合に WDT をディスエーブルします。

コード例 : `WDTSWDisable;`

3.9 I/O ポート関数

このセクションには、I/O ポートの個別関数の一覧を記載します。関数はマクロとして構成可能です。

3.9.1 個別関数

CloseINT0
CloseINT1
CloseINT2
CloseINT3
CloseINT4

説明 : この関数は、INT ピンの外部割り込みをディスエーブルします。

インクルード : ports.h

プロトタイプ :
```
void CloseINT0(void);
void CloseINT1(void);
void CloseINT2(void);
void CloseINT3(void);
void CloseINT4(void);
```

引数 : なし

戻り値 : なし

備考 : この関数は、INT ピンの割り込みをディスエーブルして、対応する割り込みフラグをクリアします。

ソース・ファイル :
```
CloseInt0.c
CloseInt1.c
CloseInt2.c
CloseInt3.c
CloseInt4.c
```

コード例 :
```
CloseINT0();
```

ConfigINT0
ConfigINT1
ConfigINT2
ConfigINT3
ConfigINT4

説明 : この関数は、INT ピンの割り込みを設定します。

インクルード : ports.h

プロトタイプ :
```
void ConfigINT0(unsigned int config);
void ConfigINT1(unsigned int config);
void ConfigINT2(unsigned int config);
void ConfigINT3(unsigned int config);
void ConfigINT4(unsigned int config);
```

引数 :

config 次のように定義される割り込みエッジ、優先順位、イネーブル / ディスエーブル情報 :

割り込みエッジの選択
RISING_EDGE_INT
FALLING_EDGE_INT

割り込みイネーブル
INT_ENABLE
INT_DISABLE

ConfigINT0
ConfigINT1
ConfigINT2
ConfigINT3
ConfigINT4 (続き)

割り込み優先順位
INT_PRI_0
INT_PRI_1
INT_PRI_2
INT_PRI_3
INT_PRI_4
INT_PRI_5
INT_PRI_6
INT_PRI_7

戻り値 : なし

備考 : この関数は、INTx ピンに対応する割り込みフラグをクリアした後に、エッジ検出極性を選択します。
次に、割り込み優先順位を設定して、割り込みをイネーブル / ディスエーブルします。

ソース・ファイル :
```
ConfigInt0.c
ConfigInt1.c
ConfigInt2.c
ConfigInt3.c
ConfigInt4.c
```

コード例 :
```
ConfigINT0(RISING_EDGE_INT & EXT_INT_PRI_5 &
EXT_INT_ENABLE);
```

ConfigCNPullups

説明 : この関数は、CN ピンのプルアップ抵抗を設定します。

インクルード : `ports.h`

プロトタイプ : `void ConfigCNPullups(long int config);`

引数 :

config	プルアップを設定する 32 ビット値。下位ワードは CNPU1 レジスタに、上位ワードは CNPU2 レジスタに、それぞれ格納されます。CNPU2 レジスタの上位 8 ビットは、実装されていません。

戻り値 : なし

備考 : なし

ソース・ファイル : `ConfigCNPullups.c`

コード例 :
```
ConfigCNPullups(0xFFF);
```

ConfigIntCN

説明 :	この関数は、CN の割り込みを設定します。
インクルード :	ports.h
プロトタイプ :	void ConfigIntCN(long int *config*);
引数 :	*config* CN 割り込みを設定する 32 ビット値 下位 24 ビットには、CN 割り込みに対する個別イネーブル / ディスエーブル情報が含まれます。ビット x (x = 0, 1, ..., 23) をセットすると、CNx 割り込みがイネーブルされます。 *config* の最上位バイトには、割り込み優先順位とイネーブル / ディスエーブル・ビットが含まれます。 下位ワードは CNEN1 レジスタに、次の上位バイトは CNEN2 レジスタにそれぞれ格納され、最上位バイトは優先順位と CN 割り込みのイネーブル / ディスエーブルの設定に使われます。
戻り値 :	なし
備考 :	この関数は CN 割り込みフラグをクリアし、CN ピンの個別割り込みをイネーブル / ディスエーブルします。 また、割り込み優先順位を設定し、CN 割り込みイネーブル・ビットをイネーブル / ディスエーブルします。
ソース・ファイル :	ConfigIntCN.c
コード例 :	`// This would enable CN0, CN1, CN2 and CN7 only.` `ConfigIntCN(CHANGE_INT_OFF & CHANGE_INT_PRI_4 & 0xFF000087);`

3.9.2 個別マクロ

EnableCN0
EnableCN1
EnableCN2
.....
EnableCN23

説明 :	このマクロは、個別変更通知割り込みをイネーブルします。
インクルード :	ports.h
引数 :	なし
備考 :	なし
コード例 :	EnableCN6;

DisableCN0
DisableCN1
DisableCN2
.....
DisableCN23

説明 :	このマクロは、個別変更通知割り込みをディスエーブルします。
インクルード :	ports.h
引数 :	なし
備考 :	なし
コード例 :	DisableCN14;

EnableINT0
EnableINT1
EnableINT2
EnableINT3
EnableINT4

説明:	このマクロは、個別外部割り込みをイネーブルします。
インクルード:	ports.h
引数:	なし
備考:	なし
コード例:	EnableINT2;

DisableINT0
DisableINT1
DisableINT2
DisableINT3
DisableINT4

説明:	このマクロは、個別外部割り込みをディスエーブルします。
インクルード:	ports.h
引数:	なし
備考:	なし
コード例:	DisableINT2;

SetPriorityInt0
SetPriorityInt1
SetPriorityInt2
SetPriorityInt3
SetPriorityInt4

説明:	このマクロは、外部割り込みの優先順位を設定します。
インクルード:	ports.h
引数:	*priority*
備考:	このマクロは、割り込み優先順位コントロール・レジスタの外部割り込み優先順位ビットをセットします。
コード例:	SetPriorityInt4(6);

3.10 入力キャプチャ関数

このセクションには、入力キャプチャ・モジュール用個別関数の一覧と、各関数の使用例を記載します。関数はマクロとして構成可能です。

3.10.1 個別関数

CloseCapture1
CloseCapture2
CloseCapture3
CloseCapture4
CloseCapture5
CloseCapture6
CloseCapture7
CloseCapture8

説明 :	この関数は、入力キャプチャ・モジュールをターンオフします。
インクルード :	InCap.h

プロトタイプ :	void CloseCapture1(void);
	void CloseCapture2(void);
	void CloseCapture3(void);
	void CloseCapture4(void);
	void CloseCapture5(void);
	void CloseCapture6(void);
	void CloseCapture7(void);
	void CloseCapture8(void);

引数 :	なし
戻り値 :	なし
備考 :	この関数は入力キャプチャ割り込みをディスエーブルした後に、モジュールをターンオフします。割り込みフラグ・ビットもクリアします。
ソース・ファイル :	CloseCapture1.c CloseCapture2.c CloseCapture3.c CloseCapture4.c CloseCapture5.c CloseCapture6.c CloseCapture7.c CloseCapture8.c
コード例 :	CloseCapture1();

ConfigIntCapture1
ConfigIntCapture2
ConfigIntCapture3
ConfigIntCapture4
ConfigIntCapture5
ConfigIntCapture6
ConfigIntCapture7
ConfigIntCapture8

説明 : この関数は、入力キャプチャ割り込みを設定します。

インクルード : InCap.h

プロトタイプ :

```
void ConfigIntCapture1(unsigned int config);
void ConfigIntCapture2(unsigned int config);
void ConfigIntCapture3(unsigned int config);
void ConfigIntCapture4(unsigned int config);
void ConfigIntCapture5(unsigned int config);
void ConfigIntCapture6(unsigned int config);
void ConfigIntCapture7(unsigned int config);
void ConfigIntCapture8(unsigned int config);
```

引数 :

config	次のように定義される入力キャプチャ割り込み優先順位とイネーブル / ディスエーブル情報 :

割り込みイネーブル / ディスエーブル
IC_INT_ON
IC_INT_OFF

割り込み優先順位
IC_INT_PRIOR_0
IC_INT_PRIOR_1
IC_INT_PRIOR_2
IC_INT_PRIOR_3
IC_INT_PRIOR_4
IC_INT_PRIOR_5
IC_INT_PRIOR_6
IC_INT_PRIOR_7

戻り値 : なし

備考 : この関数は、割り込みフラグビットをクリアした後に、割り込み優先順位を設定して、割り込みをイネーブル / ディスエーブルします。

ソース・ファイル :

ConfigIntCapture1.c
ConfigIntCapture2.c
ConfigIntCapture3.c
ConfigIntCapture4.c
ConfigIntCapture5.c
ConfigIntCapture6.c
ConfigIntCapture7.c
ConfigIntCapture8.c

コード例 :

```
ConfigIntCapture1(IC_INT_ON & IC_INT_PRIOR_1);
```

OpenCapture1
OpenCapture2
OpenCapture3
OpenCapture4
OpenCapture5
OpenCapture6
OpenCapture7
OpenCapture8

説明 : この関数は、入力キャプチャ・モジュールを設定します。

インクルード : InCap.h

プロトタイプ :

```
void OpenCapture1(unsigned int config);
void OpenCapture2(unsigned int config);
void OpenCapture3(unsigned int config);
void OpenCapture4(unsigned int config);
void OpenCapture5(unsigned int config);
void OpenCapture6(unsigned int config);
void OpenCapture7(unsigned int config);
void OpenCapture8(unsigned int config);
```

引数 :

config ICxCON レジスタに設定される次のように定義されたパラメータ :

アイドル・モード動作
IC_IDLE_CON
IC_IDLE_STOP

クロックの選択
IC_TIMER2_SRC
IC_TIMER3_SRC

割り込み毎のキャプチャ
IC_INT_4CAPTURE
IC_INT_3CAPTURE
IC_INT_2CAPTURE
IC_INT_1CAPTURE
IC_INTERRUPT

IC モードの選択
IC_EVERY_EDGE
IC_EVERY_16_RISE_EDGE
IC_EVERY_4_RISE_EDGE
IC_EVERY_RISE_EDGE
IC_EVERY_FALL_EDGE
IC_INPUTCAP_OFF

戻り値 : なし

備考 : この関数は、クロック選択、割り込み毎のキャプチャ、キャプチャ動作モードの各パラメータを入力キャプチャ・モジュール・コントロール・レジスタ (ICxCON) に設定します。

ソース・ファイル :
OpenCapture1.c
OpenCapture2.c
OpenCapture3.c
OpenCapture4.c
OpenCapture5.c
OpenCapture6.c
OpenCapture7.c
OpenCapture8.c

コード例 :

```
OpenCapture1(IC_IDLE_CON & IC_TIMER2_SRC &
IC_INT_1CAPTURE & IC_EVERY_RISE_EDGE);
```

ReadCapture1
ReadCapture2
ReadCapture3
ReadCapture4
ReadCapture5
ReadCapture6
ReadCapture7
ReadCapture8

説明: この関数は、すべての待ち状態の入力キャプチャ・バッファを読み出します。

インクルード: InCap.h

プロトタイプ:
```
void ReadCapture1(unsigned int *buffer);
void ReadCapture2(unsigned int *buffer);
void ReadCapture3(unsigned int *buffer);
void ReadCapture4(unsigned int *buffer);
void ReadCapture5(unsigned int *buffer);
void ReadCapture6(unsigned int *buffer);
void ReadCapture7(unsigned int *buffer);
void ReadCapture8(unsigned int *buffer);
```

引数:

buffer	入力キャプチャ・バッファから読み出したデータを格納するロケーションを指すポインタ

戻り値: なし

備考: この関数は、ICxCON<ICBNE> ビットがクリアされてバッファがエンプティであることが表示されるまで、すべての待ち状態の入力キャプチャ・バッファを読み出します。

ソース・ファイル:
```
ReadCapture1.c
ReadCapture2.c
ReadCapture3.c
ReadCapture4.c
ReadCapture5.c
ReadCapture6.c
ReadCapture7.c
ReadCapture8.c
```

コード例:
```
unsigned int *buffer = 0x1900;
ReadCapture1(buffer);
```

3.10.2 個別マクロ

EnableIntIC1
EnableIntIC2
EnableIntIC3
EnableIntIC4
EnableIntIC5
EnableIntIC6
EnableIntIC7
EnableIntIC8

説明 :	このマクロは、キャプチャ・イベントの割り込みをイネーブルします。
インクルード :	`InCap.h`
引数 :	なし
備考 :	このマクロは、割り込みイネーブル・コントロール・レジスタの入力キャプチャ割り込みイネーブル・ビットをセットします。
コード例 :	`EnableIntIC7;`

DisableIntIC1
DisableIntIC2
DisableIntIC3
DisableIntIC4
DisableIntIC5
DisableIntIC6
DisableIntIC7
DisableIntIC8

説明 :	このマクロは、キャプチャ・イベントの割り込みをディスエーブルします。
インクルード :	`InCap.h`
引数 :	なし
備考 :	このマクロは、割り込みイネーブル・コントロール・レジスタの入力キャプチャ割り込みイネーブル・ビットをクリアします。
コード例 :	`DisableIntIC7;`

SetPriorityIntIC1
SetPriorityIntIC2
SetPriorityIntIC3
SetPriorityIntIC4
SetPriorityIntIC5
SetPriorityIntIC6
SetPriorityIntIC7
SetPriorityIntIC8

説明 :	このマクロは、入力キャプチャ割り込みの優先順位を設定します。
インクルード :	`InCap.h`
引数 :	*priority*
備考 :	このマクロは、割り込み優先順位コントロール・レジスタの入力キャプチャ割り込み優先順位ビットをセットします。
コード例 :	`SetPriorityIntIC4(1);`

3.10.3 使用例

```
#define _ _dsPIC30F6014_ _
#include <p30fxxxx.h>
#include<InCap.h>
int Interrupt_Count = 0 , Int_flag, count;
unsigned int timer_first_edge, timer_second_edge;
void _ _attribute_ _((_ _interrupt_ _)) _IC1Interrupt(void)
{
    Interrupt_Count++;
    if(Interrupt_Count == 1)
        ReadCapture1(&timer_first_edge);
    else if(Interrupt_Count == 2)
        ReadCapture1(&timer_second_edge);
    Int_flag = 1;
    IFS0bits.IC1IF = 0;
}
int main(void)
{
    unsigned int period;
    Int_flag = 0;
    TRISDbits.TRISD0 = 0; /* Alarm output on RD0 */
    PORTDbits.RD0 = 1;
/* Enable Timer1 Interrupt and Priority to '1' */
    ConfigIntCapture1(IC_INT_PRIOR_1 & IC_INT_ON);
    T3CON = 0x8000; /* Timer 3 On */
/* Configure the InputCapture in stop in idle mode , Timer
   3 as source , interrupt on capture 1, I/C on every fall
   edge */

    OpenCapture1(IC_IDLE_STOP & IC_TIMER3_SRC &
                 IC_INT_1CAPTURE & IC_EVERY_FALL_EDGE);
    while(1)
    {
        while(!Int_flag); /* wait here till first capture event */
        Int_flag = 0;
        while(!Int_flag); /* wait here till next capture event */
/* calculate time count between two capture events */
        period = timer_second_edge -  timer_first_edge;
/* if the time count between two capture events is more than
   0x200 counts, set alarm on RD0 */
       if(period >= 0x200)
       {
/* set alarm and wait for sometime and clear alarm */
            PORTDbits.RD0 = 0;
            while(count <= 0x10)
            {
                count++;
            }
            PORTDbits.RD0 = 1;
        }
        Interrupt_Count = 0;
        count = 0;
    }
    CloseCapture1();
}
```

3.11 出力コンペア関数

このセクションには、出力コンペア・モジュール用個別関数の一覧と、各関数の使用例を記載します。関数はマクロとして構成可能です。

3.11.1 個別関数

CloseOC1
CloseOC2
CloseOC3
CloseOC4
CloseOC5
CloseOC6
CloseOC7
CloseOC8

説明 :	この関数は、出力コンペア・モジュールをターンオフします。
インクルード :	outcompare.h

プロトタイプ :	void CloseOC1(void);
	void CloseOC2(void);
	void CloseOC3(void);
	void CloseOC4(void);
	void CloseOC5(void);
	void CloseOC6(void);
	void CloseOC7(void);
	void CloseOC8(void);

引数 :	なし
戻り値 :	なし
備考 :	この関数は、出力コンペア割り込みをディスエーブルした後に、モジュールをターンオフします。割り込みフラグ・ビットもクリアします。
ソース・ファイル :	CloseOC1.c CloseOC2.c CloseOC3.c CloseOC4.c CloseOC5.c CloseOC6.c CloseOC7.c CloseOC8.c
コード例 :	CloseOC1();

ConfigIntOC1
ConfigIntOC2
ConfigIntOC3
ConfigIntOC4
ConfigIntOC5
ConfigIntOC6
ConfigIntOC7
ConfigIntOC8

説明 : この関数は、出力コンペア割り込みを設定します。

インクルード : outcompare.h

プロトタイプ :

```
void ConfigIntOC1(unsigned int config);
void ConfigIntOC2(unsigned int config);
void ConfigIntOC3(unsigned int config);
void ConfigIntOC4(unsigned int config);
void ConfigIntOC5(unsigned int config);
void ConfigIntOC6(unsigned int config);
void ConfigIntOC7(unsigned int config);
void ConfigIntOC8(unsigned int config);
```

引数 :

config 次のように定義される出力コンペア割り込み優先順位とイネーブル / ディスエーブル情報 :

割り込みイネーブル / ディスエーブル
OC_INT_ON
OC_INT_OFF

割り込み優先順位
OC_INT_PRIOR_0
OC_INT_PRIOR_1
OC_INT_PRIOR_2
OC_INT_PRIOR_3
OC_INT_PRIOR_4
OC_INT_PRIOR_5
OC_INT_PRIOR_6
OC_INT_PRIOR_7

戻り値 : なし

備考 : この関数は、割り込みフラグビットをクリアした後に、割り込み優先順位を設定して、割り込みをイネーブル / ディスエーブルします。

ソース・ファイル :
ConfigIntOC1.c
ConfigIntOC2.c
ConfigIntOC3.c
ConfigIntOC4.c
ConfigIntOC5.c
ConfigIntOC6.c
ConfigIntOC7.c
ConfigIntOC8.c

コード例 :

```
ConfigIntOC1(OC_INT_ON & OC_INT_PRIOR_2);
```

OpenOC1
OpenOC2
OpenOC3
OpenOC4
OpenOC5
OpenOC6
OpenOC7
OpenOC8

説明 : この関数は、出力コンペア・モジュールを設定します。

インクルード : outcompare.h

プロトタイプ :

```
void OpenOC1(unsigned int config,
  unsigned int value1, unsigned int value2);
void OpenOC2(unsigned int config,
  unsigned int value1, unsigned int value2);
void OpenOC3(unsigned int config,
  unsigned int value1, unsigned int value2);
void OpenOC4(unsigned int config,
  unsigned int value1, unsigned int value2);
void OpenOC5(unsigned int config,
  unsigned int value1, unsigned int value2);
void OpenOC6(unsigned int config,
  unsigned int value1, unsigned int value2);
void OpenOC7(unsigned int config,
  unsigned int value1, unsigned int value2);
void OpenOC8(unsigned int config,
  unsigned int value1, unsigned int value2);
```

引数 :

config	OCxCON レジスタに設定される次のように定義されたパラメータ : アイドル・モード動作 OC_IDLE_STOP OC_IDLE_CON クロックの選択 OC_TIMER2_SRC OC_TIMER3_SRC 出力コンペアの動作モード OC_PWM_FAULT_PIN_ENABLE OC_PWM_FAULT_PIN_DISABLE OC_CONTINUE_PULSE OC_SINGLE_PULSE OC_TOGGLE_PULSE OC_HIGH_LOW OC_LOW_HIGH OC_OFF
value1	OCxRS セカンダリ・レジスタへ格納する値
value2	OCxR メイン・レジスタへ格納する値

戻り値 : なし

OpenOC1
OpenOC2
OpenOC3
OpenOC4
OpenOC5
OpenOC6
OpenOC7
OpenOC8 (続き)

備考 :	この関数は、次のパラメータを出力コンペア・モジュール・コントロール・レジスタ (OCxCON) に設定します : クロック選択、動作モード、アイドル・モード時の動作。 また、OCxRS レジスタと OCxR レジスタも設定します。
ソース・ファイル :	OpenOC1.c OpenOC2.c OpenOC3.c OpenOC4.c OpenOC5.c OpenOC6.c OpenOC7.c OpenOC8.c
コード例 :	`OpenOC1(OC_IDLE_CON & OC_TIMER2_SRC & OC_PWM_FAULT_PIN_ENABLE, 0x80, 0x60);`

ReadDCOC1PWM
ReadDCOC2PWM
ReadDCOC3PWM
ReadDCOC4PWM
ReadDCOC5PWM
ReadDCOC6PWM
ReadDCOC7PWM
ReadDCOC8PWM

説明 : この関数は、出力コンペア・セカンダリ・レジスタからデューティ・サイクルを読み出します。

インクルード : outcompare.h

プロトタイプ :
```
unsigned int ReadDCOC1PWM(void);
unsigned int ReadDCOC2PWM(void);
unsigned int ReadDCOC3PWM(void);
unsigned int ReadDCOC4PWM(void);
unsigned int ReadDCOC5PWM(void);
unsigned int ReadDCOC6PWM(void);
unsigned int ReadDCOC7PWM(void);
unsigned int ReadDCOC8PWM(void);
```

引数 : なし

戻り値 : この関数は、出力コンペア・モジュールが PWM モードのときの OCxRS レジスタ値を返します。その他の場合は、'-1' を返します。

備考 : この関数は、出力コンペア・モジュールが PWM モードのときに、出力コンペア・セカンダリ・レジスタ (OCxRS) からデューティ・サイクルを読み出します。
PWM モードでない場合は、この関数は値 '-1' を返します。

ソース・ファイル :
```
ReadDCOC1PWM.c
ReadDCOC2PWM.c
ReadDCOC3PWM.c
ReadDCOC4PWM.c
ReadDCOC5PWM.c
ReadDCOC6PWM.c
ReadDCOC7PWM.c
ReadDCOC8PWM.c
```

コード例 :
```
unsigned int compare_reg;
compare_reg = ReadDCOC1PWM();
```

ReadRegOC1
ReadRegOC2
ReadRegOC3
ReadRegOC4
ReadRegOC5
ReadRegOC6
ReadRegOC7
ReadRegOC8

説明 : この関数は、出力コンペア・モジュールが PWM モードでないときに、デューティ・サイクル・レジスタを読み出します。

インクルード : outcompare.h

プロトタイプ :
```
unsigned int ReadRegOC1(char reg);
unsigned int ReadRegOC2(char reg);
unsigned int ReadRegOC3(char reg);
unsigned int ReadRegOC4(char reg);
unsigned int ReadRegOC5(char reg);
unsigned int ReadRegOC6(char reg);
unsigned int ReadRegOC7(char reg);
unsigned int ReadRegOC8(char reg);
```

引数 :

reg	出力コンペア・モジュールのメインまたはセカンダリ・デューティ・サイクル・レジスタを読み出したか否かを表示します。 *reg* が '1' の場合、メイン・デューティ・サイクル・レジスタ (OCxR) が読み出されました。 *reg* が '0' の場合、セカンダリ・デューティ・サイクル・レジスタ (OCxRS) が読み出されました。

戻り値 : *reg* が '1' の場合、メイン・デューティ・サイクル・レジスタ (OCxR) が読み出されました。
reg が '0' の場合、セカンダリ・デューティ・サイクル・レジスタ (OCxRS) が読み出されました。
出力コンペア・モジュールが PWM モードの場合、'-1' が返されます。

備考 : 出力コンペア・モジュールが PWM モードでない場合にのみ、デューティ・サイクル・レジスタの読み出しが発生します。その他の場合は、値 '-1' を返します。

ソース・ファイル :
```
ReadRegOC1.c
ReadRegOC2.c
ReadRegOC3.c
ReadRegOC4.c
ReadRegOC5.c
ReadRegOC6.c
ReadRegOC7.c
ReadRegOC8.c
```

コード例 :
```
unsigned int dutycycle_reg;
dutycycle_reg = ReadRegOC1(1);
```

SetDCOC1PWM
SetDCOC2PWM
SetDCOC3PWM
SetDCOC4PWM
SetDCOC5PWM
SetDCOC6PWM
SetDCOC7PWM
SetDCOC8PWM

説明 : この関数は、モジュールが PWM モードのときに、出力コンペア・セカンダリ・デューティ・サイクル・レジスタ (OCxRS) を設定します。

インクルード : `outcompare.h`

プロトタイプ :

```
void SetDCOC1PWM(unsigned int dutycycle);
void SetDCOC2PWM(unsigned int dutycycle);
void SetDCOC3PWM(unsigned int dutycycle);
void SetDCOC4PWM(unsigned int dutycycle);
void SetDCOC5PWM(unsigned int dutycycle);
void SetDCOC6PWM(unsigned int dutycycle);
void SetDCOC7PWM(unsigned int dutycycle);
void SetDCOC8PWM(unsigned int dutycycle);
```

引数 :

dutycycle	出力コンペア・セカンダリ・デューティ・サイクル・レジスタ (OCxRS) へ格納されるデューティ・サイクル値

戻り値 : なし

備考 : モジュールが PWM モードの場合にのみ、出力コンペア・セカンダリ・デューティ・サイクル・レジスタ (OCxRS) に新しい値が設定されます。

ソース・ファイル :

```
SetDCOC1PWM.c
SetDCOC2PWM.c
SetDCOC3PWM.c
SetDCOC4PWM.c
SetDCOC5PWM.c
SetDCOC6PWM.c
SetDCOC7PWM.c
SetDCOC8PWM.c
```

コード例 :

```
SetDCOC1PWM(dutycycle);
```

SetPulseOC1
SetPulseOC2
SetPulseOC3
SetPulseOC4
SetPulseOC5
SetPulseOC6
SetPulseOC7
SetPulseOC8

説明 : この関数は、モジュールが PWM モードのときに、出力コンペア・メイン・レジスタと出力コンペア・セカンダリ・レジスタ (OCxR と OCxRS) を設定します。

インクルード : outcompare.h

プロトタイプ :

```
void SetPulseOC1(unsigned int pulse_start,
  unsigned int pulse_stop);
void SetPulseOC2(unsigned int pulse_start,
  unsigned int pulse_stop);
void SetPulseOC3(unsigned int pulse_start,
  unsigned int pulse_stop);
void SetPulseOC4(unsigned int pulse_start,
  unsigned int pulse_stop);
void SetPulseOC5(unsigned int pulse_start,
  unsigned int pulse_stop);
void SetPulseOC6(unsigned int pulse_start,
  unsigned int pulse_stop);
void SetPulseOC7(unsigned int pulse_start,
  unsigned int pulse_stop);
void SetPulseOC8(unsigned int pulse_start,
  unsigned int pulse_stop);
```

引数 :

pulse_start 出力コンペアメイン・レジスタ (OCxR) へ格納する値

pulse_stop 出力コンペア・セカンダリ・レジスタ (OCxRS) へ格納する値

戻り値 : なし

備考 : モジュールが PWM モードでない場合にのみ、出力コンペア・デューティ・サイクル・レジスタ (OCxR と OCxRS) に新しい値が格納されます。

ソース・ファイル :

```
SetPulseOC1.c
SetPulseOC2.c
SetPulseOC3.c
SetPulseOC4.c
SetPulseOC5.c
SetPulseOC6.c
SetPulseOC7.c
SetPulseOC8.c
```

コード例 :

```
pulse_start = 0x40;
pulse_stop  = 0x60;
SetPulseOC1(pulse_start, pulse_stop);
```

3.11.2 個別マクロ

EnableIntOC1
EnableIntOC2
EnableIntOC3
EnableIntOC4
EnableIntOC5
EnableIntOC6
EnableIntOC7
EnableIntOC8

説明 :	このマクロは、出力コンペア一致割り込みをイネーブルします。
インクルード :	`outcompare.h`
引数 :	なし
備考 :	このマクロは、割り込みイネーブル・コントロール・レジスタの出力コンペア (OC) 割り込みイネーブル・ビットを設定します。
コード例 :	`EnableIntOC8;`

DisableIntOC1
DisableIntOC2
DisableIntOC3
DisableIntOC4
DisableIntOC5
DisableIntOC6
DisableIntOC7
DisableIntOC8

説明 :	このマクロは、出力コンペア一致割り込みをディスエーブルします。
インクルード :	`outcompare.h`
引数 :	なし
備考 :	このマクロは、割り込みイネーブル・コントロール・レジスタの OC 割り込みイネーブル・ビットをクリアします。
コード例 :	`DisableIntOC7;`

SetPriorityIntIC1
SetPriorityIntIC2
SetPriorityIntIC3
SetPriorityIntIC4
SetPriorityIntIC5
SetPriorityIntIC6
SetPriorityIntIC7
SetPriorityIntIC8

説明 :	このマクロは、出力コンペア割り込みの優先順位を設定します。
インクルード :	`outcompare.h`
引数 :	*priority*
備考 :	このマクロは、割り込み優先順位コントロール・レジスタの OC 割り込み優先順位ビットをセットします。
コード例 :	`SetPriorityIntOC4(0);`

3.11.3 使用例

```
#define _ _dsPIC30F6014_ _
#include<p30fxxxx.h>
#include<outcompare.h>
/* This is ISR corresponding to OC1 interrupt */
void _ _attribute_ _((_ _interrupt_ _)) _OC1Interrupt(void)
{
  IFS0bits.OC1IF = 0;
}
int main(void)
{
/* Holds the value at which OCx Pin to be driven high */
unsigned int pulse_start ;
/* Holds the value at which OCx Pin to be driven low */
unsigned int pulse_stop;
/* Turn off OC1 module */
    CloseOC1();
/* Configure output compare1 interrupt */
ConfigIntOC1(OC_INT_OFF & OC_INT_PRIOR_5);
/* Configure OC1 module for required pulse width */
    pulse_start = 0x40;
    pulse_stop = 0x60;
    PR3 = 0x80 ;
    PR1 = 0xffff;
    TMR1  = 0x0000;
    T3CON = 0x8000;
    T1CON = 0X8000;
/* Configure Output Compare module to 'initialise OCx pin
low and generate continuous pulse'mode */
    OpenOC1(OC_IDLE_CON & OC_TIMER3_SRC &
            OC_CONTINUE_PULSE,
            pulse_stop, pulse_start);
/* Generate continuous pulse till TMR1 reaches 0xff00 */
    while(TMR1<= 0xff00);
    asm("nop");
    CloseOC1();
    return 0;
}
```

3.12 UART 関数

このセクションには、UART モジュール用個別関数の一覧と、各関数の使用例を記載します。関数はマクロとして構成可能です。

3.12.1 個別関数

BusyUART1
BusyUART2

説明 :	この関数は、UART 転送ステータスを返します
インクルード :	`uart.h`
プロトタイプ :	`char BusyUART1(void);` `char BusyUART2(void);`
引数 :	なし
戻り値 :	`'1'` が返された場合、UART が転送中のためビジーで、かつ UxSTA<TRMT> ビットが `'0'` であることを表示します。 `'0'` が返された場合、UART がビジーでなく、かつ UxSTA<TRMT> ビットが `'1'` であることを表示します。
備考 :	この関数は、UART のステータスを返します。これは、UxSTA<TRMT> ビットが示すように、UART が転送中のためビジーであることを表示します。
ソース・ファイル :	`BusyUART1.c` `BusyUART2.c`
コード例 :	`while(BusyUART1());`

CloseUART1
CloseUART2

説明 :	この関数は、UART モジュールをターンオフします。
インクルード :	`uart.h`
プロトタイプ :	`void CloseUART1(void);` `void CloseUART2(void);`
引数 :	なし
戻り値 :	なし
備考 :	この関数は UART モジュールをターンオフした後に、UART 送信割り込みと UART 受信割り込みをディスエーブルします。また、割り込みフラグ・ビットもクリアします。
ソース・ファイル :	`CloseUART1.c` `CloseUART2.c`
コード例 :	`CloseUART1();`

ConfigIntUART1
ConfigIntUART2

説明 : この関数は、UART 割り込みを設定します。

インクルード : `uart.h`

プロトタイプ :
```
void ConfigIntUART1(unsigned int config);
void ConfigIntUART2(unsigned int config);
```

引数 : *config* 次のように定義された個別割り込みイネーブル / ディスエーブル情報 :

受信割り込みイネーブル
UART_RX_INT_EN
UART_RX_INT_DIS

受信割り込み優先順位
UART_RX_INT_PR0
UART_RX_INT_PR1
UART_RX_INT_PR2
UART_RX_INT_PR3
UART_RX_INT_PR4
UART_RX_INT_PR5
UART_RX_INT_PR6
UART_RX_INT_PR7

送信割り込みイネーブル
UART_TX_INT_EN
UART_TX_INT_DIS

送信割り込み優先順位
UART_TX_INT_PR0
UART_TX_INT_PR1
UART_TX_INT_PR2
UART_TX_INT_PR3
UART_TX_INT_PR4
UART_TX_INT_PR5
UART_TX_INT_PR6
UART_TX_INT_PR7

戻り値 : なし

備考 : この関数は、UART 送信割り込みと UART 受信割り込みをイネーブル / ディスエーブルして、割り込み優先順位を設定します。

ソース・ファイル :
ConfigIntUART1.c
ConfigIntUART2.c

コード例 :
```
ConfigIntUART1(UART_RX_INT_EN & UART_RX_INT_PR5 &
  UART_TX_INT_EN & UART_TX_INT_PR3);
```

DataRdyUART1
DataRdyUART2

説明 : この関数は、UART 受信バッファ・ステータスを返します。

インクルード : uart.h

プロトタイプ :
```
char DataRdyUART1(void);
char DataRdyUART2(void);
```

引数 : なし

戻り値 : '1' が返された場合、受信バッファには読み出すデータが存在していることを表わします。
'0' が返された場合、受信バッファには読み出すデータが存在していないことを表わします。

備考 : この関数は、UART 受信バッファのステータスを返します。
UxSTA<URXDA> ビットが示すように、UART 受信バッファに読み出すべき新しいデータが存在しているか否かを表示します。

ソース・ファイル : DataRdyUART1.c
DataRdyUART2.c

コード例 :
```
while(DataRdyUART1());
```

OpenUART1
OpenUART2

説明 : この関数は、UART モジュールを設定します。

インクルード : uart.h

プロトタイプ :
```
void OpenUART1(unsigned int config1,
  unsigned int config2, unsigned int ubrg);
void OpenUART2(unsigned int config1,
  unsigned int config2, unsigned int ubrg);
```

引数 : *config1* UxMODE レジスタに設定される次のように定義されたパラメータ :

UART イネーブル / ディスエーブル
UART_EN
UART_DIS

UART アイドル・モード動作
UART_IDLE_CON
UART_IDLE_STOP

ALT ピンとの UART 交信
UART_ALTRX_ALTTX
UART_RX_TX
ALT ピンとの UART 交信は、所定のデバイスでのみ使用可能です。該当するデータシートを参照してください。

スタート時の UART ウェイクアップ
UART_EN_WAKE
UART_DIS_WAKE

UART ループバック・モードのイネーブル / ディスエーブル
UART_EN_LOOPBACK
UART_DIS_LOOPBACK

OpenUART1
OpenUART2 (続き)

キャプチャ・モジュールへの入力
UART_EN_ABAUD
UART_DIS_ABAUD

パリティ・ビットとデータ・ビットの選択
UART_NO_PAR_9BIT
UART_ODD_PAR_8BIT
UART_EVEN_PAR_8BIT
UART_NO_PAR_8BIT

ストップ・ビット数
UART_2STOPBITS
UART_1STOPBIT

config2 — UxSTA レジスタに設定される次のように定義されたパラメータ :

UART 転送モード割り込みの選択
UART_INT_TX_BUF_EMPTY
UART_INT_TX

UART 送信ブレーク・ビット
UART_TX_PIN_NORMAL
UART_TX_PIN_LOW

UART 送信イネーブル / ディスエーブル
UART_TX_ENABLE
UART_TX_DISABLE

UART 受信割り込みモードの選択
UART_INT_RX_BUF_FUL
UART_INT_RX_3_4_FUL
UART_INT_RX_CHAR

UART アドレス検出のイネーブル / ディスエーブル
UART_ADR_DETECT_EN
UART_ADR_DETECT_DIS

UART OVERRUN ビットのクリア
UART_RX_OVERRUN_CLEAR

ubrg — ボーレートを設定するために UxBRG レジスタに書き込む値

戻り値 : なし

備考 : この関数は、UART 送信セクションと UART 受信セクションを設定し、交信ボーレートを設定します。

ソース・ファイル :
OpenUART1.c
OpenUART2.c

コード例 :

```
baud = 5;
UMODEvalue = UART_EN & UART_IDLE_CON &
             UART_DIS_WAKE & UART_EN_LOOPBACK &
             UART_EN_ABAUD & UART_NO_PAR_8BIT &
             UART_1STOPBIT;
U1STAvalue = UART_INT_TX_BUF_EMPTY &
             UART_TX_PIN_NORMAL &
             UART_TX_ENABLE &
             UART_INT_RX_3_4_FUL &
             UART_ADR_DETECT_DIS &
             UART_RX_OVERRUN_CLEAR;
OpenUART1(U1MODEvalue, U1STAvalue, baud);
```

ReadUART1
ReadUART2

説明: この関数は、UART 受信バッファ (UxRXREG) レジスタ値を返します。

インクルード: uart.h

プロトタイプ:
```
unsigned int ReadUART1(void);
unsigned int ReadUART2(void);
```

引数: なし

戻り値: この関数は、受信バッファ (UxRXREG) レジスタ値を返します。

備考: この関数は、受信バッファレジスタ値を返します。
9 ビット受信がイネーブルされている場合は、レジスタの全値が返されます。
8 ビット受信がイネーブルされている場合は、レジスタが読み出され、9 ビット目はマスクされます。

ソース・ファイル:
ReadUART1.c
ReadUART2.c

コード例:
```
unsigned int RX_data;
RX_data = ReadUART1();
```

WriteUART1
WriteUART2

説明: この関数は、送信バッファ (UxTXREG) レジスタへ転送されるデータを書き込みます。

インクルード: uart.h

プロトタイプ:
```
void WriteUART1(unsigned int data);
void WriteUART2(unsigned int data);
```

引数: *data* 転送されるデータ

戻り値: なし

備考: この関数は、送信バッファへ転送されるデータを書き込みます。
9 ビット転送がイネーブルされている場合は、9 ビット値が送信バッファへ書き込まれます。
8 ビット転送がイネーブルされている場合は、上位バイトがマスクされて、送信バッファへ書き込まれます。

ソース・ファイル:
WriteUART1.c
WriteUART2.c

コード例:
```
WriteUART1(0xFF);
```

getsUART1
getsUART2

説明: この関数は、指定された長さの文字列データを読み出して、指定されたバッファ・ロケーションへ格納します。

インクルード: uart.h

プロトタイプ:
```
unsigned int getsUART1(unsigned int length,
  unsigned int *buffer, unsigned int
uart_data_wait);
unsigned int getsUART2(unsigned int length,
  unsigned int *buffer, unsigned int
uart_data_wait);
```

引数:

length	受信する文字列の長さ
buffer	受信したデータを格納するロケーションを指すポインタ
uart_data_wait	モジュールがリターンするまでに待つタイムアウト・カウント タイムアウト・カウントが 'N' の場合は、実際のタイムアウトは約 (19 * N - 1) 命令サイクルになります。

戻り値: この関数は、これから受信するバイト数を返します。
戻り値が '0' の場合、文字列全体が受信されたことを表示します。
戻り値が非ゼロの場合、文字列全体が受信されていないことを表示します。

備考: なし

ソース・ファイル:
getsUART1.c
getsUART2.c

コード例:
```
Datarem = getsUART1(6, Rxdata_loc, 40);
```

putsUART1
putsUART2

説明: この関数は、送信文字列データを UART 送信バッファへ書き込みます。

インクルード: uart.h

プロトタイプ:
```
void putsUART1(unsigned int *buffer);
void putsUART2(unsigned int *buffer);
```

引数:

buffer	送信文字列データを指すポインタ

戻り値: なし

備考: この関数は、NULL 文字に遭遇するまで送信データを送信バッファへ書き込みます。
送信バッファがフルになると、データが送信されるまで待った後に、次のデータを送信レジスタへ書き込みます。

ソース・ファイル:
putsUART1.c
putsUART2.c

コード例:
```
putsUART1(Txdata_loc);
```

getcUART1
getcUART2

説明： この関数は、ReadUART1 および ReadUART2 と同じです。

ソース・ファイル： `#define to ReadUART1 and ReadUART2 in uart.h`

putcUART1
putcUART2

説明： この関数は、WriteUART1 および WriteUART2 と同じです。

ソース・ファイル： `#define to WriteUART1 and WriteUART2 in uart.h`

3.12.2 個別マクロ

EnableIntU1RX
EnableIntU2RX

説明： このマクロは、UART 受信割り込みをイネーブルします。

インクルード： `uart.h`

引数： なし

備考： このマクロは、割り込みイネーブル・コントロール・レジスタの UART 受信割り込みイネーブル・ビットをセットします。

コード例：
```
EnableIntU2RX;
```

EnableIntU1TX
EnableIntU2TX

説明： このマクロは、UART 送信割り込みをイネーブルします。

インクルード： `uart.h`

引数： なし

備考： このマクロは、割り込みイネーブル・コントロール・レジスタの UART 送信割り込みイネーブル・ビットをセットします。

コード例：
```
EnableIntU2TX;
```

DisableIntU1RX
DisableIntU2RX

説明： このマクロは、UART 受信割り込みをディスエーブルします。

インクルード： `uart.h`

引数： なし

備考： このマクロは、割り込みイネーブル・コントロール・レジスタの UART 受信割り込みイネーブル・ビットをクリアします。

コード例：
```
DisableIntU1RX;
```

DisableIntU1TX
DisableIntU2TX

説明 : このマクロは、UART 送信割り込みをディスエーブルします。

インクルード : uart.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタの UART 送信割り込みイネーブル・ビットをクリアします。

コード例 :
```
DisableIntU1TX;
```

SetPriorityIntU1RX
SetPriorityIntU2RX

説明 : このマクロは、UART 受信割り込みの優先順位を設定します。

インクルード : uart.h

引数 : *priority*

備考 : このマクロは、割り込み優先順位コントロール・レジスタの UART 受信割り込み優先順位ビットを設定します。

コード例 :
```
SetPriorityIntU1RX(6);
```

SetPriorityIntU1TX
SetPriorityIntU2TX

説明 : このマクロは、UART 送信割り込みの優先順位を設定します。

インクルード : uart.h

引数 : *priority*

備考 : このマクロは、割り込み優先順位コントロール・レジスタの UART 送信割り込み優先順位ビットを設定します。

コード例 :
```
SetPriorityIntU1TX(5);
```

3.12.3 使用例

```
#define _ _dsPIC30F6014_ _
#include<p30fxxxx.h>
#include<uart.h>
/* Received data is stored in array Buf  */
char Buf[80];
char * Receivedddata = Buf;
/* This is UART1 transmit ISR */
void _ _attribute_ _((_ _interrupt_ _)) _U1TXInterrupt(void)
{
   IFS0bits.U1TXIF = 0;
}
/* This is UART1 receive ISR */
void _ _attribute_ _((_ _interrupt_ _)) _U1RXInterrupt(void)
{
    IFS0bits.U1RXIF = 0;
/* Read the receive buffer till atleast one or more character can be
read */
    while( DataRdyUART1())
    {
        ( *( Receiveddata)++) = ReadUART1();
    }
}
int main(void)
{
/* Data to be transmitted using UART communication module */
char Txdata[] = {'M','i','c','r','o','c','h','i','p','
                  ','I','C','D','2','¥0'};
/* Holds the value of baud register   */
unsigned int baudvalue;
/* Holds the value of uart config reg */
unsigned int U1MODEvalue;
/* Holds the information regarding uart
TX & RX interrupt modes */
unsigned int U1STAvalue;
/* Turn off UART1module */
    CloseUART1();
/* Configure uart1 receive and transmit interrupt */
    ConfigIntUART1(UART_RX_INT_EN & UART_RX_INT_PR6 &
                   UART_TX_INT_DIS & UART_TX_INT_PR2);
/* Configure UART1 module to transmit 8 bit data with one stopbit.Also
Enable loopback mode  */
    baudvalue = 5;
    U1MODEvalue = UART_EN & UART_IDLE_CON &
                  UART_DIS_WAKE & UART_EN_LOOPBACK  &
                  UART_EN_ABAUD & UART_NO_PAR_8BIT  &
                  UART_1STOPBIT;
    U1STAvalue  = UART_INT_TX_BUF_EMPTY  &
                  UART_TX_PIN_NORMAL &
                  UART_TX_ENABLE & UART_INT_RX_3_4_FUL &
                  UART_ADR_DETECT_DIS &
                  UART_RX_OVERRUN_CLEAR;
    OpenUART1(U1MODEvalue, U1STAvalue, baudvalue);
```

```
/* Load transmit buffer and transmit the same till null character is
encountered */
    putsUART1 ((unsigned int *)Txdata);
/* Wait for  transmission to complete */
    while(BusyUART1());
/* Read all the data remaining in receive buffer which are unread */
    while(DataRdyUART1())
    {
        (*( Receiveddata)++) = ReadUART1() ;
    }
/* Turn off UART1 module */
    CloseUART1();
    return 0;
}
```

3.13 DCI 関数

このセクションには、DCI モジュール用個別関数の一覧と、各関数の使用例を記載します。関数はマクロとして構成可能です。

3.13.1 個別関数

CloseDCI

説明:	この関数は、DCI モジュールをターンオフします。
インクルード:	dci.h
プロトタイプ:	void CloseDCI(void);
引数:	なし
戻り値:	なし
備考:	この関数は、DCI モジュールをターンオフした後に、DCI 割り込みをディスエーブルします。割り込みフラグ・ビットもクリアします。
ソース・ファイル:	CloseDCI.c
コード例:	CloseDCI();

BufferEmptyDCI

説明:	この関数は、DCI 送信バッファ・フル・ステータスを返します。
インクルード:	dci.h
プロトタイプ:	char BufferEmptyDCI(void);
引数:	なし
戻り値:	TMPTY 値が '1' の場合、'1' を返して、送信バッファがエンプティであることを表示します。 TMPTY 値が '0' の場合、'0' を返して、送信バッファがエンプティでないことを表示します。
備考:	この関数は、DCISTAT<TMPTY> ビットのステータスを返します。このビットは、送信バッファがエンプティであるか否かを表示します。
ソース・ファイル:	BufferEmptyDCI.c
コード例:	while(!BufferEmptyDCI());

ConfigIntDCI

説明 : この関数は、DCI 割り込みを設定します。

インクルード : dci.h

プロトタイプ : void ConfigIntDCI(unsigned int *config*);

引数 : *config* 次のように定義される DCI 割り込み優先順位とイネーブル / ディスエーブル情報 :

DCI 割り込みのイネーブル / ディスエーブル
DCI_INT_ON
DCI_INT_OFF

DCI 割り込みの優先順位
DCI_INT_PRI_0
DCI_INT_PRI_1
DCI_INT_PRI_2
DCI_INT_PRI_3
DCI_INT_PRI_4
DCI_INT_PRI_5
DCI_INT_PRI_6
DCI_INT_PRI_7

戻り値 : なし

備考 : この関数は、割り込みフラグ (DCIIF) ビットをクリアした後に、割り込み優先順位を設定して、割り込みをイネーブル / ディスエーブルします。

ソース・ファイル : ConfigIntDCI.c

コード例 :
```
ConfigIntDCI(DCI_INT_PRI_6 & DCI_INT_ENABLE);
```

DataRdyDCI

説明 : この関数は、DCI 受信バッファのステータスを返します。

インクルード : dci.h

プロトタイプ : char DataRdyDCI(void);

引数 : なし

戻り値 : RFUL 値が '1' の場合、'1' を返して、受信バッファからのデータ読み出しが可能なことを表示します。
RFUL 値が '0' の場合、'0' を返して、受信バッファがエンプティであることを表示します。

備考 : この関数は、DCISTAT<RFUL> ビットのステータスを返します。このビットは、受信バッファ内のデータの有無を表示します。

ソース・ファイル : DataRdyDCI.c

コード例 :
```
while(!DataRdyDCI());
```

OpenDCI

説明 : この関数は DCI を設定します。

インクルード : dci.h

プロトタイプ :
```
void OpenDCI(unsigned int config1,
             unsigned int config2,
             unsigned int config3,
             unsigned int trans_mask,
             unsigned int recv_mask)
```

OpenDCI (続き)

引数 :

config1

DCION1 レジスタに設定される次のように定義されたパラメータ :

モジュールの On/Off
DCI_EN
DCI_DIS

アイドル・モード動作
DCI_IDLE_CON
DCI_IDLE_STOP

DCI ループバック・モードのイネーブル
DCI_DIGI_LPBACK_EN
DCI_DIGI_LPBACK_DIS

CSCK ピン方向の選択
DCI_SCKD_INP
DCI_SCKD_OUP

DCI サンプリング・エッジの選択
DCI_SAMP_CLK_RIS
DCI_SAMP_CLK_FAL

FS ピン方向の選択
DCI_FSD_INP
DCI_FSD_OUP

アンダーフロー時に送信するデータ
DCI_TX_LASTVAL_UNF
DCI_TX_ZERO_UNF

送信ディスエーブル時の SDO ピン・ステータス
DCI_SDO_TRISTAT
DCI_SDO_ZERO

データ端揃え制御
DCI_DJST_ON
DCI_DJST_OFF

フレーム同期モードの選択
DCI_FSM_ACLINK_20BIT
DCI_FSM_ACLINK_16BIT
DCI_FSM_I2S
DCI_FSM_MULTI

config2

DCICON2 レジスタに設定される次のように定義されたパラメータ :

バッファ長
DCI_BUFF_LEN_4
DCI_BUFF_LEN_3
DCI_BUFF_LEN_2
DCI_BUFF_LEN_1

DCI フレーム同期ジェネレータの制御
DCI_FRAME_LEN_16
DCI_FRAME_LEN_15
DCI_FRAME_LEN_14
.....
DCI_FRAME_LEN_1

OpenDCI (続き)

DCI データ・ワードのサイズ
DCI_DATA_WORD_16
DCI_DATA_WORD_15
DCI_DATA_WORD_14
.....
DCI_DATA_WORD_5
DCI_DATA_WORD_4

config3 — DCICON3 レジスタに設定するビット・クロック・ジェネレータ値

trans_mask/ recv_mask — 送信 / 受信スロット
次のように定義された、TSCON/ RSCON レジスタに設定するイネーブル・ビット :

DCI_DIS_SLOT_15
DCI_DIS_SLOT_14
.....
DCI_DIS_SLOT_1
DCI_DIS_SLOT_0
DCI_EN_SLOT_ALL
DCI_DIS_SLOT_ALL

戻り値 : なし

備考 : このルーチンは次のパラメータを設定します :

1. DCICON1 レジスタ :
   イネーブル・ビット、
   フレーム同期モード、
   データ端揃え、
   サンプル・クロック方向、
   サンプル・クロック、
   エッジ制御、
   出力フレーム同期方向制御、
   連続送信 / 受信モード、
   アンダーフロー・モード。
2. DCICON2 レジスタ :
   フレーム同期ジェネレータ制御、
   データ・ワード・サイズ・ビット、
   バッファ長コントロール・ビット。
3. DCICON3 レジスタ : クロック・ジェネレータ・コントロール・ビット
4. TSCON レジスタ : 送信タイム・スロット・イネーブル・コントロール・ビット。
5. RSCON レジスタ : 受信タイム・スロット・イネーブル・コントロール・ビット。

ソース・ファイル : OpenDCI.c

OpenDCI (続き)

コード例 :

```
DCICON1value = DCI_EN &
               DCI_IDLE_CON &
               DCI_DIGI_LPBACK_EN &
               DCI_SCKD_OUP &
               DCI_SAMP_CLK_FAL &
               DCI_FSD_OUP &
               DCI_TX_LASTVAL_UNF &
               DCI_SDO_TRISTAT &
               DCI_DJST_OFF &
               DCI_FSM_ACLINK_16BIT ;
DCICON2value = DCI_BUFF_LEN_4 &
               DCI_FRAME_LEN_2&
               DCI_DATA_WORD_16 ;
DCICON3value = 0x02 ;
RSCONvalue   = DCI_EN_SLOT_ALL &
               DCI_DIS_SLOT_15 &
               DCI_DIS_SLOT_9 &
               DCI_DIS_SLOT_2;
TSCONvalue   = DCI_EN_SLOT_ALL &
               DCI_DIS_SLOT_14 &
               DCI_DIS_SLOT_8 &
               DCI_DIS_SLOT_1;
OpenDCI(DCICON1value, DCICON2value,  DCICON3value,
TSCONvalue, RSCONvalue);
```

ReadDCI

説明 : この関数は、DCI 受信バッファの内容を読み出します。

インクルード : dci.h

プロトタイプ :

```
unsigned int ReadDCI(unsigned char buffer);
```

引数 : *buffer* 読み出す DCI バッファの番号。

戻り値 : なし

備考 : この関数は、*buffer* で指定された DCI 受信バッファの内容を返します。

ソース・ファイル : ReadDCI.c

コード例 :

```
unsigned int DCI_buf0;
DCI_buf0 = ReadDCI(0);
```

WriteDCI

説明 : この関数は、DCI 送信バッファへ転送されるデータを書き込みます。

インクルード : dci.h

プロトタイプ :

```
void WriteDCI(unsigned int data_out,
              unsigned char buffer);
```

引数 : *data_out* 送信データ。

buffer 書き込まれる DCI バッファの番号。

戻り値 : なし

備考 : この関数は、*buffer* で指定された送信バッファへ *data_out* を書き込みます。

ソース・ファイル : WriteDCI.c

コード例 :

```
unsigned int DCI_tx0 = 0x60;
WriteDCI(DCI_tx0, 0);
```

3.13.2 個別マクロ

EnableIntDCI

説明 :	このマクロは、DCI 割り込みをイネーブルします。
インクルード :	dci.h
引数 :	なし
備考 :	このマクロは、割り込みイネーブル・コントロール・レジスタの DCI 割り込みイネーブル・ビットをセットします。
コード例 :	`EnableIntDCI;`

DisableIntDCI

説明 :	このマクロは、DCI 割り込みをディスエーブルします。
インクルード :	dci.h
引数 :	なし
備考 :	このマクロは、割り込みイネーブル・コントロール・レジスタの DCI 割り込みイネーブル・ビットをクリアします。
コード例 :	`DisableIntDCI;`

SetPriorityIntDCI

説明 :	このマクロは、DCI 割り込みの優先順位をセットします。
インクルード :	dci.h
引数 :	*priority*
備考 :	このマクロは、割り込み優先順位コントロール・レジスタの DCI 割り込み優先順位ビットをセットします。
コード例 :	`SetPriorityIntDCI(4);`

3.13.3 使用例

```
#define _ _dsPIC30F6014_ _
#include<p30fxxxx.h>
#include<dci.h>
/* Received data is stored from 0x1820 onwards. */
unsigned int *  Receiveddata = ( unsigned int *)0x1820;
void _ _attribute_ _((_ _interrupt_ _)) _DCIInterrupt(void)
{
    IFS2bits.DCIIF = 0;
}
int main(void)
{
/* Data to be transmitted using DCI module */
    unsigned int data16[] = {0xabcd, 0x1234, 0x1578,
                             0xfff0, 0xf679};
/* Holds configuration information   */
    unsigned int DCICON1value;
/* Holds the value of framelength, wordsize and buffer length */
    unsigned int DCICON2value;
/* Holds the information reagarding bit clock
   generator */
    unsigned int DCICON3value ;
/* Holds the information reagarding data to be received
   or ignored during this time slot */
   unsigned int RSCONvalue ;
/* Holds the information reagarding transmit buffer
   contents are sent during the timeslot */
   unsigned int TSCONvalue ;
   int i ;
    CloseDCI();
/* Configure DCI receive / transmit interrupt */
    ConfigIntDCI( DCI_INT_ON & DCI_INT_PRI_6);
/* Configure DCI module to transmit 16 bit data with multichannel mode
*/
    DCICON1value = DCI_EN &  DCI_IDLE_CON &
                   DCI_DIGI_LPBACK_EN &
                   DCI_SCKD_OUP  &
                   DCI_SAMP_CLK_FAL &
                   DCI_FSD_OUP &
                   DCI_TX_ZERO_UNF &
                   DCI_SDO_TRISTAT &
                   DCI_DJST_OFF &
                   DCI_FSM_MULTI;
    DCICON2value = DCI_BUFF_LEN_4 & DCI_FRAME_LEN_4 &
                     DCI_DATA_WORD_16 ;
    DCICON3value = 0x00;
    RSCONvalue   = DCI_EN_SLOT_ALL & DCI_DIS_SLOT_11 &
                   DCI_DIS_SLOT_4 & DCI_DIS_SLOT_5;
    TSCONvalue   = DCI_EN_SLOT_ALL & DCI_DIS_SLOT_11 &
                   DCI_DIS_SLOT_4 &DCI_DIS_SLOT_5;
    OpenDCI(DCICON1value, DCICON2value, DCICON3value,
         TSCONvalue, RSCONvalue);
```

```
/* Load transmit buffer and transmit the same */
    i = 0;
    while( i<= 3)
    {
        WriteDCI(data16[i],i);
        i++;
    }
/* Start generating serial clock by DCI module */
    DCICON3 = 0X02;
 /* Wait for  transmit buffer to get empty  */
    while(!BufferEmptyDCI());
/* Wait till new data is available in RX buffer */
    while(!DataRdyDCI());
/*  Read all the data remaining in receive buffer which
    are unread into user defined data buffer*/
    i = 0;
    while( i<=3)
    {
        (*( Receiveddata)++) = ReadDCI(i);
        i++;
    }
/* Turn off DCI module and clear IF bit  */
    CloseDCI();
    return 0;
}
```

3.14 SPI 関数

このセクションには、SPI モジュール用個別関数の一覧と、各関数の使用例を記載します。関数はマクロとして構成可能です。

3.14.1 個別関数

ConfigIntSPI1
ConfigIntSPI2

説明 : この関数は、SPI の割り込みを設定します。

インクルード : spi.h

プロトタイプ :
```
void ConfigIntSPI1(unsigned int config);
void ConfigIntSPI2(unsigned int config);
```

引数 : *config* 次のように定義される SPI 割り込み優先順位とイネーブル / ディスエーブル情報 :

割り込みイネーブル / ディスエーブル
SPI_INT_EN
SPI_INT_DIS

割り込み優先順位
SPI_INT_PRI_0
SPI_INT_PRI_1
SPI_INT_PRI_2
SPI_INT_PRI_3
SPI_INT_PRI_4
SPI_INT_PRI_5
SPI_INT_PRI_6
SPI_INT_PRI_7

戻り値 : なし

備考 : この関数は、割り込みフラグビットをクリアした後に、割り込み優先順位を設定して、割り込みをイネーブル / ディスエーブルします。

ソース・ファイル :
ConfigIntSPI1.c
ConfigIntSPI2.c

コード例 :
```
ConfigIntSPI1(SPI_INT_PRI_3 & SPI_INT_EN);
```

CloseSPI1
CloseSPI2

説明 : この関数は、SPI モジュールをターンオフします。

インクルード : spi.h

プロトタイプ :
```
void CloseSPI1(void);
void CloseSPI2(void);
```

引数 : なし

戻り値 : なし

備考 : この関数は、SPI 割り込みをディスエーブルした後に、モジュールをターンオフします。割り込みフラグ・ビットもクリアします。

ソース・ファイル :
CloseSPI1.c
CloseSPI2.c

コード例 :
```
CloseSPI1();
```

DataRdySPI1
DataRdySPI2

説明 : この関数は、SPI バッファ内の読み出しデータの有無を調べます。

インクルード : spi.h

プロトタイプ :
```
char DataRdySPI1(void);
char DataRdySPI2(void);
```

引数 : なし

戻り値 : '1' が返された場合、受信バッファにはデータが受信され、読み出し可能であることを表わします。
'0' が返された場合、読み出しが完了していないため受信バッファはエンプティであることを表わします。

備考 : この関数は、SPI 受信バッファのステータスを返します。
SPIxSTAT<SPIRBF> ビットが示すように、SPI 受信バッファに読み出すべき新しいデータが存在しているか否かを表示します。このビットはデータがバッファから読み出されたとき、ハードウェアによりクリアされます。

ソース・ファイル : DataRdySPI1.c
DataRdySPI2.c

コード例 :
```
while(DataRdySPI1());
```

ReadSPI1
ReadSPI2

説明 : この関数は、SPI 受信バッファ (SPIxBUF) レジスタ値を読み出します。

インクルード : spi.h

プロトタイプ :
```
unsigned int ReadSPI1(void);
unsigned int ReadSPI2(void);
```

引数 : なし

戻り値 : この関数は、受信バッファ (SPIxBUF) レジスタ値を返します。
'-1' が返された場合、SPI バッファには読み出すデータが存在していないことを表わします。

備考 : この関数は、受信バッファ・レジスタ値を返します。
16 ビット交信がイネーブルされている場合、SPIxRBF レジスタのデータが返されます。
8 ビット交信がイネーブルされている場合、SPIxRBF レジスタの下位バイトが返されます。
SPISTAT<RBF> ビットで表示されるように、SPIxBUF にデータが存在する場合にのみ、読み出されます。その他の場合は、値 '-1' を返します。

ソース・ファイル : ReadSPI1.c
ReadSPI2.c

コード例 :
```
unsigned int RX_data;
RX_data = ReadSPI1();
```

WriteSPI1
WriteSPI2

説明 : この関数は、送信データを送信バッファ (SPIxBUF) レジスタへ書き込みます。

インクルード : spi.h

プロトタイプ :
```
void WriteSPI1(unsigned int data);
void WriteSPI2(unsigned int data);
```

引数 : *data* SPI バッファへ格納する送信データ。

備考 : この関数は、送信バッファへ転送されるデータ (バイト / ワード) を書き込みます。
16 ビット転送がイネーブルされている場合は、16 ビット値が送信バッファへ書き込まれます。
8 ビット転送がイネーブルされている場合は、上位バイトがマスクされて、送信バッファへ書き込まれます。

戻り値 : なし

ソース・ファイル :
```
WriteSPI1.c
WriteSPI2.c
```

コード例 :
```
WriteSPI1(0x3FFF);
```

OpenSPI1
OpenSPI2

説明 : この関数は、SPI モジュールを設定します。

インクルード : spi.h

プロトタイプ :
```
void OpenSPI1(unsigned int config1,
              unsigned int config2);
void OpenSPI2(unsigned int config1,
              unsigned int config2);
```

引数 : *config1* SPIxCON レジスタに設定される次のように定義されたパラメータ :

フレーム化 SPI サポートのイネーブル / ディスエーブル
```
FRAME_ENABLE_ON
FRAME_ENABLE_OFF
```

フレーム同期パルス方向制御
```
FRAME_SYNC_INPUT
FRAME_SYNC_OUTPUT
```

SDO ピンのコントロール・ビット
```
DISABLE_SDO_PIN
ENABLE_SDO_PIN
```

ワード / バイト交信モード
```
SPI_MODE16_ON
SPI_MODE16_OFF
```

SPI データ入力のサンプル位相
```
SPI_SMP_ON
SPI_SMP_OFF
```

SPI クロック・エッジの選択
```
SPI_CKE_ON
SPI_CKE_OFF
```

OpenSPI1
OpenSPI2 (続き)

SPI スレーブ選択イネーブル
SLAVE_SELECT_ENABLE_ON
SLAVE_SELECT_ENABLE_OFF

SPI クロック極性の選択
CLK_POL_ACTIVE_LOW
CLK_POL_ACTIVE_HIGH

SPI モード選択ビット
MASTER_ENABLE_ON
MASTER_ENABLE_OFF

セカンダリ・プリスケールの選択
SEC_PRESCAL_1_1
SEC_PRESCAL_2_1
SEC_PRESCAL_3_1
SEC_PRESCAL_4_1
SEC_PRESCAL_5_1
SEC_PRESCAL_6_1
SEC_PRESCAL_7_1
SEC_PRESCAL_8_1

プライマリ・プリスケールの選択
PRI_PRESCAL_1_1
PRI_PRESCAL_4_1
PRI_PRESCAL_16_1
PRI_PRESCAL_64_1

config2 SPIxSTAT レジスタに設定される次のように定義されたパラメータ :

SPI イネーブル / ディスエーブル
SPI_ENABLE
SPI_DISABLE

SPI アイドル動作モード
SPI_IDLE_CON
SPI_IDLE_STOP

クリア受信オーバーフロー・フラグ・ビット
SPI_RX_OVFLOW_CLR

戻り値 : なし

備考 : この関数は SPI モジュールを初期化して、アイドル動作モードを設定します。

ソース・ファイル : OpenSPI1.c
OpenSPI2.c

OpenSPI1
OpenSPI2 (続き)

コード例 :

```
config1 = FRAME_ENABLE_OFF &
          FRAME_SYNC_OUTPUT &
          ENABLE_SDO_PIN &
          SPI_MODE16_ON &
          SPI_SMP_ON &
          SPI_CKE_OFF &
          SLAVE_SELECT_ENABLE_OFF &
          CLK_POL_ACTIVE_HIGH &
          MASTER_ENABLE_ON &
          SEC_PRESCAL_7_1 &
          PRI_PRESCAL_64_1;
config2 = SPI_ENABLE &
          SPI_IDLE_CON &
          SPI_RX_OVFLOW_CLR OpenSPI1(config1,
           config2);
```

putsSPI1
putsSPI2

説明 : この関数は、送信文字列データを SPI 送信バッファへ書き込みます。

インクルード : spi.h

プロトタイプ :

```
void putsSPI1(unsigned int length,
              unsigned int *wrptr);
void putsSPI2(unsigned int length,
              unsigned int *wrptr);
```

引数 :

length	送信データのワード / バイト数。
wrptr	送信データ文字列を指すポインタ。

戻り値 : なし

備考 : この関数は、指定された長さの送信データ・ワード / バイトを送信バッファへ書き込みます。
送信バッファがフルになると、データが送信されるまで待った後に、次のデータを送信レジスタへ書き込みます。
SPITBF ビットのセット中に SPI モジュールがディスエーブルされた場合は、制御はこの関数内に留まります。

ソース・ファイル : putsSPI1.c
putsSPI2.c

コード例 :

```
putsSPI1(10,Txdata_loc);
```

getsSPI1
getsSPI2

説明 : この関数は、指定された長さの文字列データを読み出して、指定されたロケーションへ格納します。

インクルード : spi.h

プロトタイプ :

```
unsigned int getsSPI1(
            unsigned int length,
            unsigned int *rdptr,
            unsigned int  spi_data_wait);
unsigned int getsSPI2(
            unsigned int length,
            unsigned int *rdptr,
            unsigned int  spi_data_wait);
```

引数 :

length	受信する文字列の長さ
rdptr	受信したデータを格納するロケーションを指すポインタ
spi_data_wait	モジュールがリターンするまでに待つタイムアウト・カウント タイムアウト・カウントが 'N' の場合は、実際のタイムアウトは約 (19 * N - 1) 命令サイクルになります。

戻り値 : この関数は、これから受信するバイト数を返します。
戻り値が '0' の場合、文字列全体が受信されたことを表示します。
戻り値が非ゼロの場合、文字列全体が受信されていないことを表示します。

備考 : なし

ソース・ファイル :
getsSPI1.c
getsSPI2.c

コード例 :

```
Datarem = getsSPI1(6, Rxdata_loc, 40);
```

getcSPI1
getcSPI2

説明 : この関数は、ReadSPI1 および ReadSPI2 と同じです。

ソース・ファイル : #define to ReadSPI1 and ReadSPI2 in spi.h

putcSPI1
putcSPI2

説明 : この関数は、WriteSPI1 および WriteSPI2 と同じです。

ソース・ファイル : #define to WriteSPI1 and WriteSPI2 in spi.h

3.14.2 個別マクロ

EnableIntSPI1
EnableIntSPI2

説明 : このマクロは、SPI 割り込みをイネーブルします。

インクルード : `spi.h`

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタの SPI 割り込みイネーブル・ビットをセットします。

コード例 :
```
EnableIntSPI1;
```

DisableIntSPI1
DisableIntSPI2

説明 : このマクロは、SPI 割り込みをディスエーブルします。

インクルード : `spi.h`

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタの SPI 割り込みイネーブル・ビットをクリアします。

コード例 :
```
DisableIntSPI2;
```

SetPriorityIntSPI1
SetPriorityIntSPI2

説明 : このマクロは、SPI 割り込みの優先順位をセットします。

インクルード : `spi.h`

引数 : *priority*

備考 : このマクロは、割り込み優先順位コントロール・レジスタの SPI 割り込み優先順位ビットをセットします。

コード例 :
```
SetPriorityIntSPI2(2);
```

3.14.3 使用例

```
#define _ _dsPIC30F6014_ _
#include<p30fxxxx.h>
#include<spi.h>
/* Data received at SPI2 */
unsigned int datard ;
void _ _attribute_ _((_ _interrupt_ _)) _SPI1Interrupt(void)
{
    IFS0bits.SPI1IF = 0;
}
void _ _attribute_ _((_ _interrupt_ _)) _SPI2Interrupt(void)
{
    IFS1bits.SPI2IF = 0;
    SPI1STATbits.SPIROV = 0;  /* Clear SPI1 receive overflow
                                 flag if set */
}
int main(void)
{
/* Holds the information about SPI configuartion */
   unsigned int SPICONValue;
/* Holds the information about SPI Enable/Disable */
   unsigned int SPISTATValue;
/*Timeout value during which timer1 is ON */
   int timeout;
/* Turn off SPI modules */
    CloseSPI1();
    CloseSPI2();
    TMR1 = 0 ;
    timeout = 0;
    TRISDbits.TRISD0 = 0;
/* Configure SPI2 interrupt */
    ConfigIntSPI2(SPI_INT_EN &  SPI_INT_PRI_6);
/* Configure SPI1 module to transmit 16 bit timer1 value
   in master mode  */
    SPICONValue  =  FRAME_ENABLE_OFF & FRAME_SYNC_OUTPUT &
                    ENABLE_SDO_PIN & SPI_MODE16_ON &
                    SPI_SMP_ON & SPI_CKE_OFF &
                    SLAVE_SELECT_ENABLE_OFF &
                    CLK_POL_ACTIVE_HIGH &
                    MASTER_ENABLE_ON &
                    SEC_PRESCAL_7_1 &
                    PRI_PRESCAL_64_1;
    SPISTATValue  = SPI_ENABLE & SPI_IDLE_CON &
                    SPI_RX_OVFLOW_CLR;
    OpenSPI1(SPICONValue,SPISTATValue );
```

```
/* Configure SPI2 module to receive 16 bit timer value in
   slave mode */
    SPICONValue  =  FRAME_ENABLE_OFF & FRAME_SYNC_OUTPUT &
                    ENABLE_SDO_PIN  & SPI_MODE16_ON &
                    SPI_SMP_OFF & SPI_CKE_OFF &
                    SLAVE_SELECT_ENABLE_OFF  &
                    CLK_POL_ACTIVE_HIGH &
                    MASTER_ENABLE_OFF &
                    SEC_PRESCAL_7_1 &
                    PRI_PRESCAL_64_1;
    SPISTATValue =  SPI_ENABLE & SPI_IDLE_CON &
                    PI_RX_OVFLOW_CLR;
    OpenSPI2(SPICONValue,SPISTATValue );
    T1CON = 0X8000;
    while(timeout< 100 )
    {
        timeout = timeout+2 ;
    }
    T1CON = 0;
    WriteSPI1(TMR1);
    while(SPI1STATbits.SPITBF);
    while(!DataRdySPI2());
    datard =  ReadSPI2();
    if(datard <= 600)
    {
        PORTDbits.RD0 = 1;
    }

/* Turn off SPI module and clear IF bit  */
    CloseSPI1();
    CloseSPI2();
    return 0;
}
```

3.15 QEI 関数

このセクションには、QEI モジュール用個別関数の一覧と、各関数の使用例を記載します。関数はマクロとして構成可能です。

3.15.1 個別関数

CloseQEI

説明 :	この関数は、QEI モジュールをターンオフします。
インクルード :	qei.h
プロトタイプ :	void closeQEI(void);
引数 :	なし
戻り値	なし
備考 :	この関数は、QEI モジュールをディスエーブルして、QEI 割り込みイネーブル・ビットとフラグ・ビットをクリアします。
ソース・ファイル :	CloseQEI.c
コード例 :	CloseQEI();

ConfigIntQEI

説明 :	この関数は、QEI 割り込みを設定します。
インクルード :	qei.h
プロトタイプ :	void ConfigIntQEI(unsigned int *config*);
引数 :	*config* 次のように定義される QEI 割り込み優先順位とイネーブル / ディスエーブル情報 : QEI 割り込みイネーブル / ディスエーブル QEI_INT_ENABLE QEI_INT_DISABLE QEI 割り込みの優先順位 QEI_INT_PRI_0 QEI_INT_PRI_1 QEI_INT_PRI_2 QEI_INT_PRI_3 QEI_INT_PRI_4 QEI_INT_PRI_5 QEI_INT_PRI_6 QEI_INT_PRI_7
戻り値	なし
備考 :	この関数は、割り込みフラグビットをクリアした後に、割り込み優先順位を設定して、割り込みをイネーブル / ディスエーブルします。
ソース・ファイル :	ConfigIntQEI.c
コード例 :	ConfigIntQEI(QEI_INT_ENABLE & QEI_INT_PRI_1);

OpenQEI

説明 : この関数は QEI を設定します。

インクルード : qei.h

プロトタイプ : void OpenQEI(unsigned int *config1*, unsigned int *config2*);

引数 :

config1 QEIxCON レジスタに設定される次のように定義されたパラメータ :

位置カウンタ方向選択コントロール・ビット
QEI_DIR_SEL_QEB
QEI_DIR_SEL_CNTRL

タイマ・クロック源選択ビット
QEI_EXT_CLK
QEI_INT_CLK

ポジション・カウンタ・リセット・イネーブル
QEI_INDEX_RESET_ENABLE
QEI_INDEX_RESET_DISABLE

タイマ入力クロック・プリスケーラ選択ビット
QEI_CLK_PRESCALE_1
QEI_CLK_PRESCALE_8
QEI_CLK_PRESCALE_64
QEI_CLK_PRESCALE_256

タイマ・ゲーティング時間積算イネーブル・ビット
QEI_GATED_ACC_ENABLE
QEI_GATED_ACC_DISABLE

ポジション・カウンタ方向状態出力イネーブル
QEI_LOGIC_CONTROL_IO
QEI_NORMAL_IO

フェーズ A とフェーズ B 入力ワップ選択ビット
QEI_INPUTS_SWAP
QEI_INPUTS_NOSWAP

QEI 動作モード選択
QEI_MODE_x4_MATCH
QEI_MODE_x4_PULSE
QEI_MODE_x2_MATCH
QEI_MODE_x2_PULSE
QEI_MODE_TIMER
QEI_MODE_OFF

ポジション・カウンタ方向ステータス
QEI_UP_COUNT
QEI_DOWN_COUNT

アイドル・モード動作
QEI_IDLE_STOP
QEI_IDLE_CON

config2 DFLTxCON レジスタに設定するパラメータ

4X 直交カウント・モードの場合 :
インデックス・パルスと比較するフェーズ A 入力信号の状態
MATCH_INDEX_PHASEA_HIGH
MATCH_INDEX_PHASEA_LOW

OpenQEI (続き)

インデックス・パルスと比較するフェーズ B 入力信号の状態
MATCH_INDEX_PHASEB_HIGH
MATCH_INDEX_PHASEB_LOW

2X 直交カウント・モードの場合 :
インデックス状態と比較するフェーズ入力信号
MATCH_INDEX_INPUT_PHASEA
MATCH_INDEX_INPUT_PHASEB

インデックス・パルスと比較するフェーズ入力信号状態
MATCH_INDEX_INPUT_HIGH
MATCH_INDEX_INPUT_LOW

ポジション・カウント・イベント割り込みのイネーブル / ディスエーブル
POS_CNT_ERR_INT_ENABLE
POS_CNT_ERR_INT_DISABLE

QEA/QEB デジタル・フィルタのクロック分周比選択ビット
QEI_QE_CLK_DIVIDE_1_1
QEI_QE_CLK_DIVIDE_1_2
QEI_QE_CLK_DIVIDE_1_4
QEI_QE_CLK_DIVIDE_1_16
QEI_QE_CLK_DIVIDE_1_32
QEI_QE_CLK_DIVIDE_1_64
QEI_QE_CLK_DIVIDE_1_128
QEI_QE_CLK_DIVIDE_1_256

QEA/QEB デジタル・フィルタ出力イネーブル
QEI_QE_OUT_ENABLE
QEI_QE_OUT_DISABLE

戻り値 なし

備考 : この関数は、QEI モジュールの QEICON レジスタと DFLTCON レジスタを設定します。
また、この関数は QEICON<CNTERR> ビットもクリアします。

ソース・ファイル : OpenQEI.c

コード例 :
```
OpenQEI(QEI_DIR_SEL_QEB & QEI_INT_CLK &
        QEI_INDEX_RESET_ENABLE &
        QEI_CLK_PRESCALE_1 & QEI_NORMAL_IO &
        QEI_MODE_TIMER & QEI_UP_COUNT,0);
```

ReadQEI

説明 : この関数は、POSCNT レジスタからポジション・カウント値を読み出します。

インクルード : qei.h

プロトタイプ : `unsigned int ReadQEI(void);`

引数 : なし

備考 : なし

戻り値 この関数は、POSCNT レジスタ値を返します。

ソース・ファイル : ReadQEI.c

コード例 :
```
unsigned int pos_count;
pos_count = ReadQEI();
```

WriteQEI

説明:	この関数は、QEI の最大カウント値を設定します。
インクルード:	qei.h
プロトタイプ:	void WriteQEI(unsigned int *position*);
引数:	*position* MAXCNT レジスタへ格納する値
戻り値	なし
備考:	なし
ソース・ファイル:	WriteQEI.c
コード例:	unsigned int *position* = 0x3FFF; WriteQEI(*position*);

3.15.2 個別マクロ

EnableIntQEI

説明:	このマクロは、QEI 割り込みをイネーブルします。
インクルード:	qei.h
引数:	なし
備考:	このマクロは、割り込みイネーブル・コントロール・レジスタの QEI 割り込みイネーブル・ビットをセットします。
コード例:	EnableIntQEI;

DisableIntQEI

説明:	このマクロは、QEI 割り込みをディスエーブルします。
インクルード:	qei.h
引数:	なし
備考:	このマクロは、割り込みイネーブル・コントロール・レジスタの QEI 割り込みイネーブル・ビットをクリアします。
コード例:	DisableIntQEI;

SetPriorityIntQEI

説明:	このマクロは、QEI 割り込みの優先順位をセットします。
インクルード:	qei.h
引数:	*priority*
備考:	このマクロは、割り込み優先順位コントロール・レジスタの QEI 割り込み優先順位ビットをセットします。
コード例:	SetPriorityIntQEI(7);

3.15.3 使用例

```
#define _ _dsPIC30F6010_ _
#include <p30fxxxx.h>
#include<qei.h>
unsigned int pos_value;

void _ _attribute_ _((_ _interrupt_ _)) _QEIInterrupt(void)
{
    PORTDbits.RD1 = 1;    /* turn off LED on RD1 */
    POSCNT = 0;
    IFS2bits.QEIIF = 0;    /* Clear QEI interrupt flag */
}
int main(void)
{
    unsigned int max_value;
    TRISDbits.TRISD1 = 0;
    PORTDbits.RD1 = 1;    /* turn off LED on RD1 */

/* Enable QEI Interrupt and Priority to "1" */
    ConfigIntQEI(QEI_INT_PRI_1 & QEI_INT_ENABLE);

    POSCNT  = 0;
    MAXCNT = 0xFFFF;
    OpenQEI(QEI_INT_CLK & QEI_INDEX_RESET_ENABLE &
            QEI_CLK_PRESCALE_256 &
            QEI_GATED_ACC_DISABLE & QEI_INPUTS_NOSWAP &
            QEI_MODE_TIMER & QEI_DIR_SEL_CNTRL &
            QEI_IDLE_CON, 0);
    QEICONbits.UPDN = 1;
    while(1)
    {
        pos_value = ReadQEI();
        if(pos_value >= 0x7FFF)
        {
            PORTDbits.RD1 = 0; /* turn on LED on RD1 */
        }
    }
    CloseQEI();
}
```

3.16 PWM 関数

このセクションには、PWM モジュール用個別関数の一覧と、各関数の使用例を記載します。関数はマクロとして構成可能です。

3.16.1 個別関数

CloseMCPWM

説明 : この関数は、モータ・コントロール PWM モジュールをターンオフします。

インクルード : `pwm.h`

プロトタイプ :
```
void closeMCPWM(void);
```

引数 : なし

戻り値 なし

備考 : この関数は、モータ・コントロール PWM モジュールをディスエーブルし、PWM、フォールト A、フォールト B の各割り込みイネーブルと各フラグ・ビットをクリアします。
また、この関数は PTCON、PWMCON1、PWMCON2 の各レジスタもクリアします。

ソース・ファイル : `CloseMCPWM.c`

コード例 :
```
CloseMCPWM();
```

ConfigIntMCPWM

説明 : この関数は、PWM 割り込みを設定します。

インクルード : `pwm.h`

プロトタイプ :
```
void ConfigIntMCPWM(unsigned int config);
```

引数 : *config* 次のように定義される PWM 割り込み優先順位とイネーブル / ディスエーブル情報 :

PWM 割り込みイネーブル / ディスエーブル
```
PWM_INT_EN
PWM_INT_DIS
```

PWM 割り込み優先順位
```
PWM_INT_PR0
PWM_INT_PR1
PWM_INT_PR2
PWM_INT_PR3
PWM_INT_PR4
PWM_INT_PR5
PWM_INT_PR6
PWM_INT_PR7
```

フォールト A 割り込みイネーブル / ディスエーブル
```
PWM_FLTA_EN_INT
PWM_FLTA_DIS_INT
```

フォールト A 割り込み優先順位
```
PWM_FLTA_INT_PR0
PWM_FLTA_INT_PR1
PWM_FLTA_INT_PR2
PWM_FLTA_INT_PR3
PWM_FLTA_INT_PR4
PWM_FLTA_INT_PR5
PWM_FLTA_INT_PR6
PWM_FLTA_INT_PR7
```

ConfigIntMCPWM (続き)

フォールト B 割り込みイネーブル / ディスエーブル
PWM_FLTB_EN_INT
PWM_FLTB_DIS_INT

フォールト B 割り込み優先順位
PWM_FLTB_INT_PR0
PWM_FLTB_INT_PR1
PWM_FLTB_INT_PR2
PWM_FLTB_INT_PR3
PWM_FLTB_INT_PR4
PWM_FLTB_INT_PR5
PWM_FLTB_INT_PR6
PWM_FLTB_INT_PR7

戻り値 なし

備考 : この関数は、割り込みフラグビットをクリアした後に、割り込み優先順位を設定して、割り込みをイネーブル / ディスエーブルします。

ソース・ファイル : ConfigIntMCPWM.c

コード例 :
```
ConfigIntMCPWM(PWM_INT_EN & PWM_INT_PR5 &
               PWM_FLTA_EN_INT &
               PWM_FLTA_INT_PR6 &
               PWM_FLTB_EN_INT &
               PWM_FLTB_INT_PR7);
```

OpenMCPWM

説明 : この関数は、モータ・コントロール PWM モジュールを設定します。

インクルード : pwm.h

プロトタイプ :
```
void OpenMCPWM(unsigned int period,
               unsigned int sptime,
               unsigned int config1,
               unsigned int config2,
               unsigned int config3);
```

引数 :

period PTPER レジスタへ格納する PWM タイムベース周期値

sptime SEVTCMP レジスタへ格納するスペシャル・イベント・コンペア値

config1 PTCON レジスタに設定される次のように定義されたパラメータ :

PWM モジュール・イネーブル / ディスエーブル
PWM_EN
PWM_DIS

アイドル・モード・イネーブル / ディスエーブル
PWM_IDLE_STOP
PWM_IDLE_CON

出力ポストスケーラ選択
PWM_OP_SCALE1
PWM_OP_SCALE2
.....
PWM_OP_SCALE15
PWM_OP_SCALE16

OpenMCPWM (続き)

入力プリスケーラ選択
PWM_IPCLK_SCALE1
PWM_IPCLK_SCALE4
PWM_IPCLK_SCALE16
PWM_IPCLK_SCALE64

PWM 動作モード
PWM_MOD_FREE
PWM_MOD_SING
PWM_MOD_UPDN
PWM_MOD_DBL

config2 PWMCON1 レジスタに設定される次のように定義されたパラメータ :

PWM I/O ピン対
PWM_MOD4_COMP
PWM_MOD3_COMP
PWM_MOD2_COMP
PWM_MOD1_COMP
PWM_MOD4_IND
PWM_MOD3_IND
PWM_MOD2_IND
PWM_MOD1_IND

PWM H/L I/O のイネーブル / ディスエーブル選択
PWM_PEN4H
PWM_PDIS4H
PWM_PEN3H
PWM_PDIS3H
PWM_PEN2H
PWM_PDIS2H
PWM_PEN1H
PWM_PDIS1H
PWM_PEN4L
PWM_PDIS4L
PWM_PEN3L
PWM_PDIS3L
PWM_PEN2L
PWM_PDIS2L
PWM_PEN1L
PWM_PDIS1L
PWM4 に関するビット定義は、所定のデバイスでのみ使用可能です。該当するデータシートを参照してください。

config3 PWMCON2 レジスタに設定される次のように定義されたパラメータ :

スペシャル・イベント・ポストスケーラ
PWM_SEVOPS1
PWM_SEVOPS2
.....
PWM_SEVOPS15
PWM_SEVOPS16

出力オーバーライド同期選択
PWM_OSYNC_PWM
PWM_OSYNC_Tcy

PWM 更新イネーブル / ディスエーブル
PWM_UDIS
PWM_UEN

OpenMCPWM (続き)

戻り値 なし

備考 : この関数は、PTPER、SEVTCMP、PTCON、PWMCON1、PWMCON2 の各レジスタを設定します。

ソース・ファイル : OpenMCPWM.c

コード例 :

```
period = 0x7fff;
sptime = 0x0;
config1 = PWM_EN & PWM_PTSIDL_DIS &
          PWM_OP_SCALE16 &
          PWM_IPCLK_SCALE16 &
          PWM_MOD_UPDN;
config2 = PWM_MOD1_COMP & PWM_PDIS4H &
          PWM_PDIS3H & PWM_PDIS2H &
          PWM_PEN1H &  PWM_PDIS4L &
          PWM_PDIS3L & PWM_PDIS2L &
          PWM_PEN1L;
config3 = PWM_SEVOPS1 & PWM_OSYNC_PWM &
          PWM_UEN;
OpenMCPWM(period, sptime, config1,
          config2, config3);
```

OverrideMCPWM

説明 : この関数は、OVDCON レジスタを設定します。

インクルード : pwm.h

プロトタイプ : void OverrideMCPWM(unsigned int *config*);

引数 :

config OVDCON レジスタに設定される次のように定義されたパラメータ :

PWM ジェネレータから制御される出力

PWM_GEN_4H
PWM_GEN_3H
PWM_GEN_2H
PWM_GEN_1H
PWM_GEN_4L
PWM_GEN_3L
PWM_GEN_2L
PWM_GEN_1L

PWM4 に関するビット定義は、所定のデバイスでのみ使用可能です。該当するデータシートを参照してください。

POUT ビットから制御される出力

PWM_POUT_4H
PWM_POUT_4L
PWM_POUT_3H
PWM_POUT_3L
PWM_POUT_2H
PWM_POUT_2L
PWM_POUT_1H
PWM_POUT_1L

PWM4 に関するビット定義は、所定のデバイスでのみ使用可能です。該当するデータシートを参照してください。

OverrideMCPWM (続き)

PWM マニュアル出力ビット
PWM_POUT4H_ACT
PWM_POUT4H_INACT
PWM_POUT4L_ACT
PWM_POUT4L_INACT
PWM_POUT3H_ACT
PWM_POUT3H_INACT
PWM_POUT3L_ACT
PWM_POUT3L_INACT
PWM_POUT2H_ACT
PWM_POUT2H_INACT
PWM_POUT2L_ACT
PWM_POUT2L_INACT
PWM_POUT1H_ACT
PWM_POUT1H_INACT
PWM_POUT1L_ACT
PWM_POUT1L_INACT
PWM4 に関するビット定義は、所定のデバイスでのみ使用可能です。該当するデータシートを参照してください。

戻り値 なし

備考 : この関数は、OVDCON レジスタの PWM 出力オーバーライド・ビットとマニュアル・コントロール・ビットを設定します。

ソース・ファイル : OverrideMCPWM.c

コード例 :
```
config = PWM_GEN_1L &
         PWM_GEN_1H &
         PWM_POUT1L_INACT &
         PWM_POUT3L_INACT;
OverrideMCPWM(config);
```

SetDCMCPWM

説明 : この関数はデューティ・サイクル・レジスタを設定し、PWMCON2 レジスタの 'PWM 更新ディスエーブル' ビットを更新します。

インクルード : pwm.h

プロトタイプ :
```
void SetDCMCPWM(
            unsigned int dutycyclereg,
            unsigned int dutycycle,
            char updatedisable);
```

引数 :

dutycyclereg	デューティ・サイクル・レジスタを指すポインタ
dutycycle	デューティ・サイクル・レジスタへ格納する値
updatedisable	PWMCON2 レジスタの '更新ディスエーブル' ビットへ格納する値

戻り値 なし

備考 : なし

ソース・ファイル : SetDCMCPWM.c

コード例 :
```
dutycyclereg = 1;
dutycycle = 0xFFF;
updatedisable = 0;
SetDCMCPWM(dutycyclereg, dutycycle,updatedisable);
```

SetMCPWMDeadTimeAssignment

説明 : この関数は、PWM 出力対に対するデッドタイム・ユニットの割り当てを設定します。

インクルード : pwm.h

プロトタイプ : void SetMCPWMDeadTimeAssignment (unsigned int *config*);

引数 : *config* DTCON2 レジスタに設定される次のように定義されたパラメータ :

PWM4 信号に対するデッドタイム選択ビット
PWM_DTS4A_UA
PWM_DTS4A_UB
PWM_DTS4I_UA
PWM_DTS4I_UB
PWM4 に関するビット定義は、所定のデバイスでのみ使用可能です。該当するデータシートを参照してください。

PWM3 信号に対するデッドタイム選択ビット
PWM_DTS3A_UA
PWM_DTS3A_UB
PWM_DTS3I_UA
PWM_DTS3I_UB

PWM2 信号に対するデッドタイム選択ビット
PWM_DTS2A_UA
PWM_DTS2A_UB
PWM_DTS2I_UA
PWM_DTS2I_UB

PWM1 信号に対するデッドタイム選択ビット
PWM_DTS1A_UA
PWM_DTS1A_UB
PWM_DTS1I_UA
PWM_DTS1I_UB

戻り値 なし

備考 : なし

ソース・ファイル : SetMCPWMDeadTimeAssignment.c

コード例 :
```
SetMCPWMDeadTimeAssignment(PWM_DTS3A_UA &
PWM_DTS2I_UA & PWM_DTS1I_UA);
```

SetMCPWMDeadTimeGeneration

説明: この関数は、デッドタイム値とクロック・プリスケーラを設定します。

インクルード: pwm.h

プロトタイプ:
```
void SetMCPWMDeadTimeGeneration(
          unsigned int config);
```

引数: *config* DTCON1 レジスタに設定される次のように定義されたパラメータ:

デッドタイム・ユニット B プリスケール選択ビット
PWM_DTBPS8
PWM_DTBPS4
PWM_DTBPS2
PWM_DTBPS1

デッドタイム・ユニット A プリスケール選択定数
PWM_DTA0
PWM_DTA1
PWM_DTA2
.....
PWM_DTA62
PWM_DTA63

デッドタイム・ユニット B プリスケール選択定数
PWM_DTB0
PWM_DTB1
PWM_DTB2
.....
PWM_DTB62
PWM_DTB63

デッドタイム・ユニット A プリスケール選択ビット
PWM_DTAPS8
PWM_DTAPS4
PWM_DTAPS2
PWM_DTAPS1

戻り値 なし

備考: なし

ソース・ファイル: SetMCPWMDeadTimeGeneration.c

コード例:
```
SetMCPWMDeadTimeGeneration(PWM_DTBPS16 &
PWM_DT54 & PWM_DTAPS8);
```

SetMCPWMFaultA

説明 : この関数は、PWM のフォールト A オーバーライド・ビット、フォールト A モード・ビット、フォールト入力 A イネーブル・ビットを設定します。

インクルード : pwm.h

プロトタイプ : void SetMCPWMFaultA(unsigned int *config*);

引数 : *config* FLTACON レジスタに設定される次のように定義されたパラメータ :

フォールト入力 A PWM オーバーライド値ビット
PWM_OVA4H_ACTIVE
PWM_OVA3H_ACTIVE
PWM_OVA2H_ACTIVE
PWM_OVA1H_ACTIVE
PWM_OVA4L_ACTIVE
PWM_OVA3L_ACTIVE
PWM_OVA2L_ACTIVE
PWM_OVA1L_ACTIVE
PWM_OVA4H_INACTIVE
PWM_OVA3H_INACTIVE
PWM_OVA2H_INACTIVE
PWM_OVA1H_INACTIVE
PWM_OVA4L_INACTIVE
PWM_OVA3L_INACTIVE
PWM_OVA2L_INACTIVE
PWM_OVA1L_INACTIVE
PWM4 に関するビット定義は、所定のデバイスでのみ使用可能です。該当するデータシートを参照してください。

フォールト A モード・ビット
PWM_FLTA_MODE_CYCLE
PWM_FLTA_MODE_LATCH

フォールト入力 A イネーブル・ビット .
PWM_FLTA4_EN
PWM_FLTA4_DIS
PWM_FLTA3_EN
PWM_FLTA3_DIS
PWM_FLTA2_EN
PWM_FLTA2_DIS
PWM_FLTA1_EN
PWM_FLTA1_DIS
PWM4 に関するビット定義は、所定のデバイスでのみ使用可能です。該当するデータシートを参照してください。

戻り値 なし

備考 : なし

ソース・ファイル : SetMCPWMFaultA.c

コード例 :
```
SetMCPWMFaultA(PWM_OVA3L_INACTIVE &
               PWM_FLTA_MODE_LATCH &
               PWM_FLTA1_DIS);
```

SetMCPWMFaultB

説明 : この関数は、PWM のフォールト B オーバーライド・ビット、フォールト B モード・ビット、フォールト入力 B イネーブル・ビットを設定します。

インクルード : pwm.h

プロトタイプ : `void SetMCPWMFaultB(unsigned int config);`

引数 : *config* FLTBCON レジスタに設定される次のように定義されたパラメータ :
FLTBCON レジスタは、所定のデバイスでのみ使用可能です。該当するデータシートを参照してください。

フォールト入力 B PWM オーバーライド値ビット
PWM_OVB4H_ACTIVE
PWM_OVB3H_ACTIVE
PWM_OVB2H_ACTIVE
PWM_OVB1H_ACTIVE
PWM_OVB4L_ACTIVE
PWM_OVB3L_ACTIVE
PWM_OVB2L_ACTIVE
PWM_OVB1L_ACTIVE
PWM_OVB4H_INACTIVE
PWM_OVB3H_INACTIVE
PWM_OVB2H_INACTIVE
PWM_OVB1H_INACTIVE
PWM_OVB4L_INACTIVE
PWM_OVB3L_INACTIVE
PWM_OVB2L_INACTIVE
PWM_OVB1L_INACTIVE

フォールト B モード・ビット
PWM_FLTB_MODE_CYCLE
PWM_FLTB_MODE_LATCH

フォールト入力 B イネーブル・ビット .
PWM_FLTB4_EN
PWM_FLTB4_DIS
PWM_FLTB3_EN
PWM_FLTB3_DIS
PWM_FLTB2_EN
PWM_FLTB2_DIS
PWM_FLTB1_EN
PWM_FLTB1_DIS

戻り値 なし

備考 : なし

ソース・ファイル : SetMCPWMFaultB.c

コード例 :
```
SetMCPWMFaultB(PWM_OVB3L_INACTIVE &
               PWM_FLTB_MODE_LATCH &
               PWM_FLTB2_DIS);
```

3.16.2 個別マクロ

EnableIntMCPWM

説明 : このマクロは、PWM 割り込みをイネーブルします。

インクルード : pwm.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタの PWM 割り込みイネーブル・ビットをセットします。

コード例 :
```
EnableIntMCPWM;
```

DisableIntMCPWM

説明 : このマクロは、PWM 割り込みをディスエーブルします。

インクルード : pwm.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタの PWM 割り込みイネーブル・ビットをクリアします。

コード例 :
```
DisableIntMCPWM;
```

SetPriorityIntMCPWM

説明 : このマクロは、PWM 割り込みの優先順位をセットします。

インクルード : pwm.h

引数 : *priority*

備考 : このマクロは、割り込み優先順位コントロール・レジスタの PWM 割り込み優先順位ビットをセットします。

コード例 :
```
SetPriorityIntMCPWM(7);
```

EnableIntFLTA

説明 : このマクロは、FLTA 割り込みをイネーブルします。

インクルード : pwm.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタの FLTA 割り込みイネーブル・ビットをセットします。

コード例 :
```
EnableIntFLTA;
```

DisableIntFLTA

説明 : このマクロは、FLTA 割り込みをディスエーブルします。

インクルード : pwm.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタの FLTA 割り込みイネーブル・ビットをクリアします。

コード例 :
```
DisableIntFLTA;
```

SetPriorityIntFLTA

説明 : このマクロは、FLTA 割り込みの優先順位をセットします。

インクルード : pwm.h

引数 : *priority*

備考 : このマクロは、割り込み優先順位コントロール・レジスタの FLTA 割り込み優先順位ビットをセットします。

コード例 :
```
SetPriorityIntFLTA(7);
```

EnableIntFLTB

説明 : このマクロは、FLTB 割り込みをイネーブルします。

インクルード : pwm.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタの FLTB 割り込みイネーブル・ビットをセットします。

コード例 :
```
EnableIntFLTB;
```

DisableIntFLTB

説明 : このマクロは、FLTB 割り込みをディスエーブルします。

インクルード : pwm.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタの FLTB 割り込みイネーブル・ビットをクリアします。

コード例 :
```
DisableIntFLTB;
```

SetPriorityIntFLTB

説明 : このマクロは、FLTB 割り込みの優先順位をセットします。

インクルード : pwm.h

引数 : *priority*

備考 : このマクロは、割り込み優先順位コントロール・レジスタの FLTB 割り込み優先順位ビットをセットします。

コード例 :
```
SetPriorityIntFLTB(1);
```

3.16.3 使用例

```
#define _ _dsPIC30F6010_ _
#include <p30fxxxx.h>
#include<pwm.h>
void _ _attribute_ _((_ _interrupt_ _)) _PWMInterrupt(void)
{
   IFS2bits.PWMIF = 0;
}
int main()
{
/* Holds the  PWM interrupt configuration value*/
    unsigned int config;
/* Holds the value to be loaded into dutycycle register */
    unsigned int period;
/* Holds the value to be loaded into special event compare register */
    unsigned int sptime;
/* Holds PWM configuration value  */
    unsigned int config1;
/* Holds the value be loaded into PWMCON1 register */
    unsigned int config2;
/* Holds the value to configure the special event trigger
   postscale and dutycycle */
    unsigned int config3;
/* The value of 'dutycyclereg' determines the duty cycle
   register(PDCx) to be written */
    unsigned int dutycyclereg;
    unsigned int dutycycle;
    unsigned char updatedisable;

/* Configure pwm interrupt enable/disable and set interrupt
   priorties */
    config = (PWM_INT_EN & PWM_FLTA_DIS_INT & PWM_INT_PR1
             & PWM_FLTA_INT_PR0
             & PWM_FLTB_DIS_INT & PWM_FLTB_INT_PR0);
    ConfigIntMCPWM( config );
/* Configure PWM to generate square wave of 50% duty  cycle */
    dutycyclereg  = 1;
    dutycycle     = 0x3FFF;
    updatedisable = 0;

    SetDCMCPWM(dutycyclereg,dutycycle,updatedisable);
    period = 0x7fff;
    sptime = 0x0;
    config1 = (PWM_EN & PWM_PTSIDL_DIS & PWM_OP_SCALE16
               & PWM_IPCLK_SCALE16 &
                 PWM_MOD_UPDN);
    config2 = (PWM_MOD1_COMP & PWM_PDIS4H & PWM_PDIS3H &
               PWM_PDIS2H & PWM_PEN1H & PWM_PDIS4L &
               PWM_PDIS3L & PWM_PDIS2L & PWM_PEN1L);
    config3  = (PWM_SEVOPS1 & PWM_OSYNC_PWM &  PWM_UEN);
    OpenMCPWM(period,sptime,config1,config2,config3);
    while(1);
}
```

3.17 I²C™ 関数

このセクションには、I²C モジュール用個別関数の一覧と、各関数の使用例を記載します。関数はマクロとして構成可能です。

3.17.1 個別関数

CloseI2C

説明 : この関数は、I²C モジュールをターンオフします。

インクルード : `i2c.h`

プロトタイプ :
```
void CloseI2C(void);
```

引数 : なし

戻り値 なし

備考 : この関数は、I²C モジュールをディスエーブルして、マスタ割り込みイネーブル・ビット、スレーブ割り込みイネーブル・ビット、各フラグ・ビットをクリアします。

ソース・ファイル : `CloseI2C.c`

コード例 :
```
CloseI2C();
```

ConfigIntI2C

説明 : この関数は、I²C の割り込みを設定します。

インクルード : `i2c.h`

プロトタイプ :
```
void ConfigIntI2C(unsigned int config);
```

引数 : *config* 次のように定義される I²C 割り込み優先順位とイネーブル / ディスエーブル情報 :

I²C マスタ割り込みイネーブル / ディスエーブル
```
MI2C_INT_ON
MI2C_INT_OFF
```
I²C スレーブ割り込みイネーブル / ディスエーブル
```
SI2C_INT_ON
SI2C_INT_OFF
```
I²C マスタ割り込み優先順位
```
MI2C_INT_PRI_7
MI2C_INT_PRI_6
MI2C_INT_PRI_5
MI2C_INT_PRI_4
MI2C_INT_PRI_3
MI2C_INT_PRI_2
MI2C_INT_PRI_1
MI2C_INT_PRI_0
```
I²C スレーブ割り込み優先順位
```
SI2C_INT_PRI_7
SI2C_INT_PRI_6
SI2C_INT_PRI_5
SI2C_INT_PRI_4
SI2C_INT_PRI_3
SI2C_INT_PRI_2
SI2C_INT_PRI_1
SI2C_INT_PRI_0
```

戻り値 なし

ConfigIntI2C (続き)

備考 :	この関数は、割り込みフラグ・ビットをクリアし、マスタとスレーブの割り込み優先順位を設定し、割り込みをイネーブル / ディスエーブルします。
ソース・ファイル :	ConfigIntI2C.c
コード例 :	`ConfigIntI2C(MI2C_INT_ON & MI2C_INT_PRI_3 & SI2C_INT_ON & SI2C_INT_PRI_5);`

AckI2C

説明 :	I^2C バス・アクノレッジ状態を発生します。
インクルード :	i2c.h
プロトタイプ :	`void AckI2C(void);`
引数 :	なし
戻り値	なし
備考 :	この関数は、I^2C バス・アクノレッジ状態を発生します。
ソース・ファイル :	AckI2C.c
コード例 :	`AckI2C();`

DataRdyI2C

説明 :	この関数は、I2CRCV レジスタにデータが存在するとき、ステータスをユーザーに通知します。
インクルード :	i2c.h
プロトタイプ :	`unsigned char DataRdyI2C(void);`
引数 :	なし
戻り値	この関数は、I2CRCV レジスタにデータがある場合には '1' を返します。その他の場合には '0' を返して、I2CRCV レジスタにデータがないことを表示します。
備考 :	この関数は、I2CRCV レジスタから読み出すバイトの有無を調べます。
ソース・ファイル :	DataRdyI2C.c
コード例 :	`if(DataRdyI2C());`

IdleI2C

説明 :	この関数は、I^2C バスがアイドルになるまでの待ち状態を発生します。
インクルード :	i2c.h
プロトタイプ :	`void IdleI2C(void);`
引数 :	なし
戻り値	なし

IdleI2C (続き)

備考 : この関数は、I²C コントロール・レジスタのスタート状態イネーブル・ビット、ストップ状態イネーブル・ビット、受信イネーブル・ビット、アクノレッジ・シーケンス・イネーブル・ビット、および I²C ステータス・レジスタの送信ステータス・ビットがクリアされるまで、待ち状態を維持します。ハードウェア I²C ペリフェラルではバス・シーケンスのスプーリングができないために、この IdleI2C 関数が必要になります。I²C ペリフェラルは、I²C 動作を開始する前にアイドル状態にあることが必要です。そうしないと、書き込み衝突が発生します。

ソース・ファイル : IdleI2C.c

コード例 :
```
IdleI2C();
```

MastergetsI2C

説明 : この関数は、I²C バスから既定のデータ文字列長を読み出します。

インクルード : i2c.h

プロトタイプ :
```
unsigned int MastergetsI2C(unsigned int length,
unsigned char *rdptr, unsigned int i2c_data_wait);
```

引数 :

length	I²C デバイスから読み出すバイト数
rdptr	I²C デバイスから読み出したデータを格納する RAM を指す文字型ポインタ
i2c_data_wait	モジュールがリターンするまでに待つタイムアウト・カウント タイムアウト・カウントが 'N' の場合は、実際のタイムアウトは約 (20 * N - 1) 命令サイクルになります。

戻り値 すべてのバイトを送信した場合、または I²C バスからバイトを読み出したが、指定された *i2c_data_wait* タイムアウト値内にデータを読み出すことができなかった場合に、この関数は '0' を返します。

備考 : このルーチンは、I²C バスから既定のデータ文字列を読み出します。

ソース・ファイル : MastergetsI2C.c

コード例 :
```
unsigned char string[10];
unsigned char *rdptr;
unsigned int length, i2c_data_wait;
length = 9;
rdptr = string;
i2c_data_wait = 152;
MastergetsI2C(length, rdptr, i2c_data_wait);
```

MasterputsI2C

説明 : この関数は、データ文字列を I²C バスに書き込む際に使用します。

インクルード : i2c.h

プロトタイプ :
```
unsigned int MasterputsI2C(unsigned char *wrptr);
```

引数 :

wrptr	RAM 内のデータ・オブジェクトを指す文字型ポインタ。このデータ・オブジェクトが I²C デバイスに書き込まれます。

戻り値 この関数は、書き込み衝突が発生した場合 -3 を返します。この関数は、データ文字列内の null 文字に遭遇した場合に '0' を返します。

MasterputsI2C (続き)

備考 : この関数は、null 文字に遭遇するまで、文字列を I²C バスへ書き込みます。MasterputcI2C 関数のコール毎に各バイトが書き込まれます。実際にコールされる関数は、MasterWriteI2C と呼ばれます。MasterWriteI2C と MasterputcI2C は、i2c.h 内の #define 文を使って同じ関数を参照します。

ソース・ファイル : MasterputsI2C.c

コード例 :
```
unsigned char string[] = " MICROCHIP ";
unsigned char *wrptr;
wrptr = string;
MasterputsI2C( wrptr);
```

MasterReadI2C

説明 : この関数は、1 バイトを I²C バスから読み出す際に使用します。

インクルード : i2c.h

プロトタイプ : unsigned char MasterReadI2C(void);

引数 : なし

戻り値 戻り値は、I²C バスから読み出したデータ・バイトです。

備考 : この関数は、I²C バスから 1 バイトを読み出します。
この関数は、MastergetcI2C と同じ機能を実行します。

ソース・ファイル : MasterReadI2C.c

コード例 :
```
unsigned char value;
value = MasterReadI2C();
```

MasterWriteI2C

説明 : この関数は、1 データ・バイトを I²C デバイスへ書き込む際に使用します。

インクルード : i2c.h

プロトタイプ : unsigned char MasterWriteI2C(unsigned char *data_out*);

引数 : *data_out* I²C バス・デバイスへ書き込む 1 バイトのデータ

戻り値 この関数は、書き込み衝突が発生した場合は -1 を返し、その他の場合は 0 を返します。

備考 : この関数は、I²C バス・デバイスへ 1 データ・バイトを書き込みます。
この関数は、MasterputcI2C と同じ機能を実行します。

ソース・ファイル : MasterWriteI2C.c

コード例 : MasterWriteI2C('a');

NotAckI2C

説明 : I²C バスの非アクノレッジ状態を発生します。

インクルード : i2c.h

プロトタイプ : void NotAckI2C(void);

引数 : なし

戻り値 なし

NotAckI2C (続き)

備考 : この関数は、I²C バスの非アクノレッジ状態を発生します。

ソース・ファイル : NotAckI2C.c

コード例 :
```
NotAckI2C();
```

OpenI2C

説明 : I²C モジュールを設定します。

インクルード : i2c.h

プロトタイプ :
```
void OpenI2C(unsigned int config1, unsigned int config2);
```

引数 : *config1* I2CCON レジスタに設定するパラメータ

I²C イネーブル・ビット
I2C_ON
I2C_OFF

アイドル・モードでの I²C ストップ・ビット
I2C_IDLE_STOP
I2C_IDLE_CON

SCL リリース・コントロール・ビット
I2C_CLK_REL
I2C_CLK_HLD

インテリジェント・ペリフェラル・マネジメント・インターフェース・イネーブル・ビット
I2C_IPMI_EN
I2C_IPMI_DIS

10 ビット・スレーブ・アドレス・ビット
I2C_10BIT_ADD
I2C_7BIT_ADD

ディスエーブル・スルーレート・コントロール・ビット
I2C_SLW_DIS
I2C_SLW_EN

SMBus 入力レベル・ビット
I2C_SM_EN
I2C_SM_DIS

一斉コール・イネーブル・ビット
I2C_GCALL_EN
I2C_GCALL_DIS

SCL クロック・ストレッチ・イネーブル・ビット
I2C_STR_EN
I2C_STR_DIS

アクノレッジ・データ・ビット
I2C_ACK
I2C_NACK

アクノレッジ・シーケンス・イネーブル・ビット
I2C_ACK_EN
I2C_ACK_DIS

受信イネーブル・ビット
I2C_RCV_EN
I2C_RCV_DIS

OpenI2C (続き)

ストップ状態イネーブル・ビット
I2C_STOP_EN
I2C_STOP_DIS

繰り返しスタート状態イネーブル・ビット
I2C_RESTART_EN
I2C_RESTART_DIS

スタート状態イネーブル・ビット
I2C_START_EN
I2C_START_DIS

config2 ボーレート・ジェネレータに対する計算値

戻り値 なし

備考 : この関数は、I2C コントロール・レジスタと I2C ボーレート・ジェネレータ・レジスタを設定します。

ソース・ファイル : OpenI2C.c

コード例 :
```
OpenI2C();
```

RestartI2C

説明 : I^2C バスのリスタート状態を発生します。

インクルード : i2c.h

プロトタイプ :
```
void RestartI2C(void);
```

引数 : なし

戻り値 なし

備考 : この関数は、I^2C バス・リスタート状態を発生します。

ソース・ファイル : RestartI2C.c

コード例 :
```
RestartI2C();
```

SlavegetsI2C

説明 : この関数は、I^2C バスから既定のデータ文字列長を読み出します。

インクルード : i2c.h

プロトタイプ :
```
unsigned int SlavegetsI2C(unsigned char *rdptr,
unsigned int i2c_data_wait);
```

引数 :

rdptr I^2C デバイスから読み出したデータを格納する RAM を指す文字型ポインタ

i2c_data_wait モジュールがリターンするまでに待つタイムアウト・カウント
タイムアウト・カウントが 'N' の場合は、実際のタイムアウトは約 (20 * N - 1) 命令サイクルになります。

戻り値 I^2C バスから受信したバイト数を返します。

備考 : このルーチンは、I^2C バスから既定のデータ文字列を読み出します。

ソース・ファイル : SlavegetsI2C.c

SlavegetsI2C (続き)

コード例 :
```
unsigned char string[12];
unsigned char *rdptr;
rdptr = string;
i2c_data_out = 0x11;
SlavegetsI2C(rdptr, i2c_data_wait);
```

SlaveputsI2C

説明 : この関数は、データ文字列を I^2C バスに書き込む際に使用します。

インクルード : `i2c.h`

プロトタイプ :
```
unsigned int SlaveputsI2C(unsigned char *wrptr);
```

引数 :

wrptr	RAM 内のデータ・オブジェクトを指す文字型ポインタ。このデータ・オブジェクトが I^2C デバイスに書き込まれます。

戻り値 この関数は、データ文字列内の null 文字に遭遇した場合に '0' を返します。

備考 : この関数は、null 文字に遭遇するまで、データ文字列を I^2C バスへ書き込みます。

ソース・ファイル : `SlaveputsI2C.c`

コード例 :
```
unsigned char string[] ="MICROCHIP";
unsigned char *rdptr;
rdptr = string;
SlaveputsI2C(rdptr);
```

SlaveReadI2C

説明 : この関数は、1 バイトを I^2C バスから読み出す際に使用します。

インクルード : `i2c.h`

プロトタイプ :
```
unsigned char SlaveReadI2C(void);
```

引数 : なし

戻り値 戻り値は、I^2C バスから読み出したデータ・バイトです。

備考 : この関数は、I^2C バスから 1 バイトを読み出します。この関数は、`SlavegetcI2C` と同じ機能を実行します。

ソース・ファイル : `SlaveReadI2C.c`

コード例 :
```
unsigned char value;
value = SlaveReadI2C();
```

SlaveWriteI2C

説明 : この関数は、1 バイトを I^2C バスに書き込む際に使用します。

インクルード : `i2c.h`

プロトタイプ :
```
void SlaveWriteI2C(unsigned char data_out);
```

引数 :

data_out	I^2C バス・デバイスへ書き込む 1 バイトのデータ

戻り値 なし

備考 : この関数は、I^2C バス・デバイスへ 1 データ・バイトを書き込みます。この関数は、`SlaveputcI2C` と同じ機能を実行します。

SlaveWriteI2C (続き)

ソース・ファイル : SlaveWriteI2C.c

コード例 : SlaveWriteI2C('a');

StartI2C

説明 : I^2C バス・スタート状態を発生します。

インクルード : i2c.h

プロトタイプ : void StartI2C(void);

引数 : なし

戻り値 なし

備考 : この関数は I^2C バス・スタート状態を発生します。

ソース・ファイル : StartI2C.c

コード例 : StartI2C();

StopI2C

説明 : I^2C バスのストップ状態を発生します。

インクルード : i2c.h

プロトタイプ : void StopI2C(void);

引数 : なし

戻り値 なし

備考 : この関数は I^2C バス・ストップ状態を発生します。

ソース・ファイル : StopI2C.c

コード例 : StopI2C();

3.17.2 個別マクロ

EnableIntMI2C

説明 : このマクロは、マスタ I^2C 割り込みをイネーブルします。

インクルード : i2c.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタのマスタ I^2C 割り込みイネーブル・ビットをセットします。

コード例 : EnableIntMI2C;

DisableIntMI2C

説明 : このマクロは、マスタ I^2C 割り込みをディスエーブルします。

インクルード : i2c.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタのマスタ I^2C 割り込みイネーブル・ビットをクリアします。

コード例 : DisableIntMI2C;

SetPriorityIntMI2C

説明 : このマクロは、マスタ I²C 割り込みの優先順位をセットします。

インクルード : i2c.h

引数 : *priority*

備考 : このマクロは、割り込み優先順位コントロール・レジスタのマスタ I²C 割り込み優先順位ビットをセットします。

コード例 :
```
SetPriorityIntMI2C(1);
```

EnableIntSI2C

説明 : このマクロは、スレーブ I²C 割り込みをイネーブルします。

インクルード : i2c.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタのスレーブ I²C 割り込みイネーブル・ビットをセットします。

コード例 :
```
EnableIntSI2C;
```

DisableIntSI2C

説明 : このマクロは、スレーブ I²C 割り込みをディスエーブルします。

インクルード : i2c.h

引数 : なし

備考 : このマクロは、割り込みイネーブル・コントロール・レジスタのスレーブ I²C 割り込みイネーブル・ビットをクリアします。

コード例 :
```
DisableIntSI2C;
```

SetPriorityIntSI2C

説明 : このマクロは、マスタ I²C 割り込みの優先順位をセットします。

インクルード : i2c.h

引数 : *priority*

備考 : このマクロは、割り込み優先順位コントロール・レジスタのマスタ I²C 割り込み優先順位ビットをセットします。

コード例 :
```
SetPriorityIntSI2C(4);
```

3.17.3 使用例

```
#define _ _dsPIC30F6014_ _
#include <p30fxxxx.h>
#include<i2c.h>

void main (void)
{
    unsigned int config2, config1;
    unsigned char *wrptr;
    unsigned char tx_data[] =
    {'M','I','C','R','O','C','H','I','P','¥0'};
    wrptr   = tx_data;
/* Baud rate is set for 100 Khz */
    config2 = 0x11;
/* Configure I2C for 7 bit address mode */
    config1 = (I2C_ON & I2C_IDLE_CON & I2C_CLK_HLD
               & I2C_IPMI_DIS & I2C_7BIT_ADD
               & I2C_SLW_DIS & I2C_SM_DIS &
               I2C_GCALL_DIS & I2C_STR_DIS &
               I2C_NACK & I2C_ACK_DIS & I2C_RCV_DIS &
               I2C_STOP_DIS & I2C_RESTART_DIS
               & I2C_START_DIS);
    OpenI2C(config1,config2);
    IdleI2C();
    StartI2C();
/* Wait till Start sequence is completed */
    while(I2CCONbits.SEN );
/* Write Slave address and set master for transmission */
    MasterWriteI2C(0xE);
/* Wait till address is transmitted */
    while(I2CSTATbits.TBF);
    while(I2CSTATbits.ACKSTAT);
/* Transmit string of data */
    MasterputsI2C(wrptr);
    StopI2C();
/* Wait till stop sequence is completed */
    while(I2CCONbits.PEN);
    CloseI2C();
}
```

第 4 章 . 標準 C ライブラリ (算術関数付き)

4.1 序論

標準 ANSI C ライブラリ関数は、ライブラリ libc-*omf*.a と libm-*omf*.a (算術関数) に含まれています。ここで、*omf* は、選択されたオブジェクト・モジュール・フォーマットに応じて、coff または elf になります。

さらに、いくつかの 16 ビット標準 C ライブラリ・ヘルパー関数と 16 ビット・デバイス用に変更する必要のある標準関数がライブラリ libpic30-*omf*.a の中にあります。

4.1.1 アセンブリ・コード・アプリケーション

この算術関数ライブラリの無償バージョンと対応するヘッダー・ファイルは、マイクロチップのウエブサイトから提供しています。この無償バージョンには、ソース・コードは付いていません。

4.1.2 C コード・アプリケーション

MPLAB C30 C コンパイラのインストール・ディレクトリ (c:¥Program Files¥Microchip¥MPLAB C30) には、ライブラリ関連ファイルの次のサブディレクトリが含まれています :

- lib ―標準 C ライブラリ・ファイル
- *src*¥libm ―ライブラリを再ビルドする際に使う算術ライブラリ関数のソース・コードとバッチ・ファイル
- support¥h ―ライブラリのヘッダー・ファイル

さらに、ResourceGraphs.pdf ファイルがあります。このファイルには、ライブラリ内の各関数が使うリソースの図が記載されています。

4.1.3 本章の構成

本章は次のように構成されています。

- 標準 C ライブラリの使い方

libc-*omf*.a

- <assert.h> 診断
- <ctype.h> 文字処理
- <errno.h> エラー
- <float.h> 浮動小数の特性
- <limits.h> 実装による制約
- <locale.h> ローカライゼーション
- <setjmp.h> 非ロケイル・ジャンプ
- <signal.h> シグナル処理
- <stdarg.h> 変数引数リスト
- <stddef.h> 共通定義
- <stdio.h> 入力と出力
- <stdlib.h> ユーティリティ関数

- <string.h> 文字列関数
- <time.h> 日付関数と時刻関数

libm-*omf*.a

- <math.h> 算術関数

libpic30-*omf*.a

- pic30-libs

4.2 標準 C ライブラリの使い方

標準 C ライブラリを使うアプリケーションをビルドする際には、ヘッダー・ファイルとライブラリ・ファイルの 2 種類のファイルが必要です。

4.2.1 ヘッダー・ファイル

すべての標準 C ライブラリのエンティティは、1 つまたは複数の標準ヘッダー内で宣言または定義されます (**セクション 4.1.3「本章の構成」**のリストを参照)。プログラム内でライブラリ・エンティティを使うときは、関連する標準ヘッダーを指定する include ディレクティブを記述します。

標準ヘッダーの内容が、次のように include ディレクティブ内で指定されることにより、インクルードされます。

```
#include <stdio.h> /* include I/O facilities */
```

標準ヘッダーは任意の順序で記述できます。標準ヘッダーは宣言内に記述しないでください。標準ヘッダーをインクルードする前には、キーワードと同じ名前を持つマクロを定義しないでください。

標準ヘッダーが別の標準ヘッダーをインクルードすることはありません。

4.2.2 ライブラリ・ファイル

アーカイブされたライブラリ・ファイルには、各ライブラリ関数の個々のオブジェクト・ファイルがすべて含まれています。

アプリケーションをリンクするときは、ライブラリ・ファイルをリンカーに対する入力として使って (-- library または -l linker オプションを使用)、アプリケーションで使われている関数がアプリケーションにリンクできるようにする必要があります。

一般的な C アプリケーションでは、libc-*omf*.a、libm-*omf*.a、libpic30-*omf*.a の 3 つのライブラリ・ファイルが必要です (OMF 固有ライブラリの詳細については、**セクション 1.2「OMF 固有のライブラリ / スタートアップ・モジュール」**参照)。これらのライブラリは、MPLAB C30 コンパイラを使ってリンクを行うと、自動的にインクルードされます。

注 :	標準ライブラリ関数によっては、ヒープを必要とするものもあります。これらのライブラリ関数としては、ファイルをオープンする標準 I/O 関数やメモリ割り当て関数などがあります。ヒープの詳細については、*「MPLAB ASM30、MPLAB LINK30 およびユーティリティ・ユーザーズ・ガイド」* (DS51317) と *「MPLAB C30 C コンパイラ・ユーザーズ・ガイド」* (DS51284) を参照してください。

4.3 <ASSERT.H> 診断

ヘッダー・ファイル assert.h は、プログラム内のロジック・エラーをデバッグする際に役立つ 1 つのマクロで構成されています。ある条件が真になる場所に assert 文を使うと、プログラムのロジックをテストすることができます。

<assert.h> をインクルードする前に NDEBUG を定義すると、コードを削除することなく assert テスト機能をターンオフすることができます。マクロ NDEBUG を定義すると、assert() が無視されて、コードは生成されません。

assert

説明： 式が偽の場合、assertion メッセージが stderr に出力されてプログラムが中断されます。

インクルード： <assert.h>

プロトタイプ： `void assert(int expression);`

引数： *expression*　テストする式。

備考： この式はゼロまたは非ゼロに評価されます。ゼロの場合、assert は失敗し、メッセージが stderr へ出力されます。メッセージには、ソース・ファイル名 (__FILE__)、ソース行番号 (__LINE__)、評価された式、メッセージが含まれます。次に、このマクロは関数 abort() をコールします。マクロ _VERBOSE_DEBUGGING が定義された場合は、assert() がコールされる毎に、メッセージが stderr へ出力されます。

例：

```
#include <assert.h> /* for assert */

int main(void)
{
  int a;

  a = 2 * 2;
  assert(a == 4); /* if true-nothing prints */
  assert(a == 6); /* if false-print message */
                  /* and abort */
}
```

出力：

```
sampassert.c:9 a == 6 -- assertion failed
ABRT

with _VERBOSE_DEBUGGING defined:

sampassert.c:8 a == 4 -- OK
sampassert.c:9 a == 6 -- assertion failed
ABRT
```

4.4 <CTYPE.H> 文字処理

ヘッダー・ファイル ctype.h は、文字を分類およびマッピングする際に役立つ関数で構成されています。文字は、標準 C ロケイルに従って解釈されます。

isalnum

説明： 英数文字のテスト。

インクルード： <ctype.h>

プロトタイプ： int isalnum(int *c*);

引数： *c* テストする文字。

戻り値： 文字が英数文字の場合、非ゼロの整数値を返します。その他の場合は、ゼロを返します。

備考： 英数文字は、A ～ Z、a ～ z、0 ～ 9 の範囲内の文字です。

例：

```
#include <ctype.h> /* for isalnum */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = '3';
  if (isalnum(ch))
    printf("3 is an alphanumeric¥n");
  else
    printf("3 is NOT an alphanumeric¥n");

  ch = '#';
  if (isalnum(ch))
    printf("# is an alphanumeric¥n");
  else
    printf("# is NOT an alphanumeric¥n");
}
```

出力：

```
3 is an alphanumeric
# is NOT an alphanumeric
```

isalpha

説明： 英文字のテスト。

インクルード： <ctype.h>

プロトタイプ： int isalpha(int *c*);

引数： *c* テストする文字。

戻り値： 文字が英文字の場合、非ゼロの整数値を返します。その他の場合は、ゼロを返します。

備考： 英文字は、A ～ Z、a ～ z の範囲内の文字です。

isalpha (続き)

例：

```
#include<ctype.h> /* for isalpha */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = 'B';
  if (isalpha(ch))
    printf("B is alphabetic¥n");
  else
    printf("B is NOT alphabetic¥n");

  ch = '#';
  if (isalpha(ch))
    printf("# is alphabetic¥n");
  else
    printf("# is NOT alphabetic¥n");
}
```

出力：

```
B is alphabetic
# is NOT alphabetic
```

iscntrl

説明： 制御文字のテスト。

インクルード： <ctype.h>

プロトタイプ： int iscntrl(int *c*);

引数： *c* テストする文字。

戻り値： 文字が制御文字の場合、非ゼロの整数値を返します。その他の場合は、ゼロを返します。

備考： ASCII 値が 0x00 ～ 0x1F の範囲内、または 0x7F の場合に、文字は制御文字と見なされます。

例：

```
#include <ctype.h> /* for iscntrl */
#include <stdio.h> /* for printf */

int main(void)
{
  char ch;

  ch = 'B';
  if (iscntrl(ch))
    printf("B is a control character¥n");
  else
    printf("B is NOT a control character¥n");

  ch = '¥t';
  if (iscntrl(ch))
    printf("A tab is a control character¥n");
  else
    printf("A tab is NOT a control character¥n");
}
```

出力：

```
B is NOT a control character
A tab is a control character
```

isdigit

説明： 数字のテスト。

インクルード： <ctype.h>

プロトタイプ： int isdigit(int *c*);

引数： *c* テストする文字。

戻り値： 文字が数字の場合、非ゼロの整数値を返します。その他の場合は、ゼロを返します。

備考： 文字が '0' ～ '9' の範囲内の場合、数字と見なされます。

例：

```
#include <ctype.h> /* for isdigit */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = '3';
  if (isdigit(ch))
    printf("3 is a digit¥n");
  else
    printf("3 is NOT a digit¥n");

  ch = '#';
  if (isdigit(ch))
    printf("# is a digit¥n");
  else
    printf("# is NOT a digit¥n");
}
```

出力：

```
3 is a digit
# is NOT a digit
```

isgraph

説明： グラフ文字のテスト。

インクルード： <ctype.h>

プロトタイプ： int isgraph (int *c*);

引数： *c* テストする文字。

戻り値： 文字がグラフ文字の場合、非ゼロの整数値を返します。その他の場合は、ゼロを返します。

備考： スペース以外の印字可能文字である場合、グラフ文字と見なされます。

例：

```
#include <ctype.h> /* for isgraph */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

 ch = '3';
  if (isgraph(ch))
    printf("3 is a graphical character¥n");
  else
    printf("3 is NOT a graphical character¥n");
```

isgraph (続き)

```
 ch = '#';
  if (isgraph(ch))
    printf("# is a graphical character¥n");
  else
    printf("# is NOT a graphical character¥n");

 ch = ' ';
  if (isgraph(ch))
    printf("a space is a graphical character¥n");
  else
    printf("a space is NOT a graphical character¥n");
}
```

出力：

```
3 is a graphical character
# is a graphical character
a space is NOT a graphical character
```

islower

説明： 英小文字のテスト。

インクルード： <ctype.h>

プロトタイプ： int islower (int *c*);

引数： *c* テストする文字。

戻り値： 文字が英小文字の場合、非ゼロの整数値を返します。その他の場合は、ゼロを返します。

備考： 文字が 'a' ～ 'z' の範囲内の場合、英小文字と見なされます。

例：

```
#include <ctype.h> /* for islower */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = 'B';
  if (islower(ch))
    printf("B is lower case¥n");
  else
    printf("B is NOT lower case¥n");

  ch = 'b';
  if (islower(ch))
    printf("b is lower case¥n");
  else
    printf("b is NOT lower case¥n");
}
```

出力：

```
B is NOT lower case
b is lower case
```

isprint

説明： 印字可能文字（スペースも含む）のテスト。

インクルード： <ctype.h>

プロトタイプ： int isprint (int *c*);

引数： *c* テストする文字。

戻り値： 文字が印字可能文字の場合、非ゼロの整数値を返します。その他の場合は、ゼロを返します。

備考： 文字が 0x20 ～ 0x7e の範囲内の場合、印字可能文字と見なされます。

例：

```
#include <ctype.h> /* for isprint */
#include <stdio.h> /* for printf */

int main(void)
{

  int ch;

  ch = '&';
  if (isprint(ch))
    printf("& is a printable character¥n");
  else
    printf("& is NOT a printable character¥n");

  ch = '¥t';
  if (isprint(ch))
    printf("a tab is a printable character¥n");
  else
    printf("a tab is NOT a printable character¥n");
}
```

出力：

```
& is a printable character
a tab is NOT a printable character
```

ispunct

説明： 句読文字のテスト。

インクルード： <ctype.h>

プロトタイプ： int ispunct (int *c*);

引数： *c* テストする文字。

戻り値： 文字が句読文字の場合、非ゼロの整数値を返します。その他の場合は、ゼロを返します。

備考： スペースまたは英数字以外の印字可能文字である場合、句読文字と見なされます。句読文字は次から構成されています：

! " # $ % & ' ( ) ; < = > ? @ [¥] * + , - . / : ^ _ { | } ~

ispunct (続き)

例：

```
#include <ctype.h> /* for ispunct */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = '&';
  if (ispunct(ch))
    printf("& is a punctuation character¥n");
  else
    printf("& is NOT a punctuation character¥n");

  ch = '¥t';
  if (ispunct(ch))
    printf("a tab is a punctuation character¥n");
  else
    printf("a tab is NOT a punctuation character¥n");
}
```

出力：

```
& is a punctuation character
a tab is NOT a punctuation character
```

isspace

説明： 空白文字のテスト。

インクルード： <ctype.h>

プロトタイプ： int isspace (int *c*);

引数： *c* テストする文字。

戻り値： 文字が空白文字の場合、非ゼロの整数値を返します。その他の場合は、ゼロを返します。

備考： スペース (' ')、フォーム・フィード ('¥f')、ニューライン ('¥n')、キャリッジ・リターン ('¥r')、水平タブ ('¥t')、または垂直タブ ('¥v') のいずれかの場合、空白文字と見なされます。

例：

```
#include <ctype.h> /* for isspace */
#include <stdio.h> /* for printf */

int main(void)
{

  int ch;

  ch = '&';
  if (isspace(ch))
    printf("& is a white-space character¥n");
  else
    printf("& is NOT a white-space character¥n");

  ch = '¥t';
  if (isspace(ch))
    printf("a tab is a white-space character¥n");
  else
    printf("a tab is NOT a white-space character¥n");
}
```

isspace (続き)

出力：

```
& is NOT a white-space character
a tab is a white-space character
```

isupper

説明： 英大文字のテスト。

インクルード： <ctype.h>

プロトタイプ： int isupper (int *c*);

引数： *c* テストする文字。

戻り値： 文字が英大文字の場合、非ゼロの整数値を返します。その他の場合は、ゼロを返します。

備考： 文字が 'A' ～ 'Z' の範囲内の場合、英大文字と見なされます。

例：

```
#include <ctype.h> /* for isupper */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = 'B';
  if (isupper(ch))
    printf("B is upper case¥n");
  else
    printf("B is NOT upper case¥n");

  ch = 'b';
  if (isupper(ch))
    printf("b is upper case¥n");
  else
    printf("b is NOT upper case¥n");
}
```

出力：

```
B is upper case
b is NOT upper case
```

isxdigit

説明： 16 進数値のテスト。

インクルード： <ctype.h>

プロトタイプ： int isxdigit (int *c*);

引数： *c* テストする文字。

戻り値： 文字が 16 進数値の場合、非ゼロの整数値を返します。その他の場合は、ゼロを返します。

備考： 文字が '0' ～ '9'、'A' ～ 'F'、または 'a' ～ 'f' の範囲内の場合、16 進数値と見なされます。注 : 先頭の 0x は 16 進数に対するプレフィックスであり、実際の 16 進数値ではないため、0x はリストに含まれません。

isxdigit (続き)

例：

```
#include <ctype.h> /* for isxdigit */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = 'B';
  if (isxdigit(ch))
    printf("B is a hexadecimal digit¥n");
  else
    printf("B is NOT a hexadecimal digit¥n");

  ch = 't';
  if (isxdigit(ch))
    printf("t is a hexadecimal digit¥n");
  else
    printf("t is NOT a hexadecimal digit¥n");
}
```

出力：

```
B is a hexadecimal digit
t is NOT a hexadecimal digit
```

tolower

説明： 文字を英小文字へ変換します。

インクルード： <ctype.h>

プロトタイプ： int tolower (int *c*);

引数： *c* 小文字へ変換する文字。

戻り値： 引数が大文字の場合、対応する英小文字返します。その他の場合元の文字を返します。

備考： 英大文字のみが小文字に変換されます。

例：

```
#include <ctype.h> /* for tolower */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = 'B';
  printf("B changes to lower case %c¥n",
          tolower(ch));

  ch = 'b';
  printf("b remains lower case %c¥n",
          tolower(ch));

  ch = '@';
  printf("@ has no lower case, ");
  printf("so %c is returned¥n", tolower(ch));
}
```

出力：

```
B changes to lower case b
b remains lower case b
@ has no lower case, so @ is returned
```

toupper

説明： 文字を英大文字へ変換します。

インクルード： <ctype.h>

プロトタイプ： int toupper (int *c*);

引数： *c* 大文字へ変換する文字。

戻り値： 引数が小文字の場合、対応する英大文字返します。その他の場合元の文字を返します。

備考： 英小文字のみが大文字に変換されます。

例：

```
#include <ctype.h> /* for toupper */
#include <stdio.h> /* for printf */

int main(void)
{

  int ch;

  ch = 'b';
  printf("b changes to upper case %c¥n",
          toupper(ch));

  ch = 'B';
  printf("B remains upper case %c¥n",
          toupper(ch));

  ch = '@';
  printf("@ has no upper case, ");
  printf("so %c is returned¥n", toupper(ch));
}
```

出力：

```
b changes to upper case B
B remains upper case B
@ has no upper case, so @ is returned
```

4.5 <ERRNO.H> エラー

ヘッダー・ファイル errno.h は、所定のライブラリ関数 (個別関数参照) から報告されるエラー・コードを出力するマクロから構成されています。変数 errno は、ゼロより大きい値を返します。ライブラリ関数がエラーに遭遇するか否かをテストするときは、プログラム側でライブラリ関数をコールする直前に値ゼロを errno に格納しておく必要があります。別の関数コールによりこの値が変更される前に、この値をチェックする必要があります。プログラムの起動時、errno はゼロです。ライブラリ関数が errno をゼロに設定することはありません。

EDOM

説明： 領域エラーを表示します。

インクルード： <errno.h>

備考： EDOM は領域エラーを表示します。このエラーは、関数が定義されている領域の外側に入力引数がある場合に発生します。

ERANGE

説明： オーバーフロー・エラーまたはアンダーフロー・エラーを表示します。

インクルード： <errno.h>

備考： ERANGE はオーバーフロー・エラーまたはアンダーフロー・エラーを表示します。このエラーは、結果が格納できないほど大き過ぎるか小さ過ぎる場合に発生します。

errno

説明： 関数内でエラーが発生したとき、エラーの値を格納します。

インクルード： <errno.h>

備考： エラーが発生した場合ライブラリ関数により、変数 errno に非ゼロ整数値が設定されます。プログラムの起動時、errno はゼロです。これを設定する関数をコールする前に、Errno をゼロにリセットしておく必要があります。

4.6 <FLOAT.H> 浮動小数の特性

ヘッダー・ファイル float.h は、浮動小数型の種々の属性を指定するマクロで構成されています。これらの属性としては、多くの重要な数値、サイズ制限、使用する丸め処理モードなどがあります。

DBL_DIG

説明： 倍精度浮動小数値の精度を表わす桁数。

インクルード： <float.h>

値： デフォルトは 6、スイッチ -fno-short-double を使用の場合は 15。

備考： デフォルトでは、倍精度型は浮動型 (32 ビット表現) と同じサイズです。-fno-short-double スイッチを使うと、倍精度浮動小数値として IEEE 64 ビット表現が使用されます。

DBL_EPSILON

説明： 1.0 と次に大きい表現可能な倍精度浮動小数値との差。

インクルード： <float.h>

値： デフォルトは 1.192093e-07、スイッチ -fno-short-double を使用の場合は 2.220446e-16。

備考： デフォルトでは、倍精度型は浮動型 (32 ビット表現) と同じサイズです。-fno-short-double スイッチを使うと、倍精度浮動小数値として IEEE 64 ビット表現が使用されます。

DBL_MANT_DIG

説明： 倍精度浮動小数値の基数 FLT_RADIX 桁数。

インクルード： <float.h>

値： デフォルトは 24、スイッチ -fno-short-double を使用の場合は 53。

備考： デフォルトでは、倍精度型は浮動型 (32 ビット表現) と同じサイズです。-fno-short-double スイッチを使うと、倍精度浮動小数値として IEEE 64 ビット表現が使用されます。

DBL_MAX

説明： 最大の有限倍精度浮動小数値。

インクルード： <float.h>

値： デフォルトは 3.402823e+38、スイッチ -fno-short-double を使用の場合は 1.797693e+308

備考： デフォルトでは、倍精度型は浮動型 (32 ビット表現) と同じサイズです。-fno-short-double スイッチを使うと、倍精度浮動小数値として IEEE 64 ビット表現が使用されます。

DBL_MAX_10_EXP

説明： 基数 10 の倍精度浮動小数指数の最大整数値。

インクルード： <float.h>

値： デフォルトは 38、スイッチ -fno-short-double を使用の場合は 308。

備考： デフォルトでは、倍精度型は浮動型 (32 ビット表現) と同じサイズです。-fno-short-double スイッチを使うと、倍精度浮動小数値として IEEE 64 ビット表現が使用されます。

DBL_MAX_EXP

説明： 基数 FLT_RADIX の倍精度浮動小数指数部の最大整数値。

インクルード： <float.h>

値： デフォルトは 128、スイッチ -fno-short-double を使用の場合は 1024。

備考： デフォルトでは、倍精度型は浮動型 (32 ビット表現) と同じサイズです。-fno-short-double スイッチを使うと、倍精度浮動小数値として IEEE 64 ビット表現が使用されます。

DBL_MIN

説明： 最小倍精度浮動小数値。

インクルード： <float.h>

値： デフォルトは 1.175494e-38、スイッチ -fno-short-double を使用の場合は 2.225074e-308。

備考： デフォルトでは、倍精度型は浮動型 (32 ビット表現) と同じサイズです。-fno-short-double スイッチを使うと、倍精度浮動小数値として IEEE 64 ビット表現が使用されます。

DBL_MIN_10_EXP

説明： 基数 10 の倍精度浮動小数指数の負の最小整数値。

インクルード： <float.h>

値： デフォルトは -37、スイッチ -fno-short-double を使用の場合は -307。

備考： デフォルトでは、倍精度型は浮動型 (32 ビット表現) と同じサイズです。-fno-short-double スイッチを使うと、倍精度浮動小数値として IEEE 64 ビット表現が使用されます。

DBL_MIN_EXP

説明： 基数 FLT_RADIX の倍精度浮動小数指数の負の最小整数値。

インクルード： <float.h>

値： デフォルトは -125、スイッチ -fno-short-double を使用の場合は -1021。

備考： デフォルトでは、倍精度型は浮動型 (32 ビット表現) と同じサイズです。-fno-short-double スイッチを使うと、倍精度浮動小数値として IEEE 64 ビット表現が使用されます。

FLT_DIG

説明： 単精度浮動小数値の精度を表わす桁数。

インクルード： <float.h>

値： 6

FLT_EPSILON

説明： 1.0 と次に大きい表現可能な単精度浮動小数値との差。

インクルード： <float.h>

値： 1.192093e-07

FLT_MANT_DIG

説明： 単精度浮動小数値の基数 FLT_RADIX 桁数。

インクルード： <float.h>

値： 24

FLT_MAX

説明： 最大の有限単精度浮動小数値。

インクルード： <float.h>

値： 3.402823e+38

FLT_MAX_10_EXP

説明： 基数 10 の単精度浮動小数指数の最大整数値。

インクルード： <float.h>

値： 38

FLT_MAX_EXP

説明： 基数 FLT_RADIX の単精度浮動小数指数の最大整数値。
インクルード： <float.h>
値： 128

FLT_MIN

説明： 最小単精度浮動小数値。
インクルード： <float.h>
値： 1.175494e-38

FLT_MIN_10_EXP

説明： 基数 10 の単精度浮動小数指数の負の最小整数値。
インクルード： <float.h>
値： -37

FLT_MIN_EXP

説明： 基数 FLT_RADIX の単精度浮動小数指数の負の最小整数値。
インクルード： <float.h>
値： -125

FLT_RADIX

説明： 指数表現の基数。
インクルード： <float.h>
値： 2
備考： 指数の基数は、2 すなわちバイナリ。

FLT_ROUNDS

説明： 浮動小数演算の丸め処理モードを表示します。
インクルード： <float.h>
値： 1
備考： 最寄りの値にまるめ処理します。

LDBL_DIG

説明： ロング倍精度浮動小数値の精度を表わす桁数。
インクルード： <float.h>
値： 15

LDBL_EPSILON

説明： 1.0 と次に大きい表現可能なロング倍精度浮動小数値との差。
インクルード： <float.h>
値： 2.220446e-16

LDBL_MANT_DIG

説明： ロング倍精度浮動小数値の基数 FLT_RADIX の桁数。
インクルード： <float.h>
値： 53

LDBL_MAX

説明： 最大の有限ロング倍精度浮動小数値。
インクルード： <float.h>
値： 1.797693e+308

LDBL_MAX_10_EXP

説明： 基数 10 のロング倍精度浮動小数指数の最大整数値。
インクルード： <float.h>
値： 308

LDBL_MAX_EXP

説明： 基数 FLT_RADIX のロング倍精度浮動小数指数部の最大整数値。
インクルード： <float.h>
値： 1024

LDBL_MIN

説明： 最小ロング倍精度浮動小数値。
インクルード： <float.h>
値： 2.225074e-308

LDBL_MIN_10_EXP

説明： 基数 10 のロング倍精度浮動小数指数の負の最小整数値。
インクルード： <float.h>
値： -307

LDBL_MIN_EXP

説明： 基数 FLT_RADIX のロング倍精度浮動小数指数の負の最小整数値。
インクルード： <float.h>
値： -1021

4.7 <LIMITS.H> 実装による制約

ヘッダー・ファイル limits.h は、整数型の最小値と最大値を定義するマクロで構成されています。これらの各マクロは、#if 処理ディレクティブの中で使うことができます。

CHAR_BIT

説明： 型 char を表現するビット数。
インクルード： <limits.h>
値： 8

CHAR_MAX

説明： char の最大値。
インクルード： <limits.h>
値： 127

CHAR_MIN

説明： char の最小値。
インクルード： <limits.h>
値： -128

INT_MAX

説明： int の最大値。
インクルード： <limits.h>
値： 32767

INT_MIN

説明： int の最小値。
インクルード： <limits.h>
値： -32768

LLONG_MAX

説明： long long int の最大値
インクルード： <limits.h>
値： 9223372036854775807

LLONG_MIN

説明： long long int の最小値
インクルード： <limits.h>
値： -9223372036854775808

LONG_MAX

説明： long int の最大値
インクルード： <limits.h>
値： 2147483647

LONG_MIN

説明： long int の最小値
インクルード： <limits.h>
値： -2147483648

MB_LEN_MAX

説明： マルチバイト文字内の最大バイト数
インクルード： <limits.h>
値： 1

SCHAR_MAX

説明： signed char の最大値。
インクルード： <limits.h>
値： 127

SCHAR_MIN

説明： char の最大値。
インクルード： <limits.h>
値： -128

SHRT_MAX

説明： short int の最大値
インクルード： <limits.h>
値： 32767

SHRT_MIN

説明： short int の最小値
インクルード： <limits.h>
値： -32768

UCHAR_MAX

説明： unsigned char の最大値。
インクルード： <limits.h>
値： 255

UINT_MAX

説明： unsigned int の最大値
インクルード： <limits.h>
値： 65535

ULLONG_MAX

説明： long long unsigned int の最大値。
インクルード： <limits.h>
値： 18446744073709551615

ULONG_MAX

説明： long unsigned int の最大値
インクルード： <limits.h>
値： 4294967295

USHRT_MAX

説明： unsigned short int の最大値
インクルード： <limits.h>
値： 65535

4.8 <LOCALE.H> ローカライゼーション

このコンパイラはデフォルトとして C ロケイルを使い、他のロケイルをサポートしていません。したがって、ヘッダー・ファイル locale.h をサポートしません。このファイルには通常、次の内容が記載されています :

- struct lconv
- NULL
- LC_ALL
- LC_COLLATE
- LC_CTYPE
- LC_MONETARY
- LC_NUMERIC
- LC_TIME
- localeconv
- setlocale

4.9 <SETJMP.H> 非ロケイル・ジャンプ

ヘッダー・ファイル setjmp.h は、通常の関数コールとリターン処理をバイパスする制御の移動を可能にする型、マクロ、関数で構成されています。

jmp_buf

説明： プログラム環境を待避および復旧させる setjmp と longjmp により使用される配列型。

インクルード： <setjmp.h>

プロトタイプ： typedef int jmp_buf[_NSETJMP];

備考： _NSETJMP は、16 個のレジスタと 32 ビット戻りアドレスを表わす 16 + 2 として決定されます。

setjmp

説明： プログラムの現在の状態を後で使うために longjmp により待避させるマクロ。

インクルード： <setjmp.h>

プロトタイプ： #define setjmp(jmp_buf *env*)

引数： *env* 環境を格納する変数

戻り値： ダイレクト・コールからのリターンの場合、setjmp はゼロを返します。longjmp に対するコールからのリターンの場合、setjmp は非ゼロ値を返します。
注： longjmp からの引数 val が 0 の場合、setjmp は 1 を返します。

例： longjmp を参照してください。

longjmp

説明： setjmp により待避させた環境を復旧させる関数。

インクルード： <setjmp.h>

プロトタイプ： void longjmp(jmp_buf *env*, int *val*);

引数： *env* 環境を格納する変数

val setjmp コールに対して返される値

備考： 値パラメータ val は、非ゼロである必要があります。longjmp がネストされたシグナル・ハンドラから起動された場合 (すなわち、別のシグナル処理中にシグナルが発生したことにより起動された場合)、動作は不定になります。

4.10 <SIGNAL.H> シグナル処理

ヘッダー・ファイル signal.h は、型、複数のマクロ、およびプログラムの実行中にシグナルの処理方法を指定する 2 つの関数で構成されています。シグナルとは、プログラムの実行中に報告される状態を意味します。シグナルは、raise 関数を使ったソフトウェア制御のもとで同期して発生します。

シグナルは次により処理されます :

- デフォルト処理 (SIG_DFL): シグナルは致命的エラーで実行停止として扱われます。
- シグナル (SIG_IGN) の無視 : シグナルは無視され、制御がユーザー・アプリケーションに戻されます。
- signal を使って指定した関数によりシグナルを処理します。

デフォルトでは、すべての signal が SIG_DFL により指定されるデフォルト・ハンドラにより処理されます。

型 sig_atomic_t は、プログラムがアトミックにアクセスする整数型です。この型をキーワード volatile と一緒に使用すると、シグナル・ハンドラはデータ・オブジェクトを残りのプログラムと共用することができます。

sig_atomic_t

説明 : シグナル・ハンドラが使用する型

インクルード : <signal.h>

プロトタイプ : typedef int sig_atomic_t;

SIG_DFL

説明 : 2 つ目の引数および / または signal の戻り値として使い、デフォルト・ハンドラが特定のシグナルを使用するように指定します。

インクルード : <signal.h>

SIG_ERR

説明 : エラーのために signal が要求を完了できないとき、signal の戻り値として使用。

インクルード : <signal.h>

SIG_IGN

説明 : 2 つ目の引数および / または signal の戻り値として使い、シグナルを無視するように指定します。

インクルード : <signal.h>

SIGABRT

説明： 異常終了シグナルの名前。

インクルード： <signal.h>

プロトタイプ： #define SIGABRT

備考： SIGABRT は異常終了シグナルを表示し、raise または signal と組み合わせて使用。デフォルトの raise 動作 (SIG_DFL により指定される動作) は、次のように標準エラー・ストリームを出力することです :

```
abort - terminating
```

シグナル名の一般的な使い方とシグナル処理については、signal に付随している例を参照してください。

例：

```
#include <signal.h> /* for raise, SIGABRT */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGABRT);
  printf("Program never reaches here.");
}
```

出力：

```
ABRT
```

説明：

ABRT は " アボート " の省略形です。

SIGFPE

説明： ゼロによる除算や演算結果が範囲外などの浮動小数エラーの通知。

インクルード： <signal.h>

プロトタイプ： #define SIGFPE

備考： SIGFPE は、raise および / または signal の引数として使用されます。使用した場合、デフォルトの動作は演算エラー・メッセージを出力して、コールしたプログラムを停止させることです。シグナル・ハンドラの動作を決定するユーザー関数の方がこれに優先します。ユーザー定義関数の例については signal を参照してください。

例：

```
#include <signal.h> /* for raise, SIGFPE */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGFPE);
  printf("Program never reaches here");
}
```

出力：

```
FPE
```

説明：

FPE は、" 浮動小数エラー " の省略です。

SIGILL

説明： 不当命令を通知します。

インクルード： <signal.h>

プロトタイプ： #define SIGILL

備考： SIGILL は、raise および / または signal の引数として使われます。使用した場合、デフォルトの動作は不当実行可能・コード・メッセージを出力して、コールしたプログラムを停止させることです。シグナル・ハンドラの動作を決定するユーザー関数の方がこれに優先します。ユーザー定義関数の例については signal を参照してください。

例：

```
#include <signal.h> /* for raise, SIGILL */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGILL);
  printf("Program never reaches here");
}
```

出力：

```
ILL
```

説明：

ILL は、" 不当命令 " の省略形です。

SIGINT

説明： Interrupt signal.

インクルード： <signal.h>

プロトタイプ： #define SIGINT

備考： SIGINT は、raise および / または signal の引数として使われます。使用した場合、デフォルトの動作は割り込みメッセージを出力して、コールしたプログラムを停止させることです。シグナル・ハンドラの動作を決定するユーザー関数の方がこれに優先します。ユーザー定義関数の例については signal を参照してください。

例：

```
#include <signal.h> /* for raise, SIGINT */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGINT);
  printf("Program never reaches here.");
}
```

出力：

```
INT
```

説明：

INT は、" 割り込み " の省略形です。

SIGSEGV

説明： ストレージに対する無効アクセスの通知。

インクルード： <signal.h>

プロトタイプ： #define SIGSEGV

備考： SIGSEGV は、raise および / または signal の引数として使われます。使用した場合、デフォルトの動作は無効ストレージ要求メッセージを出力して、コールしたプログラムを停止させることです。シグナル・ハンドラの動作を決定するユーザー関数の方がこれに優先します。ユーザー定義関数の例については signal を参照してください。

例：

```
#include <signal.h> /* for raise, SIGSEGV */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGSEGV);
  printf("Program never reaches here.");
}
```

出力：

```
SEGV
```

説明：

SEGV は " 無効なストレージ・アクセス " の省略形です。

SIGTERM

説明： 終了要求の通知。

インクルード： <signal.h>

プロトタイプ： #define SIGTERM

備考： SIGTERM は、raise および / または signal の引数として使われます。使用した場合、デフォルトの動作は終了要求メッセージを出力して、コールしたプログラムを停止させることです。シグナル・ハンドラの動作を決定するユーザー関数の方がこれに優先します。ユーザー定義関数の例については signal を参照してください。

例：

```
#include <signal.h> /* for raise, SIGTERM */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGTERM);
  printf("Program never reaches here.");
}
```

出力：

```
TERM
```

説明：

TERM は、" 終了要求 " の省略形。

raise

説明： 同期シグナルを報告。

インクルード： <signal.h>

プロトタイプ： int raise(int *sig*);

引数： *sig* シグナル名

戻り値： 正常終了の場合 0 を返します。その他の場合は非ゼロ値を返します。

備考： raise は、sig で識別したシグナルを実行中のプログラムへ送ります。

例：

```
#include <signal.h>  /* for raise, signal, */
          /* SIGILL, SIG_DFL  */
#include <stdlib.h>  /* for div, div_t   */
#include <stdio.h>   /* for printf     */
#include <p30f6014.h> /* for INTCON1bits */

void __attribute__((__interrupt__))
_MathError(void)
{
  raise(SIGILL);
  INTCON1bits.MATHERR = 0;
}

void illegalinsn(int idsig)
{
  printf("Illegal instruction executed¥n");
  exit(1);
}
int main(void)
{
  int x, y;
  div_t z;

  signal(SIGILL, illegalinsn);
  x = 7;
  y = 0;
  z = div(x, y);
  printf("Program never reaches here");
}
```

出力：

```
Illegal instruction executed
```

説明：

この例は、リンカー・スクリプト p30f6014.gld を必要とします。この例は 3 つの部分から構成されています。

最初の部分の割り込みハンドラは、実行中のプログラムへ不当命令 signal (SIGILL) を送信することにより算術エラーを処理する割り込みベクタ _MathError 用に書かれています。割り込みハンドラ内の最後の文は、例外フラグをクリアします。

2 つ目の部分の関数 illegalinsn は、エラー・メッセージを出力してコール終了させます。

3 つ目の部分では、main 内で、signal (SIGILL、illegalinsn) が SIGILL のハンドラを関数 illegalinsn に設定します。

ゼロによる除算で算術エラーが発生すると、_MathError 割り込みベクタがコールされます。そうすると、SIGILL のハンドラ関数 (関数 illegalinsn) をコールするシグナルが発生します。これにより、エラー・メッセージが出力されて、プログラムが停止します。

signal

説明： 割り込みシグナル処理を制御します。

インクルード： <signal.h>

プロトタイプ： void (*signal(int *sig*, void(**func*)(int)))(int);

引数： *sig* シグナル名

func 実行される関数

戻り値： *func* の前の値を返します。

例：

```
#include <signal.h> /* for signal, raise, */
                    /* SIGINT, SIGILL, */
                    /* SIG_IGN, and SIGFPE */
#include <stdio.h>  /* for printf */

/* Signal handler function */
void mysigint(int id)
{
  printf("SIGINT received¥n");
}

int main(void)
{
  /* Override default with user defined function */
  signal(SIGINT, mysigint);
  raise(SIGINT);

  /* Ignore signal handler */
  signal(SIGILL, SIG_IGN);
  raise(SIGILL);
  printf("SIGILL was ignored¥n");

  /* Use default signal handler */
  raise(SIGFPE);
  printf("Program never reaches here.");
}
```

出力：

```
SIGINT received
SIGILL was ignored
FPE
```

説明：

関数 mysigint は、SIGINT に対するユーザー定義のシグナル・ハンドラです。メイン・プログラム内で、関数 signal がコールされて、シグナル SIGINT (これがデフォルトの動作を上書き) のシグナル・ハンドラ (mysigint) が設定されます。シグナル SIGINT を報告するため、関数 raise がコールされます。これにより、SIGINT のシグナル・ハンドラがユーザー定義の関数 (mysigint) をシグナル・ハンドラとして使うようになり、"SIGINT received" メッセージが出力さます。

次に、関数 signal がコールされて、シグナル SIGILL のシグナル・ハンドラ SIG_IGN が設定されます。定数 SIG_IGN を使って、シグナルが無視されることが表示されます。シグナル SIGILL が無視されることを報告するために、関数 raise がコールされます。

シグナル SIGFPE を報告するため、関数 raise が再度コールされます。SIGFPE にはユーザー定義の関数がないため、デフォルトのシグナル・ハンドラを使って、メッセージ "FPE" (" 演算エラー—終了 " の省略形) が出力されます。次に、コールしたプログラムが終了します。printf 文には到達しません。

4.11 <STDARG.H> 変数引数リスト

ヘッダー・ファイル stdarg.h は、変数引数リストを使って関数をサポートします。このヘッダー・ファイルを使うと、関数が対応するパラメータ宣言なしで引数を持つことができるようになります。名前付き引数が少なくとも 1 つ必要です。変数引数は、省略記号 (...) で表わされます。関数内部で、型 va_list のオブジェクトを宣言して引数を持てるようにする必要があります。va_start が引数リストに対する変数を初期化し、va_arg が引数リストをアクセスし、va_end が引数の使用を終了させます。

va_list

説明 : 型 va_list は、可変長引数リスト内の各引数を参照する変数を宣言します。

インクルード : <stdarg.h>

例 : va_arg を参照してください。

va_arg

説明 : 現在の引数を取得します。

インクルード : <stdarg.h>

プロトタイプ : #define va_arg(va_list *ap*, *Ty*)

引数 : *ap* 引数のリストを指すポインタ

Ty 取得する引数の型

戻り値 : 現在の引数を返します。

備考 : va_start は、va_arg の前にコールされる必要があります。

例 :

```
#include <stdio.h>   /* for printf */
#include <stdarg.h>  /* for va_arg, va_start,
                        va_list, va_end */

void tprint(const char *fmt, ...)
{
  va_list ap;

  va_start(ap, fmt);
  while (*fmt)
  {
    switch (*fmt)
    {
```

va_arg

```
        case '%':
                fmt++;
                if (*fmt == 'd')
                {
                  int d = va_arg(ap, int);
                  printf("<%d> is an integer¥n",d);
                }
                else if (*fmt == 's')
                {
                  char *s = va_arg(ap, char*);
                  printf("<%s> is a string¥n", s);
                }
                else
                {
                  printf("%%%c is an unknown format¥n",
                       *fmt);
                }
                fmt++;
                break;
          default:
                printf("%c is unknown¥n", *fmt);
                fmt++;
                break;
      }
    }
    va_end(ap);
}

int main(void)
{
  tprint("%d%s.%c", 83, "This is text.", 'a');
}
```

出力：

```
<83> is an integer
<This is text.> is a string
. is unknown
%c is an unknown format
```

va_end

説明： *ap* の使用を終了。
インクルード： <stdarg.h>
プロトタイプ： #define va_end(va_list *ap*)
引数： *ap* 引数のリストを指すポインタ
備考： va_end に対するコールの後は、引数リストのポインタ ap は無効と見なされます。va_arg に対してさらにコールすることは、次の va_start までできません。MPLAB C30 では、va_end は何も実行しません。したがって、このコールは不要ですが、可読性と移植可能性のために使っています。
例： va_arg を参照してください。

va_start

説明： 可変長引数リスト内の最初のオプションの引数に対して引数ポインタ ap を設定します。
インクルード： <stdarg.h>
プロトタイプ： #define va_start(va_list *ap*, *last_arg*)
引数： *ap* 引数のリストを指すポインタ
last_arg オプションの引数の前にある最後の名前付き引数
例： va_arg を参照してください。

4.12 <STDDEF.H> 共通定義

ヘッダー・ファイル stddef.h は、複数の型とプログラム内で広く使われるマクロで構成されています。

ptrdiff_t

説明： 2 つのポインタの減算結果の型。
インクルード： <stddef.h>

size_t

説明： sizeof オペレータの適用結果の型。
インクルード： <stddef.h>

wchar_t

説明： ワイド文字値を保持する型。
インクルード： <stddef.h>

NULL

説明： null ポインタ定数の値。
インクルード： <stddef.h>

offsetof

説明： 構造体の先頭からの構造体メンバーのオフセットを与えます。

インクルード： <stddef.h>

プロトタイプ： #define offsetof(*T*, *mbr*)

引数： *T* 構造体の名前

mbr 構造体 *T* 内のメンバーの名前

戻り値： 構造体の先頭からの、指定されたメンバー (*mbr*) のオフセットをバイト数で返します。

備考： マクロ offsetof は、ビットフィールドに対して不定になります。ビットフィールドが使用されると、エラー・メッセージが発生します。

例：

```
#include <stddef.h> /* for offsetof */
#include <stdio.h>  /* for printf */

struct info {
  char item1[5];
  int item2;
  char item3;
  float item4;
};

int main(void)
{
  printf("Offset of item1 = %d¥n",
          offsetof(struct info,item1));
  printf("Offset of item2 = %d¥n",
          offsetof(struct info,item2));
  printf("Offset of item3 = %d¥n",
          offsetof(struct info,item3));
  printf("Offset of item4 = %d¥n",
          offsetof(struct info,item4));
}
```

出力：

```
Offset of item1 = 0
Offset of item2 = 6
Offset of item3 = 8
Offset of item4 = 10
```

説明：

このプログラムは、構造体の先頭からの、各構造体メンバーのオフセットをバイト数で表示します。item1 は 5 バイト (char item1[5]) だけですが、item2 のアドレスが偶数境界になるように、パディングが行われています。同様に item3 でも、1 バイト (char item3) に対して 1 バイトのパディングが行われています。

4.13 <STDIO.H> 入力と出力

ヘッダー・ファイル stdio.h は、型、マクロ、およびファイルとストリームに対する入 / 出力動作の実行をサポートする関数で構成されています。ファイルが開かれると、ストリームに対応付けられます。ストリームは、ファイルへ入出力するデータ・フローに対する 1 つのパイプラインです。さまざまなシステムが異なる属性を使うため、ストリームが統一された属性を提供して、ファイルの読み書きを可能にします。

ストリームとしては、テキスト・ストリームまたはバイナリ・ストリームが可能です。テキスト・ストリームは、行に分割された文字シーケンスから構成されています。各行は、ニューライン ('¥n') 文字で終了します。文字の内部表現は変更可能で、特定に行の終わりに関しては変更されます。バイナリ・ストリームは、情報バイトのシーケンスで構成されています。バイナリ・ストリームへ転送されるバイトは変更されません。行の概念がなく、ファイルは単なるバイトの列でできています。

起動時に、stdin、stdout、stderr の 3 つのストリームが自動的に開かれます。stdin は標準入力のストリームを、stdout は標準出力を、stderr は標準エラーを、それぞれ提供します。その他のストリームは、fopen 関数により生成されます。許容されるさまざまなタイプのファイル・アクセスについては、fopen を参照してください。これらのアクセス・タイプは、fopen と freopen によって使用されます。

型 FILE は、開かれた各ファイル・ストリームの情報を保存するために使います。これには、エラー・インジケータ、end-of-file インジケータ、ファイル位置インジケータ、ストリームの制御に必要なその他の内部ステータス情報などが含まれます。stdio 内の多くの関数が FILE を引数として使います。

バッファリングには、バッファなし、ライン・バッファ、フル・バッファの 3 種類があります。バッファなしは、文字またはバイトが即座に転送されることを意味します。ライン・バッファでは、行全体になるまで集めて 1 回で転送します (すなわち、ニューライン文字が行の終わりを表示します)・フル・バッファを使うと、任意サイズのブロックを転送することができます。関数 setbuf と setvbuf は、ファイルのバッファリングを制御します。

また、stdio.h ファイルも、入力フォーマットと出力フォーマットを使う関数を含んでいます。入力フォーマットすなわちスキャン・フォーマットは、データの読み出しに使われます。これらの説明は scanf のところに記載されていますが、これらは fscanf と sscanf でも使用しています。出力フォーマットすなわちプリント・フォーマットは、データの書き込みに使われます。これらの説明は、printf のところに記載されています。また、これらのプリント・フォーマットは、fprintf、sprintf、vfprintf、vprintf、vsprintf ででも使われます。

コンパイラ・オプションによっては、標準 I/O の実行方法に影響を与えるものがあります。フォーマット化された I/O ルーチンのより目的にかなったバージョンを提供するために、ツール・チェインにより printf または scanf スタイルの関数に対するコールを別のコールへ変換することができます。このオプションの概要を次に示します :

- -msmart-io オプションはイネーブルされると、printf、scanf、および入出力フォーマットを使うその他の関数を整数専用型に変換しようとします。機能は C 標準形式と変わりませんが、浮動小数出力のサポートがありません。-msmart-io=0 はこの機能をディスエーブルし、変換を行いません。-msmart-io=1 または -msmart-io (デフォルト) は、I/O 関数が浮動小数変換により提供されていないことが確認できる場合、関数コールを変換します。-msmart-io=2 はデフォルトより楽観的で、非定数フォーマット・ストリングまたは未知のフォーマット・ストリングには浮動小数フォーマットが含まれていないものと仮定しています。-msmart-io=2 が浮動小数フォーマットと一緒に使用された場合には、フォーマット文字がリテラル・テキストとして表示され、対応する引数は使用されません。
- -fno-short-double は、ロング倍精度型をサポートしているかのように、コンパイラに倍精度をサポートするフォーマット化された I/O ルーチンに対するコールを生成させます。

これらのオプションを使ってコンパイルしたモジュールを混在させると、実行可能形式のサイズが大きくなるか、あるいは複数のモジュール間でラージおよびスモールの倍精度データが共用される場合には正しく実行されなくなります。

FILE

説明： ファイル・ストリームの情報を格納。
インクルード： <stdio.h>

fpos_t

説明： ファイル位置の格納に使用する変数の型。
インクルード： <stdio.h>

size_t

説明： sizeof オペレータの適用結果の型。
インクルード： <stdio.h>

_IOFBF

説明： フル・バッファリングの表示。
インクルード： <stdio.h>
備考： 関数 setvbuf により使用されます。

_IOLBF

説明： ライン・バッファリングの表示。
インクルード： <stdio.h>
備考： 関数 setvbuf により使用されます。

_IONBF

説明： バッファなしの表示。
インクルード： <stdio.h>
備考： 関数 setvbuf により使用されます。

BUFSIZ

説明： 関数 setbuf によって使用されるバッファ・サイズの定義。
インクルード： <stdio.h>
値： 512

EOF

説明： end-of-file に到達またはエラー状態を報告する負の数値。

インクルード： <stdio.h>

備考： end-of-file に到達すると、end-of-file インジケータが設定されます。エラー状態に遭遇すると、エラー・インジケータが設定されます。エラー状態には、書き込みエラーと入力すなわち読み出しエラーが含まれます。

FILENAME_MAX

説明： null ターミネータを含むファイル名内の最大文字数。

インクルード： <stdio.h>

値： 260

FOPEN_MAX

説明： 同時に開くことができる最大ファイル数の指定。

インクルード： <stdio.h>

値： 8

備考： stderr、stdin、stdout は、FOPEN_MAX のカウントに含まれます。

L_tmpnam

説明： 関数 tmpnam により生成されるテンポラリ・ファイルの最長名前の文字数の指定。

インクルード： <stdio.h>

値： 16

備考： L_tmpnam は、tmpnam により使用される配列のサイズを指定する際に使われます。

NULL

説明： null ポインタ定数の値。

インクルード： <stdio.h>

SEEK_CUR

説明： fseek がファイル・ポインタの現在位置から探すことを表示。

インクルード： <stdio.h>

例： fseek の例参照。

SEEK_END

説明： fseek がファイルの終わりから探すことを表示。
インクルード： <stdio.h>
例： fseek の例参照。

SEEK_SET

説明： fseek がファイルの先頭から探すことを表示。
インクルード： <stdio.h>
例： fseek の例参照。

stderr

説明： 標準エラー・ストリームを指すファイル・ポインタ。
インクルード： <stdio.h>

stdin

説明： 標準入力ストリームを指すファイル・ポインタ。
インクルード： <stdio.h>

stdout

説明： 標準出力ストリームを指すファイル・ポインタ。
インクルード： <stdio.h>

TMP_MAX

説明： 関数 tmpnam が生成できる独自なファイル名の最大数。
インクルード： <stdio.h>
値： 32

clearerr

説明： ストリームのエラー・インジケータのリセット。

インクルード： <stdio.h>

プロトタイプ： void clearerr(FILE *stream);

引数： *stream* エラー・インジケータをリセットするストリーム

備考： この関数は、与えられたストリームの end-of-file インジケータとエラー・インジケータをクリアします (すなわち関数 clearerr がコールされた後に、feof と ferror は偽を返します)。

例：

```
/* This program tries to write to a file that is */
/* readonly.This causes the error indicator to  */
/* be set.The function ferror is used to check */
/* the error indicator.The function clearerr is */
/* used to reset the error indicator so the next */
/* time ferror is called it will not report an  */
/* error.                    */
#include <stdio.h> /* for ferror, clearerr, */
        /* printf, fprintf, fopen,*/
        /* fclose, FILE, NULL   */


int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("sampclearerr.c", "r")) ==
       NULL)
    printf("Cannot open file¥n");
  else
  {
    fprintf(myfile, "Write this line to the "
            "file.¥n");
    if (ferror(myfile))
      printf("Error¥n");
    else
      printf("No error¥n");
    clearerr(myfile);
    if (ferror(myfile))
      printf("Still has Error¥n");
    else
      printf("Error indicator reset¥n");

    fclose(myfile);
  }
}
```

出力：

```
Error
Error indicator reset
```

fclose

説明： ストリームを閉じます。

インクルード： <stdio.h>

プロトタイプ： int fclose(FILE *stream);

引数： *stream* 閉じるストリームを指すポインタ

戻り値： 正常終了の場合 0 を返します。エラーが検出された場合は EOF を返します。

備考： fclose は、バッファ付き出力をファイルへ書き込みます。

例：

```
#include <stdio.h> /* for fopen, fclose,     */
                   /* printf,FILE, NULL, EOF */

int main(void)
{
  FILE *myfile1, *myfile2;
  int y;

  if ((myfile1 = fopen("afile1", "w+")) == NULL)
    printf("Cannot open afile1¥n");
  else
  {
    printf("afile1 was opened¥n");

    y = fclose(myfile1);
    if (y == EOF)
      printf("afile1 was not closed¥n");
    else
      printf("afile1 was closed¥n");
  }
}
```

出力：

```
afile1 was opened
afile1 was closed
```

feof

説明： end-of-file のテスト。

インクルード： <stdio.h>

プロトタイプ： int feof(FILE *stream);

引数： *stream* end-of-file をチェックするストリーム

戻り値： ストリームが end-of-file にある場合非ゼロを返します。その他の場合はゼロを返します。

例：

```
#include <stdio.h> /* for feof, fgetc, fputc, */
                   /* fopen, fclose, FILE,   */
                   /* NULL */

int main(void)
{
  FILE *myfile;
  int y = 0;

  if( (myfile = fopen( "afile.txt", "rb" )) == NULL )
    printf( "Cannot open file¥n" );
  else
  {
    for (;;)
    {
      y = fgetc(myfile);
      if (feof(myfile))
        break;
      fputc(y, stdout);
    }
    fclose( myfile );
  }
}
```

入力：

afile.txt の内容 (入力として使用):

```
This is a sentence.
```

出力：

```
This is a sentence.
```

ferror

説明： エラー・インジケータがセットされたか否かをテストします。

インクルード： <stdio.h>

プロトタイプ： int ferror(FILE *stream);

引数： *stream* FILE 構造体を指すポインタ

戻り値： エラー・インジケータがセットされている場合非ゼロ値を返します。その他の場合はゼロを返します。

例：

```
/* This program tries to write to a file that is */
/* readonly.This causes the error indicator to */
/* be set.The function ferror is used to check */
/* the error indicator and find the error.The */
/* function clearerr is used to reset the error */
/* indicator so the next time ferror is called */
/* it will not report an error.        */

#include <stdio.h> /* for ferror, clearerr, */
        /* printf, fprintf,   */
        /* fopen, fclose,    */
        /* FILE, NULL     */


int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("sampclearerr.c", "r")) ==
       NULL)
    printf("Cannot open file¥n");
  else
  {
    fprintf(myfile, "Write this line to the "
            "file.¥n");
    if (ferror(myfile))
      printf("Error¥n");
    else
      printf("No error¥n");
    clearerr(myfile);
    if (ferror(myfile))
      printf("Still has Error¥n");
    else
      printf("Error indicator reset¥n");

    fclose(myfile);
  }
}
```

出力：

```
Error
Error indicator reset
```

fflush

説明： 指定されたストリーム内のバッファをクリアします。

インクルード： <stdio.h>

プロトタイプ： int fflush(FILE *stream);

引数： *stream* クリアするストリームを指すポインタ

戻り値： 書き込みエラーが発生した場合 EOF を返します。正常終了の場合はゼロを返します。

備考： ストリームが null ポインタである場合、すべての出力バッファがファイルへ書き込まれます。fflush がバッファなしストリームへ影響を与えることはありません。

fgetc

説明： ストリームから文字を取得します。

インクルード： <stdio.h>

プロトタイプ： int fgetc(FILE *stream);

引数： *stream* オープンされたストリームを指すポインタ

戻り値： 読み出した文字を返します。読み出しエラーが発生した場合、または end-of-file に到達した場合、EOF を返します。

備考： この関数は、入力ストリームから次の文字を読み出し、ファイル位置インジケータを進め、int へ変換された unsigned char として文字を返します。

例：

```
#include <stdio.h> /* for fgetc, printf, */
                   /* fclose, FILE,      */
                   /* NULL, EOF          */

int main(void)
{
  FILE *buf;
  char y;

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt¥n");
  else
  {
    y = fgetc(buf);
    while (y != EOF)
    {
      printf("%c|", y);
      y = fgetc(buf);
    }
    fclose(buf);
  }
}
```

入力：

afile.txt の内容 (入力として使用):

```
Short
Longer string
```

出力：

```
S|h|o|r|t|
|L|o|n|g|e|r| |s|t|r|i|n|g|
|
```

fgetpos

説明： ストリームのファイル位置を取得。

インクルード： <stdio.h>

プロトタイプ： int fgetpos(FILE *stream, fpos_t *pos);

引数： *stream* ターゲット・ストリーム

pos 位置インジケータ・ストレージ

戻り値： 正常終了の場合 0 を返します。その他の場合は非ゼロ値を返します。

備考： この関数は、正常終了の場合与えられたストリームのファイル位置インジケータを *pos に格納します。その他の場合は、errno を設定します。

例：

```
/* This program opens a file and reads bytes at */
/* several different locations.The fgetpos      */
/* function notes the 8th byte.21 bytes are     */
/* read then 18 bytes are read.Next the         */
/* fsetpos function is set based on the         */
/* fgetpos position and the previous 21 bytes   */
/* are reread.                                  */

#include <stdio.h> /* for fgetpos, fread,    */
                   /* printf, fopen, fclose, */
                   /* FILE, NULL, perror,    */
                   /* fpos_t, sizeof         */

int main(void)
{
  FILE   *myfile;
  fpos_t pos;
  char   buf[25];

 if ((myfile = fopen("sampfgetpos.c", "rb")) ==
       NULL)
    printf("Cannot open file¥n");
  else
  {
   fread(buf, sizeof(char), 8, myfile);
    if (fgetpos(myfile, &pos) != 0)
      perror("fgetpos error");
    else
    {
      fread(buf, sizeof(char), 21, myfile);
      printf("Bytes read: %.21s¥n", buf);
      fread(buf, sizeof(char), 18, myfile);
      printf("Bytes read: %.18s¥n", buf);
    }

 if (fsetpos(myfile, &pos) != 0)
    perror("fsetpos error");

  fread(buf, sizeof(char), 21, myfile);
  printf("Bytes read: %.21s¥n", buf);
  fclose(myfile);
  }
}
```

fgetpos (続き)

出力：

```
Bytes read: program opens a file
Bytes read: and reads bytes at
Bytes read: program opens a file
```

fgets

説明： ストリームからストリングを取得します。

インクルード： <stdio.h>

プロトタイプ： char *fgets(char *s, int n, FILE *stream);

引数：

s	ストレージ列を指すポインタ
n	読み出す最大文字数
stream	オープンされたストリームを指すポインタ

戻り値： 正常終了の場合、ストリングを指すポインタを返します。その他の場合は null ポインタを返します。

備考： この関数は、入力ストリームから n-1 個の文字を読み出し、s によって指定されるストリングへ格納して、ニューライン文字を格納するか、あるいは end-of-file またはエラー・インジケータを設定します。文字を格納した場合、配列の次のエレメント内で最後に読み出した文字の直後に null 文字を格納します。fgets がエラー・インジケータを設定する場合、配列の内容は不定になります。

例：

```
#include <stdio.h> /* for fgets, printf, */
                   /* fopen, fclose,     */
                   /* FILE, NULL         */
#define MAX 50

int main(void)
{
  FILE *buf;
  char s[MAX];

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt¥n");
  else
  {
    while (fgets(s, MAX, buf) != NULL)
    {
      printf("%s|", s);
    }
    fclose(buf);
  }
}
```

入力：

afile.txt の内容 (入力として使用):

```
Short
Longer string
```

出力：

```
Short
|Longer string
|
```

fopen

説明： ファイルを開きます。

インクルード： <stdio.h>

プロトタイプ： FILE *fopen(const char *filename, const char *mode);

引数：

filename	ファイル名
mode	許容されるアクセス・タイプ

戻り値： オープンされたストリームを指すポインタを返します。関数が異常終了の場合は、null ポインタを返します。

備考： ファイル・アクセスのタイプを次に示します：

r –	既存テキスト・ファイルを読み出し用に開きます。
w –	空白のテキスト・ファイルを書き込み用に開きます (既存ファイルは上書きされます)。
a –	テキスト・ファイルをアペンド用に開きます (ファイルが存在しない場合は生成します)。
rb –	既存バイナリ・ファイルを読み出し用に開きます。
wb –	空白のバイナリ・ファイルを書き込み用に開きます (既存ファイルは上書きされます)。
ab –	バイナリ・ファイルをアペンド用に開きます (ファイルが存在しない場合は生成します)。
r+ –	既存テキスト・ファイルを読み出し / 書き込み用に開きます。
r+ –	空白のテキスト・ファイルを読み出し / 書き込み用に開きます (既存ファイルは上書きされます)。
a+ –	テキスト・ファイルを読み出し / アペンド用に開きます (ファイルが存在しない場合は生成します)。
r+b または rb+ –	既存バイナリ・ファイルを読み出し / 書き込み用に開きます。
w+b または wb+ –	空白バイナリ・ファイルを読み出し / 書き込み用に開きます (既存ファイルは上書きされます)。
a+b または ab+ –	バイナリ・ファイルを読み出し / アペンド用に開きます (ファイルが存在しない場合は生成します)。

例：

```
#include <stdio.h> /* for fopen, fclose, */
                   /* printf, FILE,      */
                   /* NULL, EOF          */

int main(void)
{
  FILE *myfile1, *myfile2;
  int y;
```

fopen (続き)

```
  if ((myfile1 = fopen("afile1", "r")) == NULL)
    printf("Cannot open afile1¥n");
  else
  {
    printf("afile1 was opened¥n");
    y = fclose(myfile1);
    if (y == EOF)
      printf("afile1 was not closed¥n");
    else
      printf("afile1 was closed¥n");
  }

  if ((myfile1 = fopen("afile1", "w+")) == NULL)
    printf("Second try, cannot open afile1¥n");
  else
  {
    printf("Second try, afile1 was opened¥n");
    y = fclose(myfile1);
    if (y == EOF)
      printf("afile1 was not closed¥n");
    else
      printf("afile1 was closed¥n");
  }

  if ((myfile2 = fopen("afile2", "w+")) == NULL)
    printf("Cannot open afile2¥n");
  else
  {
    printf("afile2 was opened¥n");
    y = fclose(myfile2);
    if (y == EOF)
      printf("afile2 was not closed¥n");
   else
      printf("afile2 was closed¥n");
  }

}
```

出力：

```
Cannot open afile1
Second try, afile1 was opened
afile1 was closed
afile2 was opened
afile2 was closed
```

説明：

読み出し用 (r) に開く前に afile1 が存在している必要があります。そうしないと、fopen 関数が異常終了します。fopen 関数がファイルを書き込み用 (w+) に開く場合、すでに存在している必要はありません。存在しない場合は、生成して開きます。

fprintf

説明： フォーマット化されたデータをストリームへプリントします。

インクルード： <stdio.h>

プロトタイプ： int fprintf(FILE *_stream_, const char *_format_, ...);

引数：

stream	データの出力先ストリームを指すポインタ
format	フォーマット制御文字列
...	オプションの引数

戻り値： 生成した文字数を返します。エラーが発生した場合は負の数値を返します。

備考： フォーマット引数は同じ構文を持ち、print 内で持っているものを使います。

例：

```
#include <stdio.h> /* for fopen, fclose, */
                   /* fprintf, printf,   */
                   /* FILE, NULL         */

int main(void)
{
  FILE *myfile;
  int y;
  char s[]="Print this string";
  int x = 1;
  char a = '¥n';

  if ((myfile = fopen("afile", "w")) == NULL)
    printf("Cannot open afile¥n");
  else
  {
    y = fprintf(myfile, "%s %d time%c", s, x, a);

    printf("Number of characters printed "
           "to file = %d",y);

    fclose(myfile);
  }
}
```

出力：

```
Number of characters printed to file = 25
```

afile の内容：

```
Print this string 1 time
```

fputc

説明： 文字をストリームへ出力します。

インクルード： <stdio.h>

プロトタイプ： int fputc(int c, FILE *stream);

引数： *c* 書き込まれる文字

stream オープンされたストリームを指すポインタ

戻り値： 書き込まれた文字を返します。書き込みエラーの場合は EOF を返します。

備考： この関数は、出力ストリームへ文字を書き込み、ファイル位置インジケータを進め、int へ変換された unsigned char として文字を返します。

例：

```
#include <stdio.h> /* for fputc, EOF, stdout */

int main(void)
{
  char *y;
  char buf[] = "This is text¥n";
  int x;

  x = 0;

  for (y = buf; (x != EOF) && (*y != '¥0'); y++)
  {
    x = fputc(*y, stdout);
    fputc('|', stdout);
  }
}
```

出力：

```
T|h|i|s| |i|s| |t|e|x|t|
|
```

fputs

説明： 文字列をストリームへ出力します。

インクルード： <stdio.h>

プロトタイプ： int fputs(const char *s, FILE *stream);

引数： *s* 書き込まれる文字列

stream オープンされたストリームを指すポインタ

戻り値： 正常終了の場合非負値を返します。その他の場合は EOF を返します。

備考： この関数は null 文字に遭遇するまで文字 (null を除く) を出力ストリームに書き込みます。

例：

```
#include <stdio.h> /* for fputs, stdout */

int main(void)
{
  char buf[] = "This is text¥n";

  fputs(buf,stdout);
  fputs("|",stdout);
}
```

出力：

```
This is text
|
```

fread

説明： ストリームからデータを読み出します。

インクルード： <stdio.h>

プロトタイプ： size_t fread(void *ptr, size_t size, size_t nelem, FILE *stream);

引数：

ptr	ストレージ・バッファを指すポインタ
size	項目のサイズ
nelem	読み出す最大項目数
stream	ストリームを指すポインタ

戻り値： *size* で指定されたサイズを持つ *nelem* まで読み込んだ完全なエレメントの数を返します。

備考： この関数は、与えられたストリームから ptr で指定されたバッファへ *size* * *nelem* 個の文字を読み込みます。あるいは、end-of-file またはエラー・インジケータを設定します。fread は *n*/*size* を返します。ここで、*n* は読み込んだ文字数です。n がサイズの倍数でない場合、最終エレメントの値は不定になります。この関数がエラー・インジケータを設定する場合は、ファイル位置インジケータは不定になります。

例：

```
#include <stdio.h> /* for fread, fwrite,     */
                   /* printf, fopen, fclose, */
                   /* sizeof, FILE, NULL     */

int main(void)
{
  FILE *buf;
  int x, numwrote, numread;
  double nums[10], readnums[10];

  if ((buf = fopen("afile.out", "w+")) != NULL)
  {
    for (x = 0; x < 10; x++)
    {
      nums[x] = 10.0/(x + 1);
      printf("10.0/%d = %f¥n", x+1, nums[x]);
    }

    numwrote = fwrite(nums, sizeof(double),
                      10, buf);
    printf("Wrote %d numbers¥n¥n", numwrote);
    fclose(buf);

  }
   else
    printf("Cannot open afile.out¥n");
```

fread (続き)

```
  if ((buf = fopen("afile.out", "r+")) != NULL)
  {
    numread = fread(readnums, sizeof(double),
                    10, buf);
    printf("Read %d numbers¥n", numread);
    for (x = 0; x < 10; x++)
    {
      printf("%d * %f = %f¥n", x+1, readnums[x],
             (x + 1) * readnums[x]);
    }
    fclose(buf);
  }
  else
    printf("Cannot open afile.out¥n");
}
```

出力：

```
10.0/1 = 10.000000
10.0/2 = 5.000000
10.0/3 = 3.333333
10.0/4 = 2.500000
10.0/5 = 2.000000
10.0/6 = 1.666667
10.0/7 = 1.428571
10.0/8 = 1.250000
10.0/9 = 1.111111
10.0/10 = 1.000000
Wrote 10 numbers

Read 10 numbers
1 * 10.000000 = 10.000000
2 * 5.000000 = 10.000000
3 * 3.333333 = 10.000000
4 * 2.500000 = 10.000000
5 * 2.000000 = 10.000000
6 * 1.666667 = 10.000000
7 * 1.428571 = 10.000000
8 * 1.250000 = 10.000000
9 * 1.111111 = 10.000000
10 * 1.000000 = 10.000000
```

説明：

このプログラムは、fwrite を使って 10 個の数値をバイナリ形式でファイルに保存します。この機能を使うと、プログラムが使っているビット・パターンと同じパターンで数値を保存することができるため、精確さと一貫性を向上させることができます。fprintf を使うと、数値を文字列として保存することができるため、数値が切り詰められる原因になります。各数値は 10 に分割され、さまざまな数値になります。fread を使って数値を新しい配列へ取り込み、元の数値を乗算すると、保存プロセスで数値が切り詰められなかったことが示されます。

freopen

説明： 既存ストリームを新しいファイルへ再割り当てします。

インクルード： <stdio.h>

プロトタイプ： FILE *freopen(const char *filename, const char *mode, FILE *stream);

引数：

filename	新しいファイル名
mode	許容されるアクセス・タイプ
stream	現在オープン中のストリームを指すポインタ

戻り値： 新しくオープンされたファイルを指すポインタを返します。関数が異常終了の場合は、null ポインタを返します。

備考： この関数は、fclose がコールされたかのように、ストリームに対応付けられたファイルを閉じます。次に、fopen がコールされたかのように新しいファイルを開きます。指定されたストリームが開いていない場合、freopen は異常終了します。ファイル・アクセスの許容されるタイプについては、fopen を参照してください。

例：

```
#include <stdio.h> /* for fopen, freopen, */
                   /* printf, fclose,     */
                   /* FILE, NULL          */

int main(void)
{
  FILE *myfile1, *myfile2;
  int y;

  if ((myfile1 = fopen("afile1", "w+")) == NULL)
    printf("Cannot open afile1¥n");
  else
  {
    printf("afile1 was opened¥n");

    if ((myfile2 = freopen("afile2", "w+",
         myfile1)) == NULL)
    {
      printf("Cannot open afile2¥n");
      fclose(myfile1);
    }
    else
    {
      printf("afile2 was opened¥n");
      fclose(myfile2);
    }
  }
}
```

出力：

```
afile1 was opened
afile2 was opened
```

説明：
このプログラムは、freopen がコールされたとき、myfile2 を使ってストリームを指定しています。したがって、エラーが発生した場合、myfile1 はストリームを指したままであり、正しく閉じることができます。freopen のコールが成功した場合、myfile2 を使ってストリームを正しく閉じることができます。

fscanf

説明： ストリームからのフォーマット化されたテキストをスキャンします。

インクルード： <stdio.h>

プロトタイプ： int fscanf(FILE *_stream_, const char *_format_, ...);

引数： _stream_ データの読み出し元となるオープンされたストリームを指すポインタ

format フォーマット制御文字列

... オプションの引数

戻り値： 正常に変換され、割り当てられた項目数を返します。割り当てられた項目がない場合、0 を返します。最初の変換の前に end-of-file に遭遇した場合、またはエラーが発生した場合、EOF を返します。

備考： フォーマット引数は同じ構文を持ち、scanf 内で持っているものを使います。

例：

```
#include <stdio.h> /* for fopen, fscanf,   */
                   /* fclose, fprintf,     */
                   /* fseek, printf, FILE, */
                   /* NULL, SEEK_SET       */

int main(void)
{
  FILE *myfile;
  char s[30];
  int x;
  char a;

  if ((myfile = fopen("afile", "w+")) == NULL)
    printf("Cannot open afile¥n");
  else
  {
    fprintf(myfile, "%s %d times%c",
            "Print this string", 100, '¥n');

    fseek(myfile, 0L, SEEK_SET);

    fscanf(myfile, "%s", s);
    printf("%s¥n", s);
    fscanf(myfile, "%s", s);
    printf("%s¥n", s);
    fscanf(myfile, "%s", s);
    printf("%s¥n", s);
    fscanf(myfile, "%d", &x);
    printf("%d¥n", x);
    fscanf(myfile, "%s", s);
    printf("%s¥n", s);
    fscanf(myfile, "%c", a);
    printf("%c¥n", a);

    fclose(myfile);
  }
}
```

入力：

afile の内容：
Print this string 100 times

fscanf (続き)

出力：

```
Print
this
string
100
times
```

fseek

説明： ファイル・ポインタを特定のロケーションへ移動します。

インクルード： <stdio.h>

プロトタイプ： int fseek(FILE *_stream_, long _offset_, int _mode_);

引数：

stream ファイル・ポインタを移動する相手先ストリーム

offset 現在の位置に加算する値

mode 実行するシークのタイプ

戻り値： 正常終了の場合 0 を返します。その他の場合は非ゼロ値を返し、errno を設定します。

備考： モードとしては次が可能です：

SEEK_SET ―ファイルの先頭からシーク

SEEK_CUR ― ファイル・ポインタの現在位置からシーク

SEEK_END ―ファイルの終わりからシーク

例：

```
#include <stdio.h> /* for fseek, fgets,      */
                   /* printf, fopen, fclose, */
                   /* FILE, NULL, perror,    */
                   /* SEEK_SET, SEEK_CUR,    */
                   /* SEEK_END               */

int main(void)
{
  FILE *myfile;
  char s[70];
  int  y;

  myfile = fopen("afile.out", "w+");
  if (myfile == NULL)
    printf("Cannot open afile.out¥n");
  else
  {
    fprintf(myfile, "This is the beginning, "
                    "this is the middle and "
                    "this is the end.");

    y = fseek(myfile, 0L, SEEK_SET);
    if (y)
      perror("Fseek failed");
    else
    {
      fgets(s, 22, myfile);
      printf("¥"%s¥"¥n¥n", s);
    }
```

fseek (続き)

```
    y = fseek(myfile, 2L, SEEK_CUR);
    if (y)
      perror("Fseek failed");
    else
    {
      fgets(s, 70, myfile);
      printf("¥"%s¥"¥n¥n", s);
    }

    y = fseek(myfile, -16L, SEEK_END);
    if (y)
      perror("Fseek failed");
    else
    {
      fgets(s, 70, myfile);
      printf("¥"%s¥"¥n", s);
    }
    fclose(myfile);
  }
}
```

出力：

```
"This is the beginning"

"this is the middle and this is the end."

"this is the end."
```

説明：

テキスト "This is the beginning, this is the middle and this is the end" を含むファイル afile.out が生成されます。関数 fseek は、オフセット・ゼロと SEEK_SET を使って、ファイル・ポインタをファイルの先頭に設定します。次に fgets が "This is the beginning" の 22 文字を読み込み、この文字列に null 文字を追加します。

さらに、fseek はオフセット 2 と SEEK_CURRENT を使って、ファイル・ポインタを現在位置 + 2 に設定します (カンマとスペースはスキップ)。次に、fgets が次の 70 文字を読み込みます。最初の 39 文字は、"this is the middle and this is the end" です。EOF を読み込むと停止して、文字列に null 文字を追加します。

最後に、fseek は負の 16 文字のオフセットと SEEK_END を使って、ファイル・ポインタをファイルの終わりから 16 文字の所へ設定し、fgets が 70 文字まで読み込みます。"this is the end" の 16 文字を読み込んだ後に EOF で停止し、文字列に null 文字を追加します。

fsetpos

説明： ストリームのファイル位置を設定します。

インクルード： <stdio.h>

プロトタイプ： int fsetpos(FILE *stream, const fpos_t *pos);

引数：

stream	ターゲット・ストリーム
pos	fgetpos に対する前のコールで返された位置インジケータ・ストレージ

戻り値： 正常終了の場合 0 を返します。その他の場合は非ゼロ値を返します。

備考： この関数は、正常終了の場合与えられたストリームのファイル位置インジケータを *pos に設定します。その他の場合は、errno を設定します。

fsetpos (続き)

例：

```
/* This program opens a file and reads bytes at */
/* several different locations.The fgetpos      */
/* function notes the 8th byte.21 bytes are     */
/* read then 18 bytes are read.Next the         /
/* fsetpos function is set based on the         */
/* fgetpos position and the previous 21 bytes   */
/* are reread.                                  */

#include <stdio.h> /* for fgetpos, fread,    */
                   /* printf, fopen, fclose, */
                   /* FILE, NULL, perror,    */
                   /* fpos_t, sizeof         */

int main(void)
{
  FILE   *myfile;
  fpos_t pos;
  char   buf[25];

  if ((myfile = fopen("sampfgetpos.c", "rb")) ==
                        NULL)
    printf("Cannot open file¥n");
  else
  {
    fread(buf, sizeof(char), 8, myfile);
    if (fgetpos(myfile, &pos) != 0)
      perror("fgetpos error");
    else
    {
      fread(buf, sizeof(char), 21, myfile);
      printf("Bytes read: %.21s¥n", buf);
      fread(buf, sizeof(char), 18, myfile);
      printf("Bytes read: %.18s¥n", buf);
    }

  if (fsetpos(myfile, &pos) != 0)
    perror("fsetpos error");

  fread(buf, sizeof(char), 21, myfile);
  printf("Bytes read: %.21s¥n", buf);
  fclose(myfile);
  }
}
```

出力：

```
Bytes read: program opens a file
Bytes read: and reads bytes at
Bytes read: program opens a file
```

ftell

説明： ファイル・ポインタの現在位置を取得します。

インクルード： <stdio.h>

プロトタイプ： long ftell(FILE *stream);

引数： *stream* 現在のファイル位置を取得する相手先ストリーム

戻り値： 正常終了の場合ファイル・ポインタの位置を返します。その他の場合は -1 を返します。

例：

```
#include <stdio.h> /* for ftell, fread,      */
                   /* fprintf, printf,       */
                   /* fopen, fclose, sizeof, */
                   /* FILE, NULL */

int main(void)
{
  FILE *myfile;
  char s[75];
  long y;

  myfile = fopen("afile.out", "w+");
  if (myfile == NULL)
    printf("Cannot open afile.out¥n");
  else
  {
    fprintf(myfile,"This is a very long sentence "
                    "for input into the file named "
                  "afile.out for testing.");

    fclose(myfile);

    if ((myfile = fopen("afile.out", "rb")) != NULL)
    {
      printf("Read some characters:¥n");
      fread(s, sizeof(char), 29, myfile);
      printf("¥t¥"%s¥"¥n", s);

      y = ftell(myfile);
      printf("The current position of the "
             "file pointer is %ld¥n", y);
      fclose(myfile);
    }
  }
}
```

出力：

```
Read some characters:
    "This is a very long sentence "
The current position of the file pointer is 29
```

fwrite

説明： データをストリームへ書き込みます。

インクルード： <stdio.h>

プロトタイプ： size_t fwrite(const void *ptr, size_t size, size_t nelem, FILE *stream);

引数：

ptr	ストレージ・バッファを指すポインタ
size	項目のサイズ
nelem	読み出す最大項目数
stream	オープンされたストリームを指すポインタ

戻り値： 正常に書き込まれた完全なエレメント数を返します。書き込みエラーが発生した場合、*nelem* より小さい値を返します。

備考： この関数は、*ptr* で指定されたバッファから与えられたストリームへ文字を *nelem* 個のエレメントまで書き込みます。エレメントのサイズは *size* で指定されます。ファイル位置インジケータは、正常に書き込まれた文字数だけ進められます。この関数がエラー・インジケータを設定する場合は、ファイル位置インジケータは不定になります。

例：

```
#include <stdio.h> /* for fread, fwrite,     */
                   /* printf, fopen, fclose, */
                   /* sizeof, FILE, NULL     */

int main(void)
{
  FILE *buf;
  int x, numwrote, numread;
  double nums[10], readnums[10];

  if ((buf = fopen("afile.out", "w+")) != NULL)
  {
    for (x = 0; x < 10; x++)
    {
      nums[x] = 10.0/(x + 1);
      printf("10.0/%d = %f¥n", x+1, nums[x]);
    }

    numwrote = fwrite(nums, sizeof(double),
                      10, buf);
    printf("Wrote %d numbers¥n¥n", numwrote);
    fclose(buf);

  }
  else
    printf("Cannot open afile.out¥n");
```

fwrite (続き)

```
  if ((buf = fopen("afile.out", "r+")) != NULL)
  {
    numread = fread(readnums, sizeof(double),
                    10, buf);
    printf("Read %d numbers¥n", numread);
    for (x = 0; x < 10; x++)
    {
      printf("%d * %f = %f¥n", x+1, readnums[x],
             (x + 1) * readnums[x]);
    }
    fclose(buf);
  }
  else
    printf("Cannot open afile.out¥n");
}
```

出力：

```
10.0/1 = 10.000000
10.0/2 = 5.000000
10.0/3 = 3.333333
10.0/4 = 2.500000
10.0/5 = 2.000000
10.0/6 = 1.666667
10.0/7 = 1.428571
10.0/8 = 1.250000
10.0/9 = 1.111111
10.0/10 = 1.000000
Wrote 10 numbers

Read 10 numbers
1 * 10.000000 = 10.000000
2 * 5.000000 = 10.000000
3 * 3.333333 = 10.000000
4 * 2.500000 = 10.000000
5 * 2.000000 = 10.000000
6 * 1.666667 = 10.000000
7 * 1.428571 = 10.000000
8 * 1.250000 = 10.000000
9 * 1.111111 = 10.000000
10 * 1.000000 = 10.000000
```

説明：

このプログラムは、fwrite を使って 10 個の数値をバイナリ形式でファイルに保存します。この機能を使うと、プログラムが使っているビット・パターンと同じパターンで数値を保存することができるため、精確さと一貫性を向上させることができます。fprintf を使うと、数値を文字列として保存することができるため、数値が切り詰められる原因になります。各数値は 10 に分割され、さまざまな数値になります。fread を使って数値を新しい配列へ取り込み、元の数値を乗算すると、保存プロセスで数値が切り詰められなかったことが示されます。

getc

説明： ストリームから文字を取得します。

インクルード： <stdio.h>

プロトタイプ： int getc(FILE *stream);

引数： *stream* オープンされたストリームを指すポインタ

戻り値： 読み出した文字を返します。読み出しエラーが発生した場合、または end-of-file に到達した場合、EOF を返します。

備考： getc は、関数 fgetc と同じです。

例：

```
#include <stdio.h> /* for getc, printf, */
                   /* fopen, fclose,    */
                   /* FILE, NULL, EOF   */

int main(void)
{
  FILE *buf;
  char y;

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt¥n");
  else
  {
    y = getc(buf);
    while (y != EOF)
    {
      printf("%c|", y);
      y = getc(buf);
    }
    fclose(buf);
  }
}
```

入力：

afile.txt の内容（入力として使用）:

```
Short
Longer string
```

出力：

```
S|h|o|r|t|
|L|o|n|g|e|r| |s|t|r|i|n|g|
|
```

getchar

説明： stdin から文字を取得します。

インクルード： <stdio.h>

プロトタイプ： int getchar(void);

戻り値： 読み出した文字を返します。読み出しエラーが発生した場合、または end-of-file に到達した場合、EOF を返します。

備考： 引数 stdin を使う fgetc と同じ機能。

例：

```
#include <stdio.h> /* for getchar, printf */

int main(void)
{
  char y;

    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
}
```

入力：

UartIn.txt の内容 (シミュレータ用の stdin 入力として使用) :

```
Short
Longer string
```

出力：

```
S|h|o|r|t|
```

gets

説明： stdin から文字列を取得します。

インクルード： <stdio.h>

プロトタイプ： char *gets(char *s);

引数： *s* ストレージ列を指すポインタ

戻り値： 正常終了の場合、ストリングを指すポインタを返します。その他の場合は null ポインタを返します。

備考： この関数は、ストリーム stdin から文字をニューライン文字 (これは格納しません) まで読み出し、*s* によって指定されるストリングへ格納するか、あるいは end-of-file またはエラー・インジケータを設定します。文字を読み出した場合、配列の次のエレメント内で最後に読み出した文字の直後に null 文字を格納します。gets がエラー・インジケータを設定する場合、配列の内容は不定になります。

gets (続き)

例：

```
#include <stdio.h> /* for gets, printf */

int main(void)
{
  char y[50];

  gets(y) ;
  printf("Text: %s¥n", y);
}
```

入力：

UartIn.txt の内容 (シミュレータ用の stdin 入力として使用):
Short
Longer string

出力：

Text: Short

perror

説明： エラー・メッセージを stderr へプリントします。

インクルード： <stdio.h>

プロトタイプ： void perror(const char *s);

引数： *s* プリントする文字列

戻り値： なし

備考： 文字列、コロン、スペースの順でプリントされます。次に errno に基づくエラー・メッセージがプリントされ、後ろにニューラインが続きます。

例：

```
#include <stdio.h> /* for perror, fopen, */
                   /* fclose, printf,    */
                   /* FILE, NULL         */

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("samp.fil", "r+")) == NULL)
    perror("Cannot open samp.fil");
  else
    printf("Success opening samp.fil¥n");

  fclose(myfile);
}
```

出力：

Cannot open samp.fil: file open error

printf

説明： フォーマット化されたテキストを stdout へプリントします。

インクルード： <stdio.h>

プロトタイプ： int printf(const char *format, ...);

引数：

format フォーマット制御文字列

... オプションの引数

戻り値： 生成した文字数を返します。エラーが発生した場合は負の数値を返します。

備考： フォーマット指定数と引数数が一致する必要があります。フォーマット指定数より引数数が少ないと、出力は不定になります。引数数がフォーマット指定数より多いと、残りの引数は無視されます。各フォーマット指定子はパーセント記号で始まり、その後にオプション・フィールドと必須のタイプが次のように続く必要があります：

```
%[flags][width][.precision][size]type
```

flags

-	与えられたフィールド幅内で値を左詰め
0	スペース (デフォルト) の代わりに 0 をパッド文字として使用
+	正の符号付き値に対してプラス記号を生成
space	スペースまたはプラス記号もマイナス記号も持たない符号付き値を生成
#	8 進変換ではプレフィックス 0 を、16 進数変換ではプレフィックス 0x または 0X をそれぞれ生成し、浮動小数変換では、小数点位置と小数桁 (他の場合はこれらを抑制) を生成します。

width

変換に対して生成する文字数を指定します。10 進値の代わりにアスタリスク (*) を使用する場合、次の引数 (int 型である必要があります) は、フィールド幅として使用されます。結果がフィールド幅より狭い場合は、フィールドを埋めるために左側にパッド文字が使用されます。結果がフィールド幅より大きい場合は、パディングなしで値を収容できるようにフィールドが拡張されます。

precision

フィールド幅の後ろにドット (.) と次のいずれかを指定する精度を表わす 10 進整数を続けることができます：

- 整数変換で生成する最小桁数
- e 変換、E 変換、f 変換で生成する小数部の桁数
- g 変換、G 変換で生成する整数部の最大桁数
- s 変換で C 文字列から生成する最大文字数

整数部なしでピリオドが現れると、この整数はゼロと見なされます。10 進値の代わりにアスタリスク (*) を使用する場合、次の引数 (int 型である必要があります) は、精度として使用されます。

printf (続き)

size

- h modifier — 型 d、i、o、u、x、X と組み合わせて使用され、値を short int または unsigned short int へ変換します。
- h modifier — n と組み合わせて使用され、ポインタが short int を指すことを指定します。
- l modifier — 型 d、i、o、u、x、X と組み合わせて使用され、値を long int または unsigned long int へ変換します。
- l modifier — n と組み合わせて使用され、ポインタが long int を指すことを指定します。
- l modifier — c と組み合わせて使用され、ワイド文字を指定します。
- l modifier — 型 e、E、f、F、g、G と組み合わせて使用され、値を double に変換します。
- ll modifier — 型 d、i、o、u、x、X と組み合わせて使用され、値を long long int または unsigned long long int に変換します。
- ll modifier — n と組み合わせて使用され、ポインタが long long int を指すことを指定します。
- L modifier — e、E、f、g、G と組み合わせて使用され、値を long double に変換します。

type

d、i	signed int
o	8 進の unsigned int
u	10 進の unsigned int
x	小文字 16 進数の unsigned int
X	大文字 16 進の unsigned int
e、E	科学表記法の double
f	10 進表記法の double
g、G	double (e、E、f の適切なものからの形式を使用)
c	char - 1 文字
s	string
p	ポインタ値
n	対応する引数は、書き込まれる文字数などが置かれる整数ポインタである必要があります。文字はプリントされません。
%	% 文字がプリントされます。

例 :

```
#include <stdio.h> /* for printf */

int main(void)
{
  /* print a character right justified in a 3  */
  /* character space.                          */
  printf("%3c¥n", 'a');

  /* print an integer, left justified (as      */
  /* specified by the minus sign in the format */
  /* string) in a 4 character space.Print a    */
  /* second integer that is right justified in  */
  /* a 4 character space using the pipe (|) as  */
  /* a separator between the integers.          */
  printf("%-4d|%4d¥n", -4, 4);
```

printf (続き)

```
  /* print a number converted to octal in 4    */
  /* digits.                                   */
  printf("%.4o¥n", 10);

  /* print a number converted to hexadecimal   */
  /* format with a 0x prefix.                  */
  printf("%#x¥n", 28);

  /* print a float in scientific notation      */
  printf("%E¥n", 1.1e20);

  /* print a float with 2 fraction digits      */
  printf("%.2f¥n", -3.346);

  /* print a long float with %E, %e, or %f     */
  /* whichever is the shortest version         */
  printf("%Lg¥n", .02L);
}
```

出力：

```
 a
-4 | 4
0012
0x1c
1.100000E+20
-3.35
0.02
```

putc

説明： 文字をストリームへ出力します。

インクルード： <stdio.h>

プロトタイプ： int putc(int c, FILE *stream);

引数：

c 書き込まれる文字

stream FILE 構造体を指すポインタ

戻り値： 文字を返します。エラーが発生した場合、または end-of-file に到達した場合、EOF を返します。

備考： putc は関数 fputc と同じです。

例：

```
#include <stdio.h> /* for putc, EOF, stdout */

int main(void)
{

  char *y;
  char buf[] = "This is text¥n";
  int x;

  x = 0;

for (y = buf; (x != EOF) && (*y != '¥0'); y++)
  {
    x = putc(*y, stdout);
    putc('|', stdout);
  }
}
```

putc (続き)

出力：

```
T|h|i|s| |i|s| |t|e|x|t|
|
```

putchar

説明： 文字を stdout へ出力します。

インクルード： <stdio.h>

プロトタイプ： int putchar(int *c*);

引数： *c* 書き込まれる文字

戻り値： 文字を返します。エラーが発生した場合、または end-of-file に到達した場合、EOF を返します。

備考： stdout を引数とした fputc と同じ機能です。

例：

```
#include <stdio.h> /* for putchar, printf, */
                   /* EOF, stdout          */

int main(void)
{

  char *y;
  char buf[] = "This is text¥n";
  int x;

  x = 0;


  for (y = buf; (x != EOF) && (*y != '¥0'); y++)
    x = putchar(*y);
}
```

出力：

```
This is text
```

puts

説明： 文字列を stdout へ出力します。

インクルード： <stdio.h>

プロトタイプ： int puts(const char *s);

引数： *s* 書き込まれる文字列

戻り値： 正常終了の場合非負値を返します。その他の場合は EOF を返します。

備考： この関数は、文字をストリーム stdout へ書き込みます。ニューライン文字がアペンドされます。終了させる null 文字はストリームへ書き込まれません。

例：

```
#include <stdio.h> /* for puts */

int main(void)
{
  char buf[] = "This is text¥n";

  puts(buf);
  puts("|");
}
```

puts

出力：

```
This is text

|
```

remove

説明： 指定されたファイルを削除します。

インクルード： <stdio.h>

プロトタイプ：

```
int remove(const char *filename);
```

引数： *filename* 削除するファイルの名前

戻り値： 正常終了の場合 0 を返します。その他の場合 -1 を返します。

備考： ファイル名が存在しない、または開いている場合は、削除できません。

例：

```
#include <stdio.h> /* for remove, printf */

int main(void)
{
  if (remove("myfile.txt") != 0)
    printf("Cannot remove file");
  else
    printf("File removed");
}
```

出力：

```
File removed
```

rename

説明： 指定されたファイルの名前を変更します。

インクルード： <stdio.h>

プロトタイプ：

```
int rename(const char *old, const char *new);
```

引数： *old* 古い名前を指すポインタ

new 新しい名前を指すポインタ

戻り値： 正常終了の場合 0 を返します。その他の場合非ゼロを返します。

備考： 新しい名前はすでに現在のワーキング・ディレクトリ内に存在していない必要があり、古い名前は現在のワーキング・ディレクトリ内に存在している必要があります。

例：

```
#include <stdio.h> /* for rename, printf */

int main(void)
{
  if (rename("myfile.txt","newfile.txt") != 0)
    printf("Cannot rename file");
  else
    printf("File renamed");
}
```

出力：

```
File renamed
```

rewind

説明： ファイル・ポインタをファイル先頭にリセットします。

インクルード： <stdio.h>

プロトタイプ： void rewind(FILE *stream);

引数： *stream* ファイル・ポインタをリセットするストリーム

備考： この関数は、fseek(*stream*, 0L, SEEK_SET) をコールして、与えられたストリームのエラー・インジケータをクリアします。

例：

```
#include <stdio.h> /* for rewind, fopen, */
                   /* fscanf, fclose,    */
                   /* fprintf, printf,   */
                   /* FILE, NULL         */

int main(void)
{
  FILE *myfile;
  char s[] = "cookies";
  int x = 10;

  if ((myfile = fopen("afile", "w+")) == NULL)
    printf("Cannot open afile¥n");
  else
  {
    fprintf(myfile, "%d %s", x, s);
    printf("I have %d %s.¥n", x, s);

    /* set pointer to beginning of file */
    rewind(myfile);
    fscanf(myfile, "%d %s", &x, &s);
    printf("I ate %d %s.¥n", x, s);

    fclose(myfile);
  }
}
```

出力：

```
I have 10 cookies.
I ate 10 cookies.
```

scanf

説明： stdin からのフォーマット化されたテキストをスキャンします。

インクルード： <stdio.h>

プロトタイプ： int scanf(const char *format, ...);

引数： *format* フォーマット制御文字列

... オプションの引数

戻り値： 正常に変換され、割り当てられた項目数を返します。割り当てられた項目がない場合、0 を返します。最初の変換の前に入力エラーに遭遇した場合、EOF を返します。

備考： 各フォーマット指定子はパーセント記号で始まり、その後にオプション・フィールドと必須のタイプが次のように続く必要があります：

%[*][width][modifier]type

*****

割り当ての抑制を表示します。これにより、入力フィールドがスキップされて、割り当てが行われません。

width

変換に一致する最大入力文字数を指定します。ただし、スキップできる空白スペースを含みません。

modifier

h modifier — 型 d、i、o、u、x、X と組み合わせて使用され、値を short int または unsigned short int へ変換します。

h modifier — n と組み合わせて使用され、ポインタが short int を指すことを指定します。

l modifier — 型 d、i、o、u、x、X と組み合わせて使用され、値を long int または unsigned long int へ変換します。

l modifier — n と組み合わせて使用され、ポインタが long int を指すことを指定します。

l modifier — c と組み合わせて使用され、ワイド文字を指定します。

l modifier — 型 e、E、f、F、g、G と組み合わせて使用され、値を double に変換します。

ll modifier — 型 d、i、o、u、x、X と組み合わせて使用され、値を long long int または unsigned long long int に変換します。

ll modifier — n と組み合わせて使用され、ポインタが long long int を指すことを指定します。

L modifier — e、E、f、g、G と組み合わせて使用され、値を long double に変換します。

scanf (続き)

type

d、i	signed int
o	8 進の unsigned int
u	10 進の unsigned int
x	小文字 16 進数の unsigned int
X	大文字 16 進の unsigned int
e、E	科学表記法の double
f	10 進表記法の double
g、G	double (e、E、f の適切なものからの形式を使用)
c	char - 1 文字
s	string
p	ポインタ値
n	ポインタである必要があります。文字はスキャンされません。
[...]	文字配列。文字セットの検索を可能にします。左ブラケット ([) の直後のキャレット (^) は、スキャンセットを否定するため、ブラケットで囲まれた文字を除く ASCII 文字を可能にします。ダッシュ文字 (-) を使用し、ダッシュの前の文字で始まり、ダッシュの後ろの文字で終わる範囲を指定することができます。null 文字をスキャンセットに含めることはできません。
%	% 文字がスキャンされます。

例 :

```
#include <stdio.h> /* for scanf, printf */

int main(void)
{
  int number, items;
  char letter;
  char color[30], string[30];
  float salary;

  printf("Enter your favorite number, "
         "favorite letter, ");
  printf("favorite color desired salary "
         "and SSN:¥n");
  items = scanf("%d %c %[A-Za-z] %f %s", &number,
  &letter, &color, &salary, &string);

  printf("Number of items scanned = %d¥n", items);
  printf("Favorite number = %d, ", number);
  printf("Favorite letter = %c¥n", letter);
  printf("Favorite color = %s, ", color);
  printf("Desired salary = $%.2f¥n", salary);
  printf("Social Security Number = %s, ", string);
}
```

入力 :

UartIn.txt の内容 (シミュレータ用の stdin 入力として使用):

```
5 T Green 300000 123-45-6789
```

出力 :

```
Enter your favorite number, favorite letter,
favorite color, desired salary and SSN:
Number of items scanned = 5
Favorite number = 5, Favorite letter = T
Favorite color = Green, Desired salary = $300000.00
Social Security Number = 123-45-6789
```

setbuf

説明： ストリームのバッファ方法を指定します。

インクルード： <stdio.h>

プロトタイプ： void setbuf(FILE *stream, char *buf);

引数： *stream* オープンされたストリームを指すポインタ

buf ユーザー割り当てのバッファ

備考： fopen の後で、かつこのストリーム上で動作する他の関数のコール前に、setbuf がコールされる必要があります。buf が null ポインタである場合、setbuf は関数 setvbuf(*stream*, 0, _IONBF, BUFSIZ) をコールしてバッファリング不要にします。その他の場合は、setbuf が setvbuf(*stream*, *buf*, _IOFBF, BUFSIZ) をコールして、サイズ BUFSIZ のバッファによるフル・バッファリングを行います。setvbuf を参照してください。

例：

```
#include <stdio.h> /* for setbuf, printf, */
                   /* fopen, fclose,      */
                   /* FILE, NULL, BUFSIZ  */

int main(void)
{
  FILE *myfile1, *myfile2;
  char buf[BUFSIZ];

  if ((myfile1 = fopen("afile1", "w+")) != NULL)
  {
    setbuf(myfile1, NULL);
    printf("myfile1 has no buffering¥n");
    fclose(myfile1);
  }

  if ((myfile2 = fopen("afile2", "w+")) != NULL)
  {
    setbuf(myfile2, buf);
    printf("myfile2 has full buffering");
    fclose(myfile2);
  }

}
```

出力：

```
myfile1 has no buffering
myfile2 has full buffering
```

setvbuf

説明： バッファされるストリームとバッファ・サイズを指定します。

インクルード： <stdio.h>

プロトタイプ： int setvbuf(FILE *_stream_, char *_buf_, int _mode_, size_t _size_);

引数：

stream	オープンされたストリームを指すポインタ
buf	ユーザー割り当てのバッファ
mode	バッファリングのタイプ
size	バッファのサイズ

戻り値： 正常終了の場合 0 を返します。

備考： fopen の後で、かつこのストリーム上で動作する他の関数のコール前に、setvbuf がコールされる必要があります。モードとしては次が可能です：
_IOFBF ―フル・バッファリング
_IOLBF ―ライン・バッファリング
_IONBF ―バッファリングなし

例：

```
#include <stdio.h> /* for setvbuf, fopen, */
                   /* printf, FILE, NULL, */
                   /* _IONBF, _IOFBF      */

int main(void)
{
  FILE *myfile1, *myfile2;
  char buf[256];

  if ((myfile1 = fopen("afile1", "w+")) != NULL)
  {
    if (setvbuf(myfile1, NULL, _IONBF, 0) == 0)
      printf("myfile1 has no buffering¥n");
    else
      printf("Unable to define buffer stream "
             "and/or size¥n");
  }
  fclose(myfile1);

  if ((myfile2 = fopen("afile2", "w+")) != NULL)
  {
    if (setvbuf(myfile2, buf, _IOFBF, sizeof(buf)) ==
        0)
      printf("myfile2 has a buffer of %d "
             "characters¥n", sizeof(buf));
    else
      printf("Unable to define buffer stream "
             "and/or size¥n");
  }
  fclose(myfile2);
}
```

出力：

```
myfile1 has no buffering
myfile2 has a buffer of 256 characters
```

sprintf

説明： フォーマット化されたデータをストリングへプリントします。

インクルード： <stdio.h>

プロトタイプ： int sprintf(char *s, const char *format, ...);

引数：

s	出力用ストレージ・ストリング
format	フォーマット制御文字列
...	オプションの引数

戻り値： 格納された文字数を返します。ただし、終了の null 文字は含みません。

備考： フォーマット引数は同じ構文を持ち、printf 内で持っているものを使います。

例：

```
#include <stdio.h> /* for sprintf, printf */

int main(void)
{
  char sbuf[100], s[]="Print this string";
  int x = 1, y;
  char a = '¥n';

  y = sprintf(sbuf, "%s %d time%c", s, x, a);

  printf("Number of characters printed to "
         "string buffer = %d¥n", y);
  printf("String = %s¥n", sbuf);
}
```

出力：

```
Number of characters printed to string buffer = 25
String = Print this string 1 time
```

sscanf

説明： ストリングからのフォーマット化されたテキストをスキャンします。

インクルード： <stdio.h>

プロトタイプ： int sscanf(const char *s, const char *format, ...);

引数：

s	入力用ストレージ・ストリング
format	フォーマット制御文字列
...	オプションの引数

戻り値： 正常に変換され、割り当てられた項目数を返します。割り当てられた項目がない場合、0 を返します。最初の変換の前に入力エラーに遭遇した場合、EOF を返します。

備考： フォーマット引数は同じ構文を持ち、scanf 内で持っているものを使います。

sscanf (続き)

例：

```
#include <stdio.h> /* for sscanf, printf */

int main(void)
{
  char s[] = "5 T green 3000000.00";
  int number, items;
  char letter;
  char color[10];
  float salary;

  items = sscanf(s, "%d %c %s %f", &number, &letter,
              &color, &salary);

  printf("Number of items scanned = %d¥n", items);
  printf("Favorite number = %d¥n", number);
  printf("Favorite letter = %c¥n", letter);
  printf("Favorite color = %s¥n", color);
  printf("Desired salary = $%.2f¥n", salary);

}
```

出力：

```
Number of items scanned = 4
Favorite number = 5
Favorite letter = T
Favorite color = green
Desired salary = $3000000.00
```

tmpfile

説明： テンポラリ・ファイルを生成します。

インクルード： <stdio.h>

プロトタイプ： FILE *tmpfile(void)

戻り値： 正常終了の場合、ストリングを指すポインタを返します。その他の場合は NULL ポインタを返します。

備考： tmpfile は、独自のファイル名を持つファイルを生成します。テンポラリ・ファイルは、w+b (バイナリ読み書き) モードで開かれます。exit がコールされると、自動的に削除されます。その他の場合、ファイルはディレクトリ内に残ります。

例：

```
#include <stdio.h> /* for tmpfile, printf, */
                   /* FILE, NULL           */

int main(void)
{
  FILE *mytempfile;

  if ((mytempfile = tmpfile()) == NULL)
    printf("Cannot create temporary file");
  else
    printf("Temporary file was created");
}
```

出力：

```
Temporary file was created
```

tmpnam

説明： 独自のテンポラリ・ファイル名を生成します。

インクルード： <stdio.h>

プロトタイプ：
```
char *tmpnam(char *s);
```

引数： *s* テンポラリ名を指すポインタ

戻り値： 生成したファイル名を指すポインタを返し、ファイル名を *s* へ格納します。ファイル名を生成できない場合は、NULL ポインタを返します。

備考： 生成されたファイル名は既存のファイル名と競合しません。
L_tmpnam を使って tmpnam の引数が指す配列のサイズを決定します。

例：
```
#include <stdio.h> /* for tmpnam, L_tmpnam, */
                   /* printf, NULL           */

int main(void)
{
  char *myfilename;
  char mybuf[L_tmpnam];
  char *myptr = (char *) &mybuf;

  if ((myfilename = tmpnam(myptr)) == NULL)
    printf("Cannot create temporary file name");
  else
    printf("Temporary file %s was created",
           myfilename);
}
```

出力：
```
Temporary file ctm00001.tmp was created
```

ungetc

説明： 文字をストリームへプッシュバックします。

インクルード： <stdio.h>

プロトタイプ：
```
int ungetc(int c, FILE *stream);
```

引数： *c* プッシュバックする文字

stream オープンされたストリームを指すポインタ

戻り値： 正常終了の場合プッシュした文字を返します。その他の場合は EOF を返します。

備考： ストリームに対する後続の読み出しでは、プッシュバックした文字が返されます。複数の文字をプッシュバックすると、プッシュしたときと逆順で返されます。ファイル・ポジショニング関数 (fseek、fsetpos、rewind) に対するコールが正常に行われると、プッシュされた文字はすべて取り消されます。プッシュバックの 1 文字だけが保証されます。間に読み出しなし、またはファイル・ポジショニング動作なしで、ungetc を複数回コールすると、エラーが発生します。

ungetc (続き)

例：

```
#include <stdio.h> /* for ungetc, fgetc,     */
                   /* printf, fopen, fclose, */
                   /* FILE, NULL, EOF        */

int main(void)
{
  FILE *buf;
  char y, c;

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt¥n");
  else
  {
    y = fgetc(buf);
    while (y != EOF)
    {
      if (y == 'r')
      {
        c = ungetc(y, buf);
        if (c != EOF)
        {
          printf("2");
          y = fgetc(buf);
        }
      }
      printf("%c", y);
      y = fgetc(buf);
    }
    fclose(buf);
  }
}
```

入力：

afile.txt の内容 (入力として使用):

```
Short
Longer string
```

出力：

```
Sho2rt
Longe2r st2ring
```

vfprintf

説明： 可変長引数リストを使って、フォーマット化されたデータをストリームへプリントします。

インクルード：
```
<stdio.h>
<stdarg.h>
```

プロトタイプ：
```
int vfprintf(FILE *stream, const char *format, va_list ap);
```

引数：

stream	オープンされたストリームを指すポインタ
format	フォーマット制御文字列
ap	引数のリストを指すポインタ

戻り値： 生成した文字数を返します。エラーが発生した場合は負の数値を返します。

備考： フォーマット引数は同じ構文を持ち、printf 内で持っているものを使います。

可変長引数リストをアクセスするときは、マクロ va_start を使って *ap* 変数を初期化する必要があります。また、va_arg をさらにコールして再初期化することもできます。これは、vfprintf 関数をコールする前に行う必要があります。関数がリターンした後 va_end を起動します。詳細については、stdarg.h を参照してください。

例：
```
#include <stdio.h>  /* for vfprintf, fopen, */
                    /* fclose, printf,      */
                    /* FILE, NULL           */

#include <stdarg.h> /* for va_start,        */
                    /* va_list, va_end      */

FILE *myfile;

void errmsg(const char *fmt, ...)
{
  va_list ap;

  va_start(ap, fmt);
  vfprintf(myfile, fmt, ap);
  va_end(ap);
}

int main(void)
{
  int num = 3;

  if ((myfile = fopen("afile.txt", "w")) == NULL)
    printf("Cannot open afile.txt¥n");
  else
  {
    errmsg("Error: The letter '%c' is not %s¥n", 'a',
           "an integer value.");
    errmsg("Error: Requires %d%s%c", num,
           " or more characters.", '¥n');
  }
  fclose(myfile);
}
```

vfprintf (続き)

出力：
afile.txt の内容：
Error: The letter 'a' is not an integer value.
Error: Requires 3 or more characters.

vprintf

説明： 可変長引数リストを使って、フォーマット化されたテキストを stdout へプリントします。

インクルード：
```
<stdio.h>
<stdarg.h>
```

プロトタイプ：
```
int vprintf(const char *format, va_list ap);
```

引数：
format フォーマット制御文字列
ap 引数のリストを指すポインタ

戻り値： 生成した文字数を返します。エラーが発生した場合は負の数値を返します。

備考： フォーマット引数は同じ構文を持ち、printf 内で持っているものを使います。

可変長引数リストをアクセスするときは、マクロ va_start を使って ap 変数を初期化する必要があります。また、va_arg をさらにコールして再初期化することもできます。これは、vprintf 関数をコールする前に行う必要があります。関数がリターンした後 va_end を起動します。詳細については、stdarg.h を参照してください。

例：
```
#include <stdio.h>   /* for vprintf, printf */
#include <stdarg.h>  /* for va_start,       */
                     /* va_list, va_end     */

void errmsg(const char *fmt, ...)
{
  va_list ap;

  va_start(ap, fmt);
  printf("Error: ");
  vprintf(fmt, ap);
  va_end(ap);
}

int main(void)
{
  int num = 3;

  errmsg("The letter '%c' is not %s¥n", 'a',
         "an integer value.");
  errmsg("Requires %d%s¥n", num,
         " or more characters.¥n");
}
```

出力：
Error: The letter 'a' is not an integer value.
Error: Requires 3 or more characters.

vsprintf

説明： 可変長引数リストを使って、フォーマット化されたテキストをストリングへプリントします。

インクルード：
```
<stdio.h>
<stdarg.h>
```

プロトタイプ：
```
int vsprintf(char *s, const char *format, va_list ap);
```

引数：

s	出力用ストレージ・ストリング
format	フォーマット制御文字列
ap	引数のリストを指すポインタ

戻り値： 格納された文字数を返します。ただし、終了の null 文字は含みません。

備考： フォーマット引数は同じ構文を持ち、printf 内で持っているものを使います。

可変長引数リストをアクセスするときは、マクロ va_start を使って ap 変数を初期化する必要があります。また、va_arg をさらにコールして再初期化することもできます。これは、vsprintf 関数をコールする前に行う必要があります。関数がリターンした後 va_end を起動します。詳細については、stdarg.h を参照してください。

例：
```
#include <stdio.h>   /* for vsprintf, printf */
#include <stdarg.h>  /* for va_start,        */
                     /* va_list, va_end      */

void errmsg(const char *fmt, ...)
{
  va_list ap;
  char buf[100];

  va_start(ap, fmt);
  vsprintf(buf, fmt, ap);
  va_end(ap);
  printf("Error: %s", buf);
}

int main(void)
{
  int num = 3;

  errmsg("The letter '%c' is not %s¥n", 'a',
         "an integer value.");
  errmsg("Requires %d%s¥n", num,
         " or more characters.¥n");
}
```

出力：
```
Error: The letter 'a' is not an integer value.
Error: Requires 3 or more characters.
```

4.14 <STDLIB.H> ユーティリティ関数

ヘッダー・ファイル stdlib.h は、型、マクロ、関数 (テキスト変換、メモリ管理、検索機能、ソート機能を提供)、およびその他の一般的ユーティリティで構成されています。

div_t

説明： int 型のオペランドによる符号付き整数除算の商と余りを保持する型
インクルード： <stdlib.h>
プロトタイプ： typedef struct { int quot, rem; } div_t;
備考： これは、関数 div から返される構造体型です。

ldiv_t

説明： long 型のオペランドによる符号付き整数除算の商と余りを保持する型
インクルード： <stdlib.h>
プロトタイプ： typedef struct { long quot, rem; } ldiv_t;
備考： これは、関数 ldiv から返される構造体型です。

size_t

説明： sizeof オペレータの適用結果の型。
インクルード： <stdlib.h>

wchar_t

説明： ワイド文字値を保持する型。
インクルード： <stdlib.h>

EXIT_FAILURE

説明： 異常終了を報告します。
インクルード： <stdlib.h>
備考： EXIT_FAILURE は、異常終了ステータスを返すための exit 関数の値です。
例： 使用例については exit を参照してください。

EXIT_SUCCESS

説明： 正常終了を報告します。
インクルード： <stdlib.h>
備考： EXIT_SUCCESS は、正常終了ステータスを返すための exit 関数の値です。
例： 使用例については exit を参照してください。

MB_CUR_MAX

説明： マルチバイト文字内の最大文字数
インクルード： <stdlib.h>
値： 1

NULL

説明： null ポインタ定数の値。
インクルード： <stdlib.h>

RAND_MAX

説明： rand 関数から返へすことができる最大値。
インクルード： <stdlib.h>
値： 32767

abort

説明： 現在のプロセスを中断します。
インクルード： <stdlib.h>
プロトタイプ： void abort(void);
備考： 中断によりプロセッサがリセットされます。
例：

```
#include <stdio.h>  /* for fopen, fclose, */
                    /* printf, FILE, NULL */
#include <stdlib.h> /* for abort          */

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("samp.fil", "r")) == NULL)
  {
    printf("Cannot open samp.fil¥n");
    abort();
  }
  else
    printf("Success opening samp.fil¥n");

  fclose(myfile);
}
```

出力：

```
Cannot open samp.fil
ABRT
```

abs

説明： 絶対値を計算します。

インクルード： <stdlib.h>

プロトタイプ： int abs(int *i*);

引数： *i* 整数値

戻り値： *i* の絶対値を返します。

備考： 負の数値は正として返されます。正の数値は変更されません。

例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for abs    */

int main(void)
{
  int i;

  i = 12;
  printf("The absolute value of  %d is  %d¥n",
          i, abs(i));

  i = -2;
  printf("The absolute value of  %d is   %d¥n",
          i, abs(i));

  i = 0;
  printf("The absolute value of   %d is   %d¥n",
          i, abs(i));
}
```

出力：

```
The absolute value of 12 is 12
The absolute value of -2 is  2
The absolute value of  0 is  0
```

atexit

説明： プログラムが正常終了する際にコールされる指定された関数を登録します。

インクルード： <stdlib.h>

プロトタイプ： int atexit(void(**func*)(void));

引数： *func* コールされる関数

戻り値： 正常終了の場合ゼロを返します。その他の場合は非ゼロ値を返します。

備考： 登録済み関数がコールされるためには、プログラムは exit 関数コールで終了する必要があります。

例：

```
#include <stdio.h> /* for scanf, printf */
#include <stdlib.h> /* for atexit, exit */

void good_msg(void);
void bad_msg(void);
void end_msg(void);
```

atexit (続き)

```
int main(void)
{
  int number;

  atexit(end_msg);
  printf("Enter your favorite number:");
  scanf("%d", &number);
  printf(" %d¥n", number);
  if (number == 5)
  {
    printf("Good Choice¥n");
    atexit(good_msg);
    exit(0);
  }
  else
  {
    printf("%d!?¥n", number);
    atexit(bad_msg);
    exit(0);
  }
}

void good_msg(void)
{
  printf("That's an excellent number¥n");
}

void bad_msg(void)
{
  printf("That's an awful number¥n");
}

void end_msg(void)
{
  printf("Now go count something¥n");
}
```

入力：
UartIn.txt の内容 (シミュレータ用の stdin 入力として使用):
5

出力：
```
Enter your favorite number: 5
Good Choice
That's an excellent number
Now go count something
```

入力：
UartIn.txt の内容 (シミュレータ用の stdin 入力として使用):
42

出力：
```
Enter your favorite number: 42
42!?
That's an awful number
Now go count something
```

atof

説明： ストリングを倍精度浮動小数値に変換します。

インクルード： <stdlib.h>

プロトタイプ： double atof(const char *s);

引数： *s* 変換されるストリングを指すポインタ

戻り値： 正常終了の場合変換された値を返します。その他の場合は 0 を返します。

備考： 数値は次により構成されます：

```
[whitespace] [sign] digits [.digits]
  [ { e | E }[sign]digits]
```

オプションの空白スペース、オプションの符号、オプションの小数点が付いた 1 桁または複数桁、オプションの 1 桁または複数桁、オプションの e または E、オプションの符号付き指数がこの順に並びます。変換は、最初の認識不能文字に到達すると停止します。変換は strtod(s,0,0) と同じですが、エラー・チェックを行わないので errno は設定されません。

例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for atof   */

int main(void)
{
  char a[] = " 1.28";
  char b[] = "27.835e2";
  char c[] = "Number1";
  double x;

  x = atof(a);
  printf("String = ¥"%s¥" float = %f¥n", a, x);

  x = atof(b);
  printf("String = ¥"%s¥" float = %f¥n", b, x);

  x = atof(c);
  printf("String = ¥"%s¥"  float = %f¥n", c, x);
}
```

出力：

```
String = "1.28"  float = 1.280000
String = "27.835:e2" float = 2783.500000
String = "Number1" float = 0.000000
```

atoi

説明： ストリングを整数に変換します。

インクルード： <stdlib.h>

プロトタイプ： int atoi(const char *s);

引数： *s* 変換される文字列

戻り値： 正常終了の場合変換された整数を返します。その他の場合は 0 を返します。

備考： 数値は次により構成されます：
[whitespace] [sign] digits
オプションの空白スペース、オプションの符号、1 桁または複数桁がこの順に並びます。変換は、最初の認識不能文字に到達すると停止します。変換は (int) strtol(s,0,10) と等価ですが、エラー・チェックを行わないので errno は設定されません。

例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for atoi   */

int main(void)
{

  char a[] = " -127";
  char b[] = "Number1";
  int x;

  x = atoi(a);
  printf("String = ¥"%s¥"¥tint = %d¥n", a, x);

  x = atoi(b);
  printf("String = ¥"%s¥"¥tint = %d¥n", b, x);
}
```

出力：

```
String = " -127"    int = -127
String = "Number1"   int = 0
```

atol

説明： ストリングをロング整数に変換します。

インクルード： <stdlib.h>

プロトタイプ： long atol(const char *s);

引数： *s* 変換される文字列

戻り値： 正常終了の場合変換されたロング整数を返します。その他の場合は 0 を返します。

備考： 数値は次により構成されます：
[whitespace] [sign] digits
オプションの空白スペース、オプションの符号、1 桁または複数桁がこの順に並びます。変換は、最初の認識不能文字に到達すると停止します。変換は (int) strtol(s,0,10) と等価ですが、エラー・チェックを行わないので errno は設定されません。

atol (続き)

例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for atol   */

int main(void)
{
  char a[] = " -123456";
  char b[] = "2Number";
  long x;

  x = atol(a);
  printf("String = ¥"%s¥"  int = %ld¥n", a, x);

  x = atol(b);
  printf("String = ¥"%s¥"  int = %ld¥n", b, x);
}
```

出力：

```
String = " -123456"  int = -123456
String = "2Number"   int = 2
```

bsearch

説明： バイナリ検索を実行します。

インクルード： <stdlib.h>

プロトタイプ：

```
void *bsearch(const void *key, const void *base,
 size_t nelem, size_t size,
 int (*cmp)(const void *ck, const void *ce));
```

引数：

key	検索されるオブジェクト
base	検索データの先頭を指すポインタ
nelem	エレメント数
size	エレメントのサイズ
cmp	比較関数を指すポインタ
ck	検索のキーを指すポインタ
ce	キーと比較されるエレメントを指すポインタ

戻り値： 見つかった場合、検索対象オブジェクトを指すポインタを返します。その他の場合は NULL を返します。

備考： *ck* が ce より小さい場合、比較関数から返される値は <0 になります。ck と ce が等しい場合は 0 が返されます。*ck* が *ce* より大きい場合は、>0 が返されます。

次の例では、bsearch をコールする前に、qsort を使ってリストをソートしています。bsearch は、比較関数に従ってリストをソートすることを必要とします。この comp は昇順を使っています。

bsearch (続き)

例：

```
#include <stdlib.h> /* for bsearch, qsort */
#include <stdio.h>  /* for printf, sizeof */

#define NUM 7

int comp(const void *e1, const void *e2);

int main(void)
{
  int list[NUM] = {35, 47, 63, 25, 93, 16, 52};
  int x, y;
  int *r;

  qsort(list, NUM, sizeof(int), comp);

  printf("Sorted List:   ");
  for (x = 0; x < NUM; x++)
    printf("%d  ", list[x]);

  y = 25;
  r = bsearch(&y, list, NUM, sizeof(int), comp);
  if (r)
    printf("¥nThe value %d was found¥n", y);
  else
    printf("¥nThe value %d was not found¥n", y);

  y = 75;
  r = bsearch(&y, list, NUM, sizeof(int), comp);
  if (r)
    printf("¥nThe value %d was found¥n", y);
  else
    printf("¥nThe value %d was not found¥n", y);
}

int comp(const void *e1, const void *e2)
{

  const int * a1 = e1;
  const int * a2 = e2;

  if (*a1 < *a2)
    return -1;
  else if (*a1 == *a2)
    return 0;
  else
    return 1;
}
```

出力：

```
Sorted List: 16 25 35 47 52 63 93
The value 25 was found

The value 75 was not found
```

calloc

説明： メモリ内に配列を割り当てて、エレメントを 0 に初期化します。

インクルード： <stdlib.h>

プロトタイプ： void *calloc(size_t *nelem*, size_t *size*);

引数： *nelem* エレメント数

size 各エレメントの長さ

戻り値： 正常終了の場合、割り当てられた領域を指すポインタを返します。その他の場合は null ポインタを返します。

備考： calloc から返されたメモリは、任意サイズのデータ・エレメントに合わせて整列され、ゼロに初期化されています。

例：

```
/* This program allocates memory for the      */
/* array 'i' of long integers and initializes */
/* them to zero.                              */

#include <stdio.h>  /* for printf, NULL */
#include <stdlib.h> /* for calloc, free */

int main(void)
{
  int x;
  long *i;

  i = (long *)calloc(5, sizeof(long));
  if (i != NULL)
  {
    for (x = 0; x < 5; x++)
      printf("i[%d] = %ld¥n", x, i[x]);
    free(i);
  }
  else
    printf("Cannot allocate memory¥n");
}
```

出力：

```
i[0] = 0
i[1] = 0
i[2] = 0
i[3] = 0
i[4] = 0
```

div

説明： 2 つの数値の商と余りを計算します。

インクルード： <stdlib.h>

プロトタイプ： div_t div(int *numer*, int *denom*);

引数： *numer* 分子

denom 分母

戻り値： 商と余りを返します。

備考： 返された商は、分子÷分母と同じ符号を持っています。余りの符号は、商×分母＋余りが分子に等しくなるように決めます (quot * *denom* + rem = *numer*)。ゼロ割りにより算術例外エラーが発生します。デフォルトでは、リセットが発生します。別のことを行うためには、算術エラーハンドラを用意する必要があります。

div (続き)

例 :

```
#include <stdlib.h> /* for div, div_t */
#include <stdio.h>  /* for printf     */

void __attribute__((__interrupt__))
_MathError(void)
{
  printf("Illegal instruction executed¥n");
  abort();
}

int main(void)
{
  int x, y;
  div_t z;

  x = 7;
  y = 3;
  printf("For div(%d, %d)¥n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the "
         "remainder is %d¥n¥n", z.quot, z.rem);

  x = 7;
  y = -3;
  printf("For div(%d, %d)¥n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the "
         "remainder is %d¥n¥n", z.quot, z.rem);

  x = -5;
  y = 3;
  printf("For div(%d, %d)¥n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the "
         "remainder is %d¥n¥n", z.quot, z.rem);

  x = 7;
  y = 7;
  printf("For div(%d, %d)¥n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the "
         "remainder is %d¥n¥n", z.quot, z.rem);

  x = 7;
  y = 0;
  printf("For div(%d, %d)¥n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the "
         "remainder is %d¥n¥n", z.quot, z.rem);
}
```

div (続き)

出力：

```
For div(7, 3)
The quotient is 2 and the remainder is 1

For div(7, -3)
The quotient is -2 and the remainder is 1

For div(-5, 3)
The quotient is -1 and the remainder is -2

For div(7, 7)
The quotient is 1 and the remainder is 0

For div(7, 0)
Illegal instruction executed
ABRT
```

exit

説明： クリーンアップ後プログラムを停止させます。

インクルード： <stdlib.h>

プロトタイプ： void exit(int *status*);

引数： *status* 終了ステータス

備考： exit は、atexit により登録された任意の関数を登録の逆順でコールし、バッファをクリアし、ストリームを閉じ、tmpfile により生成されたすべてのテンポラリ・ファイルを閉じ、プロセッサをリセットします。この関数はカスタマイズできます。pic30-libs を参照してください。

例：

```
#include <stdio.h>  /* for fopen, printf, */
                    /* FILE, NULL         */
#include <stdlib.h> /* for exit           */

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("samp.fil", "r" )) == NULL)
  {
    printf("Cannot open samp.fil¥n");
    exit(EXIT_FAILURE);
  }
  else
  {
    printf("Success opening samp.fil¥n");
    exit(EXIT_SUCCESS);
  }
  printf("This will not be printed");
}
```

出力：

```
Cannot open samp.fil
```

free

説明： メモリを解放します。

インクルード： <stdlib.h>

プロトタイプ： void free(void *ptr);

引数： *ptr* 開放するメモリを指すポインタ

備考： calloc、malloc、realloc により割り当てられたメモリを解放します。すでに解法済みの領域 (前の free コールまたは realloc による)、または calloc、malloc、realloc で割り当てられなかった領域に対して free を使うと、動作は不定になります。

例：

```
#include <stdio.h>  /* for printf, sizeof, */
                    /* NULL                */
#include <stdlib.h> /* for malloc, free    */

int main(void)
{
  long *i;

  if ((i = (long *)malloc(50 * sizeof(long))) ==
       NULL)
    printf("Cannot allocate memory¥n");
  else
  {
    printf("Memory allocated¥n");
    free(i);
    printf("Memory freed¥n");
  }
}
```

出力：

```
Memory allocated
Memory freed
```

getenv

説明： 環境変数の値を取得します。

インクルード： <stdlib.h>

プロトタイプ： char *get*env*(const char **name*);

引数： *name* 環境変数の名前

戻り値： 正常終了の場合、環境変数の値を指すポインタを返します。その他の場合は null ポインタを返します。

備考： この関数を説明通りに使用するためには、カスタマイズする必要があります (pic30-libs 参照)。デフォルトでは、環境リスト内に get*env* が探せる内容が存在しません。

getenv (続き)

例：

```
#include <stdio.h>  /* for printf, NULL */
#include <stdlib.h> /* for getenv       */

int main(void)
{
  char *incvar;

  incvar = getenv("INCLUDE");
  if (incvar != NULL)
    printf("INCLUDE environment variable = %s¥n",
           incvar);
  else
    printf("Cannot find environment variable "
           "INCLUDE ");
}
```

出力：

```
Cannot find environment variable INCLUDE
```

labs

説明： ロング整数の絶対値を計算します。

インクルード： <stdlib.h>

プロトタイプ： long labs(long *i*);

引数： *i* ロング整数値

戻り値： *i* の絶対値を返します。

備考： 負の数値は正として返されます。正の数値は変更されません。

例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for labs   */

int main(void)
{
  long i;

  i = 123456;
  printf("The absolute value of %7ld is %6ld¥n",
         i, labs(i));

  i = -246834;
  printf("The absolute value of %7ld is %6ld¥n",
         i, labs(i));

  i = 0;
  printf("The absolute value of %7ld is %6ld¥n",
         i, labs(i));
}
```

出力：

```
The absolute value of  123456 is 123456
The absolute value of -246834 is 246834
The absolute value of    0 is   0
```

ldiv

説明： 2 つのロング整数値の商と余りを計算します。

インクルード： <stdlib.h>

プロトタイプ： ldiv_t ldiv(long *numer*, long *denom*);

引数： *numer* 分子

denom 分母

戻り値： 商と余りを返します。

備考： 返された商は、分子÷分母と同じ符号を持っています。余りの符号は、商×分母＋余りが分子に等しくなるように決めます (quot * *denom* + *rem* = *numer*)。分母がゼロの場合、動作は不定になります。

例：

```
#include <stdlib.h> /* for ldiv, ldiv_t */
#include <stdio.h>  /* for printf       */

int main(void)
{
  long x,y;
  ldiv_t z;

  x = 7;
  y = 3;
  printf("For ldiv(%ld, %ld)¥n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld¥n¥n", z.quot, z.rem);

  x = 7;
  y = -3;
  printf("For ldiv(%ld, %ld)¥n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld¥n¥n", z.quot, z.rem);

  x = -5;
  y = 3;
  printf("For ldiv(%ld, %ld)¥n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld¥n¥n", z.quot, z.rem);

  x = 7;
  y = 7;
  printf("For ldiv(%ld, %ld)¥n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld¥n¥n", z.quot, z.rem);

  x = 7;
  y = 0;
  printf("For ldiv(%ld, %ld)¥n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld¥n¥n",
         z.quot, z.rem);
}
```

ldiv (続き)

出力：

```
For ldiv(7, 3)
The quotient is 2 and the remainder is 1

For ldiv(7, -3)
The quotient is -2 and the remainder is 1

For ldiv(-5, 3)
The quotient is -1 and the remainder is -2

For ldiv(7, 7)
The quotient is 1 and the remainder is 0

For ldiv(7, 0)
The quotient is -1 and the remainder is 7
```

説明：
最後の例 (`ldiv(7,0)`) で、分母がゼロであため、動作は不定になります。

malloc

説明： メモリを割り当てます。

インクルード： `<stdlib.h>`

プロトタイプ： `void *malloc(size_t size);`

引数： *size*　割り当てる文字数

戻り値： 正常終了の場合、割り当てられた領域を指すポインタを返します。その他の場合は null ポインタを返します。

備考： malloc は自分が返すメモリを初期化しません。

例：

```
#include <stdio.h>  /* for printf, sizeof, */
                    /* NULL                */
#include <stdlib.h> /* for malloc, free    */

int main(void)
{
  long *i;

  if ((i = (long *)malloc(50 * sizeof(long))) ==
      NULL)
    printf("Cannot allocate memory¥n");
  else
  {
    printf("Memory allocated¥n");
    free(i);
    printf("Memory freed¥n");
  }
}
```

出力：

```
Memory allocated
Memory freed
```

mblen

説明： マルチバイト文字の長さを取得します (備考参照)。

インクルード： `<stdlib.h>`

プロトタイプ：
```
int mblen(const char *s, size_t n);
```

引数：
s マルチバイト文字を指すポインタ
n チェックするバイト数

戻り値： s が null 文字を指定する場合ゼロを返します。その他の場合は 1 を返します。

備考： MPLAB C30 では、1 バイトより長いマルチバイト文字をサポートしていません。

mbstowcs

説明： マルチバイト文字列をワイド文字列へ変換します (備考参照)。

インクルード： `<stdlib.h>`

プロトタイプ：
```
size_t mbstowcs(wchar_t *wcs, const char *s,size_t n);
```

引数：
wcs ワイド文字列を指すポインタ
s マルチバイト文字列を指すポインタ
n 変換するワイド文字数

戻り値： 格納されたワイド文字数を返します。ただし、null 文字は含みません。

備考： mbstowcs は、最初に null ワイド文字に遭遇するまで *n* 個のワイド文字を変換します。MPLAB C30 では、1 バイトより長いマルチバイト文字をサポートしていません。

mbtowc

説明： マルチバイト文字をワイド文字に変換します (備考参照)。

インクルード： `<stdlib.h>`

プロトタイプ：
```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

引数：
pwc ワイド文字を指すポインタ
s マルチバイト文字を指すポインタ
n チェックするバイト数

戻り値： s が null 文字を指定する場合ゼロを返します。その他の場合は 1 を返します。

備考： 変換されたワイド文字は pwc に格納されます。MPLAB C30 では、1 バイトより長いマルチバイト文字をサポートしていません。

qsort

説明： クィック・ソートを実行します。

インクルード： <stdlib.h>

プロトタイプ：
```
void qsort(void *base, size_t nelem, size_t size,
 int (*cmp)(const void *e1, const void *e2));
```

引数：

base	配列の先頭を指すポインタ
nelem	エレメント数
size	エレメントのサイズ
cmp	比較関数を指すポインタ
e1	検索のキーを指すポインタ
e2	キーと比較されるエレメントを指すポインタ

備考： qsort は、ソートされた配列で配列を上書きします。比較関数はユーザーが用意します。次の例では、比較関数に従ってリストがソートされています。この comp は昇順を使っています。

例：
```
#include <stdlib.h> /* for qsort */
#include <stdio.h> /* for printf */

#define NUM 7

int comp(const void *e1, const void *e2);

int main(void)
{
  int list[NUM] = {35, 47, 63, 25, 93, 16, 52};
  int x;

  printf("Unsorted List: ");
  for (x = 0; x < NUM; x++)
    printf("%d  ", list[x]);

  qsort(list, NUM, sizeof(int), comp);

  printf("¥n");
  printf("Sorted List:   ");
  for (x = 0; x < NUM; x++)
    printf("%d  ", list[x]);

}

int comp(const void *e1, const void *e2)
{

  const int * a1 = e1;
  const int * a2 = e2;

  if (*a1 < *a2)
    return -1;
  else if (*a1 == *a2)
    return 0;
  else
    return 1;
}
```

出力：
```
Unsorted List: 35 47 63 25 93 16 52
Sorted List:  16 25 35 47 52 63 93
```

rand

説明： 擬似ランダム整数を生成します。

インクルード： <stdlib.h>

プロトタイプ： int rand(void);

戻り値： 0 と RAND_MAX との間の整数を返します。

備考： この関数をコールすると、範囲 [0,RAND_MAX] の擬似ランダム整数値を返します。この関数を有効に使うためには、srand 関数を使ってランダム数ジェネレータのシードを設定する必要があります。シードを使用しない場合 (下の例) または同じシード値を使用した場合には、この関数は常に整数の同じシーケンスを返します (シードの例については srand を参照してください)。

例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for rand   */

int main(void)
{
  int x;

  for (x = 0; x < 5; x++)
    printf("Number = %d¥n", rand());
}
```

出力：

```
Number = 21422
Number = 2061
Number = 16443
Number = 11617
Number = 9125
```

プログラムを 2 回目に実行した場合、数値が同じであることに注意してください。ランダム数ジェネレータのシードの設定については例を参照してください。

realloc

説明： サイズ変更を可能にするためメモリを再割り当てします。

インクルード： <stdlib.h>

プロトタイプ： void *realloc(void *ptr, size_t size);

引数：

ptr 割り当て済みメモリを指すポインタ

size 割り当てる新しいサイズ

戻り値： 正常終了の場合、割り当てられた領域を指すポインタを返します。その他の場合は null ポインタを返します。

備考： 既存オブジェクトが新しいオブジェクトより小さい場合、既存オブジェクト全体が新しいオブジェクトへコピーされ、新しいオブジェクトの残りは不定になります。既存オブジェクトが新しいオブジェクトより大きい場合、新しいオブジェクトに収容できる限り既存オブジェクトをコピーします。realloc が新しいオブジェクトの割り当てに成功すると、既存オブジェクトが開放されます。その他の場合は、既存オブジェクトを変更しません。異常終了の場合 realloc は null ポインタを返すので、既存オブジェクトを指すテンポラリ・ポインタは維持してください。

realloc (続き)

例：

```
#include <stdio.h> /* for printf, sizeof, NULL */
#include <stdlib.h> /* for realloc, malloc, free */

int main(void)
{
  long *i, *j;

  if ((i = (long *)malloc(50 * sizeof(long)))
       == NULL)
    printf("Cannot allocate memory¥n");
  else
  {
    printf("Memory allocated¥n");
    /* Temp pointer in case realloc() fails */
    j = i;

    if ((i = (long *)realloc(i, 25 * sizeof(long)))
        == NULL)
    {
      printf("Cannot reallocate memory¥n");
                /* j pointed to allocated memory */
      free(j);
    }
    else
    {
      printf("Memory reallocated¥n");
      free(i);
    }
  }
}
```

出力：

```
Memory allocated
Memory reallocated
```

srand

説明： 擬似ランダム数値シーケンスの開始シードを設定します。

インクルード： <stdlib.h>

プロトタイプ： void srand(unsigned int *seed*);

引数： *seed* 擬似ランダム数値シーケンスの開始値

戻り値： なし

備考： この関数は、rand 関数により生成される擬似ランダム数値シーケンスの開始シードを設定します。同じシード値を使用した場合には、この rand 関数は常に整数の同じシーケンスを返します。シード値 = 1 で rand をコールすると、生成される数値シーケンスは、srand を最初にコールせずに rand がコールされた場合と同じになります。

例：

```
#include <stdio.h>  /* for printf      */
#include <stdlib.h> /* for rand, srand */

int main(void)
{
  int x;

  srand(7);
  for (x = 0; x < 5; x++)
    printf("Number = %d¥n", rand());
}
```

出力：

```
Number = 16327
Number = 5931
Number = 23117
Number = 30985
Number = 29612
```

strtod

説明： 部分文字列を倍精度型の浮動小数値へ変換します。

インクルード： <stdlib.h>

プロトタイプ： double strtod(const char *s, char **endptr);

引数： *s* 変換される文字列

endptr 変換が停止する文字を指すポインタ

戻り値： 正常終了の場合変換された数値を返します。その他の場合は 0 を返します。

備考： 数値は次により構成されます：

```
[whitespace] [sign] digits [.digits]
  [ { e | E }[sign]digits]
```

オプションの空白スペース、オプションの符号、オプションの小数点が付いた 1 桁または複数桁、オプションの 1 桁または複数桁、オプションの e または E、オプションの符号付き指数がこの順に並びます。
strtod は、数値に変換できない文字に到達するまで文字列を変換します。endptr は、変換しない最初の文字から始まる残りの文字列を指します。
範囲エラーが発生した場合、errno が設定されます。

strtod (続き)

例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for strtod */

int main(void)
{
  char *end;
  char a[] = "1.28 inches";
  char b[] = "27.835e2i";
  char c[] = "Number1";
  double x;

  x = strtod(a, &end);
  printf("String = ¥"%s¥"  float = %f¥n", a, x );
  printf("Stopped at: %s¥n¥n", end );

  x = strtod(b, &end);
  printf("String = ¥"%s¥" float = %f¥n", b, x );
  printf("Stopped at: %s¥n¥n", end );

  x = strtod(c, &end);
  printf("String = ¥"%s¥"   float = %f¥n", c, x );
  printf("Stopped at: %s¥n¥n", end );
}
```

出力：

```
String = "1.28 inches" float = 1.280000
Stopped at: inches

String = "27.835e2i" float = 2783.500000
Stopped at: i

String = "Number1"  float = 0.000000
Stopped at: Number1
```

strtol

説明： 部分文字列をロング整数に変換します。

インクルード： <stdlib.h>

プロトタイプ： long strtol(const char *s, char **endptr, int base);

引数：

s	変換される文字列
endptr	変換が停止する文字を指すポインタ
base	変換で使用する基数

戻り値： 正常終了の場合変換された数値を返します。その他の場合は 0 を返します。

備考： 基数がゼロの場合、strtol は基数を自動的に決定しようとします。先頭のゼロで決まる 8 進、先頭の 0x または 0X で決まる 16 進数、その他のケースの 10 進が可能です。基数が指定された場合、strtol は一連の桁と文字 a ～ z (大文字小文字を区別) を変換します。ここで、a ～ z は数値 10 ～ 36 を表わします。基数外に遭遇したとき変換は停止します。endptr は、変換できない最初の文字で始まる残りの文字列を指します。範囲エラーが発生した場合、errno が設定されます。

例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for strtol */

int main(void)
{
  char *end;
  char a[] = "-12BGEE";
  char b[] = "1234Number";
  long x;

  x = strtol(a, &end, 16);
  printf("String = ¥"%s¥"  long = %ld¥n", a, x );
  printf("Stopped at: %s¥n¥n", end );

  x = strtol(b, &end, 4);
  printf("String = ¥"%s¥"  long = %ld¥n", b, x );
  printf("Stopped at: %s¥n¥n", end );
}
```

出力：

```
String = "-12BGEE" long = -299
Stopped at: GEE

String = "1234Number" long = 27
Stopped at: 4Number
```

strtoul

説明： 部分文字列を符号なしロング整数に変換します。

インクルード： <stdlib.h>

プロトタイプ：
```
unsigned long strtoul(const char *s, char **endptr,
 int base);
```

引数：

s	変換される文字列
endptr	変換が停止する文字を指すポインタ
base	変換で使用する基数

戻り値： 正常終了の場合変換された数値を返します。その他の場合は 0 を返します。

備考： 基数がゼロの場合、strtol は基数を自動的に決定しようとします。先頭のゼロで決まる 8 進、先頭の 0x または 0X で決まる 16 進数、その他のケースの 10 進が可能です。基数が指定された場合、strtol は一連の桁と文字 a ～ z (大文字小文字を区別) を変換します。ここで、a ～ z は数値 10 ～ 36 を表わします。基数外に遭遇したとき変換は停止します。endptr は、変換できない最初の文字で始まる残りの文字列を指します。範囲エラーが発生した場合、errno が設定されます。

例：
```
#include <stdio.h>  /* for printf  */
#include <stdlib.h> /* for strtoul */

int main(void)
{
  char *end;
  char a[] = "12BGET3";
  char b[] = "0x1234Number";
  char c[] = "-123abc";
  unsigned long x;

  x = strtoul(a, &end, 25);
  printf("String = ¥"%s¥"  long = %lu¥n", a, x );
  printf("Stopped at: %s¥n¥n", end );

  x = strtoul(b, &end, 0);
  printf("String = ¥"%s¥"  long = %lu¥n", b, x );
  printf("Stopped at: %s¥n¥n", end );

  x = strtoul(c, &end, 0);
  printf("String = ¥"%s¥"  long = %lu¥n", c, x );
  printf("Stopped at: %s¥n¥n", end );
}
```

出力：
```
String = "12BGET3" long = 429164
Stopped at: T3

String = "0x1234Number" long = 4660
Stopped at: Number

String = "-123abc" long = 4294967173
Stopped at: abc
```

system

説明： コマンドを実行します。

インクルード： <stdlib.h>

プロトタイプ：
```
int system(const char *s);
```

引数： *s* 実行するコマンド

備考： この関数を説明通りに使用するためには、カスタマイズする必要があります (pic30-libs 参照)。デフォルトでは、NULL 以外を使ってコールした場合、system はリセットを発生させます。system(NULL) は何も実行しません。

例：
```
/* This program uses system */
/* to TYPE its source file. */

#include <stdlib.h> /* for system */

int main(void)
{
  system("type sampsystem.c");
}
```

出力：
```
System(type sampsystem.c) called: Aborting
```

wctomb

説明： ワイド文字をマルチバイト文字に変換します (備考参照)。

インクルード： <stdlib.h>

プロトタイプ：
```
int wctomb(char *s, wchar_t wchar);
```

引数： *s* マルチバイト文字を指すポインタ

wchar 変換するワイド文字

戻り値： s が null 文字を指定する場合ゼロを返します。その他の場合は 1 を返します。

備考： 変換されたマルチバイト文字は *s* に格納されます。MPLAB C30 では、1 文字より長いマルチバイト文字をサポートしていません。

wcstombs

説明： ワイド文字列をマルチバイト文字列へ変換します (備考参照)。

インクルード： <stdlib.h>

プロトタイプ：
```
size_t wcstombs(char *s, const wchar_t *wcs, size_t n);
```

引数： *s* マルチバイト文字列を指すポインタ

wcs ワイド文字列を指すポインタ

n 変換する文字数

戻り値： 格納された文字数を返します。ただし、null 文字は含みません。

備考： wcstombs は、最初に null 文字に遭遇するまで *n* 個のマルチバイト文字を変換します。MPLAB C30 では、1 文字より長いマルチバイト文字をサポートしていません。

4.15 <STRING.H> 文字列関数

ヘッダー・ファイル string.h は、型、マクロ、および関数（文字列を操作するツールを提供）で構成されています。

size_t

説明： sizeof オペレータの適用結果の型。

インクルード： <string.h>

NULL

説明： null ポインタ定数の値。

インクルード： <string.h>

memchr

説明： バッファ内で文字を検索します。

インクルード： <string.h>

プロトタイプ： void *memchr(const void *s, int c, size_t n);

引数：

s バッファを指すポインタ

c 見つける文字

n チェックする文字数

戻り値： 見つかった場合、一致した対象のロケーションを指すポインタを返します。その他の場合は null を返します。

備考： memchr は、*c* が最初に見つかったとき、または *n* 個の文字を検索した後に、停止します。

例：

```
#include <string.h> /* for memchr, NULL */
#include <stdio.h> /* for printf   */

int main(void)
{
  char buf1[50] = "What time is it?";
  char ch1 = 'i', ch2 = 'y';
  char *ptr;
  int res;

  printf("buf1 : %s¥n¥n", buf1);

  ptr = memchr(buf1, ch1, 50);
if (ptr != NULL)
  {
   res = ptr - buf1 + 1;
   printf("%c found at position %d¥n", ch1, res);
  }
  else
    printf("%c not found¥n", ch1);
```

memchr (続き)

```
  printf("¥n");

  ptr = memchr(buf1, ch2, 50);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d¥n", ch2, res);
  }
  else
    printf("%c not found¥n", ch2);
}
```

出力：

```
buf1 : What time is it?

i found at position 7

y not found
```

memcmp

説明： 2 つのバッファの内容を比較します。

インクルード： <string.h>

プロトタイプ： int memcmp(const void *s1, const void *s2, size_t n);

引数：

s1 最初のバッファ

s2 2 つ目のバッファ

n 比較する文字数

戻り値： *s1* が *s2* より大きい場合は正の数値を、*s1* と *s2* が等しい場合はゼロを、*s1* が *s2* より小さい場合は負の数値を、それぞれ返します。

備考： この関数は *s1* 内の最初の *n* 個の文字と *s2* 内の最初の *n* 個の文字を比較して、2 つのバッファ値の関係 (大、小、等) を表わす値を返します。

例：

```
#include <string.h> /* memcmp     */
#include <stdio.h>  /* for printf */

int main(void)
{
  char buf1[50] = "Where is the time?";
  char buf2[50] = "Where did they go?";
  char buf3[50] = "Why?";
  int res;

  printf("buf1 : %s¥n", buf1);
  printf("buf2 : %s¥n", buf2);
  printf("buf3 : %s¥n¥n", buf3);

  res = memcmp(buf1, buf2, 6);
  if (res < 0)
    printf("buf1 comes before buf2¥n");
  else if (res == 0)
    printf("6 characters of buf1 and buf2 "
           "are equal¥n");
  else
    printf("buf2 comes before buf1¥n");
```

memcmp (続き)

```
  printf("¥n");

  res = memcmp(buf1, buf2, 20);
  if (res < 0)
    printf("buf1 comes before buf2¥n");
  else if (res == 0)
    printf("20 characters of buf1 and buf2 "
           "are equal¥n");
  else
    printf("buf2 comes before buf1¥n");

  printf("¥n");

  res = memcmp(buf1, buf3, 20);
  if (res < 0)
    printf("buf1 comes before buf3¥n");
  else if (res == 0)
    printf("20 characters of buf1 and buf3 "
           "are equal¥n");
  else
    printf("buf3 comes before buf1¥n");
}
```

出力：

```
buf1 : Where is the time?
buf2 : Where did they go?
buf3 : Why?

6 characters of buf1 and buf2 are equal

buf2 comes before buf1

buf1 comes before buf3
```

memcpy

説明： 複数の文字を 1 つのバッファから別のバッファへコピーします。

インクルード： <string.h>

プロトタイプ： void *memcpy(void *dst , const void *src , size_t n);

引数：
dst 文字をコピーする相手先バッファ
src コピーする文字が格納されているバッファ
n コピーする文字数

戻り値： *dst* を返します。

備考： memcpy は、*n* 個の文字をソース・バッファ src からディステネーション・バッファ *dst* へコピーします。バッファが重複している場合は、動作は不定になります。

例：

```
#include <string.h> /* memcpy     */
#include <stdio.h>  /* for printf */

int main(void)
{
  char buf1[50] = "";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";

  printf("buf1 : %s¥n", buf1);
  printf("buf2 : %s¥n", buf2);
  printf("buf3 : %s¥n¥n", buf3);

  memcpy(buf1, buf2, 6);
  printf("buf1 after memcpy of 6 chars of "
         "buf2: ¥n¥t%s¥n", buf1);

  printf("¥n");

  memcpy(buf1, buf3, 5);
  printf("buf1 after memcpy of 5 chars of "
         "buf3: ¥n¥t%s¥n", buf1);
}
```

出力：

```
buf1 :
buf2 : Where is the time?
buf3 : Why?

buf1 after memcpy of 6 chars of buf2:
        Where

buf1 after memcpy of 5 chars of buf3:
        Why?
```

memmove

説明： バッファ領域が重複していても、n 個の文字をソース・バッファからディステネーション・バッファへコピーします。

インクルード： <string.h>

プロトタイプ： void *memmove(void *s1, const void *s2, size_t n);

引数：
s1 文字をコピーする相手先バッファ (ディステネーション)
s2 コピーする文字が格納されているバッファ (ソース)
n *s2* から *s1* へコピーする文字数

戻り値： ディステネーション・バッファを指すポインタを返します。

備考： バッファが重複する場合、文字は最初に *s2* から読み出され、次に *s1* へ書き込まれて、バッファが壊されないようにします。

例：

```
#include <string.h> /* for memmove */
#include <stdio.h>  /* for printf  */

int main(void)
{
  char buf1[50] = "When time marches on";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";

  printf("buf1 : %s¥n", buf1);
  printf("buf2 : %s¥n", buf2);
  printf("buf3 : %s¥n¥n", buf3);

  memmove(buf1, buf2, 6);
  printf("buf1 after memmove of 6 chars of "
         "buf2: ¥n¥t%s¥n", buf1);

  printf("¥n");

  memmove(buf1, buf3, 5);
  printf("buf1 after memmove of 5 chars of "
         "buf3: ¥n¥t%s¥n", buf1);
}
```

出力：

```
buf1 : When time marches on
buf2 : Where is the time?
buf3 : Why?

buf1 after memmove of 6 chars of buf2:
        Where ime marches on

buf1 after memmove of 5 chars of buf3:
        Why?
```

memset

説明： 指定された文字をディステネーション・バッファへコピーします。

インクルード： <string.h>

プロトタイプ： void *memset(void *s, int c, size_t n);

引数：
s バッファ
c バッファに書き込まれる文字
n 回数

戻り値： 文字が書き込まれたバッファを返します。

備考： 文字 c が n 回バッファに書き込まれます。

例：

```
#include <string.h> /* for memset */
#include <stdio.h>  /* for printf */

int main(void)
{
  char buf1[20] = "What time is it?";
  char buf2[20] = "";
  char ch1 = '?', ch2 = 'y';
  char *ptr;
  int res;

  printf("memset(¥"%s¥", ¥'%c¥',4);¥n", buf1, ch1);
  memset(buf1, ch1, 4);
  printf("buf1 after memset: %s¥n", buf1);

  printf("¥n");
  printf("memset(¥"%s¥", ¥'%c¥',10);¥n", buf2, ch2);
  memset(buf2, ch2, 10);
  printf("buf2 after memset: %s¥n", buf2);
}
```

出力：

```
memset("What time is it?", '?',4);
buf1 after memset: ???? time is it?

memset("", 'y',10);
buf2 after memset: yyyyyyyyyy
```

strcat

説明： ソース文字列のコピーをディステネーション文字列の終わりにアペンドします。

インクルード： <string.h>

プロトタイプ： char *strcat(char *s1, const char *s2);

引数：
s1 コピー先の null で終了したディステネーション文字列
s2 コピー元の null で終了したソース文字列

戻り値： ディステネーション文字列を指すポインタを返します。

備考： この関数は、ソース文字列 (終了 null 文字を含む) をディステネーション文字列の終わりにアペンドします。ソース文字列の先頭の文字が、ディステネーション文字列の終わりの null 文字を上書きします。バッファが重複している場合は、動作は不定になります。

例：

```
#include <string.h> /* for strcat, strlen */
#include <stdio.h>  /* for printf         */

int main(void)
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";

  printf("buf1 : %s¥n", buf1);
  printf("¥t(%d characters)¥n¥n", strlen(buf1));
  printf("buf2 : %s¥n", buf2);
  printf("¥t(%d characters)¥n¥n", strlen(buf2));

  strcat(buf1, buf2);
  printf("buf1 after strcat of buf2: ¥n¥t%s¥n",
          buf1);
  printf("¥t(%d characters)¥n", strlen(buf1));

  printf("¥n");

  strcat(buf1, "Why?");
  printf("buf1 after strcat of ¥"Why?¥": ¥n¥t%s¥n",
         buf1);
  printf("¥t(%d characters)¥n", strlen(buf1));
}
```

出力：

```
buf1 : We're here
        (10 characters)

buf2 : Where is the time?
        (18 characters)

buf1 after strcat of buf2:
        We're hereWhere is the time?
        (28 characters)

buf1 after strcat of "Why?":
        We're hereWhere is the time?Why?
        (32 characters)
```

strchr

説明： 文字列を検索し、指定した文字が最初に現れるのを見つけます。

インクルード： <string.h>

プロトタイプ： char *strchr(const char *s, int c);

引数： *s* 文字列を指すポインタ

c 見つける文字

戻り値： 見つかった場合、一致した対象のロケーションを指すポインタを返します。その他の場合は null ポインタを返します。

備考： この関数は文字列 s を検索し、最初に現れる文字 c を見つけます。

例：

```
#include <string.h> /* for strchr, NULL */
#include <stdio.h>  /* for printf       */

int main(void)
{
  char buf1[50] = "What time is it?";
  char ch1 = 'm', ch2 = 'y';
  char *ptr;
  int res;

  printf("buf1 : %s¥n¥n", buf1);

  ptr = strchr(buf1, ch1);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d¥n", ch1, res);
  }
  else
    printf("%c not found¥n", ch1);

  printf("¥n");

  ptr = strchr(buf1, ch2);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d¥n", ch2, res);
  }
  else
    printf("%c not found¥n", ch2);
}
```

出力：

```
buf1 : What time is it?

m found at position 8

y not found
```

strcmp

説明： 2 つの文字列を比較します。

インクルード： <string.h>

プロトタイプ： int strcmp(const char *s1, const char *s2);

引数： *s1* 最初の文字列

s2 2 つ目の文字列

戻り値： *s1* が *s2* より大きい場合は正の数値を、*s1* と *s2* が等しい場合はゼロを、*s1* が *s2* より小さい場合は負の数値を、それぞれ返します。

備考： この関数は、*s1* と *s2* の連続した文字を比較し、不一致または null ターミネータに遭遇したとき比較を停止します。

例：

```
#include <string.h> /* for strcmp */
#include <stdio.h>  /* for printf */

int main(void)
{
  char buf1[50] = "Where is the time?";
  char buf2[50] = "Where did they go?";
  char buf3[50] = "Why?";
  int res;

  printf("buf1 : %s¥n", buf1);
  printf("buf2 : %s¥n", buf2);
  printf("buf3 : %s¥n¥n", buf3);

  res = strcmp(buf1, buf2);
  if (res < 0)
    printf("buf1 comes before buf2¥n");
  else if (res == 0)
    printf("buf1 and buf2 are equal¥n");
  else
    printf("buf2 comes before buf1¥n");

  printf("¥n");

  res = strcmp(buf1, buf3);
  if (res < 0)
    printf("buf1 comes before buf3¥n");
  else if (res == 0)
    printf("buf1 and buf3 are equal¥n");
  else
    printf("buf3 comes before buf1¥n");

  printf("¥n");

  res = strcmp("Why?", buf3);
  if (res < 0)
    printf("¥"Why?¥" comes before buf3¥n");
  else if (res == 0)
    printf("¥"Why?¥" and buf3 are equal¥n");
  else
    printf("buf3 comes before ¥"Why?¥"¥n");
}
```

strcmp (続き)

出力：

```
buf1 : Where is the time?
buf2 : Where did they go?
buf3 : Why?

buf2 comes before buf1

buf1 comes before buf3

"Why?" and buf3 are equal
```

strcoll

説明： 1 つの文字列を別の文字列と比較します (備考参照)。

インクルード： <string.h>

プロトタイプ： int strcoll(const char *s1, const char *s2);

引数： *s1* 最初の文字列

s2 2 つ目の文字列

戻り値： ロケイル依存規則を使って、*s1* が *s2* より大きい場合は正の数値を、*s1* と *s2* が等しい場合はゼロを、*s1* が *s2* より小さい場合は負の数値を、それぞれ返します。

備考： MPLAB C30 は代替ロケイルをサポートしていないため、この関数は strcmp と等価になります。

strcpy

説明： ソース文字列をディステネーション文字列へコピーします。

インクルード： <string.h>

プロトタイプ： char *strcpy(char *s1, const char *s2);

引数： *s1* コピー先のディステネーション文字列

s2 コピー元のソース文字列

戻り値： ディステネーション文字列を指すポインタを返します。

備考： *s2* 内の null 終了文字を含むすべての文字がコピーされます。文字列が重複している場合は、動作は不定になります。

例：

```
#include <string.h> /* for strcpy, strlen */
#include <stdio.h>  /* for printf         */

int main(void)
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";

  printf("buf1 : %s¥n", buf1);
  printf("buf2 : %s¥n", buf2);
  printf("buf3 : %s¥n¥n", buf3);

  strcpy(buf1, buf2);
  printf("buf1 after strcpy of buf2: ¥n¥t%s¥n¥n",
         buf1);
```

strcpy

```
  strcpy(buf1, buf3);
  printf("buf1 after strcpy of buf3: ¥n¥t%s¥n",
          buf1);
}
```

出力：

```
buf1 : We're here
buf2 : Where is the time?
buf3 : Why?

buf1 after strcpy of buf2:
        Where is the time?

buf1 after strcpy of buf3:
        Why?
```

strcspn

説明： 文字列の先頭の連続する文字 (ある文字セットに含まれない) の数を計算します。

インクルード： <string.h>

プロトタイプ： size_t strcspn(const char *s1, const char *s2);

引数： *s1* 検索する文字列を指すポインタ

s2 探す文字を指すポインタ

戻り値： *s1* 内にあるセグメントで、かつ *s2* に指定された文字を含まないセグメントの長さを返します。

備考： この関数は、*s1* の先頭から連続する文字で、かつその中に *s2* に指定された文字を含まない連続文字内の字数を求めます。

例：

```
#include <string.h> /* for strcspn */
#include <stdio.h>  /* for printf  */

int main(void)
{
  char str1[20] = "hello";
  char str2[20] = "aeiou";
  char str3[20] = "animal";
  char str4[20] = "xyz";
  int res;

  res = strcspn(str1, str2);
  printf("strcspn(¥"%s¥", ¥"%s¥") = %d¥n",
          str1, str2, res);

  res = strcspn(str3, str2);
  printf("strcspn(¥"%s¥", ¥"%s¥") = %d¥n",
          str3, str2, res);

  res = strcspn(str3, str4);
  printf("strcspn(¥"%s¥", ¥"%s¥") = %d¥n",
          str3, str4, res);
}
```

出力：

```
strcspn("hello", "aeiou") = 1
strcspn("animal", "aeiou") = 0
strcspn("animal", "xyz") = 6
```

strcspn (続き)

説明：
最初の結果では、e は *s2* 内にあるため h の後カウントを停止します。
2 番目の結果では a は *s2* 内にあります。
3 番目の結果では、*s1* 内のどの文字も *s2* 内にないため、すべての文字がカウントされます。

strerror

説明： 内部エラー・メッセージを取得します。

インクルード： `<string.h>`

プロトタイプ： `char *strerror(int errcode);`

引数： *errcode*　エラー・コード番号

戻り値： 指定されたエラー・コード *errcode* に対応する内部エラー・メッセージ文字列を指すポインタを返します。

備考： strerror により指定された配列は、この関数に対する後続のコールにより上書きされます。

例：

```
#include <stdio.h>  /* for fopen, fclose, */
                    /* printf, FILE, NULL */
#include <string.h> /* for strerror       */
#include <errno.h>  /* for errno          */

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("samp.fil", "r+")) == NULL)
    printf("Cannot open samp.fil: %s¥n",
           strerror(errno));
  else
    printf("Success opening samp.fil¥n");
  fclose(myfile);
}
```

出力：

```
Cannot open samp.fil: file open error
```

strlen

説明： 文字列の長さを求めます。

インクルード： `<string.h>`

プロトタイプ： `size_t strlen(const char *s);`

引数： *s*　文字列

戻り値： 文字列の長さを返します。

備考： この関数は、終了 null 文字を含まない文字列の長さを求めます。

strlen (続き)

例：

```
#include <string.h> /* for strlen */
#include <stdio.h>  /* for printf */

int main(void)
{
  char str1[20] = "We are here";
  char str2[20] = "";
  char str3[20] = "Why me?";

  printf("str1 : %s¥n", str1);
  printf("¥t(string length = %d characters)¥n¥n",
         strlen(str1));
  printf("str2 : %s¥n", str2);
  printf("¥t(string length = %d characters)¥n¥n",
         strlen(str2));
  printf("str3 : %s¥n", str3);
  printf("¥t(string length = %d characters)¥n¥n¥n",
         strlen(str3));

}
```

出力：

```
str1 : We are here
        (string length = 11 characters)

str2 :
        (string length = 0 characters)

str3 : Why me?
        (string length = 7 characters)
```

strncat

説明： 指定された数の文字をソース文字列からディステネーション文字列へアペンドします。

インクルード： <string.h>

プロトタイプ： char *strncat(char *s1, const char *s2, size_t n);

引数：

s1 コピー先のディステネーション文字列

s2 コピー元のソース文字列

n アペンドする文字数

戻り値： ディステネーション文字列を指すポインタを返します。

備考： この関数は、最大 n 個の文字 (null 文字とその後ろの文字はアペンドしません) をソース文字列からディステネーション文字列の終わりにコピーします (アペンドします)。null 文字に遭遇しない場合は、終了 null 文字が結果にアペンドされます。文字列が重複している場合は、動作は不定になります。

例：

```
#include <string.h> /* for strncat, strlen */
#include <stdio.h>  /* for printf          */

int main(void)
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";
```

strncat (続き)

```
  printf("buf1 : %s¥n", buf1);
  printf("¥t(%d characters)¥n¥n", strlen(buf1));
  printf("buf2 : %s¥n", buf2);
  printf("¥t(%d characters)¥n¥n", strlen(buf2));
  printf("buf3 : %s¥n", buf3);
  printf("¥t(%d characters)¥n¥n¥n", strlen(buf3));

  strncat(buf1, buf2, 6);
  printf("buf1 after strncat of 6 characters "
         "of buf2: ¥n¥t%s¥n", buf1);
  printf("¥t(%d characters)¥n", strlen(buf1));

  printf("¥n");

  strncat(buf1, buf2, 25);
  printf("buf1 after strncat of 25 characters "
         "of buf2: ¥n¥t%s¥n", buf1);
  printf("¥t(%d characters)¥n", strlen(buf1));

  printf("¥n");

  strncat(buf1, buf3, 4);
  printf("buf1 after strncat of 4 characters "
         "of buf3: ¥n¥t%s¥n", buf1);
  printf("¥t(%d characters)¥n", strlen(buf1));
}
```

出力：

```
buf1 : We're here
        (10 characters)

buf2 : Where is the time?
        (18 characters)

buf3 : Why?
        (4 characters)

buf1 after strncat of 6 characters of buf2:
        We're hereWhere
        (16 characters)

buf1 after strncat of 25 characters of buf2:
        We're hereWhere Where is the time?
        (34 characters)

buf1 after strncat of 4 characters of buf3:
   We're hereWhere Where is the time?Why?
        (38 characters)
```

strncmp

説明： 2 つの文字列を、指定された文字数まで比較します。

インクルード： <string.h>

プロトタイプ：
```
int strncmp(const char *s1, const char *s2,
 size_t n);
```

引数：

s1 最初の文字列

s2 2 つ目の文字列

n 比較する文字数

戻り値： *s1* が *s2* より大きい場合は正の数値を、*s1* と *s2* が等しい場合はゼロを、*s1* が *s2* より小さい場合は負の数値を、それぞれ返します。

備考： strncmp は、*s1* と *s2* の間で異なる最初の文字に基づいて値を返します。null 文字の後ろの文字は比較しません。

例：
```
#include <string.h> /* for strncmp */
#include <stdio.h>  /* for printf  */

int main(void)
{
  char buf1[50] = "Where is the time?";
  char buf2[50] = "Where did they go?";
  char buf3[50] = "Why?";
  int res;

  printf("buf1 : %s¥n", buf1);
  printf("buf2 : %s¥n", buf2);
  printf("buf3 : %s¥n¥n", buf3);

  res = strncmp(buf1, buf2, 6);
  if (res < 0)
    printf("buf1 comes before buf2¥n");
  else if (res == 0)
    printf("6 characters of buf1 and buf2 "
           "are equal¥n");
  else
   printf("buf2 comes before buf1¥n");

  printf("¥n");

  res = strncmp(buf1, buf2, 20);
  if (res < 0)
    printf("buf1 comes before buf2¥n");
  else if (res == 0)
    printf("20 characters of buf1 and buf2 "
           "are equal¥n");
  else
    printf("buf2 comes before buf1¥n");
```

strncmp (続き)

```
  printf("¥n");

  res = strncmp(buf1, buf3, 20);
  if (res < 0)
    printf("buf1 comes before buf3¥n");
  else if (res == 0)
    printf("20 characters of buf1 and buf3 "
           "are equal¥n");
else
    printf("buf3 comes before buf1¥n");
}
```

出力：

```
buf1 : Where is the time?
buf2 : Where did they go?
buf3 : Why?

6 characters of buf1 and buf2 are equal

buf2 comes before buf1

buf1 comes before buf3
```

strncpy

説明： ソース文字列内の文字を、指定された文字数だけディステネーション文字列へコピーします。

インクルード： <string.h>

プロトタイプ： char *strncpy(char *s1, const char *s2, size_t n);

引数：

s1 コピー先のディステネーション文字列

s2 コピー元のソース文字列

n コピーする文字数

戻り値： ディステネーション文字列を指すポインタを返します。

備考： ソース文字列から n 個の文字をディステネーション文字列へコピーします。ソース文字列が n 文字より短い場合、ディステネーションには n 文字になるまで null 文字が埋め込まれます。n 個の文字がコピーされ、かつ null 文字が検出されない場合は、ディステネーション文字列は null 文字で終了されません。文字列が重複している場合は、動作は不定になります。

例：

```
#include <string.h> /* for strncpy, strlen */
#include <stdio.h>  /* for printf          */

int main(void)
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";
  char buf4[7]  = "Where?";

  printf("buf1 : %s¥n", buf1);
  printf("buf2 : %s¥n", buf2);
  printf("buf3 : %s¥n", buf3);
  printf("buf4 : %s¥n", buf4);
```

strncpy (続き)

```
  strncpy(buf1, buf2, 6);
  printf("buf1 after strncpy of 6 characters "
         "of buf2: ¥n¥t%s¥n", buf1);
  printf("¥t( %d characters)¥n", strlen(buf1));

  printf("¥n");

  strncpy(buf1, buf2, 18);
  printf("buf1 after strncpy of 18 characters "
         "of buf2: ¥n¥t%s¥n", buf1);
  printf("¥t( %d characters)¥n", strlen(buf1));

  printf("¥n");

  strncpy(buf1, buf3, 5);
  printf("buf1 after strncpy of 5 characters "
         "of buf3: ¥n¥t%s¥n", buf1);
  printf("¥t( %d characters)¥n", strlen(buf1));

  printf("¥n");

  strncpy(buf1, buf4, 9);
  printf("buf1 after strncpy of 9 characters "
         "of buf4: ¥n¥t%s¥n", buf1);
  printf("¥t( %d characters)¥n", strlen(buf1));
}
```

出力：

```
buf1 : We're here
buf2 : Where is the time?
buf3 : Why?
buf4 : Where?
buf1 after strncpy of 6 characters of buf2:
        Where here
        ( 10 characters)

buf1 after strncpy of 18 characters of buf2:
        Where is the time?
        ( 18 characters)

buf1 after strncpy of 5 characters of buf3:
        Why?
        ( 4 characters)

buf1 after strncpy of 9 characters of buf4:
        Where?
        ( 6 characters)
```

strncpy (続き)

説明：
各バッファには、上記文字列が含まれ、その後ろに null 文字が続いて長さ 50 になっています。strlen を使うと、最初の null 文字までの文字列長 (null 文字は含まない) を求めることができます。
最初の例で、buf2 ("Where ") の 6 文字が buf1 ("We're ") の最初の 6 文字を置き換え、buf1 の残りの部分 ("here" + null 文字) は不変です。
2 番目の例では、18 文字が buf1 の最初の 18 文字を置き換え、残りは null 文字のままです。
3 番目の例では、buf3 の 5 文字 ("Why?" + null 終了文字) が buf1 の最初の 5 文字を置き換えています。buf1 は実際に内容を持つようになります ("Why?"、1 null 文字、" is the time?"、32 個の null 文字)。strlen は、4 文字を表示します。これは、最初の null 文字に到達したときに停止するためです。
4 番目の例では、buf4 は 7 文字だけであるため、strncpy は 2 個の null 文字を使って buf1 の最初の 9 文字を置き換えています。buf1 の結果は、6 文字 ("Where?")、3 個の null 文字、9 文字 ("the time?")、32 個の null 文字となります。

strpbrk

説明： 文字列を検索して、指定された文字セットの文字が最初に現れるのを見つけます。

インクルード： <string.h>

プロトタイプ： char *strpbrk(const char *s1, const char *s2);

引数：
s1 検索する文字列を指すポインタ
s2 探す文字を指すポインタ

戻り値： 見つかった場合、*s1* 内の一致した文字を指すポインタを返します。その他の場合は null ポインタを返します。

備考： この関数は *s1* を検索して、*s2* により指定された文字が最初に現れるのを見つけます。

例：

```
#include <string.h> /* for strpbrk, NULL */
#include <stdio.h>  /* for printf        */

int main(void)
{
  char str1[20] = "What time is it?";
  char str2[20] = "xyz";
  char str3[20] = "eou?";
  char *ptr;
  int res;

  printf("strpbrk(¥"%s¥", ¥"%s¥")¥n", str1, str2);
  ptr = strpbrk(str1, str2);
if (ptr != NULL)
  {
    res = ptr - str1 + 1;
    printf("match found at position %d¥n", res);
  }
  else
    printf("match not found¥n");
```

strpbrk (続き)

```
  printf("¥n");

  printf("strpbrk(¥"%s¥", ¥"%s¥")¥n", str1, str3);
  ptr = strpbrk(str1, str3);
if (ptr != NULL)
  {
    res = ptr - str1 + 1;
    printf("match found at position %d¥n", res);
  }
  else
    printf("match not found¥n");
}
```

出力：

```
strpbrk("What time is it?", "xyz")
match not found

strpbrk("What time is it?", "eou?")
match found at position 9
```

strrchr

説明： 文字列内で指定された文字が最後に現れるのを見つけます。

インクルード： <string.h>

プロトタイプ： char *strrchr(const char *s, int c);

引数： *s* 検索されるストリングを指すポインタ

c 見つける文字

戻り値： 見つけた場合、文字を指すポインタを返します。その他の場合は null ポインタを返します。

備考： この関数は終了 null 文字を含む文字列を検索し、文字 c が最後に現れるのを見つけます。

例：

```
#include <string.h> /* for strrchr, NULL */
#include <stdio.h>  /* for printf        */

int main(void)
{
  char buf1[50] = "What time is it?";
  char ch1 = 'm', ch2 = 'y';
  char *ptr;
  int res;

  printf("buf1 : %s¥n¥n", buf1);

  ptr = strrchr(buf1, ch1);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d¥n", ch1, res);
  }
  else
    printf("%c not found¥n", ch1);
```

strrchr (続き)

```
  printf("¥n");

  ptr = strrchr(buf1, ch2);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d¥n", ch2, res);
  }
  else
    printf("%c not found¥n", ch2);
}
```

出力：

```
buf1 : What time is it?

m found at position 8

y not found
```

strspn

説明： 文字列の先頭の連続する文字 (ある文字セットに含まれる) の数を計算します。

インクルード： <string.h>

プロトタイプ： size_t strspn(const char *s1, const char *s2);

引数： *s1* 検索する文字列を指すポインタ

s2 探す文字を指すポインタ

戻り値： *s1* の先頭から連続する文字で、かつその中に *s2* に指定された文字を含む連続文字内の字数を返します。

備考： この関数は、*s1* の文字が *s2* 内になとい検索を停止します。

例：

```
#include <string.h> /* for strspn */
#include <stdio.h>  /* for printf */

int main(void)
{
  char str1[20] = "animal";
  char str2[20] = "aeiounm";
  char str3[20] = "aimnl";
  char str4[20] = "xyz";
  int res;

  res = strspn(str1, str2);
  printf("strspn(¥"%s¥", ¥"%s¥") = %d¥n",
          str1, str2, res);

  res = strspn(str1, str3);
  printf("strspn(¥"%s¥", ¥"%s¥") = %d¥n",
          str1, str3, res);

  res = strspn(str1, str4);
  printf("strspn(¥"%s¥", ¥"%s¥") = %d¥n",
          str1, str4, res);
}
```

strspn (続き)

出力：

```
strspn("animal", "aeiounm") = 5
strspn("animal", "aimnl") = 6
strspn("animal", "xyz") = 0
```

説明：

最初の結果では l は *s2* 内にありません。
2 番目の結果では 終了 null は *s2* 内にありません。
3 番目の結果では、a が *s2* 内にないため、比較が停止します。

strstr

説明： 別の文字列内で、ある文字列が最初に現れるのを見つけます。

インクルード： `<string.h>`

プロトタイプ：

```
char *strstr(const char *s1, const char *s2);
```

引数：

s1 検索する文字列を指すポインタ

s2 見つける部分文字列を指すポインタ

戻り値： 見つかった場合、最初に一致した部分文字列を指すポインタを返します。その他の場合は null ポインタを返します。

備考： この関数は、文字列 *s1* 内で最初に現れる文字列 *s2* (null ターミネータを除く) を見つけます。*s2* が長さゼロの文字列を指す場合、*s1* が返されます。

例：

```
#include <string.h> /* for strstr, NULL */
#include <stdio.h>  /* for printf       */

int main(void)
{
  char str1[20] = "What time is it?";
  char str2[20] = "is";
  char str3[20] = "xyz";
  char *ptr;
  int res;

  printf("str1 : %s¥n", str1);
  printf("str2 : %s¥n", str2);
  printf("str3 : %s¥n¥n", str3);

  ptr = strstr(str1, str2);
if (ptr != NULL)
  {
   res = ptr - str1 + 1;
   printf("¥"%s¥" found at position %d¥n",
          str2, res);
  }
  else
    printf("¥"%s¥" not found¥n", str2);
```

strstr (続き)

```
  printf("¥n");

  ptr = strstr(str1, str3);
if (ptr != NULL)
  {
    res = ptr - str1 + 1;
    printf("¥"%s¥" found at position %d¥n",
           str3, res);
  }
  else
    printf("¥"%s¥" not found¥n", str3);
}
```

出力：

```
str1 : What time is it?
str2 : is
str3 : xyz

"is" found at position 11

"xyz" not found
```

strtok

説明： 指定されたデリミッタの代わりに null 文字を挿入して、文字列を複数の部分文字列すなわちトークンに分割します。

インクルード： <string.h>

プロトタイプ： char *strtok(char *s1, const char *s2);

引数：

s1 検索対象となる文字列 (null で終了) を指すポインタ

s2 見つける文字を指すポインタ (デリミッタとして使用)

戻り値： トークンの先頭文字 (*s2* の文字セット内に現れない、*s1* 内の最初の文字) を指すポインタを返します。トークンが見つからない場合は、null ポインタを返します。

備考： この関数を連続コールして、指定した文字を null 文字で置き換えることにより、文字列を複数の部分文字列 (すなわちトークン) に分割することができます。最初に、この関数を特定の文字列について起動します。この文字列は *s1* に渡します。次に、null 値を *s1* へ渡した状態でこの関数を起動することにより、直前のデリミッタから文字列の検査を続けさせることができます。

文字列 *s2* (デリミッタ) 内に現れるすべての先行する文字をスキップし、次に、*s2* (この文字セグメントがトークンになります) 内に現れないすべての文字をスキップして、次の文字を null 文字で上書きして、現在のトークンを終了させます。次に、関数 strtok がその後ろに続く文字を指すポインタを保存します。この後ろに続く文字から次の検索が開始しされます。strtok がデリミッタを検出する前に文字列の終わりを検出すると、現在のトークンが *s1* により指定される文字列の終わりまで拡張されます。これが strtok に対する最初のコールである場合、文字列を変更しません (null 文字を *s1* へ書き込みません)。*s2* に渡される文字セットは、strtok に対する各コールで同じである必要はありません。

最初のコールの後、strtok が *s1* に非 null パラメータを設定してコールされた場合、その文字列が検索される新しい文字列になります。前に検索された古い文字列は失われます。

strtok (続き)

例：

```
#include <string.h> /* for strtok, NULL */
#include <stdio.h> / * for printf       */

int main(void)
{
  char str1[30] = "Here, on top of the world!";
  char delim[5] = ", .";
  char *word;
  int x;

  printf("str1 : %s¥n", str1);
  x = 1;
  word = strtok(str1,delim);
  while (word != NULL)
  {
    printf("word %d: %s¥n", x++, word);
    word = strtok(NULL, delim);
  }
}
```

出力：

```
str1 : Here, on top of the world!
word 1: Here
word 2: on
word 3: top
word 4: of
word 5: the
word 6: world!
```

strxfrm

説明： ロケイル依存のルールを使って文字列を変換します (備考参照)。

インクルード： `<string.h>`

プロトタイプ：

```
size_t strxfrm(char *s1, const char *s2, size_t n);
```

引数：

s1 ディステネーション文字列

s2 変換されるソース文字列

n 変換する文字数

戻り値： 変換された文字列の長さを返します。ただし、終了 null 文字は長さに含みません。n がゼロの場合、文字列は変換されず (この場合、*s1* はポイント null)、*s2* の長さが返されます。

備考： 戻り値が n 以上の場合、*s1* の内容は不定になります。MPLAB C30 は代替ロケイルをサポートしていないので、この変換は `strcpy` と等価です。ただし、ディステネーション文字列の長さは、`n-1` 制限されます。

4.16 <TIME.H> 日付関数と時刻関数

ヘッダー・ファイル time.h は、型、マクロ、時刻を扱う関数で構成されています。

clock_t

説明： プロセッサの時刻を保存します。

インクルード： <time.h>

プロトタイプ：
```
typedef long clock_t
```

size_t

説明： sizeof オペレータの適用結果の型。

インクルード： <time.h>

struct tm

説明： 時刻と日付（カレンダ日付）の保持に使用する構造体。

インクルード： <time.h>

プロトタイプ：
```
struct tm {
int tm_sec;    /*seconds after the minute ( 0 to 61 ) */
               /* allows for up to two leap seconds */
int tm_min;    /* minutes after the hour ( 0 to 59 ) */
int tm_hour;   /* hours since midnight ( 0 to 23 ) */
int tm_mday;   /* day of month ( 1 to 31 ) */
int tm_mon;    /* month ( 0 to 11 where January = 0 ) */
int tm_year;   /* years since 1900 */
int tm_wday;   /* day of week ( 0 to 6 where Sunday = 0 ) */
int tm_yday;   /* day of year ( 0 to 365 where January 1 = 0 ) */
int tm_isdst; /* Daylight Savings Time flag */
}
```

備考： tm_isdst が正の値の場合、夏時間が有効です。ゼロの場合は、夏時間が無効です。負の値の場合、夏時間のステータスが不明です。

time_t

説明： カレンダ時刻値を表わします。

インクルード： <time.h>

プロトタイプ：
```
typedef long time_t
```

CLOCKS_PER_SEC

説明： 1 秒当たりのプロセッサ・クロック数。

インクルード： <time.h>

プロトタイプ：
```
#define CLOCKS_PER_SEC
```

値： 1

備考： MPLAB C30 は、実際の時間ではなくクロック・ティック数 (命令サイクル数) を返します。

NULL

説明： null ポインタ定数の値。

インクルード： `<time.h>`

asctime

説明： 時刻構造体を文字列へ変換します。

インクルード： `<time.h>`

プロトタイプ： `char *asctime(const struct tm *tptr);`

引数： *tptr* 時刻 / 日付構造体

戻り値： 次のフォーマットの文字列を指すポインタを返します。：
DDD MMM dd hh:mm:ss YYYY
DDD は曜日
MMM は月
dd は日
hh は時間
mm は分
ss は秒
YYYY は西暦年

例：

```
#include <time.h>  /* for asctime, tm */
#include <stdio.h> /* for printf      */

volatile int i;

int main(void)
{
  struct tm when;
  time_t whattime;

  when.tm_sec = 30;
  when.tm_min = 30;
  when.tm_hour = 2;
  when.tm_mday = 1;
  when.tm_mon = 1;
  when.tm_year = 103;

  whattime = mktime(&when);
  printf("Day and time is %s¥n", asctime(&when));
}
```

出力：
Day and time is Sat Feb 1 02:30:30 2003

clock

説明： プロセッサ時刻を計算します。

インクルード： `<time.h>`

プロトタイプ： `clock_t clock(void);`

戻り値： 経過したプロセッサ時間のクロック・ティック数を返します。

備考： ターゲット環境が経過プロセッサ時間を計測できない場合、この関数は -1 を返します (`clock_t. ( = (clock_t) -1)` としてキャスト)。デフォルトでは、MPLAB C30 は時間を命令サイクル数で返します。

clock (続き)

例：

```
#include <time.h>  /* for clock  */
#include <stdio.h> /* for printf */

volatile int i;

int main(void)
{
  clock_t start, stop;
  int ct;

  start = clock();
  for (i = 0; i < 10; i++)
  stop = clock();
  printf("start = %ld¥n", start);
  printf("stop = %ld¥n", stop);
}
```

出力：

```
start = 0
stop = 317
```

ctime

説明： カレンダ時間をローカル時間の文字列表現へ変換します。

インクルード： <time.h>

プロトタイプ： char *ctime(const time_t *tod);

引数： *tod* 保存された時間を指すポインタ

戻り値： 渡されたパラメータのローカル時間を表わす文字列のアドレスを返します。

備考： この関数は asctime(localtime(*tod*)) と等価です。

例：

```
#include <time.h>  /* for mktime, tm, ctime */
#include <stdio.h> /* for printf            */

int main(void)
{
  time_t whattime;
  struct tm nowtime;

  nowtime.tm_sec = 30;
  nowtime.tm_min = 30;
  nowtime.tm_hour = 2;
  nowtime.tm_mday = 1;
  nowtime.tm_mon = 1;
  nowtime.tm_year = 103;

  whattime = mktime(&nowtime);
  printf("Day and time %s¥n", ctime(&whattime));
}
```

出力：

```
Day and time Sat Feb 1 02:30:30 2003
```

difftime

説明： 2 つの時刻の差を求めます。

インクルード： <time.h>

プロトタイプ： double difftime(time_t *t1*, time_t *t0*);

引数： *t1* 終了時刻

t0 開始時刻

戻り値： *t1* と *t0* との間の秒数を返します。

備考： デフォルトでは、MPLAB C30 は時間を命令サイクル数で返します。したがって、difftime は *t1* と *t0* との間のティック数を返します。

例：

```
#include <time.h>  /* for clock, difftime */
#include <stdio.h> /* for printf          */

volatile int i;

int main(void)
{
  clock_t start, stop;
  double elapsed;

  start = clock();
  for (i = 0; i < 10; i++)
  stop = clock();
  printf("start = %ld¥n", start);
  printf("stop = %ld¥n", stop);
  elapsed = difftime(stop, start);
  printf("Elapsed time = %.0f¥n", elapsed);
}
```

出力：

```
start = 0
stop = 317
Elapsed time = 317
```

gmtime

説明： カレンダ時間を時刻構造体へ変換します。この時刻構造体は Universal Time Coordinated (UTC) 表わされ、Greenwich Mean Time (GMT) とも呼ばれています。

インクルード： <time.h>

プロトタイプ： struct tm *gmtime(const time_t *tod);

引数： *tod* 保存された時間を指すポインタ

戻り値： 時刻構造体のアドレスを返します。

備考： この関数は、tod 値を型 tm の時刻構造体に分割します。デフォルトでは、MPLAB C30 は時間を命令サイクル数で返します。このデフォルトでは、gmtime と localtime は等価です。ただし、gmtime は tm_isdst (夏時間フラグ) をゼロとして返して、夏時間が無効であることを表示します。

gmtime (続き)

例：

```
#include <time.h>  /* for gmtime, asctime, */
                   /* time_t, tm           */
#include <stdio.h> /* for printf           */

int main(void)
{
  time_t timer;
  struct tm *newtime;

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */

  newtime = gmtime(&timer);
  printf("UTC time = %s¥n", asctime(newtime));
}
```

出力：

```
UTC time = Mon Oct 20 16:43:02 2003
```

localtime

説明： 値をローカル時間へ変換します。

インクルード： `<time.h>`

プロトタイプ： `struct tm *localtime(const time_t *tod);`

引数： *tod* 保存された時間を指すポインタ

戻り値： 時刻構造体のアドレスを返します。

備考： デフォルトでは、MPLAB C30 は時間を命令サイクル数で返します。このデフォルトでは、localtime と gmtime は等価です。ただし、localtime は tm_isdst (夏時間フラグ) を -1 として返して、夏時間のステータスが不明であることを表示します。

例：

```
#include <time.h>  /* for localtime,      */
                   /* asctime, time_t, tm */
#include <stdio.h> /* for printf          */

int main(void)
{
  time_t timer;
  struct tm *newtime;

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */

  newtime = localtime(&timer);
  printf("Local time = %s¥n", asctime(newtime));
}
```

出力：

```
Local time = Mon Oct 20 16:43:02 2003
```

mktime

説明： ローカル時間をカレンダ値へ変換します。

インクルード： <time.h>

プロトタイプ： time_t mktime(struct tm *tptr);

引数： *tptr* 時刻構造体を指すポインタ

戻り値： time_t の値としてエンコードされたカレンダ時間を返します。

備考： カレンダ時間を表示できない場合、この関数は -1 (time_t (= (time_t) -1) へキャスト) を返します。

例：

```
#include <time.h>  /* for localtime,   */
                   /* asctime, mktime, */
                   /* time_t, tm       */
#include <stdio.h> /* for printf       */

int main(void)
{
  time_t timer, whattime;
  struct tm *newtime;

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */
  /* localtime allocates space for struct tm */
  newtime = localtime(&timer);
  printf("Local time = %s", asctime(newtime));

  whattime = mktime(newtime);
  printf("Calendar time as time_t = %ld¥n",
          whattime);
}
```

出力：

```
Local time = Mon Oct 20 16:43:02 2003
Calendar time as time_t = 1066668182
```

strftime

説明： フォーマット・パラメータに基づいて時刻構造体を文字列へフォーマット化します。

インクルード： <time.h>

プロトタイプ： size_t strftime(char *s, size_t n, const char *format, const struct tm *tptr);

引数：

s 出力文字列

n 最大文字列長

format フォーマット制御文字列

tptr tm データ構造体を指すポインタ

戻り値： 終了 null を含む合計が n より大きくない場合、配列 s 内に置かれた文字数を返します。その他の場合は、0 を返して、配列の値は不定になります。

備考： フォーマット・パラメータを次に示します：

%a 曜日の省略名

%A 曜日のフル名

%b 月の省略名

%B 月のフル名

%c 適切な日付と時刻表現

%d 日 (01 ～ 31)

strftime (続き)

%H 時間 (00 ～ 23)

%I 時間 (01 ～ 12)

%j 日 (001 ～ 366)

%m 月 (01 ～ 12)

%M 分 (00 ～ 59)

%p AM/PM の区別

%S 秒 (00 ～ 61)
最大 2 閏秒まで許容

%U 日曜日を 1 週目の初日とした通年週番号 (00 ～ 53)

%w 日曜日を 0 とした曜日 (0 ～ 6)

%W 月曜日を 1 週目の初日とした通年週番号 (00 ～ 53)

%x 適切な日付表現

%X 適切な時刻表現

%y 西暦年の下 2 桁 (00 ～ 99)

%Y 西暦年 4 桁

%Z タイム・ゾーン (多分省略型を使用)、またはタイム・ゾーンを使用しない場合は文字なし

%% パーセント文字 %

例：

```
#include <time.h>  /* for strftime, */
                   /* localtime,    */
                   /* time_t, tm    */
#include <stdio.h> /* for printf    */

int main(void)
{
  time_t timer, whattime;
  struct tm *newtime;
  char buf[128];

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */
  /* localtime allocates space for structure */
  newtime = localtime(&timer);

  strftime(buf, 128, "It was a %A, %d days into the "
           "month of %B in the year %Y.¥n", newtime);
  printf(buf);

  strftime(buf, 128, "It was %W weeks into the year "
           "or %j days into the year.¥n", newtime);
  printf(buf);
}
```

出力：

```
It was a Monday, 20 days into the month of October in the
year 2003.
It was 42 weeks into the year or 293 days into the year.
```

time

説明： 現在のカレンダ時間を計算します。

インクルード： `<time.h>`

プロトタイプ： `time_t time(time_t *tod);`

引数： *tod* 時間の格納場所を指すポインタ

戻り値： time_t の値としてエンコードされたカレンダ時間を返します。

備考： ターゲット環境が時間を決定できない場合、この関数は -1 を返します (clock_t. (= (clock_t) -1) としてキャスト)。デフォルトでは、MPLAB C30 は時間を命令サイクル数で返します。この関数はカスタマイズできます。pic30-libs を参照してください。

例：

```
#include <time.h>  /* for time   */
#include <stdio.h> /* for printf */

volatile int i;

int main(void)
{
  time_t ticks;

  time(0); /* start time */
  for (i = 0; i < 10; i++) /* waste time */
  time(&ticks); /* get time */
  printf("Time = %ld¥n", ticks);
}
```

出力：

```
Time = 256
```

4.17 <MATH.H> 算術関数

ヘッダー・ファイル math.h は、マクロと一般的な算術演算を行う種々の関数で構成されています。エラー状態は、領域エラーまたは範囲エラーにより処理されます (errno.h 参照)。

領域エラーは、関数が定義されている領域の外側に入力引数がある場合に発生します。このエラーは、EDOM の値を errno に格納し、各関数向けに定義された特定の値を返すことにより、報告されます。

範囲エラーは、結果が目標精度で表現できないほど大き過ぎるか小さ過ぎる場合に発生します。このエラーは、ERANGE の値を errno へ格納し、結果がオーバーフロー (戻り値が大き過ぎる) の場合は HUGE_VAL を、結果がアンダーフロー (戻り値が小さ過ぎる) の場合はゼロを、それぞれ返すことにより、報告されます。

NaN、ゼロ、無限などの特別な値に対する応答は、関数によって異なります。各関数の説明には、このような値に対する関数応答の定義が記載されています。

HUGE_VAL

説明 : HUGE_VAL は、範囲エラーが発生した場合に関数から返されます (例えば、関数がターゲット精度で表わすためには大き過ぎる値を返そうとする場合)。

インクルード : <math.h>

備考 : -HUGE_VAL は、関数の結果がターゲット精度で表わすためには負で大き過ぎる場合 (絶対値) に返されます。出力結果が +/- HUGE_VAL のとき、+/- inf で表わされます。

acos

説明 : 倍精度浮動小数値の arc cosine 三角関数を計算します。

インクルード : <math.h>

プロトタイプ : double acos (double *x*);

引数 : *x* arc cosine を計算する -1 ～ 1 の範囲の値

戻り値 : 0 ～ pi の範囲のラジアンで表わした arc cosine の値を返します。

備考 : *x* が -1 より小さいとき、または 1 より大きいとき、領域エラーが発生します。

例 :

```
#include <math.h>  /* for acos           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x,y;

  errno = 0;
  x = -2.0;
  y = acos (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f¥n¥n", x, y);
```

acos (続き)

```
 errno = 0;
 x = 0.10;
 y = acos (x);
 if (errno)
  perror("Error");
 printf("The arccosine of %f is %f¥n¥n", x, y);
}
```

出力：

```
Error: domain error
The arccosine of -2.000000 is nan

The arccosine of 0.100000 is 1.470629
```

acosf

説明： 単精度浮動小数値の arc cosine 三角関数を計算します。

インクルード： <math.h>

プロトタイプ： float acosf (float *x*);

引数： *x* -1 ～ 1 の範囲の値

戻り値： 0 ～ pi の範囲のラジアンで表わした arc cosine の値を返します。

備考： *x* が -1 より小さいとき、または 1 より大きいとき、領域エラーが発生します。

例：

```
#include <math.h>  /* for acosf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = 2.0F;
  y = acosf (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f¥n¥n", x, y);

  errno = 0;
  x = 0.0F;
  y = acosf (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f¥n", x, y);
}
```

出力：

```
Error: domain error
The arccosine of 2.000000 is nan

The arccosine of 0.000000 is 1.570796
```

asin

説明： 倍精度浮動小数値の arc sine 三角関数を計算します。

インクルード： <math.h>

プロトタイプ： double asin (double *x*);

引数： *x* arc sine を計算する -1 ～ 1 の範囲の値

戻り値： -pi/2 ～ +pi/2 の範囲のラジアンで表わした arc sine の値を返します。

備考： *x* が -1 より小さいとき、または 1 より大きいとき、領域エラーが発生します。

例：

```
#include <math.h>  /* for asin           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = 2.0;
  y = asin (x);
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f¥n¥n", x, y);

  errno = 0;
  x = 0.0;
  y = asin (x);
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f¥n¥n", x, y);
}
```

出力：

```
Error: domain error
The arcsine of 2.000000 is nan

The arcsine of 0.000000 is 0.000000
```

asinf

説明： 単精度浮動小数値の arc sine 三角関数を計算します。

インクルード： <math.h>

プロトタイプ： float asinf (float *x*);

引数： *x* -1 ～ 1 の範囲の値

戻り値： -pi/2 ～ +pi/2 の範囲のラジアンで表わした arc sine の値を返します。

備考： *x* が -1 より小さいとき、または 1 より大きいとき、領域エラーが発生します。

例：

```
#include <math.h>  /* for asinf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;
```

asinf (続き)

```
  errno = 0;
  x = 2.0F;
  y = asinf(x);
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f¥n¥n", x, y);

  errno = 0;
  x = 0.0F;
  y = asinf(x);
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f¥n¥n", x, y);
}
```

出力：

```
Error: domain error
The arcsine of 2.000000 is nan

The arcsine of 0.000000 is 0.000000
```

atan

説明： 倍精度浮動小数値の arc tangent 三角関数を計算します。

インクルード： <math.h>

プロトタイプ： double atan (double *x*);

引数： *x* arc tangent を計算する値

戻り値： -pi/2 ～ +pi/2 の範囲のラジアンで表わした arc tangent の値を返します。

備考： 領域エラーまたは範囲エラーは発生しません。

例：

```
#include <math.h>  /* for atan   */
#include <stdio.h> /* for printf */

int main(void)
{
  double x, y;

  x = 2.0;
  y = atan (x);
  printf("The arctangent of %f is %f¥n¥n", x, y);

  x = -1.0;
  y = atan (x);
  printf("The arctangent of %f is %f¥n¥n", x, y);
}
```

出力：

```
The arctangent of 2.000000 is 1.107149

The arctangent of -1.000000 is -0.785398
```

atanf

説明： 単精度浮動小数値の arc tangent 三角関数を計算します。

インクルード： <math.h>

プロトタイプ： float atanf (float *x*);

引数： *x* arc tangent を計算する値

戻り値： -pi/2 ～ +pi/2 の範囲のラジアンで表わした arc tangent の値を返します。

備考： 領域エラーまたは範囲エラーは発生しません。

例：

```
#include <math.h>  /* for atanf  */
#include <stdio.h> /* for printf */

int main(void)
{
  float x, y;

  x = 2.0F;
  y = atanf (x);
  printf("The arctangent of %f is %f¥n¥n", x, y);

  x = -1.0F;
  y = atanf (x);
  printf("The arctangent of %f is %f¥n¥n", x, y);
}
```

出力：

```
The arctangent of 2.000000 is 1.107149

The arctangent of -1.000000 is -0.785398
```

atan2

説明： y/*x* の arc tangent 三角関数を計算します。

インクルード： <math.h>

プロトタイプ： double atan2 (double *y*, double *x*);

引数： *y* arc tangent を計算する y 値

x arc tangent を計算する x 値

戻り値： -pi ～ +pi の範囲のラジアンで表わした arc tangent の値を返します。両パラメータの符号により象限が決定されます。

備考： *x* と y がゼロの場合、または x と y が +/- 無限の場合、領域エラーが発生します。

例：

```
#include <math.h>  /* for atan2          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y, z;
```

atan2 (続き)

```
  errno = 0;
  x = 0.0;
  y = 2.0;
  z = atan2(y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f¥n¥n",
          y, x, z);

  errno = 0;
  x = -1.0;
  y = 0.0;
  z = atan2(y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f¥n¥n",
          y, x, z);

  errno = 0;
  x = 0.0;
  y = 0.0;
  z = atan2(y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f¥n¥n",
          y, x, z);
}
```

出力：

```
The arctangent of 2.000000/0.000000 is 1.570796

The arctangent of 0.000000/-1.000000 is 3.141593

Error: domain error
The arctangent of 0.000000/0.000000 is nan
```

atan2f

説明： y/*x* の arc tangent 三角関数を計算します。

インクルード： <math.h>

プロトタイプ： float atan2f (float *y*, float *x*);

引数： *y* arc tangent を計算する y 値

x arc tangent を計算する x 値

戻り値： -pi ～ +pi の範囲のラジアンで表わした arc tangent の値を返します。両パラメータの符号により象限が決定されます。

備考： *x* と y がゼロの場合、または x と y が +/- 無限の場合、領域エラーが発生します。

例：

```
#include <math.h>  /* for atan2f         */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y, z;

  errno = 0;
  x = 2.0F;
  y = 0.0F;
  z = atan2f (y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f¥n¥n",
          y, x, z);

  errno = 0;
  x = 0.0F;
  y = -1.0F;
  z = atan2f (y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f¥n¥n",
          y, x, z);

  errno = 0;
  x = 0.0F;
  y = 0.0F;
  z = atan2f (y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f¥n¥n",
          y, x, z);
}
```

出力：

```
The arctangent of 2.000000/0.000000 is 1.570796

The arctangent of 0.000000/-1.000000 is 3.141593

Error: domain error
The arctangent of 0.000000/0.000000 is nan
```

ceil

説明： 値の上限値を計算します。

インクルード： <math.h>

プロトタイプ： double ceil(double *x*);

引数： *x* 上限値を計算する浮動小数値

戻り値： *x* 以上の最小整数値を返します。

備考： 領域エラーまたは範囲エラーは発生しません。下限値を参照してください。

例：

```
#include <math.h>  /* for ceil   */
#include <stdio.h> /* for printf */

int main(void)
{
  double x[8] = {2.0, 1.75, 1.5, 1.25, -2.0,
                 -1.75, -1.5, -1.25};
  double y;
  int i;

  for (i=0; i<8; i++)
  {
    y = ceil (x[i]);
    printf("The ceiling for  %f is  %f¥n", x[i], y);
  }
}
```

出力：

```
The ceiling for 2.000000 is 2.000000
The ceiling for 1.750000 is 2.000000
The ceiling for 1.500000 is 2.000000
The ceiling for 1.250000 is 2.000000
The ceiling for -2.000000 is -2.000000
The ceiling for -1.750000 is -1.000000
The ceiling for -1.500000 is -1.000000
The ceiling for -1.250000 is -1.000000
```

ceilf

説明： 値の上限値を計算します。

インクルード： <math.h>

プロトタイプ： float ceilf(float *x*);

引数： *x* 浮動小数値

戻り値： *x* 以上の最小整数値を返します。

備考： 領域エラーまたは範囲エラーは発生しません。下限値を参照してください。

例：

```
#include <math.h>  /* for ceilf  */
#include <stdio.h> /* for printf */

int main(void)
{
  float x[8] = {2.0F, 1.75F, 1.5F, 1.25F,
                -2.0F, -1.75F, -1.5F, -1.25F};
  float y;
  int i;

  for (i=0; i<8; i++)
  {
    y = ceilf (x[i]);
    printf("The ceiling for  %f is  %f¥n", x[i], y);
  }
}
```

出力：

```
The ceiling for 2.000000 is 2.000000
The ceiling for 1.750000 is 2.000000
The ceiling for 1.500000 is 2.000000
The ceiling for 1.250000 is 2.000000
The ceiling for -2.000000 is -2.000000
The ceiling for -1.750000 is -1.000000
The ceiling for -1.500000 is -1.000000
The ceiling for -1.250000 is -1.000000
```

cos

説明： 倍精度浮動小数値の cosine 三角関数を計算します。

インクルード： <math.h>

プロトタイプ： double cos (double *x*);

引数： *x* cosine を計算する値

戻り値： -1 ～ 1 の範囲のラジアンで表わした x の cosine 値を返します。

備考： *x* が NaN の場合、または無限の場合、領域エラーが発生します。

例：

```
#include <math.h>  /* for cos            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x,y;

  errno = 0;
  x = -1.0;
  y = cos (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f¥n¥n", x, y);
```

cos (続き)

```
  errno = 0;
  x = 0.0;
  y = cos (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f¥n¥n", x, y);
}
```

出力：

```
The cosine of -1.000000 is 0.540302

The cosine of 0.000000 is 1.000000
```

cosf

説明： 単精度浮動小数値の cosine 三角関数を計算します。

インクルード： <math.h>

プロトタイプ： float cosf (float x);

引数： x cosine を計算する値

戻り値： -1 ～ 1 の範囲のラジアンで表わした x の cosine 値を返します。

備考： x が NaN の場合、または無限の場合、領域エラーが発生します。

例：

```
#include <math.h>  /* for cosf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = cosf (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f¥n¥n", x, y);

  errno = 0;
  x = 0.0F;
  y = cosf (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f¥n¥n", x, y);
}
```

出力：

```
The cosine of -1.000000 is 0.540302

The cosine of 0.000000 is 1.000000
```

cosh

説明： 倍精度浮動小数値の双曲 cosine 関数を計算します。

インクルード： <math.h>

プロトタイプ： double cosh (double *x*);

引数： *x* 双曲 cosine を計算する値

戻り値： *x* の双曲 cosine を返します。

備考： *x* が大きい過ぎる場合、範囲エラーが発生します。

例：

```
#include <math.h>  /* for cosh           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.5;
  y = cosh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f¥n¥n",
          x, y);

  errno = 0;
  x = 0.0;
  y = cosh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f¥n¥n",
          x, y);

  errno = 0;
  x = 720.0;
  y = cosh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f¥n¥n",
          x, y);
}
```

出力：

```
The hyperbolic cosine of -1.500000 is 2.352410

The hyperbolic cosine of 0.000000 is 1.000000

Error: range error
The hyperbolic cosine of 720.000000 is inf
```

coshf

説明： 単精度浮動小数値の双曲 cosine 関数を計算します。

インクルード： <math.h>

プロトタイプ： float coshf (float *x*);

引数： *x* 双曲 cosine を計算する値

戻り値： *x* の双曲 cosine を返します。

備考： *x* が大きい過ぎる場合、範囲エラーが発生します。

例：

```
#include <math.h>  /* for coshf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = coshf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f¥n¥n",
         x, y);

  errno = 0;
  x = 0.0F;
  y = coshf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f¥n¥n",
         x, y);

  errno = 0;
  x = 720.0F;
  y = coshf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f¥n¥n",
         x, y);
}
```

出力：

```
The hyperbolic cosine of -1.000000 is 1.543081

The hyperbolic cosine of 0.000000 is 1.000000

Error: range error
The hyperbolic cosine of 720.000000 is inf
```

exp

説明： *x* の指数関数を計算します (e の x 乗、ここで x は倍精度浮動小数値)。

インクルード： <math.h>

プロトタイプ： double exp (double *x*);

引数： *x* 指数を計算する x 値

戻り値： *x* 乗を返します。オーバーフローでは inf を、アンダーフローでは 0 を、それぞれ返します。

備考： *x* が大き過ぎる場合、範囲エラーが発生します。

例：

```
#include <math.h>  /* for exp            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = 1.0;
  y = exp (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f¥n¥n", x, y);

  errno = 0;
  x = 1E3;
  y = exp (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f¥n¥n", x, y);

  errno = 0;
  x = -1E3;
  y = exp (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f¥n¥n", x, y);
}
```

出力：

```
The exponential of 1.000000 is 2.718282

Error: range error
The exponential of 1000.000000 is inf

Error: range error
The exponential of -1000.000000 is 0.000000
```

expf

説明： *x* の指数関数を計算します (e の x 乗、ここで x は単精度浮動小数値)。

インクルード： <math.h>

プロトタイプ： float expf (float *x*);

引数： *x* 指数を計算する浮動小数値

戻り値： *x* 乗を返します。オーバーフローでは inf を、アンダーフローでは 0 を、それぞれ返します。

備考： *x* が大き過ぎる場合、範囲エラーが発生します。

例：

```
#include <math.h>  /* for expf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = 1.0F;
  y = expf (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f¥n¥n", x, y);

  errno = 0;
  x = 1.0E3F;
  y = expf (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f¥n¥n", x, y);

  errno = 0;
  x = -1.0E3F;
  y = expf (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f¥n¥n", x, y);
}
```

出力：

```
The exponential of 1.000000 is 2.718282

Error: range error
The exponential of 1000.000000 is inf

Error: range error
The exponential of -1000.000000 is 0.000000
```

fabs

説明： 倍精度浮動小数値の絶対値を計算します。

インクルード： <math.h>

プロトタイプ： double fabs(double *x*);

引数： *x* 絶対値を計算する浮動小数値

戻り値： *x* の絶対値を返します (負の数値は正として返されます。正の数値は変更されません)。

備考： 領域エラーまたは範囲エラーは発生しません。

例：

```
#include <math.h>  /* for fabs   */
#include <stdio.h> /* for printf */

int main(void)
{
  double x, y;

  x = 1.75;
  y = fabs (x);
  printf("The absolute value of  %f is  %f¥n", x, y);

  x = -1.5;
  y = fabs (x);
  printf("The absolute value of %f is  %f¥n", x, y);
}
```

出力：

```
The absolute value of 1.750000 is 1.750000
The absolute value of -1.500000 is 1.500000
```

fabsf

説明： 単精度浮動小数値の絶対値を計算します。

インクルード： <math.h>

プロトタイプ： float fabsf(float *x*);

引数： *x* 絶対値を計算する浮動小数値

戻り値： *x* の絶対値を返します (負の数値は正として返されます。正の数値は変更されません)。

備考： 領域エラーまたは範囲エラーは発生しません。

例：

```
#include <math.h>  /* for fabsf  */
#include <stdio.h> /* for printf */

int main(void)
{
  float x,y;

  x = 1.75F;
  y = fabsf (x);
  printf("The absolute value of  %f is  %f¥n", x, y);

  x = -1.5F;
  y = fabsf (x);
  printf("The absolute value of %f is  %f¥n", x, y);
}
```

出力：

```
The absolute value of 1.750000 is 1.750000
The absolute value of -1.500000 is 1.500000
```

floor

説明： 倍精度浮動小数値の下限値を計算します。

インクルード： <math.h>

プロトタイプ： double floor (double *x*);

引数： *x* 下限値を計算する浮動小数値

戻り値： *x* 以下の最大整数値を返します。

備考： 領域エラーまたは範囲エラーは発生しません。ceil を参照してください。

例：

```
#include <math.h>  /* for floor  */
#include <stdio.h> /* for printf */

int main(void)
{
  double x[8] = {2.0, 1.75, 1.5, 1.25, -2.0,
                 -1.75, -1.5, -1.25};
  double y;
  int i;

  for (i=0; i<8; i++)
  {
    y = floor (x[i]);
    printf("The ceiling for %f is %f¥n", x[i], y);
  }
}
```

出力：

```
The floor for 2.000000 is 2.000000
The floor for 1.750000 is 1.000000
The floor for 1.500000 is 1.000000
The floor for 1.250000 is 1.000000
The floor for -2.000000 is -2.000000
The floor for -1.750000 is -2.000000
The floor for -1.500000 is -2.000000
The floor for -1.250000 is -2.000000
```

floorf

説明： 単精度浮動小数値の下限値を計算します。

インクルード： <math.h>

プロトタイプ： float floorf(float *x*);

引数： *x* 浮動小数値

戻り値： *x* 以下の最大整数値を返します。

備考： 領域エラーまたは範囲エラーは発生しません。ceil を参照してください。

floorf (続き)

例：

```
#include <math.h>  /* for floorf */
#include <stdio.h> /* for printf */

int main(void)
{
  float x[8] = {2.0F, 1.75F, 1.5F, 1.25F,
                -2.0F, -1.75F, -1.5F, -1.25F};
  float y;
  int i;

  for (i=0; i<8; i++)
  {
    y = floorf (x[i]);
    printf("The floor for  %f is  %f¥n", x[i], y);
  }
}
```

出力：

```
The floor for 2.000000 is 2.000000
The floor for 1.750000 is 1.000000
The floor for 1.500000 is 1.000000
The floor for 1.250000 is 1.000000
The floor for -2.000000 is -2.000000
The floor for -1.750000 is -2.000000
The floor for -1.500000 is -2.000000
The floor for -1.250000 is -2.000000
```

fmod

説明： *x*/y の余りを倍精度値として計算します。

インクルード： <math.h>

プロトタイプ： double fmod(double *x*, double *y*);

引数： *x* 倍精度浮動小数値。

y 倍精度浮動小数値。

戻り値： *x*/*y* の余りを返します。

備考： *y* = 0 の場合、領域エラーが発生します。y が非ゼロの場合、結果は x と同じ符号を持ち、結果の絶対値は y の絶対値より小さくなります。

例：

```
#include <math.h>  /* for fmod           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x,y,z;

  errno = 0;
  x = 7.0;
  y = 3.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f¥n¥n",
         x, y, z);
```

fmod (続き)

```
  errno = 0;
  x = 7.0;
  y = 7.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f¥n¥n",
         x, y, z);

  errno = 0;
  x = -5.0;
  y = 3.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f¥n¥n",
         x, y, z);

  errno = 0;
  x = 5.0;
  y = -3.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f¥n¥n",
         x, y, z);

  errno = 0;
  x = -5.0;
  y = -5.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f¥n¥n",
         x, y, z);

 errno = 0;
  x = 7.0;
  y = 0.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f¥n¥n",
         x, y, z);
}
```

出力：

```
For fmod(7.000000, 3.000000) the remainder is 1.000000

For fmod(7.000000, 7.000000) the remainder is 0.000000

For fmod(-5.000000, 3.000000) the remainder is -2.000000

For fmod(5.000000, -3.000000) the remainder is 2.000000

For fmod(-5.000000, -5.000000) the remainder is -0.000000

Error: domain error
For fmod(7.000000, 0.000000) the remainder is nan
```

fmodf

説明： *x*/y の余りを単精度値として計算します。

インクルード： <math.h>

プロトタイプ： float fmodf(float *x*, float *y*);

引数： *x* 単精度浮動小数値

y 単精度浮動小数値

戻り値： *x*/*y* の余りを返します。

備考： *y* = 0 の場合、領域エラーが発生します。y が非ゼロの場合、結果は x と同じ符号を持ち、結果の絶対値は y の絶対値より小さくなります。

例：

```
#include <math.h>  /* for fmodf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x,y,z;

  errno = 0;
  x = 7.0F;
  y = 3.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
         " %f¥n¥n", x, y, z);

  errno = 0;
  x = -5.0F;
  y = 3.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
     printf("For fmodf (%f, %f) the remainder is"
         " %f¥n¥n", x, y, z);

  errno = 0;
  x = 5.0F;
  y = -3.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
         " %f¥n¥n", x, y, z);

  errno = 0;
  x = 5.0F;
  y = -5.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
         " %f¥n¥n", x, y, z);
```

fmodf (続き)

```
  errno = 0;
  x = 7.0F;
  y = 0.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
          " %f¥n¥n", x, y, z);


  errno = 0;
  x = 7.0F;
  y = 7.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
         " %f¥n¥n", x, y, z);
}
```

出力：

```
For fmodf (7.000000, 3.000000) the remainder is 1.000000

For fmodf (-5.000000, 3.000000) the remainder is -2.000000

For fmodf (5.000000, -3.000000) the remainder is 2.000000

For fmodf (5.000000, -5.000000) the remainder is 0.000000

Error: domain error
For fmodf (7.000000, 0.000000) the remainder is nan

For fmodf (7.000000, 7.000000) the remainder is 0.000000
```

frexp

説明： 倍精度浮動小数値の小数部と指数部を取得します。

インクルード： <math.h>

プロトタイプ： double frexp (double *x*, int *exp);

引数：

x	小数部と指数部を求める浮動小数値
*exp	格納された整数の指数部を指すポインタ

戻り値： 小数部を返します。exp は指数部を指します。x が 0 の場合、関数は、小数部と指数部の両方に 0 を返します。

備考： 小数部の絶対値は、1/2 ～ 1 (1 は含まない) の範囲。領域エラーまたは範囲エラーは発生しません。

例：

```
#include <math.h>  /* for frexp  */
#include <stdio.h> /* for printf */

int main(void)
{
  double x,y;
  int n;
```

frexp (続き)

```
  x = 50.0;
  y = frexp (x, &n);
  printf("For frexp of %f¥n the fraction is %f¥n ",
         x, y);
  printf("  and the exponent is %d¥n¥n", n);

  x = -2.5;
  y = frexp (x, &n);
  printf("For frexp of %f¥n the fraction is %f¥n ",
         x, y);
  printf("  and the exponent is %d¥n¥n", n);

  x = 0.0;
  y = frexp (x, &n);
  printf("For frexp of %f¥n the fraction is %f¥n ",
         x, y);
  printf("  and the exponent is %d¥n¥n", n);
}
```

出力：

```
For frexp of 50.000000
  the fraction is 0.781250
   and the exponent is 6

For frexp of -2.500000
  the fraction is -0.625000
   and the exponent is 2

For frexp of 0.000000
  the fraction is 0.000000
   and the exponent is 0
```

frexpf

説明： 単精度浮動小数値の小数部と指数部を取得します。

インクルード： <math.h>

プロトタイプ： float frexpf (float *x*, int *exp);

引数：

x	小数部と指数部を求める浮動小数値
*exp	格納された整数の指数部を指すポインタ

戻り値： 小数部を返します。exp は指数部を指します。x が 0 の場合、関数は、小数部と指数部の両方に 0 を返します。

備考： 小数部の絶対値は、1/2 ～ 1 (1 は含まない) の範囲。領域エラーまたは範囲エラーは発生しません。

例：

```
#include <math.h>  /* for frexpf */
#include <stdio.h> /* for printf */

int main(void)
{
  float x,y;
  int n;
```

frexpf (続き)

```
  x = 0.15F;
  y = frexpf (x, &n);
  printf("For frexpf of %f¥n  the fraction is %f¥n ",
         x, y);
  printf("  and the exponent is %d¥n¥n", n);

  x = -2.5F;
  y = frexpf (x, &n);
  printf("For frexpf of %f¥n  the fraction is %f¥n ",
         x, y);
  printf("  and the exponent is %d¥n¥n", n);

  x = 0.0F;
  y = frexpf (x, &n);
  printf("For frexpf of %f¥n  the fraction is %f¥n ",
         x, y);
  printf("  and the exponent is %d¥n¥n", n);
}
```

出力：

```
For frexpf of 0.150000
  the fraction is 0.600000
   and the exponent is -2

For frexpf of -2.500000
  the fraction is -0.625000
   and the exponent is 2

For frexpf of 0.000000
  the fraction is 0.000000
   and the exponent is 0
```

ldexp

説明： 倍精度浮動小数と 2 の指数の積を計算します。

インクルード： <math.h>

プロトタイプ： double ldexp(double *x*, int *ex*);

引数： *x* 浮動小数値

ex 指数部の整数

戻り値： *x* * 2^*ex* を返します。オーバーフローの場合は inf を、アンダーフローは 0 を、それぞれ返します。

備考： オーバーフローまたはアンダーフローでが発生します。

例：

```
#include <math.h>  /* for ldexp          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x,y;
  int n;
```

ldexp (続き)

```
  errno = 0;
  x = -0.625;
  n = 2;
  y = ldexp (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d¥n",
         x, n);
  printf("  ldexp(%f, %d) = %f¥n¥n",
         x, n, y);

  errno = 0;
  x = 2.5;
  n = 3;
  y = ldexp (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d¥n",
         x, n);
  printf("  ldexp(%f, %d) = %f¥n¥n",
         x, n, y);

  errno = 0;
  x = 15.0;
  n = 10000;
  y = ldexp (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d¥n",
         x, n);
  printf("  ldexp(%f, %d) = %f¥n¥n",
         x, n, y);
}
```

出力：

```
For a number = -0.625000 and an exponent = 2
  ldexp(-0.625000, 2) = -2.500000

For a number = 2.500000 and an exponent = 3
  ldexp(2.500000, 3) = 20.000000

Error: range error
For a number = 15.000000 and an exponent = 10000
  ldexp(15.000000, 10000) = inf
```

ldexpf

説明： 単精度浮動小数と 2 の指数の積を計算します。

インクルード： <math.h>

プロトタイプ： float ldexpf(float *x*, int ex);

引数： *x* 浮動小数値

ex 指数部の整数

戻り値： *x* * 2^ex を返します。オーバーフローの場合は inf を、アンダーフローは 0 を、それぞれ返します。

備考： オーバーフローまたはアンダーフローでが発生します。

ldexpf (続き)

例：

```
#include <math.h>  /* for ldexpf         */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x,y;
  int n;

  errno = 0;
  x = -0.625F;
  n = 2;
  y = ldexpf (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d¥n",
         x, n);
  printf("  ldexpf(%f, %d) = %f¥n¥n",
         x, n, y);

  errno = 0;
  x = 2.5F;
  n = 3;
  y = ldexpf (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d¥n",
         x, n);
  printf("  ldexpf(%f, %d) = %f¥n¥n",
         x, n, y);

  errno = 0;
  x = 15.0F;
  n = 10000;
  y = ldexpf (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d¥n",
         x, n);
  printf("  ldexpf(%f, %d) = %f¥n¥n",
         x, n, y);
}
```

出力：

```
For a number = -0.625000 and an exponent = 2
  ldexpf(-0.625000, 2) = -2.500000

For a number = 2.500000 and an exponent = 3
  ldexpf(2.500000, 3) = 20.000000

Error: range error
For a number = 15.000000 and an exponent = 10000
  ldexpf(15.000000, 10000) = inf
```

log

説明： 倍精度浮動小数値の自然対数を計算します。

インクルード： <math.h>

プロトタイプ： double log(double *x*);

引数： *x* 対数を計算する正の値

戻り値： *x* の自然対数を返します。*x* が 0 の場合は -inf を、*x* が負の場合は NaN を、それぞれ返します。

備考： *x* ≦ 0 の場合、領域エラーが発生します。

例：

```
#include <math.h>  /* for log            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = 2.0;
  y = log (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f¥n¥n",
          x, y);

  errno = 0;
  x = 0.0;
  y = log (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f¥n¥n",
          x, y);

  errno = 0;
  x = -2.0;
  y = log (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f¥n¥n",
          x, y);
}
```

出力：

```
The natural logarithm of 2.000000 is 0.693147

The natural logarithm of 0.000000 is -inf

Error: domain error
The natural logarithm of -2.000000 is nan
```

log10

説明： 倍精度浮動小数値の、底を 10 とする対数を計算します。

インクルード： <math.h>

プロトタイプ： double log10(double *x*);

引数： *x* 正の倍精度浮動小数値

戻り値： *x* の底を 10 とする対数を返します。*x* が 0 の場合は -inf を、*x* が負の場合は NaN を、それぞれ返します。

備考： *x* ≦ 0 の場合、領域エラーが発生します。

例：

```
#include <math.h>  /* for log10          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = 2.0;
  y = log10 (x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f¥n¥n",
          x, y);

  errno = 0;
  x = 0.0;
  y = log10 (x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f¥n¥n",
          x, y);

  errno = 0;
  x = -2.0;
  y = log10 (x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f¥n¥n",
          x, y);
}
```

出力：

```
The base-10 logarithm of 2.000000 is 0.301030

The base-10 logarithm of 0.000000 is -inf

Error: domain error
The base-10 logarithm of -2.000000 is nan
```

log10f

説明： 単精度浮動小数値の、底を 10 とする対数を計算します。

インクルード： <math.h>

プロトタイプ： float log10f(float *x*);

引数： *x* 正の単精度浮動小数値

戻り値： *x* の底を 10 とする対数を返します。*x* が 0 の場合は -inf を、*x* が負の場合は NaN を、それぞれ返します。

備考： *x* ≦ 0 の場合、領域エラーが発生します。

例：

```
#include <math.h>  /* for log10f         */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = 2.0F;
  y = log10f(x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f¥n¥n",
         x, y);

  errno = 0;
  x = 0.0F;
  y = log10f(x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f¥n¥n",
         x, y);

  errno = 0;
  x = -2.0F;
  y = log10f(x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f¥n¥n",
         x, y);
}
```

出力：

```
The base-10 logarithm of 2.000000 is 0.301030

Error: domain error
The base-10 logarithm of 0.000000 is -inf

Error: domain error
The base-10 logarithm of -2.000000 is nan
```

logf

説明： 単精度浮動小数値の自然対数を計算します。

インクルード： <math.h>

プロトタイプ： float logf(float *x*);

引数： *x* 対数を計算する正の値

戻り値： *x* の自然対数を返します。*x* が 0 の場合は -inf を、*x* が負の場合は NaN を、それぞれ返します。

備考： *x* ≦ 0 の場合、領域エラーが発生します。

例：

```
#include <math.h>  /* for logf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = 2.0F;
  y = logf (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f¥n¥n",
          x, y);

  errno = 0;
  x = 0.0F;
  y = logf (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f¥n¥n",
          x, y);

  errno = 0;
  x = -2.0F;
  y = logf (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f¥n¥n",
          x, y);
}
```

出力：

```
The natural logarithm of 2.000000 is 0.693147

The natural logarithm of 0.000000 is -inf

Error: domain error
The natural logarithm of -2.000000 is nan
```

modf

説明： 倍精度浮動小数値を小数部と整数部に分けます。

インクルード： <math.h>

プロトタイプ： double modf(double *x*, double *pint);

引数： *x* 倍精度浮動小数値

pint 格納された整数部を指すポインタ

戻り値： 符号付き小数部を返します。pint は整数部を指します。

備考： 小数部の絶対値は、0 ～ 1 (1 は含まない) の範囲。領域エラーまたは範囲エラーは発生しません。

例：

```
#include <math.h>  /* for modf   */
#include <stdio.h> /* for printf */

int main(void)
{
  double x,y,n;

  x = 0.707;
  y = modf (x, &n);
  printf("For %f the fraction is %f¥n ", x, y);
  printf("  and the integer is %0.f¥n¥n", n);

  x = -15.2121;
  y = modf (x, &n);
  printf("For %f the fraction is %f¥n ", x, y);
  printf("  and the integer is %0.f¥n¥n", n);
}
```

出力：

```
For 0.707000 the fraction is 0.707000
   and the integer is 0

For -15.212100 the fraction is -0.212100
   and the integer is -15
```

modff

説明： 単精度浮動小数値を小数部と整数部に分けます。

インクルード： <math.h>

プロトタイプ： float modff(float *x*, float *pint);

引数： *x* 単精度浮動小数値

pint 格納された整数部を指すポインタ

戻り値： 符号付き小数部を返します。pint は整数部を指します。

備考： 小数部の絶対値は、0 ～ 1 (1 は含まない) の範囲。領域エラーまたは範囲エラーは発生しません。

例：

```
#include <math.h>  /* for modff  */
#include <stdio.h> /* for printf */

int main(void)
{
  float x,y,n;

  x = 0.707F;
  y = modff (x, &n);
  printf("For %f the fraction is %f¥n ", x, y);
  printf("  and the integer is %0.f¥n¥n", n);

  x = -15.2121F;
  y = modff (x, &n);
  printf("For %f the fraction is %f¥n ", x, y);
  printf("  and the integer is %0.f¥n¥n", n);
}
```

出力：

```
For 0.707000 the fraction is 0.707000
   and the integer is 0

For -15.212100 the fraction is -0.212100
   and the integer is -15
```

pow

説明： *x* の y 乗を計算します。

インクルード： <math.h>

プロトタイプ： double pow(double *x*, double *y*);

引数： *x* 基数

y 指数部

戻り値： *x* の *y* 乗 (*x*^*y*) を返します。

備考： *y* が 0 の場合、pow は 1 を返します。*x* が 0.0 の場合、および *y* が 0 より小さい場合には、pow は inf を返し、領域エラーが発生します。結果がオーバーフローまたはアンダーフローの場合、範囲エラーが発生します。

例：

```
#include <math.h>  /* for pow            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x,y,z;

  errno = 0;
  x = -2.0;
  y = 3.0;
  z = pow (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f¥n¥n ", x, y, z);

  errno = 0;
  x = 3.0;
  y = -0.5;
  z = pow (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f¥n¥n ", x, y, z);

  errno = 0;
  x = 4.0;
  y = 0.0;
  z = pow (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f¥n¥n ", x, y, z);

  errno = 0;
  x = 0.0;
  y = -3.0;
  z = pow (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f¥n¥n ", x, y, z);
}
```

pow (続き)

出力：

```
-2.000000 raised to 3.000000 is -8.000000

3.000000 raised to -0.500000 is 0.577350

4.000000 raised to 0.000000 is 1.000000

Error: domain error
0.000000 raised to -3.000000 is inf
```

powf

説明： *x* の *y* 乗を計算します。

インクルード： <math.h>

プロトタイプ： float powf(float *x*, float *y*);

引数： *x* 基数

y 指数部

戻り値： *x* の *y* 乗 (*x*^*y*) を返します。

備考： *y* が 0 の場合、powf は 1 を返します。*x* が 0.0 の場合、および *y* が 0 より小さい場合には、powf は inf を返し、領域エラーが発生します。結果がオーバーフローまたはアンダーフローの場合、範囲エラーが発生します。

例：

```
#include <math.h>  /* for powf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x,y,z;

  errno = 0;
  x = -2.0F;
  y = 3.0F;
  z = powf (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f¥n¥n ", x, y, z);

  errno = 0;
  x = 3.0F;
  y = -0.5F;
  z = powf (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f¥n¥n ", x, y, z);

  errno = 0;
  x = 0.0F;
  y = -3.0F;
  z = powf (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f¥n¥n ", x, y, z);
}
```

powf (続き)

出力：

```
-2.000000 raised to 3.000000 is -8.000000

 3.000000 raised to -0.500000 is 0.577350

 Error: domain error
 0.000000 raised to -3.000000 is inf
```

sin

説明： 倍精度浮動小数値の sine 三角関数を計算します。

インクルード： <math.h>

プロトタイプ： double sin (double *x*);

引数： *x* sine を計算する値

戻り値： -1 ～ 1 の範囲のラジアンで表わした x の sine 値を返します。

備考： *x* が NaN の場合、または無限の場合、領域エラーが発生します。

例：

```
#include <math.h>  /* for sin             */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno           */

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.0;
  y = sin (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f¥n¥n", x, y);

  errno = 0;
  x = 0.0;
  y = sin (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f¥n¥n", x, y);
}
```

出力：

```
The sine of -1.000000 is -0.841471

The sine of 0.000000 is 0.000000
```

sinf

説明： 単精度浮動小数値の sine 三角関数を計算します。

インクルード： <math.h>

プロトタイプ： float sinf (float *x*);

引数： *x* sine を計算する値

戻り値： -1 ～ 1 の範囲のラジアンで表わした x の sin 値を返します。

備考： *x* が NaN の場合、または無限の場合、領域エラーが発生します。

例：

```
#include <math.h>  /* for sinf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = sinf (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f¥n¥n", x, y);

  errno = 0;
  x = 0.0F;
  y = sinf (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f¥n¥n", x, y);
}
```

出力：

```
The sine of -1.000000 is -0.841471

The sine of 0.000000 is 0.000000
```

sinh

説明： 倍精度浮動小数値の双曲 sine 関数を計算します。

インクルード： <math.h>

プロトタイプ： double sinh (double *x*);

引数： *x* 双曲 sine を計算する値

戻り値： *x* の双曲 sine を返します。

備考： *x* が大きい過ぎる場合、範囲エラーが発生します。

例：

```
#include <math.h>  /* for sinh           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.5;
  y = sinh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f¥n¥n",
          x, y);

  errno = 0;
  x = 0.0;
  y = sinh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f¥n¥n",
          x, y);

  errno = 0;
  x = 720.0;
  y = sinh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f¥n¥n",
          x, y);
}
```

出力：

```
The hyperbolic sine of -1.500000 is -2.129279

The hyperbolic sine of 0.000000 is 0.000000

Error: range error
The hyperbolic sine of 720.000000 is inf
```

sinhf

説明： 単精度浮動小数値の双曲 sine 関数を計算します。

インクルード： <math.h>

プロトタイプ： float sinhf (float *x*);

引数： *x* 双曲 sine を計算する値

戻り値： *x* の双曲 sine を返します。

備考： *x* が大きい過ぎる場合、範囲エラーが発生します。

例：

```
#include <math.h>  /* for sinhf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = sinhf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f¥n¥n",
          x, y);

  errno = 0;
  x = 0.0F;
  y = sinhf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f¥n¥n",
          x, y);
}
```

出力：

```
The hyperbolic sine of -1.000000 is -1.175201

The hyperbolic sine of 0.000000 is 0.000000
```

sqrt

説明： 倍精度浮動小数値の平方根を計算します。

インクルード： <math.h>

プロトタイプ： double sqrt(double *x*);

引数： *x* 非負の浮動小数値

戻り値： *x* の非負の平方根を返します。

備考： *x* が負の場合、領域エラーが発生します。

例：

```
#include <math.h>  /* for sqrt           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = 0.0;
  y = sqrt (x);
  if (errno)
    perror("Error");
  printf("The square root of %f is %f¥n¥n", x, y);

  errno = 0;
  x = 9.5;
  y = sqrt (x);
  if (errno)
    perror("Error");
  printf("The square root of %f is %f¥n¥n", x, y);

  errno = 0;
  x = -25.0;
  y = sqrt (x);
  if (errno)
    perror("Error");
  printf("The square root of %f is %f¥n¥n", x, y);
}
```

出力：

```
The square root of 0.000000 is 0.000000

The square root of 9.500000 is 3.082207

Error: domain error
The square root of -25.000000 is nan
```

sqrtf

説明： 単精度浮動小数値の平方根を計算します。

インクルード： <math.h>

プロトタイプ： float sqrtf(float *x*);

引数： *x* 非負の浮動小数値

戻り値： *x* の非負の平方根を返します。

備考： *x* が負の場合、領域エラーが発生します。

例：

```
#include <math.h>  /* for sqrtf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x;

  errno = 0;
  x = sqrtf (0.0F);
  if (errno)
    perror("Error");
  printf("The square root of 0.0F is %f¥n¥n", x);

  errno = 0;
  x = sqrtf (9.5F);
  if (errno)
    perror("Error");
  printf("The square root of 9.5F is %f¥n¥n", x);

  errno = 0;
  x = sqrtf (-25.0F);
  if (errno)
    perror("Error");
  printf("The square root of -25F is %f¥n", x);
}
```

出力：

```
The square root of 0.0F is 0.000000

The square root of 9.5F is 3.082207

Error: domain error
The square root of -25F is nan
```

tan

説明： 倍精度浮動小数値の tangent 三角関数を計算します。

インクルード： <math.h>

プロトタイプ： double tan (double *x*);

引数： *x* tangent を計算する値

戻り値： ラジアンで表わした x の tangent を返します。

備考： *x* が NaN の場合、または無限の場合、領域エラーが発生します。

例：

```
#include <math.h>  /* for tan            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.0;
  y = tan (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f¥n¥n", x, y);

  errno = 0;
  x = 0.0;
  y = tan (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f¥n¥n", x, y);
}
```

出力：

```
The tangent of -1.000000 is -1.557408

The tangent of 0.000000 is 0.000000
```

tanf

説明： 単精度浮動小数値の tangent 三角関数を計算します。

インクルード： <math.h>

プロトタイプ： float tanf (float *x*);

引数： *x* tangent を計算する値

戻り値： *x* の tangent を返します。

備考： *x* が NaN の場合、または無限の場合、領域エラーが発生します。

例：

```
#include <math.h>  /* for tanf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;
```

tanf (続き)

```
  errno = 0;
  x = -1.0F;
  y = tanf (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f¥n¥n", x, y);

  errno = 0;
  x = 0.0F;
  y = tanf (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f¥n", x, y);
}
```

出力：

```
The tangent of -1.000000 is -1.557408

The tangent of 0.000000 is 0.000000
```

tanh

説明： 倍精度浮動小数値の双曲 tangent 関数を計算します。

インクルード： <math.h>

プロトタイプ： double tanh (double *x*);

引数： *x* 双曲 tangent を計算する値

戻り値： -1 ～ 1 の範囲の双曲 tangent の値を返します。

備考： 領域エラーまたは範囲エラーは発生しません。

例：

```
#include <math.h>  /* for tanh   */
#include <stdio.h> /* for printf */

int main(void)
{
  double x, y;

  x = -1.0;
  y = tanh (x);
  printf("The hyperbolic tangent of %f is %f¥n¥n",
          x, y);

  x = 2.0;
  y = tanh (x);
  printf("The hyperbolic tangent of %f is %f¥n¥n",
          x, y);
}
```

出力：

```
The hyperbolic tangent of -1.000000 is -0.761594

The hyperbolic tangent of 2.000000 is 0.964028
```

tanhf

説明： 単精度浮動小数値の双曲 tangent 関数を計算します。

インクルード： <math.h>

プロトタイプ： float tanhf (float *x*);

引数： *x* 双曲 tangent を計算する値

戻り値： -1 ～ 1 の範囲の双曲 tangent の値を返します。

備考： 領域エラーまたは範囲エラーは発生しません。

例：

```
#include <math.h>  /* for tanhf  */
#include <stdio.h> /* for printf */

int main(void)
{
  float x, y;

  x = -1.0F;
  y = tanhf (x);
  printf("The hyperbolic tangent of %f is %f¥n¥n",
          x, y);

  x = 0.0F;
  y = tanhf (x);
  printf("The hyperbolic tangent of %f is %f¥n¥n",
          x, y);
}
```

出力：

```
The hyperbolic tangent of -1.000000 is -0.761594

The hyperbolic tangent of 0.000000 is 0.000000
```

4.18 PIC30-LIBS

標準 C ライブラリ・ヘルパー関数を次に示します :

- _exit　　プログラムの実行を停止させます。
- brk　　プロセッサのデータ領域の終わりを設定します。
- close　　ファイルを閉じます。
- lseek　　ファイル・ポインタを指定の位置に移動します。
- open　　ファイルを開きます。
- read　　ファイルからデータを読み出します。
- sbrk　　プロセッサのデータ領域を与えられた増分だけ増やします。
- write　　データをファイルへ書き込みます。

これらの関数は標準 C ライブラリ内の他の関数からコールされ、ターゲット・アプリケーション向けに修正される必要があります。対応するオブジェクト・モジュールは、libpic30-*omf*.a アーカイブ内で配布され。ソース・コード (MPLAB C30 用) は src¥pic30 フォルダ内にあります。

さらに、いくつかの標準 C ライブラリ関数も、ターゲット・アプリケーションに合わせて修正する必要があります。このような関数としては次があります :

- getenv　　環境変数の値を取得します。
- remove　　ファイルを削除します。
- rename　　ファイルまたはディレクトリの名前を変更します。
- system　　コマンドを実行します。
- time　　システム時間を取得します。

これらの関数は標準 C ライブラリに含まれていますが、オブジェクト・モジュールは、libpic30-*omf*.a アーカイブ内で配布され。ソース・コード (MPLAB C30 用) は src¥pic30 フォルダ内にあります。これらのモジュールは、libc-*omf*.a の一部として配布されていません。

4.18.1 libpic30-omf.a ライブラリの再ビルド

デフォルトとして、本章に記載するヘルパー関数は、sim30 シミュレータ上で動作するように書かれています。ヘッダー・ファイル simio.h が、ライブラリとシミュレータとの間のインターフェースを定義しています。したがって、ライブラリを再ビルドしてシミュレータを使い続けることができます。ただし、シミュレータは組込みアプリケーション上にはないので、アプリケーションはこのインターフェースを使うことはできません。

ヘルパー関数は、ターゲット・アプリケーションに合わせて修正 / 再ビルドする必要があります。libpic30-*omf*.a ライブラリは、バッチ・ファイル makelib.bat を使って再ビルドすることができます。このバッチ・ファイはソースとして src¥pic30 内にあります。コマンド・ウインドウからバッチ・ファイルを実行してください。src¥pic30 ディレクトリ内にいることを確認してください。そして、新しくコンパイルしたファイル (libpic30-*omf*.a) を lib ディレクトリへコピーします。

4.18.2 関数の説明

このセクションでは、ターゲット環境内で標準 C ライブラリを正しく動作させるため、カスタマイズが必要な関数について説明します。デフォルト動作のセクションでは、配布時の関数動作を説明します。説明と備考では、一般的な動作を説明します。

_exit

説明： プログラムの実行を停止させます。

インクルード： なし

プロトタイプ： `void _exit (int status);`

引数： *status* 終了ステータス

備考： `exit()` 標準 C ライブラリ関数からコールされるヘルパー関数です。

デフォルト動作： 配布時、この関数は `stdout` をクリアして終了します。パラメータ・ステータスは、`exit()` 標準 C ライブラリ関数に渡されるものと同じです。

ファイル： `_exit.c`

brk

説明： プロセッサのデータ領域の終わりを設定します。

インクルード： なし

プロトタイプ： `int brk(void *endds)`

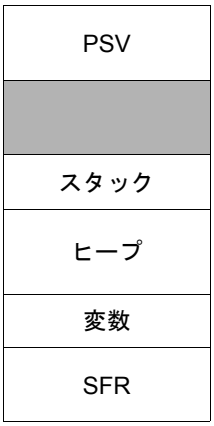
引数： *endds* データ・セグメントの終わりを指すポインタ

戻り値： 正常終了の場合 `0` を返します。その他の場合 `-1` を返します。

備考： `brk()` を使って、コール側プロセッサのデータ・セグメントに割り当てる領域の大きさを動的に変更します。プロセッサのブレーク値をリセットして、適切な領域を割り当てることにより、変更することができます。ブレーク値とは、データ・セグメントの終わりを超えた先頭ロケーションのアドレスを意味します。割り当てる領域は、ブレーク値を大きくすると増えます。
新しく割り当てた領域は初期化されていません。
`malloc()` 標準C ライブラリ関数からコールされるヘルパー関数です。

brk (続き)

デフォルト動作： 引数 endds がゼロの場合、この関数はグローバル変数 __curbrk をヒープの開始アドレスに設定し、ゼロを返します。
引数 endds が非ゼロで、かつヒープの終わりのアドレスより値が小さい場合、この関数はグローバル変数 __curbrk を終わりの値に設定し、ゼロを返します。
その他の場合は、グローバル変数 __curbrk は不変で、関数は -1 を返します。
引数 endds はヒープの範囲内である必要があります (下のデータ領域メモリマップ参照)。



スタックがヒープの直前にあるため、brk() または sbrk() を使用しても、ダイナミック・メモリ・プールのサイズに影響を与えることがないことに注意してください。brk() 関数と sbrk() 関数は本来、スタックが下向きに大きくなり、ヒープが上向きに大きくなるランタイム環境での使用を目的としたものです。
リンカーは -Wl,--heap=n オプションが指定された場合、メモリのブロックをヒープに割り当てます。ここで、n は文字数で表わした必要とされるヒープ・サイズです。ヒープの開始了アドレスと終了アドレスは、それぞれ変数 _heap と変数 _eheaps に報告されます。
MPLAB C30 の場合、brk() と sbrk() に依存する代わりに、リンカーのヒープ・サイズ・オプションを使用することは、ヒープ・サイズを制御する標準的な方法です。

ファイル： brk.c

close

説明： ファイルを閉じます。

インクルード： なし

プロトタイプ： int close(int *handle*);

引数： *handle* 開いているファイルを参照するハンドル

戻り値： ファイルが正常に閉じた場合は '0' を返します。戻り値 '-1' はエラーを表わします。

備考： fclose() 標準 C ライブラリ関数からコールされるヘルパー関数です。

デフォルト動作： 配布時、この関数はファイル・ハンドルをシミュレータに渡します。このシミュレータはホスト・ファイル・システム内でクローズを発行します。

ファイル： close.c

getenv

説明： 環境変数の値を取得します。

インクルード： `<stdlib.h>`

プロトタイプ： `char *getenv(const char *s);`

引数： *s* 環境変数の名前

戻り値： 正常終了の場合、環境変数の値を指すポインタを返します。その他の場合は null ポインタを返します。

デフォルト動作： 配布時、この関数は null ポインタを返します。環境変数に対するサポートはありません。

ファイル： `getenv.c`

lseek

説明： ファイル・ポインタを指定の位置に移動します。

インクルード： なし

プロトタイプ： `long lseek(int handle, long offset, int origin);`

引数：

handle 開いているファイルを参照するハンドル

offset origin からの文字数

origin シークを開始する位置。*origin* としては次の値 (stdio.h で定義) が可能です：
SEEK_SET ―ファイルの先頭
SEEK_CUR ―ファイル・ポインタの現在位置
SEEK_END ― End-of-file.

戻り値： 新しい位置を文字数で表わした、ファイル先頭からのオフセットを返します。戻り値 `'-1L'` はエラーを表わします。

備考： `fgetpos()`、`ftell()`、`fseek()`、`fsetpos`、`rewind()` の各標準 C ライブラリ関数からコールされるヘルパー関数です。

デフォルト動作： 配布時、パラメータはシミュレータを経由してホスト・ファイル・システムへ渡されます。戻り値は、ホスト・ファイル・システムから返された値になります。

ファイル： `lseek.c`

open

説明： ファイルを開きます。

インクルード： なし

プロトタイプ：
```
int open(const char *name, int access, int mode);
```

引数：

name	開くファイルの名前
access	ファイルを開くためのアクセス・メソッド
mode	許容されるアクセス・タイプ

戻り値： 正常終了の場合、関数はファイル・ハンドル（小さい正の整数値）を返します。このハンドルは、後続の低レベル・ファイル I/O 操作に使います。戻り値 '-1' はエラーを表わします。

備考： アクセス・フラグは、次のいずれかのアクセス・メソッドとゼロまたは複数個のアクセス修飾子の共用体です：
0 ―ファイルを開く、読み出し用。
1 ―ファイルを開く、書き込み用。
2 ―ファイルを開く、読み書き用。
次のアクセス修飾子をサポートします :
0x0008 ―ファイル・ポインタを end-of-file へ移動した後に各書き込み動作を行います。
0x0100 ―新しいファイルを書き込み用に生成して開きます。
0x0200 ―ファイルを開き、ゼロ長に短縮します。
0x4000 ―テキスト (変換済み) モードでファイルを開きます。
0x8000 ―バイナリ (未変換) モードでファイルを開きます。
モード・パラメータは次のいずれかを指定できます :
0x0100 ―読み出し専用。
0x0080 ―書き込み許可 (読み出し可能を意味します)。
fopen() と freopen() の各標準 C ライブラリ関数からコールされるヘルパー関数です。

デフォルト動作： 配布時、パラメータはシミュレータを経由してホスト・ファイル・システムへ渡されます。戻り値は、ホスト・ファイル・システムから返された値になります。ホスト・システムが値 '-1' を返す場合、グローバル変数 errno に <errno.h> 内で定義されたシンボル定数 EFOPEN の値が設定されます。

ファイル： open.c

read

説明： ファイルからデータを読み出します。

インクルード： なし

プロトタイプ：
```
int read(int handle, void * buffer, unsigned int len);
```

引数：
handle 開いているファイルを参照するハンドル
buffer 読み出しデータの格納場所を指すポインタ
len 読み出す最大文字数

戻り値： 読み出す文字数を返します。ファイル内に残っている文字数が len より少ない場合、またはファイルがテキスト・モードで開かれていて、各キャリッジ・リターン – ラインフィード (CR-LF) 対が 1 文字のラインフィード文字で置き換えられている場合、返される文字数は len より小さくなります。戻り値では、1 文字のラインフィード文字だけがカウントされます。置き換えによって、ファイル・ポインタは影響を受けません。関数が end-of-file で読み出しを行おうとすると、'0' が返されます。ハンドルが無効である場合、またはファイルが読み出し用に開かれていない場合、またはファイルがロックされている場合、この関数は '-1' を返します。

備考： fgetc()、fgets()、fread()、gets() の各標準 C ライブラリ関数からコールされるヘルパー関数です。

デフォルト動作： 配布時、パラメータはシミュレータを経由してホスト・ファイル・システムへ渡されます。戻り値は、ホスト・ファイル・システムから返された値になります。

ファイル： read.c

remove

説明： ファイルを削除します。

インクルード： <stdio.h>

プロトタイプ：
```
int remove(const char *filename);
```

引数：
filename 削除するファイルの名前

戻り値： 正常終了の場合 0 を返します。その他の場合 -1 を返します。

デフォルト動作： 配布時、パラメータはシミュレータを経由してホスト・ファイル・システムへ渡されます。戻り値は、ホスト・ファイル・システムから返された値になります。

ファイル： remove.c

rename

説明： ファイルまたはディレクトリの名前を変更します。

インクルード： <stdio.h>

プロトタイプ：
```
int rename(const char *oldname, const char *newname);
```

引数：
oldname 古い名前を指すポインタ
newname 新しい名前を指すポインタ

戻り値： 正常終了の場合は '0' を返します。エラーの場合は、非ゼロ値を返します。

デフォルト動作： 配布時、パラメータはシミュレータを経由してホスト・ファイル・システムへ渡されます。戻り値は、ホスト・ファイル・システムから返された値になります。

ファイル： rename.c

sbrk

説明： プロセッサのデータ領域を与えられた増分だけ増やします。

インクルード： なし

プロトタイプ：
```
void * sbrk(int incr);
```

引数： *incr* 増加 / 削減する文字数

戻り値： 割り当てられた新しい領域の先頭を返します。エラーの場合には '-1' を返します。

備考： sbrk() は、incr 個の文字数をブレーク値に加算して割り当て領域を変更します。incr は負の値をとることができ、この場合割り当て領域が削減されます。
sbrk() を使って、コール側プロセッサのデータ・セグメントに割り当てる領域の大きさを動的に変更します。プロセッサのブレーク値をリセットして、適切な領域を割り当てることにより、変更することができます。ブレーク値とは、データ・セグメントの終わりを超えた先頭ロケーションのアドレスを意味します。割り当てる領域は、ブレーク値を大きくすると増えます。
malloc() 標準 C ライブラリ関数からコールされるヘルパー関数です。

デフォルト動作： グローバル変数 __curbrk がゼロの場合、この関数は brk() をコールしてブレーク値を初期化します。brk() が -1 を返す場合、この関数も同じ動作をします。
incr がゼロの場合、グローバル変数 __curbrk の現在の値が返されます。
incr が非ゼロの場合、この関数はアドレス (__curbrk + *incr*) がヒープの終わりのアドレスより小さいことを確認します。小さい場合には、グローバル変数 __curbrk がその値に更新され、この関数は __curbrk の符号なし値を返します。
その他の場合、この関数は -1 を返します。
brk() の説明を参照してください。

ファイル： sbrk.c

system

説明： コマンドを実行します。

インクルード： <stdlib.h>

プロトタイプ：
```
int system(const char *s);
```

引数： *s* 実行するコマンド

デフォルト動作： 配布時、この関数はユーザー関数のスタブまたはプレースホルダとして機能します。s が NULL でない場合、エラー・メッセージが stdout へ書き込まれ、プログラムがリセットされます。その他の場合には、値 -1 が返されます。

ファイル： system.c

time

説明： システム時間を取得します。

インクルード： `<time.h>`

プロトタイプ：
```
time_t time(time_t *timer);
```

引数： *timer* 時間の格納場所を指すポインタ

戻り値： 時間経過を秒数で返します。エラー・リターンはありません。

デフォルト動作： 配布時、timer2 がイネーブルされていない場合、32 ビット・モードでイネーブルします。戻り値は、32 ビット timer2 レジスタの現在の値になります。非常に希なケースを除き、この戻り値は秒で表わした経過時間ではありません。

ファイル： time.c

write

説明： データをファイルへ書き込みます。

インクルード： なし

プロトタイプ：
```
int write(int handle, void *buffer, unsigned int count);
```

引数：
handle 開いているファイルを参照するハンドル
buffer 書き込みデータの格納場所を指すポインタ
count 書き込む文字数

戻り値： 正常終了の場合、実際に書き込まれた文字数を返します。戻り値 '-1' はエラーを表わします。

備考： ディスクの実際の空き領域がバッファ・サイズより小さい場合、この関数はディスクへの書き込みを試みて、書き込みに失敗しますが、ディスクに対するバッファの内容をクリアしません。ファイルがテキスト・モードで開かれている場合、各ラインフィード文字は出力でキャリッジ・リターンとラインフィードの対で置き換えられます。この置き換えによって、戻り値は影響を受けません。
fflush() 標準 C ライブラリ関数からコールされるヘルパー関数です。

デフォルト動作： 配布時、パラメータはシミュレータを経由してホスト・ファイル・システムへ渡されます。戻り値は、ホスト・ファイル・システムから返された値になります。

ファイル： write.c

メモ：

第 5 章 . MPLAB C30 組込関数

5.1 序論

本章では、16 ビット・デバイスに固有な MPLAB C30 の組込関数について説明します。

組込関数を使うと、現在インライン・アセンブリを使わなければアクセスできないアセンブラ・オペレータまたはマシン命令がアクセスできるようになります。これらの関数は、広範囲なアプリケーションに使用できるため十分有効です。組込関数は、構文的には関数コールに似た C ソース・ファイルとしてコーディックされていますが、関数を直接組込むアセンブリ・コードにコンパイルされるので、関数コールまたはライブラリ・ルーチンを使っていません。

インライン・アセンブリを使うプログラマに対して、組込関数を提供することが望ましい理由は多くあります。中でも次のような理由があげられます。

1. 目的によっては組込関数を提供すると、コーディックが簡素化されます。
2. インライン・アセンブリを使うと、最適化できないことがあります。組込関数にはこの制約がありません。
3. 専用レジスタを使うマシン命令の場合、レジスタ割り当てエラーを回避しながらインライン・アセンブリをコーディングすることは、非常に注意が必要なことです。組込関数を使うと、各マシン命令の特定のレジスタ要求を気にしないで済むためこれが簡単になります。

本章は次のように構成されています。

- 組込関数の一覧

5.2 組込関数の一覧

本セクションでは、MPLAB C30 C コンパイラの組込関数のプログラマ・インターフェースについて説明します。関数は " 組込み " まれているため、これに対応するヘッダー・ファイルはありません。同様に、組込関数に対応するコマンドライン・スイッチもありません—常に使用可能です。組込関数名は、プログラマの名前空間内にある関数名または変数名と競合しないように、コンパイラの名前空間に属するように選ばれています (すべてにプレフィックス `__builtin_` が付いています)。

__builtin_addab

説明 : アキュムレータ A とアキュムレータ B を加算して、結果を指定したアキュムレータに書き込みます。例えば :

```
register int result asm("A");
result = __builtin_addab();
```

このコードは次を生成します :

```
add A
```

プロトタイプ : `int __builtin_addab(void);`

引数 : なし

戻り値 : 加算結果をアキュムレータへ返します。

アセンブラ・オペレータ / マシン命令 : `addad`

エラー・メッセージ 結果がアキュムレータ・レジスタでない場合、エラー・メッセージが表示されます。

__builtin_add

説明 : `result` で指定されたアキュムレータに `value` を加算し、リテラル `shift` で指定されたシフトを行います。例えば :

```
register int result asm("A");
int value;
result = __builtin_add(value,0);
```

`value` を w0 に保持した場合、次が生成されます :

```
add w0, #0, A
```

プロトタイプ : `int __builtin_add(int value, const int shift);`

引数 : *value* アキュムレータ値に加算する整数値。
shift 結果のアキュムレータ値をシフトする量。

戻り値 : シフトしてアキュムレータへ加算した結果を返します。

アセンブラ・オペレータ / マシン命令 : `add`

エラー・メッセージ 次の場合に、エラー・メッセージが表示されます :

- 結果がアキュムレータ・レジスタでない
- シフト値が範囲内のリテラルでない

__builtin_btg

説明 : この関数は btg マシン命令を生成します。
いくつかの例を示します :

```
int i;   /* near by default */
int l __attribute__((far));

struct foo {
  int bit1:1;
} barbits;

int bar;

void some_bittoggles() {
  register int j asm("w9");
  int k;

  k = i;

  __builtin_btg(&bar,barbits.bit1);
  __builtin_btg(&i,1);
  __builtin_btg(&j,3);
  __builtin_btg(&k,4);
  __builtin_btg(&l,11);

  return j+k;
}
```

変数のアドレスをレジスタに格納するとコンパイラが警告を発生して、レジスタをスタックに待避させることに注意してください (アドレスを取得できるようにするため); この形式はお薦めできません。この注意は、プログラマが明示的にレジスタに格納した変数にのみ当てはまります。

プロトタイプ : `void __builtin_btg(unsigned int *, unsigned int0xn);`

引数 :
- `*` データ項目に対するポインタ。このデータ項目に対してビットがトグルします。
- 0x*n* 0 ～ 15 の範囲のリテラル値。便利なことに、この引数としてビットフィールド名を渡すことができます。この組込関数は、引数の代わりに指定されたフィールドのビット位置を使い、該当するビットをトグルします。

戻り値 : btg マシン命令を返します。

アセンブラ・オペレータ / マシン命令 : `btg`

エラー・メッセージ パラメータ値が範囲外の場合、エラー・メッセージが表示されます。

__builtin_clr

説明 : 指定されたアキュムレータをクリアします。例えば :

```
  register int result asm("A");
  result = __builtin_clr();
```

このコードは次を生成します :

```
  clr A
```

プロトタイプ : `int __builtin_clr(void);`

__builtin_clr (続き)

引数 : なし

戻り値 : クリアされた値の結果をアキュムレータへ返します。

アセンブラ・オペレータ / マシン命令 : clr

エラー・メッセージ 結果がアキュムレータ・レジスタでない場合、エラー・メッセージが表示されます。

__builtin_clr_prefetch

説明 : アキュムレータをクリアし、後で実行される MAC 演算に備えてデータをプリフェッチします。

xptr は null に設定されて X プリフェッチが行われないことを表示します。この場合、*xincr* と *xval* の値は無視されますが、両方の値は必要です。

xptr は null に設定されて Y プリフェッチが行われないことを表示します。この場合、*yincr* と *yval* の値は無視されますが、両方の値は必要です。

xval と *yval* は、プリフェッチされた値が格納される C 変数のアドレスを指定します。

xincr と *yincr* は、リテラル値 -6、-4、-2、0、2、4、6 または整数値をとることができます。

AWB が null でない場合は、他のアキュムレータが参照変数へ書き込まれます。

例えば :

```
register int result asm("A");
int x_memory_buffer[256]
    __attribute__((space(xmemory)));
int y_memory_buffer[256]
    __attribute__((space(ymemory)));
int *xmemory;
int *ymemory;
int awb;
int xVal, yVal;

xmemory = x_memory_buffer;
ymemory = y_memory_buffer;
result = __builtin_clr(&xmemory, &xVal, 2,
            &ymemory, &yVal, 2, &awb);
```

このコードは次を生成します :

```
clr A, [w8]+=2, w4, [w10]+=2, w5, w13
```

w13 をライトバックに使用できるようにするため、コンパイラは w13 を使う必要があります。レジスタをこのために使うことを要求することが推奨されます。

この命令の実行後 :

- 結果はクリアされます
- xVal には x_memory_buffer[0] が格納されます
- yVal には y_memory_buffer[0] が格納されます。
- xmemory と ymemory を 2 だけ増加させて、次の mac 動作に備えます。

プロトタイプ :

```
int __builtin_clr_prefetch(
  int **xptr, int *xval, int xincr,
  int **yptr, int *yval, int yincr, int *AWB);
```

__builtin_clr_prefetch (続き)

引数 :
xptr x プリフェッチに対する整数ポインタ
xval x プリフェッチの整数値
xincr x プリフェッチの整数増分値
yptr y プリフェッチに対する整数ポインタ
yval y プリフェッチの整数値
yincr y プリフェッチの整数増分値
AWB アキュムレータの選択

戻り値 : クリアされた値の結果をアキュムレータへ返します。

アセンブラ・オペレータ / マシン命令 : `clr`

エラー・メッセージ 次の場合に、エラー・メッセージが表示されます :
- 結果がアキュムレータ・レジスタでない
- *xval* は null 値であるが *xptr* は null でない
- *yval* は null 値であるが *yptr* は null でない

__builtin_divsd

説明 : この関数は、商 *num* / *den* を計算します。*den* がゼロの場合、算術エラー例外が発生します。関数の結果と同様に関数の引数は符号付きです。コマンドライン・オプション -Wconversions を使って、予期しない符号変換を検出することができます。

プロトタイプ :
```
int __builtin_divsd(const long num, const int den);
```

引数 :
num 分子
den 分母

戻り値 : 商 *num* / *den* の符号付き整数値を返します。

アセンブラ・オペレータ / マシン命令 : `div.sd`

__builtin_divud

説明 : この関数は、商 *num* / *den* を計算します。*den* がゼロの場合、算術エラー例外が発生します。関数の結果と同様に関数の引数は符号なしです。コマンドライン・オプション -Wconversions を使って、予期しない符号変換を検出することができます。

プロトタイプ :
```
unsigned int __builtin_divud(const unsigned
  long num, const unsigned int den);
```

引数 :
num 分子
den 分母

戻り値 : 商 *num* / *den* の符号なし整数値を返します。

アセンブラ・オペレータ / マシン命令 : `div.ud`

__builtin_dmaoffset

説明 : DMA メモリ内のシンボルのオフセットを取得します。
例えば :

```
int result;
char buffer[256] __attribute__((space(dma)));

result = __builtin_dmaoffset(buffer);
```

このコードは次を生成します :

```
mov #dmaoffset(buffer), w0
```

プロトタイプ :
```
int __builtin_dmaoffset(int buffer);
```

引数 : *buffer* DMA アドレス値

戻り値 : オフセットをアキュムレータへ返します。

アセンブラ・オペレータ / マシン命令 :
```
dmaoffset
```

エラー・メッセージ 結果がアキュムレータ・レジスタでない場合、エラー・メッセージが表示されます。

__builtin_ed

説明 : *sqr* を 2 乗して、結果を返します。また、データをプリフェッチし、***xptr* - ***yptr* を計算して結果を **distance* に格納して、後で行われる 2 乗動作備えます。
xincr と *yincr* は、リテラル値 -6、-4、-2、0、2、4、6 または整数値をとることができます。
例えば :

```
register int result asm("A");
int *xmemory, *ymemory;
int distance;

result = __builtin_ed(distance,
                      &xmemory, 2,
                      &ymemory, 2,
                      &distance);
```

このコードは次を生成します :

```
ed w4*w4, A, [w8]+=2, [W10]+=2, w4
```

プロトタイプ :
```
int __builtin_ed(int sqr, int **xptr, int xincr,
  int **yptr, int yincr, int *distance);
```

引数 :

sqr	2 乗される整数値
xptr	x プリフェッチを指す整数ポインタ
xincr	x プリフェッチの整数増分値
yptr	y プリフェッチを指す整数ポインタ
yincr	y プリフェッチの整数増分値
distance	distance を指す整数ポインタ

戻り値 : 2 乗結果をアキュムレータへ返します。

アセンブラ・オペレータ / マシン命令 :
```
ed
```

__builtin_ed (続き)

エラー・メッセージ 次の場合に、エラー・メッセージが表示されます :
- 結果がアキュムレータ・レジスタでない
- *xptr* が null である
- *yptr* が null である
- *distance* が null である

__builtin_edac

説明 : *sqr* を 2 乗して指定されたアキュムレータ・レジスタに加算し、その結果を返します。また、データをプリフェッチし、***xptr* - ***yptr* を計算して結果を **distance* に格納して、後で行われる 2 乗動作備えます。
xincr と *yincr* は、リテラル値 -6、-4、-2、0、2、4、6 または整数値をとることができます。
例えば :

```
register int result asm("A");
int *xmemory, *ymemory;
int distance;

result = __builtin_ed(distance,
                      &xmemory, 2,
                      &ymemory, 2,
                      &distance);
```

このコードは次を生成します :

```
ed w4*w4, A, [w8]+=2, [W10]+=2, w4
```

プロトタイプ :

```
int __builtin_edac(int sqr, int **xptr, int xincr,
  int **yptr, int yincr, int *distance);
```

引数 :

sqr	2 乗される整数値
xptr	x プリフェッチを指す整数ポインタ
xincr	x プリフェッチの整数増分値
yptr	y プリフェッチを指す整数ポインタ
yincr	y プリフェッチの整数増分値
distance	distance を指す整数ポインタ

戻り値 : 2 乗結果を指定されたアキュムレータへ返します。

アセンブラ・オペレータ / マシン命令 :

```
edac
```

エラー・メッセージ 次の場合に、エラー・メッセージが表示されます :
- 結果がアキュムレータ・レジスタでない
- *xptr* が null である
- *yptr* が null である
- *distance* が null である

__builtin_fbcl

説明 : 値の中で最初のビット変化を左側から探します。この関数は、固定小数データの動的スケーリングに有効です。例えば :

```
int result, value;
result = __builtin_fbcl(value);
```

このコードは次を生成します :

```
fbcl w4, w5
```

__builtin_fbcl (続き)

プロトタイプ :	`int __builtin_fbcl(int value);`
引数 :	*value*　最初の変化ビットを表わす整数。
戻り値 :	シフトしてアキュムレータへ加算した結果を返します。
アセンブラ・オペレータ / マシン命令 :	`fbcl`
エラー・メッセージ	結果がアキュムレータ・レジスタでない場合、エラー・メッセージが表示されます。

__builtin_lac

説明 :	*shift* (-8 ～ 7 のリテラル) だけ値をシフトし、アキュムレータ・レジスタへ格納される値を返します。例えば :
	`register int result asm("A");`
	`int value;`
	`result = __builtin_lac(value,3);`
	
	このコードは次を生成します :
	`lac w4, #3, A`

プロトタイプ :	`int __builtin_lac(int value, int shift);`
引数 :	*value*　シフトされる整数。 *shift*　シフト数を表わすリテラル。
戻り値 :	シフトしてアキュムレータへ加算した結果を返します。
アセンブラ・オペレータ / マシン命令 :	`lac`
エラー・メッセージ	次の場合に、エラー・メッセージが表示されます : • 結果がアキュムレータ・レジスタでない • シフト値が範囲内のリテラルでない

__builtin_mac

説明 : *a* x *b* を計算しアキュムレータに加算します。また、データをプリフェッチして後で実行される MAC 動作に備えます。
xptr は null に設定されて X プリフェッチが行われないことを表示します。この場合、*xincr* と *xval* の値は無視されますが、両方の値は必要です。
xptr は null に設定されて Y プリフェッチが行われないことを表示します。この場合、*yincr* と *yval* の値は無視されますが、両方の値は必要です。
xval と *yval* は、プリフェッチされた値が格納される C 変数のアドレスを指定します。
xincr と *yincr* は、リテラル値 -6、-4、-2、0、2、4、6 または整数値をとることができます。
AWB が null でない場合は、他のアキュムレータが参照変数へ書き込まれます。
例えば :

```
register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_mac(xVal, yVal,
                       &xmemory, &xVal, 2,
                       &ymemory, &yVal, 2, 0);
```

このコードは次を生成します :

```
mac w4*w5, A, [w8]+=2, w4, [w10]+=2, w5
```

プロトタイプ :

```
int __builtin_mac(int a, int b,
int **xptr, int *xval, int xincr,
int **yptr, int *yval, int yincr, int *AWB);
```

引数 :

a	整数の被乗数
b	整数の乗数。
xptr	x プリフェッチを指す整数ポインタ
xval	x プリフェッチの値を指す整数ポインタ
xincr	x プリフェッチの整数増分値
yptr	y プリフェッチを指す整数ポインタ
yval	y プリフェッチの値を指す整数ポインタ
yincr	y プリフェッチの整数増分値
AWB	アキュムレータ選択を指す整数ポインタ

戻り値 : クリアされた値の結果をアキュムレータへ返します。

アセンブラ・オペレータ / マシン命令 : mac

エラー・メッセージ 次の場合に、エラー・メッセージが表示されます :

- 結果がアキュムレータ・レジスタでない
- *xval* は null 値であるが *xptr* は null でない
- *yval* は null 値であるが *yptr* は null でない

__builtin_movsac

説明 : 計算は何も行わず、後で実行される MAC 演算に備えてデータをプリフェッチします。
xptr は null に設定されて X プリフェッチが行われないことを表示します。この場合、*xincr* と *xval* の値は無視されますが、両方の値は必要です。
xptr は null に設定されて Y プリフェッチが行われないことを表示します。この場合、*yincr* と *yval* の値は無視されますが、両方の値は必要です。
xval と *yval* は、プリフェッチされた値が格納される C 変数のアドレスを指定します。
xincr と *yincr* は、リテラル値 -6、-4、-2、0、2、4、6 または整数値をとることができます。
AWB が null でない場合は、他のアキュムレータが参照変数へ書き込まれます。
例えば :

```
  register int result asm("A");
  int *xmemory;
  int *ymemory;
  int xVal, yVal;

  result = __builtin_movsac(&xmemory, &xVal, 2,
                      &ymemory, &yVal, 2, 0);
```

このコードは次を生成します :

```
        movsac A, [w8]+=2, w4, [w10]+=2, w5
```

プロトタイプ :

```
int __builtin_movsac(
  int **xptr, int *xval, int xincr,
  int **yptr, int *yval, int yincr, int *AWB);
```

引数 :

xptr	x プリフェッチを指す整数ポインタ
xval	x プリフェッチの値を指す整数ポインタ
xincr	x プリフェッチの整数増分値
yptr	y プリフェッチを指す整数ポインタ
yval	y プリフェッチの値を指す整数ポインタ
yincr	y プリフェッチの整数増分値
AWB	アキュムレータ選択を指す整数ポインタ

戻り値 : プリフェッチ・データを返します。

アセンブラ・オペレータ / マシン命令 : movsac

エラー・メッセージ 次の場合に、エラー・メッセージが表示されます :
- 結果がアキュムレータ・レジスタでない
- *xval* は null 値であるが *xptr* は null でない
- *yval* は null 値であるが *yptr* は null でない

__builtin_mpy

説明 : *a* x *b* を計算します。また、データをプリフェッチして後で実行される MAC 動作に備えます。
xptr は null に設定されて X プリフェッチが行われないことを表示します。この場合、*xincr* と *xval* の値は無視されますが、両方の値は必要です。
xptr は null に設定されて Y プリフェッチが行われないことを表示します。この場合、*yincr* と *yval* の値は無視されますが、両方の値は必要です。
xval と *yval* は、プリフェッチされた値が格納される C 変数のアドレスを指定します。
xincr と *yincr* は、リテラル値 -6、-4、-2、0、2、4、6 または整数値をとることができます。
例えば :

```
register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_mpy(xVal, yVal,
                       &xmemory, &xVal, 2,
                       &ymemory, &yVal, 2);
```

このコードは次を生成します :

```
mac w4*w5, A, [w8]+=2, w4, [w10]+=2, w5
```

プロトタイプ :

```
int __builtin_mpy(int a, int b,
  int **xptr, int *xval, int xincr,
  int **yptr, int *yval, int yincr);
```

引数 :

a	整数の被乗数
b	整数の乗数。
xptr	x プリフェッチを指す整数ポインタ
xval	x プリフェッチの値を指す整数ポインタ
xincr	x プリフェッチの整数増分値
yptr	y プリフェッチを指す整数ポインタ
yval	y プリフェッチの値を指す整数ポインタ
yincr	y プリフェッチの整数増分値
AWB	アキュムレータ選択を指す整数ポインタ

戻り値 : クリアされた値の結果をアキュムレータへ返します。

アセンブラ・オペレータ / マシン命令 : mpy

エラー・メッセージ 次の場合に、エラー・メッセージが表示されます :

- 結果がアキュムレータ・レジスタでない
- *xval* は null 値であるが *xptr* は null でない
- *yval* は null 値であるが *yptr* は null でない

__builtin_mpyn

説明 : *-a* x *b* を計算します。また、データをプリフェッチして後で実行される MAC 動作に備えます。
xptr は null に設定されて X プリフェッチが行われないことを表示します。この場合、*xincr* と *xval* の値は無視されますが、両方の値は必要です。
xptr は null に設定されて Y プリフェッチが行われないことを表示します。この場合、*yincr* と *yval* の値は無視されますが、両方の値は必要です。
xval と *yval* は、プリフェッチされた値が格納される C 変数のアドレスを指定します。
xincr と *yincr* は、リテラル値 -6、-4、-2、0、2、4、6 または整数値をとることができます。
例えば :

```
register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_mpy(xVal, yVal,
                       &xmemory, &xVal, 2,
                       &ymemory, &yVal, 2);
```

このコードは次を生成します :

```
mac w4*w5, A, [w8]+=2, w4, [w10]+=2, w5
```

プロトタイプ :

```
int __builtin_mpyn(int a, int b,
  int **xptr, int *xval, int xincr,
  int **yptr, int *yval, int yincr);
```

引数 :

a	整数の被乗数
b	整数の乗数。
xptr	x プリフェッチを指す整数ポインタ
xval	x プリフェッチの値を指す整数ポインタ
xincr	x プリフェッチの整数増分値
yptr	y プリフェッチを指す整数ポインタ
yval	y プリフェッチの値を指す整数ポインタ
yincr	y プリフェッチの整数増分値
AWB	アキュムレータ選択を指す整数ポインタ

戻り値 : クリアされた値の結果をアキュムレータへ返します。

アセンブラ・オペレータ / マシン命令 : mpyn

エラー・メッセージ 次の場合に、エラー・メッセージが表示されます :
- 結果がアキュムレータ・レジスタでない
- *xval* は null 値であるが *xptr* は null でない
- *yval* は null 値であるが *yptr* は null でない

__builtin_msc

説明 : *a* x *b* を計算しアキュムレータから減算します。また、データをプリフェッチして後で実行される MAC 動作に備えます。
xptr は null に設定されて X プリフェッチが行われないことを表示します。この場合、*xincr* と *xval* の値は無視されますが、両方の値は必要です。
xptr は null に設定されて Y プリフェッチが行われないことを表示します。この場合、*yincr* と *yval* の値は無視されますが、両方の値は必要です。
xval と *yval* は、プリフェッチされた値が格納される C 変数のアドレスを指定します。
xincr と *yincr* は、リテラル値 -6、-4、-2、0、2、4、6 または整数値をとることができます。
AWB が null でない場合は、他のアキュムレータが参照変数へ書き込まれます。
例えば :

```
register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_msc(xVal, yVal,
                       &xmemory, &xVal, 2,
                       &ymemory, &yVal, 2, 0);
```

このコードは次を生成します :

```
msc w4*w5, A, [w8]+=2, w4, [w10]+=2, w5
```

プロトタイプ :

```
int __builtin_msc(int a, int b,
  int **xptr, int *xval, int xincr,
  int **yptr, int *yval, int yincr, int *AWB);
```

引数 :

a	整数の被乗数
b	整数の乗数。
xptr	x プリフェッチを指す整数ポインタ
xval	x プリフェッチの値を指す整数ポインタ
xincr	x プリフェッチの整数増分値
yptr	y プリフェッチを指す整数ポインタ
yval	y プリフェッチの値を指す整数ポインタ
yincr	y プリフェッチの整数増分値
AWB	アキュムレータ選択を指す整数ポインタ

戻り値 : クリアされた値の結果をアキュムレータへ返します。

アセンブラ・オペレータ / マシン命令 :

```
msc
```

エラー・メッセージ 次の場合に、エラー・メッセージが表示されます :
- 結果がアキュムレータ・レジスタでない
- *xval* は null 値であるが *xptr* は null でない
- *yval* は null 値であるが *yptr* は null でない

__builtin_mulss

説明 : この関数は、積 *p0* x *p1* を計算します。関数の引数は符号付き整数で、関数の結果は符号付きロング整数です。コマンドライン・オプション -Wconversions を使って、予期しない符号変換を検出することができます。

__builtin_mulss (続き)

プロトタイプ : `signed long __builtin_mulss(const signed int p0, const signed int p1);`

引数 :
p0 被乗数
p1 乗数

戻り値 : 積 *p0* x *p1* の符号付きロング整数値を返します。

アセンブラ・オペレータ / マシン命令 : `mul.ss`

__builtin_mulsu

説明 : この関数は、積 *p0* x *p1* を計算します。関数の引数は整数で、関数の結果は符号付きロング整数です。コマンドライン・オプション -Wconversions を使って、予期しない符号変換を検出することができます。この関数は、オペランド *p1* に対するイミディエイト・モードなどの命令の全アドレッシング・モードをサポートします。

プロトタイプ : `signed long __builtin_mulsu(const signed int p0, const unsigned int p1);`

引数 :
p0 被乗数
p1 乗数

戻り値 : 積 *p0* x *p1* の符号付きロング整数値を返します。

アセンブラ・オペレータ / マシン命令 : `mul.su`

__builtin_mulus

説明 : この関数は、積 *p0* x *p1* を計算します。関数の引数は整数で、関数の結果は符号付きロング整数です。コマンドライン・オプション -Wconversions を使って、予期しない符号変換を検出することができます。この関数は、命令の全アドレッシング・モードをサポートします。

プロトタイプ : `signed long __builtin_mulus(const unsigned int p0, const signed int p1);`

引数 :
p0 被乗数
p1 乗数

戻り値 : 積 *p0* x *p1* の符号付きロング整数値を返します。

アセンブラ・オペレータ / マシン命令 : `mul.us`

__builtin_muluu

説明 : この関数は、積 *p0* x *p1* を計算します。関数の引数は符号なし整数で、関数の結果は符号なしロング整数です。コマンドライン・オプション -Wconversions を使って、予期しない符号変換を検出することができます。この関数は、オペランド *p1* に対するイミディエイト・モードなどの命令の全アドレッシング・モードをサポートします。

プロトタイプ : `unsigned long __builtin_muluu(const unsigned int p0, const unsigned int p1);`

__builtin_muluu (続き)

引数 : *p0* 被乗数
p1 乗数

戻り値 : 積 *p0* x *p1* の符号付きロング整数値を返します。

アセンブラ・オペレータ / マシン命令 : `mul.uu`

__builtin_nop

説明 : この関数は nop 命令を生成します。

プロトタイプ :
```
void __builtin_nop(void);
```

引数 : なし

戻り値 : 無動作 (nop) を返します。

アセンブラ・オペレータ / マシン命令 : `nop`

__builtin_psvpage

説明 : この関数は、パラメータでアドレスを指定したオブジェクトの psv ページ番号を返します。引数 *p* は、EE データ、PSV または実行可能メモリ空間内にあるオブジェクトのアドレスである必要があります。そうしないと、エラー・メッセージが発生して、コンパイルに失敗します。"*MPLAB® C30 C コンパイラ・ユーザーズ・ガイド*"(DS51284) の空間属性を参照してください。

プロトタイプ :
```
unsigned int __builtin_psvpage(const void *p);
```

引数 : *p* オブジェクト・アドレス

戻り値 : パラメータでアドレスを指定したオブジェクトの psv ページ番号を返します。

アセンブラ・オペレータ / マシン命令 : `psvpage`

エラー・メッセージ この関数を正しく使用しないと、次のエラー・メッセージが発生します:
" `__builtin_psvpage()` の引数は、コード、psv、または eedata セクション内にあるオブジェクトのアドレスではありません "
引数は明示的なオブジェクト・アドレスである必要があります。
例えば、*obj* が実行可能または読み出し専用 セクション内にあるオブジェクトである場合、次の構文は有効です :
```
unsigned page = __builtin_psvpage(&obj);
```

__builtin_psvoffset

説明 : この関数は、パラメータでアドレスを指定したオブジェクトの psv ページ・オフセットを返します。引数 *p* は、EE データ、PSV または実行可能メモリ空間内にあるオブジェクトのアドレスである必要があります。そうしないと、エラー・メッセージが発生して、コンパイルに失敗します。"*MPLAB® C30 C コンパイラ・ユーザーズ・ガイド*" (DS51284) の空間属性を参照してください。

プロトタイプ :
```
unsigned int __builtin_psvoffset(const void *p);
```

引数 : *p* オブジェクト・アドレス

__builtin_psvoffset (続き)

戻り値 : パラメータでアドレスを指定したオブジェクトの psv ページ・オフセットを返します。

アセンブラ・オペレータ / マシン命令 :
```
psvoffset
```

エラー・メッセージ この関数を正しく使用しないと、次のエラー・メッセージが発生します:
" __builtin_psvoffset() の引数は、コード、psv、または eedata セクション内にあるオブジェクトのアドレスではありません "
引数は明示的なオブジェクト・アドレスである必要があります。
例えば、*obj* が実行可能または読み出し専用 セクション内にあるオブジェクトである場合、次の構文は有効です :
```
unsigned page = __builtin_psvoffset(&obj);
```

__builtin_return_address

説明 : この関数は、現在の関数またはその関数のコーラーの 1 つの戻りアドレスを返します。*level* 引数について、値 0 は現在の関数の戻りアドレスを、値 1 は現在の関数のコーラーの戻りアドレスを、以下同様に、それぞれ返します。*level* が現在のスタック・サイズを超えると、0 が返されます。この関数は、デバッグ用途に非ゼロ引数を使う場合にのみ使ってください。

プロトタイプ :
```
int __builtin_return_address (const int level);
```

引数 : *level*　コール・スタックをスキャンするフレーム数。

戻り値 : 現在の関数、またはそのコーラーの 1 つの戻りアドレスを返します。

アセンブラ・オペレータ / マシン命令 :
```
return_address
```

__builtin_sac

説明 : *shift* (-8 ～ 7 のリテラル) だけ値をシフトして値を返します。
例えば :
```
  register int value asm("A");
  int result;

  result = __builtin_sac(value,3);
```
このコードは次を生成します :
```
  sac A, #3, w0
```

プロトタイプ :
```
int __builtin_sac(int value, int shift);
```

引数 : *value*　シフトされる整数。
shift　シフト数を表わすリテラル。

戻り値 : シフトした結果をアキュムレータへ返します。

アセンブラ・オペレータ / マシン命令 :
```
sac
```

エラー・メッセージ 次の場合に、エラー・メッセージが表示されます :
- 結果がアキュムレータ・レジスタでない
- シフト値が範囲内のリテラルでない

__builtin_sacr

説明 : *shift* (-8 ～ 7 のリテラル) だけ値をシフトし、CORCONbits.RND コントロール・ビットで指定される丸め処理モードを使って丸め処理された値を返します。
例えば :

```
register int value asm("A");
int result;

result = __builtin_sac(value,3);
```

このコードは次を生成します :

```
sac.r A, #3, w0
```

プロトタイプ :

```
int __builtin_sacr(int value, int shift);
```

引数 : *value* シフトされる整数。
shift シフト数を表わすリテラル。

戻り値 : シフトした結果を CORCON レジスタへ返します。

アセンブラ・オペレータ / マシン命令 :

```
sacr
```

エラー・メッセージ 次の場合に、エラー・メッセージが表示されます :
- 結果がアキュムレータ・レジスタでない
- シフト値が範囲内のリテラルでない

__builtin_sftac

説明 : *shift* だけアキュムレータをシフトします。有効なシフト範囲は -16 ～ 16 です。
例えば :

```
register int result asm("A");
int i;

result = __builtin_sftac(i);
```

このコードは次を生成します :

```
sftac A, w0
```

プロトタイプ :

```
int __builtin_sftac(int shift);
```

引数 : *shift* シフト数を表わすリテラル。

戻り値 : シフトした結果をアキュムレータへ返します。

アセンブラ・オペレータ / マシン命令 :

```
sftac
```

エラー・メッセージ 次の場合に、エラー・メッセージが表示されます :
- 結果がアキュムレータ・レジスタでない
- シフト値が範囲内のリテラルでない

__builtin_subab

説明 : アキュムレータ A とアキュムレータ B の間で減算して、結果を指定したアキュムレータに書き込みます。例えば :

```
register int result asm("A");
result = __builtin_subab();
```

このコードは次を生成します :

```
sub A
```

プロトタイプ :

```
int __builtin_subab(void);
```

__builtin_subab (続き)

引数 : なし

戻り値 : 減算結果をアキュムレータへ返します。

アセンブラ・オペレータ / マシン命令 : subad

エラー・メッセージ 結果がアキュムレータ・レジスタでない場合、エラー・メッセージが表示されます。

__builtin_tblpage

説明 : この関数は、パラメータで指定されたアドレスにあるオブジェクトのテーブル・ページ番号を返します。引数 *p* は、EE データ、PSV または実行可能メモリ空間内にあるオブジェクトのアドレスである必要があります。そうしないと、エラー・メッセージが発生して、コンパイルに失敗します。"*MPLAB® C30 C コンパイラ・ユーザーズ・ガイド*" (DS51284) の空間属性を参照してください。

プロトタイプ :
```
unsigned int __builtin_tblpage(const void *p);
```

引数 : *p* オブジェクト・アドレス

戻り値 : パラメータでアドレスを指定したオブジェクトのテーブル・ページ番号を返します。

アセンブラ・オペレータ / マシン命令 : tblpage

エラー・メッセージ この関数を正しく使用しないと、次のエラー・メッセージが発生します:
"__builtin_tblpage() の引数は、コード、psv、または eedata セクション内にあるオブジェクトのアドレスではありません "
引数は明示的なオブジェクト・アドレスである必要があります。
例えば、*obj* が実行可能または読み出し専用 セクション内にあるオブジェクトである場合、次の構文は有効です :
```
unsigned page = __builtin_tblpage(&obj);
```

__builtin_tbloffset

説明 : この関数は、パラメータでアドレスを指定したオブジェクトのテーブル・ページ・オフセットを返します。引数 *p* は、EE データ、PSV または実行可能メモリ空間内にあるオブジェクトのアドレスである必要があります。そうしないと、エラー・メッセージが発生して、コンパイルに失敗します。"*MPLAB® C30 C コンパイラ・ユーザーズ・ガイド* " の空間属性を参照してください。

プロトタイプ :
```
unsigned int __builtin_tbloffset(const void *p);
```

引数 : *p* オブジェクト・アドレス

戻り値 : パラメータでアドレスを指定したオブジェクトのテーブル・ページ番号オフセットを返します。

アセンブラ・オペレータ / マシン命令 : tbloffset

エラー・メッセージ この関数を正しく使用しないと、次のエラー・メッセージが発生します:
"__builtin_tbloffset() の引数は、コード、psv、または eedata セクション内にあるオブジェクトのアドレスではありません "
引数は明示的なオブジェクト・アドレスである必要があります。
例えば、*obj* が実行可能または読み出し専用 セクション内にあるオブジェクトである場合、次の構文は有効です :
```
unsigned page = __builtin_tbloffset(&obj);
```

別紙 A. ASCII 文字セット

表 A-1: ASCII 文字セット

下位桁 \ 上位桁	0	1	2	3	4	5	6	7
0	NUL	DLE	Space	0	@	P	'	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	Bell	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	¥	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

メモ：

メモ：

世界各国での販売およびサービス

北米

本社
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
テクニカルサポート :
http://support.microchip.com
Web アドレス :
www.microchip.com

アトランタ
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

ボストン
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

シカゴ
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

ダラス
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

デトロイト
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

ココモ
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

ロサンゼルス
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

サンタクララ
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

トロント
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

アジア / 太平洋

アジア太平洋支社
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Habour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

オーストラリア - シドニー
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

中国 - 北京
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

中国 - 成都
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

中国 - 福州
Tel: 86-591-8750-3506
Fax: 86-591-8750-3521

中国 - 香港 SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

中国 - 青島
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

中国 - 上海
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

中国 - 瀋陽
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

中国 - 深川
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

中国 - 順徳
Tel: 86-757-2839-5507
Fax: 86-757-2839-5571

中国 - 武漢
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

中国 - 西安
Tel: 86-29-8833-7250
Fax: 86-29-8833-7256

アジア / 太平洋

インド - バンガロール
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

インド - ニューデリー
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

インド - プネ
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

日本 - 横浜
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

韓国 - 亀尾
Tel: 82-54-473-4301
Fax: 82-54-473-4302

韓国 - ソウル
Tel: 82-2-554-7200
Fax: 82-2-558-5932 または
82-2-558-5934

マレーシア - ペナン
Tel: 60-4-646-8870
Fax: 60-4-646-5086

フィリピン - マニラ
Tel: 63-2-634-9065
Fax: 63-2-634-9069

シンガポール
Tel: 65-6334-8870
Fax: 65-6334-8850

台湾 - 新竹
Tel: 886-3-572-9526
Fax: 886-3-572-6459

台湾 - 高雄
Tel: 886-7-536-4818
Fax: 886-7-536-4803

台湾 - 台北
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

タイ - バンコク
Tel: 66-2-694-1351
Fax: 66-2-694-1350

ヨーロッパ

オーストリア - ヴェルス
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

デンマーク - コペンハーゲン
Tel: 45-4450-2828
Fax: 45-4485-2829

フランス - パリ
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

ドイツ - ミュンヘン
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

イタリア - ミラノ
Tel: 39-0331-742611
Fax: 39-0331-466781

オランダ - ドリューネン
Tel: 31-416-690399
Fax: 31-416-690340

スペイン - マドリッド
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

英国 - ウォーキンガム
Tel: 44-118-921-5869
Fax: 44-118-921-5820

12/08/06