TEXAS INSTRUMENTS

参考資料

TMS320C28x オプティマイジング（最適化）C/C++ コンパイラ

ユーザーズ・マニュアル

2006 Digital Signal Processing Solutions

SPRUEP4

TMS320C28x オプティマイジング（最適化）C/C++ コンパイラ ユーザーズ・マニュアル

対応英文マニュアル：SPRU514B
2005 年 10 月

2006 年 12 月

この資料は、Texas Instruments Incorporated (TI) が英文で記述した資料を、皆様のご理解の一助として頂くために日本テキサス･インスツルメンツ（日本 TI）が英文から和文へ翻訳して作成したものです。
資料によっては正規英語版資料の更新に対応していないものがあります。
日本 TI による和文資料は、あくまでも TI 正規英語版をご理解頂くための補助的参考資料としてご使用下さい。
製品のご検討およびご採用にあたりましては必ず正規英語版の最新資料をご確認下さい。
TI および日本 TI は、正規英語版にて更新の情報を提供しているにもかかわらず、更新以前の情報に基づいて発生した問題や障害等につきましては如何なる責任も負いません。

ご注意

日本テキサス・インスツルメンツ株式会社（以下TIJといいます）及びTexas Instruments Incorporated（TIJの親会社、以下TIJおよびTexas Instruments Incorporatedを総称してTIといいます）は、その製品及びサービスを任意に修正し、改善、改良、その他の変更をし、もしくは製品の製造中止またはサービスの提供を中止する権利を留保します。従いまして、お客様は、発注される前に、関連する最新の情報を取得して頂き、その情報が現在有効かつ完全なものであるかどうかご確認下さい。全ての製品は、お客様とTIとの間に取引契約が締結されている場合は、当該契約条件に基づき、また当該取引契約が締結されていない場合は、ご注文の受諾の際に提示されるTIの標準契約約款に従って販売されます。

TIは、そのハードウェア製品が、TIの標準保証条件に従い販売時の仕様に対応した性能を有していること、またはお客様とTIとの間で合意された保証条件に従い合意された仕様に対応した性能を有していることを保証します。検査およびその他の品質管理技法は、TIが当該保証を支援するのに必要とみなす範囲で行なわれております。各デバイスの全てのパラメーターに関する固有の検査は、政府がそれ等の実行を義務づけている場合を除き、必ずしも行なわれておりません。

TIは、製品のアプリケーションに関する支援もしくはお客様の製品の設計について責任を負うことはありません。TI製部品を使用しているお客様の製品及びそのアプリケーションについての責任はお客様にあります。TI製部品を使用したお客様の製品及びアプリケーションについて想定されうる危険を最小のものとするため、適切な設計上および操作上の安全対策は、必ずお客様にてお取り下さい。

TIは、TIの製品もしくはサービスが使用されている組み合せ、機械装置、もしくは方法に関連しているTIの特許権、著作権、回路配置利用権、その他のTIの知的財産権に基づいて何らかのライセンスを許諾するということは明示的にも黙示的にも保証も表明もしておりません。TIが第三者の製品もしくはサービスについて情報を提供することは、TIが当該製品もしくはサービスを使用することについてライセンスを与えるとか、保証もしくは是認するということを意味しません。そのような情報を使用するには第三者の特許その他の知的財産権に基づき当該第三者からライセンスを得なければならない場合もあり、またTIの特許その他の知的財産権に基づきTIからライセンスを得て頂かなければならない場合もあります。

TIのデータ・ブックもしくはデータ・シートの中にある情報を複製することは、その情報に一切の変更を加えること無く、且つその情報と結び付られた全ての保証、条件、制限及び通知と共に複製がなされる限りにおいて許されるものとします。当該情報に変更を加えて複製することは不公正で誤認を生じさせる行為です。TIは、そのような変更された情報や複製については何の義務も責任も負いません。

TIの製品もしくはサービスについてTIにより示された数値、特性、条件その他のパラメーターと異なる、あるいは、それを超えてなされた説明で当該TI製品もしくはサービスを再販売することは、当該TI製品もしくはサービスに対する全ての明示的保証、及び何らかの黙示的保証を無効にし、且つ不公正で誤認を生じさせる行為です。TIは、そのような説明については何の義務も責任もありません。

なお、日本テキサス・インスツルメンツ株式会社半導体集積回路製品販売用標準契約約款もご覧下さい。
http://www.tij.co.jp/jsc/docs/stdterms.htm

Copyright © 2006, Texas Instruments Incorporated
日本語版 日本テキサス・インスツルメンツ株式会社

弊社半導体製品の取り扱い・保管について

半導体製品は、取り扱い、保管・輸送環境、基板実装条件によっては、お客様での実装前後に破壊/劣化、または故障を起こすことがあります。

弊社半導体製品のお取り扱い、ご使用にあたっては下記の点を遵守して下さい。

1. 静電気
   - 素手で半導体製品単体を触らないこと。どうしても触る必要がある場合は、リストストラップ等で人体からアースをとり、導電性手袋等をして取り扱うこと。
   - 弊社出荷梱包単位（外装から取り出された内装及び個装）又は製品単品で取り扱いを行う場合は、接地された導電性のテーブル上で（導電性マットにアースをとったもの等）、アースをした作業者が行うこと。また、コンテナ等も、導電性のものを使うこと。
   - マウンタやはんだ付け設備等、半導体の実装に関わる全ての装置類は、静電気の帯電を防止する措置を施すこと。
   - 前記のリストストラップ・導電性手袋・テーブル表面及び実装装置類の接地等の静電気帯電防止措置は、常に管理されその機能が確認されていること。
2. 温・湿度環境
   - 温度：0～40℃、相対湿度：40～85％で保管・輸送及び取り扱いを行うこと。（但し、結露しないこと。）
   - 直射日光があたる状態で保管・輸送しないこと。
3. 防湿梱包
   - 防湿梱包品は、開封後は個別推奨保管環境及び期間に従い基板実装すること。
4. 機械的衝撃
   - 梱包品（外装、内装、個装）及び製品単品を落下させたり、衝撃を与えないこと。
5. 熱衝撃
   - はんだ付け時は、最低限260℃以上の高温状態に、10秒以上さらさないこと。（個別推奨条件がある時はそれに従うこと。）
6. 汚染
   - はんだ付け性を損なう、又はアルミ配線腐食の原因となるような汚染物質（硫黄、塩素等ハロゲン）のある環境で保管・輸送しないこと。
   - はんだ付け後は十分にフラックスの洗浄を行うこと。（不純物含有率が一定以下に保証された無洗浄タイプのフラックスは除く。）

以上

2001.11

まえがき

最初にお読み下さい

本書について

本書は、以下のコンパイラ・ツールの使用方法について説明したものです。

- ❏ コンパイラ
- ❏ ポストリンク・オプティマイザ
- ❏ ライブラリ作成ユーティリティ
- ❏ インターリスト・ユーティリティ
- ❏ C++ ネーム・デマングラ・ユーティリティ

TMS320C28x™ C/C++ コンパイラは、国際標準化機構（ISO）準拠の標準 C および C++ コードを受け入れ、TMS320C28x デバイス用のアセンブリ言語ソース・コードを生成します。コンパイラは、1989 バージョンの C 言語をサポートします。

本書では、C/C++ コンパイラの特性について説明します。本書では、C/C++ プログラムの作成方法を理解していることを前提とします。ISO C 規格に準拠する C 言語についてはカーニハンとリッチーの『The C Programming Language』（第 2 版）に解説してあります。必要に応じて、本書の参考文献としてお読みください。本書において、ISO C に相違するものとして K&R C を参照する場合には、カーニハンとリッチーの『The C Programming Language』（第 1 版）で記述されている C 言語を示します。

本書の C/C++ コンパイラに関する情報を使用する前に、C/C++ コンパイラ・ツールをインストールしておいてください。

表記規則

本書では、次の表記規則を使用します。

- TMS320C28x デバイスは C28x として参照されます。
- プログラム・リスト、プログラム例、および対話表示は、タイプライタの活字に似た特殊な活字（`special typeface`）で示してあります。例は、強調のため、ボールド（**`bold version`**）で示してあります。対話表示についても、ユーザが入力するコマンドとシステムが表示する項目（プロンプト、コマンド出力、エラー・メッセージなど) と区別するために、ボールド（**`bold version`**）で示しています。

  C コードの例を次に示します。

```
#include <stdio.h>

main()

{
     printf("hello, world\n");
}
```

- 構文の記述、命令、コマンド、疑似命令はボールド（**bold**）の特殊文字、パラメータはイタリック体（*italics*）で示します。構文でボールドの部分は、その表記どおりに入力します。構文でイタリックの部分は、入力する情報の型を示しています。コマンド行で入力する構文は、次のように縁取りのある枠囲みの中に示しています。

  cl2000 -v28 [*options*] [*filenames*] [*-z* [*link_options*] [*object files*]]

  テキスト・ファイルで使用する構文は、次のように縁取りのある枠囲みに左詰めで示しています。

  inline *return-type function-name* (*parameter declarations*) **{** *function* **}**

- 大括弧（ [ ] ）は、オプション・パラメータを示します。パラメータを使用する場合、括弧内の情報を指定します。括弧そのものは入力する必要がありません。オプション・パラメータ付きのコマンドの例を次に示します。

  ac2000 *inputfile* [*outputfile*] [*options*]

  ac2000 コマンドには、3 つのパラメータがあります。最初のパラメータ *inputfile* は、必須です。2 番目と 3 番目のパラメータ *outputfile* と *options* は、それぞれオプションです。

❑ 中括弧（{ }）は、それに囲まれているパラメータのいずれかを選択する必要があることを示しています。中括弧そのものは入力不要です。実際の構文に含まれていなくても、-c または -cr のどちらかのオプションを選択する必要があることを示す中括弧が付いたコマンドの例を、次に示します。

cl2000 -v28 -z {**-c** | **-cr**} *filenames* [**-o** *name.out*] **-l** *libraryname*

❑ TMS320C2800 コアは TMS320C28x および C28x として参照されます。

当社発行の関連文献

以下の文献は、TMS320C28x および関連サポート・ツールについて説明しています。当社の刊行物を入手するには、タイトルと文献番号（表紙に記載）をご確認の上、プロダクト・インフォメーション・センター（PIC）、www.tij.co.jp/PIC/ にお問い合わせください。

TMS320C28x DSP CPU 及びインストラクション・セット リファレンス・ガイド（文献番号 SPRU813）は、中央演算処理ユニット（CPU）および TMS320C28x™ の固定小数点デジタル・シグナル・プロセッサ（DSP）のアセンブリ言語命令について解説しています。また、TMS320C28x DSP で使用できるエミュレーション機能についても解説しています。

TMS320C2xx User's Guide（文献番号 SPRU127）は、TMS320C2xx™ の 16 ビット固定小数点デジタル・シグナル・プロセッサのハードウェア面について解説しています。アーキテクチャ、命令セット、およびオンチップ・ペリフェラルを解説しています。

TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル（文献番号 SPRUEP3）は、TMS320C28x™ デバイスのアセンブリ言語ツール（アセンブラ、リンカ、その他のアセンブリ言語コードの開発用ツール）、アセンブラ疑似命令、マクロ、共通オブジェクト・ファイル・フォーマット、およびシンボリック・デバッグ用の疑似命令について解説しています。

Code Composer Studio 入門マニュアル（文献番号 SPRU567）は、Code Composer Studio 開発環境を使用して組み込み用のリアルタイム DSP アプリケーションを作成し、デバッグする方法について解説しています。

関連文献

本書の参考文献として、次の文献を参考にできます。

ISO/IEC 9899:1999, International Standard - Programming Languages - C (The C Standard)、国際標準化機構刊行

ISO/IEC 9899:1989, International Standard - Programming Languages - C (The 1989 C Standard)、国際標準化機構刊行

ISO/IEC 14882-1998, International Standard - Programming Languages - C++ (The C++ Standard)、国際標準化機構刊行

ANSI X3.159-1989, Programming Language - C (Alternate version of the 1989 C Standard)、米国規格協会刊行

C: A Reference Manual（第 4 版）、Samuel P. Harbison と Guy L. Steele Jr. の共著、Prentice-Hall（Englewood Cliffs, New Jersey）刊行（1988）

Programming in C、Kochan, Steve G. 著、Hayden Book Company 刊行

The Annotated C++ Reference Manual、Margaret A. Ellis と Bjarne Stroustrup の共著、Addison-Wesley Publishing Company（Reading, Massachusetts）刊行（1990）

The C Programming Language（第 2 版）、Brian W. Kernighan と Dennis M. Ritchie の共著、Prentice-Hall（Englewood Cliffs, New Jersey）刊行（1988）

The C++ Programming Language（第 2 版）、Bjarne Stroustrup 著、Addison-Wesley Publishing Company（Reading, Massachusetts）刊行（1990）

Understanding and Using COFF、Gintaras R. Gircys 著、O'Reilly and Associates, Inc. 刊行

商標

Code Composer Studio、TMS320C28x、および C28x は Texas Instruments の商標です。

Intel、i286、i386、および i486 は Intel Corporation の商標です。

MCS-86 は Intel Corporation の商標です。

Motorola および Motorola-S は Motorola, Inc. の商標です。

Tektronix は Tektronix, Inc. の商標です。

Windows および Windows NT は Microsoft Corporation の登録商標です。

目次

1 はじめに 1-1
TMS320C28x ソフトウェア開発ツールの概要について説明します。

1.1 ソフトウェア開発ツールの概要 1-2
1.2 C/C++ コンパイラの概要 1-5
1.2.1 C/C++ 言語機能 1-5
1.2.2 出力ファイル 1-6
1.2.3 コンパイラ・インターフェイス 1-6
1.2.4 コンパイラの操作 1-7
1.2.5 ユーティリティ 1-7

2 C/C++ コンパイラの使用方法 2-1
C/C++ コンパイラの操作方法とその概要を説明します。具体的には、C/C++ ソース・ファイルをコンパイルし、アセンブルし、リンクするコンパイラの起動方法について説明します。インターリスト・ユーティリティ、オプション、およびコンパイラ・エラーについて考察します。

2.1 コンパイラについて 2-2
2.2 C/C++ コンパイラの起動方法 2-4
2.3 オプションによるコンパイラの動作の変更 2-5
2.3.1 使用頻度の高いオプション 2-13
2.3.2 シンボリック・デバッグおよびプロファイルのオプション 2-15
2.3.3 マシン固有のオプション 2-16
2.3.4 ファイル名の指定方法 2-18
2.3.5 コンパイラによるファイル名の解釈方法の変更（-fa、-fc、-fg、-fo、および -fp オプション）........ 2-19
2.3.6 コンパイラによるファイル名の拡張子の解釈方法とファイル作成時の拡張子の付け方の変更（-ea、-ec、-eo、および -ep オプション）........ 2-20
2.3.7 ディレクトリの指定方法 2-21
2.3.8 アセンブラを制御するオプション 2-22
2.3.9 非推奨オプション 2-23
2.4 環境変数によるコンパイラの動作の変更 2-24
2.4.1 コンパイラが検索する代替ディレクトリの指定方法（C_DIR）........ 2-24
2.4.2 デフォルトのコンパイラ・オプションの設定方法（C_OPTION および C2000_C_OPTION）........ 2-24
2.4.3 一時ファイルのディレクトリ指定方法（TMP）........ 2-26
2.5 プリプロセッサの制御方法 2-27
2.5.1 事前定義マクロ名 2-27
2.5.2 #include ファイル検索パス 2-28
2.5.3 前処理リスト・ファイルの生成方法（-ppo オプション）........ 2-30

2.5.4 前処理後のコンパイルの続行方法（-ppa オプション）........ 2-30
2.5.5 コメント付き前処理リスト・ファイルの生成方法（-ppc オプション）........ 2-31
2.5.6 行の制御情報付き前処理リスト・ファイルの生成（-ppl オプション）........ 2-31
2.5.7 Make ユーティリティ用の前処理出力の生成方法（-ppd オプション）........ 2-31
2.5.8 #include 疑似命令で組み込むファイルのリストの生成方法（-ppi オプション）2-31
2.6 診断メッセージの概要........ 2-32
2.6.1 診断の制御方法........ 2-34
2.6.2 診断抑止オプションの使用方法........ 2-35
2.6.3 その他のメッセージ........ 2-36
2.7 クロスリファレンス・リスト情報の生成方法（-px オプション）........ 2-37
2.8 ロー・リスト・ファイルの生成方法（-pl オプション）........ 2-38
2.9 インライン関数展開の使用方法........ 2-40
2.9.1 組み込み演算子のインライン展開........ 2-40
2.9.2 自動インライン展開........ 2-40
2.9.3 保護されない定義制御インライン展開........ 2-41
2.9.4 保護されたインライン展開と _INLINE プリプロセッサ・シンボル........ 2-41
2.9.5 インライン展開の制約事項........ 2-43
2.10 インターリストの使用方法........ 2-44

3 コードの最適化方法........ 3-1

C/C++ コードの最適化方法について説明します。オプティマイザ使用時に行われる最適化の種類も説明します。

3.1 C/C++ コンパイラのオプティマイザの使用方法........ 3-2
3.2 ファイルレベルの最適化の実行........ 3-4
3.2.1 ファイルレベルの最適化の実行（-ol オプション）........ 3-4
3.2.2 最適化情報ファイルの作成方法（-on オプション）........ 3-5
3.3 プログラムレベルの最適化の実行（-pm および -O3 オプション）........ 3-6
3.3.1 プログラムレベルの最適化の制御（-op**n** オプション）........ 3-6
3.3.2 C/C++ とアセンブリを組み合わせた場合の最適化に関する注意事項........ 3-8
3.4 オプティマイザを使用する場合の特別な注意事項........ 3-10
3.4.1 最適化コード内で asm 文を使用する場合の注意........ 3-10
3.4.2 必要なメモリ・アクセスを行うために volatile キーワードを使用する........ 3-10
3.5 自動インライン展開（-oi オプション）........ 3-12
3.6 インターリスト・ユーティリティをオプティマイザと組み合わせて使用する方法........ 3-13
3.7 最適化されたコードのデバッグ方法........ 3-16
3.7.1 最適化されたコードのデバッグ（-g、--symdebug:dwarf、--symdebug:coff、および -O オプション）........ 3-16
3.7.2 最適化されたコードのプロファイル方法........ 3-17
3.8 コード・サイズ最適化の強化（-ms オプション）........ 3-18
3.9 実行できる最適化の種類........ 3-20
3.9.1 コストに基づいたレジスタ割り当て........ 3-21
3.9.2 エイリアスの明確化........ 3-22
3.9.3 データ・フローの最適化........ 3-22

3.9.4 式の簡略化 3-23
3.9.5 関数のインライン展開 3-24
3.9.6 誘導変数と強度換算 3-26
3.9.7 ループ不変コードの移動 3-26
3.9.8 ループの循環 3-26
3.9.9 レジスタ変数 3-26
3.9.10 レジスタ・トラッキング / レジスタ・ターゲッティング 3-26
3.9.11 テール結合 3-28
3.9.12 ゼロとの比較を除去する方法 3-30

4 C/C++ コードのリンク方法 4-1

コンパイル・ステップと独立または手順の一部としてリンクする方法と、C/C++ コードをリンクする上で特定の要求事項に合わせる方法を説明します。

4.1 リンカの起動方法（-z オプション）........ 4-2
4.1.1 独立したステップとしてリンカを起動する方法 4-2
4.1.2 コンパイル・ステップの一部としてリンカを起動する方法 4-3
4.1.3 リンカを無効にする方法（-cコンパイラ・オプション）........ 4-4
4.2 リンカ・オプション 4-5
4.3 リンク・プロセスの制御方法 4-7
4.3.1 ランタイムサポート・ライブラリのリンク方法 4-7
4.3.2 実行時の初期化 4-8
4.3.3 割り込みベクタによる初期化 4-8
4.3.4 グローバル・オブジェクト・コンストラクタ 4-9
4.3.5 初期化のタイプの指定方法 4-9
4.3.6 セクションをメモリ内のどこに配置するかを指定する方法 4-10
4.3.7 リンカ・コマンド・ファイルの例 4-11
4.4 C28x コードと C2XLP コードのリンク方法 4-13

5 ポストリンク・オプティマイザ 5-1

ポストリンク・オプティマイザの起動方法について説明します。またポストリンク・オプティマイザがどのようにコードを改良しているかを解説し、ポストリンク・プロセスでの絶対リスタの役割を規定します。

5.1 ソフトウェア開発フローにおけるポストリンク・オプティマイザの役割 5-2
5.2 冗長な DP ロード命令を除去する方法 5-4
5.3 分岐間で DP 値を調べる方法 5-4
5.4 関数呼び出し間で DP 値を調べる方法 5-5
5.5 他のポストリンク最適化 5-6
5.6 ポストリンク最適化を制御する方法 5-7
5.6.1 ファイルを除外する方法（-ex オプション）........ 5-7
5.6.2 アセンブリ・ファイル内でポストリンク最適化を制御する方法 5-7
5.6.3 ポストリンク・オプティマイザの出力を保持する方法（-k オプション）........ 5-8
5.6.4 関数呼び出し間での最適化を無効にする（-nf オプション）........ 5-8
5.7 ポストリンク・オプティマイザ使用時の制約事項 5-9
5.8 出力ファイルの指定方法（-o オプション）........ 5-10

6　TMS320C28x C/C++ 言語の実装 6-1

TMS320C28x C/C++ コンパイラ固有の仕様を ANSI/ISO C/C++ 仕様と関連付けて説明します。

6.1 TMS320C28x C の特性 6-2
6.1.1 識別子と定数 6-2
6.1.2 データ型 6-2
6.1.3 変換 6-3
6.1.4 式 6-3
6.1.5 宣言 6-3
6.1.6 プリプロセッサ 6-4
6.1.7 ヘッダ・ファイル 6-4
6.2 TMS320C28x C++ の特性 6-5
6.3 組み込み C++ モードの有効化（-pe オプション） 6-6
6.4 データ型 6-7
6.4.1 64 ビット整数のサポート 6-9
6.4.2 C28x long double 浮動小数点型による変更 6-9
6.5 レジスタ変数 6-11
6.6 asm 文 6-12
6.7 キーワード 6-14
6.7.1 const キーワード 6-14
6.7.2 volatile キーワード 6-14
6.7.3 cregister キーワード 6-15
6.7.4 interrupt キーワード 6-16
6.7.5 ioport キーワード 6-17
6.7.6 far キーワード 6-19
6.7.7 ラージ・メモリ・モデルを使用する方法（-ml オプション） 6-21
6.7.8 C++ で far メモリをアクセスする組み込み関数を使用する方法 6-22
6.8 プラグマ疑似命令 6-25
6.8.1 CODE_SECTION プラグマ 6-25
6.8.2 DATA_SECTION プラグマ 6-28
6.8.3 FAST_FUNC_CALL プラグマ 6-29
6.8.4 FUNC_EXT_CALLED プラグマ 6-30
6.8.5 INTERRUPT プラグマ 6-31
6.9 静的変数とグローバル変数の初期化方法 6-32
6.9.1 リンカを使った静的変数とグローバル変数の初期化方法 6-32
6.9.2 const 型修飾子を使った静的変数とグローバル変数の初期化方法 6-33
6.10 リンク名の生成方法 6-34
6.11 K&R C との互換性 6-35
6.12 言語モードの変更方法（-pk、-pr、および -ps オプション） 6-37
6.13 厳密 ANSI/ISO モードと緩和 ANSI/ISO モードの有効化（-ps および -pr　オプション） 6-38
6.14 コンパイラの限界 6-38

7 ランタイム環境 7-1

コンパイラがどのように TMS320C28x アーキテクチャを使用しているかに関する技術情報を説明します。具体的には、メモリやレジスタの規則、スタック構成、関数呼び出し規則、システムの初期化、および TMS320C28x C/C++ コンパイラによる最適化について説明します。C/C++ プログラムにアセンブリ言語をインターフェイスするために必要な情報を提供します。

7.1 メモリ・モデル 7-2
7.1.1 セクション 7-3
7.1.2 C/C++ システム・スタック 7-5
7.1.3 .const/.econst をプログラム・メモリに割り当てる方法 7-6
7.1.4 動的なメモリ割り当て 7-7
7.1.5 変数の初期化 7-8
7.1.6 静的変数とグローバル変数へのメモリ割り当て方法 7-8
7.1.7 フィールド / 構造体の位置合わせ 7-9
7.1.8 文字列定数 7-9
7.1.9 far 文字列定数 7-10
7.2 レジスタ規則 7-11
7.2.1 TMS320C28x レジスタの使用と保存 7-11
7.2.2 ステータス・レジスタ 7-12
7.3 関数呼び出し規則 7-13
7.3.1 関数の呼び出し方法 7-14
7.3.2 呼び出し先の関数の対応方法 7-15
7.3.3 呼び出し先関数の特殊な場合（ラージ・フレーム） 7-17
7.3.4 引数とローカル変数のアクセス方法 7-17
7.3.5 フレームを割り当てる方法とメモリ内の 32 ビット値をアクセスする方法 7-18
7.4 アセンブリ言語と C/C++ 言語間のインターフェイス 7-19
7.4.1 C/C++ コードでのアセンブリ言語モジュールの使用方法 7-19
7.4.2 アセンブリ言語のグローバル変数にアクセスする方法 7-23
7.4.3 アセンブリ言語定数へのアクセス 7-24
7.4.4 インライン・アセンブリ言語の使用方法 7-25
7.4.5 組み込み関数（intrinsics）を使用してアセンブリ言語文にアクセスする方法. 7-26
7.5 割り込み処理 7-35
7.5.1 割り込みについての一般的なポイント 7-35
7.5.2 C/C++ 割り込みルーチンの使用方法 7-36
7.6 整数式の分析 7-37
7.6.1 RTS 呼び出しを使って評価される整数演算 7-37
7.6.2 16 ビット乗算での上位 16 ビットへの C/C++ コードによるアクセス 7-38
7.7 浮動小数点式の分析 7-39
7.8 システムの初期化 7-40
7.8.1 ランタイム・スタック 7-40
7.8.2 変数の自動初期化 7-41
7.8.3 グローバル・コンストラクタ 7-41
7.8.4 初期化テーブル 7-42
7.8.5 実行時の変数の自動初期化 7-45

7.8.6 ロード時の変数の自動初期化 7-46

8 ランタイムサポート関数 8-1

C/C++ コンパイラに付属しているライブラリやヘッダ・ファイル、さらにマクロ、関数、型宣言について説明します。ランタイムサポート関数をカテゴリ（ヘッダ）に従ってまとめ、非 ANSI/ISO ランタイムサポート関数をアルファベット順に参照できるようにしています。

8.1 ライブラリ 8-2
8.1.1 オブジェクト・ライブラリとコードのリンク方法 8-2
8.1.2 ライブラリ関数の修正方法 8-2
8.1.3 さまざまなオプションによるライブラリの作成方法 8-3
8.2 far メモリのサポート 8-4
8.2.1 ランタイムサポート関数の far バージョン 8-4
8.2.2 ランタイムサポートにおけるグローバル変数と静的変数 8-5
8.2.3 C における far メモリの動的割り当て 8-5
8.2.4 ラージ・メモリ・モデル用のランタイムサポート・ライブラリの作成方法 8-6
8.2.5 C++ における far メモリの動的割り当て 8-7
8.3 C 入出力関数 8-8
8.3.1 低レベル入出力実装の概要 8-9
8.3.2 C 入出力用デバイスの追加方法 8-10
8.4 ヘッダ・ファイル 8-18
8.4.1 診断メッセージ（assert.h/cassert） 8-19
8.4.2 文字の判別と変換 （ctype.h/cctype） 8-20
8.4.3 エラー報告（errno.h/cerrno） 8-20
8.4.4 低レベル入出力関数（file.h） 8-20
8.4.5 制限値（float.h/cfloat と limits.h/climits） 8-21
8.4.6 整数型のフォーマット変換（inttypes.h） 8-23
8.4.7 代替スペリング（iso646.h/ciso646） 8-24
8.4.8 浮動小数点算術（math.h/cmath） 8-24
8.4.9 非ローカル・ジャンプ（setjmp.h/csetjmp） 8-24
8.4.10 可変引数（stdarg.h/cstdarg） 8-25
8.4.11 標準定義 （stddef.h/cstddef） 8-25
8.4.12 整数型（stdint.h） 8-26
8.4.13 入出力関数（stdio.h/cstdio） 8-27
8.4.14 汎用ユーティリティ（stdlib.h/cstdlib） 8-28
8.4.15 文字列関数（string.h/cstring） 8-29
8.4.16 時間関数（time.h/ctime） 8-29
8.4.17 例外処理（exception と stdexcept） 8-30
8.4.18 動的メモリ管理（new） 8-30
8.4.19 ランタイム型情報（typeinfo） 8-30
8.5 ランタイムサポート関数およびマクロのまとめ 8-31
8.6 ランタイムサポート関数およびマクロについて 8-43

9 ライブラリ作成ユーティリティ 9-1

コードをコンパイルするときに指定するオプションを使用して、ランタイムサポート・ライブラリをカスタマイズするユーティリティについて説明します。このユーティリティを用いてヘッダ・ファイルをディレクトリに配置したり、ソース・アーカイブからカスタム・ライブラリを作成したりすることができます。

9.1 ライブラリ作成ユーティリティの起動方法 9-2
9.2 ライブラリ作成ユーティリティのオプション 9-3
9.3 オプションのまとめ 9-4

10 C++ネーム・デマングラ・ユーティリティ 10-1

C++ ネーム・デマングラについて説明し、その起動方法および使用方法について解説します。

10.1 C++ ネーム・デマングラの起動方法 10-2
10.2 C++ ネーム・デマングラのオプション 10-3
10.3 C++ ネーム・デマングラの使用例 10-4

A 用語集 A-1

本書で使用している用語や頭字語について定義します。

図 1-1　TMS320C28x ソフトウェア開発フロー 1-2
図 2-1　C/C++ コンパイラのフロー図 2-3
図 5-1　TMS320C28x ソフトウェア開発フローにおけるポストリンク・オプティマイザ 5-2
図 7-1　関数呼び出し時のスタックの使用 7-13
図 7-2　.cinit セクション内の初期化レコードのフォーマット（デフォルトと far データ） 7-42
図 7-3　.pinit セクションのフォーマット 7-44
図 7-4　実行時の自動初期化 7-45
図 7-5　ロード時の自動初期化 7-46
図 8-1　入出力関数におけるデータ構造の相互作用 8-9
図 8-2　ストリーム・テーブル内の最初の 3 つのストリーム 8-10

表

表 2-1　コンパイラ・オプションのまとめ 2-6
表 2-2　コンパイラ下位互換オプションのまとめ 2-23
表 2-3　事前定義マクロ名 2-28
表 2-4　ロー・リスト・ファイルの識別子 2-38
表 2-5　ロー・リスト・ファイル診断識別子 2-38
表 3-1　-O3 と組み合わせて使用できるオプション 3-4
表 3-2　-ol オプションのレベルの選択方法 3-4
表 3-3　-on オプションのレベルの選択方法 3-5
表 3-4　-op オプションのレベルの選択方法 3-7
表 3-5　-op オプションを使用する場合の特別な注意事項 3-7
表 4-1　コンパイラが作成するセクション 4-10
表 6-1　TMS320C28x C のデータ型 6-8
表 6-2　有効な制御レジスタ 6-15
表 7-1　レジスタの使用および保存に関する規則 7-11
表 7-2　ステータス・レジスタ・フィールド 7-12
表 7-3　TMS320C28x C/C++ コンパイラの組み込み関数 7-27
表 8-1　整数型の範囲に関する制限値を指定するマクロ（limits.h） 8-21
表 8-2　浮動小数点の範囲に関する制限値を指定するマクロ（float.h） 8-22
表 8-3　ランタイムサポート関数およびマクロのまとめ 8-31
表 8-4　strftime の書式パラメータ文字 8-97
表 9-1　オプションとその機能のまとめ 9-4

例

例 2-1　inline キーワードの使用方法 2-41
例 2-2　ランタイムサポート・ライブラリでの _INLINE プリプロセッサ・シンボルの使用方法 2-42
例 2-3　差し込み後のアセンブリ言語ファイル 2-45
例 3-1　-O2 および -os オプションを指定してコンパイルされた C コード 3-13
例 3-2　-O2、-os、および -ss オプションを指定してコンパイルされた C コード 3-15
例 3-3　-ms オプションを指定してコンパイルされたコード 3-18
例 3-4　強度換算、誘導変数の除去、レジスタ変数 3-21
例 3-5　データ・フローの最適化と式の簡略化 3-23
例 3-6　関数のインライン展開 3-24
例 3-7　レジスタ・トラッキング / レジスタ・ターゲッティング 3-27
例 3-8　テール結合 3-28
例 3-9　ゼロとの比較を削除する方法 3-30
例 4-1　リンカ・コマンド・ファイル 4-12
例 4-2　C2xx コードと C2XLP コードをリンクする veneer 関数 4-13
例 6-1　制御レジスタの宣言方法と使用方法 6-16
例 6-2　ioport キーワードの使用方法 6-17
例 6-3　変数を宣言する方法 6-20
例 6-4　CODE_SECTION プラグマの使用方法 6-26
例 6-5　DATA_SECTION プラグマの使用方法 6-28
例 6-6　FAST_FUNC_CALL プラグマの使用方法 6-29
例 7-1　アセンブリ言語関数 7-22
例 7-2　C/C++ から .bss で定義されている変数にアクセスする方法 7-23
例 7-3　.bss 内に定義されていない変数に C からアクセスする方法 7-24
例 7-4　C からアセンブリ言語定数にアクセスする方法 7-25
例 10-1　名前のマングリング 10-4
例 10-2　C++ ネーム・デマングラ・ユーティリティ実行後の結果 10-6

注

不等号括弧でパス情報を指定する方法 2-30
関数のインライン展開によりコード・サイズが大幅に増大する可能性がある 2-40
-O3 オプションによる最適化とインライン展開 3-12
インライン展開とコード・サイズ 3-12
パフォーマンスとコード・サイズに与える影響 3-14
シンボリック・デバッグ・オプションを指定するとパフォーマンスとコード・サイズに影響を与える 3-16
プロファイル・ポイント 3-17
_c_int00 シンボル 4-8
far リンクの問題 4-10
TMS320C28x のバイトは 16 ビット 6-8
RPT 命令に対して 1 つの asm 文を使用してください 6-12
asm 文により C/C++ 環境を損なわないでください 6-13
C++ での far のサポート 6-19
ポインタの使い分け 6-19
リンカのメモリ・マップ定義 7-2
.stack セクションのリンク方法 7-5
スタック・オーバーフロー 7-5
ヒープ・サイズの制約事項 7-8
PAGE0 モード・ビットはリセットする必要がある 7-17
asm 文の使用方法 7-25
複雑な式の危険性 7-38
変数の初期化方法 7-41
変数の初期化方法 7-42
const 変数の初期化 7-43
far 組み込み関数の使用方法 8-7
一意な関数名を使用してください 8-10
ユーザ固有の clock 関数の記述 8-30
ユーザ固有の clock 関数の記述 8-52
C28x の char がホストのバイトと異なる場合の fread の使用方法 8-66
getenv 関数はターゲットシステム固有 8-71
time 関数はターゲット・システム固有 8-107
ランタイムサポート・ライブラリ 9-2
TMS320C28x のバイトは 16 ビット A-8

第 1 章

はじめに

TMS320C28x™ は、オプティマイジング（最適化）C/C++ コンパイラ、アセンブラ、リンカ、および各種ユーティリティなどが完備した開発ツールによってサポートされています。

本章では、上述のツールの概要を説明し、オプティマイジング C/C++ コンパイラの機能について説明します。アセンブラとリンカの詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

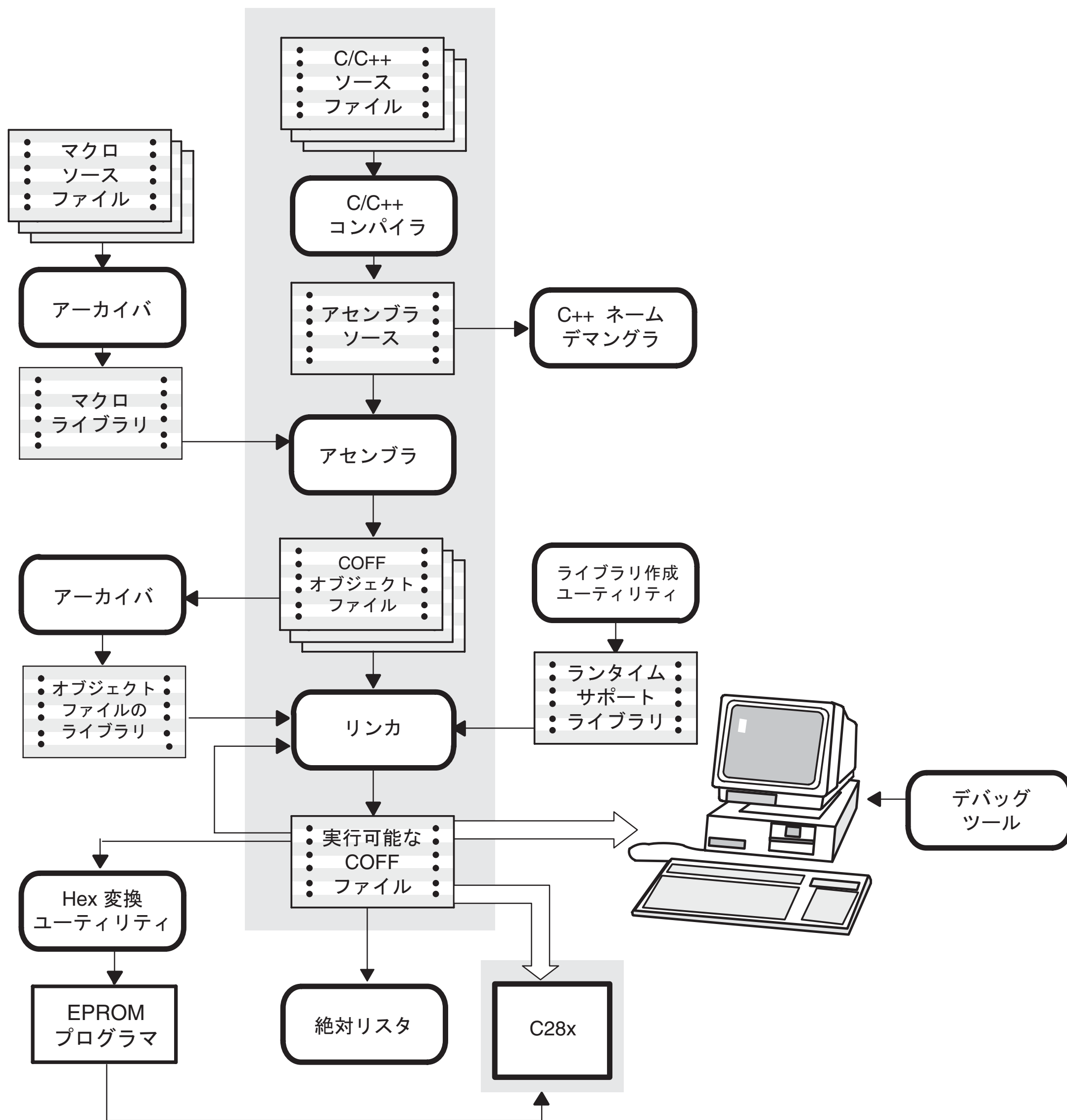
項目 **ページ**

1.1	**ソフトウェア開発ツールの概要**	**1-2**
1.2	**C/C++ コンパイラの概要**	**1-5**

1.1　ソフトウェア開発ツールの概要

図 1-1 に、TMS320C28x ソフトウェア開発フローを示します。図の中の陰影を付けた部分は、C/C++ 言語プログラムのソフトウェア開発における最も一般的な経路です。それ以外の部分は、開発プロセスを補強する周辺機能です。

図 1-1.　TMS320C28x ソフトウェア開発フロー



以下のリストは、図 1-1 のツールについて説明しています。

- **C/C++ コンパイラ**は C/C++ ソース・コードを受け付け、C28x アセンブリ言語ソース・コードを作成します。**オプティマイザ**は、コンパイラの一部です。このオプティマイザは、コードを変更して C/C++ プログラムの効率を高めます。

  C コンパイラおよびオプティマイザの起動方法については、第 2 章「C/C++ コンパイラの使用方法」を参照してください。

- **アセンブラ**（cl2000 -v28）は、アセンブリ言語ソース・コードを機械語のオブジェクト・ファイルに変換します。この機械語は、COFF（共通オブジェクト・ファイル・フォーマット）に基づいています。アセンブラの使用方法については、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

- **リンカ**（cl2000 -v28 -z）は、オブジェクト・ファイルを結合して、1 つの実行可能なオブジェクト・モジュールを作成します。また、リンカは実行可能なモジュールを作成する際に、シンボルへの参照を調整し、外部参照を解決します。リンカが入力として受け付けるのは、再配置可能な COFF オブジェクト・ファイルとオブジェクト・ライブラリです。リンカの詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

- **アーカイバ**（ar2000）を使用すると、ソースやオブジェクトなど複数のファイルをライブラリと呼ばれる 1 つのアーカイブ・ファイルにまとめることができます。さらにアーカイバでは、メンバの削除、置換、抽出、または追加によりライブラリの内容を変更できます。アーカイバは、オブジェクト・モジュールのライブラリ（オブジェクト・ライブラリ）を作成するときに大変便利なツールです。コンパイルされた RTS 関数が含まれるオブジェクト・ライブラリは、C/C++ に付属しています。アーカイバの使用方法については、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

- **ライブラリ作成ユーティリティ**（mk2000 -v28）を使って、独自にカスタマイズしたランタイムサポート・ライブラリを作成できます（第 8 章「ランタイムサポート関数」を参照）。標準ランタイムサポート・ライブラリ関数は、ANSI/ISO C/C++ ライブラリ・ソース用の rts.src の中にソース・コードとして収められています。ランタイムサポート関数用の ANSI/ISO C/C++ ライブラリ・オブジェクト・コードは、rts.lib にまとめられています。

  ランタイムサポート・ライブラリには、C28x コンパイラによってサポートされている ANSI/ISO C 規格のランタイムサポート関数、C++ ライブラリ、コンパイラ・ユーティリティ関数、および浮動小数点算術関数が入っています。詳細については、第 8 章「ランタイムサポート関数」を参照してください。

- Code Composer Studio は、実行可能な COFF ファイルを入力として受け付けますが、大半の EPROM プログラマはそれらのファイルを受け付けません。**Hex 変換ユーティリティ** (hex2000）は、COFF オブジェクト・ファイルを TI-Tagged、ASCII-hex、Intel、Motorola-S、または Tektronix オブジェクト・フォーマットに変換します。変換後のファイルは、EPROM プログラマにダウンロードできます。Hex 変換ユーティリティの使用方法については、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

- 絶対リスタ（abs2000）は、リンクされたオブジェクト・ファイルを使用して、シンボル参照の最終アドレスとコードを提供するアセンブリ・リストを作成します。絶対リスタは、リンクされたオブジェクト・ファイルを入力として使用し、アセンブラが使用できる 中間 .abs ファイルを作成します。絶対リスタの詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

- C++ ネーム・デマングラ（dem2000）は、デバッグ補助機能で、コンパイラによってマングルされた名前を C++ ソース内で宣言された元の名前に変換します。
  図 1-1 に示すように、C++ ネーム・デマングラはコンパイラが出力したアセンブリ・ファイル上で使用することも、アセンブラ・リスト・ファイルおよびリンカ・マップ・ファイル上で使用することもできます。詳細は、第 10 章「C++ネーム・デマングラ・ユーティリティ」を参照してください。

- ポストリンク・オプティマイザ（plink2000）は、不要なアセンブリ言語命令を取り除いたり変更したりして、効率のよいコードを生成します。ポストリンク・オプティマイザは、シェル・オプション -plink と組み合わせて実行する必要があります。詳細は、第 5 章「ポストリンク・オプティマイザ」を参照してください。

この開発プロセスの目的は、C28x ターゲット・システムで実行できるモジュールを生成することです。シミュレータまたはエミュレータを使用すれば、生成したコードに改善や修正を加えることができます。

1.2 C/C++ コンパイラの概要

TMS320C28x C/C++ コンパイラは、標準 ANSI/ISO C/C++ プログラムを C28x アセンブリ言語ソースに変換する豊富な機能を備えた最適化コンパイラです。次に、本コンパイラの主要な機能を紹介します。

1.2.1 C/C++ 言語機能

❏ **ANSI/ISO C 規格**

TMS320C28x コンパイラは ANSI/ISO により規定され、カーニハンとリッチーの『The C Programming Language』（第 2 版）に記述された ANSI/ISO C 規格に準拠しています。

❏ **ANSI/ISO C++ 規格**

TMS320C28x コンパイラは 一部例外がありますが、国際標準化機構 (ANSI/ISO)/IEC 14882-1998 が規定した C++ 規格をサポートします。詳細は、第 6 章「TMS320C28x C/C++ 言語の実装」を参照してください。

❏ **ANSI/ISO 規格ランタイムサポート**

本コンパイラ・ツールには、完全なランタイム・ライブラリが標準装備されています。このライブラリには、標準入出力関数、文字列操作関数、動的メモリ割り当て関数、データ変換関数、時間管理関数、三角関数、指数関数、ハイパボリック関数が収納されています。信号処理関数はターゲット・システムによって異なるため、本ライブラリから除外されています。また、このライブラリは拡張され、far メモリのアクセスをサポートするさまざまな rts 関数が組み込まれています。C++ ライブラリには、言語サポート用の構成要素に加えて ANSI/ISO C のサブセットも含まれています。詳細については、第 8 章「ランタイムサポート関数」を参照してください。

1.2.2 出力ファイル

次の特徴は、コンパイラによって作成される出力ファイルに関するものです。

❑ アセンブリ・ソース出力

本コンパイラにより検査できるアセンブリ言語ソース・ファイルが生成され、C/C++ ソース・ファイルから生成したコードを確認することができます。

❑ **COFF** オブジェクト・ファイル

共通オブジェクト・ファイル・フォーマット（COFF）により、リンク時に使用するシステムのメモリ・マップを定義できます。この定義により C/C++ コードとデータ・オブジェクトを特定のメモリ領域にリンクできるので、最大限のパフォーマンスを発揮できます。COFF ではソースレベルのデバッグ機能もサポートしています。

❑ **EPROM** プログラマ・データ・ファイル

スタンドアロン型の組み込みアプリケーションの場合は、本コンパイラを使用して、すべてのコードと初期設定データを ROM に配置することができます。この結果、C/C++ コードはリセットで実行できるようになります。コンパイラによる COFF ファイル出力は、Hex 変換ユーティリティを使うことにより、EPROM プログラマ・データ・ファイルに変換できます。Hex 変換ユーティリティの詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』（文献番号 SPRUEP3）を参照してください。

1.2.3 コンパイラ・インターフェイス

次の特徴は、コンパイラとのインターフェイスに関するものです。

❑ コンパイラのシェル・プログラム

コンパイラ・パッケージには、1 つのステップで、プログラムをコンパイルし、アセンブルし、リンクすることができるシェル・プログラムが組み込まれています。詳細は、2.1 節「コンパイラについて」（2-2 ページ）を参照してください。

❑ 柔軟なアセンブリ言語インターフェイス

本コンパイラの呼び出し規則は単純化しているので、相互に呼び出すアセンブリ関数や C/C++ 関数を簡単に記述することができます。詳細は、第 7 章「ランタイム環境」を参照してください。

1.2.4　コンパイラの操作

次の特徴は、コンパイラの操作に関するものです。

- **統合プリプロセッサ**

  C/C++ プリプロセッサにはパーサが組み込まれており、コンパイル時間を短縮します。また、前処理を独立して行ったり、前処理リストを生成したりすることもできます。詳細は、2.5 節「プリプロセッサの制御方法」（2-27 ページ）を参照してください。

- **最適化**

  本コンパイラは最適化パスを高度に使用します。この最適化パスは、数々の最新の技法を使用して C/C++ ソースから効率の良いコンパクトなコードを生成します。一般的な最適化は、どのような C/C++ コードにも適用できます。C28x に固有の最適化では、C28x アーキテクチャに固有の特長を最大限に利用しています。C/C++ コンパイラの最適化技法の詳細は、第 3 章「コードの最適化方法」を参照してください。

1.2.5　ユーティリティ

次の特徴は、コンパイラ・ユーティリティに関するものです。

- **ソース・インターリスト・ユーティリティ**

  コンパイラ・パッケージにはユーティリティが組み込まれていて、これを利用してユーザ独自の C/C++ ソース文をコンパイラのアセンブリ言語出力に差し込むことができます。このユーティリティは、各 C/C++ 文に対して生成されたアセンブリ・コードを簡単に検査できる方法を提供します。詳細は、2.10 節「インターリストの使用方法」（2-44 ページ）を参照してください。

- **ライブラリ作成ユーティリティ**

  コンパイラ・パッケージには、アーカイブされたソース・ライブラリからオブジェクト・ライブラリを 1 ステップで作成できるユーティリティ（mk2000 -v28）があります。ニーズに合ったコンパイラ・オプションを使用して、RTS ライブラリを再コンパイルするときに役立ちます。完全な RTS ライブラリ・ソースは、コンパイラ製品に付属しています。

- **C++ ネーム・デマングラ・ユーティリティ**

  C++ ネーム・デマングラ（dem2000）は デバッグ補助機能であり、コンパイラ・ツールが出力したデータにあるマングルされた名前を C++ ソース内で宣言された元の名前に変換します。詳細は、第 10 章「C++ネーム・デマングラ・ユーティリティ」を参照してください。

第 2 章

C/C++ コンパイラの使用方法

コンパイラは、ソース・プログラムを TMS320C28x™ が実行できるコードに変換します。実行可能なオブジェクト・ファイルを作成するには、ソース・ファイルをコンパイル、アセンブル、リンクする必要があります。これらすべてのステップは、コンパイラを使って一度に実行されます。本章では、プログラムのコンパイル方法、アセンブル方法、リンク方法について詳細に説明します。

本章ではまた、プリプロセッサ、オプティマイザ、インライン関数展開機能、およびインターリスト・ユーティリティについても説明します。

項目 **ページ**

2.1	コンパイラについて	2-2
2.2	C/C++ コンパイラの起動方法	2-4
2.3	オプションによるコンパイラの動作の変更	2-5
2.4	環境変数によるコンパイラの動作の変更	2-24
2.5	プリプロセッサの制御方法	2-27
2.6	診断メッセージの概要	2-32
2.7	クロスリファレンス・リスト情報の生成方法（-px オプション）	2-37
2.8	ロー・リスト・ファイルの生成方法（-pl オプション）	2-38
2.9	インライン関数展開の使用方法	2-40
2.10	インターリストの使用方法	2-44

2.1 コンパイラについて

コンパイラ (cl2000 -v28) を使用すると、コンパイルとアセンブル、さらに必要に応じてリンクを 1 つのステップで実行できます。コンパイラは、次の処理を 1 つまたは複数のソース・モジュールに対して実行します。

- ❑ **コード・ジェネレータ**はパーサおよびオプティマイザを含んでいて、C/C++ のソース・コードを取り込んで、C28x アセンブリ言語ソース・コードを生成します。

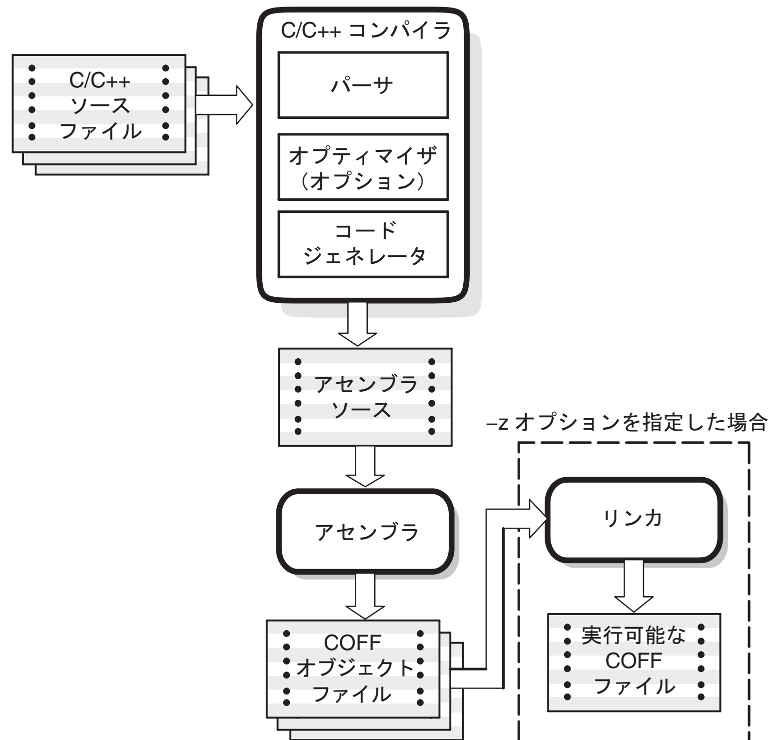
  C ファイルと C++ ファイルを 1 つのコマンドでコンパイルできます。コンパイラは、ファイル名拡張子を使用して両者を区別します（詳細は、2.3.4 項「ファイル名の指定方法」を参照）。

- ❑ **アセンブラ**は COFF オブジェクト・ファイルを生成します。

- ❑ **リンカ**はオブジェクト・ファイルを組み合わせ、1 つの実行可能なオブジェクト・モジュールを作成します。リンク手順は任意のため、複数のモジュールをコンパイルおよびアセンブルし、後からリンクすることもできます。ファイルのリンク方法については、第 4 章「C/C++ コードのリンク方法」を参照してください。

デフォルトでは、コンパイラはリンク手順を実行しません。-z コンパイラ・オプションを指定することにより、リンカを起動できます。図 2-1 に、コンパイラの経路（リンカを使用する場合と、使用しない場合）を示します。

図 2-1.　C/C++ コンパイラのフロー図



アセンブラとリンカの詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

2.2 C/C++ コンパイラの起動方法

コンパイラを起動するには、次のように入力します。

cl2000 -v28 [-*options*] *filenames* [*object files*] [**-z** [*link_options*]]

cl2000 -v28	コンパイラとアセンブラを実行させるコマンドです。
options	コンパイラによる入力ファイルの処理方法に影響を与えるオプションです（オプションは表 2-1（2-6 ページ）を参照）。
filenames	1 つまたは複数の C/C++ ソース・ファイルまたはアセンブリ・ソース・ファイルを指定します。
object files	リンク・プロセスの追加オブジェクト・ファイル名です。
-z	リンカの起動オプションです。リンカの起動方法については、第 4 章「C/C++ コードのリンク方法」を参照してください。
link_options	リンク・プロセスの制御オプションです。

cl2000 -v28 への引数は、コンパイラ・オプション、リンカ・オプション、およびファイルの 3 つのタイプのいずれかです。-z リンカ・オプションは、リンクが実行されることを表します。-z リンカ・オプションを指定する場合、コンパイラ・オプションは -z リンカ・オプションの前に、他のリンカ・オプションは -z リンカ・オプションの後にそれぞれ指定する必要があります。ソース・コード・ファイル名は、-z リンカ・オプションの前に配置する必要があります。それ以外の場合、オプションとファイル名は任意の順序で配置することができます。

たとえば、symtab.c と file.c という名前の 2 つのファイルをコンパイルし、seek.asm という名前の第 3 のファイルをアセンブルし、リンクして実行可能なファイルを作成する場合は、次のように入力します。

cl2000 -v28 symtab.c file.c seek.asm -z -llnk.cmd -lrts2800.lib

このコマンドを入力すると、次のように出力されます。

```
[symtab.c]
[file.c]
[seek.asm]
<Linking>
```

2.3　オプションによるコンパイラの動作の変更

オプションは、コンパイラとコンパイラによって実行されるプログラムの両方の動作を制御します。ここではオプションの規則について説明するとともに、各オプションについてまとめた表を示します。データ型のチェックやアセンブル用に指定するオプションなど、使用頻度の高いオプションについて詳細に説明しています。

コンパイラ・オプションには次の規則が適用されます。

- ❏ オプションの先頭には 1 つまたは 2 つのハイフンが付きます。
- ❏ オプションは大文字小文字を区別します。
- ❏ オプションは 1 つの文字か一連の複数の文字で構成されます。
- ❏ 個々のオプションを結合させることはできません。
- ❏ *必須*パラメータ付きのオプションは、パラメータの前にイコール符号を付けて指定し、パラメータとそのオプションを明確に関連付ける必要があります。たとえば名前を未定義にするオプションを指定するには、-U=*name* と入力しても同じです。推奨はしませんが、-U *name* または -U*name* のように、スペースを入れても入れなくてもオプションとパラメータを分けることはできます。
- ❏ *任意の*パラメータ付きのオプションは、パラメータの前にイコール符号を付けて指定し、パラメータとそのオプションを明確に関連付ける必要があります。たとえば、最大の最適化を指定するオプションは -O=3 と指定できます。推奨はしませんが、-O3 のように、オプションの後に直接パラメータを指定することもできます。オプションと任意のパラメータの間にはスペースを入れられないため、-O 3 は受け入れられません。
- ❏ ファイルとオプションは、-z オプションを除いて任意の順序で指定できます。-z オプションは他のすべてのコンパイラ・オプションの後、かつリンカ・オプションの前に指定する必要があります。

コンパイラに対してデフォルトのオプションを定義するには、C_OPTION または C2000_C_OPTION 環境変数を使用します。詳細は、2.4.2 項「デフォルトのコンパイラ・オプションの設定方法（C_OPTION および C2000_C_OPTION）」（2-24 ページ）を参照してください。

表 2-1 は、全オプション（リンカ・オプションを含む）をまとめたものです。表内には各オプションの詳細を説明しているページ番号が記載されています。必要に応じて参照してください。

表 2-1. コンパイラ・オプションのまとめ

(a) コンパイラ / シェルを制御するオプション

オプション	機能	ページ
-@*filename*	コマンド行の延長として、ファイルの内容を解釈します。	2-13
-abs	絶対リスト・ファイルを作成します。このオプションは、-z の後に指定する必要があります。	2-13
-b	ユーザ情報ファイルを生成します。	2-13
-c	リンクを使用不可にします（-z の無効化)。	2-13、4-4
-D*name*[=*def*]	*name* を事前定義します（D は大文字)。	2-13
-I*directory*	#include 検索パスを定義します（I は大文字)。	2-13、2-29
-k	アセンブリ言語（.asm）ファイルを保存します。	2-13
-n	コンパイルまたはアセンブリの最適化のみを行います。	2-14
-plink	ポストリンク最適化を実行します。このオプションは、-z の後に指定する必要があります。	5-1
-q	複数の進捗メッセージの出力を抑止します（静的実行)。	2-14
-s	オプティマイザのコメントをアセンブリ・ソース文に差し込みます。オプティマイザのコメントがない場合は、C/C++ のソースをアセンブリ・ソース文に差し込みます。	2-14
-ss	C/C++ のソースをアセンブリ・ソース文に差し込みます。	2-14、2-44
-U*name*	*name* を未定義にします（U は大文字)。	2-14
-z	リンクの実行を有効にします。	2-14

表 2-1.　コンパイラ・オプションのまとめ（続き）

(b)　シンボリック・デバッグおよびプロファイルを制御するオプション

オプション	機能	ページ
-g	シンボリック・デバッグを有効にします（--symdebug:dwarf と同等）。	2-15
--profile:breakpt	ブレーク・ポイントに基づいたプロファイルの実行を有効にします。	2-15
--profile:power	パワー・プロファイルの実行を有効にします。	2-15
--symdebug:coff	代替 STABS デバッグ・フォーマットを使ったシンボリック・デバッグを有効にします。	2-15、3-16
--symdebug:dwarf	DWARF デバッグ・フォーマットを使ったシンボリック・デバッグを有効にします（-g と同等）。	2-15
--symdebug:none	すべてのシンボリック・デバッグを無効にします。	2-15
--symdebug:skeletal	最適化を妨げない最小限のシンボリック・デバッグを有効にします（デフォルトの動作）。	2-15

(c)　デフォルトのファイル拡張子を変更するオプション

オプション	機能	ページ
-ea[.]*extension*	アセンブリ・ソース・ファイルのデフォルトの拡張子を設定します。	2-20
-eo[.]*extension*	オブジェクト・ファイルのデフォルトの拡張子を設定します。	2-20

(d)　ファイルを指定するオプション

オプション	機能	ページ
-fa*filename*	拡張子に関係なく、*filename* をアセンブリ・ソース・ファイルとして識別します。デフォルトでは、コンパイラは .asm または .s*（s で始まる拡張子）ファイルをアセンブリ・ソース・ファイルとして扱います。	2-19
-fc*filename*	拡張子に関係なく、*filename* を C ファイルとして識別します。デフォルトでは、コンパイラは .c または拡張子のないファイルを C ファイルとして扱います。	2-19
-fg	C 拡張子の付いたファイルを C++ ファイルとしてコンパイルします。	2-19
-fo*filename*	拡張子に関係なく、*filename* をオブジェクト・ファイルとして識別します。デフォルトでは、コンパイラは .o* をオブジェクト・ファイルとして扱います。	2-19
-fp*filename*	拡張子に関係なく、*filename* を C++ ファイルとして識別します。デフォルトでは、コンパイラは .C、.cpp、.cc、.cxx のいずれかのファイルを C++ ファイルとして扱います。†	2-19

表 2-1.　コンパイラ・オプションのまとめ（続き）

(e)　ディレクトリを指定するオプション

オプション	機能	ページ
-fb*directory*	絶対リスト・ファイルのディレクトリを指定します。	2-21
-ff*directory*	アセンブリ・リスト・ファイルのディレクトリを指定します。	2-21
-fr*directory*	オブジェクト・ファイルのディレクトリを指定します。	2-21
-fs*directory*	アセンブリ・ファイルのディレクトリを指定します。	2-21
-ft*directory*	一時ファイルのディレクトリを指定します。	2-21

(f)　マシン固有のオプション

オプション	機能	ページ
-ma	エイリアスが設定された変数と見なします。	2-16、3-11
-md	DP ロード命令の最適化を無効にします。	2-16
-me	高速分岐命令の生成を無効にします。	2-16
-mf	サイズよりコード・スピードを優先した最適化を行います。	2-16
-mi	RPT 命令の生成を無効にします。	2-16
-ml	ラージ・メモリ・モデルのコードを生成します。	2-16、6-21
-mn	-g が無効にした最適化を有効にします。	2-17、3-16
-ms	スピードよりサイズを優先した最適化を行います。	2-17、3-18
-mt	単一メモリ・モデルのコードを生成します。	2-17
-mu	C2XLP OUT 命令を C28x UOUT 命令としてエンコードします。	2-17
-mv	volatile 参照保護を有効にします。	2-17
-mx	アセンブリ・マクロを展開します。	2-18
-v28	TMS320C28x アーキテクチャを指定します。	2-18

表 2-1.　コンパイラ・オプションのまとめ（続き）

(g)　パーサを制御するオプション

オプション	機能	ページ
-pe	組み込み C++ モードを有効にします。	6-6
-pi	定義優先の制御によるインライン展開を抑止します（ただし、-O3 を指定した最適化は自動インライン展開を実行し続けます）。	2-41
-pk	K&R 互換性を許容します。	6-35
-pl *file*	ロー・リスト情報を *file* に出力します。	2-38
-pm	プログラム・モードでコンパイルします。	3-6
-pn	組み込み関数を無効にします。	--
-pr	緩和モードを有効にします。厳密な ANSI/ISO 違反を無視します。	6-38
-ps	厳密な ISO モード（C/C++ に対してであり、K&R C に対してではない）を有効にします。	6-38
-px *file*	クロスリファレンス情報を *file* に出力します。	2-37
-rtti	ランタイム型情報（RTTI）を有効にします。	6-5

(h)　前処理を制御するオプション

オプション	機能	ページ
-ppa	前処理に続けてコンパイルを実行します。	2-30
-ppc	前処理だけを実行します。入力と名前が同じで拡張子が .pp のファイルに、前処理された出力を（コメントを保持したまま）書き込みます。	2-30
-ppd	前処理だけを実行します。ただし前処理された出力を書き込むのではなく、標準 make ユーティリティへの入力に適した依存行のリストを書き込みます。	2-30
-ppi	前処理だけを実行します。ただし前処理された出力を書き込むのではなく、#include 疑似命令で組み込まれるファイルのリストを書き込みます。	2-30
-ppl	前処理だけを実行します。入力と名前が同じで拡張子が .pp のファイルに、行の制御情報（#line 疑似命令）と一緒に前処理された出力を書き込みます。	2-30
-ppo	前処理だけを実行します。入力と名前が同じで拡張子が .pp のファイルに、前処理された出力を書き込みます。	2-30

表 2-1.　コンパイラ・オプションのまとめ（続き）

(i)　診断を制御するオプション

オプション	機能	ページ
-pdel *num*	エラー数の上限を *num* に設定します。エラー数がこの指定値に達すると、コンパイラはコンパイルを中止します（デフォルトは 100 です）。	2-34
-pden	診断の識別子を、そのテキストと一緒に表示します。	2-34
-pdf *outfile*	診断情報を標準エラーではなく *outfile* に書き込みます。	2-34
-pdr	注釈（軽い警告）を発行します。	2-34
-pds *num*	*num* の識別診断を抑止します。	2-34
-pdse *num*	*num* の識別診断をエラーに分類します。	2-34
-pdsr *num*	*num* の識別診断を注釈に分類します。	2-34
-pdsw *num*	*num* の識別診断を警告に分類します。	2-34
-pdv	行の折り返し付きでオリジナル・ソースを表示する詳細な診断情報を提供します。	2-34
-pdw	警告診断を抑止します（エラーは発行されます）。	2-34

(j)　最適化を制御するオプション

オプション	機能	ページ
-O0	レジスタを最適化します。	3-2
-O1	-O0 による最適化を行い、さらにローカルな最適化を行います。	3-2
-O2 または -Oo	-O1 による最適化を行い、さらにグローバルな最適化を行います。	3-2
-O3	-O2 による最適化を行い、さらにファイルに対して最適化を行います。	3-2
-oi*size*	自動インライン展開サイズを設定します（-O3 のみ）。	3-12
-ol0 または -oL0	ファイルが標準ライブラリ関数を変更することを、オプティマイザに指示します。	3-4
-ol1 または -oL1	ファイルが標準ライブラリ関数を宣言することを、オプティマイザに指示します。	3-4
-ol2 または -oL2	ファイルがライブラリ関数の宣言も変更も行わないことを、オプティマイザに指示します。-ol0 および -ol1 オプションを取り消します。	3-4
-on0	オプティマイザ情報ファイルの生成を抑止します。	3-5
-on1	オプティマイザ情報ファイルを作成します。	3-5
-on2	詳細なオプティマイザ情報ファイルを作成します。	3-5

表 2-1.　コンパイラ・オプションのまとめ（続き）

(k)　最適化を制御するオプション（続き）

オプション	機能	ページ
-op0	コンパイラに入力されるソース・コード外から呼び出されたり変更されたりする関数と変数をモジュールが含むことを指定します。	3-6
-op1	モジュールはコンパイラに入力されるソース・コード外から変更される変数は含むが、ソース・コード外から呼び出される関数は使用しないことを指定します。	3-6
-op2	モジュールは、コンパイラに入力されるソース・コード外から呼び出されたり変更されたりする関数や変数をどちらも含まないことを指定します（デフォルト）。	3-6
-op3	モジュールはコンパイラに入力されるソース・コード外から呼び出される関数を含むが、ソース・コード外から変更される変数は使用しないことを指定します。	3-6
-os	オプティマイザの注釈をアセンブリ・ソース文に差し込みます。	3-13

(l)　アセンブラを制御するオプション

オプション	機能	ページ
-aa	絶対リスト・ファイルを生成します。	2-22
-ac	アセンブリ・ソース・ファイルで大文字と小文字を区別しません。	2-22
-ad *name*	シンボル *name* を定義します。	2-22
-ahc *filename*	指定した *filename* をコピーします。	2-22
-ahi *filename*	指定した *filename* を組み込みます。	2-22
-al	アセンブリ・リスト・ファイルを生成します。	2-22
-apd	前処理を行います。アセンブリ依存性のみをリスト表示します。	2-22
-api	前処理を行います。組み込まれた #include ファイルのみをリスト表示します。	2-22
-as	シンボル・テーブルにラベルを格納します。	2-22
-au *name*	事前定義された定数 *name* を未定義にします。	2-23
-ax	クロスリファレンス・ファイルを生成します。	2-23

表 2-1.　コンパイラ・オプションのまとめ（続き）

(m) リンカを制御するオプション

オプション	機能	ページ
-a	絶対出力を生成します。	4-5
-abs	絶対リスト・ファイルを作成します。	2-13
-ar	再配置可能な出力を生成します。	4-5
-b	シンボリック・デバッグ情報のマージを無効にします。	4-5
-c	実行時に変数を自動初期化します。	4-5
-cr	リセット時に変数を自動初期化します。	4-5
-e *global_symbol*	エントリ・ポイントを定義します。	4-5
-f *fill_value*	埋め込み値を定義します。	4-5
-g *global_symbol*	*global_symbol* をグローバルにしておきます（-h の無効化）。	4-5
-h	グローバル・シンボルを静的にします。	4-5
-heap *size*	ヒープ・サイズ（バイト数）を設定します。	4-5
-I *directory*	ライブラリ検索パスを定義します。	4-5
-l *filename*	ライブラリ名を提供します。	4-5
-m *filename*	マップ・ファイル名を指定します。	4-6
-n	メモリ疑似命令のすべての埋め込み指定を無視します。	4-6
-o *filename*	出力ファイル名を指定します。	4-6
-priority	ライブラリの代替検索メカニズムを提供します。	4-6
-r	再配置可能な出力を生成します。	4-6
-s	シンボル・テーブルを除去します。	4-6
-stack *size*	1 次スタック・サイズ（バイト数）を設定します。	4-6
-u *symbol*	シンボルを未定義にします。	4-6
-w	未定義の出力セクションが作成された場合にメッセージを表示します。	4-6
-x	ライブラリの再読み取りを強制します。	4-6

2.3.1 使用頻度の高いオプション

以下は、使用頻度の高いオプションについての詳細な説明です。

-@*filename* コマンド行エントリではなく、指定したファイルの内容を使用します。このコマンドは、ホスト・オペレーティング・システムによるコマンド行の長さに対する制限事項の回避策として使用できます。コメントを記述する場合は、コマンド・ファイルに # か ; を入力します。

コマンド行内では、スペースやハイフンが埋め込まれたファイル名およびオプション・パラメータは、引用符で囲む必要があります（例："this-file.obj"）。

-abs 絶対リスト・ファイルを生成します。このオプションは、-z オプションの後に指定してください。このオプションを指定すると、コンパイラは絶対リスタを起動します（絶対リスタについては、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照）。

-b スタック・サイズおよび関数呼び出しに関する情報を参照できる、補足情報ファイルを生成します。ファイル名は、C/C++ ソース・ファイル名に拡張子 .aux を付けます。

-c リンカを抑止し、リンクの実行を指定した -z オプションを無効にします。このオプションは、C_OPTION 環境変数で -z を指定しているがリンクしたくないといった場合に便利です。詳細は、4.1.3 項「リンカを無効にする方法（-cコンパイラ・オプション）」（4-4 ページ）を参照してください。

-D*name[=def]* プリプロセッサの定数 *name* を事前定義します。これは、それぞれの C/C++ ソース・ファイルの最上部に #define *name def* を挿入するのと同等です。オプションの [=*def*] を省略すると、*name* は 1 に設定されます。

-I*directory* コンパイラが #include ファイルを検索するディレクトリのリストに *directory* を追加します。コンパイラを呼び出すたびに、-I オプションをいくつでも使用することができます。それぞれの -I オプションとの間は必ず空白で区切ってください。ディレクトリ名を指定しないと、プリプロセッサは -I オプションを無視します。詳細は、2.5.2.1 項「#include ファイル検索パスの変更（-I オプション）」（2-29 ページ）を参照してください。

-k コンパイラのアセンブリ言語の出力を保存します。通常は、アセンブリが終了すると、コンパイラは出力されたアセンブリ言語ファイルを削除します。

-n コンパイルのみを行います。指定したソース・ファイルのコンパイルは実行されますが、アセンブルとリンクはどちらも行われません。このオプションは -z オプションを無効にします。出力は、コンパイラのアセンブリ言語の出力です。

-pn コンパイラが提供する組み込み関数を無効にします。-pn オプションを指定してコンパイルする場合、これらの組み込み関数の 1 つに呼び出しが行われると、呼び出しは通常の関数呼び出しとして扱われます。

-q 見出しおよびすべてのツールからの進捗情報を抑止します。ソース・ファイル名およびエラー・メッセージのみが出力されます。

-s インターリスト機能を起動します。この機能を使うと、オプティマイザのコメントまたは C/C++ のソースをアセンブリ・ソースに差し込むことができます。オプティマイザを起動している場合は（-O*n* オプションを指定）、オプティマイザのコメントがコンパイラのアセンブリ言語出力に差し込まれます。オプティマイザは、コードを大幅に再配置する可能性があります。オプティマイザを起動していない場合は C/C++ ソース文がコンパイラのアセンブリ言語出力に差し込まれ、これにより各 C/C++ の文に対して生成されたコードを検査できます。-s オプションを指定すると、-k オプションも暗黙指定されます。-s オプションを指定すると、-ss オプションを使っていても無効になります。オプティマイザとインターリスト機能を組み合わせて使用する方法の詳細は、3.6 節「インターリスト・ユーティリティをオプティマイザと組み合わせて使用する方法」（3-13 ページ）を参照してください。

-ss インターリスト・ユーティリティを起動します。このユーティリティは、オリジナルの C/C++ ソースを、コンパイラが生成したアセンブリ言語に差し込みます。このオプションを使うと、コードが大幅に再編成される場合があります。詳細は、2.10 節「インターリストの使用方法」（2-44 ページ）を参照してください。-s オプションを指定すると、-ss オプションを指定しても無効になります。

-U*name* 事前定義された定数 *name* を未定義にします。指定した定数に対するすべての -D オプションを無効にします。

-z 指定したオブジェクト・ファイルに対して、リンカを実行します。-z オプションおよび関連付けられたリンカ・コマンド・オプションは、コマンド行で他のすべてのオプションの後に指定します。-z の後に指定した引数は、すべてリンカに渡されます。詳細は、第 4 章「C/C++ コードのリンク方法」を参照してください。

2.3.2 シンボリック・デバッグおよびプロファイルのオプション

-g または
--symdebug:dwarf

C/C++ ソースレベル・デバッガが使用する疑似命令を生成します。これにより、アセンブラでのアセンブリ・ソース・デバッグが可能になります。ただしコード・ジェネレータによる多くの最適化はデバッガを混乱させるため、-g オプションにより抑止されます。-g オプションを -O オプションと一緒に指定すると、デバッグと互換性のある最適化を最大にできます（3.7 節「最適化されたコードのデバッグ方法」（3-16 ページ）を参照）。

DWARF デバッグ・フォーマットの詳細については、『DWARF デバッグ情報フォーマットの仕様』（1992-1993, UNIX International, Inc）を参照してください。

--profile:breakpt

ブレーク・ポイントに基づいたプロファイラを使う場合に、誤った動作を引き起こす原因となる最適化を無効にします。

--profile:power

パワー・プロファイラの命令コードを生成するパワー・プロファイルを有効にします。

--symdebug:coff

代替 STABS デバッグ・フォーマットを使ったシンボリック・デバッグを有効にします。このオプションを指定するのは、古いデバッガやカスタム・ツールを使ったデバッグをできるようにすることが必要な場合があるからです。このようなツールは DWARF フォーマットは読み込みません。

--symdebug:none

すべてのシンボリック・デバッグ出力を無効にします。このオプションは、デバッグおよび大半のパフォーマンス解析機能を妨げるため、推奨しません。

--symdebug:skeletal

最適化を妨げずに、最大限のシンボリック・デバッグ情報を生成します。一般に、このオプションはグローバル・スコープの情報のみで構成されます。このオプションは、コンパイラのデフォルトの動作を示します。

シンボリック・デバッグの非推奨オプションについては、2.3.9 項（2-23 ページ）を参照してください。

2.3.3 マシン固有のオプション

以下は、使用頻度の高いマシン固有のオプションについての詳細な説明です。

-ma

変数にエイリアスが設定されていることを想定します。コンパイラは、ポインタが指定した変数にエイリアスが設定されている場合があると見なします。したがって、コンパイラが同じオブジェクトを指し示す別のポインタが存在する可能性があると決めた場合、ポインタから代入が行われるとき、レジスタの最適化は無効になります。

-md

DP 直接アドレッシングを使用している場合、コンパイラが DP レジスタの冗長なロード命令を最適化できないようにします。

-me

コンパイラが TMS320C28x の高速分岐命令（BF）を生成できないようにします。高速分岐命令は、可能な場合にはコンパイラがデフォルトで生成します。

-mf

サイズよりスピードを優先したコードの最適化を行います。デフォルトでは、C28x のオプティマイザは、スピードを犠牲にしてコード・サイズを小さくしようとします。

-mf（スピードを優先した最適化）オプションを指定した場合、デフォルトでは高速分岐（BF）命令が生成されます。-mf オプションを指定しない場合、条件コードが NEQ、EQ、NTC、および TC のいずれかであるときにのみコンパイラは BF 命令を生成します。これは、これらの条件コードを使った BF が SBF に最適化されることがあるからです。条件コードが NEQ、EQ、NTC、および TC のいずれでもない場合、BF 命令を使う上でコードサイズのペナルティがあります。

-me オプションは、-mf オプションの下で生成された BF 命令を制御しません。つまり、-mf によって -me オプションは無効になります。条件コードが NEQ、EQ、NTC、および TC のいずれかの場合に、- me オプションはデフォルトで生成された BF 命令のみに影響を与えます。

-mi

コンパイラがリピート（RPT）命令を生成できないようにします。デフォルトでは、特定の memcpy 命令および特定の除算命令の場合に、リピート命令が生成されます。ただし、リピート命令は割り込み不可です。

-ml

ラージ・メモリ・モデルのコードを生成します。これにより、コンパイラはフラットな 22 ビット・アドレス空間を備えたものとしてアーキテクチャを見ることができます。すべてのポインタは、22 ビットのポインタと見なされます。-ml の主な使い方は、C++ コードを組み合わせて 16 ビットを超えるメモリをアクセスすることです。詳細は、6.7.7 項「ラージ・メモリ・モデルを使用する方法（-ml オプション）」（6-21 ページ）を参照してください。

-mn

-g オプションによって無効となった最適化を有効にします。-g オプションを使用した場合、コード・ジェネレータによる多くの最適化はデバッガを混乱させるため抑止されます。したがって、-mn オプションを使うと、デバッガの機能性が低くなります。

-ms

コンパイラが行うコード・サイズ優先の最適化のレベルを強化します。詳細は、3.8 節「コード・サイズ最適化の強化（-ms オプション)」(3-18 ページ）を参照してください。

-mt

メモリ・マップが 1 つの単一空間として設定されている場合に、-mt オプションを指定します。これにより、コンパイラはほとんどの memcpy 呼び出しや構造体への代入を行うために RPT PREAD 命令を生成することができます。また、これにより MAC 命令も生成することができます。また、-mt オプションを指定すると、より効率的なデータ・メモリ命令を使用して、スイッチ・テーブルがアクセスされるようになります。

-mu

C2xlp OUT 命令を C28x UOUT 命令としてエンコードします。C28x プロセッサには、保護された（OUT）命令と保護されていない（UOUT）命令があります。デフォルトでは、アセンブラは C2xlp OUT 命令を C28x 保護 OUT 命令としてエンコードします。-mu オプションは、-m20 オプションを指定しない場合には無効です。

-mv *num*

volatile 参照保護を有効にします。パイプライン・コンフリクトは、volatile と宣言されている非ローカル変数間で発生する場合があります。ある volatile 変数に書き込みして続けて別の volatile 変数からデータを読み出している間にコンフリクトが発生する場合があります。-mv オプションを使用すると、読み出しを行う前に確実に書き込みが行われるようにするために、少なくとも *num* 個の命令を 2 つの volatile 参照間に配置できます。num は任意です。num を指定しない場合、そのデフォルト値は 2 です。たとえば、-mv4 と指定すると、volatile 書き込みと volatile 読み込みは、少なくとも 4 つの命令で保護されます。

ペリフェラル・パイプライン保護ハードウェアは、すべての内部ペリフェラルと XINTF ゾーン 1 を保護します。ペリフェラルを XINTF ゾーン 0/2/6/7 に接続する場合には、-mv オプションを指定する必要があります。ハードウェア保護または -mv オプションの指定は、メモリの場合には必須ではありません。

-mx アセンブリ・ファイル内のマクロを展開して、展開されたファイルをアセンブルします。マクロを展開することによって、アセンブリ・ファイルをデバッグすることができます。-mx オプションは、アセンブリ・ファイルにのみ影響を与えます。-mx 指定時に、コンパイラは最初に -ml を付けてアセンブラを起動し、マクロを展開したソース .exp ファイルを生成します。その後、.exp ファイルがアセンブルされて、オブジェクト・ファイルが生成されます。デバッガは .exp ファイルを使用してデバッグします。.exp ファイルは中間ファイルで、このファイルに変更を加えても失われてしまいます。変更を有効にするには、元のアセンブリ・ファイルに変更を加える必要があります。

-v28 TMS320C28x アーキテクチャ固有のコードを生成します。

2.3.4 ファイル名の指定方法

コマンド行では、入力ファイルとして、C/C++ ソース・ファイル、アセンブリ・ソース・ファイル、またはオブジェクト・ファイルを指定できます。コンパイラは、ファイル名の拡張子によりファイルのタイプを判別します。

拡張子	ファイル・タイプ
.asm、.abs、または .s*（s で始まる拡張子）	アセンブリ・ソース
.obj	オブジェクト
.C、.cpp、.cxx、または .cc†	C++ ソース
.c	C ソース

† ファイル名の拡張子における大文字と小文字の区別は、OS によって決まります。OS が大文字小文字を区別しない場合、.C は C ファイルとして解釈されます。

すべのソース・ファイルには拡張子が必要です。ファイル名拡張子の規則により、C および C++ ファイルのコンパイル、およびアセンブリ・ファイルのアセンブルを 1 つのコマンドで行うことが可能になります。

コンパイラが個々のファイル名をどのように解釈するかを変更する方法については、2.3.5 項（2-19 ページ）を参照してください。アセンブリ・ソースとオブジェクト・ファイルの拡張子に対するコンパイラの解釈の仕方、またファイル作成時の拡張子の付け方を変更する方法については、2.3.6 項（2-20 ページ）を参照してください。

複数のファイルをコンパイルまたはアセンブルする場合には、ワイルドカード文字を使用することができます。ワイルドカードの使い方はシステムによって異なります。オペレーティング ・システムのマニュアルに指定されている適切な形式を使用してください。たとえば、あるディレクトリに入っているすべての C++ ファイルをコンパイルするには、次のように入力します。

```
cl2000 -v28 *.cpp
```

2.3.5 コンパイラによるファイル名の解釈方法の変更（-fa、-fc、-fg、-fo、および -fp オプション）

コンパイラがファイル名を解釈する方法は、オプションによって変更できます。コンパイラが認識できない拡張子を使用している場合は、-fa、-fc、-fo、および -fp オプションを指定することによりファイルのタイプを指定できます。オプションとファイル名の間に入れるスペースの数は任意です。指定するファイルのタイプに応じて、次のいずれか適切なものを使用してください。

-fa*filename*	アセンブリ言語ソース・ファイルの場合
-fc*filename*	C ソース・ファイルの場合
-fo*filename*	オブジェクト・ファイルの場合
-fp*filename*	C++ ソース・ファイルの場合

たとえば、file.s という C ソース・ファイルと assy というアセンブリ言語ソース・ファイルがある場合、次のように -fa と -fc オプションを指定することにより、ファイル・タイプを正しく解釈させることができます。

```
cl2000 -v28 -fc file.s -fa assy
```

ワイルドカード指定と -fa、-fc、-fo、および -fp オプションを組み合わせて使うことはできません。

-fg オプションを指定すると、コンパイラは C ファイルを C++ ファイルとして処理します。デフォルトでは、コンパイラは .c 拡張子の付いたファイルを C ファイルとして扱います。ファイル名拡張子の規則についての詳細は、2.3.4 項（2-18 ページ）を参照してください。

2.3.6 コンパイラによるファイル名の拡張子の解釈方法とファイル作成時の拡張子の付け方の変更（-ea、-ec、-eo、および -ep オプション）

コンパイラによるファイル名の拡張子の解釈方法、およびファイル作成時の拡張子の付け方をオプションにより変更できます。コマンド行では、これらのオプションは適応するファイル名の前に付ける必要があります。どのオプションについても、ワイルドカードによる指定が可能です。

指定する拡張子のタイプに応じて、次のいずれか適切なオプションを使用してください。

-ea[**.**] *new extension*	アセンブリ言語ファイルの場合
-ec[**.**] *new extension*	C ソース・ファイルの場合
-eo[**.**] *new extension*	オブジェクト・ファイルの場合
-ep[**.**] *new extension*	C++ ソース・ファイルの場合

拡張子の長さは最大で 9 文字まで有効です。

次の例ではファイル fit.rrr がアセンブルされ、fit.o という名前のオブジェクト・ファイルが作成されます。

```
cl2000 -v28 -ea .rrr -eo .o fit.rrr
```

拡張子のピリオド（.）およびオプションと拡張子間の空白は、任意です。前述の例は次のように記述することもできます。

```
cl2000 -v28 -earrr -eoo fit.rrr
```

2.3.7 ディレクトリの指定方法

デフォルトでは、コンパイラは、作成したオブジェクト・ファイル、アセンブリ・ファイル、一時ファイルをいずれもカレント・ディレクトリに格納します。これらのファイルを別のディレクトリに格納する場合は、次のオプションを指定します。

-fb 絶対リスト・ファイルを格納する宛先ディレクトリを指定します。デフォルトでは、オブジェクト・ファイルのディレクトリと同じディレクトリが使用されます。絶対リスト・ファイルのディレクトリを指定するには、次のように、コマンド行で -fb オプションに続けて該当ディレクトリのパス名を入力します。

```
cl2000 -v28 -fb d:\abso_list
```

-ff アセンブリ・リスト・ファイルとクロスリファレンス・リスト・ファイルの宛先ディレクトリを指定します。デフォルトでは、オブジェクト・ファイルのディレクトリと同じディレクトリが使用されます。アセンブリ / クロスリファレンス・リスト・ファイルのディレクトリを指定するには、次のように、コマンド行で -ff オプションに続けて該当ディレクトリのパス名を入力します。

```
cl2000 -v28 -ff d:\listing
```

-fr オブジェクト・ファイルのディレクトリを指定します。オブジェクト・ファイルのディレクトリを指定するには、次のように、コマンド行で -fr オプションに続けて該当ディレクトリのパス名を入力します。

```
cl2000 -v28 -fr d:\object
```

-fs アセンブリ・ファイルのディレクトリを指定します。アセンブリ・ファイルのディレクトリを指定するには、次のように、コマンド行で -fs オプションに続けて該当ディレクトリのパス名を入力します。

```
cl2000 -v28 -fs d:\assembly
```

-ft 一時中間ファイルのディレクトリを指定します。-ft オプションを指定すると、TMP 環境変数を無効にします。一時ファイルのディレクトリを指定するには、次のように、コマンド行で -ft オプションに続けて該当ディレクトリのパス名を入力します。

```
cl2000 -v28 -ft c:\temp
```

2.3.8 アセンブラを制御するオプション

以下は、コンパイラと組み合わせて使用できるアセンブラ・オプションです。

-aa -a アセンブラ・オプションを指定してアセンブラを起動します。-a オプションにより絶対リストが作成されます。絶対リストは、オブジェクト・コードの絶対アドレスを示します。

-ac アセンブリ・ソース・ファイルでは大文字と小文字を区別しません。たとえば、-ac はシンボル ABC と abc を同じものにします。このオプションを指定しない場合、大文字と小文字は区別されます（これがデフォルトです）。

-ad*name*[*value*] *name* シンボルを設定します。これは、アセンブリ・ファイルの最初に *name* .set [*value*] を挿入するのと同等です。*value* を省略すると、シンボルは 1 に設定されます。

-ahc *filename* 入力ファイルから任意のソース文がアセンブルされる前に filename をコピーします。アセンブラは、-ahc オプション・ファイルを .copy ファイルとほとんど同じように扱います。コピーされたファイルからアセンブルされたファイルは、アセンブリ・リスト・ファイルには記録されません。

-ahi *filename* 入力ファイルから任意のソース文がアセンブルされる前に filename をインクルードします。アセンブラは、-ahi オプション・ファイルを .include ファイルとほとんど同じように扱います。インクルード・ファイルからアセンブルされたファイルは、アセンブリ・リスト・ファイルには記録されません。

-al -l（小文字の L）アセンブラ・オプションを指定してアセンブラを起動し、アセンブリ・リスト・ファイルを生成します。

-apd アセンブリ・ファイルのための前処理を実行します。ただし前処理された出力を書き込むのではなく、標準 make ユーティリティへの入力に適した依存行のリストを書き込みます。このリストは、ソース・ファイルと名前が同じであり .ppa 拡張子をもつファイルに書き込まれます。

-api アセンブリ・ファイルのための前処理を実行します。ただし前処理された出力を書き込むのではなく、#include 疑似命令で組み込まれるファイルのリストを書き込みます。このリストは、ソース・ファイルと名前が同じであり .ppa 拡張子をもつファイルに書き込まれます。

-as -s アセンブラ・オプションを指定してアセンブラを起動し、シンボル ・テーブルにラベルを格納します。ラベル定義は、シンボリック・デバッグで使用できるように COFF シンボル・テーブルに書き込まれます。

-au*name* 事前定義された定数 *name* を未定義にし、指定した定数に対するすべての -ad オプションを無効にします。

-ax -x アセンブラ・オプションを指定してアセンブラを起動し、リスト・ファイル内にシンボリック・クロスリファレンスを作成させます。

アセンブラ・オプションの詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

2.3.9 非推奨オプション

一部のコンパイラ・オプションは非推奨になります。コンパイラは引き続きこれらのオプションを受け付けますが、その使用は推奨しません。今後リリースされるツールは、これらのオプションをサポートしません。表 2-2 に、非推奨オプションとそれを置き換えているオプションを示します。

表 2-2. コンパイラ下位互換オプションのまとめ

旧オプション	機能	新オプション
-gp	最適化されたコードの関数レベルのプロファイリングを認めます。	-g
-gt	代替 STABS デバッグ・フォーマットを使ったシンボリック・デバッグを有効にします。	--symdebug:coff
-gw	DWARF デバッグ・フォーマットを使ったシンボリック・デバッグを有効にします。	--symdebug:dwarf または -g

--symdebug:profile_coff オプションは、STABS デバッグ・フォーマット（--symdebug:coff または -gt オプション）を指定して、シンボリック・デバッグを使った最適化されたコードの関数レベルのプロファイリングを有効にするために追加されました。

2.4 環境変数によるコンパイラの動作の変更

通常使用する特定のソフトウェア・ツール・パラメータを設定する環境変数を定義できます。*環境変数*はユーザが定義してシステム初期化ファイルの文字列に関連付ける特別なシステム・シンボルです。コンパイラは、このシンボルを使って特定の種類の情報を検索または取得します。

環境変数を使用すると、デフォルト値が設定され、これらのパラメータが自動的に指定されるため、コンパイラを毎回起動することが簡単になります。ツールの起動時、コマンド行オプションを指定すると、環境変数で設定されるデフォルト値の多くを上書きできます。

2.4.1 コンパイラが検索する代替ディレクトリの指定方法（C_DIR）

コンパイラは、C_DIR 環境変数を使い、#include ファイルを含む代替ディレクトリ名を指定します。#include ファイルのディレクトリを指定するには、次の構文のいずれかを使って C_DIR を設定します。

オペレーティング・システム	入力する内容
Windows™	**set C_DIR=***directory1*[**;***directory2* ...]
UNIX（Bourne シェル）	**C_DIR="***directory1* [**;***directory2* ...]**"; export C_DIR**

環境変数は、システムを再起動するか変数をリセットするまで保持されます。

2.4.2 デフォルトのコンパイラ・オプションの設定方法（C_OPTION および C2000_C_OPTION）

C2000_C_OPTION または C_OPTION 環境変数を使用して、コンパイラ、アセンブラ、リンカのデフォルトのオプションを設定すると便利です。この方法で設定すると、コンパイラを実行するたびに、これらの変数に設定したデフォルトのオプションまたは入力ファイル名が使用されます。

C_OPTION 環境変数によりデフォルト・オプションを設定する方法は、同じオプションや入力ファイル（またはその両方）を使用して連続してコンパイラを実行する場合に便利です。コマンド行と入力ファイル名を読み込んだ後、コンパイラはまず C2000_C_OPTION 環境変数を探してから、それを読み込んで処理します。C2000_C_OPTION が検出されない場合、コンパイラは C_OPTION 環境変数を読み込んで処理します。

次の表に、C_OPTION 環境変数を設定する方法を示します。ご使用のオペレーティング・システムに対応するコマンドを選択してください。

オペレーティング・システム	入力する内容
UNIX（Bourne シェル）	**C_OPTION="***option*$_1$ [*option*$_2$ . . .]**";** **export C_OPTION**
Windows	**set C_OPTION=***option*$_1$[;*option*$_2$ . . .]

環境変数オプションはコマンド行での場合と同じように指定され、同じ意味をもちます。たとえば、Windows 上で、常に静的に実行し（-q オプション）、C/C++ ソース差し込みを有効にし（-s オプション）、リンクする（-z オプション）場合は、次のように C_OPTION 環境変数を設定します。

```
set C_OPTION=-qs -z
```

次の例では、コンパイラを実行するたびにリンカが実行されます。コマンド行または C_OPTION 内の -z の後に指定したオプションは、すべてリンカに渡されます。これにより、C_OPTION 環境変数を使用してデフォルトのコンパイラとリンカのオプションを指定した後、追加のコンパイラとリンカのオプションをコマンド行で指定できます。環境変数に -z を設定しているがコンパイルだけを行いたいという場合は、コンパイラの -c オプションを使用します。次のコマンド例では、上記で説明したように C_OPTION が設定されていることを前提としています。

```
cl2000 -v28 *c                ; compiles and links
cl2000 -v28 -c *.c            ; only compiles
cl2000 -v28 *.c -z lnk.cmd    ; compiles/links using cmd file
cl2000 -v28 -c *.c -z lnk.cmd ; only compiles -c overrides -z)
```

コンパイラ・オプションの詳細は、2.3 節「オプションによるコンパイラの動作の変更」（2-5 ページ）を参照してください。リンカ・オプションの詳細は、4.2 節「リンカ・オプション」（4-5 ページ）を参照してください。

2.4.3 一時ファイルのディレクトリ指定方法（TMP）

コンパイラは、プログラムを処理する際に中間ファイルを作成します。デフォルトでは、コンパイラは中間ファイルをカレント・ディレクトリに格納します。しかし、TMP 環境変数を使用して、一時ファイル用の特定のディレクトリを指定することができます。

TMP 環境変数を使用すると、RAM ディスクまたは他のファイル・システムを使うことができます。また、ソースが含まれているディレクトリにファイルを一切書き込まずに、別のディレクトリでソース・ファイルをコンパイルすることができます。これは書き込みが保護されているディレクトリの場合に有効です。

次の表に、TMP 環境変数を設定する方法を示します。ご使用のオペレーティング・システムに対応するコマンドを選択してください。

オペレーティング・システム	入力する内容
UNIX（Bourne シェル）	**TMP="***pathname***"**; **export TMP**
Windows	**set TMP=***pathname*

注： UNIX ワークステーションの場合、必ずディレクトリ名を引用符で囲んでください。

たとえば、Windows のハードディスク・ドライブ上に中間ファイルを格納する temp というディレクトリを一時ファイルの格納場所として指定するには、次のように入力します。

```
set TMP=c:\temp ↵
```

2.5 プリプロセッサの制御方法

ここでは、パーサの一部である C28x プリプロセッサの特殊機能について説明します。C の前処理の一般的な説明については、K&R の A12 節を参照してください。C28x C/C++ コンパイラには、標準 C/C++ 前処理機能が含まれています。この機能は、コンパイラの最初のパスに組み込まれています。プリプロセッサが処理する内容は次のとおりです。

- マクロ定義と展開
- #include ファイル
- 条件付きコンパイル
- その他のさまざまなプリプロセッサ疑似命令（ソース・ファイルの # 文字で始まる行で指定されたもの）

プリプロセッサは、一目瞭然のエラー・メッセージを生成します。エラーが発生した行番号とファイル名は、診断メッセージと一緒に表示されます。

2.5.1 事前定義マクロ名

コンパイラは 表 2-3 に示す事前定義マクロ名を保管し、認識します。

表 2-3. 事前定義マクロ名

マクロ名	説明
__DATE__ †	mm dd yyyy 形式でコンパイルした日付に展開します。
__FILE__ †	カレント・ソース・ファイル名に展開します。
_INLINE	-x または -x2 オプションを指定した場合は 1 に展開します。指定しない場合は、未定義になります。
__LINE__ †	カレント行番号に展開します。
__TI_COMPILER_VERSION__	カレント・コンパイラ・バージョン番号を示す整数値に展開します。たとえば、バージョン 1.20 は、120 として示されます。
__TIME__ †	hh:mm:ss 形式でコンパイル時刻に展開します。
__TMS320C2000__	1 に展開します（C28x または C27x のプロセッサを識別します)。
__TMS320C28XX__	1 に展開します（C28x プロセッサを識別します)。
__LARGE_MODEL__	-ml オプションを指定した場合は 1 に展開します。指定しない場合は、未定義になります。

† ANSI/ISO 規格で定義

表 2-3. のマクロ名は、他の定義名と同じように使用できます。たとえば次のとおりです。

```
printf ( "%s %s" , __TIME__ , __DATE__);
```

この場合は、次のような行に変換されます。

```
printf ("%s %s" , "13:58:17", "Jan 14 1997");
```

2.5.2 #include ファイル検索パス

#include プリプロセッサ疑似命令は､別のファイルからソース文を読み取るようにコンパイラに指示します。ファイルを指定する際には、ファイル名を二重引用符か不等号括弧で囲んでください。ファイル名には、絶対パス名、部分的なパス情報、またはパス情報なしのファイル名のいずれかを指定できます。

❑ ファイル名を二重引用符（“ ”）で囲んだ場合、コンパイラは、以下の順にディレクトリを検索して該当のファイルを探します。

1) カレント・ソース・ファイルを格納するディレクトリ。カレント・ソース・ファイルとは、コンパイラが #include 疑似命令を検出したときにコンパイルされているファイルを指します。

2) **-I** オプションで指定されたディレクトリ

3) C_DIR 環境変数で設定されたディレクトリ

❏ ファイル名を不等号括弧（< >）で囲んだ場合、コンパイラは次の順にディレクトリを検索して該当のファイルを探します。

1) **-I** オプションで指定されたディレクトリ

2) C_DIR 環境変数で設定されたディレクトリ

-I オプションの使用方法については、2.5.2.1 項「#include ファイル検索パスの変更（-I オプション）」を参照してください。

2.5.2.1 #include ファイル検索パスの変更（-I オプション）

-I オプションは、#include ファイルを格納する代替ディレクトリ名を指定します。
-I オプションの形式は次のとおりです。

-I=*directory1* [**-I**=*directory2* ...]

それぞれの -I オプションが 1 つの *directory* 名を指定します。C/C++ ソースでは、ファイルについてのディレクトリ情報を指定せずに、#include 疑似命令を使用することができます。その場合は、**-I** オプションでディレクトリ情報を指定します。たとえば、カレント・ディレクトリに source.c という名前のファイルがあるとします。この source.c ファイルには、次のような疑似命令文が入っています。

```
#include "alt.h"
```

alt.h の完全なパス名を次のように仮定します。

Windows	c:\c28xtools\files\alt.h
UNIX	/c28xtools/files/alt.h

次の表に、コンパイラの起動方法を示します。ご使用のオペレーティング・システムに対応するコマンドを選択してください。

オペレーティング・システム	入力する内容
Windows	cl2000 -v28 -I=c:\c28xtools\files source.c
UNIX	cl2000 -v28 -I=/c28xtools/files source.c

注：　不等号括弧でパス情報を指定する方法

不等号括弧でパス情報を指定すると、コンパイラは -I オプションや C_DIR 環境変数で指定されたパス情報に関連した情報を適用します。

たとえば、次のコマンドを使用して C_DIR を設定する場合

```
set C_DIR= c:\project\include
```

または、次のコマンドを使ってコンパイラを起動する場合

```
cl2000 -v28 -i c:\project\include file.c
```

いずれの場合でも結果的に file.c には、次の行が含まれます。

```
#include <module\proc.h>
```

インクルード・ファイルは、次のようなパスになります。

```
c:\project\include\module\proc.h
```

2.5.3　前処理リスト・ファイルの生成方法（-ppo オプション）

-ppo オプションを指定すると、ソース・ファイルの前処理したバージョンを作成することができます。この前処理したファイルは、ソース・ファイルと名前が同じですが、拡張子は .pp です。コンパイラの前処理機能は、ソース・ファイル上で以下の操作を実行します。

- バックスラッシュ（\）で終わっている各ソース行と、次の行との連結
- 3 文字符号系列の展開
- コメントの除去
- ファイルへの #include ファイルのコピー
- マクロ定義の処理
- すべてのマクロの展開
- #line 疑似命令、条件付きコンパイルなど、他のすべての前処理疑似命令の展開

2.5.4　前処理後のコンパイルの続行方法（-ppa オプション）

前処理を行う場合、プリプロセッサは前処理だけを行います。デフォルトではソース・コードをコンパイルしません。この機能を指定変更し、ソース・コードの前処理後にコンパイルを続行したい場合は、他の処理オプションと一緒に -ppa オプションを使用します。たとえば -ppa を -ppo と一緒に使用して前処理を実行し、前処理された出力を .pp 拡張子をもつファイルに書き込んでから、ソース・コードをコンパイルします。

2.5.5 コメント付き前処理リスト・ファイルの生成方法（-ppc オプション）

-ppc オプションはコメントの除去を除いてすべての前処理機能を実行し、.pp 拡張子が付いた前処理済みバージョンのソース・ファイルを生成します。コメントを保持したい場合は、-ppo オプションではなく -ppc オプショ ンを指定します。

2.5.6 行の制御情報付き前処理リスト・ファイルの生成（-ppl オプション）

デフォルトでは、前処理された出力ファイルにはプリプロセッサ疑似命令は入っていません。#line 疑似命令を出力させたい場合は、-ppl オプションを指定してください。-ppl オプションは前処理だけを実行し、名前がソース・ファイルと同じであり .pp 拡張子が付いたファイルに、行の制御情報（#line 疑似命令）と一緒に前処理済み出力を書き込みます。

2.5.7 Make ユーティリティ用の前処理出力の生成方法（-ppd オプション）

-ppd オプションは前処理だけを実行します。ただし前処理された出力を書き込むのではなく、標準 make ユーティリティへの入力に適した従属行のリストを書き込みます。このリストは、ソース・ファイルと名前が同じであり .pp 拡張子が付いたファイルに書き込まれます。

2.5.8 #include 疑似命令で組み込むファイルのリストの生成方法（-ppi オプション）

-ppi オプションは前処理だけを実行します。ただし前処理された出力を書き込むのではなく、#include 疑似命令で組み込まれているファイルのリストを書き込みます。このリストは、ソース・ファイルと名前が同じであり .pp 拡張子が付いたファイルに書き込まれます。

2.6 診断メッセージの概要

コンパイラの主な機能の 1 つは、ソース・プログラムの診断情報を報告することです。コンパイラは疑わしい状態を検出すると、次の形式のメッセージを表示します。

"*file.cpp*", line *n***:** *diagnostic severity***:** *diagnostic message*

"*file.cpp***"**	ファイル名を示します。
line *n***:**	診断が適用された行番号を示します。
diagnostic severity	診断メッセージの重大度を示します(各重大度カテゴリの説明が、その後に続きます)。
diagnostic message	問題を説明するテキストを示します。

診断メッセージには、次のような重大度が関連付けられています。

- **fatal error（致命的エラー）**は、コンパイルが続行できないほどの重大な問題が発生したことを示します。致命的エラーの原因となる問題の例には、コマンド行エラー、内部エラー、および include ファイルの欠落などが含まれます。複数のソース・ファイルをコンパイルしている場合には、問題の発生したカレント・ソース・ファイルより後のソース・ファイルはコンパイルされません。
- **error（エラー）**は、C/C++　言語の構文規則または意味構造規則の違反を示します。コンパイルは続行されますが、オブジェクト・コードは生成されません。
- **warning（警告）**は、有効であるが疑わしい状況を示します。コンパイルは続行され、オブジェクト・コードが生成されます（エラーが検出されない場合)。
- **remark（注釈）**は、警告より重大度が低いものです。有効であり、おそらく意図されたものであるが、チェックが必要な問題を示します。コンパイルは続行され、オブジェクト・コードが生成されます（エラーが検出されない場合）。デフォルトでは注釈は発行されません。注釈を有効にするには、-pdr コンパイラ・オプションを指定してください。

診断は、次の例のような形式で標準エラーに書き込まれます。

```
"test.c", line 5: error: a break statement may only be used
          within a loop or switch
   break;
   ^
```

デフォルトではソース行は省略されます。ソース行とエラー位置を表示できるようにするには、-pdv コンパイラ・オプションを指定してください。上記の例では、このオプションを使用しています。

このメッセージは診断が行われたファイルと行を識別し、メッセージの後に問題のあるソース行が（^ 記号が示す位置とともに）表示されます。複数の診断が 1 つのソース行に適用される場合、診断が行われるたびに上記の形式で表示されます。ソース行のテキストも複数回表示され、そのたびに該当する位置が指示されます。

メッセージが長い場合は、必要に応じて次の行に折り返されます。

コマンド行オプション（-pden）を使用すると、診断の数値識別子を診断メッセージに含めるように要求できます。診断識別子が表示される場合、診断がコマンド行で重大度を無効にすることができるかどうかも指示されます。重大度を無効にできる場合、診断識別子には接尾部 -D（*discretionary* 自由裁量）が含まれます。無効にできない場合、接尾部はありません。たとえば次のとおりです。

```
"Test_name.c",line 7: error #64-D: declaration does not
          declare anything
   struct {};
   ^

"Test_name.c",line 9: error #77: this declaration has no
          storage class or type specifier
   xxxxx;
   ^
```

エラーが自由裁量かどうかは特定のコンテキストに関連付けられたエラー重大度に基づいて決まるので、同じエラーがある場合は自由裁量で、別の場合はそうでないこともあります。warning（警告）と remark（注釈）は、すべて自由裁量です。

一部のメッセージの場合、エンティティ（関数、ローカル変数、ソース・ファイルなど）のリストが有効です。エンティティは、初期エラー・メッセージの後に表示されます。

```
"test.c", line 4: error: more than one instance of overloaded
          function "f" matches the argument list:
             function "f(int)"
             function "f(float)"
             argument types are: (double)
   f(1.5);
   ^
```

場合によっては、追加のコンテキスト情報が提供されます。特にコンテキスト情報が便利なのは、フロント・エンドがテンプレートのインスタンス生成中、またはコンストラクタ、デストラクタ、代入演算子関数のいずれかを生成中に診断を発行する場合です。たとえば次のとおりです。

```
"test.c", line 7: error: "A::A()" is inaccessible
   B x;
     ^
          detected during implicit generation of "B::B()" at
          line 7
```

コンテキスト情報がないと、エラーが参照している内容の判別が難しくなります。

2.6.1 診断の制御方法

コンパイラが提供する診断オプションを使用すると、パーサがコードを解釈する方法を修正できます。これらのオプションを使って診断を制御できます。

-pdel*num*
: エラー数の上限を *num* に設定します。任意の 10 進数値を指定できます。エラー数がこの指定値に達すると、コンパイラはコンパイルを中止します（デフォルトは 100 です)。

-pden
: 診断の数値識別子を、そのテキストと一緒に表示します。診断抑止オプション（-pds、-pdse、-pdsr、-pdsw）に指定する引数を見つけるために、このオプションを使用します。

また、このオプションは診断が自由裁量であるかどうかも表示します。自由裁量の診断とは、重大度を無効にできる診断です。自由裁量の診断には接尾部 -D が含まれ、自由裁量でない診断には接尾部がありません 。詳細は、2.6 節「診断メッセージの概要」(2-32 ページ）を参照してください。

-pdf
: 対応するソース・ファイルと同じ名前で拡張子が *.err* の診断情報ファイルを生成します。

-pdr
: 注釈（軽い警告）を発行します。注釈はデフォルトでは抑止されます。

-pds *num*
: *num* の識別する診断を抑止します。診断メッセージの数値識別子を判別するには、別のコンパイルで最初に -pden オプションを指定してください。その後、-pds *num* を使用して診断を抑止します。抑止できるのは自由裁量の診断だけです。

-pdse *num*
: *num* の識別する診断をエラーに分類します。診断メッセージの数値識別子を判別するには、別のコンパイルで最初に -pden オプションを指定してください。その後、-pds *num* を指定して診断をエラーに分類し直します。変更できるのは、自由裁量の診断の重大度だけです。

-pdsr *num*
: *num* の識別する診断を注釈に分類します。診断メッセージの数値識別子を判別するには、別のコンパイルで最初に -pden オプションを指定してください。その後、-pdsr*num* を指定して診断を注釈に分類し直します。変更できるのは、自由裁量の診断の重大度だけです。

-pdsw *num*
: *num* の識別する診断を警告に分類します。診断メッセージの数値識別子を判別するには、別のコンパイルで最初に -pden オプションを指定してください。その後、-pdsw*num* を指定して診断を警告に分類し直します。変更できるのは、自由裁量の診断の重大度だけです。

-pdv
: 行の折り返し付きでオリジナル・ソースを表示し、ソース行内のエラーの位置を示す詳細な診断情報を提供します。

-pdw
: 警告診断を抑止します（エラーは発行されます)。

2.6.2 診断抑止オプションの使用方法

次の例は、コンパイラが発行する診断メッセージの制御方法を示しています。

次のコード・セグメントを考えてみましょう。

```
int one();
int i;
int main()
{
   switch (i){
   case  1:
         return one ();
         break;
   default:
         return 0;
         break;
   }
}
```

-q オプションを指定して本コンパイラを起動すると、次のような結果になります。

```
"err.cpp", line 9: warning: statement is unreachable
"err.cpp", line 12: warning: statement is unreachable
```

フォールスルー状態を避けるために、各 case 文の終わりに break 文を入れることは標準的なプログラミング方法なので、これらの警告は無視できます。-pden オプションを指定すると、これらの警告の診断識別子を検出できます。結果は次のとおりです。

```
[err.cpp]
"err.cpp", line 9: warning #111-D: statement is unreachable
"err.cpp", line 12: warning #111-D: statement is unreachable
```

次に、診断識別子 111 を -pdsr オプションの引数として指定すると、この警告を注釈として扱うことができます。(注釈はデフォルトで抑止されるので) このコンパイルでは、診断メッセージは生成されなくなります。

こうした診断メッセージの出力を制御することは便利ですが、常に危険を伴います。コンパイラからは、問題に発展しそうな兆候を示すメッセージが出力されていることがあるからです。したがって、診断メッセージの出力を十分に分析してから、この抑止オプションを使用するようにしてください。

2.6.3 その他のメッセージ

ソースに関連していないその他のエラー・メッセージ（コマンド行の構文が正しくない、指定されたファイルが見つからないなど）は通常、致命的エラーを示します。メッセージの前には、文字列「>> WARNING:」または「>> ERROR:」のいずれかが付いています。

たとえば次のとおりです。

```
cl2000 -v28 -j
>> WARNING: invalid option -j (ignored)
>> ERROR: no source files
```

2.7 クロスリファレンス・リスト情報の生成方法（-px オプション）

-px コンパイラ・オプションを指定するとクロスリファレンス・リスト・ファイルが生成されます。このファイルには、ソース・ファイル内の識別子に対する参照情報が含まれています（-px オプションは -ax とは別です。-ax はパーサ・オプションではなく、アセンブラ・オプションです）。クロスリファレンス・リスト・ファイル内の情報は、次の形式で表示されます。

sym-id　name　X　filename　line number　column number

sym-id	各識別子に固有に割り当てられた整数
name	識別子の名前
X	次の値のいずれか 1 つ

X 値	意味
D	定義
d	宣言（定義ではない）
M	修正
A	取得アドレス
U	使用
C	変更（1 つの操作で使用され、修正された）
R	その他の種類の参照
E	エラー。参照は不確定

filename	ソース・ファイル
line number	ソース・ファイル内の行番号
column number	ソース・ファイル内のカラム番号

2.8 ロー・リスト・ファイルの生成方法（-pl オプション）

-pl オプションを指定するとロー・リスト・ファイル（.rl）が生成されます。このファイルにより、ユーザは、コンパイラがどのようにソース・ファイルを前処理するかを理解することができます。（-ppo、-ppc、および -ppl プリプロセッサ・オプションを指定して生成された）前処理リスト・ファイルはソース・ファイルの前処理済みバージョンを表示するのに対して、ロー・リスト・ファイルにはオリジナル・ソース行と前処理出力の比較情報が示されます。ロー・リスト・ファイルには次の情報が含まれています。

- 各オリジナル・ソース行
- include ファイルへの移行と復帰
- 診断結果
- 重要な処理が実行された場合は、前処理されたソース行（コメントの除去は重要でないと見なされ、他の前処理は重要と見なされます）。

ロー・リスト・ファイル内の各ソース行は、図 2-4 にリストされている識別子のいずれかで始まります。

表 2-4.　ロー・リスト・ファイルの識別子

識別子	定義
N	通常のソース行
X	展開されたソース行。重要な前処理が行われた場合、通常のソース行の直後に表示されます。
S	スキップされたソース行（偽の #if 句）
L	ソース位置の変更。これは次の形式で表示されます。 L *line number filename key* この *line number* は、ソース・ファイル内の行番号です。*key* が表示されるのは、変更の原因が include ファイルの開始 / 終了の場合だけです。*key* の値は、次のとおりです。 1 = include ファイルの開始 2 = include ファイルの終了

-pl オプションには、表 2-5 に定義されている診断識別子も含まれています。

表 2-5.　ロー・リスト・ファイル診断識別子

診断識別	定義
E	エラー（Error）
F	致命的エラー（Fatal）
R	注釈（Remark）
W	警告（Warning）

診断ロー・リスト情報は次の形式で表示されます。

S filename line number column number diagnostic

S	表 2-5 内の識別子の 1 つで、診断の重大度を示します。
filename	ソース・ファイル
line number	ソース・ファイル内の行番号
column number	ソース・ファイル内のカラム番号
diagnostic	診断メッセージ本文

ファイルの終わりの後の診断は、カラム番号 0 をもつファイルの最終行として表示されます。診断メッセージが複数行にわたる場合は、後続の各行には、同じファイル、行、およびカラム情報が入りますが、診断識別子は小文字が使用されます。診断メッセージの詳細は、2.6 節「診断メッセージの概要」（2-32 ページ）を参照してください。

2.9 インライン関数展開の使用方法

インライン関数を呼び出すと、その関数の C/C++ ソース・コードが呼び出し位置に挿入されます。これはインライン関数展開と呼ばれます。インライン関数展開は、次の理由から小さい関数には有効です。

- ❏ 1 回の関数呼び出しのオーバーヘッドを削減できます。
- ❏ インライン展開を行うと、本オプティマイザは周囲のコードに合わせて自由に関数を最適化できます。

インライン関数展開には次のような複数のタイプがあります。

- ❏ 組み込み演算子のインライン展開（組み込み関数は常にインライン展開されます）
- ❏ 自動インライン展開
- ❏ 保護されない inline キーワードを使用した定義制御インライン展開
- ❏ 保護される inline キーワードを使用した定義制御インライン展開

注： 関数のインライン展開によりコード・サイズが大幅に増大する可能性がある

関数をインライン展開するとコード・サイズが増大します。特に複数の場所で呼び出された関数をインライン展開する場合、増大します。関数のインライン展開は、少ない場所からのみ呼び出される関数および小さい関数に適しています。

2.9.1 組み込み演算子のインライン展開

C28x には多数の組み込み演算子があります。コンパイラは組み込み演算子を効率のよいコード（通常 1 つの命令）に置き換えます。インライン展開は、オプティマイザを使用するかどうかに関係なく自動的に行われます。

組み込み関数の詳細、および組み込み関数のリストについては、7.4.5 項「組み込み関数（intrinsics）を使用してアセンブリ言語文にアクセスする方法」（7-26 ページ）を参照してください。

2.9.2 自動インライン展開

-O3 オプションを指定して C/C++ ソース・コードをコンパイルする場合、インライン関数展開は小さい関数で実行されます。詳細は、3.5 節「自動インライン展開（-oi オプション）」（3-12 ページ）を参照してください。

2.9.3 保護されない定義制御インライン展開

inline キーワードは、標準の呼び出し手続きを使用するのではなく、呼び出し位置で関数をインライン展開することを指定します。コンパイラは、inline キーワードを指定して宣言された関数のインライン展開を実行します。

定義制御されたインライン展開をオンにするには、任意の -O オプション（-O0、-O1、-O2、-O3）を指定してオプティマイザを起動する必要があります。また、-O3 を指定する場合は自動インライン展開もオンになります。

次の例は inline キーワードの使用方法を示しています。ここでは、関数呼び出しが呼び出し先関数の中のコードで置き換えられます。

例 2-1. inline キーワードの使用方法

```
inline int volume_sphere(float r)
{
    return 4.0/3.0 * PI * r * r * r;
}
int foo(...)
{
    ...
    volume = volume_sphere(radius);
    ...
}
```

-pi オプションは定義制御インライン展開をオフにします。このオプションは、特定のレベルの最適化が必要で、定義制御インライン展開は不要な場合に便利です。

2.9.4 保護されたインライン展開と _INLINE プリプロセッサ・シンボル

ヘッダ・ファイル内で関数を static inline として宣言すると、-pi によりインライン展開がオフにされた場合や、オプティマイザが稼働しない場合には、コード・サイズが増大する危険性が潜在します。これを回避するためには、追加の手順が必要です。

ヘッダ・ファイル内の static inline 関数が、インライン展開がオフになったときにコード・サイズを増大させないようにするには、次の手順を実行してください。これにより、インライン展開がオフになったときに外部とのリンクが可能になります。したがって、オブジェクト・ファイル全体で 1 つの関数定義だけが存在します。

- 関数の static inline バージョンのプロトタイプを作成します。その後、その関数の代替の静的でない外部とリンクされるバージョンのプロトタイプを作成します。例 2-2 に示すように、これらの 2 つのプロトタイプを、_INLINE プリプロセッサ・シンボルを使用して条件付きで前処理します。
- 例 2-2 に示すように、.c または .cpp ファイル内に同一バージョンの関数定義を作成します。

例 2-2 には、関数の定義が 2 つあります。最初の定義はヘッダ・ファイル内のもので、インライン定義です。この定義が有効になり static inline としてプロトタイプが宣言されるのは、_INLINE が真の場合だけです（オプティマイザが使用されるときに -pi が指定されていないと、_INLINE は自動的に定義されます）。

2 番目の定義により、インライン展開が無効にされたときには必ず呼び出し可能な関数が存在することが保証されます。これはインライン関数ではないので、ヘッダ・ファイルが組み込まれ関数のプロトタイプの非インライン・バージョンが生成される前に _INLINE プリプロセッサ・シンボルは定義解除（#undef）されます。

例 2-2. ランタイムサポート・ライブラリでの _INLINE プリプロセッサ・シンボルの使用方法

(a) foo.h

```
#ifdef _INLINE
#define _IDECL static inline
#else
#define _IDECL extern
#endif
_IDECL int len (const char *str);
#ifdef _INLINE
static inline int len(const char *str)
{
   int n         = -1;
   const char *s = str - 1;
   do n++; while (*++s);
   return n;
}

#endif
```

(b) foo.c

```
#undef _INLINE

#include "foo.h"

int len(cont char * str)
{
   int n         = -1;
   const char *s = str - 1;

   do n++; while (*++s);
   return n;
}
```

2.9.5 インライン展開の制約事項

自動インライン展開と定義制御インライン展開の両方に対して、どの関数をインライン展開できるかに関していくつかの制約事項があります。ローカルな静的変数や可変数の引数をもつ関数はインライン展開されませんが、static inline として宣言された関数は展開されます。static inline として宣言された関数の場合、ローカルな静的変数が存在しても展開が行われます。また、再帰関数やノンリーフ関数に対してはインライン展開の深さが制限されます。さらに、インライン展開は小さい関数や呼び出し箇所が少ない関数に対して使用してください（ただし、コンパイラがこれを強制することはありません)。

次の条件に当てはまる関数は、インライン展開が不適格です。

- ❏ struct または union を戻す
- ❏ struct または union パラメータをもつ
- ❏ volatile パラメータをもつ
- ❏ 可変長引数リストをもつ
- ❏ struct、union、または enum 型が宣言されている
- ❏ 静的変数を含む
- ❏ volatile 変数を含む
- ❏ 再帰的である
- ❏ プラグマを含む
- ❏ スタックが大きすぎる（ローカル変数が多すぎる）

2.10 インターリストの使用方法

コンパイラ・ツールを使うと、C/C++ ソース文をコンパイラのアセンブリ言語出力に差し込むことができます。インターリストにより、各 C/C++ ソース文に対して生成されたアセンブリ・コードを検査できます。インターリストの動作は、オプティマイザの使用の有無や指定するオプションにより異なります。

インターリスト機能を起動する最も簡単な方法は、-ss オプションを指定することです。function.c というプログラムをコンパイルし、インターリストを実行するには、次のように入力します。

```
cl2000 -v28 -ss function
```

-ss オプションはコンパイラに対して、差し込まれたアセンブリ言語出力ファイルを削除しないように指示します。出力されたアセンブリ・ファイルである function.asm は、正常にアセンブルされます。

オプティマイザなしでインターリストを起動する場合には、インターリストはコード・ジェネレータとアセンブラの間の独立したパスとして実行されます。インターリストは、アセンブリ・ファイルと C/C++ ソース・ファイルの両方を読み込んでマージし、アセンブリ・ファイルの中に C/C++ ソース文をコメントとして書き込みます。

例 2-3 に、差し込み後のアセンブリ・ファイルの一般的な例を示します。オプティマイザのインターリスト・ユーティリティについての詳細は、3.6 節「インターリスト・ユーティリティをオプティマイザと組み合わせて使用する方法」(3-13 ページ)を参照してください。

例 2-3.　差し込み後のアセンブリ言語ファイル

```
;----------------------------------------------------------------------
;   1 | int main()
;----------------------------------------------------------------------

;***************************************************************
;* FNAME: _main                         FR SIZE:   0           *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                             0 Parameter,  0 Auto,  0 SOE    *
;***************************************************************

_main:
;----------------------------------------------------------------------
;   3 | printf("Hello World\n");
;----------------------------------------------------------------------
        MOVL      XAR4,#SL1             ; |3|
        LCR       #_printf              ; |3|
        ; call occurs [#_printf] ; |3|
;----------------------------------------------------------------------
;   4 | return 0;
;----------------------------------------------------------------------

;***************************************************************
;* STRINGS                                                     *
;***************************************************************
        .sect    “.const”
SL1:    .string “Hello World”,10,0
;***************************************************************
;* UNDEFINED EXTERNAL REFERENCES                               *
;***************************************************************
        .global _printf
```

第 3 章

コードの最適化方法

コンパイラ・ツールには最適化プログラムが含まれ、ループの簡略化、文と式の再配置、およびレジスタへの変数の割り当てなどを実行し、C/C++ プログラムの実行速度の向上およびプログラム・サイズの縮小を行います。

本章では、オプティマイザの起動方法と、使用時に実行される最適化について説明します。また、オプティマイザでインターリスト機能を使用する方法、および最適化されたコードのプロファイルまたはデバッグを行う方法についても説明します。

項目 **ページ**

3.1	C/C++ コンパイラのオプティマイザの使用方法	3-2
3.2	ファイルレベルの最適化の実行	3-4
3.3	プログラムレベルの最適化の実行（-pm および -O3 オプション）	3-6
3.4	オプティマイザを使用する場合の特別な注意事項	3-10
3.5	自動インライン展開（-oi オプション）	3-12
3.6	インターリスト・ユーティリティをオプティマイザと組み合わせて使用する方法	3-13
3.7	最適化されたコードのデバッグ方法	3-16
3.8	コード・サイズ最適化の強化（-ms オプション）	3-18
3.9	実行できる最適化の種類	3-20

3.1 C/C++ コンパイラのオプティマイザの使用方法

C/C++ コンパイラでは、さまざまな最適化が実行できます。オプティマイザでは最適なコードを得るために高レベルな最適化が実行されます。

最適化を起動する最も簡単な方法は、cl2000 -v28 コマンド行で -O*n* オプションを指定することです。*n* は最適化のレベル（0、1、2、3）を表し、最適化の種類と程度を制御します。

❑ **-O0**

- 制御フロー・グラフを簡略化します。
- 変数をレジスタに割り当てます。
- ループの循環を行います。
- 不要コードを除去します。
- 式と文を簡略化します。
- インライン関数として宣言された関数の呼び出しを展開します。

❑ **-O1**

-O0 のすべての最適化の他に、以下の最適化を実行します。

- ローカルなコピー / 定数の伝播を行います。
- 不要な代入を除去します。
- ローカルな共通式を除去します。

❑ **-O2**

-O1 のすべての最適化の他に、以下の最適化を実行します。

- ループの最適化を行います。
- グローバルな共通部分式を除去します。
- 不要でグローバルな代入を除去します。

-O に最適化レベルを指定しない場合は、-O2 がデフォルトとして使用されます。

❑ **-O3**

-O2 のすべての最適化の他に、以下の最適化を実行します。

- 一度も呼び出されていない関数をすべて除去します。
- 戻り値が一度も使用されていない関数を簡略化します。
- 小さな関数の呼び出しをインライン展開します。
- 呼び出し側を最適化するとき、呼び出された関数の属性がわかるように関数宣言を並べ換えます。
- すべての呼び出し側が同じ引数の位置にある同じ値を渡すときに、引数を関数本体に伝播します。
- ファイルレベルの変数特性を特定します。

-O3 を使用する場合の詳細については、3.2 節「ファイルレベルの最適化の実行」（3-4 ページ）を参照してください。

前述の最適化レベルは、スタンドアロンの最適化パスにより実行されます。コード・ジェネレータは、複数の追加の最適化を実行します。特にプロセッサ固有の最適化を実行しますが、これはオプティマイザを起動するかどうかに関わらず実行されます。これらの最適化は、オプティマイザの使用時により効率的ですが、常に有効化されています。

3.2　ファイルレベルの最適化の実行

-O3 オプションは、コンパイラにファイルレベルの最適化を行うよう指示します。-O3 オプションは、単独で使用して一般的なファイルレベルの最適化を実行することも、他のオプションと組み合わせてより具体的な最適化を実行することもできます。表 3-1 に、表中に記述する最適化を実行するために -O3 と組み合わせて使用するオプションを示します

表 3-1.　-O3 と組み合わせて使用できるオプション

状況	使用するオプション	ページ
標準ライブラリ関数を再宣言するファイルがある。	-ol*n*	3-4
最適化情報ファイルを作成する。	-on*n*	3-5
複数のソース・ファイルをコンパイルする。	-pm	3-6

3.2.1　ファイルレベルの最適化の実行（-ol オプション）

-O3 オプションを指定してオプティマイザを起動すると、いくつかの最適化に標準ライブラリ関数の定義済みのプロパティが使用されます。それらの標準ライブラリ関数のいずれかを再宣言するファイルがある場合、これらの最適化は無効になります。-ol（小文字の L）オプションは、ファイルレベルの最適化を制御します。-ol に続く番号は、レベル（0、1、2）を表します。表 3-2 を使用して、-ol オプションに指定する適切なレベルを選択してください

表 3-2.　-ol オプションのレベルの選択方法

ソース・ファイルの状況	使用するオプション
標準ライブラリ関数と同じ名前の関数を宣言し、変更している。	-ol0
標準ライブラリ関数と同じライブラリ関数の定義を含んでいるが、標準ライブラリで宣言された関数を変更していない。	-ol1
標準ライブラリ関数を変更しないが、コマンド・ファイルや環境変数で -ol0 オプションまたは -ol1 オプションを指定する。-ol2 オプションは、オプティマイザのデフォルトの動作を復元します。	-ol2

3.2.2 最適化情報ファイルの作成方法（-on オプション）

-O3 オプションを指定してオプティマイザを起動する場合には、-on オプションを指定すると読み取り可能な最適化情報ファイルを作成できます。-on に続く番号は、レベル（0、1、2）を表します。作成されたファイルには .nfo 拡張子が付きます。表 3-3 を使用して -on オプションに指定する適切なレベルを選択してください。

表 3-3. -on オプションのレベルの選択方法

状況	使用するオプション
情報ファイルを必要としないが、コマンド・ファイルや環境変数で -on1 オプションまたは -on2 オプションを指定している。-on0 オプションは、オプティマイザのデフォルトの動作を復元します。	-on0
最適化情報ファイルを作成する。	-on1
詳細な最適化情報ファイルを作成する。	-on2

3.3 プログラムレベルの最適化の実行（-pm および -O3 オプション）

-pm オプションを -O3 オプションと組み合わせて使用すると、プログラムレベルの最適化を指定できます。プログラムレベルの最適化では、すべてのソース・ファイルが *モジュール*と呼ばれる 1 つの中間ファイルにコンパイルされます。このモジュールが、コンパイラの最適化パスとコード生成パスに移動します。コンパイラはプログラム全体を参照できるので、ファイルレベルの最適化ではほとんど適用されない次のような最適化を実行します。

- 関数内の特定の引数が常に同じ値をとる場合、コンパイラはその引数を値に置き換え、引数の代わりに値を渡します。
- 関数の戻り値が一度も使用されない場合、コンパイラはその関数内の戻りコードを削除します。
- 関数が main から直接または間接に呼び出されない場合、コンパイラはその関数を除去します。

コンパイラが適用しているプログラムレベルの最適化を知るには、-on2 オプションを使用して情報ファイルを作成します。詳細は、3.2.2 項「最適化情報ファイルの作成方法（-on オプション）」（3-5 ページ）を参照してください。

3.3.1 プログラムレベルの最適化の制御（-op*n* オプション）

-pm -O3 を指定して起動するプログラムレベルの最適化は、-op オプションを使用することにより制御できます。具体的には -op オプションは、他のモジュール内の関数があるモジュールの外部関数を呼び出せるか、またはあるモジュールの外部変数を変更できるかを指定します。-op に続く番号は、呼び出しや変更を許可しようとしているモジュールに設定するレベルを指定します。-O3 オプションは、この情報を独自のファイルレベル解析と組み合わせて、そのモジュールの外部関数と変数の定義を、静的に宣言された場合と同じ扱いにするかどうかを決定します。表 3-4 を使用して、-op オプションに指定する適切なレベルを選択してください。

表 3-4.　-op オプションのレベルの選択方法

モジュールの状況	使用するオプション
他のモジュールから呼び出される関数と、他のモジュール内で変更されるグローバル変数がある。	-op0
他のモジュールから呼び出される関数はないが、他のモジュール内で変更されるグローバル変数がある。	-op1
他のモジュールから呼び出される関数も、他のモジュール内で変更されるグローバル変数もない。	-op2
他のモジュールから呼び出される関数はあるが、他のモジュール内で変更されるグローバル変数がない。	-op3

特定の環境では、コンパイラは、指定された -op レベルとは異なるレベルに戻したり、プログラムレベルの最適化をすべて使用不可にしたりする場合があります。表 3-5 に、コンパイラが別の -op レベルに戻す原因となる条件と -op レベルの組み合わせを示します。

表 3-5.　-op オプションを使用する場合の特別な注意事項

-op の指定	条件	-op レベル
指定なし	-O3 の最適化レベルが指定された。	デフォルトの -op2 になる。
指定なし	コンパイラが -O3 最適化レベルにある外部関数の呼び出しを検出した。	-op0 に戻る
指定なし	main が定義されていない。	-op0 に戻る
-op1 または -op2	エントリ・ポイントとして定義されている main をもつ関数がなく、かつ FUNC_EXT_CALLED プラグマによって特定されている関数がない。	-op0 に戻る
-op1 または -op2	割り込み関数が定義されていない。	-op0 に戻る
-op1 または -op2	FUNC_EXT_CALLED プラグマによって特定されている関数がない。	-op0 に戻る
-op3	任意の条件	-op3 のまま残る

-pm と -O3 を指定した一部の状況では、必ず -op オプションか FUNC_EXT_CALLED プラグマを使用する*必要があります*。これらの状況については、3.3.2 項「C/C++ とアセンブリを組み合わせた場合の最適化に関する注意事項」（3-8 ページ）を参照してください。

3.3.2　C/C++ とアセンブリを組み合わせた場合の最適化に関する注意事項

プログラム内でアセンブリ関数が使用されている場合は、-pm オプションを指定するときに注意が必要です。コンパイラは C/C++ ソース・コードだけを認識し、アセンブリ・コードが指定されていても認識できません。コンパイラではアセンブリ・コードによる C/C++ 関数の呼び出しや C/C++ 関数に対する変数の変更が認識されないので、-pm オプションを指定するとこのような C/C++ 関数は最適化の対象外になります。それらの関数に最適化を実行するには、それらの関数の宣言や参照の前に FUNC_EXT_CALLED プラグマ（6.8.4 項「FUNC_EXT_CALLED プラグマ」（6-30 ページ）を参照）を配置します。

プログラムの中にアセンブリ関数が指定されている場合に使用できる別の方法は、-op オプションを -pm オプションおよび -O3 オプションと組み合わせて使用することです（3.3.1 項「プログラムレベルの最適化の制御（-opn オプション）」（3-6 ページ）を参照）。

一般に、FUNC_EXT_CALLED プラグマを -pm -O3 および -op1 または -op2 と組み合わせて適切に使用することにより、最良の結果を得ることができます。

アプリケーションに以下のいずれかの状況が当てはまる場合は、ここに示す解決策を使用してください。

状況　　アプリケーションは、アセンブリ関数を呼び出す C/C++ ソース・コードで構成されています。それらのアセンブリ関数は、C/C++ 関数の呼び出しも C/C++ 変数の変更も行いません。

解決策　　コンパイルするときに -pm -O3 -op2 を指定し、外部関数が C/C++ 関数の呼び出しも C/C++ 変数の変更も行わないことをコンパイラに指示します。-op2 オプションについては、3.3.1 項を参照してください。

-pm -O3 オプションだけを指定してコンパイルすると、コンパイラの最適化レベルがデフォルトの -op2 から -op0 に戻ります。コンパイラで -op0 を使用する理由は、C/C++ で定義されているアセンブリ言語関数の呼び出しにより、他の C/C++ 関数を呼び出したり C/C++ 変数を変更したりする可能性があることを想定しているからです。

状況　　アプリケーションは、アセンブリ関数を呼び出す C/C++ ソース・コードで構成されています。それらのアセンブリ言語関数は C/C++ 関数を呼び出しませんが、C/C++ 変数を変更します。

解決策　　次の 2 つの解決策を試して、ご使用のコードでうまく機能する方を選択してください。

- -pm -O3 -op1 を指定してコンパイルします。
- アセンブリ関数により変更される可能性がある変数に volatile キーワードを付加し、-pm -O3 -op2 を指定してコンパイルします（詳細については、6.7.2 項「volatile キーワード」（6-14 ページ）を参照してください）。

-op オプションについては、3.3.1 項を参照してください。

状況　アプリケーションは、C ソース・コードおよびアセンブリ・ソース・コードで構成されています。アセンブリ関数は、C 関数を呼び出す割り込みサービス・ルーチンです。アセンブリ関数から呼び出される C 関数が C から呼び出されることはありません。これらの C 関数は main のように動作します。それらの関数は C へのエントリ・ポイントとして機能します。

解決策　割り込みによって変更される可能性がある C/C++ 変数に volatile キーワードを付加します。その後、次のどちらかの方法でコードの最適化を実行します。

- 最良の最適化を達成するには、アセンブリ言語割り込みから呼び出されるすべてのエントリ・ポイント関数に FUNC_EXT_CALLED プラグマを適用してから、-pm -O3 -op2 を指定してコンパイルします。*必ずすべてのエントリ・ポイント関数とともにプラグマを使います。*そうしないと、コンパイラは先頭に FUNC_EXT_CALL プラグマを指定していないエントリ・ポイント関数を除去します。
- -pm -O3 -op3 を指定してコンパイルします。FUNC_EXT_CALL プラグマを指定しないので、-op3 オプションを使用しなければなりません。このオプションは -op2 ほど強力ではなく、最適化があまり効率的でない場合もあります。

追加オプションを指定せずに -pm -O3 を使用すると、アセンブリ関数から呼び出される C/C++ 関数が除去されることを忘れないでください。それらの関数を残しておくには、FUNC_EXT_CALLED プラグマを指定します。

FUNC_EXT_CALLED についての詳細は 6.8.4 項（6-30 ページ）を、-op オプションについては 3.3.1 項を参照してください。

3.4 オプティマイザを使用する場合の特別な注意事項

コンパイラは正確性を保持しながら、ANSI/ISO 準拠の C/C++ プログラムの性能を高めるために設計されています。しかし、最適化に向けたコードを記述する際に、意図したようにプログラムが動作することを保証するために、以下の節で説明されている特別の考慮事項があることに注意してください。

3.4.1 最適化コード内で asm 文を使用する場合の注意

最適化コード内で asm（インライン・アセンブリ）文を使用するときは、特に注意が必要です。コンパイラはコード・セグメントを再配置し、レジスタを自由に使用し、変数や式を完全に除去する場合があります。コンパイラによる最適化で asm 文が対象となることはありませんが（その文に到達しない場合を除いて）、アセンブリ・コードが挿入されている前後の環境が C/C++ ソース・コードと大きく異なってしまう場合があります。

通常、asm 文を使用して割り込みマスクや入出力（I/O）ポートなどのハードウェア制御を操作することは安全ですが、asm 文を使用して C/C++ 環境とのインターフェイスを取ったり C/C++ 変数にアクセスしたりしようとすると、予期しない結果を生じる恐れがあります。コンパイル後にアセンブリ出力をチェックし、asm 文が誤っていないかどうか、またプログラムの整合性が維持されているかどうかを確認してください。

3.4.2 必要なメモリ・アクセスを行うために volatile キーワードを使用する

コンパイラはデータ・フローを解析し、メモリ・アクセスを可能な限り回避します。C/C++ コードに書き込まれているとおりのメモリ・アクセスに*依存*しているコードがある場合は、必ず volatile キーワードを使用してそれらのアクセスを特定*しなければなりません*。コンパイラは、volatile 変数への参照を最適化により除去することはありません。

次の例では、あるロケーションで 0xFF が読み出されるまでループが繰り返されます。

```
unsigned int *ctrl;
while (*ctrl !=0xFF);
```

この例では、*ctrl はループ不変式です。そのため、ループは 1 回のメモリ読み取りにまで簡略化されます。これを訂正するには、ctrl を次のように定義します。

```
volatile unsigned int *ctrl;
```

3.4.2.1 エイリアスが設定された変数をアクセスする場合の注意

単一のオブジェクトに何通りかの方法でアクセスする場合（たとえば、2 つのポインタが同じオブジェクトを指す場合や、1 つのポインタが 1 つの名前付きオブジェクトを指す場合など）には、エイリアス指定が行われます。エイリアス指定は、オプティマイザの動作を中断する場合があります。これは、間接参照によって別のオブジェクトが参照されてしまうからです。コンパイラはコードを解析して、エイリアス指定が発生するかどうかを判断します。その上でオプティマイザは、プログラムの正確さを損なわないようにできるだけプログラムを最適化します。コンパイラにはプログラムを保護する働きがあります。

コンパイラは、ローカル変数のアドレスが関数に渡される場合に、その関数がポインタを通して書き込むことによってローカル変数の値を変更する可能性があることを想定していますが、復帰後にそのアドレスを他の場所で使用できるようにするわけではありません。たとえば、呼び出された関数はローカルのアドレスをグローバル変数に割り当てたり、そのアドレスを戻すことができません。この前提が無効の場合には、-ma コンパイラ・オプションを指定することにより、コンパイラにワーストケースのエイリアス設定を想定させることができます。ワーストケースのエイリアス設定では、間接参照（つまり、ポインタを使う）はこのような変数を参照できます。

3.4.2.2 -ma オプションを指定して次の技法が有効であることを示す

コンパイラはオプティマイザと一緒に起動したときに、アドレスが関数に引数として渡される任意の変数は、呼び出された関数内で設定されたエイリアスによってそれ以降修正されないことを想定します。たとえば次の例が含まれます。

- ❑ 関数からのアドレスの戻り
- ❑ グローバル変数へのアドレスの割り当て

このようなエイリアスをコードに使用する場合は、コードを最適化する際に -ma オプションを指定する必要があります。たとえば、ご使用のコードが次のものとほぼ同じ場合は -ma オプションを指定します。

```
int *glob_ptr;

g()
{
    int  x = 1;
    int *p = f(&x);

    *p        = 5;    /* p aliases x    */
    *glob_ptr = 10;   /* glob_ptr aliases x */

    h(x);
}

int *f(int *arg)
{
    glob_ptr = arg;
    return arg;
}
```

3.5 自動インライン展開（-oi オプション）

-O3 オプションを指定して最適化すると、コンパイラは自動的に小さい関数をインライン展開します。コマンド行オプション -oi*size* は、size しきい値を指定します。この *size* しきい値より大きい関数は、自動的にインライン展開されることはありません。-oi*size* オプションは次のように使用できます。

- ❑ *size* パラメータを 0 (-oi0) に設定した場合、自動インライン展開は抑止されます。
- ❑ *size* パラメータをゼロ以外の整数に設定した場合、コンパイラは、自動的にインライン展開する関数のサイズに対する上限として、この size しきい値を使用します。コンパイラは、関数をインライン展開する回数（関数が外部から参照できるが関数宣言を安全に除去できない場合は、それに 1 を加算する）と、関数のサイズを乗算します。

コンパイラは、算出された値が size パラメータより小さい場合にだけ関数をインライン展開します。コンパイラは関数のサイズを任意の単位で測定しますが、最適化情報ファイル（-on1 または -on2 オプションを指定して作成する）では各関数のサイズが -oi オプションと同じ単位で報告されます。

-oi*size* オプションは、inline として明示的に宣言されていない関数のインライン展開だけを制御します。-oi*size* オプションを指定しない場合でも、コンパイラは非常に小さい関数をインライン展開します。

注：　-O3 オプションによる最適化とインライン展開

自動インライン展開をオンにするには、-O3 オプションを指定する必要があります。-O3 オプションは他の最適化をオンにします。-O3 による最適化が必要ですが、自動インライン展開を必要としない場合は、-oi0 と -O3 オプションを指定します。

注：　インライン展開とコード・サイズ

関数をインライン展開するとコード・サイズが増大します。特に複数の場所で呼び出された関数をインライン展開する場合、増大します。関数のインライン展開は、少数の場所からのみ呼び出される関数および小さい関数に適しています。インライン展開によるコード・サイズの増大を防ぐには、-oi0 および -pi オプションを指定します。これらのオプションを指定すると、コンパイラは組み込み関数だけをインライン展開できます。コード・サイズがまだ大きすぎるような場合、3.8 節「コード・サイズ最適化の強化（-ms オプション）」（3-18 ページ）を参照してください。

3.6 インターリスト・ユーティリティをオプティマイザと組み合わせて使用する方法

最適化（-O*n* オプション）と -os および -ss オプションを指定してコンパイルすると、インターリスト・ユーティリティの出力を制御できます。

- -os オプションを使用すると、コンパイラのコメントがアセンブリ・ソース文に差し込まれます。
- -ss オプションと -os オプションを一緒に指定すると、コンパイラのコメントと元の C/C++ ソースがアセンブリ・コードに差し込まれます。

最適化の際に -os オプションを指定した場合、インターリスト機能は 1 つの独立したパスとして実行*されません*。その代わり、コンパイラはコード内にコメントを挿入し、そのコメントにはコンパイラがどのようにコードの再配置と最適化を実行したかが示されます。これらのコメントは、アセンブリ言語ファイルの中に ;** で始まるコメントとして出力されます。C/C++ ソース・コードは、-ss オプションと組み合わせて指定した場合以外は差し込まれません。

インターリスト機能は C/C++ の文の境界にまたがった一部の最適化を不可能にするので、最適化後のコードに影響を及ぼす場合があります。最適化は、通常のソースの差し込みを不可能にします。これは、コンパイラが広範囲にわたってプログラムの再配置を行うからです。したがって、-os オプションを指定した場合、コンパイラは再構築後の C/C++ の文を書き込みます。

例 3-1 に、最適化（-O2）と -os オプションを指定してコンパイルした関数を示します。アセンブリ・ファイルの中で、アセンブリ・コードにコンパイラのコメントが差し込まれています。

例 3-1. -O2 および -os オプションを指定してコンパイルされた C コード

(a) C ソース

```
int copy (char *str, const char *s, int n)
{
   int i;
   for (i = 0; i < n; i ++)
      *str++ = *s++;
}
```

例 3-1.　-O2 および -os オプションを指定してコンパイルした C コード（続き）

(b)　コンパイラ出力

```
;***************************************************************
;* FNAME: _copy                        FR SIZE:   0            *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                              0 Parameter,  0 Auto,  0 SOE     *
;***************************************************************

_copy:
;*** 6  -----------------------    if ( n <= 0 ) goto g4;
        CMPB      AL,#0                 ; |6|
        B         L2,LEQ                ; |6|
        ; branch occurs ; |6|
;***    -----------------------    #pragma MUST_ITERATE(1, 4294967295, 1)
:***    -----------------------    L$1 = n-1;
        ADDB      AL,#-1
        MOVZ      AR6,AL
L1:
;***    -----------------------g3:
;*** 7  -----------------------    *str++ = *s++;
;*** 7  -----------------------    if ( (--L$1) != (-1) ) goto g3;
        MOV       AL,*XAR5++            ; |7|
        MOV       *XAR4++,AL            ; |7|
        BANZ      L1,AR6--
        ; branch occurs ; |7|
;***    -----------------------g4:
;***    -----------------------    return;
L2:
        LRETR
        ; return occurs
```

-ss オプションおよび -os オプションを最適化と組み合わせて指定すると、コンパイラはコメントを差し込み、アセンブラの前にインターリスト機能が実行されることにより、元の C/C++ ソースがアセンブリ・ファイルにマージされます。

例 3-2 に、最適化（-O2）と -os オプションを指定してコンパイルした 例 3-1 を示します。アセンブリ・ファイルの中で、アセンブリ・コードにコンパイラのコメントと C ソースが差し込まれています。

注：　パフォーマンスとコード・サイズに与える影響

-ss オプションを指定すると、パフォーマンスとコード・サイズによくない影響を及ぼします。

例 3-2.　-O2、-os、および -ss オプションを指定してコンパイルされた C コード

```
;----------------------------------------------------------------------
;   2 | int copy (char *str, const char *s, int n)
;----------------------------------------------------------------------

;***************************************************************
;* FNAME: _copy                         FR SIZE:   0           *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                           0 Parameter,  0 Auto,  0 SOE      *
;***************************************************************

_copy:
;* AR4   assigned to _str
;* AR5   assigned to _s
;* AL    assigned to _n
;* AL    assigned to _n
;* AR5   assigned to _s
;* AR4   assigned to _str
;* AR6   assigned to L$1
;*** 6  -----------------------    if ( n <= 0 ) goto g4;
;----------------------------------------------------------------------
;   4 | int i;
;----------------------------------------------------------------------
;----------------------------------------------------------------------
;   6 | for (i = 0; i < n; i++)
;----------------------------------------------------------------------
        CMPB      AL,#0                 ; |6|
        B         L2,LEQ                ; |6|
        ; branch occurs ; |6|
;***    -----------------------    #pragma MUST_ITERATE(1, 4294967295, 1)
:***    -----------------------    L$1 = n-1;
        ADDB      AL,#-1
        MOVZ      AR6,AL
        NOP
L1:
;***    -----------------------g3:
;*** 7  -----------------------    *str++ = *s++;
;*** 7  -----------------------    if ( (--L$1) != (-1) ) goto g3;
;----------------------------------------------------------------------
;   7 | *str++ = *s++;
;----------------------------------------------------------------------
        MOV       AL,*XAR5++            ; |7|
        MOV       *XAR4++,AL            ; |7|
        BANZ      L1,AR6--
        ; branch occurs ; |7|
;***    -----------------------g4:
;***    -----------------------    return;
L2:
        LRETR
        ; return occurs
```

3.7 最適化されたコードのデバッグ方法

完全に最適化されたコードのデバッグは推奨できません。その理由は、コンパイラが大幅なコードの再配置を行うほか、多数の変数が多数のレジスタに割り振られるため、ソース・コードとオブジェクト・コードを相関させるのが難しくなるからです。--symdebug:dwarf（または -g）または --symdebug:coff オプション（STABS デバッグ）を指定して生成したコードをプロファイリングすることは推奨できません。これは、これらのオプションによりパフォーマンスが大幅に低下するからです。これらの問題に対処するためには、以下の節で説明しているオプションを使用してください。デバッグまたはプロファイル可能な方法でコードを最適化できます。

3.7.1 最適化されたコードのデバッグ（-g、--symdebug:dwarf、--symdebug:coff、および -O オプション）

最適化されたコードをデバッグするには、-O オプションをシンボリック・デバッグ・オプション（--symdebug:dwarf または --symdebug:coff）のどちらか 1 つと組み合わせて指定します。シンボリック・デバッグ・オプションは C/C++ ソースレベル・デバッガで使用する疑似命令を生成しますが、コンパイラによる多数の最適化が使用できなくなります。-O オプション（最適化を起動）を --symdebug:dwarf または --symdebug:coff オプションと組み合わせて指定すると、デバッグ時でも最適化の多くが実行できるようになります。

シンボリック・デバッグを使用してかつ最も最適化したコードを生成したい場合、-mn オプションを指定します。-mn オプションを指定すると、--symdebug:dwarf や --symdebug:coff によって無効となった最適化を有効にします。しかし、-mn オプションを指定すると、デバッガ機能の信頼性が低くなります。

注：　シンボリック・デバッグ・オプションを指定するとパフォーマンスとコード・サイズに影響を与える

--symdebug:dwarf または -symdebug:coff オプションを指定すると、パフォーマンスが低下し、コード・サイズが増大する可能性があります。これらのオプションを指定するのは、デバッグするときだけにしてください。プロファイルが推奨されない場合に --symdebug:dwarf または --symdebug:coff を指定してください。

3.7.2 最適化されたコードのプロファイル方法

最適化されたコードをプロファイルするには、デバッグを指定せず最適化（-O0 〜 -O3）を使用してください。シンボリック・デバッグを使わないと、最適化されたコードを関数の細粒度でプロファイルできます。

ブレーク・ポイント・ベースのプロファイラがある場合、--profile:breakpt オプションを -O オプションと組み合わせて指定します。--profile:breakpt オプションを指定すると、ブレーク・ポイント・ベースのプロファイラを使用する場合に誤った動作を引き起こす可能性のある最適化を無効にします。

パワー・プロファイラがある場合、--profile:power オプションを -O オプションと組み合わせて指定します。--profile:power オプションは、パワー・プロファイラの計測コードを作成します。

さらに細かく Code Composer Studio の関数レベルでコードをプロファイルする必要がある場合、--symdebug:dwarf または --symdebug:coff オプションを指定できますが、推奨はしません。コンパイラがデバッグですべての最適化を使用できるわけではないため、パフォーマンスが大幅に低下する場合があります。Code Composer Studio の外側で clock() 関数を使うことを推奨します。

注：　プロファイル・ポイント

Code Composer Studio でシンボリック・デバッグを使用しない場合、関数の先頭と最後で設定できるのはプロファイル・ポイントだけです。

3.8 コード・サイズ最適化の強化（-ms オプション）

-ms オプションを指定すると、コンパイラによるコード・サイズ優先の最適化のレベルが強化されます。このような最適化は、パフォーマンスを犠牲にして行われます。最適化には、共通のコード・ブロックを関数呼び出しで置き換える手続き抽象化が含まれます。たとえば、プロローグ・コードとエピローグ・コード、特定の組み込み関数、および他の共通コード・シーケンスは、ランタイム・ライブラリで定義されている関数の呼び出しで置き換えることができます。-ms オプション指定時に、提供されているランタイム・ライブラリとリンクすることが必要です。-ms オプションを起動するために最適化を使用する必要はありません。

-ms オプションがどのように機能するか示すために、プロローグ・コードとエピローグ・コードを置き換える方法を次に説明します。次のコードは、SOE レジスタの数、フレームのサイズ、およびフレーム・ポインタを使用するかどうかに依存する関数呼び出しに変更されます。-ms オプションを指定すると、このような関数は、それぞれのファイル内で次のように定義されます。

```
_prolog_c28x_1
_prolog_c28x_2
_prolog_c28x_3
_epilog_c28x_1
_epilog_c28x_2
```

例 3-3 に、-ms オプションを指定してコンパイルされた C コードの例とその結果出力されるアセンブリ・コードを示します。

例 3-3.　-ms オプションを指定してコンパイルされたコード

(a)　C ソース・コード

```
extern int x, y, *ptr;
extern int foo();
int main(int a, int b, int c)
{
ptr[50] = foo();
   y = ptr[50] + x + y + a +b + c;
}
```

(b)　コンパイラ出力

```
FP      .set    XAR2
        .global _prolog_c28x_1
        .global _prolog_c28x_2
        .global _prolog_c28x_3
        .global _epilog_c28x_1
        .global _epilog_c28x_2
        .sect   “.text”
        .global _main

;****************************************************************
;* FNAME: _main                         FR SIZE:   6            *
;*                                                              *
;* FUNCTION ENVIRONMENT                                         *
;*                                                              *
;* FUNCTION PROPERTIES                                          *
;*                              0 Parameter,  0 Auto,  6 SOE    *
;****************************************************************

_main:
        FFC       XAR7,_prolog_c28x_1
        MOVZ      AR3,AR4               ; |5|
        MOVZ      AR2,AH                ; |5|
        MOVZ      AR1,AL                ; |5|
        LCR       #_foo                 ; |6|
        ; call occurs [#_foo] ; |6|
        MOVW      DP,#_ptr
        MOVL      XAR6,@_ptr            ; |6|
        MOVB      XAR0,#50              ; |6|
        MOVW      DP,#_y
        MOV       *+XAR6[AR0],AL        ; |6|
        MOV       AH,@_y                ; |7|
        MOVW      DP,#_x
        ADD       AH,AL                 ; |7|
        ADD       AH,@_x                ; |7|
        ADD       AH,AR3                ; |7|
        ADD       AH,AR1                ; |7|
        ADD       AH,AR2                ; |7|
        MOVB      AL,#0
        MOVW      DP,#_y
        MOV       @_y,AH                ; |7|
        FFC       XAR7,_epilog_c28x_1
        LRETR
        ; return occurs
```

3.9 実行できる最適化の種類

TMS320C28x C/C++ コンパイラは、さまざまな最適化の技法を使用して C/C++ プログラムの実行速度を向上させ、サイズを縮小します。最適化は、コンパイラ全体を通じてさまざまなレベルで行われます。

ここで説明する最適化の大半は、-O コンパイラ・オプションを使って有効化および制御する独立したオプティマイザ・パスによって実行されます（3.1 節（3-2 ページ）を参照）。しかし、コード・ジェネレータは、選択的に有効化または無効化できない一部の最適化を実行します。

コンパイラによって行われる最適化の種類を次に示します。このような最適化を行うことにより、C/C++ コードの効率性が高くなります。

最適化	ページ
コストに基づいたレジスタ割り当て	3-21
エイリアスの明確化	3-22
データ・フローの最適化 ❑ 複写伝播 ❑ 共通部分式の除去 ❑ 冗長な代入の除去	3-22
式の簡略化	3-23
ランタイムサポート・ライブラリ関数のインライン展開	3-24
誘導変数の最適化と強度換算	3-26
ループ不変コードの移動	3-26
ループの循環	3-26
レジスタ変数	3-26
レジスタ・トラッキング / レジスタ・ターゲッティング	3-26

3.9.1 コストに基づいたレジスタ割り当て

コンパイラは最適化が有効になっている場合、ユーザ変数やコンパイラの一時値に、それらの型、使用状況、使用頻度に応じてレジスタを割り当てます。ループの中で使用されている変数は、他の変数より高い優先順位が割り当てられます。他の変数と重複して使用されない変数は同じレジスタに割り当てられる場合があります。

次の例では、誘導変数 i は除去されています。これにより、カウントするループを実装する RPT 命令を使用できます。強度換算は、配列参照を、自動インクリメントと組み合わせた効率的なポインタ参照にします。

例 3-4. 強度換算、誘導変数の除去、レジスタ変数

(a) C ソース

```
int a[10];
main ()
{
    int i;
    for (i = 0; i < 10; i++)
         a[i] = 0;
}
```

(b) コンパイラ出力

```
;***************************************************************
;* FNAME: _main                         FR SIZE:   0           *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                            0 Parameter,  0 Auto,  0 SOE     *
;***************************************************************

_main:
;***    -----------------------    #pragma MUST_ITERATE(10, 10, 10)
;***    -----------------------    U$4 = &a[0];
;***    -----------------------    L$1 = 9;
;***    -----------------------g2:
;*** 7  -----------------------    *U$4++ = 0;
;*** 7  -----------------------    if ( (--L$1) != (-1) ) goto g2;
;*** 7  -----------------------    return 0;|
        MOVL      XAR4,#_a
        MOVB      AL,#0
        RPT       #9
||      MOV       *XAR4++,#0
        LRETR
        ; return occurs
```

3.9.2 エイリアスの明確化

一般に、C/C++ プログラムでは多数のポインタ変数が使用されます。多くの場合、コンパイラは 2 つ以上の l 値（小文字の L : シンボル、ポインタ参照、または構造体参照）が同じメモリ位置を参照しているかどうかを判断できません。このメモリ位置のエイリアス指定により、コンパイラがレジスタ内に値を保持できなくなる場合が少なくありません。その理由は、時間が経過するとレジスタとメモリが同じ値を保持し続けているかどうかを保証できないからです。

エイリアスの明確化は、2 つのポインタ式が同じ位置を指す可能性がなくなる時期を判別する技法です。これを使用すると、コンパイラは自由にそれらの式を最適化できるようになります。

3.9.3 データ・フローの最適化

以下のデータ・フローの最適化では、効率的な式への置換、不要な代入の検出と除去、および計算済みの値を求める演算の排除をまとめて行います。最適化が有効になっているコンパイラは、これらのデータ・フローの最適化をローカル（基本ブロック内で）とグローバル（全関数に対して）の両面から行います。

❏ 複写伝播

変数への代入が終わると、コンパイラはその変数の参照を代入された値に置換します。この値は、別の変数、定数、または共通部分式の場合もあります。その結果、定数の畳み込みや共通部分式の除去、または変数全体の除去にも十分に活用できます。例 3-5（3-23 ページ）を参照してください。

❏ 共通部分式の除去

複数の式から同じ値が得られる場合、コンパイラは値を一度だけ計算し、その値を保存し、再利用します。

❏ 冗長な代入の除去

多くの場合、複写伝播と共通部分式の除去による最適化の結果、変数（それ以降、別の代入があるまで、または関数が終了するまで次の参照がない変数）への不要な代入が生じます。コンパイラは、このような不要な代入を除去します（例 3-5 を参照）。

3.9.4 式の簡略化

最適な計算ができるように、コンパイラは式を簡略化し、命令やレジスタをほとんど必要としない同等の書式に置換します。定数間の演算では単一の定数に畳み込まれます。たとえば、a = (b + 4) - (c + 1) は a = b - c +3 になります（例 3-5 を参照）。

例 3-5 では、a に代入された定数 3 は a を使用するすべての場所に複写伝播され、a は不要な変数となり、除去されます。j に 3 を乗算した積と j に 2 を乗算した積の和は、簡略化されて b = j * 5 となります。これは、共通部分式として認識されます。c や d への代入は不要で、その式と置き換えられます。

例 3-5. データ・フローの最適化と式の簡略化

(a) C ソース

```
int simplify (int j)
{
   int a = 3;
   int b = (j * a) + (j * 2);
   return b;
}
```

(b) コンパイラ出力

```
;***************************************************************
;* FNAME: _simplify                     FR SIZE:   0           *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                            0 Parameter,  0 Auto,  0 SOE     *
;***************************************************************

_simplify:
;*** 5  -----------------------    return j*5;
        MOV       T,AL                   ; |5|
        MPYB      ACC,T,#5               ; |5|
        LRETR
        ; return occurs
```

3.9.5　関数のインライン展開

コンパイラは、小さなランタイムサポート関数の呼び出しをインライン・コードに置換することにより、関数呼び出しに関連したオーバーヘッドを減らすと同時に他の最適化を適用できる機会を増やします（例 3-6 を参照)。

例 3-6 では、コンパイラは C 関数 toupper( ) に対応するコードを見つけ、この関数の呼び出しをコードに置換します。

例 3-6.　関数のインライン展開

(a)　C ソース

```
#include <ctype.h>

char *strupper (char *s)

{

   char *cp;

   for (cp = s; *cp; cp++)
      *cp = toupper (*cp)
   return s;
}
```

例 3-6. 関数のインライン展開（続き）

(b) コンパイラ出力

```
;***************************************************************
;* FNAME: _strupper                    FR SIZE:   0            *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                            0 Parameter,  0 Auto,  0 SOE     *
;***************************************************************

_strupper:
;*** 7  -----------------------    cp = s;
;*** 7  -----------------------    goto g4;
        MOVZ      AR6,AR4               ; |7|
        B         L3,UNC                ; |7|
        ; branch occurs ; |7|
l!:
;***    -----------------------g1:
;*** 144        -----------------------    ch = C$2;  // [0]
;*** 152       -----------------------   if ( (unsigned)ch-97u >25u ) goto g3;
        MOV       AH,AL                 ; |152|
        ADDB      AH,#-97
        CMPB      AH,#25                ; |152|
        B         L2,HI                 ; |152|
        ; branch occurs ; |152|
;*** 152        -----------------------    ch -= 32;  // [0]
        ADDB      AL,#-32
L2:
;***    -----------------------g3:
;*** 153        -----------------------    *cp++ = ch;  // [0]
        MOV       *XAR6++,AL            ; |153|
L3:
;***    -----------------------g4:
;*** 8  -----------------------    if ( C$2 = *cp ) goto g1;
        MOV       AL,*+XAR6[0]          ; |8|
        BF        L1,NEQ                ; |8|
        ; branch occurs ; |8|
;*** 9  -----------------------    return s;
        LRETR
        ; return occurs

;* Inlined function references:
;* [  0] toupper
```

3.9.6 誘導変数と強度換算

誘導変数とは、ループ内での値がループの実行回数に直接関係する変数のことです。ループでの配列のインデックスと制御変数は、多くの場合誘導変数です。

強度換算とは、誘導変数を含んでいる非効率的な式をより効率的な式に置換するプロセスのことです。たとえば、インデックスを使用して一連の配列要素を参照するコードは、ポインタを使用してその配列をインクリメントするコードに置換されます。

誘導変数の解析と強度換算を併用すると、多くの場合ループ制御変数へのすべての参照が除去され、ループ制御変数を除去することができます。

3.9.7 ループ不変コードの移動

この最適化では、ループ内で常に同じ値に算出する式が特定されます。その計算は、ループの前に移動され、ループ内の個々の式は事前に計算された値への参照に置き換えられます。

3.9.8 ループの循環

コンパイラはループの最後でループ条件式を計算し、ループ外への余分な分岐が発生しないようにします。多くの場合、最初のエントリでの条件式のチェックと分岐は最適化から除去されます。

3.9.9 レジスタ変数

コンパイラは、ローカル変数、パラメータ、および一時値を格納するためにレジスタを最大限に使用することができます。メモリ内の変数をアクセスするより、レジスタに格納されている変数をアクセスする方が効率的です。レジスタ変数はポインタで特に効率的です（例 3-4（3-21 ページ）を参照）。

3.9.10 レジスタ・トラッキング / レジスタ・ターゲッティング

レジスタの内容がすぐに使用される場合に、コンパイラはその内容を記録し、値をリロードしないようにします。(a.b) などの変数、定数、および構造体参照は、直線的コードから記録されます。また、レジスタ・ターゲッティングは、レジスタ変数に代入する場合、または関数から値を戻す場合など必要に応じて特定のレジスタへの式を直接計算します（例 3-7（3-27 ページ）を参照）。

例 3-7.　レジスタ・トラッキング / レジスタ・ターゲッティング

(a)　C ソース

```
int x, y;

main()
{
   x *= 3;
   y = x;
}
```

(b)　コンパイラ出力

```
;****************************************************************
;* FNAME: _main                         FR SIZE:   0            *
;*                                                              *
;* FUNCTION ENVIRONMENT                                         *
;*                                                              *
;* FUNCTION PROPERTIES                                          *
;*                            0 Parameter,  0 Auto,  0 SOE      *
;****************************************************************

_main:
;*** 5  -----------------------    x *= 3;
;*** 6  -----------------------    y = x;
;*** 6  -----------------------    return;
        MOVZ      DP,#_x                ; |5|
        MOV       T,@_x                 ; |5|
        MPYB      ACC,T,#3              ; |5|
        MOV       @_x,AL                ; |6|
        LRETR
        ; return occurs
```

3.9.11　テール結合

コード・サイズを最適化する場合、一部の関数についてはテール結合が非常に効率的です。テール結合により、同じ命令シーケンスで終了し、共通の宛先をもつ基本ブロックを検出します。このようなブロックが検出されると、同じ命令シーケンスがそれ自身のブロックになります。その後、これらの命令は一連のブロックから削除され、新しく作成されたブロックへの分岐に置き換えられます。そのため、セット内の各ブロックについて 1 つではなく、命令シーケンスのコピーが 1 つだけあります。

例 3-8 では、3 つのケースの最後で a への加算が 1 つのブロックに結合されています。また、1 番目と 2 番目のケースでは 3 を乗算して別のブロックに結合されています。これは結果的に、3 つの命令を換算したものになります。場合によっては、この最適化は余分な分岐を導入したことにより、実行スピードに悪影響を及ぼします。

例 3-8.　テール結合

(a)　C コード

```
int func(int a)
{
   if (a < 0)
   {       a = -a;
      a += f(a)*3;
   }
   else if (a == 0)
   {
      a = g(a);
      a += f(a)*3;
   }
   else
       a += f(a);
    return a;
}
```

例 3-8.　テール結合（続き）

(b)　TMS320C28x C/C++ コンパイラの出力

```
;***************************************************************
;* FNAME: _func                        FR SIZE:   2            *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                            0 Parameter,  0 Auto,  2 SOE     *
;***************************************************************

_func:
        MOVL      *SP++,XAR2
        CMPB      AL,#0                 ; |3|
        MOVZ      AR2,AL                ; |2|
        B         L2,LT                 ; |3|
        ; branch occurs ; |3|
        CMPB      AL,#0                 ; |8|
        BF        L1,NEQ                ; |8|
        ; branch occurs ; |8|
        LCR       #_g                   ; |10|
        ; call occurs [#_g] ; |10|
        B         L3,UNC                ; |12|
        ; branch occurs ; |12|
L1:
        LCR       #_f                   ; |14|
        ; call occurs [#_f] ; |14|
        MOV       AH,AR2                ; |14|
        ADD       AH,AL                 ; |14|
        MOVZ      AR2,AH                ; |14|
        B         L4,UNC                ; |14|
        ; branch occurs ; |14|
L2:
        NEG       AL                    ; |5|
L3:
        MOVZ      AR2,AL                : |5|
        LCR       #_f                   ; |6|
        ; call occurs [#_f] ; |6|
        MOV       T,AL                  ; |6|
        MPYB      P,T,#3                ; |6|
        MOV       AL,AR2                ; |6|
        ADD       AL,PL                 ; |6|
        MOVZ      AR2,AL                ; |6|
L4:
        MOV       AL,AR2
        MOVL      XAR2,*--SP            ; |16|
        LRETR
        ; return occurs
```

3.9.12 ゼロとの比較を除去する方法

ほとんどの ALU 命令はステータス・レジスタを変更できるので、明示的なゼロとの比較は不要になる場合があります。TMS320C28x C/C++ コンパイラは、直前の命令が変更され、ステータス・レジスタを適切にセットすることができる場合にゼロとの比較を除去します。

例 3-9 では、MOV 命令に続く明示的なゼロとの比較が除去され、MOV はステータス・レジスタ自体をセットします。

例 3-9. ゼロとの比較を削除する方法

(a) C コード

```
int func (int a, int b)
{
    a = b;

    return a < 0 ? a : 1;
}
```

(b) TMS320C28x C/C++ コンパイラの出力

```
;***************************************************************
;* FNAME: _func                        FR SIZE:   0            *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                           0 Parameter,  0 Auto,  0 SOE      *
;***************************************************************

_func:
;*** 4  -----------------------    return (b < 0) ? b : 1;
        MOV       AL,AH                ; |2|
        B         L1,LT                ; |4|
        ; branch occurs ; |4|
        MOVB      AL,#1                ; |4|
L1:
        LRETR
        ; return occurs
```

第 4 章

C/C++ コードのリンク方法

TMS320C28x C/C++ コンパイラとアセンブリ言語ツールを使用してユーザ・プログラムをリンクするには、次の 2 つの方法があります。

- 複数のモジュールを個別にコンパイルしておき、後でモジュール同士をリンクします。この方法は、ソース・ファイルが複数ある場合に特に便利です。
- コンパリとリンクを 1 ステップで実行します。この方法は、ソース・モジュールが 1 つだけの場合に便利です。

本章では、それぞれの方法によるリンカの起動方法について説明します。また、C/C++ コードのリンクに関する特別な要件、たとえばランタイムサポート・ライブラリのリンク方法、初期化タイプの指定方法、プログラムをメモリに割り当てる方法についても説明します。リンカの詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

項目 **ページ**

4.1	リンカの起動方法（-z オプション）	4-2
4.2	リンカ・オプション	4-5
4.3	リンク・プロセスの制御方法	4-7
4.4	C28x コードと C2XLP コードのリンク方法	4-13

4.1　リンカの起動方法（-z オプション）

ここでは、プログラムをコンパイルしアセンブルした後で、独立したステップとして、またはコンパイラの一部としてリンカを起動する方法を説明します。

4.1.1　独立したステップとしてリンカを起動する方法

C/C++ プログラムのリンクを独立したステップで実行する場合の一般的な構文を、次に示します。

cl2000 -v28 -z {**-c** | **-cr**} *filenames* [*-options*] [**-o** *name.out*] **-l** *library* [**lnk.cmd**]

cl2000 -v28 -z　リンカを起動するコマンドです。

-c | **-cr**　C/C++ 環境で定義している特別の規則を使用することをリンカに指示するオプションです。cl2000 -v28 -z を使用する場合は、-c または -cr を必ず指定しなければなりません。-c オプションを指定すると、実行時に変数の自動初期化を行います。-cr オプションを指定すると、ロード時に変数の自動初期化を行います。

filenames　オブジェクト・ファイル、リンカ・コマンド・ファイル、またはアーカイブ・ライブラリの名前を指定します。すべての入力ファイルのデフォルト拡張子は *obj* です。これ以外の拡張子を使用する場合は、明示的に指定しなければなりません。リンカは入力ファイルがオブジェクトであるか、それともリンカ・コマンドが含まれた ASCII ファイルであるかを判別できます。-o オプションを使用して出力ファイル名を指定した場合を除き、デフォルトの出力ファイル名は *a.out* です。

options　リンカによるオブジェクト・ファイルの処理方法に影響を及ぼすオプションです。リンカ・オプションは、コマンド行の -z オプションの後にしか指定できませんが、順番は問いません（オプションについては、4.2 節「リンカ・オプション」（4-5 ページ）を参照）。

-o *name.out*　-o オプションは、出力ファイルの名前を指定します。

-l *libraryname*　（小文字の L）C/C++ ランタイムサポート関数、および浮動小数点算術関数を含んだ適切なアーカイブ・ライブラリまたはまたはリンカ・コマンド・ファイルを識別します。C/C++ コードをリンクする場合、ランタイムサポート・ライブラリを使用する必要があります。コンパイラに付属するライブラリを使用しても、または独自のランタイムサポート・ライブラリを作成しても構いません。リンカ・コマンド・ファイルにランタイムサポート・ライブラリを指定する場合、このパラメータは不要です。-l オプションは、リンカにカレント・ディレクトリの外部を参照するように指示します。4.3.1 項「ランタイムサポート・ライブラリのリンク方法」（4-7 ページ）を参照してください。

lnk.cmd　リンカのオプション、ファイル名、疑似命令、コマンドを含みます。

ライブラリをリンカへの入力として指定した場合、リンカは未定義の参照を解決するライブラリ・メンバのみをインクルードし、リンクします。たとえば、prog1.obj、prog2.obj、および prog3.obj の各モジュールから構成されている C/C++ プログラムを実行可能なファイル名 prog.out とリンクするには次のコマンドを使用します。

```
cl2000 -v28 -z -c prog1 prog2 prog3 -o prog.out rts2800.lib
```

リンカは、デフォルトの割り当てアルゴリズムを使用してプログラムをメモリに割り当てます。リンカ・コマンド・ファイルの中で MEMORY と SECTIONS の疑似命令を使用すると、割り当て処理をカスタマイズできます。詳細については、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

4.1.2 コンパイル・ステップの一部としてリンカを起動する方法

C/C++ プログラムのリンクをコンパイル・ステップの一部として実行する場合の一般的な構文を、次に示します。

cl2000 -v28 [*-options*] *filenames* **-z** {**-c** | **-cr**} *filenames* [*-options*] [**-o** *name.out*] **-l** *library* [**lnk.cmd**]

cl2000 -v28	リンカを起動するコマンドです。
options	コンパイラによる入力ファイルの処理方法に影響を与えるオプションです。オプションは、表 2-1（2-6 ページ）に示します。
filenames	1 つまたは複数の C/C++ ソース・ファイル、アセンブリ言語ソース・ファイル、直線的なアセンブリ・ファイル、オブジェクト・ファイルのいずれかを指定します。
-z	リンカを起動する方法をコンパイラに指示するオプションです。
-c | **-cr**	C/C++ 環境で定義している特別の規則を使用することをリンカに指示するオプションです。cl2000 -v28 -z を使用する場合は、-c または -cr を必ず指定しなければなりません。-c オプションを指定すると、実行時に変数の自動初期化を行います。-cr オプションを指定すると、ロード時に変数の自動初期化を行います。
filenames	オブジェクト・ファイル、リンカ・コマンド・ファイル、またはアーカイブ・ライブラリの名前を指定します。すべての入力ファイルのデフォルト拡張子は *obj* です。これ以外の拡張子を使用する場合は、明示的に指定しなければなりません。リンカは入力ファイルがオブジェクトであるか、それともリンカ・コマンドが含まれた ASCII ファイルであるかを判別できます。-o オプションを使用して出力ファイル名を指定した場合を除き、デフォルトの出力ファイル名は *a.out* です。
options	リンカによるオブジェクト・ファイルの処理方法に影響を及ぼすオプションです。リンカ・オプションは、コマンド行の -z オプションの後にしか指定できませんが、順番は問いません （オプションについては、4.2 節「リンカ・オプション」（4-5 ページ）を参照）。

-o *name.out*　-o オプションは、出力ファイルの名前を指定します。

-l *libraryname*　（小文字の L）C/C++ ランタイムサポート関数、および浮動小数点算術関数を含んだ適切なアーカイブ・ライブラリまたはまたはリンカ・コマンド・ファイルを識別します C/C++ コードをリンクする場合、ランタイムサポート・ライブラリを使用する必要があります。コンパイラに添付されているライブラリを使用しても、または独自のランタイムサポート・ライブラリを作成しても構いません。リンカ・コマンド・ファイルにランタイムサポート・ライブラリを指定する場合、このパラメータは不要です。

lnk.cmd　リンカのオプション、ファイル名、疑似命令、コマンドを含みます。

-z オプションは、コマンド行をコンパイラ・オプション（-z の前にあるオプション）とリンカ・オプション（-z の後にあるオプション）に分割します。-z オプションは、コマンド行ですべてのファイル・オプションおよびコンパイラ・オプションを指定した後に入力する必要があります。

コマンド行で -z の後に指定した引数は、すべてリンカへ渡されます。指定できる引数は、リンカ・コマンド・ファイル、追加オブジェクト・ファイル、リンカ・オプション、またはライブラリのいずれかです。

コマンド行で -z の前に指定した引数は、すべてコンパイラ引数です。指定できる引数は、C/C++ ソース・ファイル、アセンブリ・ファイル、コンパイラ・オプションのいずれかです。これらの引数については、2.2 節「C/C++ コンパイラの起動方法」を参照してください。

たとえば、prog1.c、prog2.c、および prog3.c の各モジュールから構成されている C/C++ プログラムをコンパイルし、実行可能なファイル名 prog.out とリンクするには次のコマンドを使います。

```
cl2000 -v28 prog1.c prog2.c prog3.c -z -c -o prog.out -l rts2000.lib
```

4.1.3　リンカを無効にする方法（-cコンパイラ・オプション）

-c コンパイラ・オプションを使用することにより、-z オプションを無効にできます。-c オプションは、C_OPTION 環境変数の中で -z オプションを指定しているとき、およびコマンド行でリンクを選択的に無効にする場合に特に便利です。

-c リンカ・オプションは、-c コンパイラ・オプションとは異なる独自の機能を備えています。デフォルトでは、-z オプションを使用した場合、コンパイラは -c リンカ・オプションを使用します。このオプションはリンカに対して、C/C++ のリンク規則（実行時の変数の自動初期化）を使用するように指示します。ロード時に変数を初期化する場合には、-z オプションの後に -cr リンカ・オプションを指定します。

4.2 リンカ・オプション

-z オプションの後に指定したすべてのコマンド行は、パラメータおよびオプションとしてリンカに渡されます。リンカを制御するオプションと、その影響についての説明を次に示します。

-a	絶対的な実行可能モジュールを生成します。これはデフォルトです。-a と -r をどちらも指定しなかった場合、リンカは -a が指定された場合と同じ動作をします。
-abs	絶対リスト・ファイルを作成します。
-ar	再配置可能な実行可能オブジェクト・モジュールを生成します。
-b	シンボリック・デバッグ情報のマージを無効にします。
-c	実行時に変数を自動初期化します。
-cr	ロード時に変数を自動初期化します。
-e *global_symbol*	出力モジュールの 1 次エントリ・ポイントを指定する *global_symbol* を定義します。
-f *fill_value*	出力セクション内の null 領域用のデフォルト埋め込み値を設定します。*fill_value* は 16 ビットの定数です。
-g *global_symbol*	グローバル・シンボルが -h リンカ・オプションにより静的シンボルにされていても、*global_symbol* をグローバル・シンボルとして定義します。
-h	すべてのグローバル・シンボルを静的シンボルにします。
-heap *size*	ヒープ・サイズ（動的メモリ割り当て）を *size* で指定したワード数に設定し、ヒープ・サイズを指定するグローバル・シンボルを定義します。デフォルトは、0x400 ワードです。
-farheap *size*	far ヒープ・サイズ（far 動的メモリ割り当て）を *size* で指定したワード数に設定し、far ヒープ・サイズを指定するグローバル・シンボルを定義します。デフォルトは、0x400 ワードです。
-I *directory*	ライブラリ検索アルゴリズムを変更し、デフォルト位置を検索する前に *directory* で指定したディレクトリを検索します。このオプションは -l リンカ・オプションの前に指定しなければなりません。ディレクトリ名は、オペレーティング・システムの規則に従ったものでなければなりません。-I オプションは 128 まで指定できます。
-l *filename*	（小文字 L）リンカがファイル名を検索する場所を制御します。

-m *filename* null 領域を含む入出力セクションのマップまたはリストを生成し、*filename* で指定したファイルにそのリストを格納します。ファイル名は、オペレーティング・システムの規則に従ったものでなければなりません。

-n メモリ疑似命令のすべての埋め込み指定を無視します。プロジェクトの開発段階でこのオプションを指定すると、メモリ疑似命令の埋め込み指定を使用した結果に起因する、サイズの大きい *.out* ファイルを生成しません。

-o *filename* 実行可能な出力モジュール名を指定します。*filename* は、オペレーティング・システムの規則に従ったものでなければなりません。-o オプションを指定しなかった場合、ファイル名はデフォルトの a.out になります。

-priority ライブラリの代替検索メカニズムを提供します。-priority を使うと、該当するシンボルの定義を含むコマンド行の最初のライブラリで、未解決の参照を解決できます。

-r 再配置エントリを出力モジュールの中に保持します。

-s 出力モジュールからシンボル・テーブル情報と行番号エントリを除去します。

-stack *size* C/C++ のシステム・スタック・サイズを *size* で指定したワード数に設定し、スタック・サイズを指定するグローバル・シンボルを定義します。デフォルトは、1K ワードです。

-u *symbol* 未解決の外部シンボル *symbol* を出力モジュールのシンボル・テーブルに配置します。

-w 未定義の出力セクションが作成された場合にメッセージを表示します。

-x ライブラリの再読み取りを強制実行します。リンカは参照がすべて解決されるまで、ライブラリの再読み取りを続行します。

リンカ・オプションの詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』の「リンカの説明」の章を参照してください。

4.3 リンク・プロセスの制御方法

C/C++ プログラムをリンクするときには、リンカの起動方法に関係なく次の特別な要件があります。作業すべき事項は次のとおりです。

❏ コンパイラのランタイムサポート・ライブラリを組み込む。
❏ 初期化のタイプを指定する。
❏ プログラムをメモリに割り当てる方法を決定する。

ここでは、上記の要件を制御する方法を説明し、リンカ・コマンド・ファイルの例を示します。

リンカの操作方法の詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』の「リンカの説明」の章を参照してください。

4.3.1 ランタイムサポート・ライブラリのリンク方法

ブートストラップ・ルーチン（*boot.obj* オブジェクト・モジュール）と呼ばれるプログラムを初期化および実行するコードと、すべての C/C++ プログラムをリンクする必要があります。このランタイム・ライブラリには、標準の C/C++ 関数、および本コンパイラが C/C++ 環境を管理するために使用する関数も組み込まれています。使用する C28x ランタイムサポート・ライブラリを指定するには、-l リンカ・オプションを指定する必要があります。また、-l オプションはリンカに対して、アーカイブ・パスやオブジェクト・ファイルを検索する際に –I オプションに注目し、次に C_DIR または C2000_C_DIR 環境変数に注目するように指示します。-l リンカ・オプションを指定するには、次のようにコマンド行に入力します。

cl2000 -v28 -z {**-c** | **-cr**} *filenames* **-l** *libraryname*

一般に、ライブラリはコマンド行の最後のファイル名として指定する必要があります。リンカは、コマンド行でファイルが指定された順に未解決の参照をライブラリ内で検索するからです。ライブラリの後にオブジェクト・ファイルがある場合は、それらのオブジェクト・ファイルからライブラリへの参照は解決されません。-x リンカ・オプションを指定すると、参照が解決されるまですべてのライブラリの再読み取りが強制的に行われます。ライブラリをリンカへの入力として指定すると、リンカは未定義の参照を解決するライブラリ・メンバのみを組み込み、リンクします。

デフォルトでは、あるライブラリが未解決の参照を導入していて、複数のライブラリがその定義を備えている場合、未解決の参照を導入した同じライブラリから定義が使われます。該当の定義を含むコマンド行の最初のライブラリから定義をリンカに使わせる場合、-priority オプションを指定してください。

4.3.2 実行時の初期化

ブートストラップ・ルーチンと呼ばれるプログラムを初期化および実行するためのコードを、すべての C/C++ プログラムとリンクする必要があります。ブートストラップ・ルーチンは、以下のタスクを実行します。

1) ステータス・レジスタおよびコンフィギュレーション・レジスタを設定します。
2) スタックおよび 2 次システム・スタックを設定します。
3) .cinit ランタイム初期化テーブルを処理し、グローバル変数を自動的に初期化します（-c オプションを指定している場合）。
4) すべてのグローバル・オブジェクト・コンストラクタを呼び出します（.pinit）。
5) main を呼び出します。
6) main が戻ったときに exit を呼び出します。

rts2800.lib の boot.obj にあるブートストラップ・ルーチンの例は、_c_int00 です。*エントリ・ポイント*は、通常ブートストラップ・ルーチンの開始アドレスに設定されます。

ライブラリに含まれている追加のランタイムサポート関数については、第 8 章「ランタイムサポート関数」で説明しています。これらの関数には、ANSI/ISO C 標準ランタイムサポートが含まれています。

注：　_c_int00 シンボル

-c または -cr リンカ・オプションを指定する場合、_c_int00 はプログラムのエントリ・ポイントとして自動的に定義されます。

4.3.3 割り込みベクタによる初期化

プログラムがリセットから実行を開始する場合、_c_int00 への分岐にリセット・ベクタを設定する必要があります。これにより、boot.obj はライブラリからロードされ、プログラムは正しく初期化されます。割り込みベクトルの例は、rts2800.lib の vectors.obj にあります。C28x の場合、ベクトルの最初の数行は次のとおりです。

```
          .def      _Reset
          .red      _c_int00
_Reset:   .ivec     _c_int00, USE_RETA"
```

4.3.4　グローバル・オブジェクト・コンストラクタ

コンストラクタとデストラクタを備えたグローバル C++ 変数は、コンストラクタをプログラムの初期化時に呼び出し、デストラクタをプログラムの終了処理中に呼び出す必要があります。C/C++ コンパイラは、スタートアップ時に呼び出されるコンストラクタのテーブルを作成します。

このテーブルは .pinit という名前付きセクションに含まれています。コンストラクタは、テーブル内に配置されている順に起動されます。

グローバル・コンストラクタは、他のグローバル変数の初期化後、または main() が呼び出される前に呼び出されます。グローバル・デストラクタは、atexit() を通じて登録された関数と同様に、exit() の間に起動されます。

7.8.4 項「初期化テーブル」（7-42 ページ）では、.pinit テーブルのフォーマットについて説明しています。

4.3.5　初期化のタイプの指定方法

C/C++ コンパイラは、グローバル変数を自動的に初期化するためにデータ・テーブルを作成します。7.8.4 項「初期化テーブル」（7-42 ページ）では、これらのテーブルのフォーマットについて説明しています。これらのテーブルは *.cinit* という名前付きセクションに含まれています。初期化テーブルは、次のどちらかの方法で使用されます。

- 実行時に変数を自動初期化します。グローバル変数は*実行時*に初期化されます。-c リンカ・オプションを指定してください（7.8.5 項「実行時の変数の自動初期化」（7-45 ページ）を参照）。

- ロード時に変数を自動初期化します。グローバル変数は*ロード時*に初期化されます。-cr リンカ・オプションを指定してください（7.8.6 項「ロード時の変数の自動初期化」（7-46 ページ）を参照）。

C/C++ プログラムをリンクするときには、-c と -cr のどちらかのオプションを指定する必要があります。これらのオプションはリンカに対して、自動初期化を実行時に行うかロード時に行うかを選択するように指示します。プログラムをコンパイルしリンクする場合は、-cリンカ・オプションがデフォルトになります。-c リンカ・オプションを指定する場合、-z オプションの後に指定する必要があります（4.1 節「リンカの起動方法（-z オプション）」（4-2 ページ）を参照）。次のリストは、-c または -cr で使用されるリンク規則をまとめたものです。

- _c_int00 シンボルは、プログラムのエントリ・ポイントとして定義されています。このシンボルは、boot.obj 内の C/C++ ブート・ルーチンの先頭を示します。-c または -cr を指定すると _c_int00 が自動的に参照されます。その結果、boot.obj がランタイムサポート・ライブラリから自動的にリンクされます。

- *.cinit* 出力セクションには終了レコードが埋め込まれているので、ローダ（ロード時の初期化）やブート・ルーチン（実行時の初期化）が、初期化テーブルの読み取りを停止する時期を認識できます。

❏ ロード時に自動初期化を行うと（-cr リンカ・オプション）、次のことが行われます。

  ■ リンカは *cinit* シンボルを -1 に設定します。これは初期化テーブルがメモリ内に存在しないことを示します。したがって実行時に初期化は行われません。

  ■ STYP_COPY フラグが .cinit セクション・ヘッダ内に設定されます。STYP_COPY は、ローダに対して自動初期化を直接実行し、.cinit セクションをメモリにロードしないように指示する特別な属性です。リンカは、メモリ内に .cinit セクション用のスペースを割り当てません。

❏ 実行時に自動初期化を行うと（-c リンカ・オプション）、リンカは、.cinit セクションの開始アドレスとしてシンボル *cinit* を定義します。ブート・ルーチンは、このシンボルを自動初期化用の開始点として使用します。

4.3.6 セクションをメモリ内のどこに配置するかを指定する方法

コンパイラは、コードとデータが入った再配置可能ブロックを作成します。これらのブロックは*セクション*と呼ばれ、さまざまなシステム構成に整合するように、さまざまな方法でメモリ内に割り当てられます。

コンパイラが作成するセクションには、初期化されたセクションと初期化されないセクションという 2 種類の基本セクションがあります。表 4-1 に、コンパイラが作成したセクション、およびリンクの目的でそのセクションに課せられているメモリ上の制約事項を示します。

注： far リンクの問題

C で far と宣言されているオブジェクト、またはラージ・メモリ・モデルのオブジェクトは .ebss/.econst セクションに配置されます。これは .bss/.cons セクションに配置される near オブジェクトとは別のものです。

表 4-1. コンパイラが作成するセクション

(a) 初期化されたセクション

名前	内容	制約事項
.cinit	明示的に初期化されるグローバル変数と静的変数のテーブル	プログラム
.const	明示的に初期化されるグローバル変数と静的定数変数および文字列リテラル	下位 64K データ
.econst	far 定数変数	データ内の任意の場所
.pinit	グローバル・オブジェクト・コンストラクタのテーブル	プログラム
.switch	switch 文を実装するためのテーブル	プログラム（-mt オプションを指定した場合） データ（-mt オプションを指定しない場合）
.text	実行可能なコードおよび定数	プログラム

表 4-1. コンパイラが作成するセクション（続き）

(b) 初期化されないセクション

名前	内容	制約事項
.bss	グローバル変数および静的変数	下位 64K データ
.ebss	far グローバル変数 / 静的変数	データ内の任意の場所
.stack	スタック空間	下位 64K データ
.sysmem	malloc 関数用のメモリ	下位 64K データ
.esysmem	far_malloc 関数用のメモリ	データ内の任意の場所

C/C++ ランタイム環境は、far_malloc ルーチンを提供することで、システム・ヒープ（.esysmem セクション）を far メモリへ配置できます。malloc ルーチンおよび far_malloc ルーチンの詳細は、8.5 節「ランタイムサポート関数およびマクロのまとめ」（8-31 ページ）を参照してください。

プログラムをリンクする場合、セクションをメモリ内のどこに割り当てるかを指定する必要があります。一般に、初期化されたセクションは ROM または RAM 内にリンクされ、初期化されないセクションは RAM 内にリンクされます。コンパイラが作成した初期化されたセクションと初期化されないセクションは、.text を除き、内部プログラム・メモリに割り当てることはできません。コンパイラがこれらのセクションをどのように使用するかについては、7.1.1 項「セクション」（7-3 ページ）を参照してください。

リンカは、セクションを割り当てるための MEMORY 疑似命令と SECTIONS 疑似命令を備えています。メモリへのセクション割り当て方法の詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』の「リンカの説明」の章を参照してください。

4.3.7 リンカ・コマンド・ファイルの例

例 4-1 に、C/C++ プログラムをリンクする典型的なリンカ・コマンド・ファイルを示します。この例のコマンド・ファイルは lnk.cmd という名前です。この例では、3 つのオブジェクト（a.obj、b.obj、c.obj）をリンクし、プログラム（prog.out）とマップ・ファイル（prog.map）を作成しています。

プログラムをリンクするには、次のように入力します。

```
cl2000 -v28 -z lnk.cmd
```

使用するシステムで機能させるには、MEMORY 疑似命令と SECTIONS 疑似命令に修正を加える必要があります。これらの疑似命令については、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

例 4-1.　リンカ・コマンド・ファイル

```
a.obj b.obj c.obj            /* Input filenames       */
-o prog.out                  /* Options               */
-m prog.map
-1 rts2800.lib               /* Get run-time support  */

MEMORY                       /* MEMORY directive      */
{
  RAM:  origin = 100h      length = 0100h
  ROM:  origin = 01000h    length = 0100h
}

SECTIONS                     /* SECTIONS directive    */
{
  .text:   > ROM
  .data:   > ROM
  .bss:    > RAM
  .pinit:  > ROM
  .cinit:  > ROM
  .switch: > ROM
  .const:  > RAM
  .stack:  > RAM
  .sysmem: > RAM
}
```

4.4 C28x コードと C2XLP コードのリンク方法

64 ワードのページ・サイズ（C28x）と 128 ワードのページ・サイズ（C2XLP）を備えたコードをリンクしないようにするために、C28x リンカで発生するエラーは警告に変更されています。C28x C/C++ コードから C2XLP アセンブリ関数を呼び出すことは可能です。考えられる方法の 1 つは、C2XLP 関数の呼び出しを、引数や C2XLP コード用の呼び出しスタックを正確に設定する veneer 関数と置き換えることです。たとえば、5 つの整数型の引数を受け取る C2XLP 関数の呼び出しを行うには、C28x コードを次のように変更します。

```
extern void foo_veneer(int, int, int, int, int);
void bar()
{
   /* replace the C2XLP call with a veneer call */
   /* foo(1, 2, 3, 4, 5); */
   foo_veneer(1, 2, 3, 4, 5);
}
```

例 4-2 に、veneer 関数がどうなるかを示します。

例 4-2.　C2xx コードと C2XLP コードをリンクする veneer 関数

```
   .sect ".text"
   .global _foo_veneer
   .global _foo
_foo_veneer:
   ;save registers
   PUSH AR1:AR0
   PUSH AR3:AR2
   PUSH AR5:AR4

   ;set the size of the C2XLP frame (including args size)
   ADDB SP,#10

   ;push args onto the C2XLP frame
   MOV *-SP[10],AL ;copy arg 1
   MOV *-SP[9],AH  ;copy arg 2
   MOV *-SP[8],AR4 ;copy arg 3
   MOV *-SP[7],AR5 ;copy arg 4
   MOV AL,*-SP[19]
   MOV *-SP[6],AL  ;copy arg 5

   ;save the return address
   MOV *-SP[5],#_label

   ;set AR1,ARP
   MOV AL,SP
   SUBB AL,#3
   MOV AR1,AL
   NOP *ARP1
```

例 4-2.　C2xx コードと C2XLP コードをリンクする veneer 関数（続き）

```
   ;jump to C2XLP function
   LB _foo
_label:

   ;restore register
   POP AR5:AR4
   POP AR3:AR2
   POP AR1:AR0
   LRETR
```

veneer 関数のフレームは、すべての C2XLP 呼び出しに対するフレームとして動作するので、最初の C2XLP 関数によって行われる後続の呼び出しに対しても、十分なサイズをフレームに追加する必要があります。

グローバル変数は、C28x C/C++ コードの .bss セクションで置き換えられます。C2XLP の .bss セクションは C28x コードとリンクされた場合、128 ワード境界上で開始することが保証されません。この問題を回避するには、新しいセクションを定義し、C2XLP グローバル変数を新しいセクションに変更し、リンカ・コマンド・ファイルを更新して、この新しいセクションが 128 ワード境界から確実に始まるようにします。

第 5 章

ポストリンク・オプティマイザ

TMS320C28x ポストリンク・オプティマイザは、不要なアセンブリ言語命令を取り除いたり変更したりして、効率のよいコードを生成します。ポストリンク・オプティマイザは、リンクすることで決定されるシンボルの最終アドレスを調べ、この情報を使ってコードに変更を加えます。

ポストリンク最適化を行うには、コンパイラ・オプション -plink が必要です。-plink コンパイラ・オプションは、絶対リスタ（abs2000）の実行およびアセンブラやリンカの再実行を含むツールの追加されたパスを起動します。-z コンパイラ・オプションの後に -plink オプションを指定する必要があります。

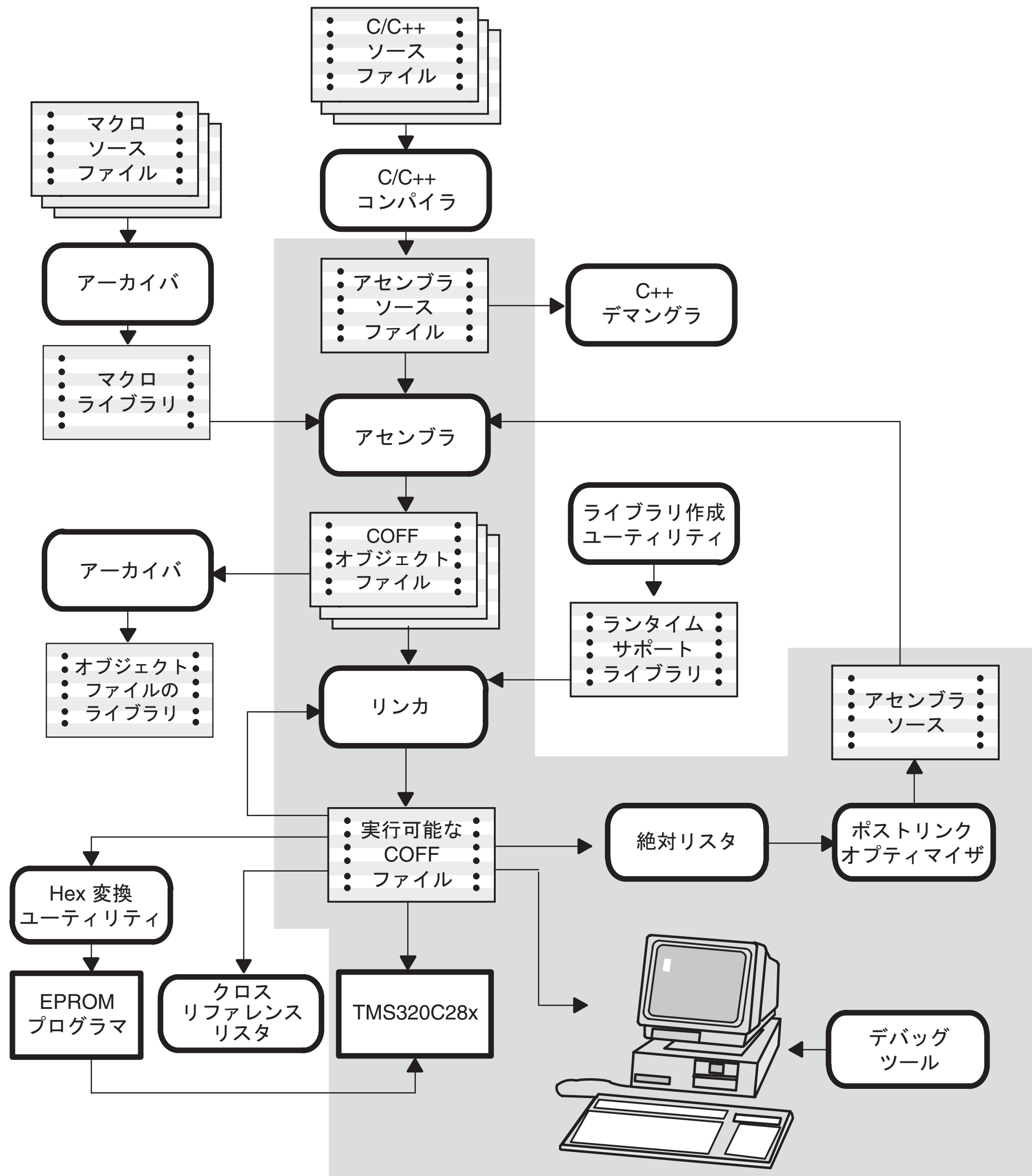
項目 **ページ**

5.1	**ソフトウェア開発フローにおけるポストリンク・オプティマイザの役割**	**5-2**
5.2	**冗長な DP ロード命令を除去する方法**	**5-4**
5.3	**分岐間で DP 値を調べる方法**	**5-4**
5.4	**関数呼び出し間で DP 値を調べる方法**	**5-5**
5.5	**他のポストリンク最適化**	**5-6**
5.6	**ポストリンク最適化を制御する方法**	**5-7**
5.7	**ポストリンク・オプティマイザ使用時の制約事項**	**5-9**
5.8	**出力ファイルの指定方法（-o オプション）**	**5-10**

5.1 ソフトウェア開発フローにおけるポストリンク・オプティマイザの役割

ポストリンク・オプティマイザは、通常の開発フローの一部ではありません。図 5-1 に、ポストリンク・オプティマイザを含むフローを示します。このフローのように行われるのは、-plink オプションを指定してコンパイラを起動した場合だけです。このフローを以下に示します。

図 5-1. TMS320C28x ソフトウェア開発フローにおけるポストリンク・オプティマイザ



フローが示すように、絶対リスタ（abs2000）もポストリンク最適化プロセスの一部になっています。絶対リスタは、グローバルに定義されているすべてのシンボルの絶対アドレスと COFF セクションを出力します。ポストリンク・オプティマイザは、入力として .abs ファイルを受け取り、これらのアドレスを使って最適化を行います。出力された .pl ファイルは 元の .asm ファイルを最適化したものです。その後、フローはアセンブラやリンカに戻り、最終出力ファイルが作成されます。

前述のフローがサポートされるのは、-plink オプションを指定してコンパイラ（cl2000 -v28）を起動した場合だけです。バッチ・ファイルを使って、各ツールを個別に起動する場合、フローを調整してコンパイラを使わなければなりません。また、-plink オプション指定時には -o オプションを使用して、出力ファイル名を指定する必要があります。詳細は、5.8 節（5-10 ページ）を参照してください。

たとえば、次のような複数の行

```
cl2000 -v28 file1.asm file1.obj
cl2000 -v28 file2.asm file2.obj
cl2000 -v28 -z file1.obj file2.obj lnk.cmd -o prog.out
```

を次の 1 行に置き換えます。

```
cl2000 -v28 file1.asm file2.asm -z lnk.cmd -o prog.out
-plink
```

5.2 冗長な DP ロード命令を除去する方法

ポストリンク・オプティマイザは冗長な DP ロード命令を除去することにより、DP レジスタ管理上の問題を軽減します。この作業を行うにために、DP の現行値を調べ、「MOV DP, #address」命令の address が DP が現在指し示すものと同じ 64 ワード・ページ上にあるかどうか判別します。現行の DP 値を使って address をアクセスできる場合には、命令は冗長なので、除去することができます。たとえば、次のコード・セグメントを考えてみましょう。

```
MOVZ       DP,#name1
ADD        @name1,#10
MOVZ       DP,#name2
ADD        @name2,#10
```

name1 と name2 が同一ページにリンクされている場合、ポストリンク・オプティマイザは name2 のアドレスを DP にロードする必要がないと判断し、その冗長なロード命令をコメントアウトします。

```
MOVZ       DP,#name1
ADD        @name1,#10
; <<REDUNDANT>>            MOVZ       DP,#name2
ADD        @name2,#10
```

この最適化は、C ファイルにも同様に適用することができます。コンパイラがあるモジュール内で定義されているすべてのグローバル変数参照のために DP を管理していても、外部で定義されているグローバル変数への任意の参照に対して DP ロード命令を引き続き発行します。ポストリンク・オプティマイザを使うと、このような場合に DP ロード命令の数を削減することができます。

5.3 分岐間で DP 値を調べる方法

分岐間で DP 値を調べるために、ポストリンク・オプティマイザは間接呼び出しや分岐がないこと、および考えられるすべての分岐の宛先はラベルを備えていることを必要とします。間接分岐や間接呼び出しがあると、ポストリンク・オプティマイザは基本ブロック内で DP 値のみを調べます。分岐の宛先にラベルがないと、ポストリンク・オプティマイザからの出力が不正確なものになる可能性があります。

ポストリンク・オプティマイザが間接呼び出しや間接分岐を見つけると、次のような警告を発行します。

```
NO POST LINK OPTIMIZATION DONE ACROSS BRANCHES
Branch/Call must have labeled destination
```

この警告が発行されるので、手書きのアセンブリ・ファイルの場合、間接呼び出しや間接分岐を直接呼び出しや直接分岐に書き換えてポストリンクから最大レベルの最適化を引き出すことができます。

5.4 関数呼び出し間で DP 値を調べる方法

同じファイル範囲内で関数が定義されている場合には、ポストリンク・オプティマイザは関数呼び出し後の DP ロード命令を最適化します。たとえば、ポストリンク・オプティマイザの実行後の次のコードを考えてみましょう。

```
_main:    LCR    #_foo
    MOVB    AL, #0
;<<REDUNDANT>>    MOVZ    DP,#_g2
    MOV    @_g2, #20
    LRETR

    .global    _foo
_foo:
    MOVZ    DP, #g1
    MOV      @_g1, #10
    LRETR
```

変数 _g1 と _g2 が同一ページ内にあって、かつ関数 _foo が DP をすでにセットしているので、_foo への関数呼び出し後の MOVZ DP はポストリンク・オプティマイザによって除去されます。

ポストリンク・オプティマイザが関数呼び出し間で最適化を行うためには、関数には return 文が 1 つだけである必要があります。関数に複数の return 文が含まれている手書きのアセンブリに対してポストリンク・オプティマイザを実行すると、ポストリンク・オプティマイザが最適化した出力は間違ったものになる可能性があります。-plink オプションの後に -nf オプションを指定することで、関数呼び出し間での最適化をオフにすることができます。

5.5 他のポストリンク最適化

外部で定義されたシンボルを定数オペランドとして使用すると、アセンブラは即値を保持するために強制的に 16 ビット・エンコーディングを実施します。ポストリンク・オプティマイザは外部で定義されたシンボル値をアクセスするので、可能であれば 16 ビット・エンコーディングを 8 ビット・エンコーディングに置き換えます。たとえば次のとおりです。

```
.ref ext_sym ; externally defined to be 4
:
:
ADD AL, #ext_sym ; assembly will encode ext_sym with 16
                   ; bits
```

ext_sym は外部で定義されているので、アセンブラは ext_sym に対して 16 ビット・エンコーディングを選択します。ポストリンク・オプティマイザは、ext_sym のエンコーディングを 8 ビット・エンコーディングに変更します。たとえば次のとおりです。

```
.ref ext_sym
:
:
; << ADD=>ADDB>> ADD AL,#ext_sym
ADDB AL, #ext_sym
```

同様に、ポストリンク・オプティマイザは次の 2 ワードの命令を 1 ワードの命令に短縮しようとします。

2 ワードの命令	**1 ワードの命令**
ADD/SUB ACC, #*imm*	ADDB/SUBB ACC, #*imm*
ADD/SUB/AND/OR/XOR/CMP AL, #*imm*	ADDB/SUBB/ANDB/ORB/XORB/CMPB AL, #*imm*
MOVL XAR*n*, #*imm*	MOVB XAR*n*, #*imm*

5.6　ポストリンク最適化を制御する方法

ポストリンク最適化を制御するには 3 つの方法があります。それはファイルを除外する方法、アセンブリ・ファイルに特定のコメントを挿入する方法、および手作業でポストリンク最適化ファイルを編集する方法です。

5.6.1　ファイルを除外する方法（-ex オプション）

-ex オプションを指定すると、特定のファイルをポストリンク最適化プロセスから除外することができます。除外されるファイルは、-ex オプションの後に指定し、ファイルの拡張子を含める必要があります。-plink オプションの後に -ex オプションを指定します。その後に他のオプションは指定してはいけません。たとえば次のとおりです。

```
cl2000 -v28 file1.asm file2.asm file3.asm -k -z -lnk.cmd -plink -o prog.out -ex file3.asm
```

file3.asm はポストリンク最適化プロセスから除外されます。

5.6.2　アセンブリ・ファイル内でポストリンク最適化を制御する方法

アセンブリ・ファイル内では特別にフォーマットされた 2 つのコメント文を用いて、ポストリンク最適化を無効にも有効にもすることができます。

```
;//NOPLINK//
;//PLINK//
```

NOPLINK コメントの後のアセンブリ文は、最適化されません。ポストリンクによる最適化は、//PLINK// コメントを使用すると再び有効になります。

PLINK と NOPLINK コメント・フォーマットは大文字小文字を区別しません。セミコロンと PLINK 区切りの間は空白にしておきます。PLINK と NOPLINK のコメントは、別々の行に単独で記述する必要があります。1 カラム目から始めてください。たとえば次のとおりです。

```
;        //PLINK//
```

5.6.3 ポストリンク・オプティマイザの出力を保持する方法（-k オプション）

-k オプションを指定すると、-plink オプションで生成されたポストリンク・ファイル（.pl）と絶対リスト・ファイル（.abs）を保持することができます。-k オプションを指定すると、ポストリンク・オプティマイザが加えた変更内容を確認できます。

.pl ファイルには、<<REDUNDANT>> または <<ADD=>ADDB>> など命令に対する改善内容を示すコメントアウトされた文が含まれています。コメントアウトされた行を除外するために、.pl ファイルをアセンブルして再びリンクします。

5.6.4 関数呼び出し間での最適化を無効にする（-nf オプション）

-nf オプションを指定すると、関数呼び出し間でのポストリンク最適化は無効になります。ポストリンク・オプティマイザは、return 文で関数の終了を認識します。また関数には return 文が 1 つだけ含まれていることを想定しています。手書きのアセンブリ・コードでは、関数に複数の return 文を記述することができます。このような場合、ポストリンクが最適化した出力は間違ったものになる可能性があります。-nf オプションを指定すると、関数呼び出し間での最適化をオフにすることができます。このオプションはすべてのファイルに影響を与えます。

5.7 ポストリンク・オプティマイザ使用時の制約事項

次の制約事項は、ポストリンク最適化に影響を与えます。

❑ ラベルのない宛先への分岐または呼び出しは、DP ロード命令の最適化を無効にします。すべての分岐の宛先にはラベルを付ける必要があります。

❑ データ・セクションの位置がコード・セクションのサイズによって変わる場合には、除去対象の DP ロード命令を決定するために使われるデータ・ページ・レイアウト情報は有効ではない場合があります。

  たとえば、次のリンク・コマンド・ファイルを考えてみましょう。

```
Sections
{     .text     > MEM,
      .mydata   > MEM,
}
```

  最適化後の .text セクションのサイズを変更すると、.bss セクションがシフトします。

  出力されたすべてのデータ・セクションは 64 ワード境界に位置合わせされることが保証されているので、このシフトに関する問題は除去されます。

  たとえば、次のリンク・コマンド・ファイルを考えてみましょう。

```
Sections
{     .text > MEM,
      .mydata align = 64 > MEM,
}
```

5.8 出力ファイルの指定方法（-o オプション）

-plink オプションを指定した場合、-o *filename* オプションも指定する必要があります。出力ファイル名をリンカ・コマンド・ファイルで指定した場合、コンパイラはコマンド・ファイルにアクセスしてポストリンク最適化の他のフェーズにファイル名を渡すことはしません。そのため、そのプロセスはうまくいきません。たとえば次のとおりです。

```
cl2000 -v28 file1.c file2.asm -z -o prog.out lnk.cmd -plink
```

ポストリンク最適化フローは絶対リスタ abs2000 を使用するので、パスに組み込んでおく必要があります。

第 6 章

TMS320C28x C/C++ 言語の実装

TMS320C28x™ C/C++ コンパイラは、C プログラミング言語を標準化するために米国規格協会と国際標準化機構（ANSI/ISO）によって制定された C 言語規格をサポートしています。

TMS320C28x がサポートしている C++ 言語は、一部例外がありますが ANSI/ISO/IEC 14882-1998 規格で規定されています。

項目 **ページ**

6.1 TMS320C28x C の特性 6-2
6.2 TMS320C28x C++ の特性 6-5
6.3 組み込み C++ モードの有効化（-pe オプション） 6-6
6.4 データ型 6-7
6.5 レジスタ変数 6-11
6.6 asm 文 6-12
6.7 キーワード 6-14
6.8 プラグマ疑似命令 6-25
6.9 静的変数とグローバル変数の初期化方法 6-32
6.10 リンク名の生成方法 6-34
6.11 K&R C との互換性 6-35
6.12 言語モードの変更方法（-pk、-pr、および -ps オプション） 6-37
6.13 厳密 ANSI/ISO モードと緩和 ANSI/ISO モードの有効化（-ps および -pr オプション） 6-38
6.14 コンパイラの限界 6-38

6.1 TMS320C28x C の特性

ANSI/ISO 規格では、ターゲット・プロセッサ、ランタイム環境あるいはホスト環境の特性により影響を受ける C 言語の一部の事項について記載しています。これらの機能は、効率性や実用性といった理由で、各標準コンパイラの間でも異なる可能性があります。ここでは、これらの機能が TMS320C28x C/C++ コンパイラにどのように実装されているかを説明します。

以下に、これらの事項をすべて取り上げ、それぞれに対する TMS320C28x C/C++ コンパイラの動作を説明します。それぞれの説明には、正式な ANSI/ISO 規格およびカーニハンとリッチー（K&R）による『The C Programming Language』（第 2 版）に対する参照も含まれています。

6.1.1 識別子と定数

次の規則が識別子と定数に適用されます。

- 識別子に関しては、すべての文字が有効です。また、識別子に対する大文字と小文字は区別されます。これらの特性は、すべての TMS320C28x ツールにおいて内部・外部を問わず、すべての識別子に適用されます。
  （ANSI/ISO 3.1.2, K&R A2.3）
- ソース（ホスト）文字セットと実行（ターゲット）文字セットは、ASCII であることを前提とします。マルチバイト文字はありません。
  （ANSI/ISO 2.2.1, K&R A12.1）
- 文字定数や文字列定数における 16 進や 8 進のエスケープ・シーケンスは、最大で 32 ビットまでの値をもちます。 （ANSI/ISO 3.1.3.4, K&R A2.5.2）
- 複数の文字をもつ文字定数は、シーケンスの最後の文字としてコード化されます。たとえば次のとおりです。

  `'abc' == 'c'` （ANSI/ISO 3.1.3.4, K&R A2.5.2）

6.1.2 データ型

次の規則がデータ型に適用されます。

- データ型の表記方法については、6.4 節「データ型」（6-7 ページ）を参照してください。 （ANSI/ISO 3.1.2.5, K&R A4.2）
- size_t 型（sizeof 演算子の結果）は unsigned long です。
  （ANSI/ISO 3.3.3.4, K&R A7.4.8）

  printf を size_t 型と組み合わせて使うためには、size_t が long なので %ld を使用します。
- ptrdiff_t 型（ポインタ減算の結果）は long です。far ポインタの減算についても同様です。 （ANSI/ISO 3.3.6, K&R A7.7）

6.1.3 変換

次の規則が変換に適用されます。

- float から int への変換では、小数点以下の部分は切り捨てられます。
  （ANSI/ISO 3.2.1.3, K&R A6.3）
- ポインタおよび整数は自由に変換できます。 （ANSI/ISO 3.3.4, K&R A6.6）

6.1.4 式

次の規則が式に適用されます。

- 2 つの符号付き整数で除算をしたとき、どちらかが負であれば商（/）は負になり、剰余（%）の符号は被除数の符号と同じになります。たとえば次のとおりです。

```
10 / -3 ==-3,     -10 / 3 == -3
10 % -3 == 1,     -10 % 3 == -1
```

  （ANSI/ISO 3.3.5, K&R A7.6）
- 符号付きの値の右シフトは、算術シフトです。つまり、符号は保存されます。
  （ANSI/ISO 3.3.7, K&R A7.8）

6.1.5 宣言

次の規則が宣言に適用されます。

- *register* 記憶クラスは、すべての char 型、short 型、int 型、ポインタ型に有効です。
  （ANSI/ISO 3.5.1, K&R A8.1）
- 構造体のメンバは、16 ビット境界または 32 ビット境界に位置合わせされます。
  （ANSI/ISO 3.5.2.1, K&R A8.3）
- int 型のビット・フィールドは、符号付きです。ビット・フィールドは下位ビットからはじまるワードにパックされ、ワード境界にまたがることはありません。したがって、ビット・フィールドは C ソースで使われているサイズに関係なく、最大で 16 ビットのサイズに制限されます。
  （ANSI/ISO 3.5.2.1, K&R A8.3）

6.1.6 プリプロセッサ

プリプロセッサは、次のプラグマ疑似命令を認識しています。他のプラグマ疑似命令はすべて無視されます。プラグマ疑似命令は、コンパイラのプリプロセッサに関数の処理方法を指示します。認識されているプラグマは次のとおりです。

- ❑ CODE_SECTION（関数、「セクション名」）
- ❑ DATA_SECTION（シンボル、「セクション名」）
- ❑ INTERRUPT（関数）
- ❑ FUNC_EXT_CALLED（関数）
- ❑ FAST_FUNC_CALL（関数）

（ANSI/ISO 3.8.6, K&R A12.8）

プラグマの詳細は、6.8 節「プラグマ疑似命令」（6-25 ページ）を参照してください。

6.1.7 ヘッダ・ファイル

次の規則がヘッダ・ファイルに適用されます。ヘッダ・ファイルの詳細は、8.4 節「ヘッダ・ファイル」（8-18 ページ）を参照してください。

- ❑ 次の ANSI/ISO C ランタイムサポート関数はサポートされていません。
  - ■ locale.h
  - ■ signal.h

（ANSI/ISO 4.1, K&R B）

- ❑ stdlib ライブラリ関数 getenv と system はサポートされていません。

（ANSI/ISO 4.10.4, K&R B5）

- ❑ 浮動小数点型の戻り値を生成する数値計算ライブラリ関数の場合、生成された値が小さすぎて表現できないと、ゼロが戻され、かつ errno は ERANGE に設定されます。

6.2 TMS320C28x C++ の特性

TMS320C28x コンパイラは、ANSI/ISO/IEC 14882:1998 C++ 規格で規定されている C++ をサポートしています。この規格に対する例外は、次のとおりです。

- ❑ 完全な C++ 標準ライブラリ・サポートは含まれていません。C のサブセット、および基本言語サポートは含まれています。
- ❑ 次に示す C のライブラリ機能用の C++ ヘッダは含まれていません。
  - <ciso646>
  - <clocale>
  - <csignal>
  - <cwchar>
  - <cwctype>
- ❑ 次に示す C++ 標準ライブラリ・ヘッダ・ファイルの C++ ヘッダのみが含まれます。
  - <new>
  - <typeinfo>

  typeinfo ヘッダ内に含まれる bad_cast や bad_type_id はサポートされていません。
- ❑ 例外処理はサポートされていません。
- ❑ ランタイム型情報（RTTI）は、デフォルトで無効にされます。RTTI は、-rtti コンパイラ・オプションを指定して有効にできます。
- ❑ reinterpret_cast 型は、クラス同士に関連がない場合、ある 1 つのクラスのメンバに対するポインタは他のクラスのメンバに対するポインタをキャストすることができません。
- ❑ 2 つのフェーズの名前をテンプレートでバインドすることは、（この規格の [temp.res] および [temp.dep] に説明されているように）できません。
- ❑ テンプレートのパラメータは実装されていません。
- ❑ テンプレート用の export キーワードは実装されていません。
- ❑ 関数型の typedef は、メンバ関数 cv-qualifier を含むことはできません。
- ❑ クラス・メンバ・テンプレートの部分的な特殊化は、クラス定義の外部に対しては追加することができません。

6.3 組み込み C++ モードの有効化（-pe オプション）

コンパイラは、C++ のサブセットである、組み込みシステム・プログラミング用の組み込み C++ のコンパイルをサポートします。組み込み C++ 規格では、組み込みシステムでサポートするにはあまり価値がない機能やコストがかかり過ぎる一部の機能は省略しています。組み込み C++ 用にコンパイルする場合、コンパイラは省略された機能を使用するための診断情報を生成します。

組み込み C++ を有効にするには、-pe コンパイラ・オプションを指定してコンパイルします。

組み込み C++ は、以下の C++ 機能を省略します。

- テンプレート
- 例外処理
- ランタイム型情報
- 新しいキャスト構文
- キーワード *mutable*
- 多重継承
- 仮想継承

組み込み C++ の標準定義では、namespace および using 宣言がサポートされていません。しかし、TMS320C28x コンパイラでは、組み込み C++ でこれらの機能が使用できます。これは、C++ ランタイムサポート・ライブラリがそれらの機能を利用するからです。さらに、これらの機能は、実行時にペナルティを課しません。

TMS320C28x コンパイラは、組み込み C++ のコンパイル用に _embedded_cplusplus マクロを定義しています。

コンパイラに付属している rts.src ライブラリは、組み込み C++ 用にコンパイルされたモジュールとリンクするために使用することができます。

6.4 データ型

次の情報がデータ型に適用されます。

- ❑ すべての整数型（char、short、int、およびその符号なし）は同じ型であり、16 ビット・バイナリ値として表されます。
- ❑ long および unsigned long 型は、32 ビット・バイナリ値として表されます。
- ❑ long long および unsigned long long 型は、64 ビット・バイナリ値として表されます。
- ❑ signed 型は、2 の補数表記で表されます。
- ❑ char 型は signed 型で、int と同じです。
- ❑ enum 型のオブジェクトは 16 ビット値として表されます。式の中では、enum 型は int と同じです。
- ❑ float と double 浮動小数点型は同じで、IEEE 単精度フォーマットとして表されます。
- ❑ long double 浮動小数点型は、IEEE 倍精度フォーマットとして表されます。

各スカラー・データ型のサイズ、表現、および範囲を表 6-1 に示します。

範囲値の多くは、ヘッダ・ファイル limits.h 内で標準マクロとして使用できます。このファイルは、コンパイラに付属しています。詳細は、6.14 節「コンパイラの限界」(6-38 ページ）を参照してください。

表 6-1. TMS320C28x C のデータ型

型	サイズ	表現	範囲	
			最小	最大
char、signed char	16 ビット	ASCII	-32768	32767
unsigned char	16 ビット	ASCII	0	65535
short	16 ビット	2 の補数	-32768	32767
unsigned short	16 ビット	バイナリ	0	65535
int、signed int	16 ビット	2 の補数	-32768	32767
unsigned int	16 ビット	バイナリ	0	65535
long、signed long	32 ビット	2 の補数	-2147483648	214783647
unsigned long	32 ビット	バイナリ	0	4294967295
long long、signed long long	64 ビット	2 の補数	-9223372036854775808	9223372036854775807
unsigned long long	64 ビット	バイナリ	0	18446744073709551615
enum	16 ビット	2 の補数	-32768	32767
float	32 ビット	IEEE 32 ビット	1.19209290e-38	3.4028235e+38
double	32 ビット	IEEE 32 ビット	1.19209290e-38	3.4028235e+38
long double	64 ビット	IEEE 64 ビット	2.22507385e-308 †	1.79769313e+308
pointers	16 ビット	バイナリ	0	0xFFFF
far pointers	22 ビット	バイナリ	0	0x3FFFFF

† 数字は最小精度。

注： TMS320C28x のバイトは 16 ビット

ANSI/ISO 定義により、sizeof 演算子はオブジェクトの保存に必要なバイト数を出します。さらに ANSI/ISO は、sizeof が char に適用される場合、結果が 1 になると規定しています。TMS320C28x の char が 16 ビットなので（個別にアドレス設定できるようにするため）、1 バイトも 16 ビットです。このため、予期しない結果になることがあります。sizeof (int) == 1 (2 ではない) となります。TMS320C28x のバイトとワードは同じ（16 ビット）です。

6.4.1 64 ビット整数のサポート

現在、TMS320C28x コンパイラは long long および unsigned long long データ型をサポートしています。範囲値は、ヘッダ・ファイル limits.h 内で標準マクロとして使用できます。

long long データ型は、レジスタのペアで格納されます。メモリ内では、ワード（32 ビット）で位置合わせされたアドレスに 64 ビット・オブジェクトとして格納されます。

long long 型の整数定数は、接尾部 ll または LL が付いています。この接尾部がないと、定数の値が定数の型を決定します。

C 入出力で long long の書式化規則は、書式文字列では ll が必要です。たとえば次のとおりです。

```
printf("%lld", 0x0011223344556677);
printf("%llx", 0x0011223344556677);
```

ランタイムサポート・ライブラリ関数 llabs()、strtoll()、および strtoull() が追加されています。

6.4.2 C28x long double 浮動小数点型による変更

TMS320C28x 専用の C/C++ コードをコンパイルする場合、long double 浮動小数点型は現在 IEEE 64 ビット倍精度です。書式を変更する浮動小数点型は他にはありません。つまり、次のようになります。

C28x の浮動小数点型

型	フォーマット
float	IEEE 32 ビット単精度
double	IEEE 32 ビット単精度
long double	IEEE 64 ビット倍精度

long double を定数に初期化する場合、接尾部 l または L を付ける必要があります。定数は、接尾部がないと double 型として扱われます。ランタイムサポートの dobule から long への変換ルーチンは初期化のために呼び出されます。このため（変換によりビットが失われるので）、精度落ちが発生します。たとえば次のとおりです。

```
long double a = 12.34L; /* correctly initializes to double precision */
long double b = 56.78;   /* converts single precision value to double precision */
```

C 入出力で long double の書式化規則は、書式文字列では大文字の 'L' が必要です。たとえば次のとおりです。

```
printf("%Lg", 1.23L);
printf("%Le", 3.45L);
```

long double 型の 64 ビット IEEE 倍精度フォーマットへの変更に対応するため、C28x 呼び出し規則が変更されました。

すべての long double 型の引数は、参照渡しされます。long double 型の戻り値は、参照によって戻されます。最初の 2 つの long double 型の引数は、XAR4 と XAR5 にそれぞれのアドレスを渡します。他のすべての long double 型の引数は、スタックにアドレスをセットして渡されます。これらの long double 型のアドレスは far メモリに置かれると想定することが必要です。したがって、呼び出された関数は、long double 型の引数のアドレスに対して常に 32 ビットを読み出します。

関数が long double の値を戻す場合、その呼び出しを行う関数は戻りアドレスを XAR6 に配置します。たとえば次のとおりです。

```
long double foo(long double a, long double b, long double c)
{
   long double d = a + b + c;
   return d;
}

long double a = 1.2L;
long double b = 2.2L;
long double c = 3.2L;
long double d;

void bar()
{
   d = foo(a, b, c);
}
```

関数 bar() では、foo() の呼び出し時に、レジスタ値は次のようになります。

レジスタ	等しくなる値
XAR4	a のアドレス
XAR5	b のアドレス
*-SP[2]	c のアドレス
XAR6	d のアドレス

必要な long double 型の算術演算子および変換関数を含むように、ランタイムサポート・ライブラリはアップデートされています。すべての C27x/C28x 浮動小数点ランタイムサポート・ルーチンには、アップデートされた名前があります。たとえば、以前の浮動小数点呼び出しの加算ルーチンは、次のようなものでした。

```
LCR F$$ADD
```

これは次のようにアップデートされています。

```
LCR FS$$ADD  ; single precision add
LCR FD$$ADD  ; double precision add
```

浮動小数点算術ルーチンまたは変換ルーチンを呼び出す C28x ルーチンはすべて、再コンパイルする必要があります。

6.5 レジスタ変数

C/C++ コンパイラは、オプティマイザを使用するかどうかに応じてレジスタ変数（register キーワードにより宣言された変数）の処理方法を変更します。

- ❑ オプティマイザと組み合わせてコンパイル

  コンパイラは register 宣言をすべて無視し、レジスタを最も効率的に使用するコスト・アルゴリズムを使用して、レジスタを変数と一時値に割り当てます。

- ❑ オプティマイザを使わずにコンパイル

  register キーワードを使用すると、変数をレジスタへの割り当ての候補として示唆できます。一時的な式の結果の割り当てに使用するのと同じレジスタのセットを、レジスタ変数の割り当てに使用します。

コンパイラは、すべての register 定義を尊重しようとします。コンパイラは、適切なレジスタを使い切ると、レジスタの内容をメモリに転送してレジスタを解放します。レジスタ変数として定義したオブジェクトの数が多すぎると、コンパイラが一時的な式の結果に使用できるレジスタの数が限られます。そのため、レジスタの内容がメモリへ転送される頻度が過度になります。

スカラ型のオブジェクト（整数、浮動小数点、ポインタ）は、レジスタ変数として宣言できます。それ以外の型のオブジェクトに register を指定しても、無視されます。

レジスタ記憶クラスは、ローカル変数だけでなく、パラメータに使用しても便利です。通常、関数の中で一部のパラメータはスタック上の位置にコピーされ、関数本体が実行される間その位置で参照されます。レジスタ・パラメータは、スタックではなくレジスタにコピーされます。この動作により、関数内でのパラメータへのアクセスが高速になります。

レジスタ変数の詳細については、7.2 節「レジスタ規則」（7-11 ページ）を参照してください。

6.6 asm 文

TMS320C28x C/C++ コンパイラでは、TMS320C28x アセンブリ言語命令や疑似命令をコンパイラのアセンブリ言語出力に直接埋め込むことができます。この機能は、C/C++ 言語への拡張機能である *asm* 文によるものです。asm 文は、構文的には、1 つの文字列定数引数をもつ asm という関数の呼び出しに似ています。

asm("*assembler text***");**

コンパイラは、引数文字列を出力ファイルに直接コピーします。アセンブラ・テキストは二重引用符で囲みます。通常の文字列エスケープ・コードは、それぞれの定義を保持します。たとえば、以下のように引用符のある .string 疑似命令を指定できます。

```
asm("STR: .string \"abc\"");
```

挿入するコードは正しいアセンブリ言語文でなければなりません。通常のアセンブリ言語文同様に、コードの先頭には、ラベル、空白、タブ、コメント（* または ; ）のいずれかがきます。コンパイラはこの文字列のチェックはしません。エラーがある場合はアセンブラが検出します。アセンブリ言語文の詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

これらの asm 文は、通常の C/C++ 文の構文上の制約を受けません。コード・ブロック外でも文や宣言として使用できます。これは、コンパイルしたモジュールの先頭に疑似命令を挿入するときに特に便利な機能です。

注：　RPT 命令に対して 1 つの asm 文を使用してください

C28x RPT 命令を追加する場合、RPT 命令とリピートされる命令に対して別々の asm 文を使用しないでください。コンパイラは、asm 疑似命令の間に debug 疑似命令を挿入することができます。またアセンブラは RPT 命令とリピートされる命令の間に疑似命令を挿入することを許可しません。たとえば、RPT MAC 命令を挿入するためには、次のように疑似命令を使います。

```
asm("\tRPT #10\n\t||MAC P, *XAR4++, *XAR7++");
```

注：　asm 文により C/C++ 環境を損なわないでください

asm 文により C/C++ 環境が損なわれないよう特に注意してください。コンパイラは挿入された命令をチェックしません。ジャンプまたはラベルを C/C++ コードに挿入すると、挿入したコードの中やその周辺で思いがけず変数が操作される場合があります。セクションを変更したり、アセンブリ環境に影響を及ぼしたりする疑似命令も、問題となる場合があります。asm 文とともにオプティマイザを使用する場合は特に注意してください。コンパイラが asm 文を削除することはありませんが（そのような文に到達しない場合を除いて）、asm 文周辺のコード順序を大幅に再配置する可能性があり、予期せぬ結果を招くことがあります。asm コマンドは定義上 C/C++ がアクセスできないハードウェアの機能をアクセスできるようにするために用意されています。

6.7 キーワード

TMS320C28x C/C++ コンパイラは、標準の const、register、および volatile キーワードをサポートしています。さらに、C28x C/C++ コンパイラは C/C++ 言語を拡張して、cregister および interrupt キーワードをサポートしています。C モードでは、C/C++ コンパイラは near および far キーワードをサポートします。

6.7.1 const キーワード

C28x C/C++ コンパイラは、ANSI/ISO 規格のキーワード *const* をサポートしています。このキーワードを使用すると、大幅に最適化することができ、特定のデータ・オブジェクトに対する記憶域の割り当てをより細かく制御できます。const 修飾子を変数または配列の定義に適用すると、その値が変更されないようにすることができます。

const キーワードを定義の中に置くことが重要です。たとえば、次の最初の文は、変数 int を指す定数ポインタ p を定義します。2 番目の文は、定数 int を指す変数ポインタ q を定義します。

```
int * const p = &x;
const int * q = &x;
```

6.7.2 volatile キーワード

オプティマイザはデータ・フローを解析し、メモリ・アクセスを可能な限り回避します。C コードに記述されているとおりのメモリ・アクセスに依存しているコードがある場合は、必ず volatile キーワードを使用してそれらのアクセスを特定しなければなりません。volatile キーワードを使って修飾されている変数は、(レジスタとは対照的に）初期化されないセクションに割り当てられます。コンパイラは、volatile 変数への参照を最適化により除去することはありません。

次の例では、あるロケーションで 0xFF が読み出されるまでループが繰り返されます。

```
unsigned int *ctrl;
while (*ctrl !=0xFF);
```

この例で、*ctrl はループ不変式です。そのため、ループは 1 回のメモリ読み取りにまで簡略化されます。これを訂正するには、*ctrl を次のように定義します。

```
volatile unsigned int *ctrl;
```

ここで *ctrl ポインタは、割り込みフラグなどハードウェア・ロケーションを参照することを想定しています。

6.7.3 cregister キーワード

コンパイラは高級言語からレジスタを制御できるようにするために、cregister キーワードを追加して C 言語を拡張しています。

cregister キーワードをオブジェクトに使用すると、コンパイラは TMS320C28x の標準制御レジスタのサブセットとオブジェクト名を比較します（表 6-2 を参照）。名前が一致した場合、コンパイラは制御レジスタを参照するコードを生成します。名前が一致しない場合、コンパイラはエラーを発行します。

表 6-2. 有効な制御レジスタ

レジスタ	説明
IER	インタラプト・イネーブル・レジスタ
IFR	インタラプト・フラグ・レジスタ

cregister キーワードは、ファイル・スコープ内でのみ使用できます。cregister キーワードは、関数境界内で宣言することは許可されていません。このキーワードは、整数型またはポインタ型のオブジェクトでのみ使用できます。cregister キーワードは、浮動小数点型のオブジェクトや構造体または共用体のオブジェクトでは許可されていません。

cregister キーワードは、オブジェクトが volatile であることを意味するものではありません。参照されている制御レジスタが volatile である（つまり、外部制御によって変更される）場合には、オブジェクトも volatile キーワードを使って宣言されている必要があります。

表 6-2 の制御レジスタを使うには、各レジスタを次のように宣言する必要があります。

extern cregister volatile unsigned int *register***;**

レジスタを宣言してしまうと、制限された方法ですがレジスタ名を直接使うことができます。IFR は読み出し専用で、即値との | (OR) 演算を使用することでのみセットできます。即値との & (AND) 演算を使用することでのみクリアできます。たとえば次のとおりです。

```
IFR |= 0x4;
IFR &= 0x0800
```

また IER レジスタは、OR と AND 以外の代入時にも使用できます。C28x アーキテクチャにはこれらのレジスタを操作する限定された命令があるので、コンパイラはこれらのレジスタの不正な使用を検出すると次のメッセージを表示して終了してしまいます。

```
>>> Illegal use of control register
```

制御レジスタの宣言方法と使用方法の例については、例 6-1 を参照してください。

例 6-1. 制御レジスタの宣言方法と使用方法

```
extern cregister volatile unsigned int IFR;
extern cregister volatile unsigned int IER;
extern int x;

main()
{
   IER = x;
   IER |= 0x100;
   printf("IER = %x\n", IER);

   IFR &= 0x100;
   IFR |= 0x100;
}
```

6.7.4 interrupt キーワード

C28x コンパイラは、関数を割り込み関数として扱うよう指定する interrupt キーワードを追加することによって、C/C++ 言語を拡張しています。

割り込みを処理する関数は、特別なレジスタ保存規則と特別なリターン・シーケンスに従います。C/C++ コードに割り込みを行う場合、割り込みルーチンは、そのルーチンが使用するか、そのルーチンが呼び出す関数が使用するすべてのマシン・レジスタの内容を保存する必要があります。関数の定義に interrupt キーワードを使用した場合、コンパイラは割り込み関数の規則に基づいてレジスタ保存を生成し、割り込み用の特別なリターン・シーケンスを生成します。

void を戻すように定義した、パラメータをもたない関数にのみ interrupt キーワードを使用できます。interrupt 関数の本体は、ローカル変数をもつことができ、スタックまたはグローバル変数を自由に使用できます。たとえば次のとおりです。

```
interrupt void int_handler()
{
    unsigned int flags;

    ...
}
```

c_int00 が C のエントリ・ポイントです。この名前は、システム・リセット割り込み用に予約されています。この特別な割り込みルーチンは、システムを初期化し、main 関数を呼び出します。このルーチンには呼び出し側がないので、c_int00 はレジスタを保存しません。

6.7.5 ioport キーワード

ioport キーワードを使うと、TMS320C28x デバイスの入出力（I/O）ポート空間へアクセスできるようになります。このキーワードの形式は次のとおりです。

ioport *type* **port***hex_num*

ioport	これがポート変数であることを示すキーワード。
type	char、short、int、unsigned int のいずれか。
port*hex_num*	ポート番号。*hex_num* は 16 進数。

ポート変数のすべての宣言は、ファイルレベルで行う必要があります。関数レベルで宣言されたポート変数は、サポートされません。

例 6-2 に、ioport キーワードの使い方を示します。例 6-2 では、入出力（I/O）ポートを int ポート 10h として宣言し、a をポート 10h に書き込み、ポート 10h を b に読み出します。次の例では、他の型の式でポート変数を使う方法も示しています。

例 6-2.　ioport キーワードの使用方法

(a)　C ソース

```
ioport int port10;
int a; int b; int c;
extern void foo(int);

void func()
{
   port10 = a;
   b = port10;
   foo(port10);
   c = port10 + b;
   port10 += a;
}
```

例 6-2. ioport キーワードの使用方法（続き）

(b) C コンパイラ出力

```
_func:
        .line   3
;----------------------------------------------------------------------
;   7 | port10 = a;
;----------------------------------------------------------------------
        MOVZ      DP,#_a
        OUT       *(010H),@_a          ; |7|
        .line   4
;----------------------------------------------------------------------
;   8 | b = port10;
;----------------------------------------------------------------------
        IN        @_b,*(010H)          ; |8|
        .line   5
;----------------------------------------------------------------------
;   9 | foo(port10);
;----------------------------------------------------------------------
        IN        AL,*(010H)           ; |9|
        LCR       #_foo                ; |9|
        ; call occurs [#_foo] ; |9|
        .line   6
;----------------------------------------------------------------------
;  10 | c = port10 + b;
;----------------------------------------------------------------------
        IN        AL,*(010H)           ; |10|
        MOVZ      DP,#_b
        ADD       AL,@_b               ; |10|
        MOV       @_c,AL               ; |10|
        .line   7
;----------------------------------------------------------------------
;  11 | port10 += a;
;----------------------------------------------------------------------
        IN        AL,*(010H)           ; |11|
        ADD       AL,@_a               ; |11|
        OUT       *(010H),AL           ; |11|
        .line   8
        LRETR
        ; return occurs
```

6.7.6 far キーワード

C/C++ コンパイラのデフォルト・アドレス空間は、メモリの下位 64K に制限されています。すべてのポインタのサイズはデフォルトで 16 ビットです。TMS320C28x は、16 ビットを超えたアドレッシングをサポートします。C では、コンパイラは far 型の修飾子を使って C 言語を拡張することで、4 メガワードのデータまでアクセスできます。far ポインタのサイズは、22 ビットになります。

注： C++ での far のサポート

TMS320C/C++ コンパイラは、C++ では far キーワードをサポートしません。C++ で far オブジェクトにアクセスするにはラージ・メモリ・モデル・オプションを指定するか、組み込み関数を使用します。詳細については、6.7.7 項「ラージ・メモリ・モデルを使用する方法（-ml オプション）」（6-21 ページ）および 6.7.8 項「C++ で far メモリをアクセスする組み込み関数を使用する方法」（6-22 ページ）を参照してください。

6.7.6.1 セマンティック

オブジェクトを far 宣言することは、そのオブジェクトを far メモリに配置することを示します。これは、デフォルトの .bss とは異なるセクションにそのオブジェクト用の空間を確保することで行われます。far 宣言したグローバル変数は、.ebss と呼ばれるセクションに配置されます。このセクションは、TMS320C28x データ・アドレス空間の任意の場所にリンク可能です。同様に、far 宣言した const オブジェクトは、.econst セクションに配置されます。

far オブジェクトを指すように宣言されたポインタのサイズは 22 ビットです。これらのオブジェクトは、データを格納するために 2 つのメモリ・ロケーションを必要とし、アドレッシングを実行するために XAR レジスタを必要とします。

注： ポインタの使い分け

far データを指すポインタを宣言する方法（far int *pf;）とポインタ自体を far 宣言する方法（int *far fp;）の使い分けがあります。

グローバル変数と静的変数だけが、far 宣言できます。関数（自動記憶クラス）内で宣言された非静的変数は far にはなりません。これは、これらの変数がスタックに割り当てられているからです。コンパイラは警告を発行し、これらの変数を near として扱います。

構造体のメンバを far 宣言するのは意味がありません。far 宣言された構造体は、その構造体のすべてのメンバが far であることを意味します。

6.7.6.2 構文

コンパイラは、far または __far を同じ意味のキーワードとして認識します。far キーワードは、修飾子型として機能します。変数宣言時に、far は型修飾子の const と volatile と同様に使われます。例 6-3 に、変数を宣言するための正しい方法を示します。

例 6-3. 変数を宣言する方法

```
int far sym;           // sym is located in far memory.
far int sym;           // sym is located in far memory.

struct S far s1;       // Likewise for structure s1.

int far *ptr;          // This declares a pointer that points to a far int.
                       // The variable ptr is itself near.

int * far ptr;         // This declares a pointer to a near int. The variable
                       // ptr, however, is located in far memory.

int far * far ptr;     // The pointer and the object it points to are both far.

int far *func();       // Function that returns a pointer to a far int.

int far *memcpy_ff(far void *dest, const far void *src, int count);

                       // Function that takes two far pointers as arguments
                       // and returns a far pointer.

int *far func()        // ERROR: Declares the function as far. Since the
                       // program address space is flat 22-bit, this has no
                       // meaning. Far applies to data only.

int func()
{
int far x;             // ERROR: Far only applies to global/static variables.
:                      // Auto variables on the stack are required to be near
:
int far *ptr           // Ok, since the pointer is on the stack, but points
                       // to far
}


struct S               // Declaring structure members as far is meaningless,
{
                       // unless it's a pointer to far.  Structure objects
int a;                 // can, of course, be declared far.
int far b;
int far *ptr;
};
```

6.7.6.3 ランタイム・ライブラリによる far のサポート

ランタイム・ライブラリは、ポインタを引数としてもっていたり、値を戻したり、RTS グローバル変数を参照したりする、ほとんどの RTS 関数の far バージョンを組み込むように拡張されています。far ヒープを管理するサポート機能もあります。RTS による far のサポートには、C 入出力ルーチンや C 入出力ルーチンを参照するような関数は組み込まれていません。RTS による far のサポートの詳細は、8.5 節「ランタイムサポート関数およびマクロのまとめ」(8-31 ページ) を参照してください。

6.7.6.4 far ポインタによる数値計算

ANSI/ISO 規格では、有効なポインタ演算は同じ型のポインタの代入、ポインタと整数の加算または減算、同一配列のメンバへの 2 つのポインタの減算と比較、ゼロへの代入および比較であると規定しています。これ以外のすべてのポインタによる演算は不正なものです。

これらの規則は、far ポインタに適用されます。2 つの far ポインタの減算の結果は、16 ビット整数型になります。これは、サイズが 64K ワードを超える大きな配列では、コンパイラはポインタによる減算をサポートしていないことを表しています。

6.7.6.5 far 型の文字列定数

文字列定数を far メモリに配置する方法の詳細は、7.1.8 項「文字列定数」(7-9 ページ) および 7.1.9 項「far 文字列定数」(7-10 ページ) を参照してください。

6.7.6.6 .econst をプログラム・メモリに割り当てる方法

ブート時に .econst セクションをプログラム・メモリからデータ・メモリへコピーする方法の詳細は、7.1.3 項「.const/.econst をプログラム・メモリに割り当てる方法」(7-6 ページ) を参照してください。

6.7.7 ラージ・メモリ・モデルを使用する方法 (-ml オプション)

C++ コードでは far キーワードはサポートされていないので、ラージ・メモリ・モデル・オプションを指定します。-ml オプションにより、コンパイラはフラットな 22 ビット・アドレス空間を備えたものとして TMS320C28x アーキテクチャを見ることができます。-ml オプションを指定してコンパイルするとき、すべてのポインタは 22 ビットのポインタと見なされます。データ型のサイズには、64K ワードの制限はありません。

6.7.7.1　ラージ・メモリ・モデル・オプション

コンパイラに対してラージ・メモリ・モデルを指定するオプションは、-ml です。各コンパイラ・ツールを別々に起動する場合、次のオプションを使う必要があります。

- ❏ パーサ : -ml オプション
- ❏ オプティマイザ : -m オプション
- ❏ コード・ジェネレータ : -l オプション
- ❏ アセンブラ : -mf オプション

アセンブラに対する -mf オプションは、ラージ・メモリ・モデルのコードをもつ 16 ビット・コードを条件付きコンパイルできるようにするために使われます。LARGE_MODEL シンボルはアセンブラによって事前定義されていて、-mf オプションを指定しない限り自動的に偽に設定されています。

6.7.7.2　ラージ・メモリ・モデルのランタイムサポート・ライブラリ

ランタイムサポート・ライブラリは、条件付きコンパイルを使ってラージ・メモリ・モデルをサポートしています。ランタイムサポート・ライブラリをコンパイルする場合、LARGE_MODEL シンボルを定義しておく必要があります。このシンボルは、size_t 引数を渡したり、または size_t 引数を戻したりする任意のランタイムサポート関数を自分のコードからアクセスする場合に必要です。このシンボルは、ランタイムサポート va_arg or offsetof( ) マクロを使用する場合にも必要です。したがって、ラージ・メモリ・モデルでコンパイルする場合には、-d コンパイラ・オプション（2-13 ページを参照）を指定して、LARGE_MODEL シンボルを事前に定義してください。

6.7.8　C++ で far メモリをアクセスする組み込み関数を使用する方法

far キーワードが拡張するのは、C 言語だけです。C++ では far キーワードはサポートされていません。ラージ・メモリ・モデルを使わない場合に、C++ の rts ライブラリのヒープ管理サポート・ルーチンと組み合わせて C++ で far メモリにアクセスできるようにするために、組み込み関数が用意されています。組み込み関数は、アドレスを表す長整数型を受け付けます。組み込み関数の戻り値は、基本データ型（word、long、float、long long、long double）にアクセスできるようにするために、参照先になりうる暗黙的な far ポインタです。

- ❏ __farptr_to_word（long 型のアドレス）
- ❏ __farptr_to_long（long 型のアドレス）
- ❏ __farptr_to_float（long 型のアドレス）
- ❏ __farptr_to_llong（long 型のアドレス）
- ❏ __farptr_to_ldouble（long 型のアドレス）

far メモリをアクセスできる long 型のアドレスを生成するには、次の 2 つの方法があります。

❑ C++ ランタイムサポート・ライブラリに付属している C++ ランタイムサポート・ライブラリのヒープ管理関数を使用することができます。

- long far_calloc (unsigned long num, unsigned long size);
- long far_malloc (unsigned long size);
- long far_realloc (long ptr, unsigned long size);
- void far_free (long ptr);

これらの関数は、far ヒープ領域にメモリを割り当てます。組み込み関数を使って、そのメモリにアクセスすることができます。たとえば次のとおりです。

```
#include <stdlib.h>

extern int x;
extern long y;
extern float z;

extern void func1 (int a);
extern void func2 (long b);
extern void func3 (float c);


//create a far object on the heap
long farint = far_malloc (sizeof (int));
long farlong = far_malloc (sizeof (long));
long farfloat = far_malloc (sizeof (float));


//assign a value to the far object
*__farptr_to_word (farint) = 1;
*__farptr_to_word (farint) = x;

*__farptr_to_long (farlong) = 78934;
*__farptr_to_long (farlong) = y;

*__farptr_to_float (farfloat) = 4.56;
*__farptr_to_float (farfloat) = z;


//use far object in expression
x = *__farprt_to_word(farint) + x;
y = *__farptr_to_long(farlong) + y;
z = *__farptr_to_float(farfloat) + z;


//use as argument to function
func1 (*__farptr_to_word (farint));
func2 (*__farptr_to_long (farlong));
func3 (*__farptr_to_float (farfloat));


//free the far object
far_free (farint);
far_free (farlong);
far_free (farfloat);
```

- far メモリにリンクされている変数をデータ・セクションに配置するために、DATA_SECTION プラグマをインライン・アセンブリと一緒に使うことができます。インライン・アセンブリを使って、これらの変数に対して long 型のアドレスを作成します。その後、組み込み関数を使って、これらの変数にアクセスできます。たとえば次のとおりです。

```
#pragma DATA_SECTION (var, “.mydata”)
int var;
extern const long var_addr;
asm ("\t .sect .const");
asm ("var_addr .long var");
int x;
x = *__farptr_to_word (var_addr);
```

6.8 プラグマ疑似命令

プラグマ疑似命令は、コンパイラのプリプロセッサに関数の処理方法を指示します。TMS320C28x C/C++ コンパイラは、次のプラグマをサポートしています。

- ❏ CODE_SECTION
- ❏ DATA_SECTION
- ❏ FAST_FUNC_CALL
- ❏ FUNC_EXT_CALLED
- ❏ INTERRUPT

関数本体の内部で、引数 *func* および *symbol* を定義または宣言することはできません。プラグマは関数の本体の外部で指定しなければなりません。しかも、プラグマは *func* 引数または *symbol* 引数のすべての宣言、定義、または参照の前に置かなければなりません。これらの規則に従わなかった場合、コンパイラは警告を発します。

プラグマの構文は、C と C++ 間で異なります。C では、プラグマを適用しようとするオブジェクトまたは関数の名前を最初の引数として指定しなければなりません。C++ では名前は省略されます。プラグマは、その後に続くオブジェクトまたは関数の宣言に適用されます。

6.8.1 CODE_SECTION プラグマ

CODE_SECTION プラグマは、*section name* というセクションに *func* のためのスペースを割り当てます。CODE_SECTION プラグマは、コード・オブジェクトを .text セクションとは別の領域内にリンクする場合に便利です。

C の構文は次のとおりです。

#pragma CODE_SECTION (*func***, "***section name***")**

C++ の構文は次のとおりです。

#pragma CODE_SECTION ("*section name***")**

例 6-4 に、CODE_SECTION プラグマの使用方法を示します。

例 6-4.　CODE_SECTION プラグマの使用方法

(a)　C ソース・ファイル

```
char bufferA[80];
char bufferB[80];

#pragma CODE_SECTION(funcA, "codeA")

char funcA(int i);
char funcB(int i);

void main()
{
   char c;
   c = funcA(1);
   c = funcB(2);
}

char funcA (int i)
{
   return bufferA[i];
}

char funcB (int j)
{
   return bufferB[j];
}
```

(b)　アセンブリ・ソース・ファイル

```
        .sect   “.text”
        .global _main

;***************************************************************
;* FNAME: _main                         FR SIZE:   2           *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                              0 Parameter,  1 Auto,  0 SOE   *
;***************************************************************

:_main:
        ADDB      SP,#2
        MOVB      AL,#1                 ; |12|
        LCR       #_funcA               ; |12|
        ; call occurs [#_funcA] ; |12|
        MOV       *-SP[1],AL            ; |12|
        MOVB      AL,#1                 ; |13|
        LCR       #_funcB               ; |13|
        ; call occurs [#_funcB] ; |13|
        MOV       *-SP[1],AL            ; |13|
        SUBB      SP,#2
        LRETR
        ; return occurs
```

例 6-4. CODE_SECTION プラグマの使用方法（続き）

```
        .sect   "codeA"
        .global _funcA
;****************************************************************
;* FNAME: _funcA                        FR SIZE:   1            *
;*                                                              *
;* FUNCTION ENVIRONMENT                                         *
;*                                                              *
;* FUNCTION PROPERTIES                                          *
;*                           0 Parameter,  1 Auto,  0 SOE       *
;****************************************************************

_funcA:
        ADDB      SP,#1
        MOV       *-SP[1],AL            ; |17|
        MOVZ      AR6,*-SP[1]           ; |18|
        ADD       AR6,#_bufferA         ; |18|
        SUBB      SP,#1                 ; |18|
        MOV       AL,*+XAR6[0]          ; |18|
        LRETR
        ;return occurs

         .sect   “.text”
         .global _funcB

;****************************************************************
;* FNAME: _funcB                        FR SIZE:   1            *
;*                                                              *
;* FUNCTION ENVIRONMENT                                         *
;*                                                              *
;* FUNCTION PROPERTIES                                          *
;*                           0 Parameter,  1 Auto,  0 SOE       *
;****************************************************************

_funcB:
        ADDB      SP,#1
        MOV       *-SP[1],AL            ; |22|
        MOVZ      AR6,*-SP[1]           ; |23|
        ADD       AR6,#_bufferB         ; |23|
        SUBB      SP,#1                 ; |23|
        MOV       AL,*+XAR6[0]          ; |23|
        LRETR
        ;return occurs
```

6.8.2 DATA_SECTION プラグマ

DATA_SECTION プラグマは、*section name* というセクションに *symbol* のためのスペースを割り当てます。

これは、.bss セクションとは別の領域内にデータ・オブジェクトをリンクする場合に便利です。例 6-5 に、DATA_SECTION プラグマの使用方法を示します。

C の構文は次のとおりです。

#pragma DATA_SECTION (*symbol***, "***section name***")**

C++ の構文は次のとおりです。

#pragma DATA_SECTION ("*section name*"**)**

例 6-5. DATA_SECTION プラグマの使用方法

(a) C ソース・ファイル

```
#pragma DATA_SECTION(bufferB, "my_sect")
char bufferA[512];
char bufferB[512];
```

(b) アセンブリ・ソース・ファイル

```
            .global _bufferA
            .bss    _bufferA,512,1
            .global _bufferB
_bufferB:   .usect  "my_sect",512,1
```

6.8.3 FAST_FUNC_CALL プラグマ

FAST_FUNC_CALL プラグマは、関数に適用された場合、その関数を呼び出すために CALL 命令ではなく、TMS320C28x FFC 命令を生成します。FFC 命令の詳細は、『TMS320C28x DSP CPU 及びインストラクション・セット リファレンス・ガイド』を参照してください。

C の構文は次のとおりです。

#pragma FAST_FUNC_CALL (*func*)

C++ の構文は次のとおりです。

#pragma FAST_FUNC_CALL

FAST_FUNC_CALL プラグマは、「LB *XAR7」命令を使って戻るアセンブリ関数の呼び出しにのみ適用してください。C/C++ とアセンブリ・コードを組み合わせる方法については、7.4.1 項「C/C++ コードでのアセンブリ言語モジュールの使用方法」(7-19 ページ) を参照してください。

このプラグマはアセンブリ関数にのみ適用してください。コンパイラがファイル・スコープで *func* の定義を検出すると、警告を発行し、このプラグマを無視します。

次の例では、FAST_FUNC_CALL プラグマの使用方法を示します。

例 6-6. FAST_FUNC_CALL プラグマの使用方法

(a) アセンブリ関数

```
_add_long:
        ADD ACC, *-SP[2]
        LB *XAR7
```

(b) C ソース・ファイル

```
#pragma FAST_FUNC_CALL (add_long);

long add_long(long, long);

void foo()
{
   long x, y;
   x = 0xffff;
   y = 0xff;
   y = add_long(x, y);
}
```

例 6-6.　FAST_FUNC_CALL プラグマの使用方法（続き）

(c)　結果的に作成されるアセンブリ・ファイル

```
;***************************************************************
;* FNAME: _foo                          FR SIZE:   6            *
;*                                                              *
;* FUNCTION ENVIRONMENT                                         *
;*                                                              *
;* FUNCTION PROPERTIES                                          *
;*                            2 Parameter,  4 Auto,  0 SOE      *
;***************************************************************

_foo:
        ADDB      SP,#6
        MOVB      ACC,#255
        MOVL      XAR6,#65535           ; |8|
        MOVL      *-SP[6],ACC
        MOVL      *-SP[2],ACC           ; |10|
        MOVL      *-SP[4],XAR6          ; |8|
        MOVL      ACC,*-SP[4]           ; |10|
        FFC       XAR7,#_add_long       ; |10|
        ; call occurs [#_add_long] ; |10|
        MOVL      *-SP[6],ACC           ; |10|
        SUBB      SP,#6
        LRETR
        ; return occurs
```

6.8.4　FUNC_EXT_CALLED プラグマ

-pm オプションを指定すると、コンパイラはプログラムレベルの最適化を使用します。このような最適化を使用した場合、コンパイラは main から直接または間接に呼び出されない関数を除去します。main ではなく、手書きのアセンブリ・コードにより呼び出される C 関数が存在する場合もあります。

FUNC_EXT_CALLED プラグマはオプティマイザに対して、それらの C 関数、またはそれらの C 関数によって呼び出される他の関数を保持しておくように指示します。それらの関数は C へのエントリ・ポイントとして機能します。

このプラグマは必ず、保持しておく必要がある関数の宣言や参照より前に置かなければなりません。

C の構文は次のとおりです。

#pragma FUNC_EXT_CALLED (*func*)

C++ の構文は次のとおりです。

#pragma FUNC_EXT_CALLED

引数 *func* は削除しない C 関数の名前です。

プログラムレベルの最適化を使用する場合、FUNC_EXT_CALLED プラグマと特定のオプションを組み合わせて使用しなければならない場合があります。詳細は、3.3.2 項「C/C++ とアセンブリを組み合わせた場合の最適化に関する注意事項」(3-8 ページ) を参照してください。

6.8.5 INTERRUPT プラグマ

INTERRUPT プラグマを使用すると、C/C++ コードで割り込みを直接的に処理することができます。C では、引数 *func* は関数の名前です。C++ では、このプラグマは次に宣言される関数に適用されます。C/C++ の割り込み関数の詳細は、6.7.4 項「interrupt キーワード」(6-16 ページ) を参照してください。

C の構文は次のとおりです。

```
#pragma INTERRUPT (func)
```

C++ の構文は次のとおりです。

```
#pragma INTERRUPT
```

6.9 静的変数とグローバル変数の初期化方法

ANSI/ISO C 規格では、明示的に初期化されていないグローバル（外部）変数および静的変数は、プログラムが実行を開始する前に 0 に初期化しておかなければならないと規定されています。この作業は、通常はプログラムのロード時に行われます。ロード処理はターゲット・アプリケーション・システム固有の環境に大きく依存するので、コンパイラ自体は、実行時に変数を事前に初期化しません。この要件を満たすのはアプリケーションです。

6.9.1 リンカを使った静的変数とグローバル変数の初期化方法

使用しているローダが変数を事前に初期化しない場合は、リンカでオブジェクト・ファイル内の変数を事前に 0 に初期化できます。たとえば、リンカ・コマンド・ファイルでは、.bss セクションに埋め込む値として 0 を使用してください。

```
SECTIONS
   {
       ...
       .bss: {} = 0x00;
       newvars: {} = 0x00;
       ...
   }
```

リンカは 0 で初期化された .bss セクションの完全なロード・イメージを出力 COFF ファイルに書き込むので、この方法では出力ファイルのサイズが大幅に増大するという望ましくない影響が発生する可能性があります。

アプリケーションを ROM に組み込む場合は、初期化が必要な変数を明示的に初期化する必要があります。前述の方法では .bss はロード時にのみ 0 に初期化され、システム・リセット時や電源投入時には初期化されません。これらの変数を実行時に 0 にするには、コードの中で明示的に定義してください。

リンカ・コマンド・ファイルおよび SECTIONS 疑似命令の詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』の「リンカの説明」の章を参照してください。

6.9.2 const 型修飾子を使った静的変数とグローバル変数の初期化方法

明示的に初期化されていない *const* 型の静的変数およびグローバル変数は、他の静的変数およびグローバル変数に似ています。これらは 0 に事前初期化されないためです（6.9 節の説明と同じ理由により）。たとえば次のとおりです。

```
const int zero;        /* may not be initialized to 0        */
```

しかし、const グローバル変数および静的変数は .const と呼ばれるセクションで宣言および初期化されているため、初期化方法が異なります。たとえば次のとおりです。

```
const int zero = 0;    /* guaranteed to be 0 */
```

これは .const セクションのエントリに対応します。

```
        .sect     .const
_zero
        .word     0
```

この機能は、大きな定数テーブルを宣言する場合に特に便利です。システム起動時に時間もスペースも無駄にならず、テーブルを初期化できるからです。さらに、リンカを使用して *.const* セクションを ROM に配置することも可能です。

上記の説明は、C での far 変数にも適用されます。ただし、そのような変数は .econst セクションに配置されるという点を除きます。たとえば次のとおりです。

```
   far const int zero=0;
```

DATA_SECTION プラグマを使うと、.const 以外のセクションに変数を格納できます。たとえば、次の C コードを参照してください。

```
#pragma DATA_SECTION (var, ".mysect");
   const int zero=0;
```

上記のコードは、次のアセンブリ・コードにコンパイルされます。

```
   .sect .mysect
_zero
   .word 0
```

6.10 リンク名の生成方法

コンパイラは、外部から見える識別子の名前をリンク名の作成時に変換します。使用されるアルゴリズムは、識別子が宣言される有効範囲により異なります。オブジェクトや関数の場合、下線（_）が識別子名の前に付きます。C++ 関数も先頭の文字に下線（_）が付きますが、関数名はさらにマングルされます。Mangling（マングリング）とは、関数のシグニチャ（そのパラメータの数と型）をその名前に組み込むプロセスです。使用されるマングリング・アルゴリズムは、『The C++ Annotated Reference Manual』（ARM）で説明されるアルゴリズムに準拠しています。マングリングにより、関数の多重定義（オーバーロード）、演算子の多重定義、および型の安全なリンクが可能になります。

func 関数に対する C++ リンク名の一般的な形式は次のとおりです。

___func__F*parmcodes*

ここで *parmcodes* は、func のパラメータ型をエンコードする一連の文字です。

次の単純な C++ ソース・ファイルがあるとします。

```
int foo(int i);  //global C++ function
{
return i;
}
```

その結果、foo 関数のアセンブリ・シンボル定義は次のようになります。

```
___foo__Fi;
```

foo のリンク名は ___foo__Fi で、foo が int 型の単独の引数を取る関数であることを示しています。検査およびデバッグができるように、名前を元の C++ ソースで検出された名前にデマングリングするネーム・デマングリング・ユーティリティが用意されています（第 10 章「C++ネーム・デマングラ・ユーティリティ」を参照）。

6.11 K&R C との互換性

ANSI/ISO C 言語は、『The C Programming Language』で規定されている事実上の標準 C 規格のスーパーセットです。ANSI/ISO 対応でない他のコンパイラ用に書かれたプログラムもほとんどは、修正なしで正しくコンパイルされ、実行されます。

ただし、C 言語には既存のコードに影響を与える微妙な変更が行われています。『The C Programming Language』の第 2 版（本書で K&R と呼ばれているもの）の付録 C には、ANSI/ISO C と初版の C 規格（本書で K&R C と呼ばれているもの）との違いがまとめられています。

TMS320C28x ANSI/ISO C コンパイラで、既存の C プログラムを簡単にコンパイルできるようにするために、コンパイラには K&R オプション（-pk）が付いています。このオプションにより言語の意味上の規則を一部変更し、古いコードとの互換性に対応しています。一般には、-pk オプションの目的は、K&R C よりも厳しくなっている ANSI/ISO C の規則を緩和することです。-pk オプションを指定することにより、関数のプロトタイプ、列挙法、初期化、あるいはプリプロセッサ構成要素などの C 言語の新しい機能が使用できなくなることはありません。-pk は、どの機能も無効にせずに ANSI/ISO 規則を緩和するだけのオプションです。

ANSI/ISO C と K&R C とで特に異なる点を以下に示します。

❑ 符号なし型をよりワイドな符号付き型に拡張することについて、整数拡張規則が変更になりました。K&R C では結果の型はワイドな型の符号なしバージョンであり、ANSI/ISO では結果の型はワイドな型の符号付きバージョンでした。これが、符号付きオペランドまたは符号なしオペランドに適用されるときに動作が異なる演算（つまり、比較、除算（および 剰余）、および右シフト）に影響を与えます。

```
unsigned short u;
int i;
if (u<i) .../* SIGNED comparison, unless -pk used */
```

❑ ANSI/ISO では、型の異なる 2 つのポインタは 1 つの演算では同時に使用できません。これに対して、ほとんどの K&R コンパイラでは警告が発せられるだけです。-pk を使用してもそのような状況の診断は行われますが、より緩和された条件の下で行われます。

```
int *p;
char *q = p;/* error without -pk, warning with -pk */
```

-pk を指定しなくとも、この規則に違反すると、code-E（回復可能）エラーになります。-pe オプションは、code-E エラーを警告に変換するので、-pk の代わりに使用できます。

- 型や記憶クラスなし（識別子のみ）で外部宣言を行うことを ANSI/ISO では禁じていますが、K&R では認めています。

```
a;                  /* illegal unless -pk used */
```

- ANSI/ISO では、初期化指定子のないファイル・スコープの定義を*仮定義*と解釈します。単独モジュールでは、この形式の複数の定義は 1 つの定義にまとめられます。K&R では、各定義が個々の定義として処理され、同じオブジェクトに対する複数の定義が生成されるので、通常はエラーとなります。たとえば次のとおりです。

```
int a;
int a;              /* illegal if -pk used, OK if not */
```

ANSI/ISO では、この 2 つの宣言はオブジェクト a の 1 つの定義になります。int の a が 2 回定義されるので、ほとんどの K&R コンパイラでは、このシーケンスは不正となります。

- ANSI/ISO では禁止していますが、K&R では外部リンクのあるオブジェクトを静的として再宣言できます。

```
extern int a;
static int a;       /* illegal unless -pk used */
```

- 文字列定数と文字定数内の認識できなかったエスケープ・シーケンスは ANSI/ISO では明らかに不正ですが、K&R では無視されます。

```
charc='\q';         /*same as 'q' if -pk used, error
                      if not */
```

- ANSI/ISO では、ビット・フィールドを int 型または unsigned 型とします。-pk を指定すると、ビット・フィールドをどの整数型でも正当に宣言できます。たとえば次のとおりです。

```
struct s
{
   short f : 2;     /* illegal unless -pk used */
};
```

- K&R 構文では、列挙型定数リスト内にカンマを続けることができます。

```
enum { a, b, c, }; /* illegal unless -pk used */
```

- K&R 構文では、プリプロセッサ疑似命令の後にトークンを続けることができます。

```
#endif NAME         /* warning unless -pk used */
```

6.12 言語モードの変更方法（-pk、-pr、および -ps オプション）

-pk、-pr、および -ps オプションを使用すると、C/C++ コンパイラがソース・コードを解釈する方法を指定できます。ソース・コードは、次のモードでコンパイルできます。

- ❑ 標準 ANSI/ISO モード
- ❑ K&R C モード
- ❑ 緩和 ANSI/ISO モード
- ❑ 厳密 ANSI/ISO モード

デフォルトは、標準 ANSI/ISO モードです。標準 ANSI/ISO モードでは、大部分の ANSI/ISO 違反にエラーが発行されます。しかし、厳密な ANSI/ISO 違反（厳密な ANSI/ISO 解釈では違反であるが、C/C++ コンパイラによって通常受け入れられるイディオムや許容値）には警告が発行されます。言語拡張機能は、ANSI/ISO C と矛盾するものであっても有効です。

C++ コードでは、ANSI/ISO モードは ANSI/ISO/IEC 14882-1998 規格を指定します。K&R C モードは、C++ コードには適用されません。

6.13 厳密 ANSI/ISO モードと緩和 ANSI/ISO モードの有効化（-ps および -pr オプション）

厳密 ANSI/ISO モードでコンパイルする場合は、-ps オプションを指定してください。このモードでは、ANSI/ISO に対応していない機能を使用するとエラー・メッセージが生成され、厳密な規格合致プログラムを無効にする言語拡張機能が使用不可になります。こうした拡張機能の例は、inline キーワードと asm キーワードです。

警告（標準 ANSI/ISO モードで発生する）またはエラー・メッセージ（厳密 ANSI/ISO モードで発生する）を発するのではなく、コンパイラに厳密な ANSI/ISO 違反を無視させる場合には -pr オプションを指定してください。緩和 ANSI/ISO モードでは、コンパイラは、ANSI/ISO C 規格の拡張機能が ANSI/ISO C と対立する場合であってもそれを受け入れます。

6.14 コンパイラの限界

C28x C/C++ コンパイラがさまざまなホスト・システムをサポートし、これらのシステムの一部には制限があるため、コンパイラは極端に大きいまたは複雑なソース・ファイルを正しくコンパイルできない場合があります。一般的に、そのようなシステムの制限を越えると、コンパイルを続行できなくなるため、コンパイラはエラー・メッセージを出力した直後に終了します。システムの制限を越えないようにするために、プログラムを簡略化してください。

一部のシステムでは、500 文字を超えるファイル名を許可していません。ファイル名は必ず 500 文字より短くしてください。

コンパイラには曖昧な制限はありませんが、ホスト・システムで使用可能なメモリ容量により制限があります。PC のような小さなホスト・システムでは、オプティマイザがメモリ不足になる場合があります。その場合、オプティマイザは停止し、シェルがコード・ジェネレータでファイルのコンパイルを続けます。その結果、ファイルが最適化されずにコンパイルされます。オプティマイザは一度に 1 つの関数をコンパイルするため、この現象の原因の大半はソース・モジュール内の大きな関数や極端に複雑な関数です。この問題を修正するには、オプションを以下のようにします。

- 疑わしいモジュールを最適化しない。
- 問題を発生させた関数を識別し、小さな関数に分解する。
- モジュールから関数を抽出し、最適化せずにコンパイルできる独立したモジュールに配置して、残りの関数を最適化できるようにする。

第 7 章

ランタイム環境

本章では、TMS320C28x™ C/C++ ランタイム環境について説明します。C/C++ プログラムを正しく実行するには、すべてのランタイム・コードが、この環境を維持する必要があります。適切な環境がないと、コードの信頼性は低くなります。また、C/C++ コードとインターフェイスするアセンブリ言語関数を記述する場合にも、本章の指示に従ってください。

項目 **ページ**

7.1	メモリ・モデル	7-2
7.2	レジスタ規則	7-11
7.3	関数呼び出し規則	7-13
7.4	アセンブリ言語と C/C++ 言語間のインターフェイス	7-19
7.5	割り込み処理	7-35
7.6	整数式の分析	7-37
7.7	浮動小数点式の分析	7-39
7.8	システムの初期化	7-40

7.1 メモリ・モデル

TMS320C28x は、メモリをプログラム・メモリとデータ・メモリの 2 つの連続したブロックとして扱います。

- **プログラム・メモリ**には、実行可能なコード、初期化レコード、およびスイッチ・テーブルが含まれます。
- **データ・メモリ**には、外部変数、静的変数、およびシステム・スタックが含まれます。

C/C++ プログラムによって生成されたコード・ブロックとデータ・ブロックは、適切なメモリ空間の連続したブロックに配置されます。

注： リンカのメモリ・マップ定義

コンパイラではなくリンカがメモリ・マップを定義し、コードおよびデータをターゲット・メモリに割り当てます。コンパイラは、使用できるメモリの型について、使用できない位置（ホール）や入出力（I/O）または制御のために確保されている位置については、何も想定していません。

コンパイラは、リンカがコードおよびデータを適切なメモリ空間に割り当てることが可能な再配置可能なコードを作成します。たとえば、リンカを使用して、グローバル変数を高速な内部 RAM に割り当てたり、実行可能なコードを内部 ROM に割り当てたりできます。各コード・ブロックまたはデータ・ブロックをそれぞれメモリに割り当てることができますが、これは一般的な方法ではありません（この例外はメモリ・マップド I/O です。これを用いると、C ポインタ型で物理メモリ位置にアクセスできます）。

7.1.1 セクション

コンパイラは、コードとデータが入った複数の再配置可能なブロックを作成します。これらのブロックは *セクション*と呼ばれ、さまざまなシステム構成に整合するように、さまざまな方法でメモリ内に割り振られます。セクションの詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

セクションの基本型は 2 つあります。

❑ 初期化されたセクションには、データ・テーブルおよび実行可能なコードが含まれます。C コンパイラは初期化されたセクション（.text、.cinit、.const、.econst、.pinit、.switch）を作成します。

- **.text** セクションは、定数と同様に実行可能なコードをすべて含む初期化されたセクションです。
- **.cinit** セクションと **.pinit** セクションは、変数および定数を初期化するためのテーブルを含みます。
- **.const** セクションは、文字列定数や明示的に初期化される（*const* で修飾された）グローバル変数と静的変数の宣言および初期化を含む初期化されたセクションです。
- **.econst** セクションは、文字列定数や明示的に初期化され、far メモリに配置される（*far const* かラージ・メモリ・モデルの使用で修飾された）グローバル変数と静的変数の宣言および初期化を含む初期化されたセクションです。
- **.switch** セクションは、スイッチ文のためのテーブルを含む初期化されたセクションです。

❑ 初期化されないセクションは、メモリ（通常 RAM）にスペースを確保します。プログラムは、実行時にこの空間を変数の作成と格納に使用できます。コンパイラは、5 つの初期化されないセクション（.bss、.ebss、.stack、.sysmem、.esysmem）を作成します。

- **.bss** セクションは、グローバル変数および静的変数のためのスペースを確保する初期化されないセクションです。プログラム起動時に、C ブート・ルーチンは（ROM 内に存在する）.cinit セクションからデータをコピーし、.bss セクションにそのデータを格納します。
- **.ebss** セクションは、far 宣言された（C のみ）、またはラージ・メモリ・モデル使用時に宣言されたグローバル変数と静的変数のスペースを確保する初期化されないセクションです。プログラム起動時に、C ブート・ルーチンは（ROM 内に存在する）.cinit セクションからデータをコピーし、.ebss セクションにそのデータを格納します。
- **.stack** セクションは、C システム・スタックのために使われる初期化されないセクションです。このメモリは、関数に引数を渡し、ローカル変数のためのスペースを割り当てるために使用します。

- **.sysmem セクション**は、動的メモリ割り当てのためにスペースを確保する初期化されないセクションです。確保された空間は、malloc 関数が使用します。malloc 関数を使用しない場合、セクションのサイズは 0 のままになります。
- **.esysmem セクション**は、動的メモリ割り当てのためにスペースを確保する初期化されないセクションです。確保された空間は、far malloc 関数が使用します。far malloc 関数を使用しない場合、セクションのサイズは 0 のままになります。

リンカは出力セクションを作成するために、さまざまなモジュールから個々のセクションを取得し、同じ名前のセクションを結合させます。完全なプログラムは、これらの出力セクションから構成されています。これらの出力セクションは、システムの仕様に合わせて、アドレス空間の任意の場所に配置できます。

.text、.cinit、switch の各セクションは、通常 ROM または RAM のいずれかにリンクされ、プログラム・メモリ（ページ 0）内に配置されなければなりません。.const セクションは、ROM または RAM のいずれかにリンクされますが、データ・メモリ（ページ 1）内に配置されなければなりません。.bss/.ebss、.stack、.sysmem/.esysmem の各セクションは、RAM にリンクされなければならず、データ・メモリ内に配置されなければなりません。次の表に、それぞれのセクションのタイプが必要とするメモリの型およびページ宛先を示します。

セクション	メモリの型	ページ
.text	ROM または RAM	0
.cinit	ROM または RAM	0
.pinit	ROM または RAM	0
.switch	ROM または RAM	0、1
.const	ROM または RAM	1
.econst	ROM または RAM	1
.bss	RAM	1
.ebss	RAM	1
.stack	RAM	1
.sysmem	RAM	1
.esysmem	RAM	1

メモリへのセクション割り当て方法の詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』の「COFF 情報の概要」を参照してください。

7.1.2　C/C++ システム・スタック

C/C++ コンパイラは、スタックを使用して以下の作業を実行します。

- ローカル変数を割り当てる
- 関数に引数を渡す
- プロセッサ・ステータスを保存する
- 関数の戻りアドレスを保存する
- 一時的な結果を保存する

ランタイム・スタックは、下位のアドレスから上位のアドレスへと増大します。デフォルトでは、このスタックは .stack セクションに割り当てられます（rts ファイル boot.asm を参照）。コンパイラは、ハードウェア・スタック・ポインタ（SP）を使用してスタックを管理します。

注：　.stack セクションのリンク方法

.stack セクションは、データ・メモリの下位 64K にリンクする必要があります。SP は 16 ビットのレジスタで、64K を超えるアドレスをアクセスすることができません。

サイズが 63 ワードを超える（SP オフセット・アドレッシング・モードの最大値に達する）フレームの場合、コンパイラはフレーム・ポインタ（FP）として XAR2 を使います。関数を呼び出すと、新しいフレームがスタックの一番上に作成され、ローカル変数と一時変数はそこから割り当てられます。FP がこのフレームの先頭を指し示すことで、SP の使用では直接参照できないメモリ位置にアクセスできるようになります。

スタック・サイズはリンカによって設定されます。またリンカはグローバル・シンボル __STACK_SIZE も作成し、このシンボルにスタック・サイズ（バイト数）を割り当てます。デフォルトのスタック・サイズは、1K ワードです。スタック・サイズは -stack リンカ・コマンド・オプションを指定することにより、リンク時に変更できます。

注：　スタック・オーバーフロー

コンパイラには、コンパイル時または実行時にスタック・オーバーフローをチェックする方法がありません。スタック・オーバーフローによりランタイム環境が損なわれ、プログラムが異常終了する恐れがあります。スタックの増大に備えて十分なスペースを割り当てるようにします。

7.1.3 .const/.econst をプログラム・メモリに割り当てる方法

システム構成が .const/.econst などの初期化されたセクションのデータ・メモリへの割り当てをサポートしない場合には、.const/.econst セクションをプログラム・メモリにロードし、データ・メモリ内で実行させる必要があります。ブート時に、プログラム・メモリから .const/.econst セクションをデータ・メモリにコピーします。この作業を実行する手順は次のとおりです。

ブート・ルーチンを変更します。

1) ソース・ライブラリから boot.asm を抽出します。

   ar2000 -x rts.src boot.asm

2) boot.asm を編集し、CONST_COPY フラグを 1 に変更します。

```
CONST_COPY .set  1
```

3) boot.asm をアセンブルします。

   cl2000 -v28 boot.asm

4) ブート・ルーチンをオブジェクト・ライブラリにアーカイブします。

   ar2000 -r rts2800.lib boot.obj

.const セクションの場合、次のエントリを含むリンカ・コマンド・ファイルを使ってリンクします。

```
MEMORY{
   PAGE 0 : PROG : ...
   PAGE 1 : DATA : ...
}

SECTIONS
{
   ...
   .const  : load = PROG PAGE 1, run = DATA PAGE 1
             {
                /* GET RUN ADDRESS   */
                 __const_run = .;
                /* MARK LOAD ADDRESS */
                 *(.c_mark)
                /* ALLOCATE .const   */
                 *(.const)
                /* COMPUTE LENGTH    */
                 __const_length = .- __const_run;
              }
   ...
}
```

同様に、.econst セクションの場合、次のエントリを含むリンカ・コマンド・ファイルを使ってリンクします。

```
SECTIONS
{
   ...
   .econst   : load = PROG PAGE 1, run = DATA PAGE 1
               {
                  /* GET RUN ADDRESS    */
                   __econst_run = .;
                  /* MARK LOAD ADDRESS */
                   *(.ec_mark)
                  /* ALLOCATE .econst    */
                   *(.econst)
                  /* COMPUTE LENGTH     */
                   __econst_length = - .__econst_run;
                }
   ...
}
```

リンカ・コマンド・ファイル内で、PROG という名前をページ 0 上のメモリ領域の名前と、DATA という名前をページ 1 上のメモリ領域の名前とそれぞれ置き換えることができます。コマンド・ファイルの残りは、上記の名前を使う必要があります。CONST_COPY を 1 に変更した場合、有効にされた boot.asm 内のコードは、このようにこれらの名前を使用するリンカ・コマンド・ファイルに依存します。いずれかの名前を変更するには、boot.asm を編集し、同じように名前を変更します。

7.1.4 動的なメモリ割り当て

コンパイラ付属のランタイムサポート・ライブラリには、実行時にメモリを変数に動的に割り当てることができる関数（malloc、calloc、realloc など）が含まれています。これは大きなメモリ・プール（ヒープ）を宣言し、かつこれらの関数を使用してヒープからメモリを割り当てることにより行われます。動的な割り当ては、C 言語の標準的なものではなく、標準ランタイムサポート関数により提供されます。

このメモリ・プール（ヒープ）は、リンカによって作成されます。またリンカはグローバル・シンボル __SYSMEM_SIZE も作成し、このシンボルにヒープ・サイズ（ワード数）を割り当てます。デフォルトのヒープ・サイズは、1K ワードです。リンク時にメモリ・プールのサイズを変更するには、-heap オプションを指定します。リンカ・コマンド行で -heap オプションの後にメモリ・プールのサイズを定数として指定してください。

また far メモリ・プール（far ヒープ）も各種 far ランタイムサポート・ライブラリ関数（far_malloc、far_calloc、far_realloc）を通して使用できます。far ヒープは、リンカによって作成されます。またリンカはグローバル・シンボル __FAR_SYSMEM_SIZE も作成し、このシンボルに far ヒープ・サイズ（ワード数）を割り当てます。デフォルトのサイズは、1 K ワードです。リンク時に far メモリ・プールのサイズを変更するには、-farheap オプションを指定します。リンカ・コマンド行で -farheap オプションの後にメモリ・プールのサイズを定数として指定してください。

注：　ヒープ・サイズの制約事項

near ヒープを実装する場合、ヒープ・サイズが 32 K ワードという制限があります。この制限は、far ヒープには適用されません。

7.1.5　変数の初期化

C/C++ コンパイラは、ROM ベース・システムのファームウェアでの使用に適したコードを作成します。このようなシステムでは、（グローバル変数と静的変数の初期化のために使用される）.cinit セクション内の初期化テーブルは ROM に格納されます。システムの初期化時に C/C++ ブート・ルーチンにより、（ROM 内の）これらのテーブルのデータは .bss（RAM）内の初期化された変数にコピーされます。

プログラムをオブジェクト・ファイルからメモリに直接ロードして実行するような場合は、メモリ内の空間が .cinit セクションで占有されるのを防ぐことができます。ローダは、（実行時ではなく）ロード時に初期化テーブルを（ROM からではなく）オブジェクト・ファイルから直接読み取り、初期化を直接実施できます。-cr リンカ・オプションを使用して、これをリンカに指定できます。システムの初期化の詳細は、7.8 節「システムの初期化」（7-40 ページ）を参照してください。

7.1.6　静的変数とグローバル変数へのメモリ割り当て方法

C/C++ プログラムで宣言されたすべての外部変数または静的変数に、固有の連続した空間が割り当てられます。リンカは空間のアドレスを決定します。コンパイラは、これらの変数の空間が複数のワードに割り当てられるようにします。それにより、各変数はワード境界に割り当てられます。

C/C++ コンパイラはグローバル変数がデータ・メモリに割り当てられることを期待します（.bss にそのための空間を確保します）。同じモジュールで宣言された変数は、単独の連続したメモリ・ブロックに割り当てられます。

7.1.7 フィールド / 構造体の位置合わせ

コンパイラが構造体にスペースを割り当てる場合、構造体のメンバすべてを保持するのに必要なだけのワードを割り当て、各メンバの位置合わせ上の制約に従います。

非フィールド型はすべてワード境界に位置合わせされます。フィールドには必要なだけのビット数が割り当てられます。隣接したフィールドは、1 つのワードの隣接したビットに埋め込まれます。しかし複数のワードにまたがることはありません。フィールドが次のワードにまたがってしまう場合は、フィールド全体が次のワードに配置されます。フィールドは出現順にパックされます。構造体ワードの最下位ビットが最初に埋められます。

7.1.8 文字列定数

C/C++ では、文字列定数の使用方法には次の 2 通りがあります。

❑ 文字列定数は、文字配列を初期化できます。たとえば次のとおりです。

```
char s[] = "abc";
```

文字列は初期化指定子として使用されると、単に初期化された配列として扱われます。個々の文字は独立した初期化指定子となります。詳細は、7.8 節「システムの初期化」（7-40 ページ）を参照してください。

❑ 文字列定数は、式の中で使用できます。たとえば次のとおりです。

```
strcpy (s, "abc");
```

文字列を式の中で使用すると、文字列自体は、その文字列（終了を示す 0 バイトも含む）を指す一意のラベルとともに .const セクションに .string アセンブラ疑似命令で定義されます。たとえば次のコードでは、文字列 abc および終了バイトを定義します。ラベル SL5 はその文字列を指します。

```
      .const
SL5   .string   "abc", 0
```

文字列ラベルの形式は SL*n* です。*n* はコンパイラが割り当てる番号であり、これによりラベルは一意になります。これらの番号は、0 から順に 1 ずつ増分して各文字列を定義します。ソース・モジュールで使用する文字列の定義は、すべてコンパイラ出力のアセンブリ言語モジュールの最後に配置されます。

ラベル SL*n* は文字列定数のアドレスを表します。コンパイラは、このラベルにより式の文字列を参照します。

同じ文字列がソース・モジュール内で複数使われていても、メモリ内で文字列の重複はありません。同じ文字列定数を使用することはすべて、文字列の 1 つの定義を共有します。

文字列は .const セクション（ほとんどの場合 ROM）内に格納され、共用される場合があるので、プログラムで文字列定数を修正することはお勧めできません。次のコードは、文字列の誤った使用例です。

```
char *a = "abc";
a[1] = 'x';             /* Incorrect! */
```

7.1.9 far 文字列定数

C では、文字列定数は .econst セクションに配置することができます。文字ポインタを初期化する場合、または式の中で文字を使用する場合、far キーワードを使用します。たとえば次のとおりです。

```
far char *ptr = “abc”;
far_strcpy (s, (far char *) “abc”);
```

far 型の文字列定数は、7.1.8 項「文字列定数」（7-9 ページ）に記述されている方法と同じように .econst セクションに配置されます。far 型の文字列ラベルは、FSLn 形式です。またこの方法は、ラージ・メモリ・モデルの場合にも使われます。

7.2 レジスタ規則

C/C++ 環境では、特定のレジスタと特定の操作は、厳密な規則で関連付けられています。アセンブリ言語ルーチンを C/C++ プログラムにインターフェイスする場合は、これらのレジスタ規則に従ってください。

レジスタ規則は、コンパイラがレジスタをどのように使用するか、および関数呼び出しの際にどのように値が保存されるかを記述したものです。エントリ時保存レジスタと呼び出し時保存レジスタという 2 種類のレジスタ変数があります。この 2 種類のレジスタ変数の違いは、関数呼び出しの際の保存方法です。レジスタの値を保存する必要がある場合に、エントリ時に保存されたレジスタを保持するのは呼び出し先の関数の役割であり、呼び出し時に保存されたレジスタを保持するのは呼び出し元の関数の役割です。

7.2.1 TMS320C28x レジスタの使用と保存

表 7-1 に、コンパイラがどのように TMS320C28x レジスタを使用しているか、また関数呼び出しの際にどのレジスタを保持するように定義しているかをまとめました。

表 7-1. レジスタの使用および保存に関する規則

レジスタ	使用方法	エントリ時に保存	呼び出し時に保存
AL	式、引数渡し。関数から 16 ビット演算結果を戻す	不要	要
AH	式および引数渡し	不要	要
XAR0	ポインタおよび式	不要	要
XAR1	ポインタおよび式	要	不要
XAR2	ポインタ、式、およびフレーム・ポインタ（必要に応じて）	要	不要
XAR3	ポインタおよび式	要	不要
XAR4	ポインタ、式、引数渡し。関数から 16 ビットおよび 22 ビットのポインタ値を戻す	不要	要
XAR5	ポインタ、式、および引数	不要	要
XAR6	ポインタおよび式	不要	要
XAR7	ポインタ、式、間接呼び出し、および間接分岐（関数へのポインタおよび switch 文を実装するために使われる）	不要	要

表 7-1.　レジスタの使用および保存に関する規則（続き）

レジスタ	使用方法	エントリ時に保存	呼び出し時に保存
SP	スタック・ポインタ	†	†
T	乗算式およびシフト式	不要	要
TL	乗算式およびシフト式	不要	要
PL	乗算式および一時変数	不要	要
PH	乗算式および一時変数	不要	要
DP	データ・ページ・ポインタ（グローバル変数をアクセスするために使われる）	不要	不要

† SP は、スタックにプッシュしたすべてのものは戻る前にポップするという規則により保存されます。

7.2.2　ステータス・レジスタ

表 7-2 に、コンパイラが使用するステータス・フィールドをすべて示します。推定値は、コンパイラが関数にエントリ時または関数からの復帰時に該当フィールドで期待する値です。この欄のダッシュ（-）は、コンパイラが特定の値を期待しないことを示します。変更の欄は、コンパイラの生成したコードがこのフィールドを変更したかどうかを示します。

表 7-2.　ステータス・レジスタ・フィールド

フィールド	名前	推定値	変更
C	キャリー	-	あり
N	負のフラグ	-	あり
OVM	オーバーフロー・モード	0†	なし
ページ 0	直接 / スタック・アドレス・モード	0†	なし
PM	積シフト・モード	1†	あり
SPA	スタック・ポインタ・アライン・ビット	-	あり（割り込み時）
SXM	符号拡張モード	-	あり
TC	テスト / 制御フラグ	-	あり
V	オーバーフロー・フラグ	-	あり
Z	ゼロ・フラグ	-	あり

† C ランタイム環境を設定する初期化ルーチンは、これらのフィールドを推定値に設定します。

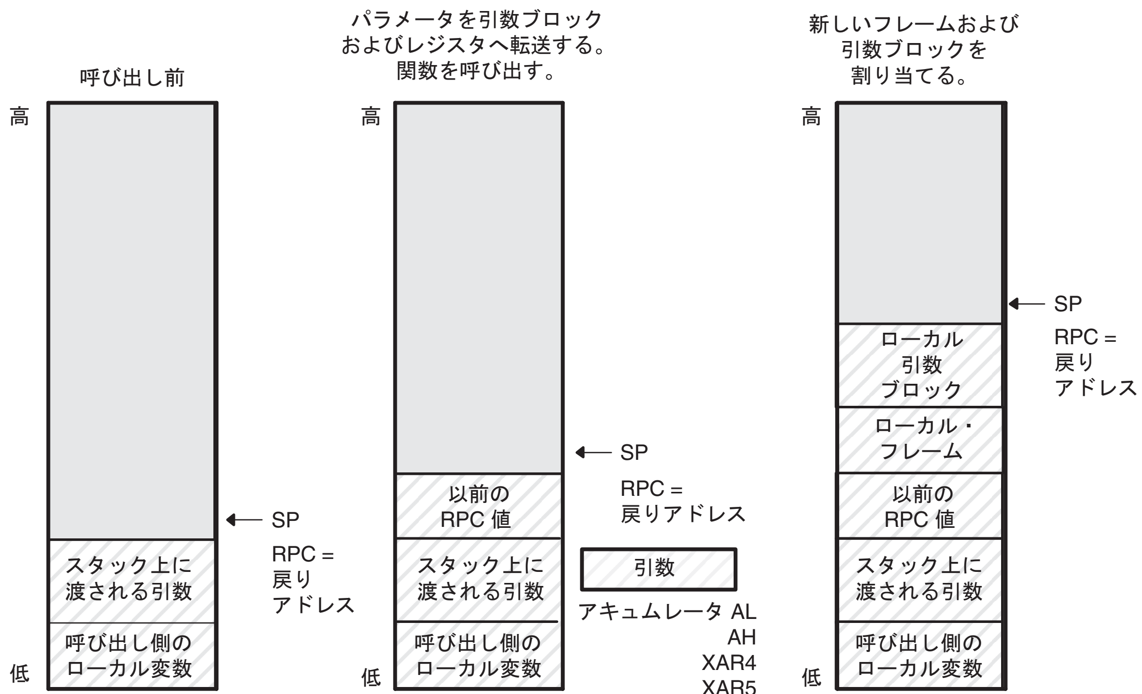
7.3 関数呼び出し規則

C/C++ コンパイラでは、関数呼び出しに対して一連の厳格な規則を適用します。特別なランタイムサポート関数を除き、C/C++ 関数を呼び出すか C/C++ 関数によって呼び出されるすべての関数は、以下の規則に従わなければなりません。これらの規則に従わない場合は C/C++ 環境が損なわれ、プログラムが異常終了する恐れがあります。

図 7-1 に、典型的な関数の呼び出しを示します。この例では、レジスタ内に配置できないパラメータはスタック上の関数に渡されます。その後、関数はローカル変数を割り当て、別の関数を呼び出します。この例では、割り当てられたローカル・フレームおよび呼び出された関数の引数ブロックを示しています。ローカル変数をもたず、かつ引数ブロックを必要としない関数は、ローカル・フレームを割り当てません。

*引数ブロック*とは、他の関数に引数を渡すために使われるローカル・フレームの部分を指します。パラメータは、スタックにプッシュされるのではなく引数ブロックの中へ転送され、関数に渡されます。ローカル・フレームおよび引数ブロックは同時に割り当てられます。

図 7-1. 関数呼び出し時のスタックの使用



7.3.1 関数の呼び出し方法

呼び出し側の関数は、別の関数を呼び出すときに次の作業を実行します。

1) 呼び出し先の関数が必ずしも値を保持しているとは限らないが関数の復帰後に必要なレジスタ（エントリ時に保存（SOE）されないレジスタ）は、スタック上に保存されます。

2) 呼び出し先の関数が構造体の場合、呼び出し元の関数が構造体のためのスペースを割り当て、そのスペースのアドレスを呼び出し先の関数に最初の引数として渡します。

3) 呼び出し先の関数に渡される引数は、レジスタに、また必要に応じてスタックに置かれます。

   以下に示す方針に従って、引数はレジスタに置かれます。

   a) 64 ビット整数引数（long long 型）がある場合、最初の引数は ACC および P に置かれます（ACC は上位 32 ビットを保持し、P は下位 32 ビットを保持します）。これ以外の 64 ビットの引数は、逆の順序でスタック上に置かれます。

      P レジスタが引数渡しのために使われる場合には、プロローグ / エピローグ抽象化は該当関数に対しては無効です。抽象化の詳細は、3.8 節「コード・サイズ最適化の強化（-ms オプション）」（3-18 ページ）を参照してください。

   b) 32 ビットの引数（long 型または float 型）がある場合、最初の引数は 32 ビットの ACC（AH/AL）に置かれます。これ以外の 32 ビットの引数は、逆の順序でスタック上に置かれます。

```
func1(long a, long long b, int c, int* d);
   stack   ACC/P        XAR5,  XAR4
```

   c) ポインタ引数は、XAR4 および XAR5 に置かれます。これ以外のポインタはすべてスタック上に置かれます。

   d) 残りの 16 ビットの引数は、使用できる場合には、AL、AH、XAR4、XAR5 の順に格納されます。

4) レジスタに格納されない残りの引数は、逆の順序でスタック上にプッシュされます。つまり、スタック上に置かれる左端の引数がスタックの最後にプッシュされます。すべての 32 ビットの引数は、スタック上の偶数アドレスに位置合わせされます。

   構造体の引数は、構造体のアドレスとして渡されます。呼び出し先の関数は、ローカル・コピーを作成する必要があります。

   省略符号を指定して宣言された関数の場合、可変個の引数を指定して関数が呼び出されたことを示し、規則はほとんど変更されません。最後に明示的に宣言された引数がスタックに渡されるので、そのスタック・アドレスは宣言されていない引数にアクセスする場合の参照の役目をします

5) スタック・ポインタ（SP）は、子関数を呼び出す前に親関数によって偶数に位置合わせされている必要があります。これを行うには、必要に応じてスタック・ポインタを 1 だけ増分します。コードを記述する場合、必要に応じて呼び出し前に SP を増分してください。

   引数が置かれる場所を示す関数呼び出しの例を以下に示します。

```
func1 (int a, int b, long c) ;
       XAR4   XAR5    AH/AL
func1 (long a, int b, long c) ;
       AH/AL   XAR4   stack
vararg (int a, int b, int c, ...)
        AL     AH     stack
```

6) 呼び出し側が LCR 命令を使用して関数を呼び出します。RPC レジスタ値は、スタック上にプッシュされます。その後、戻りアドレスは、RPC レジスタに格納されます。

7) スタックは関数境界に位置合わせされます。

7.3.2 呼び出し先の関数の対応方法

呼び出し先の関数は、以下の作業を実行します。

1) 呼び出し先の関数が XAR1、XAR2、XAR3 のいずれかを変更する場合、呼び出し先ではそのレジスタを保存する必要があります。これは呼び出し元の関数はこれらのレジスタの値が復帰時に保存されるということを想定しているからです。これら以外のレジスタでは、その値を保存せず変更される場合があります。

2) 呼び出し先の関数は、ローカル変数、一時記憶域、およびこの関数が呼び出す関数の引数用に十分なスペースをスタック上に割り当てます。この割り当ては、SP レジスタに定数を加算することにより、関数の先頭で一度行われます。

3) スタックは関数境界に位置合わせされます。

4) 呼び出し先の関数が構造体引数を要求する場合、関数は構造体を指すポインタを受け取ります。呼び出し先関数内で構造体への書き込みが行われる場合は、その構造体のローカル・コピー用のスペースをスタック上に割り当てなければなりません。また構造体のローカル・コピーは、渡されたポインタから作成する必要があります。構造体に書き込みが行われない場合は、ポインタ引数を通して間接的に呼び出し先関数で参照することができます。

   構造体引数を受け付ける関数を宣言する場合は、（構造体引数をアドレスとして渡されるように）その関数が呼び出される時点、および（その関数が構造体をローカル・コピーにコピーすることを認識するように）その関数が宣言される時点の両方で、正しく宣言するように注意が必要です。

5) 呼び出し先の関数は、関数のコードを実行します。

6) 呼び出し先の関数は値を戻します。次の規則を使用して、その値はレジスタに格納されます。

   16 ビット整数値： AL
   32 ビット整数値： ACC
   64 ビット整数値： ACC/P
   16 または 22 ビット・ポインタ： XAR4

   関数が構造体を戻す場合、呼び出し元は構造体用のスペースを割り当ててから、XAR4 内の呼び出し先の関数に戻りスペースのアドレスを渡します。構造体を戻すには、呼び出し先関数は、その構造体を追加の引数が指すメモリ・ブロックにコピーします。

   このように、呼び出し元が呼び出し先関数に構造体を戻す場所を指示するということは優れているといえます。たとえば、s= f(x) という文で、S は構造体、F は構造体を戻す関数とします。呼び出し元は、f(&s, x) として実際に呼び出しを行うことができます。関数 f は戻りの構造体を直接 s にコピーし、自動的に代入を行います。

   呼び出し元が戻りの構造体値を使用しない場合は、最初の引数としてアドレス値 0 を渡すことができます。この 0 は呼び出し先関数に対して、戻りの構造体をコピーしないように指示します。

   構造体を戻す関数を宣言する場合は、（さらに引数が渡されるように）その関数が呼び出される時点、および（その関数が結果をコピーすることを認識するように）宣言される時点の両方で、正しく宣言するように注意が必要です。64 ビットの浮動小数点値（long double 型）を戻すことは、構造体に戻すことと似ています。

7) 呼び出し先の関数は、SP に以前加算した値を減算することでフレームの割り当てを解除します。

8) 呼び出し先の関数は、ステップ 1 で保存したレジスタの値をすべて復元します。

9) 呼び出し先の関数は、LRETR 命令を使って値を戻します。PC は RPC レジスタ内の値に設定されます。以前の RPC 値は、スタックからポップされ、RPC レジスタに格納されます。

7.3.3　呼び出し先関数の特殊な場合（ラージ・フレーム）

スタックに割り当てられる必要があるスペース（前項のステップ 2）が 63 ワードより大きい場合、すべてのローカルの非レジスタ変数に確実にアクセスできるようにするには、ステップとリソースがさらに必要になります。ローカルの非レジスタ変数を参照するために、ラージ・フレームはフレーム・ポインタ・レジスタ（XAR2）を使用することを要求します。フレームにスペースを割り当てる前に、呼び出し先の関数に渡される、スタック上の最初の引数を指すようにフレーム・ポインタを設定します。渡されるはずの引数がスタック上に渡されない場合、フレーム・ポインタは呼び出し元の関数へのエントリ時にスタックの一番上にある呼び出し元の関数の戻りアドレスを指します。

大量のローカル・データを割り当てることは可能であれば避けてください。たとえば、大きな配列を関数内で宣言しないでください。

7.3.4　引数とローカル変数のアクセス方法

関数は、SP または FP（XAR2 に指定されているフレーム・ポインタ）のいずれかからローカルの非レジスタ変数およびスタック引数を間接的にアクセスします。SP を使ってアクセスできるすべてのローカル・データおよび引数のデータは、*-SP [offset] アドレッシング・モードを使用します。これは SP がスタックの一番上をすぎたデータを指すので、スタックがより大きなアドレスに向かって増大するからです。

注：　PAGE0 モード・ビットはリセットする必要がある

コンパイラは *-SP [offset] アドレッシング・モードを使うので、PAGE0 モード・ビットはリセット（0 に設定）する必要があります。

*-SP [offset] を使ったときに得られる最大のオフセットは 63 です。オブジェクトが SP から離れすぎているのでこのアクセス・モードを使用できない場合、コンパイラは FP (XAR2) を使用します。FP はフレームの最後を指すので、FP を使って行われるアクセスは *+FP [offset] または *+FP [AR0/AR1] のいずれかのアドレッシング・モードを使用します。ラージ・フレームは XAR2 および可能ならばインデックス・レジスタを利用することを要求するので、ローカル変数にアクセスするために余分なコードとリソースが必要になります。

7.3.5 フレームを割り当てる方法とメモリ内の 32 ビット値をアクセスする方法

TMS320C28x には、一度に 32 ビットのメモリを読み書きする命令があります（MOVL、ADDL など）。これらのメモリは、偶数境界上に 32 ビットのオブジェクトが割り当てられていることを要求します。これが確実に行われるようにするために、コンパイラは次の手順を実行します。

1) コンパイラは SP を偶数境界に初期化します。
2) 呼び出し命令が SP に 2 を加算するので、コンパイラは SP が偶数アドレスを指しているものと想定します。
3) フレームに割り当てられたスペースを合計すると確実に偶数になるので、SP は偶数アドレスを指します。
4) 32 ビット・オブジェクトが SP の偶数アドレスを基準にした既知の偶数アドレスに確実に割り当てられます。
5) 割り込みは SP が奇数か偶数か想定できないので、コンパイラは SP を偶数に位置合わせします。

これらの命令がメモリをアクセスする方法の詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

7.4 アセンブリ言語と C/C++ 言語間のインターフェイス

アセンブリ言語を C/C++ コードと組み合わせて使用する方法は、次のとおりです。

- アセンブルしたコードのモジュールを個別に使用し、それらのモジュールを、コンパイルした C/C++ モジュールとリンクします（7.4.1 項「C/C++ コードでのアセンブリ言語モジュールの使用方法」（7-19 ページ）を参照）。これは最も汎用性のある方法です。
- インライン・アセンブリ言語を直接 C/C++ ソース内に埋め込んで使用します（7.4.4 項「インライン・アセンブリ言語の使用方法」（7-25 ページ）を参照）。
- C/C++ ソースで組み込み関数を使用して、アセンブリ言語文を直接呼び出します（7.4.5 項「組み込み関数（intrinsics）を使用してアセンブリ言語文にアクセスする方法」（7-26 ページ）を参照）。

7.4.1 C/C++ コードでのアセンブリ言語モジュールの使用方法

7.3 節「関数呼び出し規則」（7-13 ページ）で規定された呼び出し規則、および 7.2 節「レジスタ規則」（7-11 ページ）で規定されたレジスタ規則に従っている場合には、アセンブリ言語関数と C/C++ とのインターフェイスを取ることは難しくありません。C/C++ コードからアセンブリ言語で定義された変数や呼び出し関数にアクセスでき、アセンブリ・コードからも C/C++ 変数にアクセスしたり C/C++ 関数を呼び出したりできます。

アセンブリ言語と C/C++ をインターフェイスするには、次の指針に従ってください。

- C/C++ 言語で記述される関数でもアセンブリ言語で記述される関数でもすべて、7.2 節「レジスタ規則」（7-11 ページ）で概要を説明している規則に従う必要があります。
- 関数によって変更される専用のレジスタは、保存する必要があります。専用のレジスタは次のとおりです。

  XAR1
XAR2
XAR3
SP

  通常 SP を使用する場合は、明示的に保存する必要はありません。スタック上にプッシュされるものがすべて関数から戻る前にポップされる限り（つまり SP の保存）、アセンブリ関数は自由にスタックを使用できます。

  専用でないレジスタは、すべて保存することなく自由に使用できます。
- スタック・ポインタ（SP）は、子関数を呼び出す前に親関数によって偶数に位置合わせされている必要があります。これを行うには、必要に応じてスタック・ポインタを 1 だけ増分します。コードを記述する場合、必要に応じて呼び出し前に SP を増分してください。

- スタックは関数境界に位置合わせされます。
- 割り込みルーチンは、使用するすべてのレジスタを保存しなければなりません（詳細は、7.5 節「割り込み処理」（7-35 ページ）を参照）。

- アセンブリ言語から C/C++ 関数を呼び出す場合、7.3.1 項「関数の呼び出し方法」（7-14 ページ）に記述されているように、引数を指定して特定のレジスタをロードし、残りの引数をスタック上にプッシュします。

  C/C++ 関数から渡された引数をアクセスする場合、これらの規則が同様に適用されます。

- C/C++ 呼び出し規則は RPC を使って、LCR および LRETR 命令から戻り値を格納するので、アセンブリ関数は同じ規則に従う必要があります。

- long および float は、最下位ワードが下位アドレスになるようにメモリに保存されます。

- 構造体は、7.3.2 項「呼び出し先の関数の対応方法」（7-15 ページ）に記述されているように値を戻さなければなりません。

- アセンブリ言語モジュールは、グローバル変数の自動初期化以外の目的に .cinit セクションを使用することができません。boot.asm の C/C++ 始動ルーチンは、.cinit セクションがすべて初期化テーブルで構成されているものと見なします。それ以外の情報を .cinit に入れるとテーブルが壊され、予期できない結果が生じます。

- コンパイラは、すべての識別子の先頭に下線（_）を付けます。アセンブリ言語モジュールでは、C/C++ からアクセス可能になっているすべてのオブジェクトに対して先頭に下線を付ける必要があります。たとえば、x という C/C++ オブジェクトは、アセンブリ言語では _x となります。アセンブリ言語モジュール（1 つまたは複数）の中でのみ使用される識別子の場合、下線で始まらない名前は C/C++ 識別子と競合することなく安全に使用することができます。

- アセンブリ言語内で宣言し C/C++ からアクセスまたは呼び出しが行われるオブジェクトや関数は、すべてアセンブラの中で .global または .def 疑似命令を使用して宣言しなければなりません。これにより、シンボルが外部シンボルとして定義され、リンカはそのシンボルへの参照を解決できます。

  同様に、アセンブリ言語から C/C++ 関数またはオブジェクトをアクセスするには、C/C++ オブジェクトを .global または .ref を使用して宣言します。これにより、リンカが解決できる未定義の外部参照が作成されます。

- コンパイルされたコードは PAGE0 モード･ビットをリセットして実行するので、アセンブリ言語関数で PAGE0 ビットを 1 に設定した場合は、コンパイルされたコードに戻る前にそのビットを 0 に設定し直す必要があります。

- ❑ アセンブリで構造体を定義し、extern struct を使って C でアクセスする場合は、構造体をブロックしてください。コンパイラは DP ロード命令を最適化するために、構造定義がブロックされているものと想定しています。したがって、定義ではこの前提を尊重してください。.usect または .bss 疑似命令でブロック・フラグを指定することで、構造体をブロックすることができます。これらの疑似命令の詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

例 7-1 に、asmfunc というアセンブリ言語関数を呼び出す main という C/C++ 関数を示します。asmfunc 関数は 1 つの引数を取り、gvar という C/C++ グローバル変数にその引数を加算し、その演算結果を戻します。

例 7-1. アセンブリ言語関数

(a) C++ プログラム

```
extern “C”{
extern int asmfunc(int a); /* declare external asm function */
int gvar = 0;              /* define global variable        */
}


void main()
{
   int i = 5;

   i = asmfunc(i);         /* call function normally */
}
```

(b) アセンブリ言語プログラム

```
        .global _gvar
        .global _asmfunc

_asmfunc:
        MOVZ    DP,#_gvar
        ADDB    AL,#5
        MOV     @_gvar,AL
        LRETR
```

例 7-1 の C++ プログラムでは、extern “C” 宣言でコンパイラに C 命名規則（すなわち、ネーム・マングリングなし）を使用するように指示しています。リンカが .global _asmfunc 参照を解決できる場合、アセンブリ・ファイル内の対応する定義が一致します。

パラメータ i がレジスタ AL に渡されます。

7.4.2 アセンブリ言語のグローバル変数にアクセスする方法

アセンブリ言語で定義された変数に C/C++ プログラムからアクセスできると便利な場合があります。.bss セクションにある初期化されない変数にアクセスするのは、簡単です。

1) .bss 疑似命令を使用して変数を定義します。
2) .global 疑似命令を使用して、その定義を外部定義にします。
3) 名前の前に下線を付けます。
4) C では、その変数を extern 宣言し、通常の方法でアクセスします。

例 7-2 に、.bss で 定義されている変数をアクセスする方法を示します。

例 7-2. C/C++ から .bss で定義されている変数にアクセスする方法

(a) C/C++ プログラム

```
extern int var;       /* External variable            */
.
.
.
var = 1;              /* Use the variable             */
```

(b) アセンブリ言語プログラム

```
* Note the use of underscores in the following lines

   .bss      _var,1   ; Define the variable
   .global   _var     ; Declare it as external
```

変数が必ず .bss セクションになければならないわけではありません。たとえば一般的な状況として、アセンブリ言語に定義された照合テーブルを RAM に入れたくない場合があります。この場合、オブジェクトへのポインタを定義し、間接的に C/C++ からアクセスする必要があります。

最初のステップはオブジェクトを定義することです。このオブジェクトを初期化されたセクションに入れると便利です（必ずしも入れる必要はありません)。オブジェクトの先頭を示すグローバル・ラベルを宣言すると、オブジェクトを任意のメモリ空間にリンクできます。C でこれにアクセスするには、オブジェクトを *extern* 宣言し、その名前の前に下線を付けてはいけません。これで、通常の方法でオブジェクトにアクセスできます。

例 7-3 に、.bss 内に定義されていない変数にアクセスする方法を示します。

例 7-3.　.bss 内に定義されていない変数に C からアクセスする方法

(a) C プログラム

```
extern float sine[];    /* This is the object              */
float *sine_p = sine;   /* Declare pointer to point to it */
f = sine_p[4];          /* Access sine as normal array    */
```

(b) アセンブリ言語プログラム

```
   .global  _sine       ; Declare variable as external
   .sect    "sine_tab"  ; Make a separate section
_sine:                  ; The table starts here
   .float   0.0
   .float   0.015987
   .float   0.022145
```

7.4.3　アセンブリ言語定数へのアクセス

.set、.def、.global の各疑似命令を使用することにより、アセンブリ言語の中でグローバル定数を定義できます。また、リンカ代入文を使用してリンカ・コマンド・ファイルの中でグローバル定数を定義することもできます。C/C++ からグローバル定数にアクセスする唯一の方法は、特別な演算子を使用することです。

C/C++ またはアセンブリ言語で定義した通常の変数の場合、シンボル・テーブルにはその変数の*値のアドレス*が入っています。しかしアセンブラ定数の場合は、シンボル・テーブルには定数の*値*が入っています。コンパイラは、シンボル・テーブル内のどの項目が値で、どの項目がアドレスであるか判断できません。

アセンブラ（またはリンカ）定数に名前でアクセスしようとした場合、コンパイラは、シンボル・テーブルの中で表されているアドレスから値を取り出そうとします。この望ましくない取り出しを防止するには、& 演算子（アドレス演算子）を使用して値を取り出さなければなりません。つまり x が アセンブリ言語定数の場合、その値は C/C++ では &x になります。

プログラムの中でこれらのシンボルを使いやすくするため、例 7-4 に示すように、キャストと #define を使用できます。

例 7-4.　C からアセンブリ言語定数にアクセスする方法

(a)　C プログラム

```
extern int table_size; /*external ref */
#define TABLE_SIZE ((int) (&table_size))
            .        /* use cast to hide address-of */
            .
            .
for (i=0; i<TABLE_SIZE; ++i)
                     /* use like normal symbol */
```

(b)　アセンブリ言語プログラム

```
_table_size .set     10000        ; define the constant
            .global _table_size   ; make it global
```

この場合はシンボル・テーブルに保存されたシンボルの値だけを参照しようとしているので、シンボルの宣言された型は重要ではありません。例 7-4 では、int が使用されています。リンカで定義したシンボルも、これとほとんど同じ方法で参照できます。

7.4.4　インライン・アセンブリ言語の使用方法

C/C++ プログラムの中で *asm 文*を使用すると、コンパイラで作成されたアセンブリ言語ファイルの中に、アセンブリ言語の一文を挿入できます。連続して asm 文を使用すれば、他のコードを間に入れることなくコンパイラ出力の中にアセンブリ言語を連続的に挿入できます。

注：　asm 文の使用方法

asm 文は、C/C++ から他の方法によってアクセスできないハードウェアの機能をアクセスできるようにするために用意されています。asm 文を使用する場合は、C/C++ 環境が損なわれないよう特に注意してください。コンパイラは挿入された命令をチェックまたは解析しません。

ジャンプまたはラベルを C/C++ コードに挿入すると、コード・ジェネレータが使用するレジスタ・トラッキング・アルゴリズムを混乱させることになり、予期せぬ結果が生じる場合があります。

C/C++ 変数の値を変更しないでください。ただし、変数の現行値は安全に読み出すことができます。

asm 文を使って、アセンブリ環境を変更するアセンブラ疑似命令を挿入してはいけません。

IER または IFR 制御レジスタをアクセスするアセンブリ文を追加するよりも、C/C++ で cregister キーワードを使用してこれらのレジスタにアクセスすることを推奨します。詳細は、6.7.3 項「cregister キーワード」（6-15 ページ）を参照してください。

また asm 文は、コンパイラ出力にコメントを挿入する場合にも役立ちます。以下のように、アセンブリ・コードの文字列をアスタリスク (*) から始めるだけで済みます。

```
asm("**** this is an assembly language comment");
```

7.4.5 組み込み関数（intrinsics）を使用してアセンブリ言語文にアクセスする方法

TMS320C28x コンパイラは、多くの組み込み関数を認識します。組み込み関数は、関数と同様に使われ、C/C++ では他の方法によって表現できないようなアセンブリ文を作成します。通常の関数で使用する場合と全く同じように、これらの組み込み関数でも C/C++ 変数を使用できます。

組み込み関数は名前の前に 2 つの下線を付けて指定し、関数のように呼び出すことによりアクセスできます。たとえば次のとおりです。

```
long lvar;

int  ivar;

unsigned int uivar;

lvar = __mpyxu(ivar, uivar);
```

表 7-3 に示す組み込み関数が組み込まれています。組み込み関数は TMS320C28x アセンブリ言語命令に対応します。詳細は、『TMS320C28x DSP CPU 及びインストラクション・セット リファレンス・ガイド』を参照してください。

表 7-3. TMS320C28x C/C++ コンパイラの組み込み関数

C/C++ コンパイラの組み込み関数	アセンブリ命令	説明
int __**abs16_sat**(int *src*)	**SETC OVM** **MOV AH,** *src* **ABS ACC** **MOV** *dst*, **AH** **CLRC OVM**	OVM ステータス・ビットをクリアします。src を AH にロードします。ACC の絶対値を取ります。AH を dst に格納します。OVM ステータス・ビットをクリアします。
long __**addcu**(long *src1*, unsigned int *src2*);	**ADDCU ACC,** {*mem* | *reg*}	*src2* の内容とキャリー・ビットの値を ACC（*src1*）に加算します。演算結果は ACC に入ります。
long __**mpy**(int *src1*, int *src2*);	**MPY ACC,** *src1*, #*src2*	*src1* を T レジスタに転送します。T と 16 ビット即値（*src2*）を乗算します。演算結果は、ACC に入ります。
long __**IQ**(long double *A*, int *N*);		long double 型の *A* を long 型として戻される正しい IQN 値に変換します。2 つの引数が定数の場合、コンパイラはコンパイル時に引数を IQ 値に変換します。それ以外の場合、RTS ルーチン __IQ の呼び出しが行われます。この組み込み関数を使用して、グローバル変数を .cinit セクションに初期化することができます。

表 7-3.　TMS320C28x C/C++ コンパイラの組み込み関数（続き）

C/C++ コンパイラの組み込み関数	アセンブリ命令	説明
long *dst*=**__IQmpy**(long *A*, long *B*, int *N*)**;**		*dst* は ACC または P になり、A は XT になります。
N == 0 の場合：	**IMPYL ACC/P,XT,B**	*dst* は ACC または P に入ります。*dst* が ACC の場合、命令は 2 サイクルかかります。*dst* が P の場合、命令は 1 サイクルかかります。
0 < *N* < 16 の場合：	**IMPLY P, XT, B** **QMPYL ACC, XT, B** **LSR64 ACC:P, #***N*	*dst* は P に入ります。
15 < *N* < 32 の場合：	**IMPYL P, XT, B** **QMPYL ACC, XT, B** **LSL64 ACC:P, #(32-***N***)**	*dst* は ACC に入ります。
N == 32 の場合：	**QMPYL ACC/P, XT, B**	*dst* は ACC または P のいずれかに入ります。
N が変数の場合：	**IMPYL P, XT, B** **QMPY ACC, XT, B** **MOV T,** *N* **LSR64 ACC:P, T**	*dst* は P に入ります。
long *dst* = **__IQsat**(long *A*, long *max*, long *min*)**;**		*dst* は ACC になります。*max* と *min* の両方またはそのいずれかの値に基づいて、異なるコードが生成されます。
max と *min* が 22 ビット符号なし定数の場合：	**MOVL ACC, A** **MOVL XARn, #***22bits* **MINL ACC, P** **MOVL XARn, #***22bits* **MAXL ACC, P**	
max と *min* が他の定数の場合：	**MOVL ACC, A** **MOV PL, #***max lower 16 bits* **MOV PH, #***max upper 16 bits* **MINL ACC, P** **MOV PL, #***min lower 16 bits* **MOV PH, #***min upper 16 bits* **MAXL ACC, P**	
max と *min* の両方またはそのいずれかが変数の場合：	**MOVL ACC, A** **MINL ACC,** *max* **MAXL ACC,** *min*	

表 7-3. TMS320C28x C/C++ コンパイラの組み込み関数（続き）

C/C++ コンパイラの組み込み関数	アセンブリ命令	説明
long *dst* = **__IQxmpy**(long *A*, long *B*, int *N*)**;**		*dst* は ACC または P のいずれかになります。A は XT になります。*N* の値に基づいて、コードが生成されます。
N == 0 の場合：	**IMPYL ACC/P, XT, B**	*dst* は ACC または P のいずれかに入ります。
0 < *N* < 17 の場合：	**IMPYL P, XT, B** **QMPYL ACC, XT, B** **LSL64 ACC:P, #*N***	*dst* は ACC に入ります。
0 > *N* > -17 の場合：	**QMPYL ACC, XT, B** **SETC SXM** **SFR ACC, #abs(*N*)**	*dst* は ACC に入ります。
16 < *N* < 32 の場合：	**IMPYL P, XT, B** **QMPYL ACC, XT, B** **ASR64 ACC:P, #*N***	*dst* は P に入ります。
N == 32 の場合：	**IMPYL P, XT, B**	*dst* は P に入ります。
-16 >*N* > -33 の場合：	**QMPYL ACC, XT, B** **SETC SXM** **SRF ACC, #16** **SRF ACC, #abs(*N*)-16**	*dst* は ACC に入ります。
32 < *N* < 49 の場合：	**IMPYL ACC, XT, B** **LSL ACC, #*N* -32**	*dst* は ACC に入ります。
-32 > *N* > -49 の場合：	**QMPYL ACC, XT, B** **SETC SXM** **SFR ACC, #16** **SFR ACC, #16**	*dst* は ACC に入ります。
48 < *N* < 65 の場合：	**IMPYL ACC, XT, B** **LSL64 ACC:P, #16** **LSL64 ACC:P, #*N* - 48**	*dst* は ACC に入ります。
-48 > *N* > -65 の場合：	**QMPYL ACC, XT, B** **SETC SXM** **SFR ACC, #16** **SFR ACC, #16**	*dst* は ACC に入ります。

表 7-3.　TMS320C28x C/C++ コンパイラの組み込み関数（続き）

C/C++ コンパイラの組み込み関数	アセンブリ命令	説明
int **__mov_byte**(int **src*, unsigned int *n*);	**MOVB AX.LSB,*+XARx[*n*]** または **MOVZ AR0/AR1, @*n*** **MOVB AX.LSB,*XARx[AR0/AR1]**	*src* が示すバイト・テーブルの 8 ビットの n 番目のエレメントを戻します。
long **__mpyb**(int *src1*, uint *src2*);	**MPYB {ACC | P}, T, #*src2***	*src1*（T レジスタ）と符号なし 8 ビット即値（*src2*）を乗算します。演算結果は ACC または P のいずれかに入ります。
long **__mpy_mov_t**(int *src1*, int *src2*, int **dst2*);	**MPY ACC, T, *src2*** **MOV *dst2*, T**	*src1*（T レジスタ）と *src2* を乗算します。演算結果は ACC に入ります。*src1* を *dst2* に転送します。
unsigned long **__mpyu**(uint *src1*, uint *src2*);	**MPYU {ACC / P}, T, *src2***	*src1*（T レジスタ）と *src2* を乗算します。2 つのオペランドは符号なし 16 ビットの数値として扱われます。演算結果は ACC または P のいずれかに入ります。
long **__mpyxu**(int *src1*, uint *src2*);	**MPYXU ACC, T, *{mem|reg}***	T レジスタに src1 がロードされます。*src2* はメモリによって参照されるか、レジスタにロードされます。演算結果は、ACC に入ります。
long *dst* = **__norm32**(long *src*, int **shift*);	**CSB ACC** **LSLL ACC, T** **MOV @*shift*, T**	*src* を *dst* に正規化し、*shift* で示されるビット数だけシフトします。
long long *dst* = **__norm64**(long long *src*, int **shift*);	**CSB ACC** **LSL64 ACC:P, T** **MOV *shift*, T** **CSB ACC** **LSL64 ACC:P, T** **MOV TMP16, AH** **MOV AH, T** **ADD *shift*, AH** **MOV AH, TMP16**	64 ビットの *src* を *dst* に正規化し、*shift* で足されるビット数だけシフトします。

表 7-3. TMS320C28x C/C++ コンパイラの組み込み関数（続き）

C/C++ コンパイラの組み込み関数	アセンブリ命令	説明
long **__qmpy32**(long *src32a*, long *src32b*, int *q*);	**CLRC OVM**	拡張精度 DSP Q 算術。q の値に基づいて、異なるコードが生成されます。
	SPM - 1	
	MOV T, *src32a* **+ 1**	
	MPYXU P, T, *src32b* **+ 0**	
	MOVP T, *src32b* **+ 1**	
	MPYXU P, T, *src32a* **+ 0**	
	MPYA P, T, *src32a* **+ 1**	
q = 31,30 の場合：	**SPM q - 30**	
	SFR ACC, #45 - q	
	ADDL ACC, P	
q = 29 の場合：	**SFR ACC, #16**	
	ADDL ACC, P	
q = 28 〜 24 の場合：	**SPM q - 30**	
	SFR ACC, #16	
	SFR ACC, #29 - q	
	ADDL ACC, P	
q = 23 〜 13 の場合：	**SFR ACC, #16**	
	ADDL ACC, P	
	SFR ACC, #29 - q	
q = 12 〜 0 の場合：	**SFR ACC, #16**	
	ADDL ACC, P	
	SFR ACC, #16	
	SFR ACC, #13 - q	

表 7-3.　TMS320C28x C/C++ コンパイラの組み込み関数（続き）

C/C++ コンパイラの組み込み関数	アセンブリ命令	説明
long **__qmpy32by16**(long *src32*, int src16, int *q*); q = 31,30 の場合： q = 29 ～ 14 の場合： q = 13 ～ 0 の場合：	**CLRC　OVM** **MOV　T,** *src16* **+ 0** **MPYXU　P, T,** *src32* **+ 0** **MPY　P, T,** *src32* **+ 1** **SPM　q - 30** **SFR　ACC, #46 - q** **ADDL　ACC, P** **SPM　0** **SFR　ACC, #16** **ADDL　ACC, P** **SFR　ACC, #30 - q** **SPM　0** **SFR　ACC, #16** **ADDL　ACC, P** **SFR　ACC, #16** **SFR　ACC, #14 - q**	拡張精度 DSP Q 算術。q の値に基づいて、異なるコードが生成されます。
long **__rol**(long *src*);	ROL ACC	ACC を左ローテートします。
long **__ror**(long *src*);	ROR ACC	ACC を右ローテートします。
void *result = **__rpt_mov_imm**(void **dst*, int *src*, int *count*);	**MOV** *result*, *dst* **MOV AR***x*, *dst* **RPT** #*count* **|| MOV *XAR***x*++, #*src*	*dst* レジスタを *result* レジスタに転送します。*dst* レジスタを一時（ARx）レジスタに転送します。即値 *src* を一時レジスタに *count* + 1 回コピーします。 *src* は、16 ビット即値でなければなりません。*count* は、0 ～ 255 の即値または変数です。

表 7-3. TMS320C28x C/C++ コンパイラの組み込み関数（続き）

C/C++ コンパイラの組み込み関数	アセンブリ命令	説明
far void *result = **__rpt_mov_imm_far** (far void ***dst**, int *src*, int *count*);	**MOVL** *result*, *dst* **MOVL XAR***x*, *dst* **RPT** #*count* **|| MOV *XARx++, #***src*	dst レジスタを result レジスタに転送します。dst レジスタを一時（XARx）レジスタに転送します。即値 src を一時レジスタに count + 1 回コピーします。 *src* は、16 ビット即値でなければなりません。*count* は、0 ～ 255 の即値または変数です。
int **__rpt_norm_inc**(long *src*, int *dst*, int *count*);	**MOV AR***x*, *dst* **RPT** #*count* **|| NORM ACC, AR***x*++	アキュムレータの値の正規化を *count* + 1 回繰り返します。 *count* は、0 ～ 255 の即値または変数です。
int **__rpt_norm_dec**(long *src*, int *dst*, int *count*);	**MOV AR***x*, *dst* **RPT** #*count* **|| NORM ACC, AR***x*++	アキュムレータの値の正規化を *count* + 1 回繰り返します。 *count* は、0 ～ 255 の即値または変数です。
long **__rpt_rol**(long *src*, int *count*);	**RPT** #*count* **|| ROL ACC**	アキュムレータの値の左ローテートを *count* + 1 回繰り返します。演算結果は、ACC に入ります。 *count* は、0 ～ 255 の即値または変数です。
long **__rpt_ror**(long *src*, int *count*);	**RPT** #*count* **|| ROR ACC**	アキュムレータの値の右ローテートを *count* + 1 回繰り返します。演算結果は、ACC に入ります。 *count* は、0 ～ 255 の即値または変数です。
long **__rpt_subcu**(long *dst*, int *src*, int *count*);	**RPT** *count* **|| SUBCU ACC,** *src*	*src* オペランドはメモリから参照されるか、レジスタにロードされ、SUBCU 命令のオペランドとして使われます。演算結果は ACC に入ります。*count* は、0 ～ 255 の即値または変数です。命令は、*count* + 1 回繰り返します。

表 7-3.　TMS320C28x C/C++ コンパイラの組み込み関数（続き）

C/C++ コンパイラの組み込み関数	アセンブリ命令	説明
long **__sat**(long *src*)**;**	**SAT ACC**	ACC に 32 ビット src をロードします。演算結果は、ACC に入ります。
long **__sat32**(long *src*, long *limit*)**;**	**SETC OVM** **ADDL ACC,** [*mem*|**P**] **SUBL ACC,** [*mem*|**P**] **SUBL ACC,** [*mem*|**P**] **ADDL ACC,** [*mem*|**P**] **CLRC OVM**	32 ビット値を 32 ビット・マスクに飽和します。ACC に src をロードします。制限値は、メモリから参照されるか、P レジスタにロードされます。演算結果は、ACC に入ります。
long **__sathigh16**(long *src*, int *limit*)**;**	**SETC OVM** **ADD ACC,**{ *mem* |*reg*} **<< 16** **SUB ACC,**{ *mem*|*reg*} **<< 16** **SUB ACC,** { *mem*|*reg*} **<< 16** **ADD ACC,** {*mem*|*reg*} **<< 16** **CLRC OVM** **SFR ACC,** *rshift*	32 ビット値を上位 16 ビットに飽和します。ACC に src をロードします。制限値は、メモリから参照されるか、レジスタにロードされます。演算結果は、ACC に入ります。演算結果は右シフトされ、int に格納されます。たとえば次のとおりです。 `ivar=__sathigh16(lvar, mask)>>6;`
long **__satlow16**(long *src*)**;**	**SETC OVM** **MOV T, #0xFFFF** **CLR SXM; if necessary** **ADD ACC, T<< 15** **SUB ACC, T <<15** **SUB ACC, T <<15** **ADD ACC, T <<15** **CLRC OVM**	32 ビット値を下位 16 ビットに飽和します。ACC に *src* をロードします。T レジスタに #0xFFFF をロードします。演算結果は、ACC に入ります。
long **__sbbu**(long *src1*, uint *src2*)**;**	**SBBU ACC,** *src2*	ACC（*src1*）から *src2* + C の論理反転を減算します。演算結果は、ACC に入ります。
long **__subcu**(long *src1*, int *src2*)**;**	**SUBCU ACC,** *src2*	ACC（*src1*）から 15 左シフトした *src2* を減算します。演算結果は、ACC に入ります。
if (**_ _tbit**(int *src*, int *bit*))	**TBIT** *src*, *#bit*	*src* の指定ビットが 1 の場合、TC のステータス・ビットをセットします。

7.5 割り込み処理

ここで説明するガイドラインに従えば、C/C++ 環境を損なわずに C/C++ コードに割り込み、再び C/C++ コードに戻ることができます。C/C++ 環境の初期化時に、始動ルーチンにより割り込みが有効になったり無効になったりすることはありません（システムがハードウェア・リセットによって初期化される場合、割り込みは無効です）。システムで割り込みが使用される場合は、必要となる割り込みの有効化やマスキングの処理はユーザが行わなければなりません。このような操作は、C/C++ 環境に影響を与えず、asm 文で組み込むことで簡単に実現できます。

7.5.1 割り込みについての一般的なポイント

割り込みルーチンは、グローバル変数へのアクセス、ローカル変数の割り当て、他の関数の呼び出しなど、他の関数によって実行されるタスクを実行できます。

割り込みルーチンを記述する場合、以下の点に注意してください。

- 割り込み処理ルーチンは引数を持てません。何かの引数を宣言しても、無視されます。
- 割り込み処理ルーチンは通常の C/C++ コードから呼び出すことはできますが、すべてのレジスタが保存されるので、この作業を行うことは非効率的です。
- 割り込み処理ルーチンは、単独の割り込みまたは複数の割り込みを処理できます。コンパイラは、特定の割り込み固有のコードを生成しません。ただし、システム・リセット割り込みの c_int00 は除きます。このルーチンに入った場合、ランタイム・スタックが設定されていることを想定することができないので、ローカル変数を割り当てることが できず、ランタイム・スタック上に情報を保存 でき*ません*。
- 割り込みルーチンを割り込みに関連付けるには、適切な割り込みベクタに割り込み関数のアドレスを配置する必要があります。アセンブラとリンカを使い、.sect アセンブラ疑似命令を使って割り込みアドレス・テーブルを作成することにより、この操作を実行します。
- アセンブリ言語では、シンボル名の前に下線を付けることを忘れないでください。たとえば、c_int00 は _c_int00 としてください。

7.5.2 C/C++ 割り込みルーチンの使用方法

C/C++ 割り込みルーチンが他の関数を呼び出さない場合、割り込みハンドラが使用するレジスタだけが保存され復元されます。ただし、C/C++ 割り込みルーチンが他の関数を*呼び出す*場合、そのような関数は割り込みハンドラが使用しない既知のレジスタを変更することができます。そのため、他の関数が呼び出された場合、コンパイラは呼び出し時に保存されるレジスタをすべて保存します。

C/C++ 割り込みルーチンは、ローカル変数およびレジスタ変数をもつことができる他の C/C++ 関数と似ています。ただし、引数をもたないと宣言し、void を戻す必要があります。割り込み処理関数は、直接呼び出してはいけません。

割り込みは、interrupt プラグマまたは interrupt キーワードを使うことにより、C/C++ 関数で*直接*処理できます。interrupt プラグマの詳細は、6.8.5 項「INTERRUPT プラグマ」（6-31 ページ）を参照してください。interrupt キーワードの詳細は、6.7.4 項「interrupt キーワード」（6-16 ページ）を参照してください。

7.6 整数式の分析

ここでは、整数式を評価する際に考慮すべき特別な注意事項について説明します。

7.6.1 RTS 呼び出しを使って評価される整数演算

TMS320C28x は C/C++ 整数演算の一部を直接サポートしていません。これらの演算の評価は、ランタイムサポート・ルーチンを呼び出すことで行われます。これらのルーチンは、アセンブリ言語で手書きで作成されています。これらはツールセットのオブジェクトとソースのランタイムサポート・ライブラリ（rts2800.lib と rts.src）のメンバです。

これらのルーチンを呼び出すための規則は、標準 C/C++ 呼び出し規則をモデルにしています。

演算タイプ	ランタイムサポート呼び出しを使って評価される演算
16 ビット int	除算（符号付き） 剰余
32 ビット long	除算（符号付き） 剰余（符号付き）
64 ビット long long	乗算 除算 ビット単位での AND、OR、および XOR 比較

7.6.2 16 ビット乗算での上位 16 ビットへの C/C++ コードによるアクセス

次の方法を使って、C/C++ 言語から 16 ビット乗算で上位 16 ビットへアクセスできます。

❑ 符号付き演算結果による方法：

```
int   m1, m2;
int   result;

result = ((long) m1 * (long) m2) >> 16;
```

❑ 符号なし演算結果による方法：

```
unsigned m1, m2;
unsigned result;

result = ((unsignedlong) m1 * (unsigned long) m2) >> 16;
```

注： 複雑な式の危険性

コンパイラは、自身が期待したような結果を戻すために式の構造を認識しなければなりません。次の例のような複雑な式は使わないようにしてください。

```
((long)((unsigned)((a*b)+c)<5)*(long)(z*sin(w)>6))>>16
```

7.7 浮動小数点式の分析

TMS320C28x C/C++ コンパイラは、float 型および double 浮動小数点型の値を IEEE 単精度の数値として表します。long double 浮動小数点型の値は、IEEE 倍精度の数値として表されます。単精度浮動小数点型の数値は 32 ビット値として表され、倍精度浮動小数点型の数値は 64 ビット値として表されます。

ランタイムサポート・ライブラリ rts2800.lib には、浮動小数点算術計算用の関数が含まれていて、次の演算をサポートしています。

- 加算、減算、乗算、除算
- 比較（>、<、>=、<=、==、!=）
- int または long から浮動小数点への変換および浮動小数点から int または long への変換（符号付きと符号なしの両方を含む）
- 標準エラー処理

これらのルーチンを呼び出すための規則は、整数演算ルーチンを呼び出すために使われる規則と同じです。規則は単項演算です。

7.8 システムの初期化

C/C++ プログラムを実行する前に、C/C++ ランタイム環境を作成する必要があります。この作業は、C/C++ ブート・ルーチンによって行われ、c_int00 と呼ばれる関数を使用します。このルーチンのソースは、ランタイムサポート・ソース・ライブラリの boot.asm と呼ばれるモジュール内に入っています。

システムの実行を開始するために、ハードウェア・リセットによって c_int00 関数を呼び出すことができます。この関数はランタイムサポート・ライブラリ（rts2800.lib）に入っており、C/C++ オブジェクト・モジュールとリンクしなければなりません。これをデフォルトで行うには、-c または -cr リンカ・オプションを指定し、rts2800.lib をリンカ入力ファイルの 1 つとして組み込みます。C/C++ プログラムをリンクした場合、リンカは実行可能な出力モジュールのエントリ・ポイント値をシンボル c_int00 に設定します。

c_int00 関数は、C/C++ 環境を初期化するために次の作業を実行します。

1) ランタイム・スタックのスペースを確保し、初期スタック・ポインタの値を設定します。

2) ステータス・ビットとレジスタを期待した値に初期化します。

3) グローバル変数を初期化するため、.cinit セクションの初期化テーブルから、.bss セクション内の変数に割り当てられた記憶域にデータをコピーします。RAM 自動初期化モデルでは、プログラム実行前にローダがこのステップを実行します（ブート・ルーチンでは実行されません）。

4) .pinit セクションで見つかったグローバル・コンストラクタを実行します。詳細は、7.8.3 項「グローバル・コンストラクタ」を参照してください。

5) 関数 main を呼び出して、C/C++ プログラムを実行します。

ユーザのシステム要件に合わせてブート・ルーチンの置換や変更ができます。ただし、C/C++ 環境を正しく初期化するためにブート・ルーチンでは、上記の作業を必ず実行しなければなりません。

7.8.1 ランタイム・スタック

ランタイム・スタックは、単独の連続したメモリ・ブロックに割り当てられ、下位のアドレスから上位のアドレスへと増大します。SP はスタック内で次に使用可能なワードを指します。

コードはランタイム・スタックがオーバーフローしているかどうかをチェックしません。スタックが割り当てられたメモリ空間の上限を超えると、スタックがオーバーフローします。スタックに十分なメモリを割り当てるようにします。

スタック・サイズは、リンカ・コマンド行で -stack オプションを使用し、このオプションの後に定数としてスタック・サイズを指定することで、リンク時に変更することができます。

7.8.2　変数の自動初期化

一部のグローバル変数には、C/C++ プログラムが実行を開始する前に必ず初期値を割り当てなければなりません。これらの変数のデータを取り出し、そのデータを使用して変数を初期化するプロセスを、自動初期化と呼びます。

コンパイラは、グローバル変数と静的変数を初期化するためのデータが入った .cinit という特別なセクションにテーブルを作成します。コンパイルされた各モジュールには、それらの初期化テーブルが含まれています。リンカは、それらのテーブルを 1 つのテーブル（1 つの .cinit セクション）にまとめます。ブート・ルーチンまたはローダは、このテーブルを使用して、すべてのシステム変数を初期化します。

注：　変数の初期化方法

ANSI/ISO C では、明示的に初期化されていないグローバル変数および静的変数は、プログラム実行前に 0 に設定されます。C28x C/C++ コンパイラは、初期化されていない変数の事前初期化は実行しません。初期値が 0 でなければならない変数は、すべて明示的に初期化します。

変数を初期化する最も簡単な方法は、プログラムが実行を開始する前にローダを使って .bss セクションをクリアすることです。別の方法として、.bss セクションのリンカ制御マップで、埋め込み値を 0 に設定する方法もあります

ROM に組み込まれたコードではこれらの方法は使えません。

グローバル変数は実行時にもロード時にも自動初期化されません（7.8.5 項「実行時の変数の自動初期化」（7-45 ページ）および 7.8.6 項「ロード時の変数の自動初期化」（7-46 ページ）を参照）。

7.8.3　グローバル・コンストラクタ

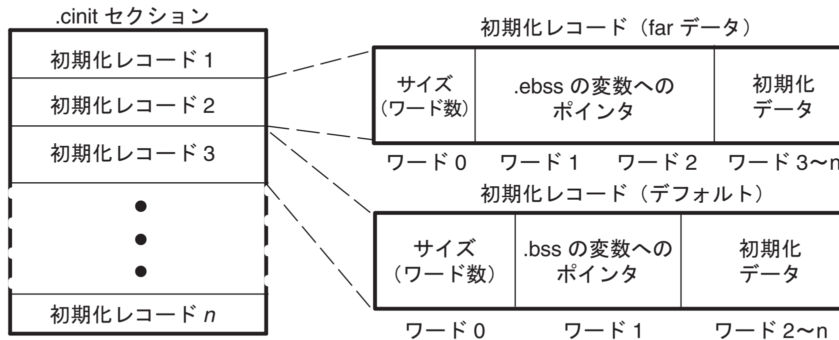
コンストラクタをもつすべてのグローバル C++ 変数は、main() の前にコンストラクタを呼び出す必要があります。コンパイラは、.pinit と呼ばれるセクション内で、main() を実行する前に順に呼び出す必要があるグローバル・コンストラクタのアドレス・テーブルを作成します。リンカは、各入力ファイルからの .pinit セクションを結合して、.pinit セクション内に 1 つのテーブルを作成します。ブート・ルーチンは、このテーブルを使用してコンストラクタを実行します。

7.8.4 初期化テーブル

.cinit レコードを生成すると、far データ・アドレスを考慮したレコードを効率的に作成します。

.cinit セクション内のテーブルは、変数サイズの初期化レコードから構成されています。図 7-2 に、.cinit セクションと初期化レコードのフォーマットを示します。

図 7-2. .cinit セクション内の初期化レコードのフォーマット（デフォルトと far データ）



初期化レコードのフィールドに含まれている情報は、次のとおりです。

❑ 最初のフィールドは、変数の初期化データのサイズ（ワード数）です。このフィールドの値が負の場合は、変数のアドレスが far であることを示します。

❑ 2 番目のフィールドは .bss セクション内の変数の先頭アドレスを示し、ここに初期化データをコピーします。変数が far の場合、このフィールドは .ebss 内の変数のスペースを指します。far データの場合、2 番目のフィールドにはアドレスを保持するために 2 ワード必要です。

❑ 3 番目のフィールドには、変数を初期化するためにコピーされたデータが含まれています。

注：　変数の初期化方法

コンパイラはサイズが 32K ワード以下のオブジェクトの初期化のみをサポートしています。

.cinit セクションには、自動初期化が必要な各変数の初期化レコードが含まれています。たとえば、2 つの初期化される変数は C では次のように定義されているものと仮定します。

```
int    i = 23;
far    int j[2] = { 1,2};
```

この場合、初期化テーブル・エントリは次のようになります。

```
       .global  _i
       .bss     _i,1,1,0
       .global  _j
_j:    .usect   .ebss,2,1,0

       .sect    ".cinit"
       .align   1
       .field         1,16
       .field         _i+0,16
       .field         23,16          ; _i @ 0


       .sect    ".cinit"
       .align   1
       .field         -IR_1,16
       .field         _j+0,32
       .field         1,16           ; _j[0] @ 0
       .field         2,16           ; _j[1] @ 16
IR_1:  .set     2
```

.cinit セクションに含まれるのは、このフォーマットの初期化テーブルだけです。アセンブリ言語モジュールを C/C++ プログラムにインターフェイスする場合は、.cinit セクションを他の目的に使用しないでください。

-c や -cr リンカ・オプションを指定すると、リンカはすべての C モジュールの .cinit セクションをまとめてリンクし、結合した .cinit セクションの最後にヌル・ワードを付けます。この終了レコードはサイズ・フィールドが 0 のレコードとなり、初期化テーブルの最後を表します。

注：　const 変数の初期化

const として修飾されている変数は、別々に初期化されます。6.9 節「静的変数とグローバル変数の初期化方法」（6-32 ページ）を参照してください。

.pinit セクション内のテーブルは、呼び出されるコンストラクタのアドレスのリストから構成されています（図 7-3 を参照）。コンストラクタは、実行されなければならない順にテーブルに記録されています。

図 7-3.　.pinit セクションのフォーマット

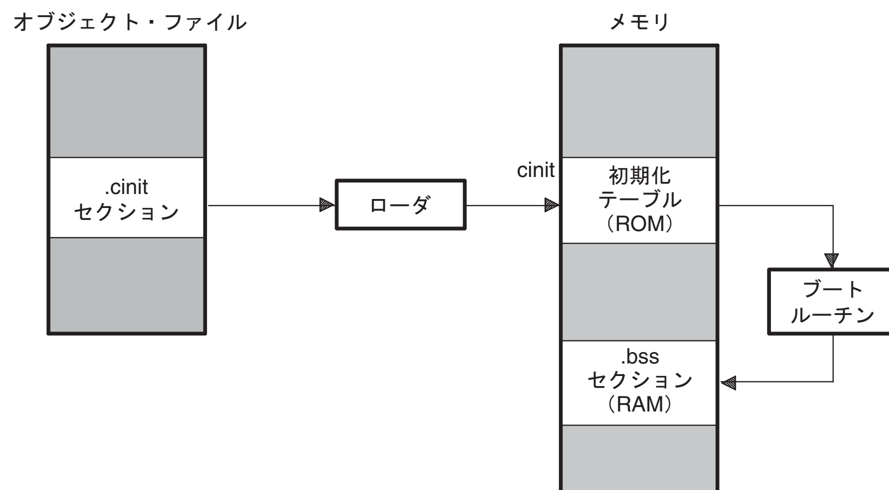
7.8.5 実行時の変数の自動初期化

実行時に変数の自動初期化を行うのが、デフォルトの自動初期化方法です。この方法を使用するには、-c オプションを指定してリンカを起動します。

この方法を使用すると、.cinit セクションは、その他の初期化されたすべてのセクションとともにメモリにロードされます。リンカは cinit という特別なシンボルを定義し、このシンボルはメモリ内の初期化テーブルの先頭を指します。プログラムの実行が開始されると、C/C++ ブート・ルーチンは、(.cinit が指す) テーブルのデータを .bss セクション内の指定された変数にコピーします。これにより初期化データを ROM に格納し、プログラムが起動するたびに RAM にコピーするという使い方ができます。

図 7-4 に、実行時の自動初期化を示します。ROM に組み込まれたコードからアプリケーションを実行するシステムでは、この方法を使用してください。

図 7-4. 実行時の自動初期化



7.8.6 ロード時の変数の自動初期化

ロード時に変数の自動初期化を行うと、ブート時間が短縮され、初期化テーブルが使用するメモリも節約できるので、パフォーマンスが向上します。この方法を使用するには、-cr オプションを指定してリンカを起動します。

-cr リンカ・オプションを指定すると、リンカは .cinit セクションのヘッダ内に STYP_COPY ビットを設定します。これにより、.cinit セクションを メモリ内にロードしないことをローダに指示します（.cinit セクションは、メモリ・マップのスペースを占有しません)。また、リンカは cinit シンボルを -1 に設定します（通常では cinit は初期化テーブルの先頭を指します)。これにより、初期化テーブルがメモリ内に存在しないことをブート・ルーチンに示します。したがって、ブート時に実行時の初期化は行われません。

ロード時に自動初期化を行うには、ローダ（これはコンパイラ・パッケージの一部ではありません）が次の作業を実行できなければなりません。

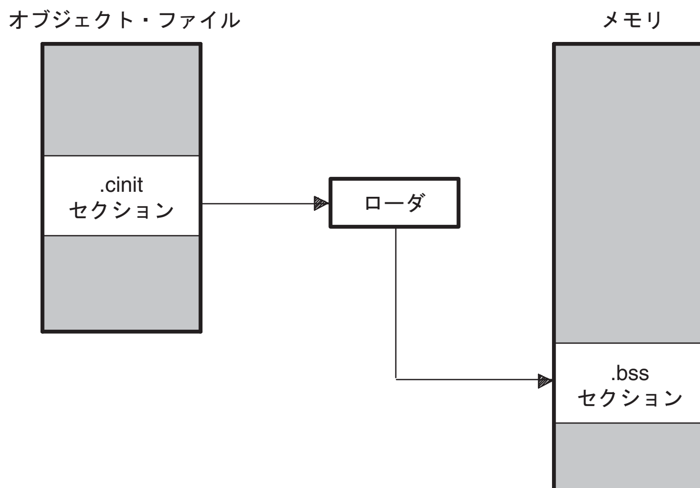
❏ オブジェクト・ファイル内の .cinit セクションの存在を検出する

❏ .cinit セクション・ヘッダ内に STYP_COPY が設定されているかどうかを判別して、.cinit セクションをメモリ内にコピーしないことを認識する

❏ 初期化テーブルのフォーマットを理解する

図 7-5 に、ロード時の変数の初期化を示します。

図 7-5. ロード時の自動初期化



第 8 章

ランタイムサポート関数

C/C++ プログラムが実行する作業（メモリ割り当て、文字列変換、文字列の検索など）の中には、C/C++ 言語の一部ではないものがあります。ランタイムサポート関数は、C/C++ コンパイラに付属していますが、前述の作業を実行する標準 ANSI/ISO 関数です。

ランタイムサポート・ライブラリ rts.src には、他の関数およびルーチンのためのソースと同様、これらの関数のソースが入っています。基本的なオペレーティング・システムを必要とするもの（信号など）を除き、すべての ANSI/ISO 関数が提供されています。

ANSI/ISO 指定の関数だけでなく、ランタイムサポート・ライブラリには、プロセッサ固有のコマンドを実行したり、C 言語から入出力要求を指示したりするルーチンが組み込まれています。またライブラリは拡張されていて、動的メモリ管理ルーチン（C/C++ の場合）の far バージョンおよび ANSI/ISO ルーチン（C の場合のみ）の far バージョンも組み込まれています。

任意のランタイムサポート関数を使用する場合、ライブラリ作成ユーティリティ（第 9 章「ライブラリ作成ユーティリティ」を参照）を使用して、必ずデバイスに応じて適切なライブラリを作成してください。また、C/C++ プログラムをリンクする時は、このライブラリを組み込んでください。

項目 **ページ**

8.1 ライブラリ 8-2

8.2 far メモリのサポート 8-4

8.3 C 入出力関数 8-8

8.4 ヘッダ・ファイル 8-18

8.5 ランタイムサポート関数およびマクロのまとめ 8-31

8.6 ランタイムサポート関数およびマクロについて 8-43

8.1 ライブラリ

TMS320C28x C/C++ コンパイラに付属しているライブラリは、次のとおりです。

- *rts2800.lib* には、ANSI/ISO C/C++ のランタイムサポート・オブジェクト・ライブラリが含まれています。
- *rts.src* には、ANSI/ISO C/C++ のランタイムサポート・ルーチンのためのソースが含まれています。
- *rts2800_ml.lib* には、C/C++ ラージ・メモリ・モデルで使用するランタイムサポート・オブジェクト・ライブラリが含まれています。

オブジェクト・ライブラリには、本章で説明している標準 C/C++ ランタイムサポート関数、浮動小数点ルーチン、およびシステム起動ルーチン _c_int00 が含まれています。オブジェクト・ライブラリは、rts.src ライブラリに入っている C/C++ およびアセンブリ・ソースから作成されます。

8.1.1 オブジェクト・ライブラリとコードのリンク方法

プログラムをリンクするとき、リンカ入力ファイルの 1 つとしてオブジェクト・ライブラリを必ず指定することにより、入出力関数およびランタイムサポート関数に対する参照が解決できるようになります。

リンカ・コマンド行でライブラリを指定するのは、*最後に* してください。リンカは、コマンド行でライブラリを検出すると、未解決参照を探すためにライブラリを検索するからです。-x リンカ・オプションを使用して、リンカが参照を解決できるものがなくなるまで強制的に繰り返し各ライブラリを検索させることもできます。

ライブラリをリンクする際、リンカは未定義の参照を解決する上で必要なライブラリ・メンバだけを組み込みます。リンク方法の詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

8.1.2 ライブラリ関数の修正方法

アーカイバを使用してソース・ライブラリから適切なソース・ファイルを抽出することにより、ライブラリ関数を検査したり修正したりできます。たとえば、次のコマンドでは 2 つのソース・ファイルを抽出できます。

```
ar2000 x rts.src atoi.c strcpy.c
```

関数を修正するには、前の例と同じようにしてソースを抽出します。そしてそのコードに必要に応じて変更を加え、再コンパイルし、新しいオブジェクト・ファイルをライブラリに再インストールします。

```
cl2000 -v28 -options atoi.c strcpy.c     ; recompile
ar2000 r rts2800.lib atoi.obj strcpy.obj; rebuild library
```

rts2800.lib に再作成するのでなく、このような方法で新しいライブラリを作成することもできます。アーカイバの詳細は、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

8.1.3 さまざまなオプションによるライブラリの作成方法

ライブラリ作成ユーティリティ mk2000 -v28 により、rts.src から新しいライブラリを作成できます。たとえば、次のコマンドを使って、最適化したランタイムサポート・ライブラリを作成できます。

```
mk2000 -v28 --u -O2 rts.src -l rts2800.lib
```

--u オプションは、ソース・アーカイブからヘッダ・ファイルを抽出するのではなく、カレント・ディレクトリにあるヘッダ・ファイルを使用するよう、mk2000 -v28 ユーティリティに指示します。新しく作成されたライブラリは、TMS320C28x 用にコンパイルされたすべてのコードと互換性があります。（-O2）オプションを使用すると、これらのオプションを指定せずにコンパイルされたコードとの互換性に影響を与えません。ライブラリ作成ユーティリティの詳細は、第 9 章を参照してください。

8.2 far メモリのサポート

6.7.6 項「far キーワード」(6-19 ページ) で説明したように、C/C++ コンパイラは far キーワードを提供することで (C++ ではなく) C 言語を拡張しています。far キーワードは、22 ビットのアドレス空間をアクセスするために使われます。

ランタイムサポート・ライブラリは拡張され、far に対応した ANSI/ISO ルーチンおよび動的メモリ管理ルーチンの far バージョンが組み込まれています。

ラージ・メモリ・モデルの C++ のランタイムサポート・ライブラリは、すべてのポインタが 22 ビットになっているので、far メモリは自動的にサポートされます。

8.2.1 ランタイムサポート関数の far バージョン

C ランタイム・ライブラリで far メモリをサポートするために、ポインタ型の引数を取ったりポインタ型の値を戻すほとんどのランタイムサポート関数に対して、far バージョンが定義されています。次の例では、atoi() ランタイムサポート関数は、文字列 (char へのポインタ) 引数を取り、文字列で表される整数値を戻します。

```
#include <stdlib.h>
char * st = "1234";
.
.
.
.
int    ival = atoi(st);   /* ival is 1234 */
```

atoi() 関数の far バージョン far_atoi() は、far 型の文字列 (far 型の文字へのポインタ) 引数を取り、整数値を戻します。

```
#include <stdlib.h>
far char * fst = "1234";
.
.
.
.
int    ival = far_atoi(fst); /* ival is 1234 */
```

ランタイムサポート関数の far バージョンは、far オブジェクトへのポインタや far オブジェクトへのポインタを戻すことを除き、通常のランタイムサポート関数と同様の演算を実行します。表 8-3 (k)「関数の far バージョンのまとめ」(8-40 ページ) に、ランタイムサポート・ライブラリに含まれる far 関数をすべて示します。

8.2.2 ランタイムサポートにおけるグローバル変数と静的変数

ランタイムサポート・ライブラリは、さまざまなグローバル変数と静的変数を使っています。stderr.h で定義されているグローバル変数 errno の場合のように、内部で使われているもの、およびユーザにステータスやその他の情報を渡すためのものがあります。デフォルトでは、これらの変数は .bss セクションに配置され、near オブジェクトと見なされます。詳細は、7.1.1 項「セクション」（7-3 ページ）を参照してください。

C 入出力関数には、対応する far バージョンはありません。また、C 入出力関数を内部で使う関数にも対応する far バージョンはありません。

（C++ ではなく）C ランタイム・ライブラリ作成時に、_FAR_RTS を定義すること（-d_FAR_RTS）で、グローバル変数と静的変数を far メモリ（.ebss セクション）に配置することができます。コマンド行から次のように入力すると、far メモリにグローバル変数と静的変数を配置する C ランタイムサポート・ライブラリを作成できます。ヘッダ・ファイル linkage.h には、far ランタイムサポート・ライブラリ作成に必要なマクロが定義されています。

```
mk2000 -v28 -d_FAR_RTS rts.src -l rts2800.lib
```

8.2.3 C における far メモリの動的割り当て

実行時に、far メモリを動的に割り当てることができます。far メモリは、.esysmem セクションに定義されるグローバル・プール（far ヒープ）から割り当てられます。詳細は、7.1.4 項「動的なメモリ割り当て」（7-7 ページ）を参照してください。

ランタイムサポート・ライブラリには、実行時に far メモリを動的に割り当てることを可能にする次の関数が組み込まれています。

far void * far_malloc (unsigned long *size*);
far void * far_calloc (unsigned long *num*, unsigned long *size*);
far void * far_realloc (far void **ptr*, unsigned long *size*);
void far_free (far void **ptr*);
long far_free_memory(void);
long far_max_free(void);

次の C コードでは、100 個の far オブジェクト用にメモリを割り当て、最後にそのメモリを解放しています。

```
#include <stdlib.h>

struct big {
   int a, b;
   char c[80];
};

int main()
{
   far struct big *table;
   table=(far struct big *)far_malloc(100*sizeof(struct big));
   .
   .
   /* use the memory here */
   .
   .
   far_free(table);
   /* exit code here */
}
```

8.2.4 ラージ・メモリ・モデル用のランタイムサポート・ライブラリの作成方法

-ml シェル・オプションを指定することで、C/C++ のラージ・メモリ・モデル用のライブラリを作成することができます。ランタイムサポート・ライブラリのソース・コードは、条件付きコンパイルを使ってラージ・メモリ・モデルをサポートしています。ランタイムサポート・ライブラリをコンパイルする場合、LARGE_MODEL シンボルを定義しておく必要があります。たとえば、C++ のラージ・メモリ・モデル用のライブラリを作成するには、次のコマンドを使用します。

```
mk2000 -v28 -ml -DLARGE_MODEL -o rtscpp.src -l rtscpp_ml.lib
```

8.2.5 C++ における far メモリの動的割り当て

C++ モードでは、コンパイラは far キーワードをサポートしません。ラージ・メモリ・モデルを使用しない場合に、far メモリをアクセスするには、far 組み込み関数を使用します。詳細は、6.7.8 項「C++ で far メモリをアクセスする組み込み関数を使用する方法」（6-22 ページ）を参照してください。far メモリにあるオブジェクトを動的に定義し、far 組み込み関数を使用してそのオブジェクトにアクセスできます。far ポインタには long 型が使用されます。

C++ ランタイムサポート・ライブラリは、C ランタイムサポート・ライブラリと同じ far に対応した動的なメモリ割り当て関数のセットを提供します。C++ 関数は、戻り値として far ポインタを扱えるように long 型を使用します。そのため、far 組み込み関数を使ってメモリをアクセスすることができます。C++ の far メモリの動的割り当て関数は、次のとおりです。

long std::far_malloc (unsigned long *size*);
long std::far_calloc (unsigned long *num*, unsigned long *size*);
long std::far_realloc (long *ptr*, unsigned long *size*);
void std::far_free (long *ptr*);
long std::far_free_memory (void);
long std::far_max_free (void);

次の C++ コードでは、100 個の far オブジェクトにメモリを割り当て、最後にそのメモリを解放しています。

```
#include <cstdlib>

struct big {
   int a, b;
   char c[80];
};

int main()
{
   long table;//Note-use of long to hold address.
   table = std::far_malloc(100 * sizeof(struct big));
   .
   .
   /* use the memory here using intrinsic*/
   .
   .
   std::far_free(table);
   /* exit code here */
}
```

注： far 組み込み関数の使用方法

rts.src に含まれているファイル farmemory.cpp は、far 組み込み関数を使用して far に対応した動的メモリ割り当て関数を実現しています。far 組み込み関数の使用方法に関する例としてこのファイルを参照できます。

8.3 C 入出力関数

C 入出力関数を使用すると、ホストのオペレーティング・システムにアクセスして入出力を実行できます (デバッガを使用)。たとえば、プログラムで実行される printf 文は、デバッガ・コマンド・ウィンドウに表示されます。デバッグ・ツールと組み合わせて使用した場合には、ホスト上で入出力を実行できるので、コードのデバッグとテストに利用できるオプションの数が多くなります。

入出力関数を使用するには、次のようにします。

- ❑ 関数を参照するモジュールごとに、ヘッダ・ファイル stdio.h をインクルードします。
- ❑ プログラムで使用する各入出力ストリームに対して、320 バイトのヒープ空間を考慮します。ストリームは、端末またはキーボードなどのペリフェラルに関連付けられているデータの送信元または送信先です。ストリームはヒープから取られる動的に割り当てられたメモリを使用して、バッファリングされます。(malloc() の呼び出しによって) 動的に割り当てられるメモリをさらに使用するプログラムをサポートするために、より多くのヒープ空間が必要になる場合があります。ヒープ・サイズを設定するには、リンク時に -heap オプションを指定します。
  -heap オプションの詳細は、4.2 節「リンカ・オプション」(4-5 ページ) を参照してください。

たとえば、main.c というファイル内に次のプログラムが指定されていると仮定します。

```
#include <stdio.h>

main()
{
 FILE *fid;

   fid = fopen("myfile","w");
   fprintf(fid,"Hello, world\n");
   fclose(fid);

   printf("Hello again, world\n");
}
```

次のシェル・コマンドを発行すると、ファイル main.out のコンパイル、リンク、および作成が行われます。

```
cl2000 -v28 main.c -z -l rts2800.lib -o main.out
```

PC ホスト上でデバッガから main.out を実行すると、次の作業が行われます。

1) デバッガが起動されたディレクトリでファイル myfile をオープンする。
2) そのファイルに文字列 *Hello, world* を出力する。
3) ファイルをクローズする。
4) デバッガ・コマンド・ウィンドウに文字列 *Hello again, world* を出力する。

正しく作成されたデバイス・ドライバにより、C 入出力関数を使用して、ユーザ指定のデバイス上で入出力を実行できます。

8.3.1 低レベル入出力実装の概要

入出力を実装するコードは、論理的に 3 つの層（高レベル、低レベル、デバイス・レベル）に分類されます。

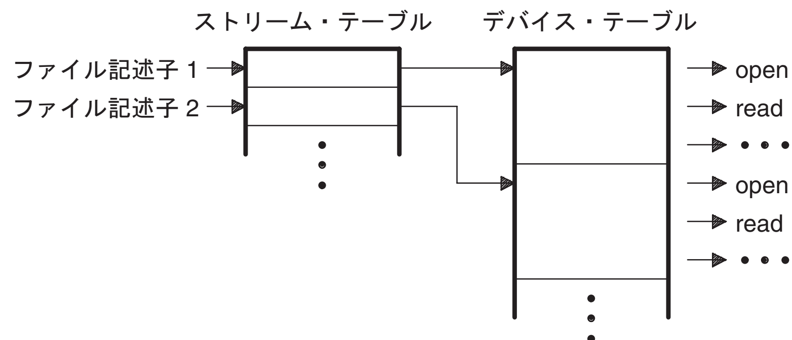
高レベル関数は、ストリーム入出力ルーチン（printf、scanf、fopen、getchar など）の標準 C ライブラリです。これらのルーチンは、低レベル・シェルによって処理される 1 つまたは複数の入出力コマンドに、入出力要求をマップします。

低レベル関数は、基本入出力関数（OPEN、READ、WRITE、CLOSE、LSEEK、RENAME、UNLINK）から構成されています。指定したデバイス上で入出力コマンドを実際に実行する高レベル関数とデバイス・レベル・ドライバ間のインターフェイスが、これらの低レベル関数によって提供されます。

また、ファイル記述子をデバイスに関連付けるストリーム・テーブルの定義と保守も、低レベル関数によって行われます。ストリーム・テーブルはデバイス・テーブルと相互作用し、ストリーム上で実行する入出力コマンドに対して、デバイス・レベルのルーチンを確実に実行させます。

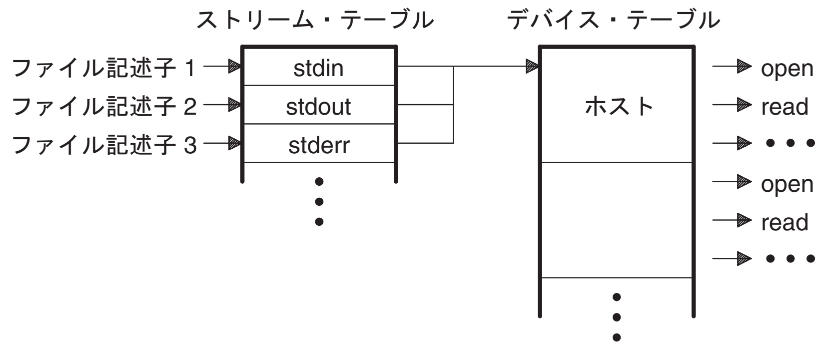
データ構造は、図 8-1 に示すように相互作用します。

図 8-1. 入出力関数におけるデータ構造の相互作用



ストリーム・テーブル内の最初の 3 つのストリームは stdin、stdout、stderr に事前定義されていて、ホスト・デバイスおよび関連するデバイス・ドライバを指します。

図 8-2. ストリーム・テーブル内の最初の 3 つのストリーム



次のレベルに来るのは、ユーザが定義できるデバイス・レベル・ドライバです。このデバイス・レベル・ドライバは、低レベル入出力関数に直接マップします。C 入出力ライブラリには、デバッガが稼働中のホストで C 入出力を実行するために必要なデバイス・ドライバが入っています。

低レベルのルーチンとインターフェイスするようにデバイス・レベルのルーチンを作成する際の仕様については、8.3.2 項「C 入出力用デバイスの追加方法」(8-10 ページ) を参照してください。各関数が、必要に応じてそれぞれに固有のデータ構造を設定し保持するように作成する必要があります。何の処置も実行せず、ただ戻るように関数を定義する場合もあります。

8.3.2 C 入出力用デバイスの追加方法

実行時に入出力用デバイスを追加して使用できる機能が、低レベル関数にはあります。この機能を使用するための手順は、次のとおりです。

1) 8.3.1 項「低レベル入出力実装の概要」(8-9 ページ) に示すように、デバイス・レベルの関数を定義します。

注： 一意な関数名を使用してください

関数名 open()、close()、read() などは、低レベルのルーチンで使用されています。作成するデバイス・レベルの関数には、他の名前を使ってください。

2) 低レベル関数 add_device() を使用して、device_table にデバイスを追加します。デバイス・テーブルは、*n* 個のデバイスをサポートする静的に定義された配列です。この *n* は、stdio.h の中にあるマクロ _NDEVICE で定義されています。またデバイスを表す構造体も stdio.h の中で定義され、次のフィールドから構成されます。

name デバイス名を表す文字列

flags デバイスが複数のストリームをサポートするかどうかを指定するフラグ

function pointers 次のデバイス・レベル関数を指すポインタ

- ❏ CLOSE
- ❏ LSEEK
- ❏ OPEN
- ❏ READ
- ❏ RENAME
- ❏ WRITE
- ❏ UNLINK

デバイス・テーブルの最初のエントリは、デバッガが稼働中のホスト・デバイスになるように事前定義されます。低レベル・ルーチン add_device() は、デバイス・テーブルの最初の空き位置を検出し、渡された引数でデバイス・フィールドを初期化します。add_device 関数の詳細は、8-12 ページを参照してください。

3) デバイスを追加した後に fopen() を呼び出してストリームをオープンし、このストリームをそのデバイスに関連付けます。fopen() への最初の引数として *devicename:filename* を使います。

add_device　デバイス・テーブルへのデバイスの追加

C の構文

```
#include <file.h>
int add_device(char *name,
               unsigned flags,
               int (*dopen)(), parameters,
               int (*dclose)(), parameters,
               int (*dread)(), parameters,
               int (*dwrite)(), parameters,
               fpos_t (*dlseek)(), parameters,
               int (*dunlink)(), parameters,
               int (*drename)(), parameters);
```

定義される場所

rts.src の lowlev.c

説明

add_device 関数は、デバイス・レコードをデバイス・テーブルに追加し、そのデバイスを C から入出力に使用できるようにします。デバイス・テーブルの最初のエントリは、デバッガが稼働中のホスト・デバイスになるよう事前定義されています。関数 add_device() はデバイス・テーブル内で最初の空の位置を検出し、追加されるデバイスに対応する構造体のフィールドを初期化します。

新たに追加したデバイス上でストリームをオープンするには、書式 *devicename*:*filename* の文字列を第 1 引数にして、fopen() を使用してください。

- *name* は、デバイス名を示す文字列です。
- *flags* は、デバイス特性です。フラグは以下のとおりです。

  _SSA　一度に 1 つのストリームしかオープンできないことを示します。

  _MSA　複数のストリームをオープンできることを示します。

  フラグを stdio.h/cstdio の中に定義すると、より多くのフラグを追加できます。

- dopen、dclose、dread、dwrite、dlseek、dunlink、drename の各指定子は、指定したデバイスで入出力を実行するために低レベル関数により呼び出されるデバイス・ドライバへの関数ポインタです。これらの関数を宣言する場合には、必ず 8.3.1 項「低レベル入出力実装の概要」(8-9 ページ) に指定されているインターフェイスを使用してください。デバッガが実行されるホストのデバイス・ドライバは、C/C++ 入出力ライブラリに組み込まれています。

戻り値

この関数は、以下の値のどれかを戻します。

0　　成功した場合

-1　　失敗した場合

例

この例では、以下の操作が実行されます。

1) デバイス *mydevice* をデバイス・テーブルに追加する

2) そのデバイス上で *test* という名前のファイルをオープンし、ファイル **fid* に関連付ける

3) そのファイルに文字列 *Hello, world* を出力する

4) ファイルをクローズする

```
#include <stdio.h>

/****************************************************************************/
/* Declarations of the user-defined device drivers                          */
/****************************************************************************/
extern int my_open(const char *path, unsigned flags, int fno);
extern int my_close(int fno);
extern int my_read(int fno, char *buffer, unsigned count);
extern int my_write(int fno, const char *buffer, unsigned count);
extern int my_lseek(int fno, long offset, int origin);
extern int my_unlink(const char *path);
extern int my_rename(const char *old_name, const char *new_name);

main()
{
   FILE *fid;
   add_device("mydevice", _MSA, my_open, my_close, my_read, my_write, my_lseek,
                              my_unlink, my_rename);

   fid = fopen("mydevice:test","w");

   fprintf(fid,"Hello, world\n");

   fclose(fid);
}
```

far 関数 なし

close 入出力用ファイルまたはデバイスのクローズ

C の構文

#include <stdio.h>
#include <file.h>

int close(int *file_descriptor***);**

C++ の構文

#include <cstdio>
#include <file.h>

説明

close 関数は、*file_descriptor* に関連付けられているデバイスまたはファイルをクローズします。

file_descriptor は、オープンされたデバイスまたはファイルに関連付けられている低レベル・ルーチンによって割り当てられるストリーム番号です。

戻り値

この関数は、以下の値のどれかを戻します。

0　　成功した場合
-1　　失敗した場合

lseek　ファイル位置標識の設定

C の構文

```
#include <stdio.h>
#include <file.h>

long lseek(int file_descriptor, long offset, int origin);
```

C++ の構文

```
#include <cstdio>
#include <file.h>

long std::lseek(int file_descriptor, long offset, int origin);
```

説明

lseek 関数は、指定されたファイルの位置標識を *origin* + *offset* に設定します。ファイル位置標識では、ファイルの先頭から文字単位で位置が測定されます。

- ❑ *file_descriptor* は、オープンされたファイルまたはデバイスにデバイス・レベル・ドライバが関連付けなければならない低レベル・ルーチンによって割り当てられるストリーム番号です。
- ❑ *offset* は、*origin* からの文字単位の相対位置を指示します。
- ❑ *origin* は、どの基本位置から *offset* を測定するかを指示するために使用します。*origin* は、次のいずれかのマクロによって戻される値でなければなりません。

  SEEK_SET　(0x0000) ファイルの先頭
  SEEK_CUR　(0x0001) ファイル位置標識の現行値
  SEEK_END　(0x0002) ファイルの最後

戻り値

この関数は、以下の値のどれかを戻します。

#	成功した場合は、ファイル位置標識の新しい値
EOF	失敗した場合

open 入出力用ファイルまたはデバイスのオープン

C の構文

#include <stdio.h>
#include <file.h>

int open(char ***_path_, unsigned** _flags_**, int** _mode_**);**

C++ の構文

#include <cstdio>
#include <file.h>

int std::open(char ***_path_, unsigned** _flags_**, int** _mode_**);**

説明

open 関数は、*path* で指定するデバイスまたはファイルをオープンし、入出力用に準備します。

❑ *path* はオープンするファイルのファイル名で、パス情報を含んでいます。

❑ *flags* は、デバイスまたはファイル処理方法を指定する属性です。次のシンボルを使用してフラグを指定します。

```
O_RDONLY (0x0000)   /* open for reading */
O_WRONLY (0x0001)   /* open for writing */
O_RDWR   (0x0002)   /* open for read & write */
O_APPEND (0x0008)   /* append on each write */
O_CREAT  (0x200)    /* open with file create */
O_TRUNC  (0x400)    /* open with truncation */
O_BINARY (0x8000)   /* open in binary mode */
```

データがデバイスにどのように解釈されるかによりますが、これらのパラメータを無視できる場合も時々あります。ただし、高レベル入出力呼び出しはファイルが fopen 文でどのようにオープンされたかを調べて、オープン属性に応じて特定の処理が行われないようにします。

❑ *mode* は必須ですが、値は無視されます。

戻り値

この関数は、以下の値のどれかを戻します。

>= 0　成功した場合
< 0　失敗した場合

read バッファからの文字の読み出し

C の構文

#include <stdio.h>
#include <file.h>

int read(int *file_descriptor*, **char** **buffer*, **unsigned** *count*);

C++ の構文

#include <cstdio>
#include <file.h>

int std::read(int *file_descriptor*, **char** **buffer*, **unsigned** *count*);

説明

read 関数は、*count* で指定した文字数を、*file_descriptor* に関連付けられているデバイスまたはファイルから読み取って *buffer* に入れます。

❑ *file_descriptor* は、オープンされたファイルまたはデバイスに関連付けられている低レベル・ルーチンによって割り当てられるストリーム番号です。

❑ *buffer* は、読みとられた文字が入るバッファのロケーションです。

❑ *count* は、デバイスまたはファイルから読み取る文字数です。

戻り値

この関数は、以下の値のどれかを戻します。

0	読み取りが完了する前に EOF が検出された場合
#	上記または下記の場合以外に読み取られた文字数
-1	失敗した場合

rename ファイルの名前変更

C の構文

#include <stdio.h>
#include <file.h>

int rename(char **old_name*, **char** **new_name*);

C++ の構文

#include <cstdio>
#include <file.h>

int std::rename(char **old_name*, **char** **new_name*);

説明

rename 関数は、ファイルの名前を変更します。

❑ *old_name* は、ファイルの現在の名前です。

❑ *new_name* は、ファイルの新しい名前です。

戻り値

この関数は、以下の値のどれかを戻します。

0	成功した場合
0 以外	失敗した場合

unlink　ファイルの削除

C の構文

```
#include <stdio.h>
#include <file.h>

int unlink(char *path);
```

C++ の構文

```
#include <cstdio>
#include <file.h>

int std::unlink(char *path);
```

説明

unlink 関数は、*path* で指定したファイルを削除します。

path は削除するファイルのファイル名で、パス情報を含んでいます。

戻り値

この関数は、以下の値のどれかを戻します。

0　　成功した場合
-1　　失敗した場合

write　バッファへの文字の書き込み

C の構文

```
#include <stdio.h>
#include <file.h>

int write(int file_descriptor, char *buffer, unsigned count);
```

C++ の構文

```
#include <cstdio>
#include <file.h>

int write(int file_descriptor, char *buffer, unsigned count);
```

説明

write 関数は、*count* で指定した文字数を *buffer* から *file_descriptor* に関連付けられているデバイスまたはファイルに書き込みます

- ❑ *file_descriptor* は、オープンされたファイルまたはデバイスに関連付けられている低レベル・ルーチンによって割り当てられるストリーム番号です。
- ❑ *buffer* は、書き込まれた文字が入るバッファのロケーションです。
- ❑ *count* は、デバイスまたはファイルに書き込む文字数です。

戻り値

この関数は、以下の値のどれかを戻します。

#　　成功した場合に書き込まれた文字数
-1　　失敗した場合

8.4 ヘッダ・ファイル

各ランタイムサポート関数は、ヘッダ・ファイル 内に宣言されています。各ヘッダ・ファイルで宣言する内容は、次のとおりです。

- ❏ 一連の関連する関数（またはマクロ）
- ❏ 関数を使用するために必要な型
- ❏ 関数を使用するために必要なマクロ

ANSI/ISO C ランタイムサポート関数を宣言するヘッダ・ファイルは次のとおりです。

assert.h	inttypes.h	setjmp.h	stdio.h
ctype.h	iso646.h	stdarg.h	stdlib.h
errno.h	limits.h	stddef.h	string.h
float.h	math.h	stdint.h	time.h

ANSI/ISO C ヘッダ・ファイルに加えて、次の C++ ヘッダ・ファイルも含まれます。

cassert	climits	cstdio	new
cctype	cmath	cstdlib	stdexcept
cerrno	csetjmp	cstring	typeinfo
cfloat	cstdarg	ctime	
ciso646.h	cstddef	exception	

先頭文字が **c** ではじまる C++ ヘッダ・ファイルは、その c という文字がない ANSI/ISO C ヘッダに対応しています（つまり、cerrno は errno.h に対応)。先頭文字が c ではじまる C++ ヘッダ・ファイルは、対応する ANSI/ISO C ヘッダ・ファイルと同じ機能がありますが、すべての型と関数を std ネームスペースに導入します。ANSI/ISO C ヘッダ・ファイルを C++ アプリケーションと組み合わせて使うことはできますが、そのような使用方法は推奨しません。

ランタイムサポート関数を使用するには、まず #include プリプロセッサ疑似命令を使用してその関数を宣言するヘッダ・ファイルを組み込まなければなりません。たとえば、C アプリケーションでは、isdigit 関数は ctype.h ヘッダで宣言されています。isdigit 関数を使用するには、次のようにまず ctype.h を組み込む必要があります。

```
#include  <ctype.h>
   .
   .
   .
   val = isdigit(num);
```

ヘッダの組み込み順序に指定はありません。ただし、そのヘッダで宣言する関数やオブジェクトを参照する前にヘッダを組み込まなければなりません。

8.4.1 項から 8.4.9 項では、C/C++ コンパイラに付属しているヘッダ・ファイルについて説明します。8.5 節「ランタイムサポート関数およびマクロのまとめ」（8-31 ページ）には、これらのヘッダが宣言する関数が掲載されています。

8.4.1 診断メッセージ（assert.h/cassert）

assert.h/cassert ヘッダは assert マクロを定義します。これは、実行時に障害診断メッセージをプログラムに挿入するマクロです。assert マクロは、実行時に式をテストします。

- ❑ 式が真（0 以外の値）の場合、プログラムは実行を継続します。
- ❑ 式が偽の場合、assert マクロはその式、ソース・ファイル名、および式を含む文の行番号が入っているメッセージを出力します。その後プログラムは（abort 関数により）終了します。

assert.h/cassert ヘッダは NDEBUG マクロを参照します (assert.h/cassert では NDEBUG の定義は行いません)。assert.h/cassert を組み込むときに NDEBUG がマクロ名としてすでに定義されている場合、assert は無効になり何も実行されません。NDEBUG が定義*されていない*場合、assert は有効です。

assert.h/cassert ヘッダは NASSERT マクロを参照します。assert.h/cassert を組み込むときに NASSERT がマクロ名としてすでに定義されている場合、assert は実行時にテストした式が真であると見なします。これにより、オプティマイザはコードの残りの部分を最適化するために式が真であることを利用するだけでなく、assert マクロを除去することができます。

assert マクロ の定義は次のとおりです。

```
#ifdef NDEBUG
#define assert(ignore) ((void)0)

#elseif defined (NASSERT)
#define assert (expression)   _nassert(expression)

#else
#define assert(expr) ((void)((_expr) ? 0 :
       (printf("Assertion failed, ("#_expr"), file %s,  \
       line %d\n, __FILE__, __LINE__),                  \
       abort () )))
#endif
```

8.4.2 文字の判別と変換 （ctype.h/cctype）

ctype.h/cctype ヘッダは、文字をテスト（判別）し、文字を変換する関数を宣言します。

文字判別関数は、文字をテストし、その文字が英字か、印字文字か、16 進数字かなどを判定します。これらの関数は、*真*（0 以外の値）か*偽*（0）を戻します。文字変換関数は、文字を小文字、大文字、あるいは ASCII に変換し、変換後の文字を戻します。文字判別関数の名前の形式は、**is***xxx*（*isdigit* など）です。文字変換関数の名前の形式は、**to***xxx*（*toupper* など）です。

ctype.h/cctype ヘッダには、これらと同じ処理を実行するマクロ定義も含まれています。マクロの方が、同じ機能をもつ関数より処理速度が高速です。これらのマクロの 1 つに副次作用がある引数を渡す場合は、対応する関数を使用してください。文字判別マクロは、フラグの配列（この配列は ctype.c 内に定義されています）の中でルックアップ演算に展開されます。マクロの名前は対応する関数と同じですが、各マクロの先頭には下線（たとえば *_isdigit*）が付きます。

これらの文字判別関数および文字変換関数については、表 8-3 (b)（8-31 ページ）を参照してください。

8.4.3 エラー報告（errno.h/cerrno）

errno.h/cerrno ヘッダは errno 変数を宣言します。errno 変数は算術関数のエラーを宣言します。無効なパラメータ値が関数に渡された場合、または定義されている範囲外の結果を関数が戻した場合は、算術関数でエラーが起きる可能性があります。このような場合、errno という変数は、次のマクロのいずれかの値に設定されます。

- ❑ EDOM - 領域エラーの場合（パラメータが不正）
- ❑ ERANGE - 範囲エラーの場合（結果が不正）

算術関数を呼び出す C コードでは、errno の値を読み取ってエラーの状態を調べることができます。errno 変数は errno.h/cerrno 内で宣言され、errno.c 内で定義されています。

8.4.4 低レベル入出力関数（file.h）

入出力操作を実行するために使用する低レベル入出力関数は file.h ヘッダで宣言されます。8.3 節「C 入出力関数」（8-8 ページ）では、C28x で入出力を実装する方法を説明しています。

8.4.5 制限値（float.h/cfloat と limits.h/climits）

float.h/cfloat ヘッダと limits.h/climits ヘッダでは、TMS320C28x の数値表現の有効な制限値やパラメータに展開するマクロを定義しています。表 8-1 と表 8-2 に、これらのマクロおよび関連付けられた制限値のリストを示します。

表 8-1. 整数型の範囲に関する制限値を指定するマクロ（limits.h）

マクロ	値	説明
CHAR_BIT	16	char 型のビット数
SCHAR_MIN	(-SCHAR_MAX -1)	signed char の最小値
SCHAR_MAX	32767	signed char の最大値
UCHAR_MAX	65535	unsigned char の最大値
CHAR_MIN	SCHAR_MIN	char の最小値
CHAR_MAX	SCHAR_MAX	char の最大値
SHRT_MIN	-32768	short int の最小値
SHRT_MAX	32767	short int の最大値
USHRT_MAX	65535	unsigned short int の最大値
INT_MIN	(-INT_MAX - 1)	int の最小値
INT_MAX	32767	int の最大値
UINT_MAX	65535	unsigned int の最大値
LONG_MIN	(-LONG_MAX - 1)	long int の最小値
LONG_MAX	2147483647	long int の最大値
ULONG_MAX	4294967295	unsigned long int の最大値
LLONG_MIN	(-LLONG_MAX - 1)	long long int の最小値
LLONG_MAX	9223372036854775807	long long int の最大値
ULLONG_MAX	18446744073709551615	unsigned long long int の最大値

注： この表の負の値は、その型が正確になるように、実際のヘッダ・ファイルの中では式として定義されています。

表 8-2. 浮動小数点の範囲に関する制限値を指定するマクロ（float.h）

マクロ	値	説明
FLT_RADIX	2	指数表現の底や基数
FLT_ROUNDS	1	浮動小数点加算の端数処理モード
FLT_DIG DBL_DIG LDBL_DIG	6 6 15	float、double、または long double に対する精度の 10 進桁数
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	24 24 53	float、double、または long double の仮数部の底 FLT_RADIX の桁数
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	-125 -125 -1021	FLT_RADIX の n-1 乗が正規化した float、double、または long double になるような整数で、最小の負の整数 n
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	128 128 1024	FLT_RADIX の n-1 乗が表現可能な有限の float、double、または long double になるような整数で、最大の正の整数 n
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1.19209290e-07 1.19209290e-07 2.22044605e-16	$1.0 + x \neq 1.0$ となる float、double、または long double の正の最小値 x
FLT_MIN DBL_MIN LDBL_MIN	1.17549435e-38 1.17549435e-38 2.22507386e-308	float、double、または long double の正の最小値
FLT_MAX DBL_MAX LDBL_MAX	3.40282347e+38 3.40282347e+38 1.79769313e+308	float、double、または long double の正の最大値
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	-37 -37 -307	10 の n 乗が正規化した float、double、または long double の範囲内に収まるような整数で、最小の負の整数 n
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	38 38 308	10 の n 乗が表現可能な有限の float、double、または long double の範囲内に収まるような整数で、最大の正の整数 n

凡例： FLT_ は、float 型に適用します。
DBL_ は、double 型に適用します。
LDBL_ は、long double 型に適用します。

注： この表の中の一部の値は、読みやすくするために精度を下げてあります。プロセッサで実施される完全な精度については、コンパイラで提供される float.h ヘッダ・ファイルを参照してください。

8.4.6 整数型のフォーマット変換（inttypes.h）

stdint.h ヘッダは、特定の幅の整数型を宣言し、対応するマクロのセットを定義します。inttypes.h ヘッダは stdint.h を含みます。また、一連の整数型を、複数のマシン間で整合性を保持し、オペレーティング・システムおよび他の実装の特異性から独立した状態で定義します。inttypes.h ヘッダは、最大幅の整数を操作し、数字文字列を最大幅の整数に変換する関数を宣言します。

typedef を介して、inttypes.h はさまざまなサイズの整数型を定義します。標準 C 整数型として、また inttypes.h に用意されている型として、自由に整数型を型定義できます。一貫して inttypes.h ヘッダを使用することにより、複数のプラットフォーム間でユーザ・プログラムの移植性が大幅に高まります。

ヘッダは、3 つの型を宣言します。

- *imaxdiv_t* 型 - imaxdiv 関数により戻される値の型である構造体型です。
- *intmax_t* 型 - signed int 型の値を表すのに十分な大きさの int 型です。
- *uintmax_t* 型 - unsigned int 型の値を表すのに十分な大きさの int 型です。

ヘッダは、複数のマクロおよび関数を宣言します。

- アーキテクチャ上で使用可能な型および stdint.h に提供されている型のそれぞれのサイズ用に、複数の fprintf および fscanf マクロがあります。たとえば、符号付き整数用に 3 つの fprintf マクロは、PRId32、PRIdLEAST32、および PRIdFAST32 があります。これらのマクロの使用例は以下のとおりです。

```
printf("The largest integer value is %020"
      PRIxMAX “\n”, i);
```

- intmax_t 型の整数の絶対値を計算する *imaxabs* 関数。
- strtol、strtoll、strtoul、および strtoull 関数と同等の *strtoimax* 関数および *strtoumax* 関数。文字列の先頭部分は、それぞれ intmax_t および uintmax_t に変換されます。

inttypes.h ヘッダの詳細は、『ISO/IEC 9899:1999, International Standard - Programming Languages - C (The C Standard)』を参照してください。

8.4.7 代替スペリング（iso646.h/ciso646）

iso646.h/ciso646 ヘッダは、対応するトークンに拡張する以下の 11 のマクロを定義します。

マクロ	トークン	マクロ	トークン
and	&&	not_eq	!=
and_eq	&=	or	||
bitand	&	or_eq	|=
bitor	|	xor	^
compl	~	xor_eq	^=
not	!		

8.4.8 浮動小数点算術（math.h/cmath）

math.h/cmath ヘッダは、三角関数、指数関数、ハイパボリック（双曲線）関数という算術関数を宣言します。これらの算術関数は、倍精度浮動小数点引数を要求し、倍精度浮動小数点値を戻します。

math.h/cmath ヘッダは *HUGE_VAL* というマクロも定義します。算術関数はこのマクロを使って範囲外の値を表します。関数が浮動小数点値を戻すときに、その値が大きすぎて表現できない場合は、代りに HUGE_VAL を戻します。

これらの関数を表 8-3 (c)（8-32 ページ）に示します。

8.4.9 非ローカル・ジャンプ（setjmp.h/csetjmp）

setjmp.h/csetjmp ヘッダは、通常の関数の呼び出しと復帰に関する規律をバイパスするための型、マクロ、および関数を定義します。これらについて、表 8-3 (e)（8-33 ページ）に示します。また次の内容が含まれます。

- *jmp_buf* - 呼び出し環境の復元に必要な情報の保存に適した配列型。
- *setjmp* - 後で longjmp 関数で使用できるように、呼び出し環境を jmp_buf 引数に保存するマクロ。
- *longjmp* - jmp_buf 引数を使用してプログラム環境を復元する関数。

また setjmp.h は jmp_buf、setump、longjmp の far バージョン far_jmp_buf、far_setjmp、far_longjmp も宣言します。

8.4.10 可変引数（stdarg.h/cstdarg）

関数の中には、型が変化する可変数の引数をもつものがあります。このような関数を*可変引数関数*といいます。stdarg.h/cstdarg ヘッダでは、可変引数関数を使用する上で役立つ 3 つのマクロと 1 つの型を宣言しています。

- ❏ 3 つのマクロとは、*va_start*、*va_arg*、*va_end* です。これらのマクロは、引数の数と型が関数の呼び出しごとに異なるときに使用します。
- ❏ 型 *va_list* は、*va_start*、*va_end*、*va_arg* の情報を保持できるポインタ型です。

可変引数関数は、stdarg.h/cstdarg で宣言されたマクロを使用してその引数リストを実行時に 1 つずつ処理することができます。このとき、マクロを使用している関数は、実際に渡された引数の数と型を認識します。可変引数関数を呼び出すためには、引数が正しく処理されるように関数のプロトタイプ宣言を明確に行う必要があります。

これらの関数を表 8-3 (f)（8-33 ページ）に示します。

8.4.11 標準定義 （stddef.h/cstddef）

stddef.h/cstddef ヘッダは 2 つの型と 2 つのマクロを定義します。型は次のとおりです。

- ❏ *ptrdiff_t* - 2 つのポインタの減算結果のデータ型を示す signed int 型です。
- ❏ *size_t* - *sizeof* 演算子のデータ型である unsigned int 型です。

マクロは次のとおりです。

- ❏ *NULL* - ヌル・ポインタ定数（0）に展開されます。
- ❏ *offsetof*（*type*、*identifier*）- *size_t* 型の整数に展開されます。演算結果は、構造体（型）の先頭から構造体メンバ（識別子）までのオフセットの値（バイト数）です。

これらの型およびマクロは、いくつかのランタイムサポート関数で使われます。

8.4.12 整数型（stdint.h）

stdint.h ヘッダは、特定の幅の整数型を宣言し、対応するマクロのセットを定義します。また、他の標準ヘッダで定義されている型に対応する整数型の限界を指定するマクロを定義します。型は次のカテゴリで定義されます。

- ある正確な幅をもつ整数型で、符号付き形式 int*N*_t および符号なし形式 uint*N*_t があります
- ある最小の幅をもつ整数型で、符号付き形式 int_least*N*_t および符号なし形式 uint_least*N*_t があります
- ある最小の幅をもつ整数型で、符号付き形式 int_fast*N*_t および符号なし形式 uint_fast*N*_t があります
- ポインタ値を保持するのに十分な大きさの整数型で、符号付き *intprt_t* および符号なし *uintptr_t* の整数型があります
- 任意の整数型の値を表現するのに十分な大きさの整数型で、符号付き *intmax_t* および符号なし *uintmax_t* の整数型があります

stdint.h の提供する各符号付き型に対しては、上限または下限を指定するマクロがあります。各マクロ名は、上記の類似した型名に対応しています。

INT*N*_C（*value*）マクロは、指定された値（*value*）と型 int_least*N*_t をもつ符号付き整数定数に展開されます。符号なし UINT*N*_C（*value*）マクロは、指定された値（*value*）と型 uint_least*N*_t をもつ符号なし整数定数に展開されます。

次の例では、少なくとも 16 ビットを保持できる最小の整数を使用する stdint.h で定義されているマクロを示しています。

```
       typedef    uint_least_16 id_number;
extern id_number  lookup_user(char *uname);
```

stdint.h ヘッダの詳細は、『ISO/IEC 9899:1999, International Standard - Programming Languages - C (The C Standard)』を参照してください。

8.4.13 入出力関数（stdio.h/cstdio）

stdio.h/cstdio ヘッダは、7 つのマクロ、2 つの型、1 つの構造体および多くの関数を定義します。型および構造体は次のとおりです。

- *size_t* - *sizeof* 演算子のデータ型である unsigned int 型です。元の宣言は stddef.h にあります。
- *fpos_t* - ファイル内のすべての場所を一意に指定できる unsigned long 型です。
- *FILE* - ストリームの制御に必要な情報をすべて記録する構造体です。

マクロは次のとおりです。

- *NULL* - ヌル・ポインタ定数（0）に展開されます。元の宣言は stddef.h にあります。すでに定義されている場合、再定義は行われません。
- *BUFSIZ* - setbuf() が使用するバッファのサイズに展開されます。
- *EOF* - ファイルの終わりを示します。
- *FOPEN_MAX* - 一度にオープンできるファイルの最大数に展開されます。
- *FILENAME_MAX* - 最長ファイル名の長さ（文字数）に展開されます。
- *L_tmpnam* - tmpnam() が 生成できる最長のファイル名文字列に展開されます。
- *TMP_MAX* - tmpnam() が生成できる一意なファイル名の最大数に展開されます。

これらの関数を表 8-3 (g)（8-33 ページ）に示します。

8.4.14 汎用ユーティリティ（stdlib.h/cstdlib）

stdlib.h ヘッダは、2 つの型、1 つのマクロ、およびいくつかの共通のライブラリ関数を宣言します。型は次のとおりです。

- ❏ *div_t* - div 関数が戻す値の構造体の型。
- ❏ *ldiv_t* - ldiv 関数が戻す値の構造体の型。

マクロ *RAND_MAX* は、rand 関数が戻す最大の乱数です。

共通のライブラリ関数は次のとおりです。

- ❏ *メモリ管理*関数 - メモリのパケットの割り当てと解放を行うことができます。デフォルトでは、これらの関数は 2K バイトのメモリを使用します。この量をリンク時に変更するには、リンカを起動するときに、-heap オプションを指定し、このオプションの直後に必要なヒープ・サイズを定数として指定します。
- ❏ *文字列変換* 関数 - 文字列を数値表現に変換します。
- ❏ *検索*関数および*ソート*関数 - 配列の検索とソートを行います。
- ❏ *シーケンス生成*関数 - 疑似乱数シーケンスを生成します。シーケンスの開始点を選択することもできます。
- ❏ *プログラム出口* 関数 - プログラムを正常終了または異常終了させることができます。
- ❏ *整数算術* - C 言語の標準的なものではありません。

これらの関数を表 8-3 (h)（8-36 ページ）に示します。

stdlib.h にも必要な C の汎用ユーティリティ関数の far バージョンが含まれています。これらの関数を表 8-3 (k)（8-40 ページ）に示します。

8.4.15 文字列関数（string.h/cstring）

string.h/cstring ヘッダでは、文字配列（文字列）に関して以下の作業を実行するための標準関数を宣言します。

- ❏ 文字列の一部あるいは全体の移動やコピー
- ❏ 文字列の連結
- ❏ 文字列の比較
- ❏ 文字や他の文字列における文字列の検索
- ❏ 文字列の長さの検出

C/C++ では、すべての文字列の最後は 0（ヌル）文字です。str*xxx* という文字列関数は、すべてこの規則に従って処理を行います。string.h/cstring には別の関数も宣言されていて、この関数を使用するとオブジェクトの最後が 0 になっていない任意のバイト・シーケンス（データ・オブジェクト）に対して、これと同じ処理ができます。これらの関数の名前は mem*xxx* などです。

文字列の移動やコピーを行う関数を使用するときは、デスティネーションに結果を格納するだけの十分な大きさがあることを確認しておいてください。

これらの関数を表 8-3 (i)（8-38 ページ）に示します。

string.h ヘッダにも C の文字列関数の far バージョンが含まれます。これらの関数を表 8-3 (k)（8-40 ページ）に示します。

8.4.16 時間関数（time.h/ctime）

time.h/ctime ヘッダでは、1 つのマクロ、いくつかの型、および日付と時刻を操作する関数を宣言します。時刻を表すには 2 つの方法があります。

- ❏ *time_t* 型の算術値。この方法で表されるときは、時刻は 1900 年 1 月 1 日午前 0 時からの秒数で表されます。time_t 型は unsigned long 型と同義です。

- ❏ *struct_tm* 型の構造体。この構造体には、年、月、日、時、分、秒を組み合わせて時刻を表すメンバが含まれています。このような方法で表した時間を詳細時刻と呼びます。この構造体のメンバは次のとおりです。

```
int    tm_sec;     /* seconds after the minute (0-59) */
int    tm_min;     /* minutes after the hour (0-59)   */
int    tm_hour;    /* hours after midnight (0-23)     */
int    tm_mday;    /* day of the month (1-31)         */
int    tm_mon;     /* months since January (0-11)     */
int    tm_year;    /* years since 1900                */
int    tm_wday;    /* days since Saturday (0-6)       */
int    tm_yday;    /* days since January 1 (0-365)    */
int    tm_isdst;   /* daylight savings time flag      */
```

時間は、time_t または struct_tm のいずれの型で表される場合でも、次のように異なる計測基準で表すことができます。

- カレンダ時は、グレゴリオ暦で現在の日付と時刻を表します。
- 地方時は、特定のタイム・ゾーンに関して表されたカレンダ時です。

地方時は夏時間に調整できます。当然、地方時はタイム・ゾーンによって異なります。time.h/ctime ヘッダ は、tmzone という構造体型およびこの _tz という型の変数を宣言します。実行時にまたは tmzone.c を編集し初期設定を変更することにより、この構造体を変更し、タイム・ゾーンを変更できます。デフォルトのタイム・ゾーンは、米国の CST（中部標準時）です。

time.h/ctime 内のすべての関数の基本となるのは、clock と time の 2 つのシステム関数です。time は現在の時刻（time_t フォーマット）を示し、clock はシステム時刻（任意の単位）を示します。clock により戻される値は、CLOCKS_PER_SEC マクロで除算して秒数に変換できます。これらの関数と CLOCKS_PER_SEC マクロはシステム固有のものなので、ライブラリにはスタブだけが入っています。その他の時間関数を使用するには、これらの関数をシステム用にカスタマイズする必要があります。

これらの関数を表 8-3 (j)（8-39 ページ）に示します。

注： ユーザ固有の clock 関数の記述

clock 関数はホスト・システム固有のものです。したがって独自で clock 関数を記述する必要があります。また、clock() が戻す値 - クロック・ティック数 - を CLOCKS_PER_SEC で割ることで、値（秒数）を生成できるように、clock の単位に応じてマクロ CLOCKS_PER_SEC を定義する必要があります。

8.4.17 例外処理（exception と stdexcept）

例外処理はサポートされていません。C++ 専用の exception と stdexcept のインクルード・ファイルは空です。

8.4.18 動的メモリ管理（new）

C++ 専用の new ヘッダは、new、new[]、delete、delete[]、およびその配置バージョンの関数を定義します。

メモリ割り当て時のエラー回復をサポートするために new_handler 型と set_new_handler() 関数も提供されています。

8.4.19 ランタイム型情報（typeinfo）

C++ 専用の typeinfo ヘッダは、実行時に C++ 型情報を表すのに使用される type_info 構造体を定義します。

8.5 ランタイムサポート関数およびマクロのまとめ

表 8-3 に、TMS320C28x ANSI/ISO C/C++ コンパイラに付属するランタイムサポート・ヘッダ・ファイルをアルファベット順にまとめています。説明されている関数の大部分は、ANSI/ISO C 規格に従って用意されていて、規格で規定されているとおりに動作します。

表 8-3 に掲載されている関数とマクロは、8.6 節「ランタイムサポート関数およびマクロについて」(8-43 ページ) で詳しく説明しています。関数またはマクロの詳細は、指示されているページを参照してください。

表 8-3. ランタイムサポート関数およびマクロのまとめ

(a) エラー・メッセージ・マクロ（assert.h/cassert）

マクロ	説明	ページ
void **assert**(int expression);	診断メッセージをプログラムに挿入します。	8-46

(b) 文字の判別と変換関数（ctype.h/cctype）

関数	説明	ページ
int **isalnum**(int c);	英数字 ASCII 文字かどうか、c をテストします。	8-72
int **isalpha**(int c);	英字 ASCII 文字かどうか、c をテストします。	8-72
int **isascii**(int c);	ASCII 文字かどうか、c をテストします。	8-72
int **iscntrl**(int c);	制御文字かどうか、c をテストします。	8-72
int **isdigit**(int c);	数字かどうか、c をテストします。	8-72
int **isgraph**(int c);	空白以外の印字文字かどうか、c をテストします。	8-72
int **islower**(int c);	ASCII の英小文字かどうか、c をテストします。	8-72
int **isprint**(int c);	印刷可能な ASCII 文字かどうか、c をテストします（スペースを含む)。	8-72
int **ispunct**(int c);	ASCII の句読点文字かどうか、c をテストします。	8-72
int **isspace**(int c);	ASCII のスペース・バー、タブ（水平か垂直)、復帰、書式送り、または改行文字かどうか、c をテストします。	8-72
int **isupper**(int c);	ASCII の英大文字かどうか、c をテストします。	8-72
int **isxdigit**(int c);	16 進数字かどうか、c をテストします。	8-72
int **toascii**(int c);	c を有効な ASCII 値にマスクします。	8-108
int **tolower**(int c);	c が大文字の場合は、小文字に変換します。	8-109
int **toupper**(int c);	c が小文字の場合は、大文字に変換します。	8-109

注： -x オプションが指定されている場合、ctype.h/cctype に含まれている関数はインライン展開されます。

(c)　浮動小数点算術関数（math.h/cmath）

関数	説明	ページ
double **acos**(double x);	x のアーク・コサインを戻します。	8-44
double **asin**(double x);	x のアーク・サインを戻します。	8-45
double **atan**(double x);	x のアーク・タンジェントを戻します。	8-47
double **atan2**(double y, double x);	y/x の逆タンジェントを戻します。	8-47
double **ceil**(double x);	≥ x の条件を満たす最小の整数を戻します。-x オプションが指定されている場合、インライン展開します。	8-51
double **cos**(double x);	x のコサインを戻します。	8-53
double **cosh**(double x);	x のハイパボリック・コサインを戻します。	8-53
double **exp**(double x);	x の指数関数を戻します。-x0 オプションが指定されている場合を除き、インライン展開します。	8-56
double **fabs**(double x);	x の絶対値を戻します。	8-57
double **floor**(double x);	≤ x の条件を満たす最大の整数を戻します。-x オプションが指定されている場合、インライン展開します。	8-63
double **fmod**(double x, double y);	x/y の浮動小数点の剰余を戻します。-x オプションが指定されている場合、インライン展開します。	8-64
double **frexp**(double value, int *exp);	浮動小数点数を正規化した小数部と 2 の整数乗に分けます。	8-68
double **ldexp**(double x, int exp);	x と 2 の累乗を乗算します。	8-74
double **log**(double x);	x の自然対数を戻します。	8-75
double **log10**(double x);	底が 10 の x の（常用）対数を戻します。	8-75
double **modf**(double value, double *iptr);	値を、符号付き整数と符号付き小数に分けます。	8-81
double **pow**(double x, double y);	x の y 乗を戻します。	8-82
double **sin**(double x);	x のサインを戻します。	8-90
double **sinh**(double x);	x のハイパボリック・サインを戻します。	8-90
double **sqrt**(double x);	x の負でない平方根を戻します。	8-91
double **tan**(double x);	x のタンジェントを戻します。	8-106
double **tanh**(double x);	x のハイパボリック・タンジェントを戻します。	8-107

(d)　低レベル入出力関数（file.h）

関数	説明	ページ
int **add_device**(char *name, unsigned flags, int (*dopen)(), int (*dclose)(), int (*dread)(), int (*dwrite)(), fpos_t (*dlseek)(), int (*dunlink)(), int (*drename)());	デバイス・レコードをデバイス・テーブルに追加します。	8-12

(e)　非ローカル・ジャンプ・マクロと関数（setjmp.h/csetjmp）

マクロまたは関数	説明	ページ
int **setjmp**(jmp_buf env);	後で longjmp 関数で使用できるように呼び出し環境を保存します。-x オプションが指定されている場合、インライン展開します。	8-88
void **longjmp**(jmp_buf env, int _val);	jmp_buf 引数を使用して、以前に保管されたプログラム環境を復元します。	8-88

(f)　可変引数マクロ（stdarg.h/cstdarg）

マクロ	説明	ページ
type **va_arg**(_ap, type);	可変引数リスト内の型 *type* の次の引数にアクセスします。	8-110
void **va_end**(_ap);	va_arg を使用した後で呼び出しメカニズムをリセットします。	8-110
void **va_start**(_ap, parmN);	可変引数リスト内の第 1 オペランドを指すように ap を初期化します。	8-110

(g)　C/C++ 入出力関数（stdio.h/cstdio）

関数	説明	ページ
void **clearerr**(FILE *_fp);	_fp が指すストリームの EOF 標識とエラー標識をクリアします。	8-52
int **fclose**(FILE *_fp);	_fp が指すストリームをフラッシュし、そのストリームに関連したファイルをクローズします。	8-61
int **feof**(FILE *_fp);	_fp が指すストリームの EOF 標識をテストします。	8-61
int **ferror**(FILE *_fp);	_fp が指すストリームのエラー標識をテストします。	8-61
int **fflush**(register FILE *_fp);	_fp が指すストリームの入出力バッファをフラッシュします。	8-62

(g) C/C++ 入出力関数（stdio.h/cstdio）（続き）

関数	説明	ページ
int **fgetc**(register FILE *_fp);	_fp が指すストリーム内の次の文字を読み取ります。	8-62
int **fgetpos**(FILE *_fp, fpos_t *_pos);	_fp が指すストリームのファイル位置標識の現行値を _pos が指すオブジェクトに保存します。	8-62
char ***fgets**(char *_ptr, register int _size, register FILE *_fp);	_fp が指すストリームから配列 _ptr に、次の _size から 1 引いた数の文字を読み取ります。	8-63
FILE ***fopen**(const char *_fname, const char *_mode);	_fname が指すファイルをオープンします。_mode は、ファイルをオープンする方法を記述する文字列を指します。	8-64
int **fprintf**(FILE *_fp, const char *_format, ...);	_fp が指すストリームに書き込みます。	8-64
int **fputc**(int _c, register FILE *_fp);	_fp が指すストリームに 1 文字 _c を書き込みます。	8-65
int **fputs**(const char *_ptr, register FILE *_fp);	_fp が指すストリームに _ptr が指す文字列を書き込みます。	8-65
size_t **fread**(void *_ptr, size_t _size, size_t _count, FILE *_fp);	_fp が指すストリームから読み取り、_ptr が指す配列に入力データを保存します。	8-66
FILE ***freopen**(const char *_fname, const char *_mode, register FILE *_fp);	_fp が指すストリームを使用して、_fname が指すファイルをオープンします。_mode は、ファイルをオープンする方法を記述する文字列を指します。	8-67
int **fscanf**(FILE *_fp, const char *_fmt, ...);	_fp が指すストリームから、書式化された入力データを読み取ります。	8-68
int **fseek**(register FILE *_fp, long _offset, int _ptrname);	_fp が指すストリームのファイル位置標識を設定します。	8-69
int **fsetpos**(FILE *_fp, const fpos_t *_pos);	_fp が指すストリームのファイル位置標識を _pos に設定します。ポインタ _pos の値は、同じストリームに対する fgetpos() からの値を指定する必要があります。	8-69
long **ftell**(FILE *_fp);	_fp が指すストリームのファイル位置標識の現行値を取得します。	8-69
size_t **fwrite**(const void *_ptr, size_t _size, size_t _count, register FILE *_fp);	_ptr が指すメモリから、_fp が指すストリームにデータ・ブロックを書き込みます。	8-70
int **getc**(FILE *_fp);	_fp が指すストリーム内の次の文字を読み取ります。	8-70

(g) C/C++ 入出力関数（stdio.h/cstdio）（続き）

関数	説明	ページ
int **getchar**(void);	fgetc() を呼び出し、stdin を引数として指定するマクロです。	8-70
char ***gets**(char *_ptr);	stdin を入力ストリームとして使用する fgets() と同じ機能を実行します。	8-71
void **perror**(const char *_s);	_s のエラー番号を文字列にマップし、エラー・メッセージを出力します。	8-81
int **printf**(const char *_format, ...);	fprintf() と同じ機能を実行しますが、stdout をその出力ストリームとして使用します。	8-82
int **putc**(int _x, FILE *_fp);	fputc() と同じように実行するマクロです。	8-82
int **putchar**(int _x);	fputc() を呼び出し、stdout を出力ストリームとして使用するマクロです。	8-83
int **puts**(const char *_ptr);	_ptr が指す文字列を stdout に書き込みます。	8-83
int **remove**(const char *_file);	_file が指す名前のファイルを、その名前では使用できないようにします。	8-86
int **rename** (const char *_old_name, const char *_new_name);	_old_name が指す名前のファイルを、_new_name が指す名前で認識されるようにします。	8-86
void **rewind** (register FILE *_fp);	_fp が指すストリームのファイル位置標識を、ファイルの先頭に設定します。	8-86
int **scanf**(const char *_fmt, ...);	fscanf() と同じ機能を実行しますが、stdin から入力データを読み取ります。	8-87
void **setbuf**(register FILE *_fp, char *_buf);	値を戻しません。setbuf() は setvbuf() の制限付きバージョンであり、バッファを定義し、ストリームに関連付けます。	8-87
int **setvbuf** (register FILE *_fp, register char *_buf, register int _type, register size_t _size);	バッファを定義してストリームに関連付けます。	8-87
int **sprintf**(char *_string, const char *_format, ...);	fprintf() と同じ機能を実行しますが、_string が指す配列に書き込みます。	8-91
int **sscanf** (const char *_str, const char *_fmt, ...);	fscanf() と同じ機能を実行しますが、_str が指す文字列から読み取ります。	8-92
FILE ***tmpfile**(void);	一時ファイルを作成します。	8-108
char ***tmpnam**(char *_s);	有効なファイル名である（つまりファイル名がまだ使用されていない）文字列を生成します。	8-108
int **ungetc**(int _c, register FILE *_fp);	_c が指定する文字を、_fp が指す入力ストリームに戻します。	8-109

(g) C/C++ 入出力関数（stdio.h/cstdio）（続き）

関数	説明	ページ
int **vfprintf** (FILE *_fp, const char *_format, va_list _ap);	fprintf() と同じ機能を実行しますが、引数リストを _ap に置き換えます。	8-111
int **vprintf**(const char *_format, va_list _ap);	printf() と同じ機能を実行しますが、引数リストを _ap に置き換えます。	8-111
int **vsprintf** (char *_string, const char *_format, va_list _ap);	sprintf() と同じ機能を実行しますが、引数リストを _ap に置き換えます。	8-112

(h) 汎用ユーティリティ （stdlib.h/cstdlib）

関数	説明	ページ
void **abort**(void);	プログラムを異常終了させます。	8-43
int **abs**(int j);	j の絶対値を戻します。-x0 が指定されている場合を除き、インライン展開します。	8-44
void **atexit**(void (*fun)(void));	プログラムの正常終了時に引数を指定せずに呼び出される fun が指す関数を登録します。	8-48
double **atof**(const char *st);	文字列を浮動小数点値に変換します。-x が指定されている場合、インライン展開します。	8-48
int **atoi**(register const char *st);	文字列を整数値に変換します。	8-48
long **atol**(register const char *st);	文字列を long int の値に変換します。-x が指定されている場合、インライン展開します。	8-48
long long **atoll**(register const char *st);	文字列を long long int の値に変換します。-pi が指定されている場合を除き、インライン展開します。	8-48
void ***bsearch**(register const void *key, register const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));	nmemb 個のオブジェクトの配列から、key が指すオブジェクトを検索します。	8-50
void ***calloc**(size_t num, size_t size);	各々が size バイトの num 個のオブジェクト用にメモリの割り当てとクリアを行います。	8-51
div_t **div**(register int numer, register int denom);	numer を denom で除算し、商と剰余を生成します。	8-55
void **exit**(int status);	プログラムを正常終了します。	8-56
void **free**(void *packet);	malloc、calloc、realloc のいずれかによって割り当てられたメモリ空間を解放します。	8-66
int **free_memory**(void);	割り当てで使用できる動的なメモリの合計を戻します。	8-67

(h)　汎用ユーティリティ （stdlib.h/cstdlib）（続き）

関数	説明	ページ
char ***getenv**(const char *_string);	_string に関連付けられている環境変数の情報を戻します。	8-71
long **labs**(long i);	i の絶対値を戻します。-x0 が指定されている場合を除き、インライン展開します。	8-73
ldiv_t **ldiv**(long numer, long denom);	numer を denom で除算します。	8-74
long long **llabs**(long long i);	i の絶対値を戻します。インライン展開します。	8-73
lldiv_t **lldiv**(long long numer, long long denom);	numer を denom で除算します。	8-74
int **ltoa**(long val, char *buffer);	val を等価の文字列に変換します。	8-76
void ***malloc**(size_t size);	size バイトのオブジェクト用にメモリを割り当てます。	8-76
int **max_free**(void);	動的なメモリ割り当てが可能な最大のサイズを戻します。	8-77
void **qsort**(void *_base, size_t nmemb, size_t size, int (*compar) (void));	nmemb 個のメンバの配列をソートします。*base* はソートされていない配列の最初のメンバを指し、*size* は各メンバのサイズを指定します。	8-83
int **rand**(void);	0 ～ RAND_MAX の範囲の整数の疑似乱数シーケンスを戻します。	8-84
void ***realloc**(void *packet, size_t size);	割り当てられたメモリ空間のサイズを変更します。	8-85
void **srand**(unsigned int seed);	乱数発生ルーチンをリセットします。	8-92
double **strtod**(const char *st, char **endptr);	文字列を浮動小数点値に変換します。	8-104
long **strtol**(const char *st, char **endptr, int base);	文字列を long int の値に変換します。	8-104
long long **strtoll**(const char *st, char **endptr, int base);	文字列を long long int の値に変換します。	8-104
unsigned long **strtoul**(const char *st, char **endptr**,** int base)**;**	文字列を unsigned long int の値に変換します。	8-104
unsigned long long **strtoull**(const char *st, char **endptr**,** int base)**;**	文字列を unsigned long long int の値に変換します。	8-104

(i)　文字列関数（string.h/cstring）

関数	説明	ページ
void ***memchr**(const void *cs, int c, size_t n);	cs の先頭の n 個の文字の中で c が最初に現れる位置を検出します。-x が指定されている場合、インライン展開します。	8-77
int **memcmp**(const void *cs, const void *ct, size_t n);	cs の先頭の n 個の文字を ct と比較します。-x が指定されている場合、インライン展開します。	8-78
void ***memcpy**(void *s1, const void *s2, register size_t n);	n 個の文字を s2 から s1 にコピーします。	8-78
void ***memmove**(void *s1, const void *s2, size_t n);	n 個の文字を s2 から s1 に移動します。	8-79
void ***memset**(void *mem, register int ch, register size_t length);	mem の先頭の length 個の文字に ch の値をコピーします。-x が指定されている場合、インライン展開します。	8-79
char ***strcat**(char *string1, const char *string2);	string2 を string1 の末尾に付加します。	8-92
char ***strchr**(const char *string, int c);	string の中で文字 c が最初に現れる位置を検出します。-x が指定されている場合、インライン展開します。	8-93
int **strcmp**(register const char *string1, register const char *string2);	文字列を比較し、値を戻します。string1 が string2 より小さい場合は <0、string1 が string2 と等しい場合は 0、string1 が string2 より大きい場合は >0 です。-x が指定されている場合、インライン展開します。	8-94
int **strcoll**(const char *string1, const char *string2);	文字列を比較し、値を戻します。string1 が string2 より小さい場合は <0、string1 が string2 と等しい場合は 0、string1 が string2 より大きい場合は >0 です。	8-94
char ***strcpy**(register char *dest, register const char *src);	文字列 src を dest にコピーします。-x が指定されている場合、インライン展開します。	8-95
size_t **strcspn**(register const char *string, const char *chs);	chs 内にない文字だけから構成された string の先頭部分の長さを戻します。	8-95
char ***strerror**(int errno);	errno のエラー番号をエラー・メッセージ文字列にマップします。	8-96
size_t **strlen**(char *string);	文字列の長さを戻します。	8-98
char ***strncat**(char *dest, const char *src, register size_t n);	最大で n 個の文字を src から dest に付加します。	8-98
int **strncmp**(const char *string1, const char *string2, size_t n);	2 つの文字列の最大 n 個の文字を比較します。-x が指定されている場合、インライン展開します。	8-99

(i) 文字列関数（string.h/cstring）（続き）

関数	説明	ページ
char ***strpbrk**(const char *string, const char *chs);	string の中で、chs の*いずれかの*文字が最初に現れる位置を検索します。	8-101
char ***strrchr**(const char *string, int c);	string の中で文字 c が最後に現れる位置を検出します。-x が指定されている場合、インライン展開します。	8-102
size_t **strspn**(register const char *string, const char *chs);	chs の文字だけで構成された string の先頭部分の長さを戻します。	8-102
char ***strstr**(register const char *string1, const char *string2);	string1 を検索して string2 が最初に現れる位置を検出します。	8-103
char ***strtok**(char *str1, const char *str2);	str1 を、str2 の文字で区切られる一連のトークンに分割します。	8-105
size_t **strxfrm**(register char *to, register const char *from, register size_t n);	n 個の文字を *from* から *to* に変換します。	8-106

(j) 時間関数（time.h/ctime）

関数	説明	ページ
char ***asctime**(const struct tm *timeptr);	時間を文字列に変換します。	8-45
clock_t **clock**(void);	プロセッサが使用した時間を判定します。	8-52
char ***ctime**(const time_t *timer);	カレンダ時を地方時に変換します。	8-54
double **difftime**(time_t time1, time_t time0);	2 つのカレンダ時の差を戻します。	8-54
struct tm ***gmtime**(const time_t *timer);	カレンダ時をグリニッジ標準時に変換します。	8-72
struct tm ***localtime**(const time_t *timer);	カレンダ時を地方時に変換します。	8-74
time_t **mktime**(register struct tm *tptr);	地方時をカレンダ時に変換します。	8-80
size_t **strftime**(char *out, size_t maxsize, const char *format, const struct tm *time);	時間を文字列に形式設定します。	8-96
time_t **time**(time_t *timer);	現在のカレンダ時を戻します。	8-107

(k) 関数の far バージョンのまとめ

関数	説明	ページ
int **far_atoi**(const far char *st);	atoi() 関数の far バージョン	8-48
long **far_atol**(const far char *st);	atol() 関数の far バージョン	8-48
long long **far_atoll**(const far char *st);	atoll() 関数の far バージョン	8-48
double **far_atof**(const far char *st);	atof() 関数の far バージョン	8-48
far void ***far_bsearch**(const far void *key, const far void *base, size_t nmemb, size_t size, int (*compar) (const far void *, const far void *));	bsearch() 関数の far バージョン	8-50
far void ***far_calloc**(unsigned long num, unsigned long size);	calloc() 関数の far バージョン	8-51
void **far_free** (far void *ptr);	free() 関数の far バージョン	8-66
long **far_free_memory**(void);	free_memory() 関数の far バージョン	8-67
double **far_frexp**(double x, far int *exp);	frexp() 関数の far バージョン	8-68
void **far_longjmp**(far_jmp_buf env, int val);	longjmp() 関数の far バージョン	8-88
far void ***far_malloc**(unsigned long size);	malloc() 関数の far バージョン	8-76
long **far_max_free**(void);	max_free() 関数の far バージョン	8-77
far void ***far_memchr**(const far void *s, int c, size_t n);	memchr() 関数の far バージョン	8-77
int **far_memcmp**(const far void *s1, const far void *s2, size_t n);	memcmp() 関数の far バージョン	8-78
void far ***far_memcpy**(far void *_s1, far const void *_s2, size_t _n);	memcpy() 関数の far バージョン	8-78
far void ***far_memlcpy**(far void *to, const far void *from, unsigned long n);	ブロックが 64K ワードを超えるメモリ・ブロックのコピー（オーバーラップなし）。	8-59
far void ***far_memlmove**(far void *to, const far void *from, unsinged long n);	ブロックが 64K ワードを超えるメモリ・ブロックのコピー（オーバーラップあり）。	8-60
void far ***far_memmove**(far void *s1, far const void *s2, size_t n);	memmove() 関数の far バージョン	8-79
far void ***far_memset**(far void *s, int c, size_t n);	memset() 関数の far バージョン	8-79
double **far_modf**(double x, far double *y);	modf() 関数の far バージョン	8-81

(k)　関数の far バージョンのまとめ（続き）

関数	説明	ページ
void **far_qsort**(far void *base, size_t nmemb, size_t size, int (*compar) (const far void *, const far void *));	qsort() 関数の far バージョン	8-83
far void ***far_realloc**(far void *prt, unsigned long size);	realloc() 関数の far バージョン	8-85
int **far_setjmp**(far_jmp_buf env);	setjmp() 関数の far バージョン	8-88
far char ***far_strcat**(far char *s1, const far char *s2);	strcat() 関数の far バージョン	8-92
far char ***far_strchr**(const far char *s, int c);	strchr() 関数の far バージョン	8-93
int **far_strcmp**(const far char *s1, const far char *s2);	strcmp() 関数の far バージョン	8-94
int **far_strcoll**(const far char *s1, const far char *s2);	strcoll() 関数の far バージョン	8-94
char far ***far_strcpy**(char far *s1, far const char *s2, size_t n);	strcpy() 関数の far バージョン	8-95
size_t **far_strcspn**(const far char *s1, const far char *s2);	strcspn() 関数の far バージョン	8-95
far char ***far_strerror**(int errno);	strerror() 関数の far バージョン	8-96
size_t **far_strlen**(const far char *s);	strlen() 関数の far バージョン	8-98
far char ***far_strncat**(far char *s1, const far char *s2, size_t n);	strncat() 関数の far バージョン	8-98
int **far_strncmp**(const far char *s1, const far char *s2, size_t n);	strncmp() 関数の far バージョン	8-99
far char ***far_strncpy**(far char *s1, far const char *s2, size_t n);	strncpy() 関数の far バージョン	8-100
far char ***far_strpbrk**(const far char *s1, const far char *s2);	strpbrk() 関数の far バージョン	8-101
far char ***far_strrchr**(const far char *s, int c);	strrchr() 関数の far バージョン	8-102
size_t **far_strspn**(const far char *s1, const far char *s2);	strspn() 関数の far バージョン	8-102
far char ***far_strstr**(const far char *s1, const far char *s2);	strstr() 関数の far バージョン	8-103

(k)　関数の far バージョンのまとめ（続き）

関数	説明	ページ
double **far_strtod**(const far char *st, far char **endprt);	strtod() 関数の far バージョン	8-104
far char ***far_strtok**(far char *s1, const far char *s2);	strtok() 関数の far バージョン	8-105
long **far_strtol**(const far char *st, far char **endprt, int base);	strtol() 関数の far バージョン	8-104
long long **far_strtoll**(const far char *st, far char **endprt, int base);	strtoll() 関数の far バージョン	8-104
unsigned long **far_strtoul**(const far char *st, far char **endprt, int base);	strtoul() 関数の far バージョン	8-104
unsigned long long **far_strtoull**(const far char *st, far char **endprt, int base);	strtoull() 関数の far バージョン	8-104
size_t **far_strxfrm**(far char *s1, const far char *s2, size_t n);	strxfrm() 関数の far バージョン	8-106

8.6 ランタイムサポート関数およびマクロについて

ここでは、ランタイムサポート関数およびマクロについて説明します。関数またはマクロごとに、C と C++ の両方の構文が記載されています。しかし、関数とマクロは C のヘッダ・ファイルから生成されているので、プログラムは C コードでのみ記述されています。C++ コードでは、同じプログラムであっても、ヘッダ・ファイルで宣言される型と関数が std ネームスペースに導入されている点では異なっています。

abort 中止

C の構文

#include <stdlib.h>

void abort(void)**;**

C++ の構文

#include <cstdlib>

void std::abort(void)**;**

定義される場所

rts.src の exit.c

説明

abort 関数は、プログラムを終了させます。

例

```
void abort(void)
{
    exit(EXIT_FAILURE);
}
```

exit 関数（8-56 ページ）を参照してください。

far バージョン

なし

abs/labs/llabs 絶対値

C の構文

```
#include <stdlib.h>
```

int abs(int i)**;**
long labs(long j)**;**
long long llabs(long long k)**;**

C++ の構文

```
#include <cstdlib>
```

int std::abs(int i)**;**
long std::labs(long j)**;**
long long std::llabs(long long k)**;**

定義される場所

rts.src の abs.c

説明

C/C++ コンパイラは、以下のように、整数の絶対値を戻す 3 つの関数をサポートしています。

- ❑ abs 関数は、整数 i の絶対値を戻します。
- ❑ labs 関数は、長整数 j の絶対値を戻します。
- ❑ llabs 関数は、倍長整数 k の絶対値を戻します。

far バージョン

なし

acos アーク・コサイン

C の構文

```
#include <math.h>
```

double acos(double x)**;**

C++ の構文

```
#include <cmath>
```

double std::acos(double x)**;**

定義される場所

rts.src の acos.c

説明

acos 関数は、浮動小数点引数 x のアーク・コサインを戻します。x の範囲は、[-1,1] とします。戻り値は、範囲 [0,π] のラジアンの角度です。

例

```
double realval, radians;

return (rrealval = 1.0;
radians = acos(realval);
return (radians);    /* acos return pi/2 */
```

far バージョン

なし

asctime　内部時間から文字列への変換

C の構文

#include　<time.h>

char *asctime(const struct tm *timeptr**);**

C++ の構文

#include　<ctime>

char *std::asctime(const struct tm *timeptr**);**

定義される場所

rts.src の asctime.c

説明

asctime 関数は、詳細時刻を以下の形式の文字列に変換します。

```
Mon Jan 11 11:18:36 1988 \n\0
```

この関数は、変換された文字列へのポインタを戻します。

time.h/ctime ヘッダで宣言と定義が行われる関数と型の詳細は、8.4.16 項「時間関数（time.h/ctime）」（8-29 ページ）を参照してください。

far バージョン

なし

asin　アーク・サイン

C の構文

#include　<math.h>

double asin(double x)**;**

C++ の構文

#include　<cmath>

double std::asin(double x)**;**

定義される場所

rts.src の asin.c

説明

asin 関数は、浮動小数点引数 x のアーク・サインを戻します。x の範囲は、[-1, 1] とします。戻り値は、範囲 [-π/2, π/2] のラジアンの角度です。

例

```
double realval, radians;
realval = 1.0;
radians = asin(realval);  /* asin returns pi/2 */
```

far バージョン

なし

assert　診断情報の挿入

C の構文

#include <assert.h>

void assert(int expr)**;**

C++ の構文

#include <cassert>

void assert(int expr)**;**

定義される場所

マクロとしての assert.h/cassert

説明

assert マクロは、式をテストします。式の値に基づき、メッセージを発行して実行を打ち切るか、実行を継続します。このマクロはデバッグするときに便利です。

- ❑ expr が偽の場合、assert マクロは、失敗した特定の呼び出しに関する情報を標準出力に書き込み、実行を打ち切ります。
- ❑ expr が真の場合、assert マクロは何も実行しません。

assert マクロを宣言するヘッダ・ファイルは、別のマクロ NDEBUG を参照します。assert.h ヘッダがソース・ファイルに組み込まれるときに NDEBUG をマクロ名として定義した場合、assert マクロは次のように定義されます。

```
#define assert(ignore)
```

説明

次の例では、整数 i を別の整数 j で除算します。0 による除算は不正な演算なので、この例では除算の前に assert マクロで j をテストしています。このコードを実行したときに j = = 0 の場合、assert はメッセージを発行し、プログラムを中止します。

```
int   i, j;
assert(j);
q = i/j;
```

far バージョン

なし

atan　極座標のアーク・タンジェント

C の構文

#include　<math.h>

double atan(double x)**;**

C++ の構文

#include　<cmath>

double std::atan(double x)**;**

定義される場所

rts.src の atan.c

説明

atan 関数は、浮動小数点引数 x のアーク・タンジェントを戻します。戻り値は、範囲 [-π/2, π/2] のラジアンの角度です。

例

```
double realval, radians;

realval = 0.0;
radians = atan(realval);     /* return value = 0 */
```

far バージョン

なし

atan2　デカルト座標のアーク・タンジェント

C の構文

#include　<math.h>

double atan2(double y, double x)**;**

C++ の構文

#include　<cmath>

double std::atan2(double y, double x)**;**

定義される場所

rts.src の atan2.c

説明

atan2 関数は、y/x の逆タンジェントを戻します。この関数は、これらの引数の符号を使用して、戻り値の座標象限を判定します。どちらの引数も 0 にすることはできません。戻り値は、範囲 [-π, π] のラジアンの角度です。

例

```
double rvalu, rvalv;
double radians;

rvalu  = 0.0;
rvalv  = 1.0;
radians = atan2(rvalr, rvalu);   /* return value = 0 */
```

far バージョン

なし

atexit

Exit () から呼び出される関数の登録

C の構文

```
#include    <stdlib.h>

void atexit(void (*fun)(void));
```

C++ の構文

```
#include    <cstdlib>

void std::atexit(void (*fun)(void));
```

定義される場所

rts.src の exit.c

説明

atexit 関数は、プログラムの正常終了時に引数なしで呼び出される（*fun* が指す）関数を登録します。最大で 32 個までの関数を登録できます。

exit 関数の呼び出しによりプログラムが終了すると、登録された関数は、登録順とは逆の順序で引数なしで呼び出されます。

far バージョン

なし

atof/atoi/atol/atoll

文字列から数値への変換

C の構文

```
#include    <stdlib.h>

double atof(const char *st);
int atoi(const char *st);
long atol(const char *st);
long long atoll(const char *st);
```

C++ の構文

```
#include    <cstdlib>

double std::atof(const char *st);
int std::atoi(const char *st);
long std::atol(const char *st);
long long std::atoll(char *st);
```

定義される場所

rts.src の atof.c、atoi.c、atol.c、および atoll.c
rts.src の faratof.c、faratoi.c、faratol.c、および faratoll.c

説明

次の 4 つの関数は、文字列を数値表現に変換します。

- atof 関数は文字列を浮動小数点値に変換します。引数 st は文字列を指します。文字列の形式は、次のとおりです。

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

- atoi 関数は文字列を整数に変換します。引数 st は文字列を指します。文字列の形式は、次のとおりです。

  [*space*] [*sign*] *digits*

- atol 関数は文字列を長整数に変換します。引数 st は文字列を指します。文字列の形式は、次のとおりです。

  [*space*] [*sign*] *digits*

- atoll 関数は文字列を倍長整数に変換します。引数 st は文字列を指します。文字列の形式は、次のとおりです。

  [*space*] [*sign*] *digits*

space は、スペース（文字）、水平タブか垂直タブ、復帰、書式送り、あるいは改行文字で表します。*space* の後は、オプションの *sign*、さらに数値の整数部を示す *digits* が続きます。atof ストリームでは、その後に数値の小数部が続き、オプションの *sign* をもつ指数部が続きます。

文字列は、数値以外の文字が現れた時点で終わります。

TMS320C28x C/C++ では、int と long は機能的には同じです。したがって、atoi 関数と atol 関数も機能的には同じになります。

これらの関数は、変換の結果発生したオーバーフローを処理しません。

far バージョン（C）

```
double far_atof(const far char *st);
int far_atoi(const far char *st);
long far_atol(const far char *st);
long long far_atoll(const far char *st);
```

bsearch　配列の検索

C の構文

```
#include    <stdlib.h>

void *bsearch(register const void *key, register const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
```

C++ の構文

```
#include    <cstdlib>

void *std::bsearch(register const void *key, register const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
```

定義される場所

rts.src の bsearch.c および farbsearch.c

説明

bsearch 関数は、nmemb 個のオブジェクトの配列から、key が指定するオブジェクトと一致するメンバを検索します。引数の base は配列の先頭のメンバを指します。size は各メンバのサイズ（バイト数）を指定します。

配列の内容は、昇順にソートされている必要があります。一致するメンバが存在する場合、この関数はその配列メンバへのポインタを戻します。一致するメンバが存在しない場合、この関数はヌル・ポインタ（0）を戻します。

引数 compar は、キーを配列要素と比較する関数を指します。比較関数は、次のように宣言します。

```
int  cmp(const void *ptr1, const void *ptr2);
```

cmp 関数は、ptr1 と ptr2 が指すオブジェクトを比較し、次の値のいずれかを戻します。

< 0	*ptr1 が *ptr2 より小さいとき。
0	*ptr1 が *ptr2 と等しいとき。
> 0	*ptr1 が *ptr2 より大きいとき。

far バージョン（C）

```
far void *far_bsearch(const far void *key, const far void *base, size_t nmemb,
size_t size, int (*compar) (const far void *, const far void *));
```

calloc　メモリの割り当ておよびクリア

C の構文

#include　<stdlib.h>

void *calloc(size_t num, size_t size)**;**

C++ の構文

#include　<cstdlib>

void *std::calloc(size_t num, size_t size)**;**

定義される場所

rts.src の memory.c および farmemory.c

説明

calloc 関数は、各々が size バイト（size は符号なし整数または size_t）の num 個のオブジェクト用に空間を割り当て、その空間へのポインタを戻します。この関数は、割り当てられたメモリをすべて 0 に初期化します。メモリを割り当てることができない場合（つまり、メモリ不足のとき）、この関数は、ヌル・ポインタ（0）を戻します。

calloc が使用するメモリは、特別なメモリ・プール（ヒープ）の中のメモリです。定数 __SYSMEM_SIZE により、ヒープのサイズは 1K ワードに定義されています。この量はリンク時に変更できます。そのためにはリンカを起動するときに、-heap オプションを指定し、このオプションの直後にヒープについて希望するサイズ（バイト数）を指定します。詳細は、7.1.4 項「動的なメモリ割り当て」（7-7 ページ）を参照してください。

例

次の例では、calloc ルーチンで 20 バイトを割り当て、クリアしています。

```
prt = calloc (10,2)  ;   /*Allocate and clear 20 bytes */
```

far バージョン（C）

far void *far_calloc(unsigned long num, unsigned long size)**;**

far バージョン（C++）

long std::far_calloc(unsigned long num, unsigned long size)**;**

ceil　切り上げ

C の構文

#include　<math.h>

double ceil(double x)**;**

C++ の構文

#include　<cmath>

double std::ceil(double x)**;**

定義される場所

rts.src の ceil.c

説明

ceil 関数は、x 以上の最小の整数を表す浮動小数点数を戻します。

例

```
extern double ceil();
double answer;
answer = ceil(3.1415);    /* answer = 4.0 */
answer = ceil(-3.5);      /* answer = -3.0 */
```

far バージョン

なし

clearerr　EOF およびエラー標識のクリア

C の構文

```
#include    <stdio.h>

void clearerr(FILE *_fp);
```

C++ の構文

```
#include    <cstdio>

void std::clearerr(FILE *_fp);
```

定義される場所　rts.src の clearerr.c

説明　clearerr 関数は、_fp が指すストリームの EOF 標識とエラー標識をクリアします。

far バージョン　なし

clock　プロセッサ時間

C の構文

```
#include    <time.h>

clock_t clock(void);
```

C++ の構文

```
#include    <ctime>

clock_t std::clock(void);
```

定義される場所　rts.src の clock.c

説明　clock 関数は、プロセッサが使用した時間の合計を判定します。プログラムの実行開始時以降のプロセッサ使用時間の概数値を戻します。戻り値をマクロ CLOCKS_PER_SEC の値で割ると、秒数に変換できます。

プロセッサ時間が利用不能または表現できない場合は、clock 関数は [(clock_t) -1] の値を戻します。

注： ユーザ固有の clock 関数の記述

clock 関数はホスト・システム固有のものです。したがって独自で clock 関数を記述する必要があります。また、clock() が戻す値 - クロック・ティック数 - を CLOCKS_PER_SEC で割ることで、値（秒数）を生成できるように、clock の単位に応じてマクロ CLOCKS_PER_SEC を定義する必要があります。

far バージョン　なし

cos コサイン

C の構文

#include <math.h>

double cos(double x)**;**

C++ の構文

#include <cmath>

double std::cos(double x)**;**

定義される場所

rts.src の cos.c

説明

cos 関数は、浮動小数点数 x のコサインを戻します。角度 x は、ラジアンで表します。引数の値が大きすぎると、有意性のある結果がほとんど得られないか、全く得られなくなります。

例

```
double radians, cval; /* cos returns cval */
radians = 3.1415927;
cval = cos(radians);  /* return value = -1.0 */
```

far バージョン

なし

cosh ハイパボリック・コサイン

C の構文

#include <math.h>

double cosh(double x)**;**

C++ の構文

#include <cmath>

double std::cosh(double x)**;**

定義される場所

rts.src の cosh.c

説明

cosh 関数は、浮動小数点数 x のハイパボリック・コサインを戻します。引数の値が大きすぎると、範囲エラーになります。

例

```
double x, y;

x = 0.0;
y = cosh(x);/* return value = 1.0 */
```

far バージョン

なし

ctime　カレンダ時

C の構文

#include　<time.h>

char *ctime(const time_t *timer)**;**

C++ の構文

#include　<ctime>

char *std::ctime(const time_t *timer)**;**

定義される場所

rts.src の ctime.c

説明

ctime 関数は、カレンダ時（timer が指す時刻）を文字列の形式の地方時に変換します。これは、次のように指定しても同じ結果になります。

```
asctime(localtime(timer))
```

この関数は、asctime 関数が戻すポインタを戻します。

time.h/ctime ヘッダで宣言と定義が行われる関数と型の詳細は、8.4.16 項「時間関数（time.h/ctime）」（8-29 ページ）を参照してください。

far バージョン

なし

difftime　時差

C の構文

#include　<time.h>

double difftime(time_t time1, time_t time0)**;**

C++ の構文

#include　<ctime>

double std::difftime(time_t time1, time_t time0)**;**

定義される場所

rts.src の difftime.c

説明

difftime 関数は、2 つのカレンダ時の差、time1 から time0 を引いた値を計算します。戻り値の単位は秒です。

time.h/ctime ヘッダで宣言と定義が行われる関数と型の詳細は、8.4.16 項「時間関数（time.h/ctime）」（8-29 ページ）を参照してください。

far バージョン

なし

div/ldiv/lldiv　除算

C の構文

#include <stdlib.h>

div_t div(int numer, denom)**;**
ldiv_t ldiv(long numer, denom)**;**
lldiv_t lldiv(long long numer, long long denom)**;**

C++ の構文

#include <cstdlib>

div_t std::div(int numer, denom)**;**
ldiv_t std::ldiv(long numer, denom)**;**
lldiv_t std::lldiv(long long numer, long long denom)**;**

定義される場所

rts.src の div.c および lldiv.c

説明

2 つの関数による整数の除算では、numer（分子）を denom（分母）で割った値が戻されます。これらの関数を使用すれば、1 回の演算で商と剰余の両方を得ることができます。

- ❑ div 関数は、整数の除算を行います。入力する引数は整数です。この関数は、商と剰余を型 div_t の構造体で戻します。構造体は次のように定義します。

```
typedef struct
{
     int  quot;        /*  quotient       */
     int  rem;         /* remainder       */
} div_t;
```

- ❑ ldiv 関数は、長整数の除算を行います。入力する引数は長整数です。この関数は、商と剰余を型 ldiv_t の構造体で戻します。構造体は次のように定義します。

```
typedef struct
{
     long int  quot;   /*  quotient       */
     long int  rem;    /* remainder       */
} ldiv_t;
```

- ❑ lldiv 関数は、long long 整数の除算を行います。入力する引数は long long 整数です。この関数は、商と剰余を型 lldiv_t の構造体で戻します。構造体は次のように定義します。

```
typedef struct
{
     long long  quot; /*  quotient       */
     long long  rem;  /* remainder       */
} lldiv_t;
```

どちらかのオペランド（両方ではない）が負の場合、商の符号は負になります。剰余の符号は、被除数の符号と同じになります。

TMS320C28x C/C++ では、int と long の型は同じです。したがって、ldiv 関数と div 関数も同じになります。

far バージョン なし

exit 正常な終了

C の構文

```
#include    <stdlib.h>

void exit(int status);
```

C++ の構文

```
#include    <cstdlib>

void std::exit(int status);
```

定義される場所 rts.src の exit.c

説明

exit 関数は、プログラムを正常に終了します。atexit 関数で登録された関数は、すべてその登録の順序とは逆の順序で呼び出されます。exit 関数は EXIT_FAILURE を値として使用できます（abort 関数（8-43 ページ）を参照）。

exit 関数を変更してアプリケーション固有のシャットダウン作業を行うことができます。変更されなければ、関数はシステムがリセットされるまで無限ループに入ります。

exit 関数は呼び出し側には戻れないので注意してください。

far バージョン なし

exp 指数

C の構文

```
#include    <math.h>

double exp(double x);
```

C++ の構文

```
#include    <cmath>

double std::exp(double x);
```

定義される場所 rts.src の exp.c

説明

exp 関数は、実数 x の指数値を戻します。戻り値は e^x です。x の値が大きすぎると範囲エラーになります。

例

```
double x, y;

x = 2.0;
y = exp(x);      /* y = 7.38, which is e**2.0 */
```

far バージョン なし

fabs　絶対値

C の構文

#include　<math.h>

double fabs(double x)**;**

C++ の構文

#include　<cmath>

double std::fabs(double x)**;**

定義される場所

rts.src の fabs.c

説明

fabs 関数は、浮動小数点数 x の絶対値を戻します。

例

```
double x, y;

x = -57.5;
y = fabs(x);          /* return value = +57.5 */
```

far バージョン

なし

far_calloc　far メモリの割り当てとクリア

C の構文

#include <stdlib.h>
far void *far_calloc(unsigned long num, unsigned long size)**;**

C++ の構文

#include <cstdlib>
long std::far_calloc(unsigned long num, unsigned long size)**;**

定義される場所

rts.src の farmemory.c

説明

far_calloc 関数は、各々が size バイト（size は符号なし長整数）の num 個のオブジェクト用に far ヒープ内に空間を割り当て、その空間へのポインタを戻します。この関数は、割り当てられたメモリをすべて 0 に初期化します。メモリを割り当てることができない場合（つまり、メモリ不足のとき）、この関数は、ヌル・ポインタ（0）を戻します。

far_calloc が使用するメモリは、特別なメモリ・プール（far ヒープ）の中のメモリです。定数 _ _FAR_SYSMEM_SIZE により、ヒープのサイズは 1K ワードに定義されています。この量はリンク時に変更できます。そのためにはリンカを起動するときに、-farheap オプションを指定し、このオプションの直後にヒープについて希望するサイズ（バイト数）を指定します。詳細は、7.1.4 項「動的なメモリ割り当て」（7-7 ページ）を参照してください。

C++ の far_calloc 関数は、far ヒープ内にオブジェクト用の空間を割り当て、割り当てられたメモリをクリアし、long 型のアドレスを戻します。C++ の far に対応した組み込み関数とこのアドレスを組み合わせて使用します。詳細は、6.7.8 項「C++ で far メモリをアクセスする組み込み関数を使用する方法」（6-22 ページ）を参照してください。

例

次のコード（C）では、far_calloc ルーチンで、far メモリに 20 バイトを割り当て、クリアしています。

```
char *ptr = (char*) far_calloc(10,2);
/* allocate and clear 20 bytes */
```

次のコードは、C++ の例です。

```
long ptr = std::far_calloc(10,2);
/* allocate and clear 20 bytes */
```

far_free　far メモリの解放

C の構文

#include <stdlib.h>
void far_free(far void *ptr**);**

C++ の構文

#include <cstdlib>
void std::far_free(long ptr**);**

定義される場所

rts.src の farmemory.c

説明

far_free 関数は、far_malloc、far_calloc、far_realloc のいずれかの呼び出しが以前割り当てていた（ptr が指す）far メモリ空間を解放します。これにより、メモリ空間を再び利用できます。割り当てていない空間を解放しようとすると、予期せぬ結果が生じます。詳細は、7.1.4 項「動的なメモリ割り当て」(7-7 ページ) を参照してください。

例

次のコード（C）では、10 バイトを割り当ててから、解放します。

```
far char *x = (far char*) far_malloc(10);
/*allocate 10 bytes*/

far_free (x);
/*free 10 bytes*/
```

次のコードは、C++ の例です。

```
long x = std::far_malloc(10);  /*allocate 10 bytes*/
std::far_free(x);              /*free 10 bytes*/
```

far_malloc　far メモリの割り当て

C の構文

#include <stdlib.h>
far void *far_malloc(unsigned long size**);**

C++ の構文

#include <cstdlib>
long std::far_malloc(unsigned long size**);**

定義される場所

rts.src の farmemory.c

説明

far_malloc 関数は far メモリ内に size バイトのオブジェクト用に空間を割り当て、その空間へのポインタを戻します。far_malloc でパケットを割り当てることができない場合（つまり、メモリ不足のとき）は、ヌル・ポインタ（0）が戻ります。この関数では、割り当てたメモリの内容変更はしません。

far_malloc が使用するメモリは、特別なメモリ・プール（ヒープ）内にあります。定数 _ _FAR_SYSMEM_SIZE により、ヒープのサイズは 1K ワードに定義されています。この量はリンク時に変更できます。そのためにはリンカを起動するときに、-farheap オプションを指定し、このオプションの直後にヒープについて希望するサイズ（バイト数）を指定します。詳細は、7.1.4 項「動的なメモリ割り当て」（7-7 ページ）を参照してください。

C++ の far_malloc 関数は、far ヒープ内にオブジェクト用の空間を割り当て、long 型のアドレスを戻します。C++ の far に対応した組み込み関数とこのアドレスを組み合わせて使用します。詳細は、6.7.8 項「C++ で far メモリをアクセスする組み込み関数を使用する方法」（6-22 ページ）を参照してください。

例

次のコード（C）では、10 バイトのデータを far メモリに割り当てます。

```
char *ptr = (char *) far_malloc(10);
/*allocate 10 bytes */
```

次のコードは、C++ の例です。

```
long ptr = std:far_malloc(10);   /*allocate 10 bytes */
```

far_memlcpy　64K ブロックを超える far メモリ・ブロックのコピー - 非オーバーラップ

C の構文

#include <string.h>
far void *far_memlcpy(far void *s1, const far void *s2, unsigned long n**);**

C++ の構文

#inlcude <cstring>
long std::far_memlcpy(long s1, const long s2, unsigned long n**);**

定義される場所

rts.src の farmemory.c

説明

far_memlcpy 関数は s2 が指すオブジェクトから s1 が指すオブジェクトに *n* 文字をコピーします。コピーされる文字数は、64K より多くなる可能性があります。オーバーラップしたオブジェクトの文字をコピーする場合、この関数の動作は予測できません。この関数は s1 の値を戻します。

far_memlmove　64K ブロックを超える far メモリ・ブロックのコピー - オーバーラップ

C の構文

#include <string.h>
far void *far_memlmove(far void *s1, const far void *s2, unsigned long n**);**

C++ の構文

#inlcude <cstring>
long std::far_memlmove(long s1, const long s2, unsigned long n**);**

定義される場所

rts.src の farmemory.c

説明

far_memlmove 関数は s2 が指すオブジェクトから s1 が指すオブジェクトに *n* 文字を転送します。この関数は s1 の値を戻します。コピーされる文字数は、64K より多くなる可能性があります。far_memlmove 関数では、オーバーラップしたオブジェクト間でも正しく文字をコピーできます。

far_realloc　ヒープ・サイズの変更

C の構文

#include <stdlib.h>
void *far_realloc(void *packet, unsigned long size**);**

C++ の構文

#include <cstdlib>
long std::far_realloc(long packet, unsigned long size**);**

定義される場所

rts.src の farmemroy.c

説明

far_realloc 関数は、packet が指す割り当て済みのメモリのサイズを size バイトに変更します。メモリ空間の内容（旧サイズと新規サイズのうちの小さい方のサイズまで）は、変更されません。

- ❏ packet が 0 の場合、far_realloc は far_malloc と同じ処理をします。
- ❏ 割り当てられていない空間を packet が指す場合、far_realloc は処理をしないで 0 を戻します。
- ❏ 空間を割り当てることができない場合には、元のメモリ空間は変更されないで、far_realloc は 0 を戻します。
- ❏ size = = 0 のときに packet がヌルでない場合、far_realloc は packet が指す空間を解放します。

より多くの空間を割り当てるためにオブジェクト全体を移動する必要がある場合、far_realloc は新しい空間を指すポインタを戻します。この操作により解放されるメモリは、割り当てを解除されます。エラーが発生した場合、この関数はヌル・ポインタ（0）を戻します。

far_realloc が使用するメモリは、特別なメモリ・プール（ヒープ）内にあります。定数 _ _FAR_SYSMEM_SIZE により、ヒープのサイズは 1K ワードに定義されています。この量はリンク時に変更できます。そのためにはリンカを起動するときに、-farheap オプションを指定し、このオプションの直後にヒープについて希望するサイズ（バイト数）を指定します。詳細は、7.1.4 項「動的なメモリ割り当て」（7-7 ページ）を参照してください。

fclose　ファイルのクローズ

C の構文

#include　<stdio.h>

int fclose(FILE *_fp)**;**

C++ の構文

#include　<cstdio>

int std::fclose(FILE *_fp)**;**

定義される場所

rts.src の fclose.c

説明

fclose 関数は、_fp が指すストリームをフラッシュし、そのストリームに関連付けられているファイルをクローズします。

far バージョン

なし

feof　EOF 標識のテスト

C の構文

#include　<stdio.h>

int feof(FILE *_fp)**;**

C++ の構文

#include　<cstdio>

int std::feof(FILE *_fp)**;**

定義される場所

rts.src の feof.c

説明

feof 関数は、_fp が指すストリームの EOF 標識をテストします。

far バージョン

なし

ferror　エラー標識のテスト

C の構文

#include　<stdio.h>

int ferror(FILE *_fp)**;**

C++ の構文

#include　<cstdio>

int std::ferror(FILE *_fp)**;**

定義される場所

rts.src の ferror.c

説明

ferror 関数は、_fp が指すストリーム のエラー標識をテストします。

far バージョン

なし

fflush　入出力バッファのフラッシュ

C の構文

#include <stdio.h>

int fflush(register FILE *_fp)**;**

C++ の構文

#include <cstdio>

int std::fflush(register FILE *_fp)**;**

定義される場所

rts.src の fflush.c

説明

fflush 関数は、_fp が指すストリームの入出力バッファをフラッシュします。

far バージョン

なし

fgetc　次の文字の読み込み

C の構文

#include <stdio.h>

int fgetc(register FILE *_fp)**;**

C++ の構文

#include <cstdio>

int std::fgetc(register FILE *_fp)**;**

定義される場所

rts.src の fgetc.c

説明

fgetc 関数は、_fp が指すストリーム内の次の文字を読み込みます。

far バージョン

なし

fgetpos　オブジェクトの格納

C の構文

#include <stdio.h>

int fgetpos(FILE *_fp**,** fpos_t *pos)**;**

C++ の構文

#include <cstdio>

int std::fgetpos(FILE *_fp**,** fpos_t *pos)**;**

定義される場所

rts.src の fgetpos.c

説明

fgetpos 関数は、_fp が指すストリームのファイル位置標識の現行値を、pos が指すオブジェクトに格納します。

far バージョン

なし

fgets　次の複数文字の読み込み

C の構文

#include <stdio.h>

char *fgets(char *_ptr, register int _size, register FILE *_fp**);**

C++ の構文

#include <cstdio>

char *std::fgets(char *_ptr, register int _size, register FILE *_fp**);**

定義される場所

rts.src の fgets.c

説明

fgets 関数は、指定した数の文字を、_fp が指すストリームから読み込みます。文字は、_ptr で指定された配列に入れます。読み込まれる文字の数は _size -1 です。

far バージョン

なし

floor　切り捨て

C の構文

#include <math.h>

double floor(double x**);**

C++ の構文

#include <cmath>

double std::floor(double x**);**

定義される場所

rts.src の floor.c

説明

floor 関数は、x 以下の最大の整数を表す浮動小数点数を戻します。

例

```
double answer;

answer = floor(3.1415);    /* answer =  3.0 */
answer = floor(-3.5);      /* answer = -4.0 */
```

far バージョン

なし

fmod　浮動小数点の剰余

C の構文　#include　<math.h>

double fmod(double x, double y)**;**

C++ の構文　#include　<cmath>

double std::fmod(double x, double y)**;**

定義される場所　rts.src の fmod.c

説明　fmod 関数は、x を y で割った浮動小数点の剰余を戻します。y==0 の場合、関数は 0 を戻します。

例

```
double x, y, r;

x = 11.0;
y = 5.0;
r = fmod(x, y);       /* fmod returns 1.0 */
```

far バージョン　なし

fopen　ファイルのオープン

C の構文　#include　<stdio.h>

FILE ***fopen**(const char *_fname, const char *_mode)**;**

C++ の構文　#include　<cstdio>

FILE ***std::fopen**(const char *_fname, const char *_mode)**;**

定義される場所　rts.src の fopen.c

説明　fopen 関数は、_fname が指すファイルをオープンします。_mode が指す文字列にはファイルのオープン方法が記述されています。

far バージョン　なし

fprintf　ストリームの書き込み

C の構文　#include　<stdio.h>

int fprintf(FILE *_fp, const char *_format, ...)**;**

C++ の構文　#include　<cstdio>

int std::fprintf(FILE *_fp, const char *_format, ...)**;**

定義される場所　rts.src の fprintf.c

説明　fprintf 関数は、_fp が指すストリームへの書き込みを行います。_format が指す文字列には、ストリームを書き込む方法が記述されています。

far バージョン　なし

fputc　文字の書き込み

C の構文

#include <stdio.h>

int fputc(int _c, register FILE *_fp)**;**

C++ の構文

#include <cstdio>

int std::fputc(int _c, register FILE *_fp)**;**

定義される場所

rts.src の fputc.c

説明

fputc 関数は、_fp が指すストリームに 1 文字を書き込みます。

far バージョン

なし

fputs　文字列の書き込み

C の構文

#include <stdio.h>

int fputs(const char *_ptr, register FILE *_fp)**;**

C++ の構文

#include <cstdio>

int std::fputs(const char *_ptr, register FILE *_fp)**;**

定義される場所

rts.src の fputs.c

説明

fputs 関数 は、_fp が指すストリームに _ptr が指す文字列を書き込みます。

far バージョン

なし

fread　ストリームの読み込み

C の構文

#include　<stdio.h>

size fread(void *_ptr, size_t size, size_t count, FILE *_fp)**;**

C++ の構文

#include　<cstdio>

size std::fread(void *_ptr, size_t size, size_t count, FILE *_fp)**;**

定義される場所

rts.src の fread.c

説明

fread 関数は、_fp が指すストリームからの読み込みを行います。入力は、_ptr が指す配列に保存されます。読み込まれるオブジェクトの数は _count です。オブジェクトのサイズは _size です。

far バージョン

なし

注： C28x の char がホストのバイトと異なる場合の fread の使用方法

C28x の char がホストのバイトと異なる場合の入出力の処理方法については、『Reading and Writing Binary Files on Targets with More Than 8-bit Chars』（文献番号 SPRA757）を参照してください。

free　メモリの解放

C の構文

#include　<stdlib.h>

void free(void *ptr)**;**

C++ の構文

#include　<cstdlib>

void std::free(void *ptr)**;**

定義される場所

rts.src の memory.c

説明

free 関数は、malloc、calloc、realloc のいずれかの呼び出しにより割り当てられた（ptr が指す）メモリ空間を解放します。これにより、メモリ空間を再び利用できます。割り当てていない空間を解放しようとすると、予期せぬ結果が生じます。free() を使うときは事前に malloc() を実行する必要があります。そうしないと、free() はゼロを返します。詳細は、7.1.4 項「動的なメモリ割り当て」（7-7 ページ）を参照してください。

例

この例では 10 バイトを割り当ててから、解放します。

```
char *x;
x = malloc(10);       /*  allocate 10 bytes   */
free(x);              /*  free 10 bytes       */
```

far バージョン（C） **void far_free**(void *ptr)**;**

far バージョン（C++） **void std::far_free**(long)

far_free（8-58 ページ）を参照してください。

free_memory 割り当てで使用できる動的なメモリの合計の検出

C の構文

#include <stdlib.h>

int free_memory(void)**;**

C++ の構文

#include <cstdlib>

int std:free_memory (void)**;**

定義される場所 rts.src の memory.c および farmemory.c

説明 この関数は、割り当てで使用できる動的なメモリの合計を検出します。メモリが使用できない場合には、この関数は 0 を戻します。

far バージョン（C） **long far_free_memory**(void)**;**

far バージョン（C++） **long std::far_free_memory**(void)**;**

freopen ファイルのオープン

C の構文

#include <stdio.h>

FILE ***freopen(**const char *_fname, const char *_mode, register FILE *_fp)**;**

C++ の構文

#include <cstdio>

FILE ***std::freopen(**const char *_fname, const char *_mode, register FILE *_fp)**;**

定義される場所 rts.src の freopen.c

説明 freopen 関数は、_fp が指すファイルをクローズし、_fname が指すファイルをオープンし、_fp とストリームを関連付けます。_mode が指す文字列にはファイルのオープン方法が記述されています。

far バージョン なし

frexp　小数部と指数部

C の構文

#include　<math.h>

double frexp(double value**,** int *exp);

C++ の構文

#include　<cmath>

double std::frexp(double value**,** int *exp);

定義される場所

rts.src の frexp.c および farfrexp.c

説明

frexp 関数は、浮動小数点数を正規化した小数部と 2 の整数乗に分けます。関数は範囲 [1/2, 1] または 0 の値を戻すため、value = = $x \times 2^{exp}$ となります。frexp 関数は、累乗を exp が指す整数に保存します。value が 0 の場合、小数部、指数部ともに 0 が戻ります。

例

```
double fraction;
int exp;

fraction = frexp(3.0, &exp);
/* after execution, fraction is .75 and exp is 2 */
```

far バージョン

double far_frexp(double value, far int *exp)**;**

fscanf　ストリームの読み込み

C の構文

#include　<stdio.h>

int fscanf(FILE *_fp, const char *_fmt, ...)**;**

C++ の構文

#include　<cstdio>

int std::fscanf(FILE *_fp, const char *_fmt, ...)**;**

定義される場所

rts.src の fscanf.c

説明

fscanf 関数は、_fp が指すストリームからの読み込みを行います。_fmt が指す文字列には、ストリームの読み込み方法が記述されています。

far バージョン

なし

fseek　ファイル位置標識の設定

C の構文

```
#include    <stdio.h>
```

int fseek(register FILE *_fp, long _offset, int _ptrname)**;**

C++ の構文

```
#include    <cstdio>
```

int std::fseek(register FILE *_fp, long _offset, int _ptrname)**;**

定義される場所　rts.src の fseek.c

説明　fseek 関数は、_fp が指すストリームのファイル位置標識を設定します。起点の位置は _ptrname に指定されます。バイナリ・ファイルの場合、_offset を使用して _ptrname からの位置を設定します。テキスト・ファイルの場合、_offset は必ずゼロに設定してください。

far バージョン　なし

fsetpos　ファイル位置標識の設定

C の構文

```
#include    <stdio.h>
```

int fsetpos(FILE *_fp, const fpos_t *_pos)**;**

C++ の構文

```
#include    <cstdio>
```

int std::fsetpos(FILE *_fp, const fpos_t *_pos)**;**

定義される場所　rts.src の fsetpos.c

説明　fsetpos 関数は、_fp が指すストリームのファイル位置標識を _pos に設定します。ポインタ _pos の値は、同じストリームに対する fgetpos() から得た値を指定する必要があります。

far バージョン　なし

ftell　現行のファイル位置標識の取得

C の構文

```
#include    <stdio.h>
```

long ftell(FILE *_fp)**;**

C++ の構文

```
#include    <cstdio>
```

long std::ftell(FILE *_fp)**;**

定義される場所　rts.src の ftell.c

説明　ftell 関数は、_fp が指すストリームのファイル位置標識の現行値を取得します。

far バージョン　なし

fwrite データ・ブロックの書き込み

C の構文

#include <stdio.h>

size_t fwrite(const void *_ptr, size_t _size, size_t _count, register FILE *_fp)**;**

C++ の構文

#include <cstdio>

size_t std::fwrite(const void *_ptr, size_t _size, size_t _count, register FILE *_fp)**;**

定義される場所

rts.src の fwrite.c

説明

fwrite 関数は _ptr が指すメモリから _fp が指すストリームにデータ・ブロックを書き込みます。

far バージョン

なし

getc 次の文字の読み込み

C の構文

#include <stdio.h>

int getc(FILE *_fp)**;**

C++ の構文

#include <cstdio>

int std::getc(FILE *_fp)**;**

定義される場所

rts.src の fgetc.c

説明

getc 関数は、_fp が指すファイル内の次の文字を読み込みます。

far バージョン

なし

getchar 標準入力からの次の文字の読み込み

C の構文

#include <stdio.h>

int getchar(void)**;**

C++ の構文

#include <cstdio>

int std::getchar(void)**;**

定義される場所

rts.src の fgetc.c

説明

getchar 関数は、標準入力デバイスから次の文字を読み込みます。

far バージョン

なし

getenv　環境情報の取得

C の構文

#include <stdlib.h>

char *getenv(const char *_string)**;**

C++ の構文

#include <cstdlib>

char *std::getenv(const char *_string)**;**

定義される場所

rts.src の fgetenv.c

説明

getenv 関数は、_string で示される環境変数の情報を戻します。

> **注： getenv 関数はターゲットシステム固有**
>
> getenv 関数はターゲットシステムによって異なるため、ユーザ固有の getenv 関数を記述する必要があります。

far バージョン

なし

gets　標準入力からの次行の読み込み

C の構文

#include <stdio.h>

char *gets(char *_ptr)**;**

C++ の構文

#include <cstdio>

char *std::gets(char *_ptr)**;**

定義される場所

rts.src の fgets.c

説明

gets 関数は、標準入力デバイスから入力行を読み込みます。文字は、_ptr で指定された配列に入れます。

far バージョン

なし

gmtime　グリニッジ標準時

C の構文

```
#include    <time.h>

struct tm *gmtime(const time_t *timer);
```

C++ の構文

```
#include    <ctime>

struct tm *std::gmtime(const time_t *timer);
```

定義される場所

rts.src の gmtime.c

説明

gmtime 関数は、カレンダ時（timer が指す）をグリニッジ標準時で表される詳細時刻に変換します。

time.h/ctime ヘッダで宣言と定義が行われる関数と型の詳細は、8.4.16 項「時間関数（time.h/ctime）」（8-29 ページ）を参照してください。

far バージョン

なし

isxxx　文字の判別

C の構文

```
#include    <ctype.h>

int isalnum(int c);        int islower(int c);
int isalpha(int c);        int isprint(int c);
int isascii(int c);        int ispunct(int c);
int iscntrl(int c);        int isspace(int c);
int isdigit(int c);        int isupper(int c);
int isgraph(int c);        int isxdigit(int c);
```

C++ の構文

```
#include    <cctype>

int std::isalnum(int c);   int std::islower(int c);
int std::isalpha(int c);   int std::isprint(int c);
int std::isascii(int c);   int std::ispunct(int c);
int std::iscntrl(int c);   int std::isspace(int c);
int std::isdigit(int c);   int std::isupper(int c);
int std::isgraph(int c);   int std::isxdigit(int c);
```

定義される場所

rts.src の isxxx.c および ctype.c
またはマクロとして ctype.h/cctype にて定義

説明

これらの関数は 1 つの引数 c をテストし、それが 英字、英数字、数字、ASCII など特定の種類の文字であるかどうかを調べます。テストの結果が真の場合、この関数は 0 以外の値を戻します。テストの結果が偽の場合、この関数は 0 を戻します。文字判別関数には次のような関数があります。

isalnum	英数字 ASCII 文字を認識します（isalpha や isdigit が真となるすべての文字をテストします）。
isalpha	英字 ASCII 文字を認識します（islower や isupper が真となるすべての文字をテストします）。
isascii	ASCII 文字（0 ～ 127 の範囲の文字）を認識します。
iscntrl	制御文字（0 ～ 31 の範囲と 127 の ASCII 文字）を認識します。
isdigit	数字（0 ～ 9）を認識します。
isgraph	空白以外の印字文字を認識します。
islower	英小文字 ASCII 文字を認識します。
isprint	空白を含む表示可能な ASCII 文字（32 ～ 126 の範囲の ASCII 文字）を認識します。
ispunct	ASCII 句読文字を認識します。
isspace	ASCII のタブ（水平か垂直）、スペースバー、復帰、書式送り、および改行文字を認識します。
isupper	英大文字 ASCII 文字を認識します。
isxdigit	16 進数字（0 ～ 9、a ～ f、A ～ F）を認識します。

C/C++ コンパイラは、これらの関数と同じ機能をもつ一連のマクロもサポートしています。これらのマクロの名前はこれらの関数と同じですが、先頭に下線が付いている点が異なります。たとえば、_isascii は isascii 関数と同じ機能をもつマクロです。一般に、マクロは関数より処理速度が高速です。

far バージョン

なし

labs/llabs

abs/labs/llab（8-44 ページ）を参照してください。

ldexp　2 の累乗との乗算

C の構文

#include　<math.h>

double ldexp(double x**,** int exp**);**

C++ の構文

#include　<cmath>

double std::ldexp(double x**,** int exp**);**

定義される場所

rts.src の ldexp.c

説明

ldexp 関数は、浮動小数点数と 2 の累乗を乗算し、$x \times 2^{exp}$ を戻します。exp は、負の値または正の値のどちらでも構いません。結果の値が大きすぎると、範囲エラーになります。

例

```
double result;

result = ldexp(1.5, 5);      /* result is 48.0 */
result = ldexp(6.0, -3);     /* result is 0.75 */
```

far バージョン

なし

ldiv/lldiv　div/ldiv/lldiv（8-55 ページ）を参照してください。

localtime　地方時

C の構文

#include　<time.h>

struct tm *localtime(const time_t *timer**);**

C++ の構文

#include　<ctime>

struct tm *std::localtime(const time_t *timer**);**

定義される場所

rts.src の localtime.c

説明

localtime 関数は、カレンダ時（timer が指す）を地方時で表される詳細時刻に変換します。この関数は、変換された時刻へのポインタを戻します。

time.h/ctime ヘッダで宣言と定義が行われる関数と型の詳細は、8.4.16 項「時間関数（time.h/ctime）」（8-29 ページ）を参照してください。

far バージョン

なし

log　自然対数

C の構文

#include　<math.h>

double log(double x)**;**

C++ の構文

#include　<cmath>

double std::log(double x)**;**

定義される場所

rts.src の log.c

説明

log 関数は、実数 x の自然対数を戻します。x が負の場合は、領域エラーになります。x が 0 の場合は、範囲エラーになります。

説明

```
float x, y;

x = 2.718282;
y = log(x);          /* Return value = 1.0 */
```

far バージョン

なし

log10　常用対数

C の構文

#include　<math.h>

double log10(double x)**;**

C++ の構文

#include　<cmath>

double std::log10(double x)**;**

定義される場所

rts.src の log10.c

説明

log10 関数は、底が 10 の実数 x の対数を戻します。x が負の場合は、領域エラーになります。x が 0 の場合は、範囲エラーになります。

例

```
float x, y;

x = 10.0;
y = log(x);          /* Return value = 1.0 */
```

far バージョン

なし

longjmp　setjmp/longjmp （8-88 ページ）を参照してください。

ltoa　長整数から ASCII へ

C の構文

プロトタイプはありません。

int ltoa(long val, char *buffer)**;**

C++ の構文

プロトタイプはありません。

int std::ltoa(long val, char *buffer)**;**

定義される場所

rts.src の ltoa.c および farltoa.c

説明

ltoa 関数は標準外（非 ANSI/ISO）関数で、非 ANSI/ISO コードと互換性を確保するために提供されています。これと同等の標準関数は sprintf です。ltoa 関数は、rts.src にはプロトタイプが宣言されていません。長整数 n を等価な ASCII 文字列に変換してバッファに書き込みます。入力した数値 val が負の場合には、先頭に負符号が出力されます。ltoa 関数は、バッファに書き込まれた文字数を戻します。

far バージョン

int far_ltoa(long val, far char *buffer)**;**

malloc　メモリの割り当て

C の構文

#include <stdlib.h>

void *malloc(size_t size)**;**

C++ の構文

#include <cstdlib>

void *std::malloc(size_t size)**;**

定義される場所

rts.src の memory.c

説明

malloc 関数は size バイトのオブジェクト用に空間を割り当て、その空間のポインタを戻します。malloc でパケットを割り当てることができない場合（つまり、メモリ不足のとき）は、ヌル・ポインタ（0）が戻ります。この関数では、割り当てたメモリの内容変更はしません。

malloc が使用するメモリは、特別なメモリ・プール（ヒープ）内にあります。定数 __SYSMEM_SIZE により、ヒープのサイズは 1K ワードに定義されています。この量はリンク時に変更できます。そのためにはリンカを起動するときに、-heap オプションを指定し、このオプションの直後にヒープについて希望するサイズ（バイト数）を指定します。詳細は、7.1.4 項「動的なメモリ割り当て」（7-7 ページ）を参照してください。

far バージョン（C）

far void *far_malloc(unsigned long size)**;**

far バージョン（C++）

long std::far_malloc(unsigned long size)**;**

far_malloc（8-59 ページ）を参照してください。

max_free 使用できる動的なメモリの最大割り当てサイズの検出

C の構文 #include <stdlib.h>**int max_free**(void)**;**

C++ の構文 #include <cstdlib>**int std::max_free**(void)**;**

定義される場所 rts.src の memory.c、farmemory.c、および memory.cpp

説明 malloc、calloc、realloc のいずれかを使ってメモリを割り当てるときに使用できる連続したメモリ・ブロックの最大サイズを戻します。メモリが使用できない場合には、この関数は 0 を戻します。

far バージョン（C） **long far_max_free**(void)**;**

far バージョン（C++） **long std::far_max_free**(void)**;**

memchr バイトの最初の出現を検出

C の構文 #include <string.h>

void *memchr(const void *cs, int c, size_t n)**;**

C++ の構文 #include <cstring>

void *memchr(const void *cs, int c, size_t n)**;**

定義される場所 rts.src の memchr.c および farmemchr.c

説明 memchr 関数は、cs が指すオブジェクトの先頭の n 文字の中で最初に現れる c を検出します。文字を検出した場合、memchr はその文字へのポインタを戻します。検出しなかった場合は、ヌル・ポインタ（0）を戻します。

memchr 関数は strchr に似ていますが、memchr が検索するオブジェクトに 0 を含めることができ、また c に 0 を指定できる点が異なります。

far バージョン（C） **void *far_memchr(**const far void *cs, int c, size_t n)**;**

memcmp　メモリ比較

C の構文

```
#include    <string.h>
```

int memcmp(const void *cs, const void *ct, size_t n)**;**

C++ の構文

```
#include    <cstring>
```

int std::memcmp(const void *cs, const void *ct, size_t n)**;**

定義される場所

rts.src の memcmp.c および farmemcmp.c

説明

memcmp 関数は、cs で指定したオブジェクトと、ct で指定したオブジェクトの先頭の n 文字を比較します。この関数は、以下の値のどれかを戻します。

< 0	*cs が *ct より小さいとき。
0	*cs が *ct と等しいとき。
> 0	*cs が *ct より大きいとき。

memcmp 関数は strncmp に似ていますが、memcmp が比較するオブジェクトに 0 を含めることができる点が異なります。

far バージョン（C）

int far_memcmp(const far void *cs, const far void *ct, size_t n)**;**

memcpy　メモリ・ブロック・コピー - 非オーバーラップ

C の構文

```
#include    <string.h>
```

void *memcpy(void *s1, const void *s2, size_t n)**;**

C++ の構文

```
#include    <cstring>
```

void *std::memcpy(void *s1, const void *s2, size_t n)**;**

定義される場所

rts.src の memcpy.c および farmemcpy.c

説明

memcpy 関数は s2 で指定したオブジェクトから s1 で指定したオブジェクトに n 文字をコピーします。オーバーラップしたオブジェクトの文字をコピーする場合の、この関数の動作は予測できません。この関数は s1 の値を戻します。

memcpy 関数は strncpy に似ていますが、memcpy がコピーするオブジェクトに 0 を含めることができる点が異なります。

far バージョン（C）

far void *far_memcpy(far void *s1; const far foid *s2, size_t n)**;**

memmove　メモリ・ブロック・コピー - オーバーラップ

C の構文

#include　<string.h>

void *memmove(void *s1, const void *s2, size_t n)**;**

C++ の構文

#include　<cstring>

void *std::memmove(void *s1, const void *s2, size_t n)**;**

定義される場所

rts.src の memmove.c および farmemmove.c

説明

memmove 関数は、s2 で指定したオブジェクトから s1 で指定したオブジェクトに n 文字を移動します。この関数は s1 の値を戻します。memmove 関数では、オーバーラップしたオブジェクト間でも正しく文字をコピーできます。

far バージョン（C）

far void *far_memmove(far void *s1, const far void *s2, size_t n)**;**

memset　メモリ内での値のコピー

C の構文

#include　<string.h>

void *memset(void *mem, register int ch, size_t length)**;**

C++ の構文

#include　<cstring>

void *std::memset(void *mem, register int ch, size_t length)**;**

定義される場所

rts.src の memset.c および farmemset.c

説明

memset 関数は、mem が指すオブジェクトの先頭の length 文字に ch の値をコピーします。この関数は mem の値を戻します。

far バージョン（C）

far void *far_memset(far void *mem, register int ch, size_t length)**;**

mktime カレンダ時への変換

C の構文

#include <time.h>

time_t *mktime(struct tm *timeptr)**;**

C++ の構文

#include <ctime>

time_t *std::mktime(struct tm *timeptr)**;**

定義される場所

rts.src の mktime.c

説明

mktime 関数は、地方時で表された詳細時刻を対応するカレンダ時に変換します。timeptr 引数は、詳細時刻を保持する構造体を指します。

この関数では、tm_wday と tm_yday の元の値は無視されます。また、構造体内に設定する値の範囲を制限しません。時間の変換が正常に完了すると tm_wday と tm_yday は適切に設定され、構造体の他の構成要素には制限範囲内の値が設定されます。tm_mday の最終値は、tm_mon と tm_year が決定されるまで設定されません。

戻り値は型 time_t の値としてエンコードされます。カレンダ時を表現できない場合、この関数は値 -1 を戻します。

time.h/ctime ヘッダで宣言と定義が行われる関数と型の詳細は、8.4.16 項「時間関数（time.h/ctime）」（8-29 ページ）を参照してください。

例

この例では、2001 年の 7 月 4 日が何曜日になるかを求めます。

```
#include <time.h>
static const char *const wday[] = {
             "Sunday", "Monday", "Tuesday", "Wednesday",
             "Thursday", "Friday", "Saturday" };

struct tm time_str;

time_str.tm_year  = 2001 - 1900;
time_str.tm_mon   = 7;
time_str.tm_mday  = 4;
time_str.tm_hour  = 0;
time_str.tm_min   = 0;
time_str.tm_sec   = 1;
time_str.tm_isdst = 1;

mktime(&time_str);

/* After calling this function, time_str.tm_wday    */
/*    contains the day of the week for July 4, 2001 */
```

far バージョン

なし

modf　符号付き整数および符号付き小数

C の構文

#include　<math.h>

double modf(double value**,** double *iptr**);**

C++ の構文

#include　<cmath>

double std::modf(double value**,** double *iptr**);**

定義される場所

rts.src の modf.c および farmodf.c

説明

modf 関数は、値を符号付き整数と符号付き小数に分けます。この 2 つの部分の符号は、入力した引数の符号と同じです。この関数は値の小数部を戻し、整数部を iptr で指定したオブジェクトに倍精度浮動小数点値として保存します。

例

```
double value, ipart, fpart;

value = -3.1415;

fpart = modf(value, &ipart);

/* After execution, ipart contains -3.0,*/
/* and fpart contains -0.1415.          */
```

far バージョン

double far_modf(double value, far double *iptr**);**

perror　エラー番号のマップ

C の構文

#include　<stdio.h>

void perror(const char *_s**);**

C++ の構文

#include　<cstdio>

void std::perror(const char *_s**);**

定義される場所

rts.src の perror.c

説明

perror 関数は、s で示されるエラー番号を文字列にマップし、エラー・メッセージを出力します。

far バージョン

なし

pow 累乗

C の構文

#include <math.h>

double pow(double x, double y)**;**

C++ の構文

#include <cmath>

double std::pow(double x, double y)**;**

定義される場所

rts.src の pow.c

説明

pow 関数は、x の y 乗を戻します。x ＝0 かつ y ≤ 0 の場合、または x が負で y が整数でない場合は、領域エラーになります。結果の値が大きすぎて表現できない場合は、範囲エラーになります。

例

```
double x, y, z;

x = 2.0;
y = 3.0;
x = pow(x, y);        /* return value = 8.0 */
```

far バージョン

なし

printf 標準出力への書き込み

C の構文

#include <stdio.h>

int printf(const char *_format, ...)**;**

C++ の構文

#include <cstdio>

int std::printf(const char *_format, ...)**;**

定義される場所

rts.src の printf.c

説明

printf 関数は、標準出力デバイスへの書き込みを行います。_format が指す文字列には、ストリームを書き込む方法が記述されています。

far バージョン

なし

putc 文字の書き込み

C の構文

#include <stdio.h>

int putc(int _x, FILE *_fp)**;**

C++ の構文

#include <cstdlib>

int std::putc(int _x, FILE *_fp)**;**

定義される場所

rts.src の putc.c

説明

putc 関数は、_fp が指すストリームに 1 文字を書き込みます。

far バージョン

なし

putchar　標準出力への文字の書き込み

C の構文

#include　<stdlib.h>

int putchar(int _x)**;**

C++ の構文

#include　<cstdlib>

int std::putchar(int _x)**;**

定義される場所

rts.src の putchar.c

説明

putchar 関数は、標準出力デバイスに 1 文字を書き込みます。

far バージョン

なし

puts　標準出力への書き込み

C の構文

#include　<stdlib.h>

int puts(const char *_ptr)**;**

C++ の構文

#include　<cstdlib>

int std::puts(const char *_ptr)**;**

定義される場所

rts.src の puts.c

説明

puts 関数は、_ptr が指す文字列を標準出力デバイスに書き込みます。

far バージョン

なし

qsort　配列のソート

C の構文

#include　<stdlib.h>

void qsort(void *base, size_t nmemb, size_t size, int (*compar) ())**;**

C++ の構文

#include　<cstdlib>

void std::qsort(void *base, size_t nmemb, size_t size, int (*compar) ())**;**

定義される場所

rts.src の qsort.c および farqsort.c

説明

qsort 関数は、nmemb 個のメンバから構成される配列をソートします。引数 base はソートされていない配列の最初のメンバを指します。引数 size は各メンバのサイズを示します。

この関数は、配列を昇順にソートします。

引数 compar は、キーを配列要素と比較する関数を指します。比較関数は、次のように宣言します。

```
int  cmp(const void *ptr1, const void *ptr2);
```

cmp 関数は、ptr1 と ptr2 が指すオブジェクトを比較し、次の値のいずれかを戻します。

< 0	*ptr1 が *ptr2 より小さいとき。
0	*ptr1 が *ptr2 と等しいとき。
> 0	*ptr1 が *ptr2 より大きいとき。

far バージョン（C）

void far_qsort(far void *base, size_t nmemb, size_t size, int (*compar) ()**);**

rand/srand　乱整数

C の構文

```
#include <stdlib.h>

int rand(void);
void srand(unsigned int seed);
```

C++ の構文

```
#include <cstdlib>

int std::rand(void);
void std::srand(unsigned int seed);
```

定義される場所

rts.src の rand.c

説明

2 つの関数がともに機能して、疑似乱数シーケンスを生成します。

- rand 関数は、0 ～ RAND_MAX の範囲で疑似乱整数を戻します。
- 引き続き rand 関数を呼び出した場合に新しい疑似乱数シーケンスが生成できるように、srand 関数はシード（種）の値を設定します。srand 関数は値を戻しません。

srand を呼び出す前に rand を呼び出すと、シードの値が 1 で srand を呼び出したときに生成される場合と同じシーケンスが生成されます。同じシード値で srand を呼び出すと、rand は同じシーケンスの乱数を生成します。

far バージョン

なし

realloc　ヒープ・サイズの変更

C の構文

```
#include    <stdlib.h>
```

void *realloc(void *packet, size_t size)**;**

C++ の構文

```
#include    <cstdlib>
```

void *std::realloc(void *packet, size_t size)**;**

定義される場所

rts.src の memory.c

説明

realloc 関数は、packet が指す割り当て済みのメモリのサイズを、size で指定されるサイズ（バイト数）に変更します。メモリ空間の内容（旧サイズと新規サイズのうちの小さい方のサイズまで）は、変更されません。

- ❑ packet が 0 の場合、realloc は malloc と同じ処理をします。
- ❑ 割り当てられていない空間を packet が指す場合、realloc は処理をしないで 0 を戻します。
- ❑ 空間を割り当てることができない場合には、元のメモリ空間は変更されないで、realloc は 0 を戻します。
- ❑ size == 0 のときに packet がヌルでない場合、realloc は packet が指す空間を解放します。

より多くの空間を割り当てるためにオブジェクト全体を移動する必要がある場合、realloc は新しい空間を指すポインタを戻します。この操作により解放されるメモリは、割り当てを解除されます。エラーが発生した場合、この関数はヌル・ポインタ（0）を戻します。

realloc が使用するメモリは、特別なメモリ・プール（ヒープ）内にあります。定数 __SYSMEM_SIZE により、ヒープのサイズは 1K ワードに定義されています。この量はリンク時に変更できます。そのためにはリンカを起動するときに、-heap オプションを指定し、このオプションの直後にヒープについて希望するサイズ（バイト数）を指定します。詳細は、7.1.4 項「動的なメモリ割り当て」（7-7 ページ）を参照してください。

far バージョン（C）

far void *far_realloc(far void *packet, size_t size**);**

far バージョン（C++）

long std::far_realloc(long packet, unsigned long size**);**

far_realloc（8-60 ページ）を参照してください。

remove　ファイルの除去

C の構文

#include <stdlib.h>

int remove(const char *_file)**;**

C++ の構文

#include <cstdlib>

int std::remove(const char *_file)**;**

定義される場所

rts.src の remove.c

説明

remove 関数は、_file が指すファイルをその名前では使用できないようにします。

far バージョン

なし

rename　ファイルの名前変更

C の構文

#include <stdlib.h>

int rename(const char *old_name, const char *new_name)**;**

C++ の構文

#include <cstdlib>

int std::rename(const char *old_name, const char *new_name)**;**

定義される場所

rts.src の rename.c

説明

rename 関数は、old_name が指すファイルの名前を変更します。新しい名前は、new_name が指しています。

far バージョン

なし

rewind　ファイルの先頭へのファイル位置標識の設定

C の構文

#include <stdlib.h>

void rewind(register FILE *_fp)**;**

C++ の構文

#include <cstdlib>

void std::rewind(register FILE *_fp)**;**

定義される場所

rts.src の rewind.c

説明

rewind 関数は、_fp が指すストリームのファイル位置標識を、ファイルの先頭に設定します。

far バージョン

なし

scanf 標準入力からのストリームの読み込み

C の構文

#include <stdlib.h>

int scanf(const char *_fmt, ...)**;**

C++ の構文

#include <cstdlib>

int std::scanf(const char *_fmt, ...)**;**

定義される場所

rts.src の fscanf.c

説明

scanf 関数は、標準入力デバイスからストリームを読み込みます。_fmt が指す文字列には、ストリームの読み込み方法が記述されています。

far バージョン

なし

setbuf ストリーム用のバッファの指定

C の構文

#include <stdlib.h>

void setbuf(register FILE *_fp, char *_buf)**;**

C++ の構文

#include <cstdlib>

void std::setbuf(register FILE *_fp, char *_buf)**;**

定義される場所

rts.src の setbuf.c

説明

setbuf 関数は、_fp が指すストリームに使用されるバッファを指定します。_buf をヌルに設定すると、バッファリングはオフになります。戻り値はありません。

far バージョン

なし

setjmp/longjmp 非ローカル・ジャンプ

C の構文

#include <setjmp.h>

int setjmp(jmp_buf env)**;**
void longjmp(jmp_buf env, int _val)**;**

C++ の構文

#include <csetjmp>

int std::setjmp(jmp_buf env)**;**
void std::longjmp(jmp_buf env, int _val)**;**

定義される場所

rts.src の setjmp.asm および farsetjmp.asm

説明

setjmp.h/csetjmp ヘッダは、通常の関数の呼び出しと復帰に関する規律をバイパスするための型、マクロ、および関数を定義します。

❑ **jmp_buf** 型 - 呼び出し環境の復元に必要な情報の保存に適している配列型です。

❑ **setjmp** マクロ - 後で longjmp 関数で使用できるように、呼び出し環境を jmp_buf 引数に保存します。

直接の呼び出しから戻る場合、setjmp マクロは 0 を戻します。呼び出しから longjmp 関数へ戻る場合、setjmp マクロは 0 以外の値を戻します。

❑ **longjmp** 関数 - setjmp マクロの一番最後の呼び出しにより jmp_buf 引数に保存された環境を復元します。setjmp マクロが呼び出されなかった場合や setjmp マクロが異常終了した場合には、longjmp の動作は予測できません。

longjmp が完了した後、対応する setjmp の呼び出しにより _val で指定した値が戻った場合と同様に、プログラムは引き続き実行されます。たとえ _val が 0 でも、longjmp 関数は、setjmp に値 0 を戻すことはありません。_val が 0 の場合、setjmp マクロは値 1 を戻します。

setjmp.h/csetjmp は、far_jmp_buf 型（jmp_buf の far に対応した型）も定義しています。

例

これらの関数は一般的には、ネストの深い関数呼び出しから直ちに戻れるようにするために使用されます。

```
#include <setjmp.h>
jmp_buf env;

main()
{
       int errcode;

       if ((errcode = setjmp(env)) == 0)
           nest1();
       else
           switch (errcode)
       . . .
}
       . . .
nest42()
{
       if (input() == ERRCODE42)
       /* return to setjmp call in main */
           longjmp (env, ERRCODE42);
       . . .
}
```

far バージョン（C）

int far_setjmp(far_jmp_buf env)**;**
void far_longjmp(far_jmp_buf env, int val)**;**

setvbuf　バッファの定義およびストリームへの関連付け

C の構文

#include <stdlib.h>

int **setvbuf**(register FILE *_fp, register char *_buf, register int _type, register size_t _size)**;**

C++ の構文

#include <cstdlib>

int **std::setvbuf**(register FILE *_fp, register char *_buf, register int _type, register size_t _size)**;**

定義される場所

rts.src の setvbuf.c

説明

setvbuf 関数は、_fp が指すストリームに使用されるバッファを定義し、関連付けます。_buf をヌルに設定すると、バッファが割り当てられます。_buf でバッファを指定すると、そのバッファがストリームに使用されます。_size はバッファのサイズを指定します。_type は、バッファリングのタイプを次のように指定します。

_IOFBF	完全なバッファリングが行われます。
_IOLBF	行バッファリングが行われます。
_IONBF	バッファリングは行われません。

far バージョン

なし

sin　サイン

C の構文

#include　<math.h>

double sin(double x)**;**

C++ の構文

#include　<cmath>

double std::sin(double x)**;**

定義される場所

rts.src の sin.c

説明

sin 関数は、浮動小数点数 x のサインを戻します。角度 x は、ラジアンで表します。引数の値が大きすぎると、有意性のある結果がほとんど得られないか、全く得られなくなります。

例

```
double radian, sval;     /* sval is returned by sin */

radian = 3.1415927;
sval = sin(radian);      /* -1 is returned by sin  */
```

far バージョン

なし

sinh　ハイパボリック・サイン

C の構文

#include　<math.h>

double sinh(double x)**;**

C++ の構文

#include　<cmath>

double std::sinh(double x)**;**

定義される場所

rts.src の sinh.c

説明

sinh 関数は、浮動小数点数 x のハイパボリック・サインを戻します。引数の値が大きすぎると、範囲エラーになります。

例

```
double x, y;

x = 0.0;
y = sinh(x);       /* return value = 0.0 */
```

far バージョン

なし

sprintf　ストリームの書き込み

C の構文

#include　<stdio.h>

int sprintf(char _string, const char *_format, ...)**;**

C++ の構文

#include　<cstdio>

int std::sprintf(char _string, const char *_format, ...)**;**

定義される場所

rts.src の sprintf.c

説明

sprintf 関数は、_string が指す配列への書き込みを行います。_format が指す文字列には、ストリームを書き込む方法が記述されています。

far バージョン

なし

sqrt　平方根

C の構文

#include　<math.h>

double sqrt(double x)**;**

C++ の構文

#include　<cmath>

double std::sqrt(double x)**;**

定義される場所

rts.src の sqrt.c

説明

sqrt 関数は、実数 x の非負数の平方根を戻します。この引数が負の場合は領域エラーになります。

例

```
double x, y;

x = 100.0;
y = sqrt(x);        /* return value = 10.0 */
```

far バージョン

なし

srand

rand/srand（8-84 ページ）を参照してください。

sscanf

ストリームの読み込み

C の構文

#include <stdlib.h>

int sscanf(const char *str, const char *format, ...)**;**

C++ の構文

#include <cstdlib>

int std::sscanf(const char *str, const char *format, ...)**;**

定義される場所

rts.src の sscanf.c

説明

sscanf 関数 は、str が指す文字列からの読み込みを行います。format が指す文字列には、ストリームを読み込む方法が記述されています。

far バージョン

なし

strcat

文字列の連結

C の構文

#include <string.h>

char *strcat(char *string1, char *string2)**;**

C++ の構文

#include <cstring>

char *std::strcat(char *string1, char *string2)**;**

定義される場所

rts.src の strcat.c および farstrcat.c

説明

strcat 関数は、string2 のコピー（終了ヌル文字を含む）を string1 の末尾に付加します。string2 の先頭文字は、本来 string1 の終了文字であった ヌル文字に上書きされます。この関数は string1 の値を戻します。

例

次の例では、*a、*b、*c が指す文字列は、コメントに示されている文字列を指すように割り当てられています。コメントの中の \0 の表記は、ヌル文字を表します。

```
char *a, *b, *c;
.
.
.

/* a --> "The quick black fox\0"                           */
/* b --> " jumps over \0"                                  */
/* c --> "the lazy dog.\0"                                 */


strcat (a,b);

/* a --> "The quick black fox jumps over \0"               */
/* b --> " jumps over \0"                                  */
/* c --> "the lazy dog.\0"                                 */

strcat (a,c);

/* a --> "The quick black fox jumps over the lazy dog.\0"*/
/* b --> " jumps over \0"                                  */
/* c --> "the lazy dog.\0"                                 */
```

far バージョン（C）

far char *far_strcat(far char *string1, far char *string2**);**

strchr 文字の最初の出現を検出

C の構文

#include <string.h>

char *strchr(const char *string, int c**);**

C++ の構文

#include <cstring>

char *std::strchr(const char *string, int c**);**

定義される場所

rts.src の strchr.c および farstrchr.c

説明

strchr 関数は、string において最初に現れる c を検出します。strchr で目的の文字が検出されると、その文字へのポインタが戻ります。その文字が存在しない場合はヌル・ポインタ（0）が戻ります。

例

```
char *a = "When zz comes home, the search is on for zs.";
char *b;
char the_z = 'z';

b = strchr(a,the_z);
```

この例では、*b は zz の最初の z を指します。

far バージョン（C）

far char *far_strchr(const far char *string, int c**);**

strcmp/strcoll　文字列の比較

C の構文

```
#include    <string.h>
```

int strcmp(const char *string1, const char *string2)**;**
int strcoll(const char *string1, const char *string2)**;**

C++ の構文

```
#include    <cstring>
```

int std::strcmp(const char *string1, const char *string2)**;**
int std::strcoll(const char *string1, const char *string2)**;**

定義される場所

rts.src の strcmp.c および strcoll.c
rts.src の farstrcmp.c および farstrcoll.c

説明

strcmp 関数と strcoll 関数は、string2 を string1 と比較します。これらの関数は等価です。どちらも ANSI/ISO C/C++ との互換性を確保するための関数です。

この関数は、以下の値のいずれかを戻します。

< 0	*string1 が *string2 より小さいとき。
0	*string1 が *string2 と等しいとき。
> 0	*string1 が *string2 より大きいとき。

例

```
char *stra = "why ask why";
char *strb = "just do it";
char *strc = "why ask why";

if (strcmp(stra, strb) > 0)
    {
        /*    statements here will be executed     */
    }
if (strcoll(stra, strc) == 0)
    {
        /*    statements here will be executed also*/
    }
```

far バージョン（C）

int far_strcmp(const far char *string1, const far char *string2)**;**
int far_strcoll(const far char *string1, const far char *string2)**;**

strcpy　文字列のコピー

C の構文

#include　<string.h>

char *strcpy(char *dest, register const char *src)**;**

C++ の構文

#include　<cstring>

char *std::strcpy(char *dest, register const char *src)**;**

定義される場所

rts.src の strcpy.c および farstrcpy.c

説明

strcpy 関数は、s2（終了ヌル文字を含む）を s1 にコピーします。オーバーラップする文字列をコピーした場合の関数の動作は予測できません。この関数は s1 へのポインタを戻します。

例

次の例で、*a および *b が指す文字列は、2 つの独立した別々のメモリの位置です。コメントの中の \0 の表記は、ヌル文字を表します。

```
char *a = "The quick black fox";
char *b = " jumps over ";

/* a --> "The quick black fox\0"  */
/* b --> " jumps over \0"         */

strcpy(a,b);

/* a --> " jumps over \0"         */
/* b --> " jumps over \0"         */
```

far バージョン（C）

far char *far_strcpy(far char *dest, const far char *src)**;**

strcspn　不一致文字数の検出

C の構文

#include　<string.h>

size_t strcspn(const char *string, const char *chs)**;**

C++ の構文

#include　<cstring>

size_t std::strcspn(const char *string, const char *chs)**;**

定義される場所

rts.src の strcspn.c および farstrcspn.c

説明

strcspn 関数は、chs 内にない文字からだけ構成される string の先頭部分の長さを戻します。string の中の先頭文字が chs にある場合、この関数は 0 を戻します。

例

```
char *stra = "who is there?";
char *strb = "abcdefghijklmnopqrstuvwxyz";
char *strc = "abcdefg";
size_t length;

length = strcspn(stra,strb);   /* length = 0 */
length = strcspn(stra,strc);   /* length = 9 */
```

far バージョン（C） **size_t far_strcspn**(const far char *string, const far char *chs)**;**

strerror 文字列エラー

C の構文

#include <string.h>

char *strerror(int errno)**;**

C++ の構文

#include <cstring>

char *std::strerror(int errno)**;**

定義される場所 rts.src の strerror.c および farstrerror.c

説明 strerror 関数は、文字列「string error」を戻します。この関数は、ANSI/ISO C との互換性を確保するための関数です。

far バージョン（C） **far char *far_strerror**(int errno)**;**

strftime 時間の書式化

C の構文

#include <time.h>

size_t *strftime(char *s, size_t maxsize, const char *format,
const struct tm *timeptr)**;**

C++ の構文

#include <ctime>

size_t *std::strftime(char *s, size_t maxsize, const char *format,
const struct tm *timeptr)**;**

定義される場所 rts.src の strftime.c

説明 strftime 関数は format 文字列に基づいて（timeptr が指す）時間を書式化し、書式化された時間を文字列 s で戻します。s には、最大で maxsize 文字数を書き込むことができます。format パラメータは strftime 関数に時間の書式化方法を指示するための文字列です。表 8-4 に、有効な文字と、それぞれの展開内容を示します。

表 8-4.　strftime の書式パラメータ文字

文字	展開内容
%a	*曜日*の省略形（Mon、Tue など）
%A	*曜日*の正式名
%b	*月*名の省略形（Jan、Feb など）
%B	地方の*月*名の正式名
%c	*日付*と*時刻*の表記
%d	10 進数（0 ～ 31）で表した*日付*
%H	10 進数（00 ～ 23）で表した*時刻*（24 時間制）
%I	10 進数（01 ～ 12）で表した*時刻*（12 時間制）
%j	10 進数（001 ～ 366）で表した*日付*
%m	10 進数（01 ～ 12）で表した*月*
%M	10 進数（00 ～ 59）で表した*分*
%p	各地域ごとの*午前*や*午後*の呼び方
%S	10 進数（00 ～ 59）で表した*秒*
%U	10 進数（00 ～ 52）で表したその年の*週*番号（週の初日は日曜日）
%x	*日付*の表記
%X	*時間*の表記
%y	10 進数（00 ～ 99）で表した*西暦*の下 2 桁
%Y	10 進数で表した*西暦*
%Z	*タイム・ゾーン*名。タイム・ゾーンがない場合は文字なし

time.h/ctime ヘッダで宣言と定義が行われる関数と型の詳細は、8.4.16 項「時間関数（time.h/ctime）」（8-29 ページ）を参照してください。

far バージョン

なし

strlen　文字列の長さの検出

C の構文

#include　<string.h>

size_t **strlen**(const char *string)**;**

C++ の構文

#include　<cstring>

size_t **std::strlen**(const char *string)**;**

定義される場所

rts.src の strlen.c および farstrlen.c

説明

strlen 関数は文字列の長さを戻します。C/C++ では、文字列は値が 0 の文字（ヌル文字）で終了します。戻された結果にはヌル文字は含まれません。

例

```
char *stra = "who is there?";
char *strb = "abcdefghijklmnopqrstuvwxyz";
char *strc = "abcdefg";
size_t length;

length = strlen(stra);    /* length = 13 */
length = strlen(strb);    /* length = 26 */
length = strlen(strc);    /* length = 7  */
```

far バージョン（C）

size_t far_strlen(const far char *string)**;**

strncat　文字列の連結

C の構文

#include　<string.h>

char *strncat(char *dest, const char *src**,** size_t n)**;**

C++ の構文

#include　<cstring>

char *std::strncat(char *dest, const char *src**,** size_t n)**;**

定義される場所

rts.src の strncat.c および farstrncat.c

説明

strncat 関数は、s2 の最大で n 個の文字（終了ヌル文字を含む）を dest に付加します。元の dest の終了文字のヌル文字は、src の先頭文字に上書きされます。strncat 関数は結果にヌル文字を付加します。この関数は dest の値を戻します。

例

次の例では、*a、*b、*c が指す文字列には、コメントに示されている値が割り当てられています。コメントの中の \0 の表記は、ヌル文字を表します。

```
char *a, *b, *c;
size_t size = 13;
.
.
.

/* a--> "I do not like them,\0"               */;
/* b--> " Sam I am, \0"                       */;
/* c--> "I do not like green eggs and ham\0" */;

strncat (a,b,size);

/* a--> "I do not like them, Sam I am, \0"   */;
/* b--> " Sam I am, \0"                       */;
/* c--> "I do not like green eggs and ham\0" */;

strncat (a,c,size);

/* a--> "I do not like them, Sam I am, I do not like\0"   */;
/* b--> " Sam I am, \0"                                   */;
/* c--> "I do not like green eggs and ham\0"              */;
```

far バージョン（C）

far char *far_strncat(far char *dest, const far char *src, size_t n)**;**

strncmp　文字列の比較

C の構文

#include <string.h>

int strncmp(const char *string1, const char *string2, size_t n)**;**

C++ の構文

#include <cstring>

int std::strncmp(const char *string1, const char *string2, size_t n)**;**

定義される場所

rts.src の strncmp.c および farstrncmp.c

説明

strncmp 関数は、s2 の最大で n 個の文字を s1 と比較します。この関数は、以下の値のどれかを戻します。

< 0	*string1 が *string2 より小さいとき。
0	*string1 が *string2 と等しいとき。
> 0	*string1 が *string2 より大きいとき。

例

```
char *stra = "why ask why";
char *strb = "just do it";
char *strc = "why why not?";
size_t size = 4;

if (strncmp(stra, strb, size) > 0)
   {
      /* statements here will get executed */
   }
if (strncmp(stra, strc, size) == 0)
   {
      /* statements here will get executed also */
   }
```

far バージョン（C） **int far_strncmp**(const far char *string1, const far char *string2, size_t n)**;**

strncpy 文字列のコピー

C の構文

#include <string.h>

char *strncpy(register char *dest, register const char *src**,**
register size_t n)**;**

C++ の構文

#include <cstring>

char *strncpy(register char *dest, register const char *src**,**
register size_t n)**;**

定義される場所

rts.src の strncpy.c および farstrncpy.c

説明

strncpy 関数は、最大で n 個の文字を src から dest にコピーします。src の長さが n 文字以上の場合は、src の終わりにヌル文字はコピーされません。重複した文字列から文字をコピーすると、この関数の動作は予測できません。src の長さが n 文字未満の場合、strncpy はヌル文字を dest に追加し、dest の文字数が n 文字になるように調整します。この関数は dest の値を戻します。

例

strb の前には空白があって、この文字列が 5 文字の長さになっていることに注意してください。また strc の最初の 5 文字は I、空白、ワード am、空白となっているので、次に strncpy を実行すると、stra は後ろに 2 つの空白が続く I am というフレーズで始まります。コメントの中の \0 の表記は、ヌル文字を表します。

```
char *stra = "she's the one mother warned you of";
char *strb = " he's";
char *strc = "I am the one father warned you of";
char *strd = "oops";
int length = 5;

strncpy (stra,strb,length);

/* stra--> " he's the one mother warned you of\0" */;
/* strb--> " he's\0"                              */;
/* strc--> "I am the one father warned you of\0"  */;
/* strd--> "oops\0"                               */;

strncpy (stra,strc,length);

/* stra--> "I am  the one mother warned you of\0" */;
/* strb--> " he's\0"                              */;
/* strc--> "I am the one father warned you of\0"  */;
/* strd--> "oops\0"                               */;

strncpy (stra,strd,length);

/* stra--> "oops\0"                               */;
/* strb--> " he's\0"                              */;
/* strc--> "I am the one father warned you of\0"  */;
/* strd--> "oops\0"                               */;
```

far バージョン（C） **far char *far_strncpy(**far char *dest, const far char *src, size_t n**);**

strpbrk 一致する文字の検出

C の構文

#include <string.h>

char *strpbrk(const char *string, const char *chs**);**

C++ の構文

#include <cstring>

char *std::strpbrk(const char *string, const char *chs**);**

定義される場所

rts.src の strpbrk.c および farstrpbrk.c

説明

strpbrk 関数は、string の中で、chs の*いずれかの*文字が最初に現れる位置を検索します。一致する文字を検出すると、strpbrk はその文字を指すポインタを戻します。その文字が存在しない場合は、ヌル・ポインタ（0）を戻します。

例

```
char *stra = "it wasn't me";
char *strb = "wave";
char *a;

a = strpbrk (stra,strb);
```

この例の実行後、*a は was't の中の w を指します。

far バージョン（C） **far char *far_strpbrk(**const far char *string, const far char *chs**);**

strrchr 文字の最後の出現を検出

C の構文

#include <string.h>

char *strrchr(const char *string, int c**);**

C++ の構文

#include <cstring>

char *std::strrchr(const char *string, int c**);**

定義される場所 rts.src の strrchr.c および farstrrchr.c

説明 strrchr 関数は、string の中で最後に現れる c を検出します。その文字を検出すると、strrchr はその文字を指すポインタを戻します。その文字が存在しない場合は、ヌル・ポインタ（0）を戻します。

例

```
char *a = "When zz comes home, the search is on for zs";
char *b;
char the_z = 'z';
b = strrchr (a, the_z);
```

この例の実行後、*b は文字列の終わりに近い zs の中の z を指します。

far バージョン（C） **far char *far_strrchr(**const far char *string, int c**);**

strspn 一致する文字数の検出

C の構文

#include <string.h>

size_t strspn(const char *string, const char *chs**);**

C++ の構文

#include <cstring>

size_t std::strspn(const char *string, const char *chs**);**

定義される場所 rts.src の strspn.c および farstrspn.c

説明 strspn 関数は、chs にある文字だけで構成された string の先頭部分の長さを戻します。string の中の先頭文字が chs にない場合、strspn 関数は 0 を戻します。

例

```
char *stra = "who is there?";
char *strb = "abcdefghijklmnopqrstuvwxyz";
char *strc = "abcdefg";
size_t length;

length = strcspn(stra,strb);    /* length = 3 */
length = strcspn(stra,strc);    /* length = 0 */
```

far バージョン（C）

size_t far_strspn(const far char *string, const far char *chs**);**

strstr

一致する文字列の検出

C の構文

#include <string.h>

char *strstr(const char *string1, const char *string2**);**

C++ の構文

#include <cstring>

char *std::strstr(const char *string1, const char *string2**);**

定義される場所

rts.src の strstr.c および farstrstr.c

説明

strstr 関数は、string1 の中で string2（終了ヌル文字を除く）が最初に現れる位置を検索します。strstr は一致する文字列を見つけると、見つかったその文字列へのポインタを戻します。一致する文字列が見つからなかった場合、この関数はヌル・ポインタを戻します。string2 が長さ 0 の文字列を指す場合、strstr は string1 を戻します。

例

```
char *stra = "so what do you want for nothing?";
char *strb = "what";
char *ptr;

ptr = strstr(stra,strb);
```

ポインタ *ptr は、現在、最初の文字列 what の中の w を指しています。

far バージョン（C）

far char *far_strstr(const far char *string1, const far char *string2**);**

strtod/strtol/strtoll/strtoul/strtoull

文字列から数値への変換

C の構文

#include <stdlib.h>

double strtod(const char *st**,** char **endptr**);**
long strtol(const char *st**,** char **endptr, int base**);**
long long strtoll(const char *st**,** char **endptr, int base**);**
unsigned long strtoul(const char *st, char **endptr, int base**);**
unsigned long long strtoull(const char *st, char **endptr, int base**);**

C++ の構文

#include <cstdlib>

double std::strtod(const char *st**,** char **endptr**);**
long std::strtol(const char *st**,** char **endptr, int base**);**
long long strtoll(const char *st**,** char **endptr, int base**);**
unsigned long std::strtoul(const char *st, char **endptr, int base**);**
unsigned long long strtoull(const char *st, char **endptr, int base**);**

定義される場所

rts.src の strtod.c、strtol.c、strtoll.c、strtoul.c、および strtoull.c
rts.src の farstrtod.c、farstrtol.c、farstrtoll.c、farstrtoul.c、および farstrtoull.c

説明

上記の 5 つの関数は、ASCII 文字列を数値に変換します。それぞれの関数の引数 st は元の文字列を指します。引数 endptr はポインタを指します。これらの関数は、変換された文字列の後にある最初の文字を指すようにこのポインタを設定します。整数への変換を行う関数には、3 番目の引数 base もあります。この引数は、どの底で文字列を変換するかを関数に指示します。

- strtod 関数は、文字列を浮動小数点値に変換します。文字列の形式は次のとおりです。

  [*space*] [*sign*] *digits* [*.digits*] [e|E [*sign*] *integer*]

  この関数は、変換後の文字列を戻します。元の文字列が空のときや、その形式が正しくないときは、この関数は 0 を戻します。変換される文字列による値がオーバーフローすると、この関数は、±HUGE_VAL を戻します。変換される文字列による値がアンダーフローすると、この関数は 0 を戻します。変換される文字列による値がオーバーフローやアンダーフローすると、errno が ERANGE の値に設定されます。

- strtol 関数は、文字列を長整数に変換します。文字列の形式は、次のとおりです。

  [*space*] [*sign*] *digits* [*.digits*] [e|E [*sign*] *integer*]

- strtoll 関数は、文字列を long long 整数に変換します。文字列の形式は、次のとおりです。

  [*space*] [*sign*] *digits* [*.digits*] [e|E [*sign*] *integer*]

- strtoul 関数は、文字列を符号なし長整数に変換します。文字列の形式は、次のとおりです。

  [*space*] [*sign*] *digits* [*.digits*] [e|E [*sign*] *integer*]

- ❑ strtoull 関数は、文字列を符号なし倍長整数に変換します。文字列の形式は、次のとおりです。

  [*space*] [*sign*] *digits* [*.digits*] [e|E [*sign*] *integer*]

space は、水平タブか垂直タブ、スペースバー、復帰、書式送り、改行を組み合わせて示します。space の後は、オプションの sign、さらに数値の整数部を示す digits が続きます。その後に数値の小数部が続き、オプションの sign をもつ指数部が続きます。

認識できない文字が初めて現れた時点で、文字列は終わります。endptr が指すポインタは、この文字を指すように設定されます。

far バージョン（C）

double far_strtod(const far char *st, far char **endptr)**;**
long far_strtol(const far char *st, far char **endptr, int base)**;**
long long far_strtoll(const far char *st, far char **endptr, int base)**;**
unsigned long far_strtoul(const far char *st, far char **endprt, int base)**;**
unsigned long long far_strtoull(const far char *st, far char **endprt, int base)**;**

strtok 文字列をトークンに分割

C の構文

#include <string.h>

char *strtok(char *str1, const char *str2)**;**

C++ の構文

#include <cstring>

char *std::strtok(char *str1, const char *str2)**;**

定義される場所

rts.src の strtok.c および strtok.c

説明

strtok 関数を連続して呼び出すと、str1 は str2 の文字で区切られる一連のトークンに分割されます。呼び出すたびに次のトークンへのポインタが戻ります。

例

次の例では strtok を最初に呼び出すと、ポインタ stra は文字列 excuse\0 を指します。これは、最初の空白のあった位置に strtok がヌル文字を挿入したからです。コメントの中の \0 の表記は、ヌル文字を表します。

```
char *stra = "excuse me while I kiss the sky";
char *ptr;

ptr = strtok (stra," ");  /* ptr --> "excuse\0" */
ptr = strtok (0," ");     /* ptr --> "me\0"     */
ptr = strtok (0," ");     /* ptr --> "while\0"  */
```

far バージョン（C）

far char *far_strtok(far char *str1, const far char *str2)**;**

strxfrm　文字の変換

C の構文

#include　<string.h>

size_t strxfrm(char *to, const char *from, size_t n)**;**

C++ の構文

#include　<cstring>

size_t std::strxfrm(char *to, const char *from, size_t n)**;**

定義される場所

rts.src の strxfrm.c および farstrxfrm.c

説明

strxfrm 関数は from が指す n 個の文字を to が指す n 個の文字に変換します。

far バージョン（C）

size_t far_strxfrm(far char *to, const far char *from, size_t n)**;**

tan　タンジェント

C の構文

#include　<math.h>

double tan(double x)**;**

C++ の構文

#include　<cmath>

double std::tan(double x)**;**

定義される場所

rts.src の tan.c

説明

tan 関数は、浮動小数点数 x のタンジェントを戻します。角度 x は、ラジアンで表します。引数の値が大きすぎると、有意性のある結果がほとんど得られないか、全く得られなくなります。

例

```
double x, y;

x = 3.1415927/4.0;
y = tan(x);          /* return value = 1.0 */
```

far バージョン

なし

tanh　ハイパボリック・タンジェント

C の構文

#include　<math.h>

double tanh(double x)**;**

C++ の構文

#include　<cmath>

double std::tanh(double x)**;**

定義される場所

rts.src の tanh.c

説明

tanh 関数は、浮動小数点数 x のハイパボリック・タンジェントを戻します。

例

```
double x, y;

x = 0.0;
y = tanh(x);          /* return value = 0.0 */
```

far バージョン

なし

time　時間

C の構文

#include　<time.h>

time_t time(time_t *timer)**;**

C++ の構文

#include　<ctime>

time_t std::time(time_t *timer)**;**

定義される場所

rts.src の time.c

説明

time 関数は、秒数で表される現在のカレンダ時を判定します。カレンダ時を使用できないとき、この関数は -1 を戻します。timer がヌル・ポインタでない場合、この関数は timer が指すオブジェクトへの戻り値の代入も行います。

time.h/ctime ヘッダで宣言と定義が行われる関数と型の詳細は、8.4.16 項「時間関数（time.h/ctime）」（8-29 ページ）を参照してください。

> **注： time 関数はターゲット・システム固有**
>
> time 関数はターゲット・システムによって異なるため、ユーザ固有の time 関数を記述する必要があります。

far バージョン

なし

tmpfile　一時ファイルの作成

C の構文

#include　<stdlib.h>

FILE ***tmpfile**(void)**;**

C++ の構文

#include　<cstdlib>

FILE ***std::tmpfile**(void)**;**

定義される場所

rts.src の tmpfile.c

説明

tmpfile 関数は、一時ファイルを作成します。

far バージョン

なし

tmpnam　有効なファイル名の生成

C の構文

#include　<stdlib.h>

char *tmpnam(char *_s)**;**

C++ の構文

#include　<cstdlib>

char *std::tmpnam(char *_s)**;**

定義される場所

rts.src の tmpnam.c

説明

tmpnam 関数は、有効なファイル名である文字列を生成します。

far バージョン

なし

toascii　ASCII への変換

C の構文

#include　<ctype.h>

int toascii(int c)**;**

C++ の構文

#include　<cctype>

int std::toascii(int c)**;**

定義される場所

rts.src の toascii.c

説明

toascii 関数は、下位 7 ビットをマスクすることにより c を有効な ASCII 文字にすることができます。この関数には、同機能のマクロ呼び出し _toascii があります。

far バージョン

なし

tolower/toupper　大文字と小文字の変換

C の構文

#include　<ctype.h>

int tolower(int c)**;**
int toupper(int c)**;**

C++ の構文

#include　<cctype>

int std::tolower(int c)**;**
int std::toupper(int c)**;**

定義される場所

rts.src の tolower.c および toupper.c

説明

これら 2 つの関数は、1 文字の英字 c を大文字または小文字に変換します。

- ❑ tolower 関数は、大文字の引数を小文字に変換します。c が大文字でない場合、tolower はそのまま戻します。
- ❑ toupper 関数は、小文字の引数を大文字に変換します。c が小文字でない場合、toupper はそのまま戻します。

この関数には、同機能のマクロ _tolower と _toupper があります。

far バージョン

なし

ungetc　ストリームへの文字の書き込み

C の構文

#include　<stdlib.h>

int ungetc(int c, FILE *_fp)**;**

C++ の構文

#include　<cstdlib>

int std::ungetc(int c, FILE *_fp)**;**

定義される場所

rts.src の ungetc.c

説明

ungetc 関数は、_fp が指すストリームに文字 c を書き込みます。

far バージョン

なし

va_arg/va_end/ va_start 可変引数マクロ

C の構文

```
#include    <stdarg.h>

typedef  char *va_list;
         va_arg(_ap, _type);
         void va_end(_ap);
         void va_start(_ap, parmN);
```

C++ の構文

```
#include    <cstdarg>

typedef  char *va_list;
         va_arg(_ap, _type);
         void va_end(_ap);
         void va_start(_ap, parmN);
```

定義される場所

rts.src の stdarg.h および cstdarg

説明

関数の中には、型が変化する可変個の引数で呼び出されるものがあります。このような関数は可変引数関数といいますが、以下のマクロを使用して、その引数リストを実行時に 1 つずつ処理することができます。_ap パラメータは可変引数リスト内の引数を指します。

- va_start マクロは、可変引数関数の引数リスト内の先頭の引数を指すように _ap を初期化します。parmN パラメータは、宣言された固定リスト内の右端のパラメータを指します。
- va_arg マクロは、可変引数関数に対する呼び出しで次の引数の値を戻します。va_arg を連続的に呼び出すことで、可変引数関数の一連の引数を戻せるようにするため、va_arg を呼び出すたびに _ap が変更されます（va_arg は、リスト内の次の引数を指すように _ap を変更します）。type パラメータは型の名前を示します。このパラメータは、リスト内の現在の引数の型を示します。
- va_end マクロは、va_start と va_arg を使用後にスタック環境をリセットします。

va_arg や va_end を呼び出す前に、va_start を呼び出して ap を初期化してください。

例

```
int  printf (char *fmt...);
     va_list ap;
     va_start(ap, fmt);
     .
     .
     .
     i = va_arg(ap, int);    /* Get next arg, an integer */
     s = va_arg(ap, char *); /* Get next arg, a string   */
     l = va_arg(ap, long);   /* Get next arg, a long     */
     .
     .
     .
     va_end(ap);             /* Reset                    */
}
```

far バージョン なし

vfprintf ストリームへの書き込み

C の構文

#include <stdlib.h>

int vfprintf(FILE *_fp, const char *_format, va list _ap)**;**

C++ の構文

#include <cstdlib>

int std::vfprintf(FILE *_fp, const char *_format, va list _ap)**;**

定義される場所 rts.src の vfprintf.c

説明

vfprintf 関数は、_fp が指すストリームへの書き込みを行います。_format が指す文字列には、ストリームを書き込む方法が記述されています。引数リストは _ap で指定します。

far バージョン なし

vprintf 標準出力への書き込み

C の構文

#include <stdlib.h>

int vprintf(const char *_format, va list _ap)**;**

C++ の構文

#include <cstdlib>

int std::vprintf(const char *_format, va list _ap)**;**

定義される場所 rts.src の vprintf.c

説明

vprintf 関数は、標準出力デバイスへの書き込みを行います。_format が指す文字列には、ストリームを書き込む方法が記述されています。引数リストは _ap で指定します。

far バージョン なし

vsprintf　ストリームの書き込み

C の構文

#include <stdlib.h>

int vsprintf(char *string, const char *_format, va list _ap)**;**

C++ の構文

#include <cstdlib>

int std::vsprintf(char *string, const char *_format, va list _ap)**;**

定義される場所

rts.src の vsprintf.c

説明

vsprintf 関数は、_string が指す配列への書き込みを行います。_format が指す文字列には、ストリームを書き込む方法が記述されています。引数リストは _ap で指定します。

far バージョン

なし

第 9 章

ライブラリ作成ユーティリティ

C/C++ コンパイラを使用すると、多くの異なる構成と互いに互換性を確保する必要のないオプションに基づいてコードをコンパイルすることができます。個々のランタイムサポート・ライブラリを可能なすべての組み合わせで作成して組み込む作業はかなり煩雑であるため、このパッケージにはソース・アーカイブである rts.src が組み込まれています。rts.src には、ランタイムサポート関数がすべて組み込まれています。

パッケージに付属している事前作成済みのランタイムサポート・ライブラリは、次のとおりです。

- rts2800.lib（C/C++ ランタイム・オブジェクト・ライブラリ）
- rts2800_ml.lib（C/C++ ラージ・メモリ・モデルのランタイムサポート・オブジェクト・ライブラリ）

アーカイバと mk2000 -v28 ユーティリティを使用することで、独自のランタイムサポート・ライブラリを作成できます。mk2000 -v28 ユーティリティについては、本章で説明します。アーカイバについては、『TMS320C28x アセンブリ言語ツール ユーザーズ・マニュアル』を参照してください。

項目 **ページ**

9.1 ライブラリ作成ユーティリティの起動方法 9-2
9.2 ライブラリ作成ユーティリティのオプション 9-3
9.3 オプションのまとめ 9-4

9.1 ライブラリ作成ユーティリティの起動方法

ライブラリ作成ユーティリティを起動する構文は、以下のとおりです。

mk2000 -v28 [*options*] *src_arch1* [**-l***obj.lib1*] [*src_arch2* [**-l***obj.lib2*]]...

mk2000 -v28 ユーティリティを起動するコマンドです。

options オプションによって、ライブラリ作成ユーティリティによるファイルの処理方法が制御されます。このオプションは、コマンド行またはリンカ・コマンド・ファイルの任意の場所に指定できます（オプションについては、9.2 節および 9.3 節を参照）。

src_arch ソース・アーカイブ・ファイルの名前です。mk2000 -v28 は、コマンド行オプションで指定されたランタイム・モデルに従って、指定されたソース・アーカイブのオブジェクト・ライブラリを作成します。

-l*obj.lib* オプションのオブジェクト・ライブラリ名です。ライブラリ名が指定されていないと、mk2000 -v28 はソース・アーカイブの名前に接尾部 .lib を付けます。指定されたそれぞれのソース・アーカイブ・ファイルに対して、対応するオブジェクト・ライブラリ・ファイルが作成されます。複数のソース・アーカイブ・ファイルから 1 つのオブジェクト・ライブラリを作成することはできません。

mk2000 -v28 ユーティリティは、アーカイブ内の各ソース・ファイルのコンパイルとアセンブルの両方、またはそのいずれかを行うため、各ソース・ファイルに対してシェル・プログラムを実行します。次にすべてのオブジェクト・ファイルを収集して、1 つのオブジェクト・ライブラリを作成します。ツールはすべて、PATH 環境変数に指定した場所になければなりません。このユーティリティでは、環境変数 TMP、C_OPTION、および C_DIR は無視されます。

注：　ランタイムサポート・ライブラリ

ランタイムサポート・ライブラリを作成する場合、--rts オプションを指定しなければなりません。このオプションは、ランタイムサポート・ライブラリを適切に作成するために必要となる固有のオプションをコンパイラに渡します。

9.2 ライブラリ作成ユーティリティのオプション

コマンド行のオプションのほとんどは、コンパイラ、アセンブラ、リンカ、およびシェルが使用する同じ名前のオプションに直接対応しています。ライブラリ作成ユーティリティにだけ適用するオプションは次のとおりです。

--c ソース・アーカイブに含まれる C ソース・ファイルを抽出します。これらのファイルは、ユーティリティの実行完了後にカレント・ディレクトリに残ります。

--h ソース・アーカイブに含まれるヘッダ・ファイルを使用します。このヘッダ・ファイルは、ユーティリティの実行完了後にカレント・ディレクトリに残ります。ツールに付属している rtsc.src または rtscpp.src アーカイブからランタイムサポート・ヘッダ・ファイルをインストールするときに、このオプションを使用します。

--k ファイルを上書きします。デフォルトでは、このユーティリティが作成するオブジェクト・ファイルと同じ名前をもつオブジェクト・ファイルがすでにカレント・ディレクトリ内に存在する場合、このユーティリティは終了します。この場合、ユーザが指定したオブジェクト・ファイル名であるか、またはユーティリティが生成したファイル名であるかどうかは関係ありません。

--q ヘッダ情報を抑止します（静的）。

--rts ランタイムサポート・ライブラリ作成固有のオプションをコンパイラに渡します。--rts オプションを指定しなければならないのは、ランタイムサポート・ライブラリを作成する場合です。

--u （ソース・アーカイブに含まれているヘッダ・ファイルではなく）既存のヘッダ・ファイルを使用して、オブジェクト・ライブラリを作成します。必要なヘッダがすでにカレント・ディレクトリ内にある場合は、これらのヘッダ・ファイルを再インストールする必要はありません。このオプションを使用すると、独自のアプリケーションに合わせてランタイムサポート関数を自由に変更できます。

--v ユーティリティの実行時に進捗情報を画面に表示します。通常は、ユーティリティの実行時にはそのような情報は表示されません（画面メッセージなし）。

オプションの概要をオンライン表示するには、コマンド行でパラメータを指定せずに **mk2000 -v28** と入力します。

9.3 オプションのまとめ

ライブラリ作成ユーティリティと組み合わせて使用できるその他のオプションは、コンパイラやアセンブラと組み合わせて使用するオプションに直接対応しています。表 9-1 に、このようなオプションのリストを示します。また、これらのオプションの詳細は「ページ」の欄に示すページを参照してください。

表 9-1. オプションとその機能のまとめ

(a) コンパイラ / シェルを制御するオプション

オプション	機能	ページ
-D*name*[=*def*]	*name* を事前定義します。	2-13
-g	シンボリック・デバッグを有効にします。	2-15
-U*name*	*name* を未定義にします。	2-14

(b) パーサを制御するオプション

オプション	機能	ページ
-pi	定義優先の制御によるインライン展開を抑止します（ただし、-O3 最適化は自動インライン展開を実行し続けます)。	2-41
-pk	K&R 互換コードを生成します。	6-35
-pr	緩和モードを有効にします。厳密な ANSI/ISO 違反を無視します。	6-38
-ps	厳密な ANSI/ISO モード（C/C++ に対してであり、K&R C に対してではない）を有効にします。	6-38

(c) 診断を制御するオプション

オプション	機能	ページ
-pdr	注釈（軽い警告）を発行します。	2-34
-pdv	行の折り返し付きでオリジナル・ソースを表示する詳細な診断情報を提供します。	2-34
-pdw	警告診断を抑止します（エラーは発行されます)。	2-34

表 9-1. オプションとその機能のまとめ(続き)

(d) 最適化レベルを制御するオプション

オプション	機能	ページ
-O0	レジスタ最適化を行ってコンパイルします。	3-2
-O1	-O0 最適化とローカル最適化を行ってコンパイルを実行します。	3-2
-O2 (または -O)	-O1 最適化とグローバル最適化を行ってコンパイルを実行します。	3-2
-O3	-O2 による最適化を行い、さらにファイルに対して最適化を行ってコンパイルを実行します。 mk2000 -v28 によって -ol0 と -op0 が自動的に設定されることに注意してください。	3-2

(e) マシン固有のオプション

オプション	機能	ページ
-ma	エイリアスが設定された変数と見なします。	3-11
-md	DP ロード命令の最適化を無効にします。	2-16
-me	高速分岐命令の生成を無効にします。	2-16
-mf	サイズではなく、スピードを優先した最適化を行います。	2-16
-mi	RPT 命令の生成を無効にします。	2-16
-ml	ラージ・メモリ・モデルのコードを生成します。	6-21
-mn	-g が無効にした最適化を有効にします。	2-17
-ms	スピードよりサイズを優先した最適化を行います。	2-17、 3-18
-mt	単一メモリを有効にします。	2-17
-v28	TMS320C28x アーキテクチャに対応したコードを生成します。	2-18

(f) アセンブラを制御するオプション

オプション	機能	ページ
-as	ラベルをシンボルとして保持します。	2-22

(g) デフォルトのファイル拡張子を変更するオプション

オプション	機能	ページ
-ea[.]*extension*	アセンブリ・ファイルのデフォルトの拡張子を設定します。	2-20
-eo[.]*extension*	オブジェクト・ファイルのデフォルトの拡張子を設定します。	2-20

第 10 章

C++ ネーム・デマングラ・ユーティリティ

C++ コンパイラでは、関数の多重定義、演算子の多重定義、および型を気にする必要がないリンクを実現するために、その関数の識別記号（シグニチャ）をリンクレベル名にエンコードしています。シグニチャをリンク名にエンコードするプロセスは、*ネーム・マングリング*と呼ばれています。アセンブリ・ファイルやリンカ出力においてマングルされた名前を調べる場合、C++ ソース・コードの名前とマングルされた名前を関連付けるのが難しいことがあります。C++ *ネーム・デマングラ*はデバッグ補助機能であり、各マングル名を C++ ソース・コード中の元の名前に変換します。

本章では、C++ ネーム・デマングラの起動方法と使用方法を説明します。C++ ネーム・デマングラは入力データを読み込み、マングルされた名前を探します。マングルされていないすべてのテキストは、変更されずにそのまま出力にコピーされます。マングルされた名前は、すべてデマングルされてから出力にコピーされます。

項目 **ページ**

項目	ページ
10.1 C++ ネーム・デマングラの起動方法	10-2
10.2 C++ ネーム・デマングラのオプション	10-3
10.3 C++ ネーム・デマングラの使用例	10-4

10.1　C++ ネーム・デマングラの起動方法

C++ ネーム・デマングラを起動するには、次のように入力します。

dem2000 [*options*] [*filenames*]

dem2000	ネーム・デマングラを実行するコマンドです。
options	デマングラの動作方法を制御するオプションです （オプションについては、10.2 節「C++ ネーム・デマングラのオプション」（10-3 ページ）を参照)。
filenames	コンパイラによるアセンブリ・ファイル出力、アセンブラ・リスト・ファイル、リンカ・マップ・ファイルなどのテキスト入力ファイルです。コマンド行にファイル名が指定されていない場合、dem2000 は標準入力を使用します。

デフォルトでは、C++ ネーム・デマングラは標準出力に出力します。ファイルに出力する場合は、-o *file* オプションを指定します。

10.2　C++ ネーム・デマングラのオプション

C++ ネーム・デマングラを制御するオプションと、その機能についての説明を以下に示します。

-h　　C++ ネーム・デマングラのオプションの概要をオンライン表示するヘルプ画面を出力します。

-o *file*　　標準出力ではなく、指定された *file* に出力します。

-u　　外部名に C++ のプレフィックスがないことを指定します。

-v　　詳細モードを有効にします（見出しを出力します）。

10.3 C++ ネーム・デマングラの使用例

例 10-1 (a) に、C++ のサンプル・プログラムと TMS320C28x コンパイラによって出力される結果のアセンブリ・コードを示します。例 10-1 (b) では、foo() および compute() のリンク名はマングルされます。つまり、この 2 つのシグネチャ情報はそれぞれの名前にエンコードされます。

例 10-1. 名前のマングリング

(a) C++ プログラム

```
int compute(int val, int *err);

int foo(int val, int *err)
{
   static int last_err = 0;
   int        result   = 0;

   if (last_err == 0)
   result = compute(val, &last_err);

   *err = last_err;
   return result;
}
```

例 10-1. 名前のマングリング（続き）

(b) 結果的に作成される foo() のアセンブリ・コード

```
;***************************************************************
;* FNAME: _foo__FiPi                     FR SIZE:   4          *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                            0 Parameter,  3 Auto,  0 SOE     *
;***************************************************************

_foo__FiPi:
       ADDB       SP,#4
       MOVZ       DP,#_last_err$1
       MOV        *-SP[1],AL            ; |4|
       MOV        AL,@_last_err$1       ; |8|
       MOV        *-SP[2],AR4           ; |4|
       MOV        *-SP[3],#0            ; |6|
       BF         L1,NEQ                ; |8|
       ; branch occurs ; |8|
       MOVL       XAR4,#_last_err$1     ; |9|
       MOV        AL,*-SP[1]            ; |9|
       LCR        #_compute__FiPi       ; |9|
       ; call occurs [#_compute__FiPi]  ; |9|
       MOV        *-SP[3],AL            ; |9|
L1:
       MOVZ       AR6,*-SP[2]           ; |11|
       MOV        *+XAR6[0],*(0:_last_err$1) ; |11|
       MOV        AL,*-SP[3]            ; |12|
       SUBB       SP,#4                 ; |12|
       LRETR
       ; return occurs
```

C++ ネーム・デマングラ・ユーティリティを実行すると、マングル対象と見なされるすべての名前をデマングルします。次のように入力すると仮定します。

% dem2000 foo.asm

例 10-2 に、その結果を示します。foo() および compute() のリンク名はデマングルされていることに注意してください。

例 10-2. C++ ネーム・デマングラ・ユーティリティ実行後の結果

```
;***************************************************************
;* FNAME: foo(int, int *)               FR SIZE:   4           *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                            0 Parameter,  3 Auto,  0 SOE     *
;***************************************************************

foo(int, int *):
       ADDB       SP,#4
       MOVZ       DP,#_last_err$1
       MOV        *-SP[1],AL              ; |4|
       MOV        AL,@_last_err$1        ; |8|
       MOV        *-SP[2],AR4             ; |4|
       MOV        *-SP[3],#0              ; |6|
       BF         L1,NEQ                  ; |8|
       ; branch occurs ; |8|
       MOVL       XAR4,#_last_err$1      ; |9|
       MOV        AL,*-SP[1]              ; |9|
       LCR        #compute(int, int *)         ; |9|
       ; call occurs [#compute(int, int *)]  ; |9|
       MOV        *-SP[3],AL              ; |9|
L1:
       MOVZ       AR6,*-SP[2]             ; |11|
       MOV        *+XAR6[0],*(0:_last_err$1) ; |11|
       MOV        AL,*-SP[3]              ; |12|
       SUBB       SP,#4                   ; |12|
       LRETR
       ; return occurs
```

付録 A

用語集

A

ANSI（米国規格協会）： 米国規格を承認する委員会。

B

.bss： デフォルトの COFF セクションの 1 つ。.bss 疑似命令を使用するとメモリ・マップに一定量の空間を確保でき、その空間に後でデータを格納することができます。.bss セクションは初期化されません。

C

C コンパイラ： C のソース文をアセンブリ言語のソース文に変換するプログラム。

D

.data： デフォルトの COFF セクションの 1 つ。.data セクションは、初期化されたデータを含む初期化されたセクションです。.data 疑似命令を使用すると、コードを .data セクションの中にアセンブルできます。

G

GROUP： 指定された出力セクションを（グループとして）連続して強制的に割り当てるために使用される SECTIONS 疑似命令のオプション。

H

Hex 変換ユーティリティ：　COFF オブジェクト・ファイルを EPROM プログラマにロードするのに適した複数の標準 ASCII 16 進数フォーマットのいずれかに変換するユーティリティ。

I

ISO（国際標準化機構）：　国際的組織、政府、業界、企業、および消費者団体の代表と協力して取り組んでいる 140 カ国から構成される国家規格を制定する委員会のネットワーク。ISO は公共部門と民間部門をつなぐ架け橋となっていて、ISO 9000 および企業、政府、および各種団体のための 13,000 以上の国際規格を制定しています。

K

K&R：　カーニハンとリッチーの C。『The C Programming Language』の第 2 版で規定された、事実上の標準規格。以前の ISO に対応しない C コンパイラ用に記述された K&R C のプログラムの大部分は、修正なしで正しくコンパイルされ、実行されます。

R

RAM：　ランダム・アクセス・メモリ。

ROM：　読み出し専用メモリ。

S

SDRAM：　シンクロナス・ダイナミック・ランダム・アクセス・メモリ。

T

.text：　デフォルトの COFF セクションの 1 つ。実行可能なコードを含む初期化されたセクションです。.text 疑似命令を使用すると、コードを .text セクションにアセンブルできます。

あ

アーカイバ： 複数のファイルをアーカイブ・ライブラリと呼ばれる単一のファイルにグループ化するためのソフトウェア・プログラム。アーカイバを使うと、新しいメンバの追加だけでなく、アーカイブ・ライブラリ内のファイルを削除、抽出、置換することができます。

アーカイブ・ライブラリ： 単一のファイル内にグループ化されたファイルの集合。

アセンブラ： アセンブリ言語命令、疑似命令、およびマクロ疑似命令を含むソース・ファイルから、機械語のプログラムを作成するソフトウェア・プログラム。アセンブラはシンボリックな命令コードを絶対的な命令コードに置き換え、シンボリックなアドレスを絶対アドレスまたは再配置可能なアドレスに置き換えます。

アライメント： 「位置合わせ」を参照。

い

位置合わせ： リンカが *n* ビット境界に当たるアドレスに出力セクションを格納するプロセス。ここで、*n* は 2 の累乗。SECTIONS リンカ疑似命令を使って位置合わせを指定することができます。

インクリメンタル・リンク： すでにリンクされているファイルをリンクすること。

インターリスト・ユーティリティ： 元の C ソース文をアセンブラから出されるアセンブリ言語出力に（コメントとして）差し込む機能。

え

エイリアス指定： 単一のオブジェクトに複数の方法でアクセスできる場合、たとえば、2 つのポインタが単一のオブジェクトを指す場合などに実行されます。エイリアス指定を実行すると、間接参照によって別のオブジェクトが潜在的に参照される可能性があるので、最適化が正しく行われない場合もあります。

エミュレータ： TMS320C28x と同じ入力を受け付け、同じ出力を作成するハードウェア開発システム。

エントリ・ポイント： プログラムが実行を開始するターゲット・メモリ内のロケーション。

お

オーバーレイ・ページ：　異なる時間に同じアドレス空間を占めることのできる物理メモリの複数のエリア。TMS320C28x デバイスは、ハードウェア選択信号に対応して別々のページを同じアドレス空間にマップすることができます。

オブジェクト・ファイル：　アセンブルまたはリンクされていて、機械語オブジェクト・コードを含むファイル。

オブジェクト・モジュール：　実行可能なプログラムを作成するために一緒にリンクされているオブジェクト・ファイルのグループ。

オブジェクト・ライブラリ：　複数のオブジェクト・ファイルで構成されているアーカイブ・ライブラリ。

オプション：　ソフトウェア・ツールの起動時に、追加の機能や特定の機能を実行させるために使用するコマンド行パラメータ。

オプション・ヘッダ：　COFF オブジェクト・ファイルの一部分。リンカがダウンロード時に、再配置を行うために使用します。

オプティマイザ：　ターゲット・アーキテクチャの機能を最大限に利用するようにコードの一部を書き換えることで、C プログラムの実行速度の向上およびサイズの縮小を行うソフトウェア・ツール。

か

外部シンボル：　現行のプログラム・モジュールで使用されるが、その定義が異なるプログラム・モジュールで行われるシンボル。

関数のインライン展開：　関数のコードを呼び出し位置に挿入するプロセス。これにより、関数呼び出しによるオーバーヘッド（関数の開始と終了に必要なコード）を軽減します。また、関数をインライン展開することでオプティマイザは関数コードを周囲のコードとのコンテキストを配慮した効率のよいものにすることができるようになります。

き

記憶クラス：　シンボルへのアクセス方法を示すシンボル・テーブルのエントリ。

共通オブジェクト・ファイル・フォーマット（COFF）：　*セクション*の考え方をサポートすることにより、モジュラー・プログラミングを促進するオブジェクト・ファイル・フォーマット。

行番号エントリ：　アセンブリ・コードの行を元となるオリジナルの C ソース・ファイルにマップする COFF 出力モジュール内の項目。

共用体変数：　実行時間によって異なる型およびサイズのオブジェクトを保持する可能性のある変数。

く

グローバル・シンボル：　次のいずれかの条件を満たすシンボルの一種です。1) 現行のモジュールで定義されていて、他のモジュールでアクセスされている、または 2) 現行のモジュールでアクセスされているが、他のモジュールで定義されている。

クロスリファレンス・リスト：　アセンブラにより作成される出力ファイル。定義されたシンボル、シンボルを定義した行、シンボルを参照する行、およびその最終値が表示されます。

こ

構成メモリ：　リンカが割り当てるために指定するメモリ。

構造体：　グループ化されて 1 つの名前を付けられた、単数または複数の変数の集まり。

コード・ジェネレータ：　パーサまたはオプティマイザによって生成される中間ファイルを受け取り、アセンブリ言語ソース・ファイルを生成するコンパイラ・ツール。

コマンド・ファイル：　コンパイラまたはリンカのオプション、ファイル名、疑似命令、コメントを含むファイル。

コメント：　ソース・ファイルを文書化したり、読みやすくしたりするために使われるソース文（またはその一部）。コメントは、コンパイル、アセンブル、およびリンクされません。つまり、オブジェクト・ファイルには何の影響も及ぼしません。

さ

再配置：　シンボルのアドレスが変更されるときに、リンカがそのシンボルに対するすべての参照を調整するプロセス。

し

式：　定数、シンボル、または算術演算子によって区切られた一連の定数とシンボル。

実行可能モジュール：　ターゲット・システムで実行できるリンクされたオブジェクト・ファイル。

実行時自動初期化モデル：　リンカが C コードのリンク時に使用する自動初期化方法。リンカを起動するときに、-c オプションを指定すると、このモデルが使用されます。ROM モデルでは、リンカにより .cinit セクションのデータ・テーブルがメモリにロードされ、変数が実行時に初期化されます。

自動初期化：　プログラムの実行を開始する前に、（.cinit セクションに保持されている）C のグローバル変数を初期化するプロセス。

出力セクション：　リンクされた実行可能なモジュール内の、最終的な割り当て済みセクション。

出力モジュール：　ターゲット・システム上にダウンロードして実行することのできる、リンクされた実行可能なオブジェクト・ファイル。

条件付き処理：　指定された式を評価することによって、ソース・コードのあるブロックまたは代替ブロックを処理する方法。

初期化されたセクション：　実行可能なコードまたは初期化されたデータを含む COFF セクション。初期化されたセクションは、.data 疑似命令、.text 疑似命令、.sect 疑似命令のいずれかで構成されます。

初期化されないセクション：　メモリ・マップ内に空間は確保されるが、実際の内容はもたない COFF セクション。このセクションは .bss 疑似命令または .usect 疑似命令から構成されます。

シンボル：　アドレスまたは値を表す英数字の文字列。

シンボル・テーブル：　COFF オブジェクト・ファイルの一部分。ファイルで定義されて使用されるシンボルについての情報が含まれます。

せ

整合定義式：　式で使用される前に定義されているシンボルまたはアセンブリ時定数のみを含む式。

静的変数：　スコープが 1 つの関数または 1 つのプログラムに制限されている変数の一種。静的変数の値は、その関数またはプログラムが終了しても破棄されず、それらの関数またはプログラムが再び開始すると、前の値が再び使用されます。

セクション：　メモリ・マップ内の連続した空間に入れられるコードまたはデータの再配置可能なブロック。

セクション・プログラム・カウンタ：　SPC を参照してください。

セクション・プログラム・カウンタ（SPC）： セクション内での、現行メモリ・ロケーションを記録するアセンブラの要素。各セクションには、独自の SPC があります。

セクション・ヘッダ： ファイルのセクションに関する情報を含む COFF オブジェクト・ファイルの一部分。各セクションには独自のヘッダがあります。そのヘッダに、そのセクションの開始アドレスやサイズなどの情報が示されています。

絶対リスタ： 絶対アドレスを含むアセンブラ・リスト・ファイルを作成できるデバッグ・ツール。

そ

統合プリプロセッサ： パーサが組み込まれているプリプロセッサで、高速コンパイルが可能です。

ソース・ファイル： C コードまたはアセンブリ言語コードを含んだファイル。コードをコンパイルまたはアセンブルすることによって、オブジェクト・ファイルを作成します。

た

代入文： 変数に値を割り当てる文。

ターゲット・メモリ： 実行可能なオブジェクト・コードがロードされる TMS320C28x ベース・システムの物理メモリ。

て

定数： オペランドとして使われる数値。

データ・メモリ： 外部変数、静的変数、およびシステム・スタックを含むメモリ・セクション。

と

動的メモリ割り当て： いくつかの関数（malloc、calloc、realloc など）によって作成されるメモリ割り当て。これらの関数を使用することで、実行時に変数用のメモリを動的に割り当てることができます。これは大きなメモリ・プール（ヒープ）を宣言し、これらの関数を使用してヒープからメモリを割り当てることにより行われます。

な

名前付きセクション： .sect 疑似命令を使って定義される初期化されたセクション、または .usect 疑似命令を使って定義される初期化されないセクション。

に

入力セクション： 実行可能なモジュールにリンクされるオブジェクト・ファイルに含まれているセクション。

は

バイト： 伝統的に、1 ユニットとして操作対象となる隣接した 8 ビットのシーケンス。ただし、TMS320C28x のバイトは 16 ビットです。

注： TMS320C28x のバイトは 16 ビット

ISO C 定義により、演算子のサイズはオブジェクトの保存に必要なバイト数になります。さらに ISO では、sizeof が char に適用される場合、結果が 1 になると規定しています。TMS320C28x の char が 16 ビットなので（個別にアドレス指定できるようにするために）、1 バイトも 16 ビットです。このため、予期しない結果になることがあります。たとえば sizeof (int) = 1（2 ではない）となります。TMS320C28x のバイトとワードは同じ（16 ビット）です。

バインディング： 出力セクションまたはシンボルに異なるアドレスを指定するプロセス。

パーサ： ソース・ファイルを読み取り、プリプロセッサ機能を実行し、構文チェックし、オプティマイザやコード・ジェネレータの入力として使用できる中間ファイルを生成するソフトウェア・ツール。

ひ

引数ブロック： 他の関数に引数を渡すために使われるローカル・フレームの部分。

ふ

ファイル・ヘッダ： COFF オブジェクト・ファイルの一部分。セクション・ヘッダ、オブジェクト・ファイルのダウンロード先になるシステムのタイプ、シンボル・テーブルのシンボル数、およびシンボル・テーブルの先頭アドレスなどオブジェクト・ファイルの一般的な情報が含まれます。

フィールド： TMS320C28x では、1 ～ 16 ビットの範囲内で長さを任意の値に設定できるソフトウェアで設定可能なデータ型。

符号拡張： 値の符号ビットを使って、値の未使用の MSB を埋め込むプロセス。

符号なし値： 実際の符号にかかわらず、正数として取り扱われる値の一種。

部分リンク： 再度リンク可能なようにファイルをリンクすること。

プラグマ疑似命令： プラグマ疑似命令はプリプロセッサに関数の処理方法を指示します。

プリプロセッサ： マクロ定義、インクルード・ファイル、条件付きコンパイル、およびプリプロセッサ疑似命令を展開するソフトウェア・ツール。

プログラム・メモリ： 実行可能なコードを含むメモリ・セクション。

ブロック： 中括弧でまとめられた一連の宣言または文。

へ

変数： 一連の値のうちのどれかを想定する、ある量を表すシンボル。

ほ

補助エントリ： シンボル・テーブルに含まれる場合があるシンボル用の余分なエントリ。このエントリには、シンボルに関する追加の情報（シンボルがファイル名、セクション名、関数名などのいずれかどうか）が含まれています。

ホール： 出力セクションを構成する入力セクションの間にあるコードもデータも含まない領域。

ま

マクロ： 命令として使用されるユーザ定義ルーチン。

マクロ定義： マクロを構成する名前とコードを定義する、ソース文のブロック。

マクロ展開： マクロ呼び出しと置き換えられ、それ以降アセンブルされるソース文。

マクロ呼び出し： マクロを起動するプロセス。

マップ・ファイル： リンカの作成する出力ファイル。メモリ構成、セクション構成、セクションの割り当て、およびシンボルとシンボルが定義されているアドレスが記述されています。

マクロ・ライブラリ： マクロで構成されているアーカイブ・ライブラリ。ライブラリ内の各ファイルには、マクロが 1 つ含まれている必要があります。ファイル名は定義するマクロ名と同じで、その拡張子は .asm でなければなりません。

み

未構成メモリ： メモリ・マップの一部として定義されず、かつコードまたはデータがロードされないメモリ。

見出し： コンパイラの 1 パスを表すコンパイラ・リストに含まれている情報。この情報によってコンパイラのパスが識別できます。

め

メモリ・マップ： ターゲット・システムのメモリ空間のマップ。複数の機能ブロックに区画分けされています。

も

文字列テーブル： 8 文字より長いシンボル名を格納するマトリックス（8 文字以上のシンボル名はシンボル・テーブルには格納されません。文字列テーブルに格納されます）。シンボルのエントリの名前部分は、文字列テーブルにある文字列の位置を指します。

ら

ラベル： アセンブラ・ソース文の 1 カラム目から始まるシンボルで、その文のアドレスに対応します。ラベルは、1 カラム目から始めることのできる唯一のアセンブラ文です。

ランタイムサポート関数： C 言語には含まれない作業（メモリの割り当て、文字列の変換、文字列の検索など）を実行する ISO 標準関数。

ランタイムサポート・ライブラリ： 他の関数やルーチンだけでなく、ランタイムサポート関数のソースが納められている、ライブラリ・ファイル rts.src。

ランタイム（実行時）環境： システムが稼働する条件。条件には、メモリおよびレジスタの規則、スタックの構成、関数の呼び出し規則、およびシステムの初期化が含まれます。

り

リスト・ファイル： アセンブラの作成する出力ファイル。ソース文、その行番号、およびセクション・プログラム・カウンタ（SPC）への影響が記述されています。

リンカ： オブジェクト・ファイルを結合して、オブジェクト・モジュールを生成するソフトウェア・ツール。生成されたモジュールは、システム・メモリに割り当てられ、デバイスで実行されます。

ろ

ローダ： 実行可能なモジュールをシステム・メモリに格納するデバイス。

ロード時自動初期化モデル： リンカが C コードのリンク時に使用する自動初期化方法。リンカを起動するときに、-cr オプションを指定すると、このモデルが使用されます。RAM モデルを使用すると、実行時ではなくロード時に変数を初期化することができます。

わ

ワード： 単位として格納、アドレス指定、転送、演算が行われるビットまたは文字のシーケンス。ターゲット・メモリ内の 16 ビット・アドレス指定可能なロケーション。

割り当て： リンカが出力セクションの最終的なメモリ・アドレスを計算するプロセス。

索引

記号

>> 記号　2-36
-@ コンパイラ・オプション　2-13
_ 接頭辞　7-21

数字

0
　比較を除去　3-30
2 の累乗との乗算　8-74
3 文字符号
　系列　2-30
64K ブロックを超える far メモリ・ブロックのコピー - オーバーラップ　8-60
64K ブロックを超える far メモリ・ブロックのコピー - 非オーバーラップ　8-59

A

-a リンカ・オプション　4-5
-aa アセンブラ・オプション　2-22
abort 関数　8-43
.abs 拡張子　2-18
abs 関数　8-44
-abs コンパイラ・オプション　2-13
-abs リンカ・オプション　4-5
-ac アセンブラ・オプション　2-22
acos 関数　8-44
-ad アセンブラ・オプション　2-22
add_device 関数　8-12
-al コンパイラ・オプション　2-22
alt.h パス名　2-29
-apd アセンブラ・オプション　2-22
-api アセンブラ・オプション　2-22
-ar リンカ・オプション　4-5
-as シェル・オプション　2-22
ASCII 変換関数　8-48
asctime 関数　8-45, 8-54
asin 関数　8-45
.asm 拡張子　2-7, 2-18
　変更　2-20
asm 文
　C 言語　7-25
　最適化コード内で必要な注意事項　3-10
　説明　6-12
assert 関数　8-46
assert.h ヘッダ　8-19, 8-31
atan 関数　8-47
atan2 関数　8-47
atexit 関数　8-48, 8-56
atof 関数　8-48
atoi 関数　8-48
atol 関数　8-48
-ax シェル・オプション　2-23

B

-b オプション
　コンパイラ　2-13
　リンカ　4-5
boot.obj　4-7, 4-9
bsearch 関数　8-50
.bss 疑似命令　7-23
.bss セクション　4-11, 7-3
　定義　A-1
　変数を事前に初期化するために使用　6-32

C

-c オプション
　コンパイラ　2-13
　シェルとリンカでのオプションの違い　4-4
　リンカ　4-2, 4-3, 4-5, 4-9, 7-4
.C 拡張子　2-7, 2-18
.c 拡張子　2-7
C 言語
　C プログラミング言語　vi

const キーワード　6-14
cregister キーワード　6-15
far キーワード　6-19 ～ 6-21
near キーワード　6-19
アセンブリへの差し込み方法　2-44
整数式の分析　7-37
特性　6-2
プリプロセッサ　6-4
C 言語の実装　6-1 ～ 6-38
C コンパイラ
概要　1-5
定義　A-1
C ソース・ファイル拡張子　2-18
C 入出力
関数とマクロのまとめ　8-33 ～ 8-36
実装　8-9
低レベル・ルーチン　8-9
デバイスの追加方法　8-10 ～ 8-11
ライブラリ　8-8 ～ 8-11
--c ライブラリ作成ユーティリティ・オプション　9-3
C++ ネーム・デマングラ
オプション　10-3
起動方法　10-2
使用例　10-4 ～ 10-6
説明　1-4, 10-1
ソフトウェア開発フローにおける　1-4
C/C++ コードのコンパイル　2-2
オプティマイザを使って　3-2
概要、コマンド、およびオプション　2-2
コンパイルのみ　2-14
C/C++ 割り込みルーチン　7-36
C28x UOUT 命令としての C28XLP OUT　2-17
C28x コードと C2XLP コードのリンク方法　4-13
C28x 用のコード生成　2-18
calloc 関数　7-7, 8-51, 8-66
cassert ヘッダ　8-19, 8-31
.cc 拡張子　2-7
cctype ヘッダ　8-20, 8-31
C_DIR 環境変数　9-2
ceil 関数　8-51
cerrno ヘッダ　8-20
cfloat ヘッダ　8-21
.cinit　7-21
.cinit セクション　4-10, 7-3, 7-40, 7-43
自動初期化での使用　4-9
_c_int00　4-9
c_int00　7-40
ciso646 ヘッダ　8-24
cl2000 コマンド　2-4
clearerr 関数　8-52
climits ヘッダ　8-21
clock 関数　8-52
CLOCKS_PER_SEC マクロ　8-30, 8-52
close 入出力関数　8-13
cmath ヘッダ　8-24, 8-32
CODE_SECTION プラグマ　6-25
COFF　1-3, 1-6, 7-3
定義　A-4
const 型修飾子　6-33
const キーワード　6-14
.const セクション　4-10, 7-3
プログラム・メモリへの割り当て　7-6
変数の初期化に使用　6-33
C_OPTION 環境変数　9-2
cos 関数　8-53
cosh 関数　8-53
.cpp 拡張子　2-7, 2-18
-cr リンカ・オプション　4-2, 4-3, 4-5, 4-9, 7-8
cregister キーワード　6-15
csetjmp ヘッダ　8-24, 8-33
cstdarg ヘッダ　8-25, 8-33
cstddef ヘッダ　8-25
cstdio ヘッダ　8-27, 8-33 ～ 8-36
cstdlib ヘッダ　8-28, 8-36 ～ 8-38
cstring ヘッダ　8-29, 8-38 ～ 8-40
ctime 関数　8-54
ctime ヘッダ　8-29, 8-39
ctype.h ヘッダ　8-20, 8-31
.cxx 拡張子　2-7, 2-18

D

-d オプション　2-13
.data セクション　A-1
DATA_SECTION プラグマ　6-28
_ _DATE_ _ マクロ　2-28
dem2000　10-2
difftime 関数　8-54
div 関数　8-55
div_t 型　8-28
DWARF デバッグ・フォーマット　2-15

E

-e リンカ・オプション　4-5
-ea シェル・オプション　2-20
.ebss セクション　4-11, 7-3
-ec コンパイラ・オプション　2-20
EDOM マクロ　8-20
-eo コンパイラ・オプション　2-20
-eo シェル・オプション　2-20
EOF クリア関数　8-52
EOF 標識のテスト　8-61
EOF マクロ　8-27
EPROM プログラマ　1-3
ERANGE マクロ　8-20
errno.h ヘッダ　8-20
.esysmem セクション　4-11, 7-4
-ex ポストリンク・オプティマイザ・オプション　5-7
exception ヘッダ　8-30
exit 関数　8-43, 8-48, 8-56
exp 関数　8-56

F

-f リンカ・オプション　4-5
-fa コンパイラ・オプション　2-19
fabs 関数　8-57
far キーワード　6-19
far に対応した関数
　まとめの表　8-40
far メモリ解放関数　8-58
far メモリ割り当て関数　8-59
far メモリ割り当てと far メモリ・クリアを行う関数　8-57
far_calloc 関数　8-57
far_free 関数　8-58
far_malloc 関数　8-59
far_memlcpy 関数　8-59
far_memlmove 関数　8-60
far_realloc 関数　8-60
FAST_FUNC_CALL プラグマ　6-29
-fb コンパイラ・オプション　2-21
-fc コンパイラ・オプション　2-19
fclose 関数　8-61
feof 関数　8-61
ferror 関数　8-61
-ff コンパイラ・オプション　2-21
fflush 関数　8-62
fgetc 関数　8-62
fgetpos 関数　8-62
fgets 関数　8-63
FILE 構造体　8-27
_ _FILE_ _ マクロ　2-28
file.h ヘッダ　8-20
FILENAME_MAX マクロ　8-27
float.h ヘッダ　8-21
floor 関数　8-63
fmod 関数　8-64
-fo コンパイラ・オプション　2-19
fopen 関数　8-64
FOPEN_MAX マクロ　8-27
-fp コンパイラ・オプション　2-19
fpos_t データ型　8-27
fprintf 関数　8-64
fputc 関数　8-65
fputs 関数　8-65
-fr コンパイラ・オプション　2-21
fread 関数　8-66
free 関数　8-66
free_memory 関数　8-67
freopen 関数　8-67
frexp 関数　8-68
-fs コンパイラ・オプション　2-21
fscanf 関数　8-68
fseek 関数　8-69
fsetpos 関数　8-69
-ft コンパイラ・オプション　2-21
ftell 関数　8-69
FUNC_EXT_CALLED プラグマ
　-pm オプションと組み合わせて使う　3-8
　説明　6-30
fwrite 関数　8-70

G

-g オプション
　シェル　2-15
　リンカ　4-5
-g コンパイラ・オプション　2-15

getc 関数　8-70
getchar 関数　8-70
getenv 関数　8-71
gets 関数　8-71
.global 疑似命令　7-21, 7-23
gmtime 関数　8-72

H

-h オプション
　デマングラ　10-3
　リンカ　4-5
--h ライブラリ作成ユーティリティ・オプション　9-3
-heap オプション　4-5
-heap リンカ・オプション　7-7, 8-8, 8-51, 8-76, 8-85
Hex 変換ユーティリティ　1-3
　説明　A-2
HUGE_VAL　8-24

I

-i オプション
　コンパイラ　2-29
　説明　2-13
　リンカ　4-5, 4-7
#include
　ファイル
　　検索されるディレクトリの追加　2-13
　　検索パスの指定方法　2-28
　プリプロセッサ疑似命令　2-28
inline キーワード　2-41
_INLINE プリプロセッサ・シンボル　2-41
interrupt キーワード　6-16
INTERRUPT プラグマ　6-31
int_fastN_t 整数型　8-26
int_leastN_t 整数型　8-26
intmax_t 整数型　8-26
INTN_C マクロ　8-26
intN_t 整数型　8-26
intprt_t 整数型　8-26
isalnum 関数　8-72
isalpha 関数　8-72
isascii 関数　8-72
iscntrl 関数　8-72
isdigit 関数　8-72
isgraph 関数　8-72
islower 関数　8-72
ISO C
　K&R C との互換性　6-35
　TMS320C28x は異なる　6-2
　規格
　　概要　1-5
　　紹介　6-1
　言語　8-94, 8-96
iso646.h ヘッダ　8-24
isprint 関数　8-72
ispunct 関数　8-72
isspace 関数　8-72
isupper 関数　8-72
isxdigit 関数　8-72
isxxx 関数　8-20, 8-72

J

jmpbuf 型　8-24

K

-k オプション
　コンパイラ　2-13
　ポストリンク・オプティマイザ　5-8
--k ライブラリ作成ユーティリティ・オプション　9-3
K&R C　vi
　互換性　6-1
　定義　A-2
K&R C との互換性　6-35

L

-l オプション
　ライブラリ作成ユーティリティ　9-2
　リンカ　4-2, 4-4, 4-5, 4-7
labs 関数　8-44
ldexp 関数　8-74
ldiv 関数　8-55
ldiv_t 型　8-28
limits.h ヘッダ　8-21

_ _LINE_ _ マクロ　2-28
lnk2000 コマンド　4-3
localtime 関数　8-54, 8-74, 8-80
log 関数　8-75
log10 関数　8-75
longjmp 関数　8-24, 8-88
lseek 入出力関数　8-14
L_tmpnam マクロ　8-27
ltoa 関数　8-76

M

-m リンカ・オプション　4-6
-ma コンパイラ・オプション　2-16
-ma シェル・オプション　3-11
malloc 関数　7-7, 8-66, 8-76
　確保された空間　7-4
math.h ヘッダ　8-24, 8-32
max_free 関数　8-77
-md コンパイラ・オプション　2-16
-me コンパイラ・オプション　2-16
memchr 関数　8-77
memcmp 関数　8-78
memcpy 関数　8-78
memmove 関数　8-79
memset 関数　8-79
-mf コンパイラ・オプション　2-16
-mi コンパイラ・オプション　2-16
mk2000 コマンド　9-2
mktime 関数　8-80
-ml コンパイラ・オプション　2-16
-mn コンパイラ・オプション　2-17
modf 関数　8-81
-ms コンパイラ・オプション　2-17
-ms シェル・オプション　3-18
-mt コンパイラ・オプション　2-17
-mu コンパイラ・オプション　2-17
-mv コンパイラ・オプション　2-17
-mx コンパイラ・オプション　2-18

N

-n オプション
　コンパイラ　2-14
　リンク　4-6
NDEBUG マクロ　8-19, 8-46
near キーワード　6-19
new ヘッダ　8-30
new_handler 型　8-30
-nf ポストリンク・オプティマイザ・オプション　5-8
.nfo 拡張子　3-5
NULL マクロ　8-25, 8-27

O

-o オプション
　コンパイラ　3-2
　デマングラ　10-3
　リンカ　4-6
.o* 拡張子　2-7, 2-18
.obj 拡張子　2-20
offsetof マクロ　8-25
-oi オプティマイザ・オプション　3-12
-ol シェル・オプション　3-4
-on シェル・オプション　3-5
-op シェル・オプション　3-6
open 入出力関数　8-15

P

-pdel コンパイラ・オプション　2-34
-pden コンパイラ・オプション　2-34
-pdf コンパイラ・オプション　2-34
-pdr コンパイラ・オプション　2-34
-pds コンパイラ・オプション　2-34
-pdse コンパイラ・オプション　2-34
-pdsr コンパイラ・オプション　2-34
-pdsw コンパイラ・オプション　2-34
-pdv コンパイラ・オプション　2-34
-pdw コンパイラ・オプション　2-34
-pe シェル・オプション　6-6
perror 関数　8-81
-pg コンパイラ・オプション　2-30
-pi シェル・オプション　2-41
.pinit セクション　4-10
　とグローバル・コンストラクタ　4-9, 7-40, 7-41, 7-44
　とシステム初期化　7-40

フォーマット　7-44
-pk パーサ・オプション　6-35
-pm シェル・オプション　3-6
-pn コンパイラ・オプション　2-14
pow 関数　8-82
-ppa コンパイラ・オプション　2-30
-ppc コンパイラ・オプション　2-31
-ppd コンパイラ・オプション　2-31
-ppi コンパイラ・オプション　2-31
-ppl コンパイラ・オプション　2-31
-ppo コンパイラ・オプション　2-30
-pr コンパイラ・オプション　6-38
printf 関数　8-82
-priority リンカ・オプション　4-6
--profile: breakpt コンパイラ・オプション　2-15
--profile: power コンパイラ・オプション　2-15
-ps コンパイラ・オプション　6-38
ptrdiff_t 型　6-2
ptrdiff_t データ型　8-25
putc 関数　8-82
putchar 関数　8-83
puts 関数　8-83

Q

-q オプション　2-14
--q ライブラリ作成ユーティリティ・オプション
9-3
qsort 関数　8-83

R

-r リンカ・オプション　4-6
rand 関数　8-84
RAND_MAX マクロ　8-28
read 入出力関数　8-16
realloc 関数　7-7, 8-66, 8-85
register
記憶クラス　6-3
rename 関数　8-86
rename 入出力関数　8-16
rewind 関数　8-86
RTS　7-37
--rts ライブラリ作成ユーティリティ・オプション
9-3
rts.lib
リンク方法　7-40
rts.src　1-3, 8-2

S

-s オプション
コンパイラ　2-14
シェル　2-44
リンカ　4-6
.s* 拡張子　2-7, 2-18
scanf 関数　8-87
.sect 疑似命令　7-35
setbuf 関数　8-87
setjmp 関数　8-88
setjmp マクロ　8-24
setjmp.h ヘッダ　8-24, 8-33
set_new_handler() 関数　8-30
setvbuf 関数　8-89
sin 関数　8-90
sinh 関数　8-90
size_t 型　6-2
size_t データ型　8-25, 8-27
SP レジスタ　7-40
sprintf 関数　8-91
sqrt 関数　8-91
srand 関数　8-84
-ss オプション　2-14
-ss シェル・オプション　3-13
sscanf 関数　8-92
STABS デバッグ・フォーマット　2-15
.stack セクション　4-11, 7-3
-stack リンカ・オプション　4-6
_ _STACK_SIZE 定数　7-5
stdarg.h ヘッダ　8-25, 8-33
stddef.h ヘッダ　8-25
stdexcept ヘッダ　8-30
stdint.h ヘッダ　8-26
stdio.h ヘッダ　8-27, 8-33
関数のまとめ　8-33
stdlib.h ヘッダ　8-28, 8-36
strcat 関数　8-92
strchr 関数　8-93
strcmp 関数　8-94

strcoll 関数　8-94
strcpy 関数　8-95
strcspn 関数　8-95
strerror 関数　8-96
strftime 関数　8-96
string.h ヘッダ　8-29, 8-38
strlen 関数　8-98
strncat 関数　8-98
strncmp 関数　8-99
strncpy 関数　8-100
strpbrk 関数　8-101
strrchr 関数　8-102
strspn 関数　8-102
strstr 関数　8-103
strtod 関数　8-104
strtok 関数　8-105
strtol 関数　8-104
strtoul 関数　8-104
struct_tm　8-29
strxfrm 関数　8-106
STYP_CPY フラグ　4-10
.switch セクション　4-10, 7-3
--symdebug: coff コンパイラ・オプション　2-15
--symdebug: dwarf コンパイラ・オプション　2-15
--symdebug: none コンパイラ・オプション　2-15
--symdebug: skeletal コンパイラ・オプション　2-15
.sysmem セクション　4-11, 7-4
_ _SYSMEM_SIZE
　グローバル・シンボル　7-7
　メモリ管理　8-28

T

tan 関数　8-106
tanh 関数　8-107
.text セクション　4-10
　定義　A-2
_ _TI_COMPILER_VERSION_ _　2-28
_ _TIME_ _ マクロ　2-28
time.h ヘッダ　8-29, 8-39
time_t データ型　8-29
tm 構造体　8-29
TMP 環境変数　2-26, 9-2
tmpfile 関数　8-108
TMP_MAX マクロ　8-27
tmpnam 関数　8-108
_ _TMS320C2000_ _　2-28
TMS320C28x C 言語
　ANSI C との互換性　6-35
　関連文献　vi
_ _TMS320C28XX_ _　2-28
toascii 関数　8-108
tolower 関数　8-109
toupper 関数　8-109
type_info 構造体　8-30
typeinfo ヘッダ　8-30

U

-u オプション
　コンパイラ　2-14
　デマングラ　10-3
　リンカ　4-6
--u ライブラリ作成ユーティリティ・オプション　9-3
uint_fastN_t 符号なし整数型　8-26
uint_leastN_t 符号なし整数型　8-26
uintmax_t 符号なし整数型　8-26
UINTN_C マクロ　8-26
uintN_t 符号なし整数型　8-26
uintprt_t 符号なし整数型　8-26
ungetc 関数　8-109
unlink 入出力関数　8-17

V

-v デマングラ・オプション　10-3
--v ライブラリ作成ユーティリティ・オプション　9-3
-v28 コンパイラ・オプション　2-18
va_arg 関数　8-110
va_end 関数　8-110
va_start 関数　8-110
vfprintf 関数　8-111
volatile　3-10
volatile キーワード　6-14
volatile 参照保護の有効化　2-17
vprintf 関数　8-111
vsprintf 関数　8-112

W

-w リンカ・オプション　4-6
write 入出力関数　8-17

X

-x オプション　4-6

Y

y/x の逆タンジェント関数　8-47

Z

-z オプション
　上書き　4-4
　コンパイラ　2-2, 2-14, 4-3

あ

アーカイバ
　説明　1-3
　定義　A-3
アーカイブ・ライブラリ
　定義　A-3
　リンク方法　4-7
アーク・コサイン関数　8-44
アーク・サイン関数　8-45
アーク・タンジェント関数　8-47
アキュムレータ　7-19
アキュムレータ A の使用
　関数の呼び出しの際の　7-13, 7-14
アセンブラ　1-3
　オプション　2-22
　オプションのまとめ　2-11
　説明　1-3
　定義　A-3
アセンブラの制御　2-22
アセンブリ言語
　C コードを差し込む方法　2-44
　C プログラムに埋め込む方法　6-12
　出力の保存方法　2-13
　モジュール　7-19
　割り込みルーチン　7-35
アセンブリ言語と C のインターフェイス
　asm 文　7-25
　アセンブリ言語モジュール　7-19
　コンパイラ出力の変更方法　7-26
　変数の定義とアクセス　7-23
アセンブリ言語における変数の定義方法　7-23
アセンブリ・ソース・デバッグ　2-15
アセンブリ・リスト・ファイル
　作成方法　2-22

い

一時ファイル作成関数　8-108
一致する文字の検出　8-101, 8-102
一致する文字列の検出　8-103
インターリスト・ユーティリティ　1-7
　オプティマイザを組み合わせて使用する　3-13
　起動方法　2-14
　シェル・プログラムから起動する方法　2-44
　定義　A-3
インライン
　関数の宣言方法　2-41
　定義制御の　2-41
　無効化　2-41
インライン化
　関数展開　2-40
　組み込み演算子　2-40
　自動　3-12
　制約事項　2-43
　保護されない定義制御の　2-41
インライン・アセンブリ言語　7-25
インライン・アセンブリ構造（asm）　7-25

え

エイリアス指定
　説明　3-11
　定義　A-3
エイリアス設定された変数の想定　2-16
エイリアスの明確化
　説明　3-22
エスケープ・シーケンス　6-2, 6-36
エラー
　標識関数　8-52
　プリプロセッサからのメッセージ　2-27

ポインタの組み合わせの処理方法　6-35
報告　8-20
マップ関数　8-81
エラー番号のマップ　8-81
エラー・テスト関数　8-61
エラー・メッセージ
assert マクロ　8-31, 8-46
オプションの処理　2-35
診断メッセージも参照
エントリ時に保存されるレジスタ　7-11
エントリ・ポイント
C/C++ コードの　4-9
_c_int00　4-9
システム・リセット　7-35
定義　A-3
リセット・ベクタ　4-9

お

オーバーフロー
ランタイム・スタック　7-40
大文字と小文字の区別
アセンブリ・ソース内での　2-22
ファイル名拡張子における　2-18
オブジェクト格納関数　8-62
オブジェクト・ファイル
定義　A-4
オブジェクト・モジュール
説明　1-3
オブジェクト・ライブラリ
コードのリンク方法　8-2
定義　A-4
オプション
C++ ネーム・デマングラ・ユーティリティ　10-3
規則　2-5
診断　2-10, 2-34
ファイル指定子　2-20
ファイルの指定　2-7
プリプロセッサ　2-30
オプティマイザ　1-7
オプション
-oi　3-12
オプションのまとめ　2-10, 2-11
シェル・オプションから起動する方法　3-2
定義　A-4
特別な注意事項　3-10, 3-11

か

開発
ツール　1-2
フロー　1-2
外部
宣言　6-36
外部変数　7-8
書き込み関数
fprintf　8-64
fputc　8-65
fputs　8-65
fwrite　8-70
printf　8-82
putc　8-82
putchar　8-83
puts　8-83
sprintf　8-91
ungetc　8-109
vfprintf　8-111
vprintf　8-111
vsprintf　8-112
拡張子
C ソース・ファイル　2-19
C++ ソース・ファイル　2-19
nfo　3-5
アセンブリ言語ソース・ファイル　2-19
オブジェクト・ファイル　2-19
指定方法　2-19
下線を前に付ける　7-21
可変引数関数およびマクロ　8-25, 8-33, 8-110
仮定義　6-36
カレンダ時間関数　8-29, 8-54, 8-80, 8-107
環境情報関数　8-71
環境変数
C28X_OPTION　2-24
C_DIR　2-24, 9-2
C_OPTION　2-24, 9-2
TMP　2-26, 9-2
関数
インライン展開　2-40
定義　A-4
プロトタイプ
要件の緩和方法　6-35
呼び出し　7-14
規則　7-13
スタックの使用方法　7-5
関連文献　vi
緩和 ANSI/ISO モード　6-38

き

キーワード
const 6-14
cregister 6-15
far 6-19
inline 2-41
interrupt 6-16
near 6-19
volatile 6-14
疑似乱数 8-84
起動方法
C++ ネーム・デマングラ 10-2
コンパイル・ステップの一部としてのリンカ 4-3
シェル・プログラム 2-4
独立したステップとしてのリンカ 4-2
ライブラリ作成ユーティリティ 9-2
強度換算の最適化 3-26
極座標のアーク・タンジェント関数 8-47

く

組み込み C++ モード 6-6
組み込み演算子 2-40
組み込み関数
インライン展開演算子 2-40
演算子のテーブル 7-26
組み込み関数の無効化 2-14
グリニッジ標準時 8-72
グレゴリオ暦 8-29
グローバル変数 7-8
確保された空間 7-3
コンストラクタ 4-9
自動初期化 7-41
初期化方法 6-32
定義 A-5
グローバル・コンストラクタ 4-9, 7-40, 7-41
グローバル・デストラクタ 4-9
クロスリファレンス・リスト
アセンブラで作成 2-23
コンパイラで作成 2-37

け

警告メッセージ 2-32
型の異なるポインタ 6-35
検索 8-50
厳密 ANSI/ISO モード 6-38

こ

構造体
メンバ 6-3
構造体のパック 7-9
コード・サイズ最適化の強化レベル 2-17
コード・サイズ最適化の強化 3-18
コード・ジェネレータ
定義 A-5
コサイン関数 8-53
コストに基づいたレジスタ割り当ての最適化 3-21
コマンド行の長さに対する制限事項の回避策 2-13
コマンド・ファイル
定義 A-5
リンカ 4-11
コンストラクタ
グローバル 4-9, 7-40, 7-41
コンパイラ 1-6
オプション
アセンブラ 2-11
オプティマイザ 2-10
規則 2-5
コンパイラ 2-6
出力ファイル 2-8
診断 2-10
シンボリック・デバッグ 2-7
入力ファイルの拡張子 2-7
パーサ 2-9
非推奨 2-23
プロファイル 2-7
マシン固有 2-8
まとめ 2-6
オプティマイザ 3-2
概要 1-5, 2-2
起動方法 2-4
使用頻度の高いオプション 2-13
診断オプション 2-34
診断メッセージ 2-32
制限値 6-38
セクション 4-10
説明 2-1
プリプロセッサ・オプション 2-30

補足ユーザ情報ファイルの生成　2-13
コンパイラ出力の変更方法　7-26

さ

最初の出現の検出
　バイト　8-77
　文字　8-93
サイズよりスピードを優先した最適化　2-16
最適化
　インライン展開　3-24
　エイリアスの明確化　3-22
　強度換算　3-26
　コストに基づいたレジスタ割り当て　3-21
　式の簡略化　3-23
　情報ファイル・オプション　3-5
　ゼロとの比較を除去する方法　3-30
　データ・フロー　3-22
　テール結合　3-28
　のリスト　3-20
　ファイルレベル　3-4
　プログラムレベル　3-6
　誘導変数　3-26
　ループの循環　3-26
　ループ不変コードの移動　3-26
　レジスタ変数　3-26
　レジスタ・ターゲッティング　3-26
　レジスタ・トラッキング　3-26
　レベル　3-2
　レベルの制御方法　3-6
最適化されたコード
　デバッグ　3-16
最適化されたコードのプロファイル方法　3-17
サイン関数　8-90
三角関数　8-24

し

シェル
　オプション
　　パーサ　2-9
時間関数
　asctime　8-45
　clock　8-52
　ctime　8-54
　difftime　8-54
　gmtime　8-72
　localtime　8-74
　mktime　8-80
　strftime　8-96
　time　8-107
　説明　8-30
　まとめの表　8-39
時間書式の %　8-96
時間の書式化関数　8-96
式　6-3
　簡略化　3-23
　説明　6-3
　定義　A-5
式の分析
　整数　7-37
　浮動小数点　7-39
識別子　6-2
指数関数　8-24, 8-56
システムの初期化　7-40
　スタック　7-40
システムの制約
　_ _STACK_SIZE　7-5
　_ _SYSMEM_SIZE　7-7
システム・スタック　7-5
自然対数関数　8-75
事前定義された名前
　_ _TI_COMPILER_VERSION_ _　2-28
事前定義マクロ名　2-27
実行時自動初期化モデル
　初期化　7-8
　定義　A-6
実装が定義する動作　6-2
自動初期化
　実行時の　7-45
　説明　7-8
　定義　A-6
　のタイプ　4-9
　変数の　7-41
　ロード時の　7-46
シフト　6-3
出力
　ファイルの概要　1-6
　ファイル・オプションのまとめ　2-8
詳細時間関数　8-29, 8-45, 8-72, 8-74, 8-80
小数部と指数部の関数　8-68
冗長なロード命令
　除去方法　5-4
使用できる動的なメモリの最大割り当てサイズ　8-77
剰余　7-37
常用対数関数　8-75
初期化

タイプ　4-9
初期化されたセクション　7-3
定義　A-6
メモリに割り当てる方法　4-10
初期化されないセクション　7-3
定義　A-6
メモリに割り当てる方法　4-10
除算
説明　6-3
除算と剰余　7-37
診断識別子
ロー・リスト・ファイルの　2-38
診断メッセージ
assert　8-46
エラー　2-32
警告　2-32
制御方法　2-34
生成方法　2-34
説明　2-32
その他のメッセージ　2-36
致命的エラー　2-32
注釈　2-32
と assert.h/cassert ヘッダ・ファイル　8-19
フォーマット　2-32
抑止方法　2-34
診断メッセージの制御方法　2-34
進捗情報の抑止　2-14
シンボリック
クロスリファレンス　2-23
デバッグ
疑似命令の生成方法　2-15
シンボリック・デバッグ
DWARF 疑似命令　2-15
STABS フォーマットの使用方法　2-15
最小限（デフォルト）　2-15
無効化　2-15
シンボル　2-22
シンボル・テーブル
ラベルの作成方法　2-22

す

スタック　7-5, 7-40
オーバーフロー
ランタイム・スタック　7-40
確保された空間　7-3
スタック管理　7-5
スタック・ポインタ　7-5, 7-40
ストリーム書き込み関数　8-64, 8-91, 8-112
ストリームへの書き込み関数　8-109, 8-111
ストリーム用にバッファを定義し関連付ける関数
8-87

せ

制御レジスタ
アクセス方法
C からの　6-15
制限値
コンパイラ　6-38
整数式の分析　7-37
除算と剰余　7-37
整数除算関数　8-55
生成方法
シンボリック・デバッグ疑似命令　2-15
有効なファイル名の関数　8-108
リンク名　6-34
静的変数　7-8
確保された空間　7-3
説明　6-32
定義　A-6
セクション　7-3
.bss　6-32, 7-3
.cinit　7-3, 7-40, 7-43
.const　7-3
初期化　6-33
.ebss　7-3
.econst　7-3
.esysmem　7-4
.pinit　4-9, 7-40, 7-41, 7-44
.stack　7-3
.switch　7-3
.sysmem　7-4
.text　7-3
コンパイラが作成する　4-10
定義　A-6
メモリを割り当てる方法　4-10
絶対値関数　8-44, 8-57
絶対コンパイラの限界　6-38
絶対リスト
アセンブラでの作成方法　2-22
絶対リスタでの作成方法　2-13
ゼロとの比較
除去方法　3-30
ゼロとの比較を除去する方法　3-30
宣言　6-3
専用レジスタ　7-12

そ

ソース・ファイル
　拡張子　2-19
ソート関数　8-83

た

タンジェント関数　8-106

ち

地方時関数　8-29, 8-54
致命的エラー　2-32
注釈　2-32

て

底が 10 の対数関数　8-75
定数
　C 言語　6-2
　.const セクション　6-33
　定義　A-7
　文字
　　ASCII デフォルト　6-2
　　エスケープ・シーケンス　6-36
　文字列
　　エスケープ・シーケンス　6-36
　　エスケープ・シーケンスの値　6-2
定数の未定義化　2-14
ディレクトリ
　アセンブリおよびクロスリファレンス・リスト・ファイル　2-21
　アセンブリ・ファイル　2-21
　一時ファイル　2-21
　インクルード・ファイルの　2-13, 2-29
　インクルード・ファイルの代替　2-29
　オブジェクト・ファイル　2-21
　指定方法　2-21
　絶対リスト・ファイル　2-21
データ型　6-2, 6-7
データ・フローの最適化　3-22
データ・ブロック書き込み関数　8-70
データ・メモリ　7-2
　定義　A-7
テール結合　3-28
デカルト座標のアーク・タンジェント関数　8-47
デストラクタ　4-9
デバッグ
　最適化されたコード　3-16
デバッグの目的で無効にした最適化を有効にする　2-17
デマングラ、C++ ネーム・デマングラも参照
展開されたマクロとアセンブルされたファイル　2-18

と

動的なメモリ割り当て　7-7
　定義　A-7
動的メモリ管理　8-30
トークン　8-105

な

夏時間調整　8-29

に

入出力
　関数のまとめ　8-33
入出力関数
　close　8-13
　lseek　8-14
　open　8-15
　read　8-16
　rename　8-16
　unlink　8-17
　write　8-17
　バッファ・フラッシュ　8-62
入出力バッファ・フラッシュ関数　8-62
入力ファイル
　拡張子
　　オプションのまとめ　2-7

ね

ネーム・デマングラ
C++ ネーム・デマングラを参照

は

パーサ
オプションのまとめ　2-9
定義　A-8
ハイパボリック算術関数
math.h/cmath ヘッダに定義されている　8-24
コサイン　8-53
サイン　8-90
タンジェント　8-107
パイプライン・コンフリクトと -mv コンパイル・オプションの指定　2-17
配列検索関数　8-50
配列ソート関数　8-83
バッファ
指定関数　8-87
定義および関連付け関数　8-89
汎用ユーティリティ関数
abort　8-43
abs　8-44
atexit　8-48
atof　8-48
atoi　8-48
atol　8-48
bsearch　8-50
calloc　8-51
div　8-55
exit　8-56
free　8-66
labs　8-44
ldiv　8-55
ltoa　8-76
malloc　8-76
qsort　8-83
rand　8-84
realloc　8-85
srand　8-84
strtod　8-104
strtol　8-104
strtoul　8-104
説明　8-28
まとめ　8-36

ひ

ヒープ　7-7
確保された空間　7-4
ヒープ・サイズ　7-7
ヒープ・サイズの変更　8-51, 8-76, 8-85
ヒープ・サイズ変更関数　8-60
引数
アクセス方法　7-17
引数ブロック
説明　7-13
定義　A-8
ビット
アドレッシング　7-9
フィールド　6-3, 6-36, 7-9
標準出力への書き込み関数　8-82, 8-83, 8-111
非ローカル・ジャンプ　8-24, 8-33, 8-88

ふ

ファイル
除去関数　8-86
名前変更関数　8-86
ファイル位置取得関数　8-69
ファイル位置設定関数
fseek 関数　8-69
fsetpos 関数　8-69
ファイル位置標識設定関数
現行のファイル位置標識の取得　8-69
ファイル位置標識にあるオブジェクトの格納　8-62
ファイル位置標識の設定　8-69
ファイルの先頭の位置指定　8-86
ファイル除去関数　8-86
ファイル名
拡張子の使い方　2-19
指定方法　2-18
生成関数　8-108
ファイルレベルの最適化　3-4
ファイル・オープン関数　8-64, 8-67
ファイル・クローズ関数　8-61
フィールドの操作　7-9
不一致文字の検出　8-95
符号付き整数および符号付き小数　8-81
浮動小数点
式の分析　7-39
浮動小数点型　6-9

浮動小数点算術関数
acos　8-44
asin　8-45
atan　8-47
atan2　8-47
ceil　8-51
cos　8-53
cosh　8-53
exp　8-56
fabs　8-57
floor　8-63
fmod　8-64
ldexp　8-74
log　8-75
log10　8-75
math.h/cmath.h ヘッダ･ファイルで定義された　8-24
modf　8-81
pow　8-82
sin　8-90
sinh　8-90
sqrt　8-91
tan　8-106
tanh　8-107
概要　8-32
浮動小数点の剰余　8-64
プラグマ疑似命令
CODE_SECTION　6-25
DATA_SECTION　6-28
FAST_FUNC_CALL　6-29
FUNC_EXT_CALLED　6-30
INTERRUPT　6-31
プリプロセッサ
_INLINE シンボル　2-41
エラー・メッセージ　2-27
オプション　2-30
制御方法　2-27
定義　A-9
定数 name の事前定義方法　2-13
プリプロセッサ疑似命令
C 言語　6-4
トークンを続ける方法　6-36
プリプロセッサ・リスト・ファイル
アセンブリ include ファイル　2-22
アセンブリ依存行　2-22
プログラム終了関数
abort (exit)　8-43
atexit　8-48
exit　8-56
プログラムレベルの最適化
実施　3-6
制御　3-6
プログラム・メモリ　7-2
定義　A-9
プロセッサ時間関数　8-52
ブロック
メモリ割り当て　4-10

平方根関数　8-91
ヘッダ・ファイル　6-4, 8-18
ciso646　8-24
file.h　8-20
iso646.h　8-24
stdint.h　8-26
変換
ASCII への　8-108
C 言語　6-3
大文字小文字　8-109
カレンダ時からグリニッジ標準時への　8-72
カレンダ時から地方時への　8-54, 8-74
時間から文字列への　8-45
地方時からカレンダ時への　8-80
長整数から ASCII への　8-76
文字　8-20, 8-104, 8-106
文字列から数値への　8-48, 8-104
変数
自動初期化　7-41
定義　A-9
ローカル　7-17
変数の初期化方法　6-32
変数の割り当て　7-8

ポインタの組み合わせ
禁止　6-35
保護されない定義制御のインライン展開　2-41
ポストリンク・オプティマイザ
-ex オプション　5-7
-k オプション　5-8
-nf オプション　5-8
最適化を制御する方法　5-7
冗長な DP ロード命令を除去する方法　5-4
制約事項　5-9
ソフトウェア開発フローにおける役割　5-2
補足ユーザ情報ファイル
生成方法　2-13

ま

マクロ
事前定義された名前　2-27
定義　A-9
展開　2-27
マクロの展開と展開されたファイルのアセンブル　2-18
マルチバイト文字　6-2

み

見出しの抑止　2-14

む

無効化
DP の最適化　2-16
高速分岐命令の生成　2-16
シンボリック・デバッグ　2-15
リピート命令　2-16

め

メモリ
データ　7-2
プログラム　7-2
メモリ解放関数　8-66
メモリ管理関数
calloc　8-51
free　8-66
malloc　8-76
realloc　8-85
メモリ内の値をコピーする関数　8-79
メモリ比較　8-78
メモリ・プール　8-76
_ _SYSMEM_SIZE シンボル　7-7
確保された空間　7-4
メモリ割り当て関数　8-76
メモリ割り当てとメモリ・クリアを行う関数　8-51
メモリ・セクションの割り当て　4-10
メモリ・ブロック・コピー　8-78, 8-79
メモリ・マップ
1 つの単一空間として設定　2-17
メモリ・モデル　7-2
構造体のパック　7-9
実行時モデル　7-8
スタック　7-5
セクション　7-3
動的なメモリ割り当て　7-7, 8-30
フィールドの操作　7-9
変数の割り当て方法　7-8
ロード時モデル　7-8

も

文字
定数　6-36
変換関数　8-106
文字列定数　7-9
読み込み関数
単一文字　8-62
複数文字　8-63
文字書き込み関数　8-65, 8-82, 8-83
文字セット　6-2
文字の最後の出現を検出　8-102
文字判別関数
説明　8-20, 8-72
まとめ　8-31
文字列書き込み関数　8-65
文字列関数　8-29, 8-38
文字列コピー関数　8-95, 8-100
文字列定数
エスケープ・シーケンス　6-36
エスケープ・シーケンスの値　6-2
文字列のコピー　8-100
文字列の長さの検出　8-98
文字列の比較　8-94, 8-99
文字列を連結する関数　8-92, 8-98
文字列をトークンに分割する関数　8-105
文字を続けること　6-36

ゆ

有効なファイル名作成関数　8-108
ユーティリティ
概要　1-7

よ

抑止方法
  診断メッセージ 2-34
呼び出された関数 7-15, 7-17
呼び出し先の関数 7-15
呼び出し時に保存されるレジスタ 7-11
読み込み
  ストリーム関数
    標準入力からの 8-87
    文字列 8-68, 8-92
    文字列から配列への 8-66
  文字関数
    単一文字 8-62
    次の文字 8-62, 8-70
    複数文字 8-63
読み込み関数 8-71

ら

ラージ・メモリのコード生成 2-16
ラージ・メモリ・モデルのコード生成 2-16
ライブラリ
  C 入出力 8-8
ライブラリ作成ユーティリティ 1-7, 9-3
  オプション 9-2, 9-3 ～ 9-5
  オプションのオブジェクト・ライブラリ 9-2
  起動 9-2
  説明 1-3
ラベル
  保持する方法 2-22
乱整数関数 8-84
ランタイム環境 7-1 ～ 7-26
  アセンブリ言語と C/C++ 言語間のインターフェイス 7-19
  アセンブリ言語における変数の定義方法 7-23
  インライン・アセンブリ言語 7-25
  関数呼び出し規則 7-13 ～ 7-18
  コンパイラ出力の変更方法 7-26
  システムの初期化 7-40 ～ 7-46
  スタック 7-5
  整数式の分析 7-37
  定義 A-11
  浮動小数点式の解析 7-39
  メモリ・モデル
    構造体のパック 7-9
    セクション 7-3
    動的なメモリ割り当て 7-7
    フィールドの操作 7-9
    変数の初期化方法 7-8
    変数の割り当て方法 7-8
  レジスタ規則 7-11 ～ 7-12
  割り込み処理
    C/C++ コードでの説明 7-35 ～ 7-36
    レジスタの保存方法 6-16
ランタイムサポート 8-2
  関数
    概要 8-1
    説明 8-43 ～ 8-112
    まとめ 8-31 ～ 8-42
  関数、定義 A-10
  マクロ、まとめ 8-31, 8-33, 8-34, 8-35
  ライブラリ 8-2
    説明 1-3, 9-1
    定義 A-11
    ライブラリ作成ユーティリティを使った作成時のオプション 9-3
    を指定したリンク方法 4-2, 4-4, 4-7
ランタイム自動初期化モデル 7-8
ランタイム・スタック 7-40

り

リスト・ファイル
  クロスリファレンスの作成方法 2-23
  定義 A-11
リンカ 1-3
  オプション 4-5
  起動方法
    コンパイラ・オプション 2-14
    コンパイル・ステップの一部として 4-3
    独立したステップとして 4-2
  コマンド・ファイル 4-11
  制御方法 4-7
  定義 A-11
  無効化 4-4
  抑止方法 2-13
リンク方法
  C コード 4-1
  C28x コードと C2XLP コード 4-13
  オブジェクト・ライブラリ 8-2
  コンパイル・ステップの一部として 4-3
  独立したステップとして 4-2
  ランタイムサポート・ライブラリとの 4-7
リンク名
  C++ ネーム・マングリング 6-34
  生成方法 6-34

る

ループ最適化　3-26
ループ循環の最適化　3-26
ループ不変の最適化　3-26

れ

例外処理　8-30
レジスタ
　SP　7-17
　アキュムレータ　7-13
　エントリ時に保存された　7-11
　関数呼び出し時　7-14
　使用規則　7-11
　制御
　　C からのアクセス方法　6-15
　専用　7-19
　変数
　　C 言語　6-11
　呼び出し時に保存された　7-11
　割り込み時の保存方法　6-16
レジスタ規則　7-11
　専用のレジスタ　7-12
レジスタ変数
　最適化　3-26
列挙型定数リスト
　カンマを続ける　6-36

ろ

ローカル変数　7-17
ローダ
　説明　6-32
　定義　A-11
ロード時自動初期化モデル
　定義　A-11
ロード時初期化モデル　7-8
ロー・リスト・ファイル
　-pl オプションを指定した生成方法　2-38
　識別子　2-38

わ

ワイルドカード
　使用　2-18
割り当てで使用できる動的なメモリの合計　8-67
割り込み時のレジスタの保存方法　6-16
割り込み処理
　C/C++ コードの、説明　7-35 〜 7-36
　レジスタの保存方法　6-16
割り込みルーチン　7-36

日本テキサス・インスツルメンツ株式会社

本　　　　　　　　社　〒160-8366　東京都新宿区西新宿６丁目24番１号　西新宿三井ビルディング３階　☎03（4331）2000（番号案内）

西日本ビジネスセンター　〒530-6026　大阪市北区天満橋１丁目８番30号　OAPオフィスタワー26階　☎06（6356）4500（代　　表）
工　　　　　　　　場　大分県・日出町／茨城県・美浦村／静岡県・小山町（センサーズ＆コントロールズ事業部）
研究開発センター　茨城県・つくば市（筑波テクノロジー・センター）／神奈川県・厚木市（厚木テクノロジー・センター）
■お問い合わせ先
プロダクト・インフォメーション・センター（PIC）――――――――――　FAX 0120-81-0036
URL: http: //www.tij.co.jp/pic/

TMS320C28x オプティマイジング （最適化） C/C++ コンパイラ ユーザーズ・マニュアル

第 1 版 2006 年 12 月

発行所 日本テキサス・インスツルメンツ株式会社
〒160-8366
東京都新宿区西新宿 6-24-1（西新宿三井ビルディング）

日本テキサス・インスツルメンツ株式会社

Made in Japan
'06.12
日本テキサス・インスツルメンツ株式会社
SPRUEP4