TEXAS INSTRUMENTS

TMS320C6000
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

2005

Digital Signal Processing Solutions

SPRU187 翻訳版

JAJU019 WAS SPRU419

TMS320C6000
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：SPRU187L　翻訳版
2004年 5月

2005 年 7 月

TEXAS INSTRUMENTS

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

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

最初にお読み下さい

このマニュアルについて

本書は以下のコンパイラ・ツールについて説明したものです。

- ❏ コンパイラ
- ❏ アセンブリ・オプティマイザ
- ❏ スタンドアロン・シミュレータ
- ❏ ライブラリ作成ユーティリティ
- ❏ C++ ネーム・デマングラ

TMS320C6000™ C/C++ コンパイラは、これらの言語に関する国際標準化機構（ISO）準拠の標準 C および C++ コードを受け入れ、TMS320C6x デバイス用のアセンブリ言語を生成します。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 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。

- 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**）、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、入力する情報の種類を示しています。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心に示しています。

  cl6x [*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、任意のパラメータを特定します。任意のパラメータを使用する場合、指定内容はこの括弧内に入力します。括弧そのものは入力する必要がありません。任意のパラメータ付きのコマンドの例を次に示します。

  load6x [*options*] *filename.out*

  load6x コマンドには 2 つのパラメータがあります。最初のパラメータ *options* は任意です。2 番目のパラメータ *filename.out* は必須です。

- 中括弧（{ }）は、それに囲まれているパラメータのど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。

  cl6x -z {**-c** | **-cr**} *filenames* [**-o** *name.out*] **-l** *libraryname*

- TMS320C6200 のコアは C6200 として参照されます。TMS320C6400 のコアは 6400 として参照されます。TMS320C6700 のコアは 6700 として参照されます。TMS320C6000 と C6000 は、C6200、6400、C6700 のいずれかとして参照されます。

当社発行の関連文献

以下の文献は、TMS320C6000、および関連サポート・ツールの解説書であり、当社が発行しています。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）に FAX（0120-81-0036）にてお問い合わせください。PIC へのお問い合わせには、Web サイト（http://www.tij.co.jp/pic）もご利用ください。日本語の文献に関しては、当社の DSP 製品ホームページを参照してください。URL は http://www.tij.co.jp/dsp です。

TMS320C6000 Assembly Language Tools User's Guide（文献番号 SPRU186）は、C6000 世代の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 の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。

TMS320C6000 DSP/BIOS User's Guide（文献番号 SPRU303）は、TMS320C6000™ DSP/BIOS ツールと API 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解析する方法について解説しています。

TMS320C6000 CPU and Instruction Set Reference Guide（文献番号 SPRU189）は、C6000 CPU のアーキテクチャ、命令セット、パイプライン、およびこれらのディジタル・シグナル・プロセッサへの割り込みについて解説しています。

TMS320C6201/C6701 Peripherals Reference Guide（文献番号 SPRU190）は、TMS320C6201 および TMS320C6701 ディジタル・シグナル・プロセッサ上で使用可能な共通ペリフェラルについて解説しています。本書には、内部データ・メモリと内部プログラム・メモリ、外部メモリ・インターフェイス（EMIF）、ホスト・ポート・インターフェイス（HPI）、マルチチャネル・バッファド・シリアル・ポート（McBSP）、ダイレクト・メモリ・アクセス（DMA）、拡張 DMA（EDMA）、拡張バス、クロック・フェーズ・ロックド・ループ（PLL）、およびパワーダウン・モードに関する情報が含まれています。

TMS320C6000 Programmer's Guide（文献番号 SPRU198）は、TMS320C6000™ DSP 用の C およびアセンブリ・コードを最適化する方法について解説しています。本書には、アプリケーション・プログラム例も掲載されています。

TMS320C6000 Technical Brief（文献番号 SPRU197）は、C6000 プラットフォームのディジタル・シグナル・プロセッサ、開発ツール、およびサードパーティ・サポートについて紹介しています。

関連文献

このユーザーズ・マニュアルの参考文献として、次の文献を参考にできます。

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 および Guy L. Steele Jr. との共著、Prentice Hall, Englewood Cliffs（New Jersey）刊行

International Standard ISO 14882 (1998) - Programming Languages - C++（C 標準）

ISO/IEC 14882-1998, 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 9899:1999, International Standard - Programming Languages - C (The C Standard)、国際標準化機構刊行

Programming Embedded Systems in C and C++ ― Michael Barr 著、Andy Oram 編集、O.Reilly & Associates 刊行、ISBN:1565923545（February 1999）

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

The Annotated C++ Reference Manual、Margaret A. Ellis および Bjame Stroustrup 著、Addison-Wesley Publishing Company, Reading（Massachusetts）刊行（1990）

The C Programming Language（第 2 版）― Brian W. Kernighan および Dennis M. Ritchie 著、Prentice-Hall, Englewood Cliffs（New Jersey）刊行（1988）

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

商標

Solaris と SunOS は、Sun Microsystems,Inc. の商標です。

UNIX は X/Open Company Limited から独占的にライセンスされ る、米国その他の国における登録商標です。

Windows と Windows NT は、Microsoft Corporation の登録商標です。

Texas Instruments のロゴ、および Texas Instruments は、Texas Instruments Inc. の登録商標です。Texas Instruments の登録商標には、TI、XDS、Code Composer、Code Composer Studio、TMS320、TMS320C6000、および 320 Hotline On-line が含まれます。

その他のすべてのブランド名または製品名は、それぞれの会社または団体の商標あるいは登録商標です。

目次

1 はじめに 1-1

TMS320C6000 ソフトウェア開発ツール、特に最適化 C/C+ コンパイラの概要を説明します。

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 ISO 標準 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
1.3 Code Composer Studio とコンパイラ 1-8

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を説明します。C/C++ ソース・ファイルのコンパイル、アセンブル、およびリンクを実行するコンパイラの起動方法を説明します。インターリスト機能、オプション、およびコンパイラ・エラーについても説明します。

2.1 コンパイラの概要 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 2-15
2.3.2 ターゲット CPU バージョンの選択方法（-mv オプション） 2-17
2.3.3 シンボリック・デバッグとプロファイル・オプション 2-18
2.3.4 ファイル名の指定方法 2-19
2.3.5 コンパイラ・プログラムによるファイル名の解釈方法の変更（-fa、-fc、-fg、-fl、-fo、および -fp オプション） 2-20
2.3.6 コンパイラ・プログラムによる拡張子の解釈、およびファイル作成時の拡張子の付け方の変更（-ea、-ec、-el、-eo、-ep、および -es オプション） 2-21
2.3.7 ディレクトリの指定方法（-fb、-ff、-fr、-fs、および -ft オプション） 2-22
2.3.8 アセンブラを制御するオプション 2-23
2.3.9 非推奨オプション 2-24
2.4 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 C_C6X_OPTION） 2-25

2.5 プリプロセッサの制御方法 2-26
2.5.1 事前定義マクロ名 2-26
2.5.2 #include ファイル検索パス 2-27
2.5.3 前処理リスト・ファイルの生成方法（-ppo オプション） 2-29
2.5.4 前処理後のコンパイルの続行方法（-ppa オプション） 2-29
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） 2-30
2.5.6 行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） 2-30
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） 2-30
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） 2-30
2.6 診断メッセージの概要 2-31
2.6.1 診断の制御方法 2-33
2.6.2 診断抑止オプションの使用方法 2-34
2.7 その他のメッセージ 2-35
2.8 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） 2-35
2.9 ロー・リスト・ファイルの生成方法（-pl オプション） 2-36
2.10 インライン関数展開の使用方法 2-38
2.10.1 組み込み演算子のインライン展開 2-38
2.10.2 自動インライン展開 2-38
2.10.3 保護されない定義制御のインライン展開 2-39
2.10.4 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 2-40
2.10.5 インライン展開の制約事項 2-42
2.11 割り込み柔軟性オプション（-mi オプション） 2-43
2.12 C6400 コードを C6200/C6700/ 旧世代 C6400 オブジェクト・コードにリンクする方法 2-45
2.13 インターリストの使用方法 2-46

3 コードの最適化 3-1

ソフトウェアによるパイプライン処理、ループの簡略化など C コードを最適化する方法について説明します。また、オプティマイザを使用したときに行われる最適化の種類についても説明します。

3.1 オプティマイザの起動方法 3-2
3.2 ソフトウェア・パイプライン化による最適化 3-4
3.2.1 ソフトウェア・パイプライン化の取り止め（-mu オプション） 3-5
3.2.2 ソフトウェア・パイプライン化情報 3-5
3.2.3 パフォーマンスとコード・サイズを改善するためのプロローグとエピローグの縮小 3-14
3.3 冗長なループ 3-16
3.4 コード・サイズの縮小方法（-ms オプション） 3-17
3.5 ファイル・レベルの最適化の実行（-O3 オプション） 3-18
3.5.1 ファイル・レベルの最適化の制御方法（-oln オプション） 3-18
3.5.2 最適化情報ファイルの作成方法（-onn オプション） 3-19

3.6 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）... 3-20
3.6.1 プログラム・レベルの最適化の制御方法（-opn オプション）... 3-21
3.6.2 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... 3-22
3.7 特定のエイリアス指定技法を使用するかどうかの指定方法 ... 3-25
3.7.1 特定のエイリアス使用時の -ma オプションの使用... 3-25
3.7.2 エイリアス技法を使用しないことを示す -mt オプションの使用 ... 3-26
3.7.3 アセンブリ・オプティマイザでの -mt オプションの使用 ... 3-27
3.8 加法の浮動小数点演算の並べ換えの防止 ... 3-28
3.9 最適化コード内で asm 文を使用する場合の注意 ... 3-28
3.10 自動インライン展開（-oi オプション）... 3-29
3.11 最適化でのインターリスト機能の使用方法 ... 3-30
3.12 最適化されたコードのデバッグ方法とプロファイル方法 ... 3-33
3.12.1 最適化されたコードのデバッグ方法（--symdebug:dwarf、--symdebug:coff、および -O オプション）3-33
3.12.2 最適化されたコードのプロファイル方法... 3-34
3.13 実行できる最適化の種類... 3-35
3.13.1 コストに基づいたレジスタ割り当て... 3-36
3.13.2 エイリアスの明確化... 3-38
3.13.3 分岐の最適化と制御フローの簡略化... 3-38
3.13.4 データ・フローの最適化... 3-41
3.13.5 式の簡略化... 3-41
3.13.6 関数のインライン展開... 3-42
3.13.7 誘導変数と強度換算... 3-43
3.13.8 ループ不変コードの移動... 3-44
3.13.9 ループの循環... 3-44
3.13.10 レジスタ変数... 3-44
3.13.11 レジスタのトラッキング／ターゲッティング... 3-44
3.13.12 ソフトウェア・パイプライン... 3-45

4 アセンブリ・オプティマイザの使用方法 ... 4-1

命令をスケジュールし、レジスタを割り当てるアセンブリ・オプティマイザについて説明します。また、アセンブリ・オプティマイザとともに使用する擬似命令に関する情報など、アセンブリ・オプティマイザ用のコードを作成する方法を説明します。

4.1 パフォーマンスを改善するためのコード開発フロー ... 4-2
4.2 アセンブリ・オプティマイザの概要 ... 4-4
4.3 リニア・アセンブリを記述するために必要な知識 ... 4-4
4.3.1 リニア・アセンブリ・ソース文の形式... 4-7
4.3.2 リニア・アセンブリのレジスタ指定方法... 4-8
4.3.3 リニア・アセンブリの機能ユニット指定方法... 4-10
4.3.4 リニア・アセンブリ・ソース・コメントの使用方法... 4-11
4.3.5 シンボリック・レジスタ名を保持するアセンブリ・ファイル... 4-12
4.4 アセンブリ・オプティマイザ疑似命令 ... 4-13

4.5 アセンブリ・オプティマイザを使用したメモリ・バンク競合の回避方法 4-33
4.5.1 メモリ・バンク競合の防止方法 4-34
4.5.2 メモリ・バンクの競合を回避する内積例 4-37
4.5.3 インデックス・ポインタのメモリ・バンク競合 4-41
4.5.4 メモリ・バンク競合アルゴリズム 4-42
4.6 メモリ・エイリアスの明確化 4-43
4.6.1 アセンブリ・オプティマイザによるメモリ参照の処理方法（デフォルト） 4-43
4.6.2 メモリ参照を処理するための -mt オプションの使用方法 4-43
4.6.3 .no_mdep 疑似命令の使用方法 4-43
4.6.4 .mdep 疑似命令を使用した特定のメモリ依存関係の識別 4-44
4.6.5 メモリ・エイリアス例 4-46

5 C/C++コードのリンク 5-1

別作業でまたはコンパイラの作業の一部としてリンクする方法、およびC コードのリンクに必要な特別な要件に適合する方法を説明します。

5.1 コンパイラを使用してリンカを起動する方法（-z オプション） 5-2
5.1.1 独立したステップでリンカを起動する方法 5-2
5.1.2 コンパイル・ステップの一部でリンカを起動する方法 5-3
5.1.3 リンカを無効にする方法（-c コンパイラ・オプション） 5-4
5.2 リンカ・オプション 5-5
5.3 リンク・プロセスの制御方法 5-8
5.3.1 ランタイム・サポート・ライブラリとのリンク方法 5-8
5.3.2 実行時の初期化 5-9
5.3.3 グローバル・オブジェクト・コンストラクタ 5-10
5.3.4 初期化のタイプの指定方法 5-10
5.3.5 セクションをメモリ内のどこに割り振るかを指定する方法 5-11
5.3.6 リンカ・コマンド・ファイルの例 5-12
5.3.7 関数のサブセクションを使用する方法（-mo コンパイラ・オプション） 5-13

6 スタンドアロン・シミュレータの使用方法 6-1

スタンドアロン・シミュレータを起動する方法を説明し、具体例を示します。

6.1 スタンドアロン・シミュレータの起動方法 6-2
6.2 スタンドアロン・シミュレータのオプション 6-4
6.3 ローダを使用してプログラムに引数を渡す方法 6-6
6.3.1 引数が影響を及ぼすプログラムを判別する方法 6-6
6.3.2 引数を格納するターゲット・メモリの予約方法（--args リンカ・オプション） 6-7
6.4 スタンドアロン・シミュレータのプロファイル機能の使用方法 6-8
6.5 シミュレートするシリコン・リビジョンの選択方法（-rev オプション） 6-9
6.6 スタンドアロン・シミュレータの例 6-10

7 TMS320C6000 C/C++ 言語の実装 7-1

ISO C 仕様に関連した TMS320C6000 C/C++ コンパイラ固有の特性を説明します。

7.1 TMS320C6000 C の特性 7-2
7.1.1 識別子と定数 7-2
7.1.2 データ型 7-3
7.1.3 変換 7-3
7.1.4 式 7-3
7.1.5 宣言 7-4
7.1.6 プリプロセッサ 7-4
7.2 TMS320C6000 C++ の特性 7-5
7.3 データ型 7-6
7.4 キーワード 7-7
7.4.1 const キーワード 7-7
7.4.2 cregister キーワード 7-8
7.4.3 interrupt キーワード 7-10
7.4.4 near キーワードと far キーワード 7-11
7.4.5 restrict キーワード 7-14
7.4.6 volatile キーワード 7-15
7.5 レジスタ変数およびパラメータ 7-16
7.6 asm 文 7-17
7.7 プラグマ疑似命令 7-18
7.7.1 CODE_SECTION プラグマ 7-19
7.7.2 DATA_ALIGN プラグマ 7-20
7.7.3 DATA_MEM_BANK プラグマ 7-20
7.7.4 DATA_SECTION プラグマ 7-22
7.7.5 FUNC_CANNOT_INLINE プラグマ 7-23
7.7.6 FUNC_EXT_CALLED プラグマ 7-23
7.7.7 FUNC_INTERRUPT_THRESHOLD プラグマ 7-24
7.7.8 FUNC_IS_PURE プラグマ 7-25
7.7.9 FUNC_IS_SYSTEM プラグマ 7-25
7.7.10 FUNC_NEVER_RETURNS プラグマ 7-26
7.7.11 FUNC_NO_GLOBAL_ASG プラグマ 7-26
7.7.12 FUNC_NO_IND_ASG プラグマ 7-27
7.7.13 INTERRUPT プラグマ 7-27
7.7.14 MUST_ITERATE プラグマ 7-28
7.7.15 NMI_INTERRUPT プラグマ 7-30
7.7.16 PROB_ITERATE プラグマ 7-30
7.7.17 STRUCT_ALIGN プラグマ 7-31
7.7.18 UNROLL プラグマ 7-32
7.8 リンク名の生成 7-33

7.9 静的変数とグローバル変数の初期化方法 7-34
7.9.1 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 7-34
7.9.2 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 7-35
7.10 ISO C 言語モードの変更 7-36
7.10.1 K&R C との互換性（-pk オプション） 7-36
7.10.2 厳密 ISO モードと緩和 ISO モードの有効化（-ps および -pr オプション） 7-38
7.10.3 組み込み C++ モードの有効化（-pe オプション） 7-38

8 ランタイム（実行時） 8-1

メモリとレジスタの規約、スタック構成、関数呼び出し規約、およびシステムの初期化を説明します。アセンブリ言語を C プログラムにインターフェイスするために必要な情報を説明します。

8.1 メモリ・モデル 8-2
8.1.1 セクション 8-2
8.1.2 C/C++ システム・スタック 8-4
8.1.3 動的なメモリ割り当て 8-5
8.1.4 変数の初期化 8-5
8.1.5 メモリ・モデル 8-6
8.1.6 位置に依存しないデータ 8-7
8.2 オブジェクトの表記 8-8
8.2.1 データ型の記憶域 8-8
8.2.2 ビット・フィールド 8-15
8.2.3 文字列定数 8-16
8.3 レジスタ規則 8-17
8.4 関数の構造と呼び出し規則 8-19
8.4.1 関数の呼び出し方法 8-19
8.4.2 呼び出し先関数の対応方法 8-20
8.4.3 引数とローカル変数へのアクセス方法 8-22
8.5 アセンブリ言語と C および C++ 言語間のインターフェイス 8-23
8.5.1 C/C++ コードでのアセンブリ言語モジュールの使用方法 8-23
8.5.2 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 8-26
8.5.3 位置合わせされていないデータと 64 ビット値の使用方法 8-36
8.5.4 MUST_ITERATE と _nassert を使用して SIMD を使用可能にし、ループについてのコンパイラの知識を拡張する方法8-37
8.5.5 データを位置合わせする方法 8-38
8.5.6 SAT ビットの副次作用 8-42
8.5.7 IRP と AMR 規則 8-42
8.5.8 インライン・アセンブリ言語の使用方法 8-43
8.5.9 アセンブリ言語変数に C/C++ からアクセスする方法 8-44
8.6 割り込み処理 8-46
8.6.1 割り込み時のレジスタの保存方法 8-46
8.6.2 C/C++ 割り込みルーチンの使用方法 8-46
8.6.3 アセンブリ言語割り込みルーチンの使用方法 8-47

8.7 ランタイム・サポート算術ルーチン 8-48
8.8 システムの初期化 8-51
8.8.1 変数の自動初期化 8-52
8.8.2 グローバル・コンストラクタ 8-52
8.8.3 初期化テーブル 8-53
8.8.4 実行時の変数の自動初期化 8-56
8.8.5 ロード時の変数の初期化 8-57

9 ランタイムサポート関数 9-1

C/C++ コンパイラに付属するライブラリとヘッダ・ファイルに加え、マクロ、関数、関数宣言の型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）別にまとめています。非 ISO ランタイムサポート関数をアルファベット順に示します。

9.1 ライブラリ 9-2
9.1.1 コードとオブジェクト・ライブラリとのリンク方法 9-2
9.1.2 ライブラリ関数の修正方法 9-3
9.1.3 さまざまなオプションによるライブラリの作成方法 9-3
9.2 C 入出力関数 9-4
9.2.1 低レベル入出力実装の概要 9-5
9.2.2 C 入出力用デバイスの追加方法 9-14
9.3 ヘッダ・ファイル 9-16
9.3.1 診断メッセージ（assert.h/cassert） 9-17
9.3.2 文字の判別と変換（ctype.h/cctype） 9-17
9.3.3 エラー報告（errno.h/cerrno） 9-18
9.3.4 低レベル入出力関数（file.h） 9-18
9.3.5 高速マクロ／静的インライン関数（gsm.h） 9-18
9.3.6 制限値（float.h/cfloat と limits.h/climits） 9-19
9.3.7 整数型のフォーマット変換（inttypes.h） 9-21
9.3.8 代替表現（iso646.h/ciso646） 9-22
9.3.9 near または far としての関数呼び出し（linkage.h） 9-22
9.3.10 浮動小数点算術（math.h/cmath） 9-22
9.3.11 非ローカル・ジャンプ（setjmp.h/csetjmp） 9-23
9.3.12 可変引数（stdarg.h/cstdarg） 9-23
9.3.13 標準定義（sttdef.h/cstddef） 9-24
9.3.14 整数型（stdint.h） 9-24
9.3.15 入出力関数（stdio.h/cstdio） 9-25
9.3.16 汎用ユーティリティ（stdlib.h/cstdlib） 9-26
9.3.17 文字列関数（string.h/cstring） 9-26
9.3.18 時間関数（time.h/ctime） 9-27
9.3.19 例外処理（exception と stdexcept） 9-28
9.3.20 動的メモリ管理（new） 9-28
9.3.21 実行時の型情報（typeinfo） 9-28
9.4 ランタイムサポート関数とマクロのまとめ 9-29
9.5 ランタイム・サポート関数とマクロの解説 9-41

10 ライブラリ作成ユーティリティ 10-1

コードをコンパイルするときに使用するオプションに対して、ランタイムサポート・ライブラリをカスタマイズして利用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使用すると、ヘッダ・ファイルをディレクトリにインストールしたり、ソース・アーカイブからカスタム・ライブラリを作成することができます。

10.1 標準ランタイムサポート・ライブラリ 10-2
10.2 ライブラリ作成ユーティリティの起動方法 10-3
10.3 ライブラリ作成ユーティリティのオプション 10-4
10.4 オプションのまとめ 10-5

11 C++ ネーム・デマングラ 11-1

C++ ネーム・デマングラについて説明し、その起動方法と使用方法を説明します。

11.1 C++ ネーム・デマングラの起動方法 11-2
11.2 C++ ネーム・デマングラのオプション 11-2
11.3 C++ ネーム・デマングラの使用例 11-3

A 用語集 A-1

本書で使用している用語と略語を定義します。

図

図 1–1　TMS320C6000 ソフトウェア開発のフロー 1-2
図 2–1　C/C++ コンパイラ 2-3
図 3-1　最適化による C/C++ プログラムのコンパイル方法 3-2
図 3-2　ソフトウェア・パイプライン・ループ 3-4
図 4–1　4 バンク・インターリーブド・メモリ 4-33
図 4–2　2 つのメモリ・スペースがある 4 バンク・インターリーブド・メモリ 4-34
図 8–1　char および short のデータ保存形式 8-9
図 8–2　32 ビット・データの保存形式 8-10
図 8–3　40 ビット・データの保存形式 8-11
図 8–4　64 ビット・データの保存形式 8-12
図 8–5　倍精度浮動小数点データの保存形式 8-13
図 8–6　ビッグエンディアン形式とリトルエンディアン形式のビット・フィールド・パッキング 8-15
図 8–7　レジスタ引数の規則 8-20
図 8–8　.cinit セクション内の初期化レコードの形式 8-53
図 8–9　.pinit セクション内の初期化レコードの形式 8-55
図 8–10　実行時の自動初期化 8-56
図 8–11　ロード時の初期化 8-57
図 9–1　入出力関数でのデータ構造の相互作用 9-5
図 9–2　ストリーム・テーブル内の最初の 3 つのストリーム 9-6

表

表 2–1　コンパイラ・オプションのまとめ 2-6
表 2–2　コンパイラの下位互換性オプションのまとめ 2-24
表 2–3　事前定義マクロ名 2-26
表 2–4　ロー・リスト・ファイルの識別子 2-36
表 2–5　ロー・リスト・ファイル診断識別子 2-37
表 3-1　-O3 と組み合わせて使用できるオプション 3-18
表 3-2　-ol オプションのレベルの選択方法 3-18
表 3-3　-on オプションのレベルの選択方法 3-19
表 3-4　-op オプションのレベルの選択方法 3-21
表 3-5　-op オプションを使用する場合の特別な注意事項 3-22
表 4-1　アセンブリ・オプティマイザ疑似命令のまとめ 4-13
表 5–1　コンパイラが作成するセクション 5-11
表 7–1　TMS320C6000 C/C++ のデータ型 7-6
表 7–2　有効な制御レジスタ 7-8
表 8–1　レジスタとメモリ内のデータ表記 8-8
表 8–2　レジスタの用途 8-18
表 8–3　TMS320C6000 C/C++ コンパイラの組み込み関数 8-27
表 8–4　TMS320C64x C/C++ コンパイラの組み込み関数 8-31
表 8–5　TMS320C67x C/C++ コンパイラの組み込み関数 8-35
表 8–6　ランタイムサポート算術関数のまとめ 8-48
表 9–1　整数型の範囲に関する制限値を指定するマクロ（limits.h/climits） 9-19
表 9–2　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/cfloat） 9-20
表 9–3　ランタイム・サポート関数およびマクロのまとめ 9-30
表 10–1　オプションとその機能のまとめ 10-5

例

例 2–1 inline キーワードの使用方法 2-39
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 2-41
例 2–3 差し込み後のアセンブリ言語ファイル 2-47
例 3-1 ソフトウェア・パイプライン情報 3-6
例 3-2 -O2 および -os オプションを指定してコンパイルした 例 2–3 の関数 3-31
例 3-3 例 2–3 の関数を -O2、-os、および -ss オプションを指定してコンパイルした結果 3-32
例 3-4 強度換算、誘導変数除去、レジスタ変数、およびソフトウェア・パイプライン化 3-36
例 3-5 制御フローの簡略化と複写伝播 3-39
例 3-6 データ・フローの最適化と式の簡略化 3-42
例 3-7 関数のインライン展開 3-43
例 3-8 レジスタのトラッキング／ターゲッティング 3-45
例 4-1 内積の計算用のリニア・アセンブリ・コード 4-9
例 4-2 内積の計算用の C コード 4-10
例 4-3 コメントを表示する Lmac 関数コード 4-12
例 4-4 メモ リ・バンク情報を指定するロード命令とストア命令 4-36
例 4-5 内積用の C コード 4-37
例 4-6 内積用のリニア・アセンブリ 4-38
例 4-7 内積ソフトウェア・パイプライン・カーネル 4-38
例 4-8 メモリ・バンクの競合を防止するように展開された内積用のリニア・アセンブリの内積 4-39
例 4-9 内積ソフトウェア・パイプライン・カーネル から展開された内積カーネル 4-40
例 4-10 インデックス付きポインタ用の .mptr の使用方法 4-41
例 4-11 メモリ参照の注釈付け 4-44
例 4-12 .mdep ld1, st1 を使用したソフトウェア・パイプライン 4-45
例 4-13 .mdep st1, ld1 および .mdep ld1, st1 を使用したソフトウェア・パイプライン 4-45
例 5–1 リンカ・コマンド・ファイルの例 5-13
例 6–1 スタンドアロン・シミュレータの見出しの例 6-3
例 6–2 コマンド行から引数を渡す方法 6-6
例 6–3 内積ルーチンのプロファイル 6-8
例 6–4 Clock 関数を使用した C コード 6-10
例 6–5 例 6–4 のコンパイルとリンク後のスタンドアロン・シミュレータの結果 6-10
例 7–1 制御レジスタの定義と使用 7-9
例 7–2 ポインタと restrict 型修飾子との使用 7-14
例 7–3 配列と restrict 型修飾子との使用 7-14
例 7–4 CODE_SECTION プラグマの使用方法 7-19
例 7–5 DATA_MEM_BANK プラグマの使用方法 7-21

例 7–6　DATA_SECTION プラグマの使用方法 7-22
例 8–1　アセンブリ言語関数を C/C++ から呼び出す方法 8-25
例 8–2　_lo および _hi 組み込み関数の使用方法 8-36
例 8–3　_lo および _hi 組み込み関数を long long 関数と一緒に使用する方法 8-37
例 8–4　構造体の配列 8-40
例 8–5　クラスの配列 8-40
例 8–6　C から アセンブリ言語変数にアクセスする方法 8-44
例 8–7　C からアセンブリ言語定数にアクセスする方法 8-45
例 8–8　AMR と SAT の処理 8-47
例 8–9　初期化テーブル 8-54
例 11–1　名前のマングリング 11-3
例 11–2　C++ ネーム・デマングラ実行後の結果 11-5

注

ファイル名拡張子の大文字と小文字の区別 2-19
ソース・ファイル用のデフォルトの拡張子は想定されません 2-19
不等号括弧によるパス情報の指定 2-29
関数をインライン展開するとコード・サイズが大幅に増えます 2-38
RTS ライブラリ・ファイルは -mi オプションでは作成されていない 2-44
-mi オプションを使用する特殊な場合 2-44
コード・サイズを制御するために最適化レベルを下げないでください 3-3
-On オプションはアセンブリ・オプティマイザに適用されます 3-4
ソフトウェア・パイプライン化はコード・サイズを大幅に増加させる可能性がある 3-5
ソフトウェア・パイプライン情報の詳細 3-5
冗長ループの取り止め 3-17
コード・サイズ最適化の抑止または最適化レベルの引き下げ 3-17
-ms オプションは -ms0 オプションと等価です 3-17
コード・サイズを制御するために最適化レベルを引き下げない 3-18
-pm および -k オプションを指定してのファイルのコンパイル 3-20
-O3 最適化とインライン展開 3-29
インライン展開とコード・サイズ 3-29
パフォーマンスとコード・サイズに与える影響 3-30
シンボリック・デバッグ・オプションはパフォーマンスとコード・サイズに影響を与えます 3-33
プロファイル・ポイント 3-34
スケジュールされたアセンブリ・コードをソースとして使用しないでください 4-6
レジスタ A4 と A5 の予約 4-29
メモリ依存関係の例外 4-43
メモリ依存関係／バンク競合 4-46
リンカにおいて引数を処理する順序 5-3
_c_int00 シンボル 5-9
アセンブリ・コード内でグローバル変数を定義する方法 7-12
asm 文で C/C++ 環境を損なわないでください 7-17
リンカはメモリ・マップを定義する 8-2
プログラム・メモリではコードだけを使用する 8-3
スタック・オーバーフロー 8-4
SP の意味構造 8-25
スタックの割り当て 8-26
C とアセンブリ言語の組み込み関数命令 8-26
_nassert の C++ 構文 8-38
プログラム・レベルの最適化による位置合わせ 8-41

asm 文の使用方法 8-43
変数の初期化方法..... 8-52
C 入出力バッファの障害 9-4
固有の関数名を使用してください..... 9-14
ユーザ固有の clock 関数の記述 9-28
ユーザ固有の clock 関数を記述してください..... 9-52
minit の実行後、以前に割り当てられたオブジェクトは使用できなくなります 9-78
time 関数はターゲット・システムに固有です 9-106

第 1 章

はじめに

TMS320C6000 は、オプティマイジング（最適化）C/C++ コンパイラ、アセンブリ・オプティマイザ、アセンブラ、リンカ、および各種ユーティリティなど、一連のソフトウェア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。アセンブリ・オプティマイザについては、第 4 章で説明します。アセンブラとリンカの詳細については、TMS320C6000 Assembly Language Tools User's Guide を参照してください。

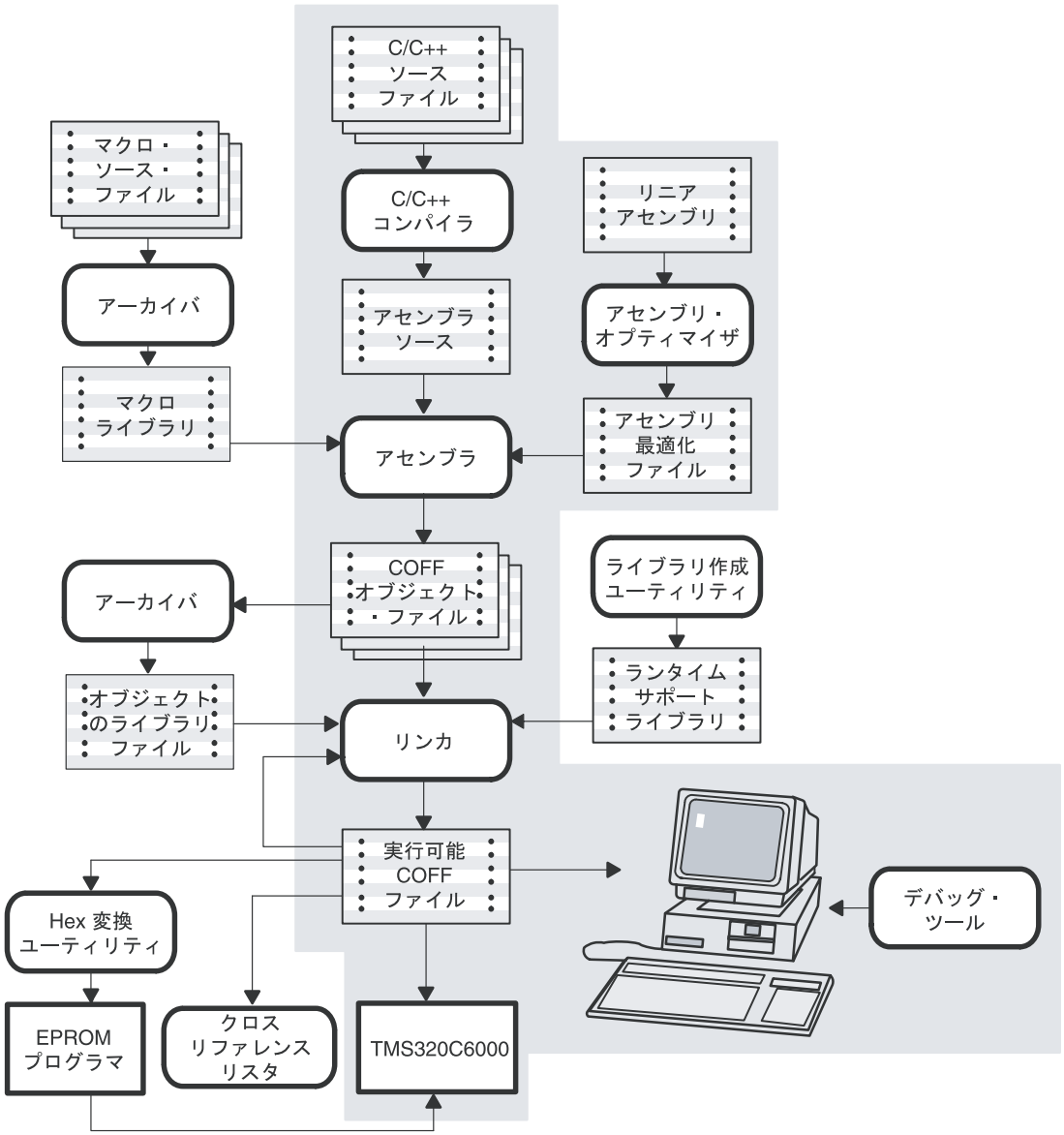
項目 **ページ**

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
1.3	Code Composer Studio とコンパイラ	1-8

1.1 ソフトウェア開発ツールの概要

図 1–1 は、C6000 ソフトウェア開発のフローを示しています。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パスです。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. TMS320C6000 ソフトウェア開発のフロー



以下のリストは、図 1–1 で示されているツールについて説明しています。

- **アセンブリ・オプティマイザ**を使用すると、パイプライン構造やレジスタの割り当てを意識せずにリニア・アセンブリ・コードを書くことができます。また、レジスタ割り当てが済んでいないアセンブリ・コードや未スケジュールのアセンブリ・コードを受け入れます。アセンブリ・オプティマイザはレジスタの割り当てやループ最適化を行い、リニア・アセンブリ・コードをソフトウェア・パイプラインを利用する高度のパラレル・アセンブリ・コードに変えます。アセンブリ・オプティマイザの起動方法、リニア・アセンプリ・コード（.sa ファイル）の書き方、機能ユニットの設定方法、アセンブリ・オプティマイザの疑似命令の使用方法については、第 4 章「アセンブリ・オプティマイザの使用方法」を参照してください。

- **C/C++ コンパイラ**は、C/C++ ソース・コードを受け入れ、C6000 アセンブリ言語ソース・コードを生成します。**コンパイラ**、**オプティマイザ**、および**インターリスト機能**は、コンパイラの一部です。

  - コンパイラを使用すると、ソース・モジュールのコンパイル、アセンブル、およびリンクを 1 ステップで実行することができます。入力ファイルが .sa 拡張子をもつ場合、コンパイラ・プログラムはアセンブリ・オプティマイザを起動します。
  - オプティマイザは、コードを変更して C プログラムの効率を高めます。
  - インターリスト機能は、C/C++ ソース文をアセンブリ言語の出力に差し込みます。

  コンパイラ・プログラムを使用した C/C++ コンパイラ、オプティマイザ、およびインターリスト機能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**は、アセンブリ言語のソース・ファイル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、TMS320C6000 Assembly Language Tools User’s Guide を参照してください。

- **リンカ**は、複数のオブジェクト・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、リンカは実行可能モジュールを作成する際に再配置を行い、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。リンカの起動方法については、第 5 章「C/C++コードのリンク」を参照してください。リンカの詳細は、TMS320C6000 Assembly Language Tools User’s Guide を参照してください。

- **アーカイバ**を使用すると、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を作成するときに大 変便利なツールです。アーカイバの使用方法については、TMS320C6000 Assembly Language Tools User’s Guide を参照してください。

- **ライブラリ作成ユーティリティ**を使用して、カスタマイズした独自のランタイムサポート・ライブラリを作成できます（第 10 章「ライブラリ作成ユーティリティ」を参照）。C および C++ 用の標準ランタイムサポート・ライブラリ関数は、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のオブジェクト・コードは、リトルエンディアン・モード用かビッグエンディアン・モード用の次のような標準ライブラリへコンパイルすることができます。

  - リトルエンディアン C および C++ コードでは、rts6200.lib、rts6400.lib、および rts6700.lib
  - ビッグエンディアン C および C++ コードでは、rts6200e.lib、rts6400e.lib、および rts6700e.lib

  ランタイムサポート・ライブラリには、C6000 コンパイラによってサポートされている ISO 標準ランタイムサポート関数、コンパイラユーティリティ関数、浮動小数点算術関数、および C 入出力関数が入っています。第 8 章「ランタイム環境」を参照してください。

- **Hex 変換ユーティリティ**は、COFF オブジェクト・ファイルを TI-Tagged、ASCII-hex、Intel、Motorola-S、Tektronix のいずれかのオブジェクト・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、TMS320C6000 Assembly Language Tools User's Guide を参照してください。

- **クロスリファレンス・リスタ**は、オブジェクト・ファイルからクロスリファレンス・リストを生成します。生成されたリストには、シンボル、シンボルの定義、およびリンク先ソース・ファイル内のシンボル参照リストなどが表示されています。クロスリファレンス・ユーティリティの使用方法については、TMS320C6000 Assembly Language Tools User's Guide を参照してください。

- この開発プロセスの主な目的は、**TMS320C6000** デバイスで実行できるモジュールを生成することです。いずれかのデバッグ・ツールを使用すれば、生成したコードに改善や修正を加えることができます。使用できるデバッグ・ツールを次に示します。

  - 命令の正確さおよびクロックの正確さを検証するソフトウェア・シミュレータ
  - XDS エミュレータ

- 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。

  これらのデバッグ・ツールについては、TMS320C6000 Code Composer Studio Tutorial および Code Composer Studio 入門マニュアルを参照してください。

1.2 C/C++ コンパイラの概要

C6000 C/C++ コンパイラは、標準 ISO C プログラムを C6000 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 ISO 標準

次の特徴は ISO 標準に関するものです。

❏ **ISO 標準 C**

C6000 C/C++ コンパイラは ISO の規格により定義され、カーニハンとリッチーの The C Programming Language（K&R）第 2 版に記述された ISO C 規格に完全準拠しています。ISO 標準 C には C 言語の拡張機能が含まれています。これらの拡張機能により、移植性の増強と機能の強化を実現しています。

❏ **ISO 標準 C++**

C6000 C/C++ コンパイラは ISO C++ の規格により定義され、Ellis および Stroustrup の共著 The Annotated C++ Reference Manual（ARM）に記述された C++ をサポートしています。また、コンパイラは組み込み C++ もサポートしています。

サポートされていない C++ 機能の詳細は、7.2 節「TMS320C6000 C++ の特性」（7-5 ページ）を参照してください。

❏ **ISO 標準ランタイム・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。すべてのライブラリ関数は、ISO C/C++ ライブラリ規格に準拠しています。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。信号用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。ライブラリには、言語サポート用の構成要素に加えて ISO C サブセットも含まれています。詳細は、第 8 章「ランタイム環境」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簡単に検査できるアセンブリ言語ソース・ファイルが生成されるので、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。また、COFF はソースレベル・デバッグもサポートします。

- **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 に収納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用して、EPROM プログラマ・データ・ファイルに変換できます。詳細は、TMS320C6000 Assembly Language Tools User's Guide 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・プログラム**

  本コンパイラ・ツールにはコンパイラ・プログラムが含まれています。これを使用すると、プログラムのコンパイル、アセンブリ最適化、アセンブル、およびリンクを 1 ステップで実行できます。詳細は、2.1 節「コンパイラの概要」（2-2 ページ）を参照してください。

- **柔軟性のある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 関数を記述することができます。詳細は、第 8 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、高速コンパイルが可能です。また、前処理を独立して行ったり、前処理リストを生成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」（2-26 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。また、C6000 に固有の最適化では、C6000 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」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・インターリスト機能**

  本コンパイラ・ツールには、オリジナル 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ユーティリティが含まれています。このユーティリティにより、C/C++ 文ごとに生成されたアセンブリ・コードを調べることができます。詳細は、2.13 節「インターリストの使用方法」（2-46 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**

  ライブラリ作成ユーティリティ（mk6x）を使用すると、ソースからユーザ独自のオブジェクト・ライブラリを作成し、任意の組み合わせのランタイム・モデルやターゲット CPU に使用することができます。詳細は、第 10 章「ライブラリ作成ユーティリティ」を参照してください。

- **スタンドアロン・シミュレータ**

  スタンドアロン・シミュレータ（load6x）は、実行可能な COFF.out ファイルをロードし、実行します。C 入出力ライブラリと同時に使用する場合は、スタンドアロン・シミュレータは、スクリーンへの標準出力を含めてすべての C 入出力関数をサポートします。詳細は、第 6 章「スタンドアロン・シミュレータの使用方法」を参照してください。

- **C++ ネーム・デマングラ**

  C++ ネーム・デマングラ（dem6x）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。詳細は、第 11 章「C++ ネーム・デマングラ」を参照してください。

1.3 Code Composer Studio とコンパイラ

Code Composer Studio は、コード生成ツールを使用するグラフィカル・インターフェイスを装備しています。

Code Composer Studio プロジェクトは、ターゲット・プログラムまたはライブラリの作成に必要な情報を管理します。プロジェクトは以下の情報を記録します。

- ソース・コードおよびオブジェクト・ライブラリのファイル名
- コンパイラ、アセンブラ、およびリンカ・オプション
- インクルード・ファイルの依存性

Code Composer Studio のプロジェクトを作成すると、プログラムのコンパイル、アセンブル、およびリンクを実行する適切なコード生成ツールが起動します。

Code Composer Studio の「Build Options」ダイアログで、コンパイラ、アセンブラ、およびリンカのオプションを指定することができます。このダイアログでは、大部分のコマンド行オプションが表示されます。表示されないオプションは、ダイアログの上部に表示される編集可能なテキスト・ボックスに直接入力して指定できます。

このマニュアルでは、コマンド行インターフェイスからコード生成ツールを使用する方法について説明しています。Code Composer Studio の使用方法については、Code Composer Studio 入門マニュアルを参照してください。Code Composer Studio 内のコード生成ツール・オプションの設定の詳細は、「Code Generation Tools Help」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 TMS320C6x が実行可能な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・コードのコンパイル、アセンブル、リンクを実行する必要があります。コンパイラ cl6x を使用すると、これらすべてのステップを一度に実行できます。

本章では、cl6x を使用した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・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 コンパイラの概要 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 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 C_C6X_OPTION） 2-25
2.5 プリプロセッサの制御方法 2-26
2.6 診断メッセージの概要 2-31
2.7 その他のメッセージ 2-35
2.8 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） 2-35
2.9 ロー・リスト・ファイルの生成方法（-pl オプション） 2-36
2.10 インライン関数展開の使用方法 2-38
2.11 割り込み柔軟性オプション（-mi オプション） 2-43
2.12 C6400 コードを C6200/C6700/ 旧世代 C6400 オブジェクト・コードにリンクする方法 2-45
2.13 インターリストの使用方法 2-46

2.1 コンパイラの概要

コンパイラ（cl6x）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、次のステップを 1 つ以上のソース・モジュールで実行します。

- **コンパイラ**は、パーサとオプティマイザから成り、C/C++ のソース・コードを取り込んで、C6x アセンブリ言語ソース・コードを生成します。

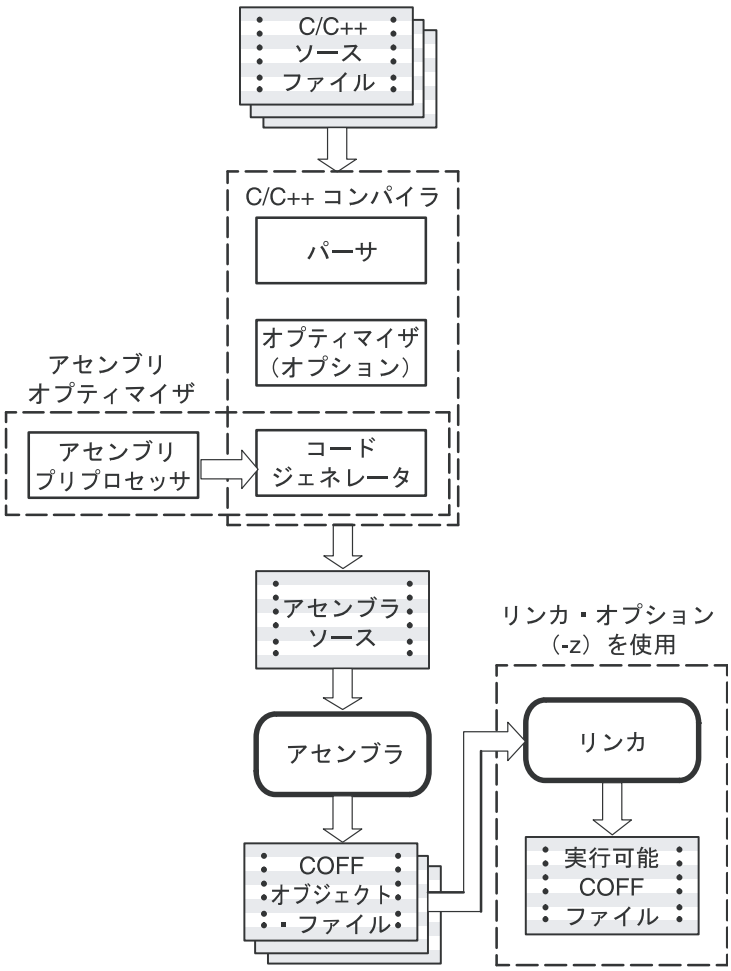
  C ファイルと C++ ファイルを 1 つのコマンドでコンパイルできます。コンパイラは、ファイル名拡張子の規則を使用して異なるファイル・タイプを区別します。詳細は、2.3.4 項「ファイル名の指定方法」（2-19 ページ）を参照してください。

- **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・ファイルを結合して、実行可能オブジェクト・モジュールを生成します。リンク・ステップはオプションのため、多数のモジュールを個別にコンパイルしておき、後でリンクすることも可能です。独立したステップでファイルをリンクする方法については、第 5 章「C/C++コードのリンク」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・ステップを実行しません。-z コンパイラ・オプションを使用して、リンカを起動できます。図 2–1 は、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）を示したものです。

図 2–1.　C/C++ コンパイラ



アセンブラとリンカの詳細は、TMS320C6000 Assembly Language Tools User's Guide 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。

cl6x [*options*] [*filenames*] [**-z** [*link_options*] [*object files*]]

cl6x	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。各オプションは、表 2–1 に掲載されています。
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、リニア・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	リンカの起動オプションです。リンカの起動方法については、第 5 章「C/C++コードのリンク」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。
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。

cl6x の引数には、次の 3 つのタイプがあります。

- ❏ コンパイラ・オプション
- ❏ リンカ・オプション
- ❏ ファイル

-z リンカ・オプションは、リンクが実行されることを示す符号です。-z を使用する場合、すべてのコンパイラ・オプションは -z の前に指定し、他のすべてのリンカ・オプションは -z の後に指定する必要があります。

ソース・コードのファイル名は、-z の前に指定する必要があります。追加オブジェクト・ファイルのファイル名は、-z の後に指定できます。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指定できます。

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、find.sa という名前の第 4 のファイルのアセンブリ最適化を行い、myprogram.out という名前の実行可能プログラムを生成するためにリンクを実行する場合は、次のように入力します。

```
cl6x -q symtab.c file.c seek.asm find.sa -z -llnk.cmd -lrts6200.lib -o myprogram.out
```

cl6x は各ソース・ファイルを検出すると、C/C++ ファイル名とアセンブリ言語ファイル名は角括弧（[]）で囲み、リニア・アセンブリ・ファイルは中括弧（{}）で囲んで出力します。この例のコマンドを入力すると、次のメッセージが表示されます。

```
[symtab.c]
[file.c]
[seek.asm]
{find.sa}
<Linking>
```

2.3 オプションによるコンパイラの動作の変更

オプションの指定により、コンパイラとコンパイラが実行するプログラムの両方を制御することができます。この節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。また、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も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ハイフンが 1 つまたは 2 つ付きます。
- ❏ オプションは大文字と小文字を区別します。
- ❏ オプションは単独の文字または連続した複数の文字です。
- ❏ 個々のオプションを続けて指定することができません。
- ❏ 必須パラメータをもつオプションの場合は、オプションとパラメータを明確に関連付けるために、パラメータの前に等号を使用する必要があります。たとえば、定数を未定義にするオプションは -U=*name* のように指定できます。推奨できませんが、-U *name* または -U*name* のように、オプションとパラメータを空白で区切り、あるいは空白で区切らずに指定することができます。
- ❏ 任意のパラメータをもつオプションの場合は、オプションとパラメータを明確に関連付けるために、パラメータの前に等号を使用する必要があります。たとえば、最大量の最適化に指定するオプションは -O=3 のように指定できます。推奨できませんが、-O3 のようにオプションのすぐ後にパラメータを指定することができます。オプションとオプション・パラメータの間に空白を入れることはできません。つまり、-O 3 のように指定することはできません。
- ❏ 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 C6X_C_OPTION 環境変数を使用します。これらの環境変数の詳細は、2.4 節「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 C_C6X_OPTION）」（2-25 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）についてまとめたものです。表内には各オプションの詳しい説明が載っているページが記載されています。必要に応じて参照してください。

オプションのまとめをオンラインで参照したい場合は、コマンド行でパラメータを指定せずに **cl6x** と入力してください。

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。-@ を複数指定することも可能です。	2-15
-c	リンクを使用不可にします（-z の無効化）。	2-15 5-4
-D*name*[=*def*]	*name* を事前に定義します。	2-15
-h	ヘルプを表示します。	2-16
-I*directory*	#include 検索パスを定義します。	2-16, 2-28
-k	アセンブリ言語（.asm）ファイルを保存します。	2-16
-n	コンパイルまたはアセンブリ最適化だけを実行します。	2-16
-q	進捗メッセージの出力を抑止します（静的実行）。	2-16
-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 をアセンブリ・ソース文に差し込みま す。	2-17
-ss	C のソースをアセンブリ文に差し込みます。	2-17, 3-30
-U*name*	*name* を未定義にします。	2-17
--verbose	見出しと進捗情報を表示します。	--
-z	リンクの実行を有効にします。	2-17

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作成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 と等価です）。	2-18
--profile:breakpt	ブレークポイント・ベースのプロファイリングを有効にします。	2-18
--symdebug:coff	代替 STABS デバッグ・フォーマットを使用するシンボリック・デバッグを有効にします。	2-18, 3-33
--symdebug:dwarf	DWARF デバッグ・フォーマットを使用するシンボリック・デバッグを有効にします（-g と等価です）。	2-18
--symdebug:none	すべてのシンボリック・デバッグを抑止します。	2-18
--symdebug:skeletal	最適化を阻害しない最小シンボリック・デバッグを有効にします（デフォルトの動作です）。	2-18

（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。	2-21
-ec[.]*extension*	C ソース・ファイルのデフォルトの拡張子を設定します。	2-21
-el[.]*extension*	リニア・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。	2-21
-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。	2-21
-ep[.]*extension*	C++ ソース・ファイルのデフォルトの拡張子を設定します。	2-21
-es[.]*extension*	リスト・ファイルのデフォルトの拡張子を設定します。	2-21

表 2–1. コンパイラ・オプションのまとめ（続き）

（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・ファイルとして認識します。デフォルトでは、コンパイラとアセンブラは .asm ファイルをアセンブリ・ソース・ファイルとして扱います。	2-20
-fc*filename*	拡張子に関係なく *filename* を C ソース・ファイルとして認識します。デフォルトでは、コンパイラは .c ファイルを C ソース・ファイルとして扱 います。	2-20
-fg	C 拡張子の付いたすべてのソース・ファイルを C++ ソース・ファイルとして処理します。	2-20
-fl*filename*	拡張子に関係なく *filename* をリニア・アセンブリ・ソース・ファイルとして認識します。デフォルトでは、コンパイラとアセンブラは .sa ファイルをリニア・アセンブリ・ソース・ファイルとして扱います。	2-20
-fo*filename*	拡張子に関係なく *filename* をオブジェクト・コード・ファイルとして認識します。デフォルトでは、コンパイラとリンカは、.obj ファイルをオブジェクト・コード・ファイルとして扱います。	2-20
-fp*filename*	拡張子に関係なく *filename* を C++ ファイルとして認識します。デフォルトでは、コンパイラは .C、.cpp、.cc および .cxx ファイルを C++ ファイルとして扱います。	2-20

（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。	2-22
-ff*directory*	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・ファイルのディレクトリを指定します。	2-22
-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。	2-22
-fs*directory*	アセンブリ・ファイルのディレクトリを指定します。	2-22
-ft*directory*	一時ファイルのディレクトリを指定します。	2-22

表 2–1. コンパイラ・オプションのまとめ（続き）

（f） マシン固有のオプション

オプション	機能	ページ
--consultant	コンパイラのコンサルタント・アドバイスを生成します。	2-15
-ma	特定のエイリアス技法が使用されることを指定します。	3-25
-mb	バージョン 4.0 のツールまたは C6200/C6700 オブジェクト・コードの配列位置合わせ制限と互換性があるように C6400 コードをコンパイルします。	2-45
-mc	加法の浮動小数点演算の並べ換えを防止します。	3-28
-me	ビッグエンディアン形式でオブジェクト・コードを作成します。	2-16
-speculate_loads*n*	アドレス範囲がバインドされたロードの見込み実行を可能にします。	3-14
-mi*n*	割り込みしきい値を定義します。	2-43
-ml*n*	near および far の前提事項を、4 つのレベル（-ml0、-ml1、-ml2、-ml3）で変更します。	2-16, 7-11
-mo	関数のサブセクションを有効にします。	5-13
-mr*n*	ランタイムサポート関数への呼び出しを、near（-mr0）または far（-mr1）にします。	7-12
-ms*n*	コード・サイズを 4 つのレベル（-ms0、-ms1、-ms2、-ms3）で制御します。	3-17
-mt	コンパイラが、エイリアス指定およびループについて特定の仮定をすることを認めます。	3-26, 4-43
-mu	ソフトウェア・パイプラインを取り止めます。	3-5
-mv*n*	ターゲット CPU バージョンを選択します。	2-17
-mw	冗長ソフトウェア・パイプライン・レポートを生成します。	3-5

表 2–1. コンパイラ・オプションのまとめ（続き）

（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。	7-38
-pi	定義制御の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）。	2-40
-pk	K&R 互換性を許容します。	7-36
-pl	ロー・リスト・ファイルを生成します。	2-36
-pm	ソース・ファイルを結合してプログラム・レベルの最適化を実行します。	3-20
-pr	緩和モードを可能にします。厳密な ISO 違反を無視します。	7-38
-ps	厳密な ISO モードを有効にします（K&R C ではなく、C/C++ に対して）。	7-38
-px	クロスリファレンス・リスト・ファイルを生成します。	2-35
-rtti	ランタイム型情報（RTTI）を有効にします。	7-5

（h）前処理を制御するパーサのオプション

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。	2-29
-ppc	前処理だけを実行します。名前が入力と同じで拡張子が .pp のファイルに、前処理された出力（コメントを保持）を書き込みます。	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。名前が入力と同じで拡張子が .pp のファイルに、行制御情報（#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。名前が入力と同じで拡張子が .pp のファイルに、前処理された出力を書き込みます。	2-29

表 2–1. コンパイラ・オプションのまとめ（続き）

(i) 診断を制御するパーサのオプション

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）。	2-33
-pden	診断の識別子を、そのテキストと一緒に表示します。	2-33
-pdf	診断情報ファイルを生成します。	2-33
-pdr	注釈（軽い警告）を発行します。	2-33
-pds *num*	*num* により識別される診断を抑止します。	2-33
-pdse *num*	*num* により識別される診断をエラーとして分類します。	2-33
-pdsr *num*	*num* により識別される診断を注釈として分類します。	2-33
-pdsw *num*	*num* により識別される診断を警告として分類します。	2-33
-pdv	行の折り返し付きでオリジナル・ソースを表示する冗長診断を提供します。	2-34
-pdw	診断の警告メッセージを抑止します（エラー・メッセージは発行されます）。	2-34

表 2–1. コンパイラ・オプションのまとめ（続き）

(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	レジスタの使用状況を最適化します。	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 -O	-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。	3-3
-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。	3-3
-oi*size*	自動インライン展開サイズを設定します（-O3 のみ）。*size* を指定しない場合、デフォルトの 1 が適用されます。	3-29
-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、オプティマイザに通知します。	3-18
-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、オプティマイザに通知します。	3-18
-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、オプティマイザに通知します。-ol0 および -ol1 オプションを取り消します（デフォルト）。	3-18
-on0	最適化情報ファイルの生成を抑止します。	3-19
-on1	最適化情報ファイルを生成します。	3-19
-on2	詳細な最適化情報ファイルを生成します。	3-19
-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。	3-21
-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。	3-21
-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）。	3-21
-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。	3-21
-os	オプティマイザの注釈をアセンブリ文に差し込みます。	3-30

マシン固有の -ma、-mh*n*、-mi*n*、-ms*n*、および -mt オプション（表 2–1 (f) を参照）も最適化に影響します。

表 2–1. コンパイラ・オプションのまとめ（続き）

（k）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	絶対リストを生成します。	2-23
-ac	アセンブリ・ソース・ファイル内で大文字と小文字を区別しません。	2-23
-ad*name*	*name* シンボルを設定します。	2-23
-ahc*filename*	アセンブリ・モジュールの先頭に、指定されたファイルをコピーします。	2-23
-ahi*filename*	アセンブリ・モジュールの先頭に、指定されたファイルをインクルードします。	2-23
-al	アセンブリ・リスト・ファイルを生成します。	2-23
-apd	前処理を実行し、アセンブリの依存関係だけを含むリストを表示します。	2-23
-api	前処理を実行し、#include ファイルだけを含むリストを表示します。	2-23
-as	シンボル・テーブルにラベルを格納します。	2-24
-au*name*	事前定義された定数 *name* を未定義にします。	2-24
-ax	クロスリファレンス・ファイルを生成します。	2-24

表 2–1. コンパイラ・オプションのまとめ（続き）

(l) リンカを制御するオプション

オプション	機能	ページ
-a	実行可能な絶対出力を生成します。	5-5
-abs	絶対リスト・ファイルを生成します。	5-5
-ar	再配置可能で実行可能な出力を生成します。	5-5
-b	シンボリック・デバッグ情報のマージを抑止します。	5-5
-c	実行時に変数を自動初期化します。	5-5, 8-51
-cr	ロード時に変数を初期化します。	5-5, 8-51
-e *global_symbol*	エントリ・ポイントを定義します。	5-5
-f *fill_value*	埋め込み値を定義します。	5-5
-g *global_symbol*	*global_symbol* のグローバルを維持します（-h の無効化）。	5-5
-h	グローバル・シンボルを静的にします。	5-5
-heap *size*	ヒープ・サイズ（バイト数）を設定します。	5-6
-l *directory*	ライブラリ検索パスを定義します。	5-6
-j	条件付きリンクを抑止します。	5-6
-l *libraryname*	ライブラリまたはコマンド・ファイル名を指定します。	5-6
-m *filename*	マップ・ファイルを指定します。	5-6
-o *name.out*	出力ファイルを指定します。	5-6
-q	進捗メッセージの出力を抑止します（静的実行）。	5-6
-priority	そのシンボルの定義を含む最初のライブラリによって各未解決参照を解決します。	5-6
-r	再配置可能で使用不能な出力モジュールを生成します。	5-6
-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。	5-6
-stack *size*	スタック・サイズ（バイト数）を設定します。	5-7
-U *symbol*	未解決の外部シンボルを作成します。	5-7
-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。	5-7
-x	ライブラリの再読み取りを強制します。	5-7

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* コマンド行にファイル内容を付加します。このオプションは、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、または C スタイル・コメントに対する制限事項の回避策として使用できます。コメントを挿入する場合は、コマンド・ファイルの行頭に # か ; を入力します。また、/* および */ で区切ってもコメントを挿入できます。

-@ オプションを複数回使用して、複数のファイルを指定することができます。たとえば次の式は、file3 はソースとしてコンパイルされ、file1 と file2 は -@ ファイルであることを示しています。

```
cl6x -@ file1 -@ file2 file3
```

-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 または C6X_C_OPTION 環境変数で -z を指定しているが、リンクをしたくないといった場合に便利です。詳細は、5.1.3 項「リンカを無効にする方法（-c コンパイラ・オプション）」（5-4 ページ）を参照してください。

--consultant コンパイラ・コンサルタント・アドバイス・ツールを使用して、コンパイル時のループ情報を生成します。コンパイラ・コンサルタント・アドバイス・ツールの詳細は、「TMS320C6000 Code Composer Studio Online Help 」を参照してください。

-D*name[=def]* プリプロセッサの定数 *name* を事前に定義します。これは、各 C ソース・ファイルの先頭に #define *name def* を挿入する場合と等価です。オプションの [=*def*] を省略すると、*name* は 1 に設定されます。引用符で囲まれた文字列を定義し、引用符を保持したい場合は、以下のいずれかを実行します。

- ❏ Windows の場合、-Dname="¥"string def¥"" を使用します。たとえば、`-Dcar="¥"sedan"¥"` と入力します。
- ❏ UNIX の場合、use -Dname='"string def"' を使用します。たとえば、`-Dcar='"sedan"'` と入力します。
- ❏ Code Composer Studio の場合、ファイルに定義を入力して、-@ オプションを使用してそのファイルを組み込みます。

-h ヘルプを表示します。

-I*directory*
（大文字の i）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。各パス名の先頭に **-I** オプションを付ける必要があります。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）」（2-28 ページ）を参照してください。

-k
コンパイラまたはアセンブリ・オプティマイザからのアセンブリ言語出力を保存します。通常は、アセンブリが終了すると、コンパイラは出力アセンブリ言語ファイルを削除します。

-me
ビッグエンディアン形式でコードを作成します。デフォルトでは、リトルエンディアン・コードが作成されます。

-ml*n*
ラージメモリ・モデル・コードを、4 つのレベル（-ml0、-ml1、-ml2、および -ml3）で生成します。

- **-ml0** はデフォルトでは、集合データ（構造体と配列）を far に解釈します。
- **-ml1** はデフォルトでは、すべての関数呼び出しを far に解釈します。
- **-ml2** はデフォルトでは、すべての集合データと呼び出しを far に解釈します。
- **-ml3** はデフォルトでは、すべてのデータと呼び出しを far に解釈します。

レベルを指定しないと、すべてのデータと関数は near にデフォルト解釈されます。near データはデータ・ページ・ポインタを使用してアクセスすると効率が上昇し、near 呼び出しは PC 相対分岐を使用して実行すると効率が上昇します。
static および extern データが多すぎて .bss セクションの先頭から 15 ビット分スケールされたオフセット内に収まらない場合、または呼び出された関数が呼び出し側から ± 1 M ワード以上離れている場合、これらのオプションを使用してください。これらの状態が発生すると、リンカがエラー・メッセージを発行します。詳細は、7.4.4 項「near キーワードと far キーワード」（7-11 ページ）および 8.1.5 項「メモリ・モデル」（8-6 ページ）を参照してください。

-mv *num*
ターゲット CPU のバージョンを選択します（-mv オプションの詳細は、2-17 ページを参照）。

-mw
冗長ソフトウェア・パイプライン・レポートを生成します。

-n
コンパイルまたはアセンブリ最適化だけを実行します。指定されたソース・ファイルのコンパイルまたはアセンブリ最適化は実行されますが、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。

-q
すべてのツールから得られる見出しや進捗情報の出力を抑止します。ソース・ファイル名とエラー・メッセージだけが出力されます。

-s インターリスト機能を起動します。この機能は、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・ソースに差し込みます。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 オプションは、パフォーマンスまたはコード・サイズにマイナスの影響を与える場合があります。

-ss インターリスト機能を起動します。この機能は、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。差し込まれた C 文は、順序が正しくないように見える場合があります。-os と -ss オプションを結合すると、インターリスト機能をオプティマイザで使用できます。詳細は、2.13 節「インターリストの使用方法」（2-46 ページ）を参照してください。-ss オプションは、パフォーマンスとコード・サイズにマイナスの影響を与える場合があります。

-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 指定されたオブジェクト・ファイルに対して、リンカを実行します。-z オプションとそのパラメータは、コマンド行で他のオプションの一番後ろに入力します。-z の後ろにある引数は、すべてリンカに渡されます。詳細は、5.1 節「コンパイラを使用してリンカを起動する方法（-z オプション）」（5-2 ページ）を参照してください。

2.3.2 ターゲット CPU バージョンの選択方法（-mv オプション）

TMS320C6000 製品番号の最後の 4 桁を使用して、ターゲット CPU バージョンを選択します。この選択により、ターゲット固有の命令や位置合わせの使用が制御できます（たとえば -mv6701 や -mv6412）。あるいは、その製品のファミリーを指定することもできます（たとえば -mv6400 や -mv6700）。このオプションを使用しない場合、コンパイラは C6200 製品のコードを生成します。-mv オプションが指定されない場合、生成されたコードはすべての C6000 製品で実行されます。しかし、コンパイラはターゲット固有の命令も位置合わせも利用しません。

2.3.3 シンボリック・デバッグとプロファイル・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。多くの最適化はデバッガを混乱させるため、-g オプションは多くのコード・ジェネレータ最適化を抑止します。-g オプションを -o オプションと一緒に使用すると、デバッグと両立可能な最適化の量を最大にできます（3.12 節「最適化されたコードのデバッグ方法とプロファイル方法」（3-33 ページ）を参照）。

DWARF デバッグ・フォーマットの詳細は、DWARF Debugging Information Format Specification, 1992-1993, UNIX International, Inc. を参照してください。

--profile:breakpt
ブレークポイント・ベースのプロファイラを使用するときに、誤った動作を引き起こす最適化を抑止します。

-symdebug:coff
代替 STABS デバッグ・フォーマットを使用するシンボリック・デバッグを有効にします。これは、DWARF フォーマットを読み取らない古いデバッガまたはカスタム・ツールによるデバッグを可能にするために必要です。

-symdebug:none
すべてのシンボリック・デバッグ出力を抑止します。デバッグとほとんどのパフォーマンス解析機能を無効にするため、このオプションの使用はお勧めしません。

--symdebug:skeletal
最適化を妨げることなく、可能な限り多くのデバッグ情報を生成します。通常、この情報はグローバル・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反映します。

非推奨シンボリック・デバッグ・オプションのリストについては、2.3.9 項（2-24 ページ）を参照してください。

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 ソース・ファイル、C++ ソース・ファイル、アセンブリ・ソース・ファイル、リニア・アセンブリ・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
.c	C ソース
.C	オペレーティング・システムによる
.cpp、.cxx、.cc	C++ ソース
.sa	リニア・アセンブリ
.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

注：ファイル名拡張子の大文字と小文字の区別
ファイル名拡張子の大文字と小文字の区別は、オペレーティング・システムによって決まります。オペレーティング・システムが大文字と小文字を区別しない場合、.C 拡張子は C ファイルして解釈されます。オペレーティング・システムが大文字と小文字を区別する場合、.C 拡張子は C++ ファイルして解釈されます。

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・ファイルの最適化とアセンブルを 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「コンパイラ・プログラムによるファイル名の解釈方法の変更（-fa、-fc、-fg、-fl、-fo、および -fp オプション）」（2-20 ページ）を参照してください。コンパイラがアセンブリ・ソース・ファイルとオブジェクト・ファイルの拡張子を解釈する方法、およびファイルを作成するときの拡張子の付け方を変更する方法については、2.3.7 項「ディレクトリの指定方法（-fb、-ff、-fr、-fs、および -ft オプション）」（2-22 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、ディレクトリ内のすべてのファイルを拡張子 .cpp を使用してコンパイルする場合は、次のように入力します。

```
cl6x *.cpp
```

注：ソース・ファイル用のデフォルトの拡張子は想定されません
コマンド行でファイル名として *example* を指定する場合、コンパイラはファイル名全体を *example.c* ではなく *example* とみなします。拡張子を含まないファイルにデフォルトの拡張子は追加されません。

2.3.5 コンパイラ・プログラムによるファイル名の解釈方法の変更（-fa、-fc、-fg、-fl、-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、-fl、-fo、および -fp オプションを使用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は、任意でスペースを入れることができます。指定するファイルのタイプに応じて、次のうち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
-fl*filename*	リニア・アセンブリ・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、シェルにファイル・タイプを正しく解釈させることができます。

```
cl6x -fc file.s -fa assy
```

-fa、-fc、-fl、および -fo オプションでは、ワイルドカードによる指定はできません。

-fg オプションを使用すると、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。ファイル名拡張子規則の詳細は、2.3.4 項（2-19 ページ）を参照してください。

2.3.6 コンパイラ・プログラムによる拡張子の解釈、およびファイル作成時の拡張子の付け方の変更（-ea、-ec、-el、-eo、-ep、および -es 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、-ea、-el、および -eo オプションは処理対象のファイル名の前に入力しなければなりません。どちら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。拡張子の長さは 9 文字まで有効です。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
-el[.] *new extension*	リニア・アセンブリ・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
-es[.] *new extension*	リスト・ファイルのデフォルトの拡張子を設定します。

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。

```
cl6x -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の間のスペースは入れても入れなくても構いません。上記の例は、次のように入力することもできます。

```
cl6x -earrr -eoo fit.rrr
```

2.3.7 ディレクトリの指定方法（-fb、-ff、-fr、-fs、および -ft オプション）

デフォルトでは、コンパイラ・プログラムは、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したい場合は、次のオプションを指定します。

-fb*directory* 絶対リスト・ファイルを格納する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。

```
cl6x -fb d:\abso_list
```

-ff*directory* アセンブリ・リスト・ファイルおよびクロスリファレンス・リスト・ファイルを格納する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。

```
cl6x -ff d:\listing
```

-fr*directory* オブジェクト・ファイルを格納する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。

```
cl6x -fr d:\object
```

-fs*directory* アセンブリ・ファイルを格納する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。

```
cl6x -fs d:\assembly
```

-ft*directory* 一時中間ファイルのディレクトリを指定します。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。

```
cl6x -ft c:\temp
```

2.3.8 アセンブラを制御するオプション

以下は、コンパイラで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 アセンブリ言語ソース・ファイル内で大文字と小文字を区別しません。たとえば、-c はシンボル ABC と abc を同じものにします。このオプションを使用しない場合、大文字と小文字を区別します（これはデフォルトです）。

-ad*name* アセンブラの定数 *name* を事前に定義します。オプションの [=*def*] を省略すると、*name* は 1 に設定されます。引用符で囲まれた文字列を定義し、引用符を保持したい場合は、以下のいずれかを実行します。

- Windows の場合、-ad*name*="¥"string def¥"" を使用します。たとえば、`-adcar="¥"sedan¥""` と入力します。
- UNIX の場合、-ad*name*="'string def"' を使用します。たとえば、`-adcar='"sedan"'` と入力します。
- Code Composer Studio の場合、ファイルに定義を入力して、-@ オプションを使用してそのファイルを組み込みます。

-ahc*filename* -hc 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・モジュールに指定されたファイルをコピーするようにアセンブラに指示します。ファイルは、ソース・ファイル文の前に挿入されます。コピーされたファイルは、アセンブリ・リスト・ファイルに記録されます。

-ahi*filename* -hi 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・モジュールに指定されたファイルをインクルードするようにアセンブラに指示します。ファイルはソース・ファイル文の前にインクルードされます。インクルード・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、名前がソース・ファイルと同じであり .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 疑似命令でインクルードされるファイルのリストを書き込みます。このリストは、名前がソース・ファイルと同じであり .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* を未定義にします。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・ファイルにシンボルのクロスリファレンスを生成します。

シンボルの詳細は、TMS320C6000 Assembly Language Tools User's Guide を参照してください。

2.3.9 非推奨オプション

いくつかのコンパイラ・オプションの使用が推奨されません。コンパイラはこれらのオプションを受け入れますが、使用が推奨されません。将来のツールのリリースでは、これらのオプションはサポートされなくなります。表 2–2 に、非推奨オプションとこれに置き換わるオプションを示します。

表 2–2. コンパイラの下位互換性オプションのまとめ

古いオプション	機能	新しいオプション
-gp	最適化されたコードを関数レベルでプロファイルできるようにします。	-g
-gt	代替 STABS デバッグ・フォーマットを使用するシンボリック・デバッグを有効にします。	--symdebug:coff
-gw	DWARF デバッグ・フォーマットを使用するシンボリック・デバッグを有効にします。	--symdebug:dwarf または -g

さらに、STABS デバッグ・フォーマット（--symdebug:dwarf または -g）を使用するシンボリック・デバッグにより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可能にする --symdebug:profile_coff オプションが追加されました。

2.4 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 C_C6X_OPTION）

C_OPTION または C6X_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 C6X_C_OPTION または C_OPTION 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。

これらの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 C6X_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。
C6X_C_OPTION が検出されない場合、シェルは C_OPTION 環境変数を読み込んで処理します。

次の表は、C_OPTION 環境変数を設定する方法を示しています。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
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方法で指定され、同じ意味をもちます。たとえば、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）、Windows 用にリンクしたい（-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=-q -s -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。

```
cl6x      *c                  ; compiles and links
cl6x      -c *.c              ; only compiles
cl6x      *.c -z lnk.cmd      ; compiles and links using a
                              ; command file
cl6x      -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、5.2 節「リンカ・オプション」（5-5 ページ）を参照してください。

2.5 プリプロセッサの制御方法

この節では、パーサの一部である C6000 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。C6000 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。プリプロセッサは次のものを処理します。

- ❏ 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、自明のエラー・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、診断メッセージとともに表示されます。

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
_TMS320C6X	常に定義されます。
_TMS320C6200	ターゲットが C6200 である場合に定義されます。
_TMS320C6400	ターゲットが C6400 である場合に定義されます。
_TMS320C6700	ターゲットが C6700 である場合に定義されます。
_BIG_ENDIAN	ビッグ・エンディアン・モードが選択される（-me オプションが使用される）場合に定義されます。選択されない場合は未定義です。
_INLINE	最適化が使用される場合は 1 に展開します。使用されない場合は未定義です。最適化が使用されるかどうかにかかわらず、-pi が使用されるときは常に未定義です。
_LARGE_MODEL	ラージ・モデル・モードが選択される（-ml オプションが使用される）場合に定義されます。選択されない場合は未定義です。
_LARGE_MODEL_OPTION	-mln により指定されるラージ・モデルに設定されます。それ以外の場合は未定義です。
_LITTLE_ENDIAN	リトル・エンディアン・モードが選択される（-me オプションが使用されない）場合に定義されます。選択されない場合は未定義です。

† ISO 規格による指定

表 2–3. 事前定義マクロ名（続き）

マクロ名	説明
_SMALL_MODEL	スモール・モデル・モードが選択される（-ml オプションが使用されない）場合に定義されます。選択されない場合は未定義です。
__LINE__†	カレント行番号に展開します。
__FILE__†	カレント・ソース・ファイル名に展開します。
__DATE__†	*mmm dd yyyy* 形式でコンパイル日に展開します。
__TIME__†	*hh:mm:ss* 形式でコンパイル時刻に展開します。
__TI_COMPILER_VERSION__	メジャー・バージョン番号で構成する 3 桁（またはそれ以上）の整数および 2 桁のマイナー・バージョン番号に定義されます。数値に少数点は含まれません。たとえば、バージョン 5.14 は 514 と表現されます。
__STDC__†	コンパイラが ISO C 規格に準拠することを指示するために定義されます。ISO C 準拠の例外については、7.1 節「TMS320C6000 C の特性」（7-2 ページ）を参照してください。

† ISO 規格による指定

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) C6X_C_DIR または 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。

  4） -I オプションで指定されたディレクトリ

  5） C6X_C_DIR で設定し、次に 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）」（2-28 ページ）を参照してください。C_DIR 環境変数については、コード生成ツール CD-ROM 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* ...]

コンパイラの呼び出しごとに指定する -I オプションの数に制限はありません。-I オプションごとに 1 つのディレクトリを指定します。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。

UNIX /6xtools/files/alt.h

Windows c:¥6xtools¥files¥alt.h

次の表はコンパイラの起動方法を示しています。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
UNIX	**cl6x** -I=6xtools/files source.c
Windows	**cl6x** -Ic:¥6xtools¥files source.c

注 : 不等号括弧によるパス情報の指定

不等号括弧でパス情報を指定する場合、コンパイラは -I オプションと C_DIR または C6X_C_DIR 環境変数で指定したパス情報に関連する情報を適用します。

たとえば、次のコマンドを使用して C_DIR をセットアップする場合、

```
C_DIR "/usr/include;/usr/ucb"; export C_DIR
```

または、次のコマンドを使用してコンパイラを起動する場合、

```
cl6x -I=/usr/include file.c
```

かつ、file.c に次の行が含まれる場合、

```
#include <sys/proc.h>
```

この結果、インクルード・ファイルは次のパスに配置されます。

```
/usr/include/sys/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。このファイルには、.pp という拡張子が付きます。コンパイラの前処理機能は、ソース・ファイル上で次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 拡張子をもつ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 オプションを使用します。

2.5.6 行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 オプションを使用してください。-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 拡張子をもつファイルに、行制御情報（#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依存関係の情報のリストを書き込みます。このリストは、名前がソース・ファイルと同じであり .pp 拡張子をもつ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、名前がソース・ファイルと同じであり .pp 拡張子をもつ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.c***", line** *n*:*diagnostic severity*:*diagnostic message*

"*file.c***"**	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・ファイルの後のソース・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・コードが生成されません。

- **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・オプションを使用してください。

診断は、次の例のような形式で標準エラー出力に書き込まれます。

```
"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するには、-pdv コンパイラ・オプションを使用してください。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、ソース行自体が（^ 文字によって指定される位置とともに）メッセージの後に続きます。複数の診断が 1 つのソース行に適用される場合、診断ごと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。

長いメッセージは、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・メッセージの後に掲載されます。

```
"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、または代入演算子関数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、エラーが何を参照しているかの判別が難しくなります。

2.6.1 診断の制御方法

C/C++ 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。次のオプションが診断を制御します。

-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」（2-31 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 により識別され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 オプションを使用してください。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 により識別される診断をエラーとして分類します。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 オプションを使用してください。その後、-pdse *num* を使用して診断をエラーとして分類し直します。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 により識別される診断を注釈として分類します。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 オプションを使用してください。その後、-pdsr *num* を使用して診断を注釈として分類し直します。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 により識別される診断を警告として分類します。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 オプションを使用してください。その後、-pdsw *num* を使用して診断を警告として分類し直します。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.c", line 9:warning:statement is unreachable
"err.c", line 12:warning:statement is unreachable
```

他の case 文も適用してしまう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 オプションを使用すると、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.c]
"err.c", line 9:warning #111-D:statement is unreachable
"err.c", 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使用すると、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。

2.7 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 >> 記号が付いています。

2.8 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 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 とは別のものです。-ax はコンパイラ・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・ファイルの名前は、ソース・ファイルと同じ名前に拡張子 *.crl* を付けたものです。

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 の値	意味
D	定義
d	宣言（定義ではない）
M	修正
A	アドレス
U	使用
C	変更（1 つの操作で使用され、修正される）
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

2.9 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。前処理リスト・ファイル（-ppo、-ppc、-ppl、および -ppf プリプロセッサ・オプションを使用して生成された）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で、ロー・リスト・ファイルによりオリジナルのソース行と前処理された出力とを比較できます。ロー・リスト・ファイルの名前は、対応するソース・ファイルと同じ名前に拡張子 *.rl* を付けたものです。

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
- ❏ 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、表 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 文節）
L	ソース位置の変更。これは次の形式で指定されます。 L *line number filename key* ここで、*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

診断識別子	定義
E	エラー
F	致命的エラー
R	注釈
W	警告

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

ファイルの最後に続く診断は、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」（2-31 ページ）を参照してください。

2.10 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、次の理由から短い関数には便利です。

- ❏ 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
- ❏ 保護された inline キーワードを使用した定義制御インライン展開

注 : 関数をインライン展開するとコード・サイズが大幅に増えます
関数をインライン展開すると、特に多数の場所で呼び出される関数をインライン展開する場合にコード・サイズが増加します。関数のインライン展開は、少数の場所でのみ呼び出される関数と小さい関数の場合に最適です。コード・サイズが大きすぎると思われる場合は、3.4 節「コード・サイズの縮小方法（-ms オプション）」（3-17 ページ）を参照してください。

2.10.1 組み込み演算子のインライン展開

C6000 には多数の組み込み演算子があります。それらはすべて、コンパイラにより自動的にインライン展開されます。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、8.5.2 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」（8-26 ページ）を参照してください。

2.10.2 自動インライン展開

-O3 オプションを使用して 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、3.10 節「自動インライン展開（-oi オプション）」（3-29 ページ）を参照してください。

2.10.3 保護されない定義制御のインライン展開

inline キーワードは、標準の呼び出しプロシージャ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 を使用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 キーワードの使用法を示しています。ここでは、関数呼び出しの代わりに呼び出された関数のコードが使用されます。

例 2–1. inline キーワードの使用方法

```
inline float 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されたインライン展開をオフにします。このオプションが有効なのは、特定レベルの最適化が必要であるが定義制御インライン展開は必要ない場合です。

2.10.4 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・サイズが増加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・サイズを上昇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 に表示されているように、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 に表示されているように、.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 では、strlen 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 番目の定義はライブラリ用です。これにより、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 strlen が存在することが確保されます。これはインライン関数ではないので、string.h が組み込まれ strlen 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）string.h

```
/*****************************************************************************/
/* string.h  vx.xx                                                           */
/* Copyright (c) 1993-1999 Texas Instruments Incorporated                    */
/* Excerpted ...                                                             */
/*****************************************************************************/
#ifdef _INLINE
#define _IDECL static inline
#else
#define _IDECL extern _CODE_ACCESS
#endif

_IDECL size_t  strlen(const char *_string);

#ifdef _INLINE

/*****************************************************************************/
/*  strlen                                                                   */
/*****************************************************************************/
static inline size_t strlen(const char *string)
{
   size_t      n = (size_t)-1;
   const char *s = string - 1;

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

#endif
```

（b）strlen.c

```
/*****************************************************************************/
/*  strlen                                                                   */
/*****************************************************************************/
#undef _INLINE
#include <string.h>
{
_CODE_ACCESS size_t strlen(cont char * string)
   size_t      n = (size_t)-1;
   const char *s = string - 1;

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

2.10.5 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インライン展開できる関数には、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個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数カ所でしか呼び出され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、インライン展開が無効になります。

- ❏ 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 型を宣言する
- ❏ 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）

2.11 割り込み柔軟性オプション（-mi オプション）

C6000 体系では、分岐の遅延スロット内で割り込みを発生することはできません。場合によってコンパイラは、多数のサイクル数になる可能性がある期間、割り込みができないコードを生成することがあります。所定のリアルタイム・システムの場合、割り込みを無効にできる期間に厳しい制限がある場合があります。

-mi*n* オプションは、割り込みしきい値 *n* を指定します。このしきい値は、コンパイラが割り込みを無効にできる最大サイクル数を指定します。*n* を省略すると、コンパイラはコードが割り込まれないものと想定します。Code Composer Studio では、コードが割り込まれないことを指定するには、「Compiler」タブ、「Advanced」カテゴリの「Build Options」ダイアログ・ボックス内にある「Interrupt Threshold」チェック・ボックスを選択して、テキスト・ボックスを空白のままにしておく必要があります。

-mi*n* を指定しないと、ソフトウェア・パイプライン・ループに対してのみ、明示的に割り込みが無効になります。-mi*n* オプションを使用する場合、コンパイラは、ループ構造とループ・カウンタを分析して 1 つのループを実行するのにかかる最大サイクル数を判別します。最大サイクル数がしきい値より小さいことが判別できる場合、コンパイラは最高速／最適バージョンのループを生成します。ループが 6 サイクルより小さい場合、ループは常に分岐の遅延スロット内で実行されるため、割り込みを発生することはできません。そうでない場合、コンパイラは割り込み可能なループを生成し（正しい結果 - 単一の代入コードを生成）できますが、ほとんどの場合ループのパフォーマンスが低下します。

-mi*n* オプションはメモリ・システムの影響を含みません 1 つのループに対する実行サイクルの最大数を判別する際、コンパイラは、速度の遅いオフチップ・メモリまたはメモリ・バンクの競合を使用する影響を計算しません。メモリ・システムの影響に合わせて調整するには、控え目なしきい値を使用することをお勧めします。

詳細は、7.7.7 項「FUNC_INTERRUPT_THRESHOLD プラグマ」（7-24 ページ）、または TMS320C6000 Programmer's Guide を参照してください。

注 : RTS ライブラリ・ファイルは -mi オプションでは作成されていない

コンパイラに付属するランタイム・サポート・ライブラリ・ファイルは、割り込み柔軟性オプションでは作成されていません。使用するリリースでランタイム・サポート・ライブラリ・ファイルがどのように生成されたかを知るには、README ファイルを参照してください。割り込み柔軟性オプションを使用して、独自のランタイム・サポート・ライブラリ・ファイルを作成するには、第 10 章「ライブラリ作成ユーティリティ」を参照してください。

注 : -mi オプションを使用する特殊な場合

-mi0 オプションは、ソフトウェア・パイプライン・ループへの割り込みを無効にする、-mi オプションを使用しない場合と同じコードを生成します。

-mi オプション（しきい値を省略した場合）は、ソフトウェア・パイプライン・ループへの割り込みを無効にするためにコードを追加しないことを意味します。また、コンパイラは、分岐命令の遅延スロット内にはないループ・カーネルに少なくとも 1 つのサイクルが存在することを確認し、ループを割り込み可能にしようとしないためループのパフォーマンスは低下しません。

2.12 C6400 コードを C6200/C6700/ 旧世代 C6400 オブジェクト・コードにリンクする方法

特定のパック・データの最適化を容易にするために、C6400 ファミリー用の最上位の配列の位置合わせは 4 バイトから 8 バイトに変更されました（C6200 および C6700 コードの場合、最上位の配列の位置合わせは常に 4 バイトです）。

C6400 コードを C6200/6700 コードや旧世代の C6400 コードにリンクする場合は、互換性を確実にするために複数のステップを行う必要があります。以下は、潜在的な位置合わせ競合、およびそれに対する可能な解決方法を掲載しています。

潜在的な位置合わせ競合は、次の場合に発生します。

- C6400 の新しいコードを、4.0 ツールを使用してコンパイルした C6400 コードにリンクする場合
- C6400 の新しいコードを、（いずれかのバージョンを使用してコンパイルされた）C6200 または C6700 ファミリー用のコードとリンクする場合

解決方法（どちらかを選択）

- -mv6400 スイッチを使用して、アプリケーション全体を再コンパイルします。この解決方法はアプリケーションのパフォーマンスを向上させるので、（可能であれば）推奨します。
- -mb オプションを使用して新しいコードをコンパイルします。-mv6400 スイッチを使用すると、-mb スイッチは最上位の配列を 4 バイトに変更します。

2.13 インターリストの使用方法

コンパイラ・ツールには、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機能が含まれています。この機能を使用すると、各 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動作が異なります。

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

```
cl6x -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機能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。インターリスト機能は、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。

-ss オプションを使用すると、パフォーマンスが低下するか、またはコード・サイズが大きくなる可能性があります。

f は、差し込み後のアセンブリ・ファイルの一般的な例を示したものです。

例 2–3. 差し込み後のアセンブリ言語ファイル

```
_main:
           STW     .D2     B3,*SP--(12)
           STW     .D2     A10,*+SP(8)
;------------------------------------------------------------------------------
;   5 | printf("Hello, world\n");
;------------------------------------------------------------------------------
           B       .S1     _printf
           NOP             2
           MVKL    .S1     SL1+0,A0
           MVKH    .S1     SL1+0,A0
||         MVKL    .S2     RL0,B3
           STW     .D2     A0,*+SP(4)
||         MVKH    .S2     RL0,B3
RL0:       ; CALL OCCURS
;------------------------------------------------------------------------------
;   6 | return 0;
;------------------------------------------------------------------------------
           ZERO    .L1     A10
           MV      .L1     A10,A4
           LDW     .D2     *+SP(8),A10
           LDW     .D2     *++SP(12),B3
           NOP             4
           B       .S2     B3
           NOP             5
           ; BRANCH OCCURS
```

オプティマイザと一緒にインターリスト機能を使用する方法の詳細は、3.11 節「最適化でのインターリスト機能の使用方法」（3-30 ページ）を参照してください。

第 3 章

コードの最適化

コンパイラ・ツールを使用することにより、ループの簡略化、ソフトウェア・パイプライン、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のタスクを実行し、C および C++ プログラムの実行速度を向上させたりプログラム・サイズを縮小させたりするなど、数々の最適化の機能を実行できます。

本章では、さまざまなレベルの最適化の起動方法と、各レベルで実行される最適化について説明します。また、最適化の実行時に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**

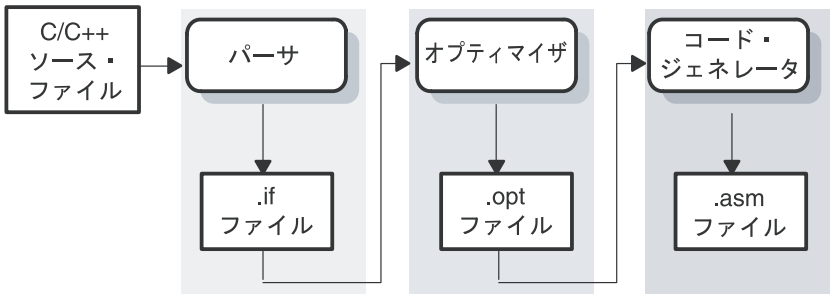
3.1	オプティマイザの起動方法	3-2
3.2	ソフトウェア・パイプライン化による最適化	3-4
3.3	冗長なループ	3-16
3.4	コード・サイズの縮小方法（-ms オプション）	3-17
3.5	ファイル・レベルの最適化の実行（-O3 オプション）	3-18
3.6	プログラム・レベルの最適化の実行（-pm および -O3 オプション）	3-20
3.7	特定のエイリアス指定技法を使用するかどうかの指定方法	3-25
3.8	加法の浮動小数点演算の並べ換えの防止	3-28
3.9	最適化コード内で asm 文を使用する場合の注意	3-28
3.10	自動インライン展開（-oi オプション）	3-29
3.11	最適化でのインターリスト機能の使用方法	3-30
3.12	最適化されたコードのデバッグ方法とプロファイル方法	3-33
3.13	実行できる最適化の種類	3-35

3.1 オプティマイザの起動方法

C/C++ コンパイラでは、さまざまな最適化が実行できます。高レベルの最適化はオプティマイザで実行され、低レベルのターゲット固有の最適化はコード・ジェネレータで行われます。コードの最適化を実現するためには、高レベルの最適化を行う必要があります。

図 3-1 は、オプティマイザとコード・ジェネレータを使用したコンパイラの実行の流れを示しています。

図 3-1. 最適化による C/C++ プログラムのコンパイル方法



最適化を起動する最も簡単な方法は、cl6x コマンド行で -O*n* オプションを指定して、cl6x コンパイラ・プログラムを実行する方法です。*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。
- inline と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。

- ソフトウェアのパイプライン化を実行します（3.2 節「ソフトウェア・パイプライン化による最適化」（3-4 ページ）を参照）。
- 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。
- ループ内の配列参照を、増分されるポインタ形式に変換します。
- ループの展開を行います。

-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。
- すべての呼び出しが同じ引数位置で同じ値を渡す場合、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は、3.5 節「ファイル・レベルの最適化の実行（-O3 オプション）」（3-18 ページ）を参照してください。

上述の最適化のレベルは、独立した最適化パスで実行されます。コード・ジェネレータは他にもいくつかの最適化、とりわけプロセッサ固有の最適化を実行します。この最適化は、オプティマイザを起動するかどうかにかかわらず実行されますが、オプティマイザを使用するとより効果的に実行されます。

注：コード・サイズを制御するために最適化レベルを下げないでください

コード・サイズを縮小しようとして最適化レベルを下げてはなりません。代わりに、-ms オプションを使用して、コード・サイズとパフォーマンスのトレードオフを制御します。通常、高い -O レベルを高い -ms レベルと組み合わせるとコード・サイズを最小化することができます。詳細は、3.4 節「コード・サイズの縮小方法（-ms オプション）」（3-17 ページ）を参照してください。

注 : -On オプションはアセンブリ・オプティマイザに適用されます

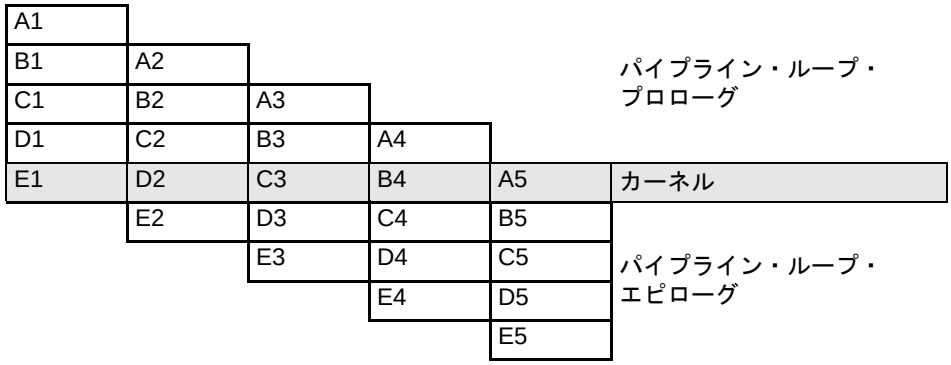
また、-On オプションをアセンブリ・オプティマイザでも使用する必要があります。アセンブリ・オプティマイザは、この章で説明するすべての最適化を実行するわけではありませんが、ソフトウェア・パイプライン化およびループの展開などの主要な最適化は -O オプションを指定する必要があります。

3.2 ソフトウェア・パイプライン化による最適化

ソフトウェア・パイプライン化とは、ループの複数の反復を並行して実行できるように、ループからの命令をスケジュールする技法です。コンパイラは、常にソフトウェアのパイプライン化を試みます。一般に、-O2 または -O3 オプションを使用するとコード・サイズとパフォーマンスが向上します。(3.1 節「オプティマイザの起動方法」(3-2 ページ) を参照)。また、コード・サイズを縮小するには、-ms オプションも使用する必要があります。

図 3-2 は、ソフトウェア・パイプライン・ループを示しています。ループの各段階は A、B、C、D、および E で表されています。この図では、一度にループを最高 5 回反復して実行できます。陰影のある部分は、ループのカーネルを表しています。ループのカーネルでは、5 つの段階がすべて並列して実行されます。カーネルの上にある部分はパイプライン・ループ・プロローグと呼ばれ、カーネルの下にある部分はパイプライン・ループ・エピローグと呼ばれます。

図 3-2. ソフトウェア・パイプライン・ループ



アセンブリ・オプティマイザもまた、ループのソフトウェア・パイプライン化を行います。アセンブリ・オプティマイザの詳細は、第 4 章を参照してください。ソフトウェア・パイプラインの詳細は、TMS320C6000 Programmer's Guide を参照してください。

3.2.1 ソフトウェア・パイプライン化の取り止め（-mu オプション）

最適化レベル -O2 および -O3 では、コンパイラはループのソフトウェア・パイプライン化を試みます。デバッグのためにループをソフトウェア・パイプライン化したくない場合があります。コードがシリアルに表示されないため、ソフトウェア・パイプライン・ループのデバッグが困難になることがあるからです。-mu オプションは、コンパイルされた C/C++ コードとアセンブリ最適化されたコードの両方に影響を与えます。

注：ソフトウェア・パイプライン化はコード・サイズを大幅に増加させる可能性がある

コード・サイズを縮小するには、パフォーマンス上重要ではないコードに対して -mu オプションではなく -ms2 または -ms3 オプションを使用します。これらのコード・サイズ・オプションはソフトウェア・パイプライン化を無効にすると同時に、コード・サイズ縮小最適化を有効にします。

3.2.2 ソフトウェア・パイプライン化情報

コンパイラは、.asm ファイルにソフトウェア・パイプライン・ループ情報を出力します。この情報は、C/C++ コードまたはリニア・アセンブリ・コードを最適化するのに使用されます。

ソフトウェア・パイプライン情報は、.asm ファイル内ではループの前のコメントとして表示され、その情報はアセンブリ・オプティマイザが稼働している間表示されます。例 3-1 は、ループごとに生成される情報を示しています。

-mw オプションは、ループ・カーネルの各サイクルでのレジスタ使用状況を示す情報を追加し、ソフトウェア・パイプライン・ループを 1 回反復するように指示する命令を表示します。

注：ソフトウェア・パイプライン情報の詳細

各ループの前のソフトウェア・パイプライン情報コメント・ブロックに表示されるすべての情報とメッセージの詳細は、TMS320C6000 Programmer's Guide の第 2 章または付録 A を参照してください。

例 3-1.　ソフトウェア・パイプライン情報

```
;*----------------------------------------------------------------------------*

;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count         : 2
;*      Known Maximum Trip Count         : 2
;*      Known Max Trip Count Factor      : 2
;*      Loop Carried Dependency Bound(^) : 4
;*      Unpartitioned Resource Bound     : 4
;*      Partitioned Resource Bound(*)    : 5
;*      Resource Partition:
;*        A-side   B-side
;*      .L units         2        3
;*      .S units         4        4
;*      .D units         1        0
;*      .M units         0        0
;*      .X cross paths   1        3
;*      .T address paths 1        0
;*      Long read paths  0        0
;*      Long write paths 0        0
;*      Logical  ops (.LS)           0        1     (.L or .S unit)
;*      Addition ops (.LSD)          6        3     (.L or .S or .D unit)
;*      Bound(.L .S .LS) 3        4
;*      Bound(.L .S .D .LS .LSD)     5*       4
;*

;*      Searching for software pipeline schedule at ...
;*         ii = 5  Register is live too long
;*         ii = 6  Did not find schedule
;*         ii = 7  Schedule found with 3 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 1
;*
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad :2 bytes
;*
;*      Minimum safe trip count     : 2
;*
;*----------------------------------------------------------------------------*
```

以下に定義される用語は、ソフトウェア・パイプライン情報内に表示されます。各用語の詳細は、TMS320C6000 Programmer's Guide を参照してください。

- **Loop unroll factor**
  ループが、ソフトウェア・パイプライン・ループ内のリソース制約に基づいて、特にパフォーマンスを向上させるために展開された回数。

- **Known minimum trip count**
  ループが実行される最少回数。

- **Known maximum trip count**
  ループが実行される最多回数。

- **Known max trip count factor**
  ループ・トリップ・カウントを常に均等に分割する係数。この情報はループの展開に使用できます。

- **Loop label**
  リニア・アセンブリ入力ファイル内でループに指定したラベル。C/C++ コードの場合、このフィールドは表示されません。

- **Loop carried dependency bound**
  最大ループ・キャリー・パスの距離。ループ・キャリー・パスが生じるのは、ループの 1 回の反復により、次回の反復で読み取る必要がある値が書き込まれた場合です。ループ・キャリー境界の一部である命令には ^ 記号が付きます。

- **Iteration interval** (ii)
  ループの反復周期のサイクル数。反復間隔が短いほど、1 つのループの実行にかかるサイクル数が少なくなります。

- **Resource bound**
  最もよく利用されるリソースが、最小反復間隔に制約を加えます。たとえば 4 つの命令が .D ユニットを必要としている場合には、実行するのに少なくとも 2 サイクルが必要です（4 つの命令／2 つの 並列 .D ユニット）。

- **Unpartitioned resource bound**
  ループ内の命令が特定のサイドに配分される前の、可能な限り最良のリソース境界値。

- **Partitioned resource bound (*)**
  命令が配分された後のリソース境界値。

- **Resource partition**
  この表は、命令がどのように配分されたかをまとめたものす。この情報を使用すると、リニア・アセンブリを書き込む際に機能ユニットを割り当てるのに役立ちます。表の各項目には、A サイドと B サイドのレジスタの値があります。リソース境界値を判別する項目にマークを付けるのに、アスタリスクが使用されています。この表の項目は、次の用語を表しています。

  - **.L units** は、.L ユニットを必要とする命令の合計数です。

- **.S units** は、.S ユニットを必要とする命令の合計数です。
- **.D units** は、.D ユニットを必要とする命令の合計数です。
- **.M units** は、.M ユニットを必要とする命令の合計数です。
- **.X cross paths** は .X クロス・パスの合計数です。
- **.T address paths** はアドレス・パスの合計数です。
- **Long read path** は、長い読み取りポート・パスの合計数です。
- **Long write path** は、長い書き込みポート・パスの合計数です。
- **Logical ops (.LS)** は、.L または .S ユニットのどちらかを使用できる命令の合計数です。
- **Addition ops (.LSD)** は、.L または .S または .D ユニットのどれかを使用できる命令の合計数です。

❏ **Bound(.L .S .LS)**

.L および .S ユニットを使用する命令数によって決定されるリソース境界値です。これは次の式で計算されます。

Bound(.L .S .LS) = ceil((.L + .S + .LS) / 2)

❏ **Bound(.L .S .D .LS .LSD)**

.D、.L、および .S ユニットを使用する命令数によって決定されるリソース境界値です。これは次の式で計算されます。

Bound(.L .S .D .LS .SLED) = ceil((.L + .S + .D + .LS + .LSD) / 3)

❏ **Minimum required memory pad**

見込み実行が可能である場合に読み取られるバイト数。詳細は、3.2.3 項「パフォーマンスとコード・サイズを改善するためのプロローグとエピローグの縮小」（3-14 ページ）を参照してください。

3.2.2.1 ソフトウェア・パイプラインに不適格なループのメッセージ

ループがソフトウェア・パイプラインに全く不適格な場合は、次のメッセージが表示されます。

❏ **Bad loop structure.**
このエラーが発生するのは非常に稀であり、次の原因が考えられます。

- C コード内部ループに挿入された asm 文であるため
- リニア・アセンブリ・オプティマイザへの入力として使用される並列命令
- GOTO 文や break などの複雑な制御フロー

❏ **Loop contains a call.**
場合によっては、コンパイラがループ内にある関数呼び出しをインライン展開できない可能性があります。コンパイラが関数呼び出しをインライン展開できなかったので、ループをソフトウェア・パイプライン化できませんでした。

❏ **Too many instructions.**
ソフトウェア・パイプライン化するには、ループ内にある命令が多過ぎます。

❏ **Software pipelining disabled.**
コマンド行オプションにより、ソフトウェア・パイプライン化が無効になっています。-mu オプションを使用するか、-O2 オプションも -O3 オプションも使用しないか、-ms2 または -ms3 オプションを使用する場合、パイプライン化は実行されません。

❏ **Uninitialized trip counter.**
トリップ・カウンタが初期値に設定されていない可能性があります。

❏ **Suppressed to prevent code expansion.**
-ms1 オプションが指定されているので、ソフトウェア・パイプライン化が抑止されている可能性があります。-ms1 オプションを使用すると、コード・サイズの縮小があまり期待できない場合にはソフトウェア・パイプラインが抑止されます。パイプラインを有効にするには、-ms0 を使用するか、-ms オプションをすべて省略します。

❏ **Loop carried dependency bound too large.**
ループに複雑なループ制御がある場合、3.2.3.2 項「最善のしきい値の選択方法」（3-15 ページ）にある推奨事項にしたがって -mh を試行してください。

❏ **Cannot identify trip counter.**
ループ・トリップ・カウンタが特定できなかったか、あるいはループ本体内で誤って使用されました。

3.2.2.2 パイプライン障害メッセージ

コンパイラまたはアセンブリ・オプティマイザがソフトウェア・パイプラインを処理しているときに、障害が発生すると、以下のメッセージが表示される場合があります。

- **Address increment is too large.**
  オフセットが C6000 のオフセット・アドレッシング・モードの範囲外なので、アドレス・レジスタのオフセットを調整する必要があります。アドレス・レジスタのオフセットを最小限に抑える必要があります。

- **Cannot allocate machine registers.**
  ソフトウェア・パイプライン・スケジュールが検出されたにもかかわらず、そのスケジュール用にマシン・レジスタを割り当てることができません。ループを簡略化する必要があります。

  特定の ii で検出されたスケジュールのレジスタ使用状況が表示されます。この情報は、レジスタ・ファイルの両側でレジスタの圧力のバランスを取るためにリニア・アセンブリを書き込むときに使用できます。たとえば次のとおりです。

```
ii = 11 Cannot allocate machine registers
Regs Live Always : 3/0  (A/B-side)
Max Regs Live : 20/14
Max Cond Regs Live : 2/1
```

  - **Regs Live Always** は、ループ本体全体の処理中にレジスタに割り当てる必要がある値の数。これは、これらの値が、ループ用に検出された任意の所定スケジュールに対してレジスタに常に割り当てる必要があるという意味です。
  - **Max Regs Live** は、レジスタに割り当てる必要がある、ループ内の所定サイクルで有効な値の最大数。これは、検出されたスケジュールが必要とするレジスタの最大数を示します。
  - **Max Cond Regs Live** は、条件レジスタに割り当てる必要がある、ループ・カーネル内の任意の所定サイクルで有効なレジスタの最大数。

- **Cycle count too high - Not profitable.**
  コンパイラがループについて検出したスケジュールでは、非ソフトウェア・パイプライン・バージョンを使用する方が効率的です。

- **Did not find schedule.**
  コンパイラが、所定の ii でソフトウェア・パイプライン用のスケジュールを検出できませんでした。ループを簡略化し、ループ保持依存性を排除する必要があります。

- **Iterations in parallel > minimum or maximum trip count.**
  ソフトウェア・パイプライン・スケジュールが検出されましたが、そのスケジュールには最小または最大のループ・トリップ・カウントよりも多くの並行した反復があります。冗長なループを有効にするか、トリップ情報を伝達する必要があります。

- **Speculative threshold exceeded.**
  -mh オプションによって現在指定されているしきい値を超えてロードすることが必要です。アセンブリ・ファイルに出力されたソフトウェア・パイプライン・フィードバックで推奨されるように、-mh しきい値を上げる必要があります。

❑ **Register is live too long.**
レジスタには、ii サイクルよりも長い間有効な値を保持する必要があります。MV 命令を挿入すると、長過ぎるレジスタ・ライフタイムを分割できます。

アセンブリ・オプティマイザが使用中である場合、ライフタイムが長過ぎるレジスタを定義し、使用する命令の .sa ファイル行番号がこの障害メッセージの後に出力されます。

ii = 9 Register is live too long
|10| → |17|

これは、レジスタ値を定義する命令が sa ファイルの 10 行目にあり、レジスタ値を使用する命令が 17 行目にあることを意味します。

❑ **Too many predicates live on one side.**
'C6000 には、条件つき命令で使用できる述部、つまり条件レジスタがあります。'C6200 および 'C6700 には 5 つの述部レジスタがあり、'C6400 には 6 つの述部レジスタがあります。A サイドには 2 つか 3 つ、B サイドには 3 つあります。場合によっては、特定の区画とスケジュールの組み合わせにこれらの使用可能なレジスタより多くのレジスタが必要です。

❑ **Schedule found with N iterations in parallel.**
並列して実行される N 反復で、ソフトウェア・パイプライン・スケジュールが検出されました。

❑ **Too many reads of one register.**
'C6200 または 'C6700 コアでは、サイクルごとに最高 4 回、同一レジスタを読み取ることができます。'C6400 コアは、サイクルごとに任意の回数、同一レジスタを読み取ることができます。

❑ **Trip variable used in loop - Can't adjust trip count.**
このループ・トリップ・カウンタは、ループ内でループ・トリップ・カウンタ以外に使用されています。

3.2.2.3 調査フィードバック

コンパイラまたはアセンブリ・オプティマイザが、ソフトウェア・パイプラインでパフォーマンスを改善できることを検出した場合には、以下のメッセージが表示されます。

- **Loop carried dependency bound is much larger than unpartitioned resource bound.**
  メモリ・エイリアスの明確化に問題がある可能性があります。これは、同じ位置を指しているかどうかにかかわらず 2 つのポインタがあり、コンパイラはそれらのポインタが同じ位置を指すものと想定する必要があるという意味です。これが原因で、実際には存在しない依存関係（多くの場合、一方のポインタのロードともう一方のポインタのストアとの間の依存関係）が生じる可能性があります。ソフトウェア・パイプライン・ループの場合は、これが原因でパフォーマンスが大きく低下することがあります。

- **Two loops are generated, one not software pipelined.**
  トリップ・カウントが小さすぎる場合は、ソフトウェア・パイプライン・バージョンのループを実行してはいけません。この場合コンパイラは、最小トリップ・カウントが常に安全にパイプライン・バージョンを実行できる大きさであることを保証できません。したがって、非パイプライン・バージョンも生成しました。実行時に適切なバージョンのループが実行されるように、コードが生成されます。

- **Uneven resources.**
  特定の操作を実行するためのリソース数が奇数のときには、ループの展開が有益な場合があります。ループに 3 つの乗算が必要な場合、これを実行するには最小反復間隔が 2 サイクルである必要があります。このループが展開されると A サイドと B サイドにまたがって 6 つの乗算が均等に分割され、最小 ii が 3 サイクルになり、パフォーマンスが改善される可能性があります。

- **Larger outer loop overhead in nested loop.**
  ネストされたループの内部ループ・カウントが相対的に小さいと、外部ループを実行する時間が総実行時間に占める割合が大きくなり始める場合があります。これが全体的なループのパフォーマンスを大きく低下させる場合には、内部ループの展開が必要になる可能性があります。

- **There are memory bank conflicts.**
  コンパイラが 1 サイクルで 2 つメモリ・アクセスするコードを生成し、それらのアクセスが 'C620x で 8 バイト離れているか、'C670x で 16 バイト離れているか、または 'C640x で 32 バイト離れているときにかつ両方のアクセスが同じメモリ・ブロック内にある場合には、メモリ・バンクの停止が発生します。メモリ・バンクの競合を完全に回避するには、この 2 つのアクセスを異なるメモリ・ブロック内に入れるか、またはリニア・アセンブリを書き込み、.mptr 疑似命令を使用してメモリ・バンクを制御します。

- **T address paths are resource bound.**
  T アドレス・パスでは、ループの反復ごとにアドレス・バス上に送出しなければならないメモリ・アクセスの数が定義されています。これらがループ用に拘束されたリソースである場合には、実行中の短いアクセスに対してワード・アクセス（LDW/STW）を実行すると、多くの場合アクセス数を減らすことができます。

3.2.2.4 -mw オプションによって生成されたレジスタ使用状況テーブル

-mw オプションを指定すると、生成したアセンブリ・ファイルに追加ソフトウェア・パイプライン・フィードバックが出力されます。この情報には、ソフトウェア・パイプライン・ループの 1 回反復するようにスケジュールされたビューが含まれています。

ソフトウェア・パイプライン化が特定のループで続行され、コンパイル・プロセス中に -mw オプションが使用された場合、生成したアセンブリ・コード内のソフトウェア・パイプライン情報のコメント・ブロックにレジスタ使用状況テーブルが追加されます。

各行の数値はループ・カーネル内のサイクル番号を表します。

各列は TMS320C6000 上の 1 つのレジスタを表します（すべての C6000 プロセッサに対する A0 ～ A15、B0 ～ B15、さらに C6400 の場合には A16 ～ A31、B16 ～ B31）。レジスタは、レジスタ使用状況テーブルの最初の 3 行でラベルが指定され、列方向に読み取ります。

テーブル・エントリの * は、列ヘッダで示したレジスタが、各行に表示されたサイクル番号によって指定されたカーネルの実行パケット上で有効になっていることを示します。

レジスタ使用状況テーブルの例を以下に示します。

```
;*      Searching for software pipeline schedule at
;*         ii = 15 Schedule found with 2 iterations in parallel
;*
;*      Register Usage Table:
;*          +---------------------------------+
;*          |AAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBB|
;*          |0000000000111111|0000000000111111|
;*          |0123456789012345|0123456789012345|
;*          |----------------+----------------|
;*       0: |***    ****     |*** ******     |
;*       1: |****   ****     |*** ******     |
;*       2: |****   ****     |*** ******     |
;*       3: | **   *****     |*** ******     |
;*       4: | **   *****     |*** ******     |
;*       5: | **   *****     |*** ******     |
;*       6: | **   *****     |**********     |
;*       7: |***   *****     |** *******     |
;*       8: |**** *****      |**********     |
;*       9: |**********      |** ********    |
;*      10: |**********      |** *********   |
;*      11: |**********      |** *********   |
;*      12: |**********      |**********    |
;*      13: |**** *****      |** ******* *   |
;*      14: |***  *****      |*** ****** *   |
;*          +---------------------------------+
;*
```

この例は、ループ・カーネルのサイクル 0（最初の実行パケット）のレジスタ A0、A1、A2、A6、A7、A8、A9、B0、B1、B2、B4、B5、B6、B7、B8、および B9 がサイクル中に有効であることを示しています。

3.2.3 パフォーマンスとコード・サイズを改善するためのプロローグとエピローグの縮小

ループのソフトウェア・パイプライン化が行われると、一般的にプロローグとエピローグが必要になります。プロローグはループのパイプ接続に使用され、エピローグはループのパイプ切断に使用されます。

一般にループは、ソフトウェア・パイプライン・バージョンを安全に実行する前に、最小数の反復を実行しておく必要があります。既知の最小トリップ・カウントが小さすぎる場合、冗長なループが追加されるか、ソフトウェア・パイプラインが使用不可になります。ループのプロローグとエピローグを縮小すると、パイプライン・ループを安全に実行するのに必要な最小トリップ・カウントが減少する場合があります。

縮小は、コード・サイズも大幅に減少させることができます。このコード・サイズの拡大の原因の一部は、冗長なループです。その他の原因はプロローグとエピローグです。

ソフトウェア・パイプライン・ループのプロローグとエピローグは、長さ *ii* の最高 *p*-1 段階で構成されます。この場合、*p* は定常状態中に並行して実行される反復数であり、*ii* はパイプライン化されたループ本体のサイクル・タイムです。プロローグとエピローグの縮小中に、コンパイラはできる限り段階を縮小しようとします。しかし、縮小し過ぎるとパフォーマンスにマイナスの影響を与える場合があります。したがってデフォルトでは、コンパイラはパフォーマンスを低下させずに、可能な限りの段階数を縮小しようとします。-ms0/-ms1 オプションが起動されると、コンパイラはパフォーマンスよりコード・サイズをさらに優先します。

3.2.3.1 見込み実行

プロローグとエピローグが縮小されると、命令は見込みで実行される可能性があります。それにより、ループ内で明示的に読み取られる範囲の上限を超えるアドレスへロードされる可能性があります。不正のメモリ位置から読み取る可能性があるので、デフォルトでは、コンパイラはロードを見込みで実行することはできません。場合によっては、コンパイラはこれらのロードを予測し、過剰な実行を防止できます。しかし、これによりレジスタへの負担が増大し、実行可能な合計縮小量が減少する可能性があります。

--speculate_loads*n* オプションを使用する場合、見込みしきい値はデフォルトの 0 から *n* に増加します。このしきい値が *n* である場合、コンパイラが読み取るメモリ位置がループ内で明示的に読み取られた位置の前後 *n* バイトを超えないときに、コンパイラはロードを見込みで実行することを許可できます。*n* を省略すると、コンパイラは見込みしきい値が制限されないものと見なします。Code Composer Studio でこれを指定するには、「Compiler」タブ、「Advanced」カテゴリの「Build Options」ダイアログ・ボックス内にある「Speculate Threshold」チェック・ボックスを選択して、テキスト・ボックスを空白のままにしておきます。

縮小により、通常は最小安全トリップ・カウントが減少する可能性があります。既知の最小トリップ・カウントが最小安全トリップ・カウントより小さい場合には、冗長なループが必要です。それ以外の場合は、パイプライン化を抑止する必要があります。これらの値は、どちらもソフトウェア・パイプライン・ループに先行するコメント・ブロック内で検出できます。

```
;*      Known Minimum Trip Count   : 1
....
;*      Minimum safe trip count    : 7
```

最小安全トリップ・カウントが既知の最小トリップ・カウントより大きい場合には、-mh を使用することを特にお勧めします。これはコード・サイズのためだけでなく、パフォーマンスのためでもあります。

-mh を使用する場合には、見込み実行される可能性があるロードが無許可の読み取りを行わないようにしなければなりません。これは必要に応じて、必要な両方向のメモリ埋め込みによるデータ・セクションまたはスタック（あるいは両方）の埋め込みによって行うことができます。特定のソフトウェア・パイプライン・ループに必要なメモリ埋め込みも、そのループのコメント・ブロックに指定されます。

```
;*      Minimum required memory pad : 8 bytes
```

3.2.3.2 最善のしきい値の選択方法

ループのソフトウェア・パイプライン化が行われると、ループに先行するコメント・ブロックが次の情報を提供します。

- ❏ このループに必要なメモリ埋め込み
- ❏ このソフトウェア・パイプライン・スケジュールと縮小レベルの実現に必要な最小値 *n*
- ❏ 追加の縮小を可能にする、より大きな値 *n* の推奨

この情報は、コメント・ブロック化内で次のように表示されます。

```
;*   Minimum required memory pad :5 bytes
;*   Minimum threshold value     :-mh7
;*
;*   For further improvement on this loop, try option -mh14
```

安全のために、このループ例では、このループ内で参照される配列データの前後に 5 バイト以上の埋め込みが必要です。この埋め込みは他のプログラム・データで構成できます。この埋め込みは修正されません。多くの場合、しきい値（つまり特定のスケジュールと縮小レベルの実現に必要な、-mh に対する引数の最小値）は、埋め込みと同じです。しかし、異なる場合はコメント・ブロックに最小しきい値も含まれます。このループの場合、このレベルの縮小を実現するには、しきい値は 7 以上でなければなりません。

もう 1 つの興味深い問題は、追加の縮小を促す、より大きなしきい値があるかどうかです。ある場合は、この情報も提供されます。たとえば上記のコメント・ブロックでは、しきい値 14 がさらに縮小を促します。

3.3 冗長なループ

ループは、そのループの終了前に何回かの反復をしています。ループで反復する回数はトリップ・カウントと呼ばれます。反復の回数を記録するのに使用される変数がトリップ・カウンタです。トリップ・カウンタがトリップ・カウントの限界値に達すると、そのループは終了します。C6000 ツールは、トリップ・カウントを使用してループをパイプライン化できるかどうかを判別します。パイプラインを埋め込むかプライムするために、ソフトウェア・パイプライン・ループの構造には、最小のループ反復回数（最小トリップ・カウント）の実行が必要です。

ソフトウェア・パイプライン・ループの最小トリップ・カウントは、並列に実行される反復の回数によって決まります。図 3-2 では、最小トリップ・カ ウントは 5 です。次の例では、A、B、および C は、ソフトウェア・パイプライン内の命令であるので、この単一サイクルのソフトウェア・パイプライン・ループの最小トリップ・カウントは 3 です。

```
A
B   A
C   B   A     ← 並列した 3 つの反復 = 最小トリップ・カウント
    C   B
        C
```

C6000 ツールがループのトリップ・カウントを判別できない場合には、デフォルトにより 2 つのループと制御ロジックが生成されます。最初のループはパイプライン化されず、ランタイム・トリップ・カウントがそのループの最小トリップ・カウントより小さい場合に実行されます。2 番目のループはソフトウェア・パイプライン・ループであり、ランタイム・トリップ・カウントが最小トリップ・カウントより大きいか等しい場合に実行されます。常にどちらかのループが、冗長なループになります。:

```
foo(N)  /* N is the trip count */
{
   for (i=0; i < N; i++) /* i is the trip counter */
}
```

ループのソフトウェア・パイプラインを検出した後、コンパイラは、そのループの最小トリップ・カウントが 3 であるものと想定して、以下のように foo() を変換します。そのループの 2 つのバージョンが生成され、どちらのバージョンが実行されるべきかを判別するために、以下の比較が使用されます。

```
foo(N)
{
   if (N < 3)
   {
      for (i=0; i < N; i++)   /* Unpipelined version */
   }
   else
   }
      for (i=0; i < N; i++)   /* Pipelined version   */
   }
}
foo(50); /* Execute software pipelined loop          */
foo(2);  /* Execute loop (unpipelined)*/
```

-pm -O3（3.6 節「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）」（3-20 ページ）を参照）または MUST_ITERATE プラグマ（7.7.14 項「MUST_ITERATE プラグマ」（7-28 ページ）を参照）を使用すると、コンパイラの冗長なループの生成を回避するのに役立ちます。

注 : 冗長ループの取り止め
冗長ループを取り止めるには、-ms オプションを指定します。

3.4 コード・サイズの縮小方法（-ms オプション）

-O または -O*n* オプションを使用すると、コードを最適化するようにコンパイラに指示することになります。*n* の値が大きいほど、コンパイラがコードの最適化を行う負担が増大します。しかしながら最適化の優先順位については、コンパイラに指示を与える必要があります。

デフォルトでは、-O2 または -O3 が指定されると、コンパイラは第一にパフォーマンスを確保するために最適化を行います（これより低い最適化レベルでは、優先されるのはコンパイル時間とデバッグの容易さです）。コード・サイズ・フラグ -ms*n* を使用すると、パフォーマンスとコード・サイズとの間で優先順位を調整できます。-ms0、-ms1、-ms2、-ms3 オプションの順に、パフォーマンスよりコード・サイズが優先されるようになります。

パフォーマンスを最優先させるコードでは、コード・サイズ・フラグを使用しないことをお勧めします。パフォーマンスを最優先させるコード以外のすべてのコードには、ms0 または -ms1 を使用することをお勧めします。めったに実行されないコードには、-ms2 または -ms3 を使用することをお勧めします。コード・サイズを最小にする必要がある場合にも -ms2 か -ms3 のどちらかを使用してください。どの場合にも、一般に、コード・サイズ・フラグと -O2 または -O3 を組み合わせることをお勧めします。

注 : コード・サイズ最適化の抑止または最適化レベルの引き下げ
最適化レベルを引き下げ、コード・サイズ・フラグを使用しない場合、あるいはそのどちらかの場合、コード・サイズ最適化が抑止され、パフォーマンスが犠牲になります。

注 : -ms オプションは -ms0 オプションと等価です
コード・サイズ・レベルの番号を使用せずに -ms オプションを使用する場合、オプション・レベルはデフォルトで -ms0 になります。

3.5 ファイル・レベルの最適化の実行（-O3 オプション）

-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*	3-18
最適化情報ファイルを作成する。	-on*n*	3-19
複数のソース・ファイルをコンパイルする。	-pm	3-20

注 : コード・サイズを制御するために最適化レベルを引き下げない
コード・サイズを縮小しようとして最適化レベルを下げないでください。実際に、最適化レベルを引き下げるとコード・サイズが増大する場合があります。代わりに、-ms オプションを使用して、コード・サイズとパフォーマンスのトレードオフを制御してください。詳細は、3.4 節「コード・サイズの縮小方法（-ms オプション）」（3-17 ページ）を参照してください。

3.5.1 ファイル・レベルの最適化の制御方法（-ol*n* オプション）

-O3 オプションを指定してコンパイラを起動すると、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ている。	-ol0
標準ライブラリで宣言された関数を含んでいるが、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 オプションを使用する。-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

3.5.2 最適化情報ファイルの作成方法（-on*n* オプション）

-O3 オプションを指定してコンパイラ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 オプションを使用している。-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
冗長最適化情報ファイルを作成する。	-on2

3.6 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未適用のままになっている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・レベルの最適化を確認するには、-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、3.5.2 項「最適化情報ファイルの作成方法（-on*n* オプション）」（3-19 ページ）を参照してください。

Code Composer Studio で -pm オプションを使用すると、同じオプションをもつ C および C++ ファイルは一緒にコンパイルされます。ただし、プロジェクト全体のオプションとして選択されないファイル固有のオプションをもつファイルがある場合、そのファイルは個別にコンパイルされます。たとえば、プロジェクトにおけるすべての C および C++ ファイルがファイル固有の異なる一連のオプションをもつ場合は、プログラム・レベルの最適化が指定されていても、各ファイルは個別にコンパイルされます。すべての C および C++ ファイルを一緒にコンパイルするには、それらのファイルがファイル固有のオプションをもたないようにする必要があります。-ma のようなファイル固有のオプションを以前に使用した場合には、C および C++ ファイルを一緒にコンパイルするのは安全ではないかもしれないということをご承知ください。

注 : -pm および -k オプションを指定してのファイルのコンパイル

-pm および -k オプションを指定してすべてのファイルをコンパイルすると、コンパイラは .asm ファイルをただ 1 つ作成し、対応するソース・ファイルごとには作成しません。

3.6.1 プログラム・レベルの最適化の制御方法（-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宣言を、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 は、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の一覧です。

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る main を含む関数がなく、関数は FUNC_EXT_CALLED プラグマによって特定されていない。	-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	関数は FUNC_EXT_CALLED プラグマによって特定されている。	-op1 または -op2 のまま残る
-op3	任意の条件	-op3 のまま残る

-pm と -O3 を使用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、3.6.2 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」(3-22 ページ) を参照してください。

3.6.2　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

プログラムの中にアセンブリ関数が使用されている場合は、-pm オプションを使用するときに注意が必要です。コンパイラは C/C++ のソース・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 (7.7.6 項「FUNC_EXT_CALLED プラグマ」(7-23 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*n* 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 (3.6.1 項「プログラム・レベルの最適化の制御方法（-opn オプション)」(3-21 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++ 変数の変更も実行できません。

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知らせます。-op2 オプションについては、3.6.1 項「プログラム・レベルの最適化の制御方法（-opn オプション）」（3-21 ページ）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、コンパイラは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　次の両方の解決策を試して、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。

-op*n* オプションについては、3.6.1 項「プログラム・レベルの最適化の制御方法（-op*n* オプション）」（3-21 ページ）を参照してください。

状況　アプリケーションは、C/C++ ソース・コードとアセンブリ・ソース・コードで構成されています。アセンブリ関数は、C/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/C++ 関数が C/C++ から呼び出されることはありません。それらの C/C++ 関数は main と同じように機能し、C/C++ へのエントリ・ポイントとして機能します。

解決策　割り込みにより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、その後、-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・ポイント関数にこのプラグマを使用してください。そうしないと、コンパイラは先頭に FUNC_EXT_CALL プラグマが指定していないエントリ・ポイント関数を除去するかもしれません。

- -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、最適化があまり効果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。

3.7 特定のエイリアス指定技法を使用するかどうかの指定方法

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、間接参照では常に他方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。その上でコンパイラは、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。

以下の節では、コードで使用されるいくつかのエイリアス指定技法を説明します。これらの技法は ISO C 規格により有効であり、C6000 コンパイラによって受け入れられます。ただし、これらの技法はコンパイラがコードを完全に最適化することを妨げます。

3.7.1 特定のエイリアス使用時の -ma オプションの使用

コンパイラは、最適化付きで起動した場合、アドレスが引数として関数に渡される任意の変数が、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前提とします。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内で使用する場合は、コードを最適化する際に -ma オプションを使用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 オプションを使用します。

```
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;
}
```

3.7.2 エイリアス技法を使用しないことを示す -mt オプションの使用

-mt オプションはコンパイラに対して、ご使用のコードでエイリアスを使用する方法について特定の前提事項を指定することを知らせます。これらの前提事項により、コンパイラは最適化を改善できます。さらに -mt オプションは、ループ不変カウンタの増加および減少はゼロ以外であることを指定します。ループ不変とは、式の値がループ内で変化しないということです。

- -mt オプションは、3.7.1 項「特定のエイリアス使用時の -ma オプションの使用」（3-25 ページ）節で説明されているエイリアス指定技法をコードで使用しないことを指定します。コードでその技法を使用する場合は、-mt オプションを使用しないでください。ただし、-ma オプションを指定してコンパイルする必要があります。

  -ma オプションは -mt オプションと一緒に使用しないでください。使用すると -mt オプションが -ma オプションを上書きします。

- -mt オプションは、文字タイプを指すポインタが別のタイプのオブジェクトをエイリアス指定（ポイント）しないことを指定します。つまり、ISO 仕様の 3.3 節で指定されたこれらのタイプの一般的なエイリアス指定規則に対する特殊な例外は無視されます。以下の例とほぼ同じコードを使用する場合は、-mt オプションを使用しないでください。

```
{
    long  l;
    char *p = (char *) &l;

    p[2] = 5;
}
```

- -mt オプションは、P および Q の 2 つのポインタが実行時に同じ呼び出しで活動化された同じ関数の別々のパラメータである場合、P および Q の間接参照がエイリアスではないことを指定します。以下の例とほぼ同じコードを使用する場合は、-mt オプションを使用しないでください。

```
g(int j)
{
    int a[20];

    f(&a, &a)         /* Bad */
    f(&a+42, &a+j)    /* Also Bad */
}

f(int *ptr1, int *ptr2)
{
    ...
}
```

- -mt オプションは、配列参照 A[E1]..[En] 内の各添字式が、対応する宣言済み配列境界より小さい負ではない数値になることを指定します。以下の例とほぼ同じコードを使用する場合は、-mt オプションを使用しないでください。

```
static int ary[20][20];

int g()
{
    return f(5, -4); /* -4 is a negative index */
    return f(0, 96); /* 96 exceeds 20 as an index */
    return f(4, 16); /* This one is OK */
}

int f(int i, int j)
{
    return ary[i][j];
}
```

  この例では、ary[5][-4]、ary[0][96]、および ary[4][16] は同じメモリ位置をアクセスします。参照 ary[4][16] だけが -mt オプションで受け入れられます。これは、そのインデックスの両方が境界（0..19）内にあるからです。

- -mt オプションは、ループ不変カウンタの増加、およびループ・カウンタの減少はゼロ以外であることを指定します。ループ不変とは、式の値がループ内で変化しないということです。

ご使用のコードに、上記で説明したエイリアス指定技法が含まれていない場合は、-mt オプションを使用してコードの最適化を改善する必要があります。しかし -mt オプションは慎重に使用する必要があります。これらのエイリアス指定技法がコードに存在し、かつ -mt オプションが使用される場合には、予期しない結果が生じる可能性があります。

3.7.3 アセンブリ・オプティマイザでの -mt オプションの使用

-mt オプションにより、アセンブリ・オプティマイザは、リニア・アセンブリ内にメモリ・エイリアスがない、つまりメモリ参照が互いに依存しないことを前提とすることができます。しかし、アセンブリ・オプティマイザは .mdep 疑似命令で指定したメモリ依存関係をまだ認識します。.mdep 疑似命令の詳細は、4-21 ページと 4-44 ページを参照してください。

3.8 加法の浮動小数点演算の並べ換えの防止

コンパイラは、加法の浮動小数点演算を自由に並べ換えます。コンパイラに加法の浮動小数点演算を並べ換えさせたくない場合は、-mc オプションを使用してください。-mc オプションを指定するとパフォーマンスが低下する可能性があります。

3.9 最適化コード内で asm 文を使用する場合の注意

最適化コードの中で 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元の 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。

3.10 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 で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* オプションは次の方法で使用できます。

- *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限界として、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 を加える）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 オプションで作成する）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* オプションを使用しない場合、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 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 オプションを使用する必要があります。-O3 オプションは、その他の最適化をオンにします。-O3 オプションを使用し、自動インライン展開を行いたくない場合は、-oi0 オプションと -O3 オプションを組み合わせて使用します。

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場合、特に多数の場所で呼び出される関数をインライン展開するとコード・サイズが増加します。関数のインライン展開は、あまり呼び出されない関数やサイズが小さい関数に適しています。インライン展開によりコード・サイズが増加するのを防止するためには、-oi0 オプションと -pi オプションを使用します。これらのオプションを使用すると、コンパイラは組み込み関数だけをインライン展開します。まだコード・サイズが大きすぎると思われる場合は、3.4 節「コード・サイズの縮小方法（-ms オプション）」（3-17 ページ）を参照してください。

3.11 最適化でのインターリスト機能の使用方法

-os および -ss オプションを指定して、最適化（-O*n* オプション）付きでコンパイルすると、インターリスト機能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使用すると、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。

最適化で -os オプションを使用した場合､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の中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使用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 オプションを使用した場合、コンパイラは再構築後の C/C++ の文を書き込みます。

例 3-2 は、最適化（-O2）と -os オプションを指定してコンパイルした例 2–3（2-47 ページ）の関数を示しています。アセンブリ・ファイルでは、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。

注 : パフォーマンスとコード・サイズに与える影響
-ss オプションはパフォーマンスとコード・サイズに悪影響を与える可能性があります。

例 3-2.　-O2 および -os オプションを指定してコンパイルした 例 2–3 の関数

```
_main:
;** 5  ------------------------    printf("Hello, world\n");
;** 6  ------------------------    return 0;
           STW     .D2     B3,*SP--(12)
    .line 3
           B       .S1     _printf
           NOP             2
           MVKL    .S1     SL1+0,A0
           MVKH    .S1     SL1+0,A0
||         MVKL    .S2     RL0,B3
           STW     .D2     A0,*+SP(4)
||         MVKH    .S2     RL0,B3
RL0:; CALL OCCURS
    .line 4
           ZERO    .L1     A4
    .line 5
           LDW     .D2     *++SP(12),B3
           NOP             4
           B       .S2     B3
           NOP             5
           ; BRANCH OCCURS
    .endfunc  7,000080400h,12
```

最適化で -ss オプションと -os オプションを指定すると、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++ ソースをアセンブリ・ファイルにマージします。

例 3-3 は、最適化（-O2）と -ss および -os オプションを指定してコンパイルした例 2–3（2-47 ページ）の関数を示しています。アセンブリ・ファイルでは、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。

例 3-3.　例 2–3 の関数を -O2、-os、および -ss オプションを指定してコンパイルした結果

```
_main:
;** 5  -----------------------    printf("Hello, world\n");
;** 6  -----------------------    return 0;
           STW     .D2     B3,*SP--(12)
;----------------------------------------------------------------------------
;   5 | printf("Hello, world\n");
;----------------------------------------------------------------------------
           B       .S1     _printf
           NOP             2
           MVKL    .S1     SL1+0,A0
           MVKH    .S1     SL1+0,A0
||         MVKL    .S2     RL0,B3
           STW     .D2     A0,*+SP(4)
||         MVKH    .S2     RL0,B3
RL0:; CALL OCCURS
;----------------------------------------------------------------------------
;   6 | return 0;
;----------------------------------------------------------------------------
           ZERO    .L1     A4
           LDW     .D2     *++SP(12),B3
           NOP             4
           B       .S2     B3
           NOP             5
           ; BRANCH OCCURS
```

3.12 最適化されたコードのデバッグ方法とプロファイル方法

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、レジスタに対しての変数が多対多で割り振られるため、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）オプションまたは --symdebug:coff オプション（STABS デバッグ）を指定して作成されたコードのプロファイルも、パフォーマンスを大幅に低下させる可能性があるためお勧めしません。これらの問題に対処するためには、以下の節で説明され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。

3.12.1 最適化されたコードのデバッグ方法（--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 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 つと一緒に使用します。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、多くのコンパイラ最適化が使用できなくなります。-O オプション（最適化を起動する）を --symdebug:dwarf または --symdebug:coff オプションと組み合わせて使用すると、デバッグに対応できる最適化の多くが実行できるようになります。

コード内のループのデバッグに問題がある場合、-mu オプションを使用してソフトウェア・パイプラインをオフにできます。詳細は、3.2.1 項「ソフトウェア・パイプライン化の取り止め（-mu オプション）」（3-5 ページ）を参照してください。

注 : シンボリック・デバッグ・オプションはパフォーマンスとコード・サイズに影響を与えます

--symdebug:dwarf または -symdebug:coff オプションを使用すると、コードのパフォーマンスが大幅に低下するか、またはコード・サイズが大きくなる可能性があります。これらのオプションは、デバッグだけに使用してください。プロファイル作成時に --symdebug:dwarf または -symdebug:coff を使用することはお勧めしません。

3.12.2 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・オプションなしで最適化（-O0 ～ -O3）を使用します。デフォルトで、コンパイラは、最適化、コード・サイズ、またはパフォーマンスに影響を与えないような最低限のデバッグ情報を生成します。

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は、--profile:breakpt オプションを -O オプションと一緒に使用します。--profile:breakpt オプションは、ブレークポイント・ベースのプロファイラを使用するときに、誤った動作を引き起こす可能性のある最適化を抑止します。

Code Composer Studio の関数レベルで、より微細に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 オプションを使用することができます。しかし、この方法は推奨できません。コンパイラは --symdebug:dwarf または -symdebug:coff を指定すると一部の最適化を使用できないため、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 の外部で clock( ) 関数を使用することをお勧めします。

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、関数の先頭と最後にだけプロファイル・ポイントを設定できます。

3.13 実行できる最適化の種類

TMS320C6000 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、サイズを小さくします。

コンパイラによって行われる最適化は、次のとおりです。

最適化	ページ
コストに基づいたレジスタ割り当て	3-36
エイリアスの明確化	3-38
分岐の最適化と制御フローの簡略化	3-38
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	3-41
式の簡略化	3-41
関数のインライン展開	3-42
誘導変数の最適化と強度換算	3-43
ループ不変コードの移動	3-44
ループの循環	3-44
レジスタ変数	3-44
レジスタのトラッキング／ターゲッティング	3-44
ソフトウェア・パイプライン	3-45

3.13.1 コストに基づいたレジスタ割り当て

コンパイラは、最適化が有効にされた場合、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、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。

誘導変数の除去とループ・テストの置換を使用すると、コンパイラはループを単純なカウントするループとして認識でき、そのループをソフトウェア・パイプライン化、展開、または除去できます。強度換算は、配列参照をオート・インクリメントを使用した効率的なポインタ参照にします。

例 3-4. 強度換算、誘導変数除去、レジスタ変数、およびソフトウェア・パイプライン化

(a) C ソース

```
int a[10];

main()
{
    int i;

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

例 3–4. 強度換算、誘導変数除去、レジスタ変数、およびソフトウェア・パイプライン化（続き）

（b）コンパイラ出力

```
FP  .set   A15
DP  .set   B14
SP  .set   B15

;   opt6x -O2 j3_32.if j3_32.opt
    .sect ".text"
    .global  _main
_main:
;**------------------------------------------------------------*
           MVK     .S1     _a,A0
           MVKH    .S1     _a,A0

           MV      .L2X    A0,B4
||         ZERO    .L1     A3
||         ZERO    .D2     B5
||         MVK     .S2     2,B0          ; |7|
;**------------------------------------------------------------*
L2:; PIPED LOOP PROLOG
   [ B0]   B       .S1     L3            ; |7|
   [ B0]   B       .S1     L3            ;@ |7|
   [ B0]   B       .S1     L3            ;@@ |7|

   [ B0]   B       .S1     L3            ;@@@ |7|
|| [ B0]   SUB     .L2     B0,2,B0       ;@@@@ |7|

           ADD     .S2     8,B4,B4       ; |8|
|| [ B0]   B       .S1     L3            ;@@@@ |7|
|| [ B0]   SUB     .L2     B0,2,B0       ;@@@@@ |7|
;**------------------------------------------------------------*
L3:; PIPED LOOP KERNEL
           STW     .D1T1   A3,*A0++(8)   ; |8|
||         STW     .D2T2   B5,*-B4(4)    ; |8|
||         ADD     .S2     8,B4,B4       ;@ |8|
|| [ B0]   B       .S1     L3            ;@@@@@ |7|
|| [ B0]   SUB     .L2     B0,2,B0       ;@@@@@@ |7|
;**------------------------------------------------------------*
L4:; PIPED LOOP EPILOG
;**------------------------------------------------------------*
           B       .S2     B3            ; |9|
           NOP             5
           ; BRANCH OCCURS ; |9|

    .global   _a
    .bss    _a,40,4
```

3.13.2 エイリアスの明確化

一般に、C および 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（小文字の L）値（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。

3.13.3 分岐の最適化と制御フローの簡略化

コンパイラは、プログラムの分岐先を解析し、オペレーションを直線的なシーケンス（基本ブロック）に再配置することにより、分岐や冗長な条件を除去します。アクセスが不可能なコードは削除され、分岐への分岐はバイパスされ、無条件分岐を介した条件付き分岐は単一の条件付き分岐に簡略化されます。

条件の値が（複写伝播またはその他のデータ・フロー解析を通じて）コンパイル時に決定される場合、コンパイラは条件付き分岐を削除できます。switch 文の case リストも条件付き分岐と同じ方法で解析され、場合によっては全面的に除去されます。単純な制御フロー構造が条件付き命令に簡略化され、分岐の必要が全くなくなります。

例 3-5 の単純な有限状態機械の例では、switch 文および state 変数が完全に最適化された後に一連の再編成された条件付き分岐が残ります。

例 3-5.　制御フローの簡略化と複写伝播

(a) C ソース

```
fsm()
{
    enum { ALPHA, BETA, GAMMA, OMEGA } state = ALPHA;
    int *input;
    while (state != OMEGA)
        switch (state)
        {
          case ALPHA:   state = (*input++ == 0) ?BETA:GAMMA; break;
          case BETA:    state = (*input++ == 0) ?GAMMA:ALPHA; break;
          case GAMMA:   state = (*input++ == 0) ?GAMMA:OMEGA; break;
        }
}

main()
{
    fsm();
}
```

例 3–5. 制御フローの簡略化と複写伝播（続き）

（b）コンパイラ出力

```
FP  .set   A15
DP  .set   B14
SP  .set   B15
;  OPT6X.EXE -O3 fsm.if fsm.opt
   .sect  ".text"
   .global   _fsm
;******************************************************************************
;* FUNCTION NAME:_fsm                                                         *
;*                                                                            *
;*   Regs Modified     :B0,B4                                                 *
;*   Regs Used         :B0,B3,B4                                              *
;*   Local Frame Size  :0 Args + 0 Auto + 0 Save = 0 byte                     *
;******************************************************************************
_fsm:
;** --------------------------------------------------------------------------*
;** --------------------------------------------------------------------------*
L2:
           LDW     .D2T2   *B4++,B0     ; |8|
;** --------------------------------------------------------------------------*
L3:
           NOP             4
   [ B0]   B       .S1     L7           ; |8|
           NOP             4
   [ B0]   LDW     .D2T2   *B4++,B0     ; |10|
           ; BRANCH OCCURS ; |8|
;** --------------------------------------------------------------------------*
           LDW     .D2T2   *B4++,B0     ; |9|
           NOP             4
   [ B0]   B       .S1     L3           ; |9|
           NOP             4
   [ B0]   LDW     .D2T2   *B4++,B0     ; |8|
           ; BRANCH OCCURS ; |9|
;** --------------------------------------------------------------------------*
L5:
           LDW     .D2T2   *B4++,B0     ; |10|
;** --------------------------------------------------------------------------*
L6:
;** --------------------------------------------------------------------------*
L7:
           NOP             4
   [!B0]   B       .S1     L6           ; |10|
           NOP             4
   [!B0]   LDW     .D2T2   *B4++,B0     ; |10|
           ; BRANCH OCCURS ; |10|
;** --------------------------------------------------------------------------*
           B       .S2     B3           ; |12|
           NOP             5
           ; BRANCH OCCURS ; |12|
```

3.13.4 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。最適化が指定された場合、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）の両面で行います。

- **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。この値には、別の変数、定数、または共通部分式を指定できます。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 と例 3-6 を参照）。

- **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、それを保存し、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、変数への不要な代入（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）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 3-6 を参照）。

3.13.5 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 3-6 を参照）。

例 3-6 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。a と b への代入は除去され、それらの値が戻されます。

例 3-6.　データ・フローの最適化と式の簡略化

(a) C ソース

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

(b) コンパイラ出力

```
FP   .set    A15
DP   .set    B14
SP   .set    B15

;    opt6x -O2 t1.if t1.opt
     .sect         ".text"
     .global       _simplify

_simplify:
   B         .S2      B3
   NOP                2
   MPY       .M1      5,A4,A0
   NOP                1
   EXT       .S1      A0,24,24,A4
   ; BRANCH OCCURS
```

3.13.6 関数のインライン展開

コンパイラは、小さな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 3-7 を参照）。

例 3-7 では、コンパイラは C 関数 plus() 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 3-7. 関数のインライン展開

(a) C ソース

```
int plus (int x, int y)
{
    return x + y;
}
main ()
{
    int a = 3;
    int b = 4;
    int c = 5;

    return plus (a, plus (b, c));
}
```

(b) コンパイラ出力

```
FP  .set  A15
DP  .set  B14
SP  .set  B15

;   opt6x -O3 t2.if t2.opt
    .sect     ".text"
    .global   _main;

_main:
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>ENTERING plus()
;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<LEAVING plus()
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>ENTERING plus()
;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<LEAVING plus()
       B  .S2      B3
       NOP         4
       MVK.S1      12,A4
       ; BRANCH OCCURS
```

3.13.7 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 (例 3-4 (3-36 ページ) を参照)。

3.13.8 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を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。

3.13.9 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により除去されます。

3.13.10 レジスタ変数

コンパイラは、ローカル変数、パラメータ、および一時値の保管のためにレジスタを最大限に使用します。レジスタに保管された変数にアクセスするのは、メモリ内の変数にアクセスするよりも効率的です。レジスタ変数は、ポインタに特に有効です（例 3-4（3-36 ページ）を参照）。

3.13.11 レジスタのトラッキング／ターゲッティング

コンパイラは、値がすぐに再度使用される場合に値を再ロードしなくても済むように、レジスタの内容を追跡します。変数、定数、および構造体の参照（たとえば（a.b））は、直線的コードを介して追跡されます。またレジスタのターゲッティングも、レジスタ変数への代入または関数からの値の戻りの場合のように、必要に応じて式を計算して特定のレジスタに直接、値を入れます（例 3-8（3-45 ページ）を参照）。

例 3-8.　レジスタのトラッキング／ターゲッティング

（a）C ソース

```
int x, y;

main()
{
   x += 1;
   y = x;
}
```

（b）コンパイラ出力

```
FP  .set  A15
DP  .set  B14
SP  .set  B15

;   opt6x -O2 t3.if t3.opt
    .sect     ".text"
    .global   _main

_main:
          LDW  .D2    *+B14(_x),B4
          NOP         1
          B    .S2    B3
          NOP         2
          ADD  .L2    1,B4,B4
          STW  .D2    B4,*+B14(_y)
          STW  .D2    B4,*+B14(_x)
          ; BRANCH OCCURS

    .global   _x
    .bss      _x,4,4
    .global   _y
    .bss      _y,4,4
```

3.13.12ソフトウェア・パイプライン

ソフトウェア・パイプラインとは、ループの複数の反復を並列に実行するようにループからスケジュールするのに使用される技法です。詳細は、3.2 節「ソフトウェア・パイプライン化による最適化」（3-4 ページ）を参照してください。

第 4 章

アセンブリ・オプティマイザの使用方法

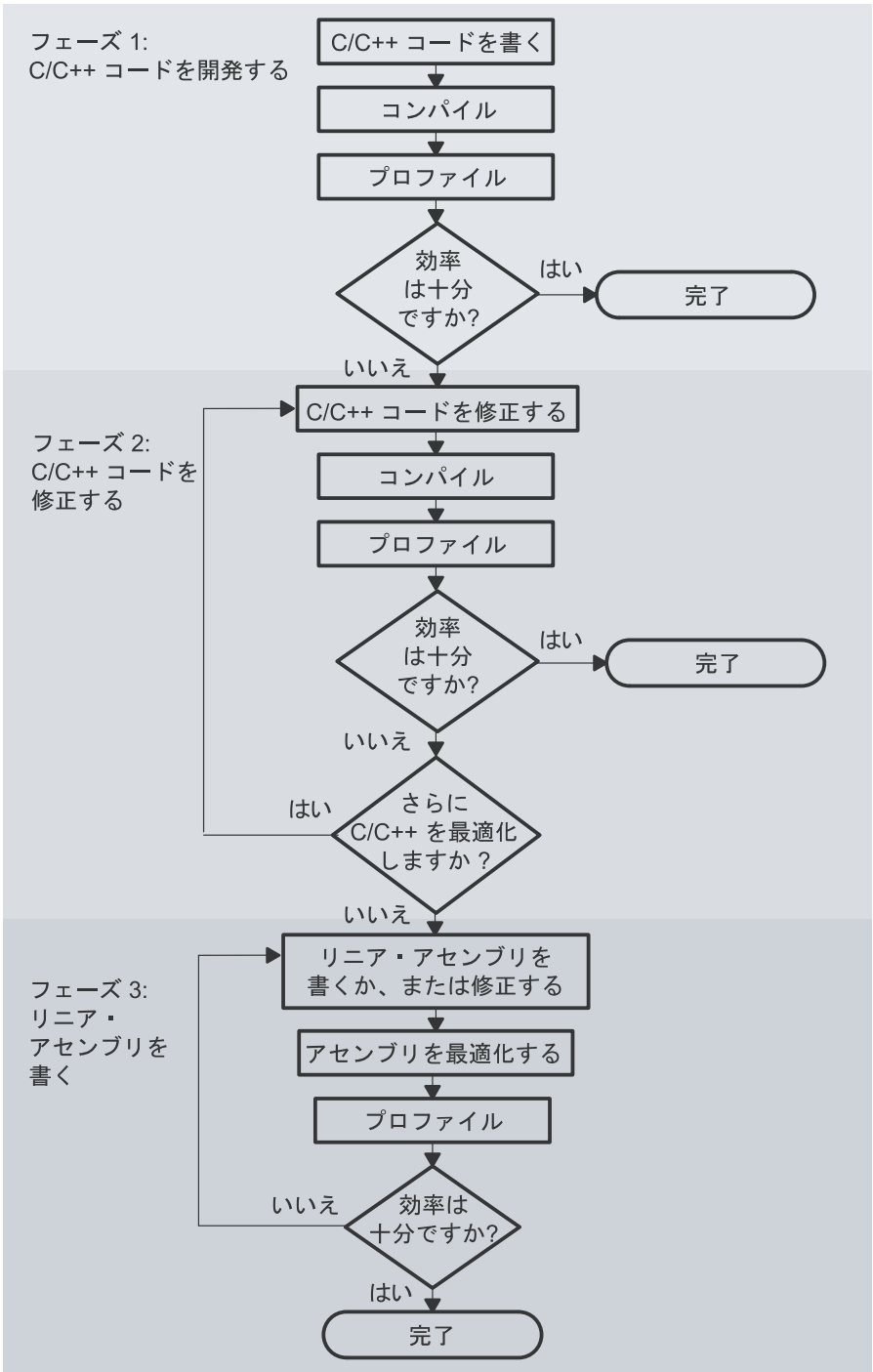
アセンブリ・オプティマイザを使用すると、C6000 のパイプライン構造やレジスタ割り当てを意識せずにアセンブリ・コードを書くことができます。アセンブリ・オプティマイザは、リニア・アセンブリ・コードを受け入れます。これは、レジスタ割り当てが実行済みになっているかもしれませんが、スケジュールはまだされていないアセンブリ・コードのことです。アセンブリ・オプティマイザは、レジスタの割り当てやループ最適化を行って、リニア・アセンブリを高度にパラレル・アセンブリへ変換します。

項目 **ページ**

4.1 パフォーマンスを改善するためのコード開発フロー 4-2
4.2 アセンブリ・オプティマイザの概要 4-4
4.3 リニア・アセンブリを記述するために必要な知識 4-4
4.4 アセンブリ・オプティマイザ疑似命令 4-13
4.5 アセンブリ・オプティマイザを使用したメモリ・バンク競合の回避方法 4-33
4.6 メモリ・エイリアスの明確化 4-43

4.1 パフォーマンスを改善するためのコード開発フロー

ユーザがプログラムを書く際やデバッグしようとする場合、次のフローに従って作業を進めると、C6000 コードのパフォーマンスを最大にできます。



C6000 のコードの開発には、次の 3 つのフェーズがあります。

- **フェーズ 1：C/C++ で書く**

  フェーズ 1 の C/C++ コードの開発には、C6000 の知識は不要です。-g オプションを指定してスタンドアロン・シミュレータを使用し（6.4 節「スタンドアロン・シミュレータのプロファイル機能の使用方法」（6-8 ページ）を参照）、C/C++ コード内の効率が悪い領域を特定します。コードのパフォーマンスを改善するにはフェーズ 2 に進みます。

- **フェーズ 2：C/C++ コードを修正する**

  フェーズ 2 では、本書で説明されている組み込み関数とコンパイラ・オプションを使用して C/C++ コードを改善します。-g オプションを指定してスタンドアロン・シミュレータを使用し、変更されたコードのパフォーマンスをチェックします。C/C++ コード改良のヒントは、TMS320C6000 Programmer's Guide を参照してください。コードの効率がまだ十分でない場合は、フェーズ 3 に進みます。

- **フェーズ 3：リニア・アセンブリを書く**

  このフェーズでは、時間が重要な領域を C/C++ コードから抽出し、コードをリニア・アセンブリで書き直します。アセンブリ・オプティマイザを使用すると、このコードを最適化できます。リニア・アセンブリで最初にコードを作成する場合は、パイプライン構造やレジスタの割り当てを意識する必要はありません。後でリニア・アセンブリ・コードを修正する場合、レジスタの分割化などの詳細をコードに追加する場合があります。

  この段階でパフォーマンスを改善するには、フェーズ 2 より多くの時間がかかるので、フェーズ 3 に進む前にできるだけコードを修正しておいてください。その後、さらに小さいコード・セクションをこのフェーズで処理します。

4.2 アセンブリ・オプティマイザの概要

使用可能なすべての C/C++ 最適化を適用した後でも C/C++ コードのパフォーマンスにまだ満足できない場合、アセンブリ・オプティマイザを使用すると、C6000 のアセンブリ・コードを簡単に記述することができます。

アセンブリ・オプティマイザは、次のものを含めて複数のタスクを実行します。

- ❏ C6000 の命令レベルのパラレリズムを使用して、パフォーマンスを最大にするように命令をスケジュールする。
- ❏ 命令が C6000 のレイテンシ要件に準拠していることを確認する。
- ❏ ソース・コードにレジスタを割り当てる。

C/C++ コンパイラのように、アセンブリ・オプティマイザはソフトウェア・パイプライン化を実行します。ソフトウェア・パイプライン化とは、ループの複数の反復を並行して実行できるように、ループからの命令をスケジュールする技法です。コード生成ツールは、ユーザの入力、およびプログラムから収集する情報を使用してコードのソフトウェア・パイプライン化を試みます。詳細は、3.2 節「ソフトウェア・パイプライン化による最適化」（3-4 ページ）を参照してください。

アセンブリ・オプティマイザを起動するには、コンパイラ・プログラム（cl6x）を使用します。入力ファイルのいずれかに 拡張子 *.sa* がある場合、アセンブリ・オプティマイザはコンパイラ・プログラムによって自動的に起動されます。リニア・アセンブリ・ファイルと一緒に C/C++ ソース・ファイルを指定できます。コンパイラ・プログラムの詳細は、2.1 節「コンパイラの概要」（2-2 ページ）を参照してください。

4.3 リニア・アセンブリを記述するために必要な知識

C6000 プロファイル・ツールを使用すると、リニア・アセンブリとして書き直しが必要なコードの中の時間が重要なセクションを特定できます。アセンブリ・オプティマイザ用に書くソース・コードは、アセンブリ・ソース・コードとほぼ同じです。ただし、リニア・アセンブリ・コードは、分割し、スケジュールし、またはレジスタを割り当てる必要はありません。この目的は、ユーザに代わってアセンブリ・オプティマイザにこの情報を判別させることです。リニア・アセンブリ・コードを書く場合は、次の項目についての知識が必要です。

- ❏ **アセンブリ・オプティマイザ疑似命令**

  リニア・アセンブリ・ファイルは、アセンブリ・オプティマイザ・コードと通常のアセンブリ・ソースとの組み合わせにすることができます。アセンブリ・オプティマイザ・コードを通常のアセンブリ・コードと区別し、コードについての追加情報をアセンブリ・オプティマイザに提供するには、アセンブリ・オプティマイザの疑似命令を使用します。アセンブリ・オプティマイザ疑似命令については、4.4 節「アセンブリ・オプティマイザ疑似命令」（4-13 ページ）を参照してください。

❑ アセンブリ・オプティマイザの動作に影響を与えるオプション

次のコンパイラ・オプションは、アセンブリ・オプティマイザの動作に影響を与えます。

オプション	機能	ページ
-el	アセンブリ・オプティマイザ・ソース・ファイルのデフォルトの拡張子を変更します。	2-21
-fl	アセンブリ・オプティマイザ・ソース・ファイルの識別方法を変更します。	2-20
-k	アセンブリ言語（.asm）ファイルを保存します。	2-16
-mi*n*	割り込みしきい値を定義します。	2-43
-ms*n*	コード・サイズを 4 つのレベル（-ms0、-ms1、-ms2、-ms3）で制御します。	3-17
-mt	メモリのエイリアス指定を推定しません。	3-27
-mu	ソフトウェア・パイプライン化を取り止めます。	3-5
-mv*n*	ターゲット・バージョンを選択します。	2-17
-mw	冗長ソフトウェア・パイプライン情報を生成します。	3-5
-n	コンパイルまたはアセンブリ最適化だけを実行します（アセンブルしません）。	2-16
-o*n*	最適化のレベルを上げます（-o0、-o1、-o2、および -o3）。	3-2
-q	進捗メッセージを抑止します。	2-16
--speculate_loads=*n*	アドレス範囲がバインドされたロードの見込み実行を可能にします。	3-14

❑ TMS320C6000 命令

リニア・アセンブリ・コードを記述しているときは、コード中で次の事項を指示する必要はありません。

- パイプライン・レイテンシ
- レジスタの使用状況
- 使用されるユニット

他のコード生成ツールと同様に、パフォーマンスに納得できるまでリニア・アセンブリ・コードを修正しなければならない場合があります。その場合、ユーザはリニア・アセンブリに詳細項目を追加する必要があるかもしれません。たとえば、いくつかのレジスタを分割または割り当てたい場合があります。

注 : スケジュールされたアセンブリ・コードをソースとして使用しないでください

アセンブリ・オプティマイザは、入力ファイルの命令が、ユーザが実行したい論理的順序（つまり、リニア・アセンプリ・コード）で記述されているものと想定します。並列命令は不正となります。これに対して、アセンブラは、ユーザがパイプライン・レイテンシによる遅延スロットを考慮する位置に命令を記述してあるものと想定します。したがって、アセンブラ用に記述したコード（つまり、スケジュールされたアセンブリ・コード）、またはアセンブリ・オプティマイザの出力をアセンブリ・オプティマイザの入力として使用することは妥当ではありません。

❑ リニア・アセンブリ・ソース文の構文

リニア・アセンブリ・ソース・プログラムは、アセンブリ・オプティマイザ疑似命令、アセンブリ言語命令、およびコメントを含む複数のソース文で構成されています。ソース文の要素の詳細については、4.3.1 項「リニア・アセンブリ・ソース文の形式」（4-7 ページ）を参照してください。

❑ レジスタまたはレジスタ・サイドの指定

レジスタは、ユーザ・シンボルに明示的に割り当てることができます。あるいは、コンパイラが実際のレジスタ割り当てを行うようにしたまま、シンボルを A サイドまたは B サイドに割り当てることができます。レジスタの指定の詳細は、4.3.2 項「リニア・アセンブリのレジスタ指定方法」（4-8 ページ）を参照してください。

❑ 機能ユニットの指定

機能ユニット指定子の使用は、通常のアセンブリ・コードとリニア・アセンブリ・コードの両方でオプションです。機能ユニットを指定すると、レジスタ・ファイルのどちらのサイドを命令に使用するかを制御できます。これは、アセンブリ・オプティマイザが機能ユニットとレジスタ割り当てを実行するのに役立ちます。この方法は陳腐化し、レジスタの指定が優先されます。機能ユニットの指定の詳細は、4.3.3 項「リニア・アセンブリの機能ユニット指定方法」（4-10 ページ）を参照してください。

❑ ソース・コメント

アセンブリ・オプティマイザは、入力リニア・アセンブリからの命令に関するコメントを出力ファイルに付加します。2 組の <x, y> をコメントに付加して、命令がソフトウェア・パイプラインにおいてループのどの反復とサイクルに属するかを指定します。ゼロ（0）を基点とする数値 x は、カーネルの最初の実行時の命令の反復を表します。ゼロ（0）を基点とする数値 y は、ループの 1 回の反復内に命令がスケジュールされるサイクルを表します。ソース・コメントとアセンブリ・オプティマイザの出力例については、4.3.4 項「リニア・アセンブリ・ソース・コメントの使用方法」（4-11 ページ）を参照してください。

4.3.1 リニア・アセンブリ・ソース文の形式

ソース文には、順番に並べられた 5 つのフィールド（ラベル、ニーモニック、ユニット指定子、オペランド・リスト、およびコメント）を指定できます。ソース文の一般の構文は、次のとおりです。

[*label*[:]] [**[***register***]**] *mnemonic* [*unit specifier*] [*operand list*] [;*comment*]

label[:] すべてのアセンブリ言語命令、および大部分の（すべてではない）アセンブリ・オプティマイザ疑似命令の場合、ラベルはオプションです。ラベルを使用する場合、ラベルはソース文のカラム 1 から始まる必要があります。ラベルの後ろにコロンを続けることができます。

[*register***]** 条件付き命令は、大括弧（[]）で囲まれます。マシン命令ニーモニックは、括弧内のレジスタの値に基づいて実行されます。有効なレジスタ名は、A0（C6400 の場合のみ）、A1、A2、B0、B1、B2、または記号です。

mnemonic ニーモニックは、マシン命令（たとえば、ADDK、MVKH、B）、またはアセンブリ・オプティマイザ疑似命令（たとえば、.proc、.trip）です。

unit specifier オプションのユニット指定子を使用すると、機能ユニット・オペランドを指定できます。指定されたユニット・サイドだけが使用され、その他の指定は無視されます。優先方法はレジスタ・サイドの指定です。

operand list オペランド・リストは、すべての命令または疑似命令に必要なわけではありません。オペランドにはシンボル、定数、または式を指定することができ、コンマで区切る必要があります。

comment コメントはオプションです。カラム 1 から始まるコメントは、セミコロンまたはアスタリスクで始める必要があります。他のカラムから始まるコメントはセミコロンで始める必要があります。

C6000 アセンブリ・オプティマイザは、1 行当たり最高 200 文字を読み取ります。200 を超える文字は、すべて切り捨てられます。正しいアセンブリのためには、ソース文の作動部分（つまりコメント以外のすべて）の長さを 200 文字未満にしてください。コメントはその文字限界を超えて拡張できますが、切り捨てられた部分は .asm ファイル内に含まれません。

リニア・アセンブリ・コードを作成する際には、次のガイドラインに従ってください。

- すべての文は、ラベル、ブランク、アスタリスク、またはセミコロンで始める必要があります。
- ラベルはオプションです。ラベルを指定する場合はカラム 1 から始めなければなりません。
- 各フィールド間は、1 つまたは 2 つ以上のブランクで区切る必要があります。タブ文字はブランクとして解釈されます。オペランド・リストと先行フィールドは、ブランクで区切る必要があります。

- ❏ コメントはオプションです。コメントをカラム 1 から 始める場合は、アスタリスクまたはセミコロン（* または ;）で始めます。カラム 2 以降から始める場合は、セミコロン（ ; ）で始めなければなりません。
- ❏ 条件付き命令をセットアップする場合、レジスタは大括弧で囲みます。
- ❏ ニーモニックはカラム 1 から始めることはできません。カラム 1 から始まるニーモニックは、ラベルとして解釈されてしまいます。

条件つき命令、ラベル、およびオペランドを含んでいる C6000 命令の構文については、TMS320C6000 Assembly Language Tools User's Guide を参照してください。

4.3.2 リニア・アセンブリのレジスタ指定方法

C6000 には、2 つしかクロス・パスがありません。このため、C6000 は 1 サイクル当たり各データ・パスの対向レジスタ・ファイルから読み取られるソースが 1 つに制限されます。コンパイラは、命令ごとに 1 つのサイドを選択しなければなりません。これは分割化と呼ばれます。

最初は、リニア・アセンブリ・ソース・コードを手作業で分割しないことをお勧めします。これにより、コンパイラが自由にコードを分割し、最適化できるようになります。コンパイラがソフトウェア・パイプライン・ループの最適な分割化を検出しない場合、機能ユニット指定子を使用して多くの命令を手作業で分割することで、最適な分割化を実行することができます。リニア・アセンブリ・ソース・コードに機能ユニット指定子を追加すると、これらの命令および後続の命令の進行先に関する情報がコンパイラに与えられます。

.reg 疑似命令で分割化用のレジスタ、または .rega および .regb 疑似命令でレジスタ・サイドを選択します（4-28 ページを参照）。

.reg 疑似命令を使用すると、レジスタに保管されている値に対して記述名を使用できます。アセンブリ・オプティマイザは、その値で動作する命令に対して選択された機能ユニットとレジスタの用途とが一致するように、ユーザに代わってレジスタを選択します。.reg 疑似命令の詳細と例については、4-26 ページを参照してください。

レジスタは 2 つの疑似命令を使用して直接分割できます。**.rega** 疑似命令は、シンボリック名を A サイドのレジスタに制約するために使用します。**.regb** 疑似命令は、シンボリック名を B サイドのレジスタに制約するために使用します。.rega および .regb 疑似命令の詳細は、4-28 ページを参照してください。

例 4-1 は、内積を計算する手書きのリニア・アセンブリ・プログラムです。これを、C コードを例示している 例 4-2 と比較してください。

例 4-1.　内積の計算用のリニア・アセンブリ・コード

```
_dotp:.cproc a_0, b_0

       .rega     a_4, tmp0, sum0, prod1, prod2
       .regb     b_4, tmp1, sum1, prod3, prod4
       .reg      cnt, sum
       .reg      val0, val1

       ADD    4, a_0, a_4
       ADD    4, b_0, b_4
       MVK    100, cnt
       ZERO   sum0
       ZERO   sum1

loop:.trip   25
       LDW          *a_0++[2], val0      ; load a[0-1]
       LDW          *b_0++[2], val1      ; load b[0-1]
       MPY          val0, val1, prod1    ; a[0] * b[0]
       MPYH         val0, val1, prod2    ; a[1] * b[1]
       ADD          prod1, prod2, tmp0   ; sum0 += (a[0]*b[0]) +
       ADD          tmp0, sum0, sum0     ;         (a[1]*b[1])

       LDW          *a_4++[2], val0      ; load a[2-3]
       LDW          *b_4++[2], val1      ; load b[2-3]
       MPY          val0, val1, prod3    ; a[2] * b[2]
       MPYH         val0, val1, prod4    ; a[3] * b[3]
       ADD          prod3, prod4, tmp1   ; sum1 =+ (a[2]*b[2]) +
       ADD          tmp1, sum1, sum1     ;         (a[3]*b[3])

[cnt]  SUB    cnt, 4, cnt        ; cnt -= 4
[cnt]  B      loop               ; if (cnt!=0) goto loop

       ADD    sum0, sum1, sum    ; compute final result

       .return sum
       .endproc
```

例 4-2 は、内積の計算用に改良された C コードです。

例 4-2.　内積の計算用の C コード

```
int dotp(short a[], short b[])
{
    int sum0 = 0;
    int sum1 = 0;

    int sum, i;

    for (i = 0; i < 100/4; i +=4)
    {
        sum0 += a[i]   * b[i];
        sum0 += a[i+1] * b[i+1];

        sum1 += a[i+2] * b[i+2];
        sum1 += a[i+3] * [b[i+3];
    }
    sum = sum0 + sum1;
    return sum;
}
```

命令を分割して間接的にレジスタを分割する古い方法は、引き続き使用できます。サイドおよび機能ユニット指定子も命令で引き続き使用できます。ただし、機能ユニット指定子（.L/.S/.D/.M）は無視されます。サイド指定子は、対応するシンボリック名（ある場合）の分割化の制約に変換されます。たとえば次のとおりです。

```
MV  .1     x, y     ; translated to .REGA y
LDW .D2T2 *u, v:w   ; translated to .REGB u, v, w
```

4.3.3　リニア・アセンブリの機能ユニット指定方法

機能ユニットの指定方法は、レジスタを直接分割する機能により非推奨方法となりました。（詳細は、4.3.2 項「リニア・アセンブリのレジスタ指定方法」（4-8 ページ）を参照してください。）リニア・アセンブリ内のユニット指定子フィールドは使用できますが、コンパイラはレジスタ・サイドの情報のみを使用します。

機能ユニットを指定するには、アセンブラ命令の後にピリオド（.）、および機能ユニット指定子を続けます。1 つの命令サイクル内の機能ユニットごとに、1 つずつの命令を割り当てることができます。機能タイプごとに、2 つずつで合計 8 つの機能ユニット、および 2 つのアドレス・パスがあります。各機能タイプの 2 つの機能ユニットは、A または B がそれぞれ使用するデータ・パスにより区別されます。

.D1 および **.D2**　　データ／加算／減算の演算

.L1 および **.L2**　　論理演算装置（ALU）／比較／長精度データ演算

.M1 および **.M2**　　乗算の演算

.S1 および **.S2**　　シフト／ ALU ／ブランチ／フィールドの演算

.T1 および **.T2**　　アドレス・パス

リニア・アセンブリ内のユニット指定子フィールドを入力するには、次のようにいくつかの方法があります。これらの内、特定のレジスタ・サイド情報だけが認識され、使用されます。

- ❏ 特定の機能ユニット（たとえば .D1）を指定できます。
- ❏ .D1 または .D2 機能ユニットの後ろに T1 または T2 を続けて指定して、非メモリ・オペランドが特定のレジスタ・サイドにあることを指定できます。T1 はサイド A を指定し、T2 はサイド B を指定します。たとえば、次のとおりです。

```
LDW  .D1T2    *A3[A4], B3
LDW  .D1T2    *src, dst
```

- ❏ 機能タイプ（たとえば .M）だけを指定できます。アセンブリ・オプティマイザは特定のユニット（たとえば .M2）を割り当てます。
- ❏ データ・パス（たとえば 1）だけを指定できます。アセンブリ・オプティマイザは機能タイプ（たとえば .L1）を割り当てます。

機能ユニットの指定の有無に応じて、アセンブリ・オプティマイザは、ニーモニック・フィールドに基づいて機能ユニットを選択します。

どのマシン命令ニーモニックがどの機能タイプを必要とするかを含んだ機能ユニットの詳細は、TMS320C6000 CPU and Instruction Set Reference Guide を参照してください。

4.3.4 リニア・アセンブリ・ソース・コメントの使用方法

リニア・アセンブリ内のコメントは、任意のカラムから始めることができ、ソース行の最後まで拡張されます。コメントには、ブランクを含めて任意の ASCII 文字を指定できます。コメントはリニア・アセンブリ・ソース・リストに出力されますが、リニア・アセンブリには影響を与えません。

コメントだけが入っているソース文は有効です。コメントがカラム 1 から始まる場合は、セミコロン（;）またはアスタリスク（*）で始める必要があります。行の他の部分から始まるコメントはセミコロンで始める必要があります。アスタリスクがコメントとして識別されるのは、カラム 1 に指定される場合だけです。

アセンブリ・オプティマイザは命令をスケジュールします。つまり命令を再配置します。独立型のコメントは、命令のブロックの先頭に移動します。命令文の終わりにあるコメントは、その命令と一緒に元の位置に残ります。

リニア・アセンブリ入力ファイルで命令のコメントを入力すると、アセンブリ・オプティマイザはコメントを追加情報とともに出力ファイルに移動します。2 組の <x, y> をコメントに付加して、命令がソフトウェア・パイプラインにおいてループのどの反復とサイクルに属するかを指定します。ゼロ（0）を基点とする数値 x は、ループ・カーネルの最初の実行時の命令の反復を表します。ゼロ（0）を基点とする数値 y は、ループの 1 回の反復内に命令がスケジュールされるサイクルを表します。

例 4-3 は、コメントを含む Lmac と呼ばれる関数のコードを示します。

例 4-3.　コメントを表示する Lmac 関数コード

```
Lmac:    .cproc   A4,B4

         .reg    t0,t1,p,i,sh:sl

         MVK     100,i
         ZERO    sh
         ZERO    sl

loop:.trip   100

         LDH           *a4++, t0       ; t0 = a[i]
         LDH           *b4++, t1       ; t1 = b[i]
         MPY           t0,t1,p         ; prod = t0 * t1
         ADD           p,sh:sl,sh:sl   ; sum += prod
[i]      ADD           -1,i,i          ; --i
[i]      B             loop            ; if (i) goto loop

        .return sh:sl

        .endproc
```

4.3.5　シンボリック・レジスタ名を保持するアセンブリ・ファイル

出力アセンブリ・ファイルでは、レジスタ・オペランドにはシンボル名が入っています。これは、リニア・アセンブリ・ファイルをデバッグし、リニア・アセンブリ出力の断片をアセンブリ・ファイルに貼り付ける場合に便利です。

アセンブリ関数の先頭にある .map 疑似命令（4-20 ページを参照）は、シンボル名を実際のレジスタに関連付けます。つまり、シンボル名は実際のレジスタのエイリアスとなります。

コンパイラがユーザ・シンボルを 2 つのシンボルに分割し、それぞれが別々のマシン・レジスタにマップされる場合、接尾部がシンボル名のインスタンスに付加され、独自の名前が生成されます。この結果、独自の名前は 1 つのマシン・レジスタに関連付けられます。

たとえば、コンパイラが、シンボル名 y をある命令で A5 に関連付け、別の命令で B6 に関連付ける場合、出力アセンブリ・コードは次のようになります。

```
.MAP y/A5
.MAP y'/B6
...
ADD .S2X y, 4, y'  ; Equivalent to add A5, 4, B6
```

このシンボル名をもつ形式を無効にし、代わりに実際のレジスタをもつアセンブリ命令を表示するには、--machine_regs オプションを指定してコンパイルします。

4.4 アセンブリ・オプティマイザ疑似命令

アセンブリ・オプティマイザ疑似命令は、アセンブリ最適化プロセスにデータを提供し、そのプロセスを制御します。アセンブリ・オプティマイザは、プロシージャに含まれているリニア・アセンブリ・コード、つまり .proc と .endproc 疑似命令に囲まれた部分のコード、または .cproc と .endproc 疑似命令に囲まれた部分のコードを最適化します。リニア・アセンブリ・ファイルでこれらの疑似命令を使用しないと、アセンブリ・オプティマイザはコードを最適化しません。この節では、これらの疑似命令、およびアセンブリ・オプティマイザで使用できるその他の疑似命令について説明します。

表 4-1 は、アセンブリ・オプティマイザ疑似命令をまとめたものです。各疑似命令の構文、各疑似命令の説明、注意が必要な制約事項、および詳細情報のページを記載しています。

表 4-1. アセンブリ・オプティマイザ疑似命令のまとめ

構文	説明	制約事項	ページ
.call [*ret_reg* =] *func_name* **(***arg1*, *arg2***)**	関数を呼び出します。	プロシージャ内でのみ有効です。	4-15
.circ *variable1*|*register1*[**,** *variable2*|*register2*]	サーキュラ・アドレッシングを宣言します。	サーキュラ・アドレッシングの設定／分解コードを手作業で挿入しなければなりません。	4-17
label **.cproc** [*variable*$_1$ [**,** *variable*$_2$, ...]]	C/C++ 呼び出し可能プロシージャを開始します。	.endproc と一緒に使用しなければなりません。	4-17
.endproc	C/C++ 呼び出し可能プロシージャを終了します。	.cproc と一緒に使用しなければなりません。	4-17
.endproc [*register*$_1$ [**,** *register*$_2$, ...]]	プロシージャを終了します。	.proc と一緒に使用しなければなりません。	4-24
.map *variable1*|*register1*[**,** *variable2*|*register2*]	シンボルをレジスタに割り当てます。	実際のマシン・レジスタを使用しなければなりません。	4-20
.mdep [*symbol1*[, *symbol2*]]	メモリ依存性を示します。	プロシージャ内でのみ有効です。	4-21
.mptr {*register*|*symbol*}, *base* [**+** *offset*][**,** *stride*]	メモリ・バンクの競合を回避します。	プロシージャ内でのみ有効です。register パラメータに変数を使用できます。	4-21
.**no_mdep**	関数内にメモリ・エイリアスがありません。	プロシージャ内でのみ有効です。	4-23
.pref *variable*|*register*[*|register...*], ...	シンボルをセット内のレジスタに割り当てます。	実際のマシン・レジスタを使用しなければなりません。	4-23
label **.proc** [*register*$_1$ [**,** *register*$_2$, ...]]	プロシージャを開始します。	.endproc と一緒に使用しなければなりません。	4-24
.reg *variable*$_1$ [**,** *variable*$_2$, ...]	変数を宣言します。	プロシージャ内でのみ有効です。	4-26

表 4-1　アセンブリ・オプティマイザ疑似命令のまとめ（続き）

構文	説明	制約事項	ページ
.rega *variable*$_1$ [**,** *variable*$_2$**,** ...]	シンボルを A サイドのレジスタに分割します。	プロシージャ内でのみ有効です。	4-28
.regb *variable*$_1$ [**,** *variable*$_2$**,** ...]	シンボルを B サイドのレジスタに分割します。	プロシージャ内でのみ有効です。	4-28
.reserve [*register*$_1$ [**,** *register*$_2$**,** ...]]	コンパイラによるレジスタの割り当てを防止します。	プロシージャ内でのみ有効です。	4-28
.return [*argument*]	プロシージャに値を戻します。	.cproc プロシージャ内でのみ有効です。	4-29
label **.trip** *min*	トリップ・カウント値を指定します。	プロシージャ内でのみ有効です。	4-30
.volatile *variable*$_1$ [**,** *variable*$_2$**,** ...]	メモリ参照を揮発性として指定します。	割り込み時に参照が変更される場合は -mi1 を使用します。	4-32

.call　関数の呼び出し

構文　**.call**　[*ret_reg* =] *func_name* **(**[*arg1, arg2,...*]**)**

説明　**.call** 疑似命令を使用して関数を呼び出します。オプションで、呼び出しの結果を代入するレジスタを指定できます。このレジスタはシンボリック・レジスタでも、マシン・レジスタでもかまいません。.call 疑似命令は、C/C++ コンパイラと同じレジスタ規則および関数呼び出し規則に従います。詳細は、8.3 節「レジスタ規則」（8-17 ページ）および 8.4 節「関数の構造と呼び出し規則」（8-19 ページ）を参照してください。レジスタまたは関数の代替呼び出し規則はサポートされていません。

printf のような可変数の引数をもつ関数を呼び出すことはできません。引数の正しい数とタイプを確認するためのエラー検査は行われません。.call 疑似命令を使用して構造体を渡したり、戻したりすることはできません。

以下に、.call 疑似命令パラメータについて説明します。

ret_reg　（オプション）呼び出しの結果が代入されるシンボリック・レジスタ、またはマシン・レジスタ。指定されない場合、アセンブリ・オプティマイザは、レジスタ A5 および A4 に結果が代入されるものと推定します。

func_name　呼び出す関数の名前、または間接呼び出し用のシンボリック・レジスタまたはマシン・レジスタの名前。レジスタのペアは使用できません。呼び出された関数のラベルをファイル内に定義しなければなりません。関数のコードがファイル内にない場合は、.global または .ref 疑似命令を使用してラベルを定義しなければなりません。C/C++ 関数を呼び出す場合は、その関数の該当するリンク名を使用しなければなりません。詳細は、7.8 節「リンク名の生成」（7-33 ページ）を参照してください。

arguments　（オプション）引数として渡すシンボリック・レジスタまたはマシン・レジスタ。この引数は、この順序で渡されます。定数、メモリ参照、またはその他の式にすることはできません。

cl6x -ml*n* オプションを使用すると、呼び出 しが near か far かを指定できます。-ml*n* オプションが 0 に設定される場合、またはレベルが指定されない（デフォルト）場合、呼び出しは near になります。-ml*n* オプションが 1、2、または 3 に設定される場合、呼び出しは far になります。far コールを強制実行するには、関数のアドレスを明示的にレジスタにロードしてから間接呼び出しを発行しなければなりません。たとえば次のとおりです。

```
MVK    func,reg
MVKH   func,reg
call   reg(op1)              ; forcing a far call
```

間接呼び出しに * を使用する場合は、C/C++ 構文規則に従って次の代替構文を使用しなければなりません。

.call [*ret_reg* =] (* *ireg*) **(**[*arg1, arg2,...*]**)**

たとえば次のとおりです。

```
.call (*driver)(op1, op2)   ; indirect call

.reg  driver
.call driver(op1, op2)      ; also an indirect call
```

.call 構文を使用できるその他の有効な例は、次のとおりです。

```
.call fir(x, h, y)          ; void function

.call minimal()             ; no arguments

.call sum = vecsum(a, b)    ; returns an int

.call hi:lo = _atol(string) ; returns a long
```

シンボリック・レジスタを使用できる任意の場所でマシン・レジスタ名を使用できるので、関数呼び出し規則を変更できるように見えるかもしれません。たとえば次のとおりです。

```
.call   A6 = compute()
```

一見すると A4 ではなく A6 に結果が戻されるように見えますが、これは正しくありません。マシン・レジスタを使用しても、呼び出し規則は変更されません。*compute* 関数から戻って結果が A4 に戻された後、MV 命令でその結果を A6 に転送してしまうからです。

完全な .call の例は、次のとおりです。

```
        .global _main
        .global _puts, _rand, _ltoa
        .sect   ".const"
string1:        .string "The random value returned is ", 0
string2:        .string " ", 10, 0  ; '10' == newline
        .bss    charbuf, 20
        .text
_main:.cproc
        .reg    random_value, bufptr, ran_val_hi:ran_val_lo
        .call   random_value = _rand()              ; get a random value

        MVKL    string1, bufptr                     ; load address of string1
        MVKH    string1, bufptr
        .call   _puts(bufptr)                       ; print out string1
        MV      random_value, ran_val_lo
        SHR     ran_val_lo, 31, ran_val_hi          ; sign extend random value
        .call   _ltoa(ran_val_hi:ran_val_lo, bufptr)  ; convert it to a string
        .call   _puts(bufptr)                       ; print out the random
value
        MVKL    string2, bufptr                     ; load address of string2
        MVKH    string2, bufptr
        .call   _puts(bufptr)                       ; print out a newline
        .endproc
```

.circ　サーキュラ・アドレッシングの宣言

構文　**.circ** *variable*|*register* [**,** *variable*|*register*, ...]

説明　**.circ** 疑似命令は変数名をマシン・レジスタに割り当て、この変数をサーキュラ・アドレッシングとして使用できるよう宣言します。次に、コンパイラは変数をレジスタに割り当て、この状況ですべてのコード変換が安全に行われるようにします。サーキュラ・アドレッシングの設定／分解コードを挿入する必要があります。

variable　レジスタに割り当てる有効なシンボル名。変数は最大 128 文字の長さで、先頭を文字で開始する必要があります。変数の残りの文字には、英数字、下線（_）、およびドル記号（$）の組み合わせを使用できます。

register　変数に割り当てる実際のレジスタの名前。

コンパイラは、明示的に宣言されたサーキュラ・アドレッシング変数をアドレス・ポインタとして使用するロードを安全に実行できると仮定し、この仮定を利用して最適化を実行します。

.circ 疑似命令を使用して変数を宣言する場合、.reg 疑似命令を使用して変数を宣言する必要はありません。

.circ 疑似命令は、サーキュラ宣言で .map（4-20 ページを参照）を使用する場合と同じ働きをします。

例　以下に、シンボル名 Ri を実際のレジスタ Mi に割り当て、Ri がサーキュラ・アドレッシングで使用される可能性があるように宣言する例を示します。

```
.CIRC R1/M1, R2/M2 ...
```

.cproc/ .endproc　C 呼び出し可能プロシージャの定義

構文　*label*　**.cproc**　[*variable1* [**,** *variable2*, ?]]
　　.endproc

説明　アセンブリ・オプティマイザに最適化させ、C/C++ 呼び出し可能関数として処理させたいコード内のセクションを区切るには、**.cproc/.endproc** 疑似命令のペアを使用します。このセクションはプロシージャと呼ばれます。セクションの先頭で .cproc を 使用し、セクションの最後で .endproc を使用する点では、.cproc 疑似命令は .proc 疑似命令とほぼ同じです。このようにして、最適化したいアセンブリ・コードのセクションを関数のように区切ることができます。この疑似命令は、ペアで使用しなければなりません。.cproc を使用するときは、必ず .endproc を対応させてください。.cproc 疑似命令を使用する際にはラベルを指定してください。1 つのリニア・アセンブリ・ファイル内に複数のプロシージャをもつことができます。

コンパイラが .cproc 領域を C/C++ 呼び出し可能な関数として扱う点で、.cproc 疑似命令は .proc 疑似命令とは異なります。関数を C/C++ 呼び 出し規則および C/C++ レジスタ使用規則に準拠させるために、アセンブリ・オプティマイザは .cproc 領域内で自動的にいくつかの操作を実行します。

これらの操作には、次のものが含まれます。

- エントリ時保存レジスタ（A10 ～ A15 および B10 ～ B15）を使用する場合、アセンブリ・オプティマイザは、これらのレジスタをスタックに保存し、プロシージャの最後に元の値に戻します。
- コンパイラが、.reg 疑似命令（4-26 ページを参照）を使用して指定されたシンボリック・レジスタ名にマシン・レジスタを割り当てることができない場合、ローカル一時スタック変数を使用します。コンパイラは .cproc を使用してスタック・ポインタを管理し、スタック上でこれらの変数にスペースが割り当てられることを確認します。

詳細は、8.3 節「レジスタ規則」（8-17 ページ）および 8.4 節「関数の構造と呼び出し規則」（8-19 ページ）を参照してください。

.cproc 領域に入れることのできない命令のタイプについては、「.proc 疑似命令」（4-25 ページ）を参照してください。

関数パラメータを表すには、オプションの *variable* を使用します。この variable エントリは、C/C++ 関数で宣言されるパラメータと非常によく似ています。.cproc 疑似命令に対する引数は、次のタイプにすることができます。

- **マシン・レジスタ名**
  マシン・レジスタ名を指定する場合、引数リスト内のその位置は C の引数パス規則に対応しなければなりません。たとえば、C/C++ コンパイラは、レジスタ A4 内の関数に最初の引数を渡します。これは、.cproc 疑似命令内の最初の引数が A4 またはシンボル名でなければならないという意味です。.cproc 疑似命令には最高 10 個までの引数を使用できます。
- **変数名**
  変数名を指定する場合、アセンブリ・オプティマイザは、引数を渡す適切なレジスタに変数名を割り当てるか、引数を渡すレジスタを、変数名に割り当てられるレジスタに複写するかのどちらかを実行します。たとえば次の .cproc 疑似命令を指定する場合には、C/C++ 呼び出し内の最初の引数はレジスタ A4 で渡されます。

  ```
  frame   .cproc arg1
  ```

  アセンブリ・オプティマイザは、arg1 を A4 に割り当てるか、または arg1 が異なるレジスタ（たとえば B7）に割り当てられるかによって、MV A4、B7 が自動的に生成されます。

❑ **レジスタ・ペア**
レジスタのペアは arghi:arglo のように指定され、40 ビット引数、または 'C6700 の場合は 64 ビットのタイプ double 引数を表します。たとえば、.cproc が次のように定義されるとします。

```
_fcn:  .cproc arg1, arg2hi:arg2lo, arg3, B6, arg5, B9:B8
       ...
       .return res
       ...
       .endproc
```

これは、次のように宣言された C 関数に対応します。

```
int fcn(intarg1,longarg2,intarg3,intarg4,intarg5,longarg6);
```

この例では、.cproc の 4 番目の引数がレジスタ B6 です。これが可能なのは、C/C++ 呼び出し規則内の 4 番目の引数が B6 で渡されるからです。.cproc の 6 番目の引数は、実際にはレジスタ・ペア B9:B8 です。これが可能なのは、C/C++ 呼び出し規則内の 6 番目の引数が B8 で渡され、long の場合は B9:B8 で渡されるからです。

C++ ソースからプロシージャを呼び出す場合は、そのプロシージャ・ラベルに適したリンク名を使用しなければなりません。それ以外の場合は、extern C 宣言を使用すると C 命名規則を強制実行できます。詳細は、7.8 節「リンク名の生成」（7-33 ページ）および 8.5 節「アセンブリ言語と C および C++ 言語間のインターフェイス」（8-23 ページ）を参照してください。

.endproc は .cproc 疑似命令と一緒に使用される場合、引数をもつことはできません。.cproc 領域に設定されたリブ・アウトは、その .cproc 領域に出現する任意の .return 疑似命令により決まります（値がプロシージャの前またはプロシージャ内で定義され、プロシージャからの出力として使用される場合、その値はリブ・アウトです）。.cproc 領域から値を戻すことは、.return 疑似命令によって処理されます。戻り分岐は、自動的に .cproc 領域内で生成されます。.return 疑似命令については、4-29 ページを参照してください。

プロシージャ内のコードだけが最適化されます。アセンブリ・オプティマイザは、プロシージャ外にあるすべてのコードを出力ファイルに複写し、それを変更しません。.cproc 領域内で使用できない命令タイプについては、4-25 ページを参照してください。

例　.cproc および .endproc が使用される例は、次のとおりです。

```
_if_then:.cproc   a, cword, mask, theta

        .reg    cond, if, ai, sum, cntr

        MVK         32,cntr             ; cntr = 32
        ZERO        sum                 ; sum = 0

LOOP:
        AND         cword,mask,cond     ; cond = codeword & mask
 [cond] MVK         1,cond              ; !(!(cond))
        CMPEQ       theta,cond,if       ; (theta == !(!(cond)))
        LDH         *a++,ai             ; a[i]
 [if]   ADD         sum,ai,sum          ; sum += a[i]
 [!if]  SUB         sum,ai,sum          ; sum -= a[i]
        SHL         mask,1,mask         ; mask = mask << 1
 [cntr] ADD         -1,cntr,cntr        ; decrement counter
 [cntr] B           LOOP                ; for LOOP

        .return sum

        .endproc
```

.map　変数をレジスタに割り当てる

構文　**.map** *variable*|*register* [**,** *variable*|*register*, ...]

説明　**.map** 疑似命令は変数名をマシン・レジスタに割り当てます。シンボル／変数は代替シンボル・テーブルに保存されます。シンボリック名と実際のレジスタの関連付けは、各リニア・アセンブリ関数の先頭と最後で解消されます。.map 疑似命令は、アセンブリとリニア・アセンブリ・ファイルで使用できます。

variable　レジスタに割り当てる有効なシンボル名。代替シンボルは最大 128 文字の長さで、先頭を文字で開始する必要があります。この変数の残りの文字には、英数字、下線（_）、およびドル記号（$）の組み合わせを使用できます。

register　変数に割り当てる実際のレジスタの名前。

.map 疑似命令を使用して変数を宣言する場合、.reg 疑似命令を使用して変数を宣言する必要はありません。

例　.map 疑似命令を使用して x をレジスタ A6 に割り当て、y をレジスタ B7 に割り当てる例は次のとおりです。変数は move 文で使用されます。

```
.MAP x/A6, y/B7

MV  x, y                ; Equivalent to MV A6, B7
```

.mdep　メモリ依存性の指定

構文　**.mdep***symbol1*, *symbol2*

説明　**.mdep** 疑似命令は、特定のメモリへの依頼性を識別します。

.mdep 疑似命令パラメータについて、次に説明します。

symbol　symbol パラメータは、メモリ参照の名前です。

メモリ参照の名前指定に使用されるシンボルには、アセンブリ・シンボルと同じ構文制約事項があります。(シンボルの詳細は、TMS320C6000 Assembly Language Tools User's Guide を参照してください)。シンボリック・レジスタと同じスペース内にあります。シンボリック・レジスタとメモリ参照の注釈付けに同じ名前を使用できません。

.mdep 疑似命令は、2 つのメモリ参照間に依存関係があることをアセンブリ・オプティマイザに指示します。

.mdep 疑似命令はプロシージャ内でのみ有効です。つまり、.proc と .endproc 疑似命令のペア、または .cproc と .endproc 疑似命令のペアの範囲内でのみ有効です。

例　.mdep は 2 つのメモリ参照間の依存関係を示すために使用されます。その例は次のとおりです。

```
.mdep ld1, st1

LDW    *p1++ {ld1}, inp1      ;name memory reference "ld1"
;other code ...
STW    outp2, *p2++ {st1}     ;name memory reference "st1"
```

.mptr　メモリ・バンクの競合の回避

構文　**.mptr** {*register*|*symbol*}, *base* [**+** *offset*] [**,** *stride*]

説明　**.mptr** 疑似命令は、あるレジスタに特定情報を関連付けて、アセンブリ・オプティマイザに 2 つのメモリ操作にメモリ・バンクの競合があるかどうかを自動判別させるようにします。アセンブリ・オプティマイザが、2 つのメモリ操作にメモリ・バンクの競合が起きていると判別すると、その 2 つの操作を並列してスケジュールすることはありません。

メモリ・バンクの競合は、所定のサイクル内で 1 つのバンクに対して 2 つのアクセスがあるときに、メモリ待ちが発生し、1 サイクルの間すべてのパイプライン操作が停止すると同時に 2 番目の値がメモリから読み取られる場合に発生します。メモリ・バンクの競合を防止するために .mptr 疑似命令を使用する方法を含めて、メモリ・バンク競合の詳細は 4.5 節「アセンブリ・オプティマイザを使用したメモリ・バンク競合の回避方法」(4-33 ページ) を参照してください。

.mptr 疑似命令パラメータについて、次に説明します。

register|symbol	特定のメモリ参照の名前指定に使用されるレジスタまたはシンボルの名前。
base	関連したメモリ・アクセスを関連づけるシンボル。
offset	先頭のベース・シンボルからのオフセット（バイト単位）。offset はオプション・パラメータであり、デフォルトでは 0 に解釈されます。
stride	レジスタ・ループの増分（バイト単位）。stride はオプション・パラメータであり、デフォルトでは 0 に解釈されます。

.mptr 疑似命令は、アセンブリ・オプティマイザに対して、レジスタまたはシンボル名が LD(B/BU)(H/HU)(W) または ST(B/H/W) 命令内のメモリ・ポインタとして使用されるときに *base* + *offset* を指すように初期設定され、ループを通じて毎回 *stride* により増分されることを指示します。

.mptr 疑似命令はプロシージャ内でのみ有効です。つまり、.proc と .endproc 疑似命令のペア、または .cproc と .endproc 疑似命令のペアの範囲内でのみ有効です。

ベース・シンボル名に使用されるシンボルは、他のすべてのラベルとは別個のネーム・スペース内にあります。これは、シンボリック・レジスタまたはアセンブリ・ラベルが、メモリ・バンクのベース名と同じ名前を持つことができるという意味です。たとえば次のとおりです。

```
     .mptr  Darray,Darray
```

例　次の例では、メモリ・バンクの競合を回避するために .mptr が使用されています。

```
_blkcp:   .cproc i

          .reg   ptr1, ptr2, tmp1, tmp2

          MVK    0x0, ptr1              ; ptr1 = address 0
          MVK    0x8, ptr2              ; ptr2 = address 8

loop:     .trip   50

          .mptr  ptr1, a+0, 4
          .mptr  foo, a+8, 4

                                        ; potential conflict
          LDW    *ptr1++, tmp1          ; load  *0, bank 0
          STW    tmp1, *ptr2++{foo}     ; store *8, bank 0

   [i]    ADD    -1,i,i                 ; i--
   [i]    B      loop                   ; if (!0) goto loop

          .endproc
```

.no_mdep　関数内にメモリ・エイリアスなし

構文　**.no_mdep**

説明　**.no_mdep** 疑似命令はアセンブリ・オプティマイザに対して、その関数内にメモリの依存関係が発生しないことを指示します。ただし、.mdep 疑似命令を使用してポイントされる依存関係はその指示から除外されています。

例　.no_mdep の使用例は、次のとおりです。

```
fn:             .cproc dst, src, cnt
                .no_mdep;no memory aliasing in this function
                ...
                .endproc
```

.pref　変数をレジスタにセットで割り当てる

構文　**.pref** *variable*[,]*register1*[[,]*register2...*]

説明　**.pref** 疑似命令は優先設定と通信して、レジスタのリストのいずれかに変数を割り当てます。シンボル／変数は代替シンボル・テーブルに保存されます。シンボリック名と実際のレジスタの関連付けは、各リニア・アセンブリ関数の先頭と最後で解消されます。

variable　レジスタに割り当てる有効なシンボル名。代替シンボルは最大 128 文字の長さで、先頭を文字で開始する必要があります。この変数の残りの文字には、英数字、下線（_）、およびドル記号（$）の組み合わせを使用できます。

register　変数に割り当てる実際のレジスタのリスト。

変数が指定したグループ内のレジスタに割り当てられる保証はありません。コンパイラは優先設定を無視する場合があります。

.pref 疑似命令を使用して変数を宣言する場合、.reg 疑似命令を使用して変数を宣言する必要はありません。

例　優先設定に指定された x を A6 または B7 に割り当てる例を以下に示します。ただし、コンパイラが x を代わりに B3（たとえば）に割り当て方が正しい場合があります。

```
.PREF x/A6/B7 ; Preference to assign x to either A6 or B7
```

.proc/.endproc プロシージャの定義

構文

label **.proc** [*register1* [**,** *register2***,** ?]]
.endproc [*register1* [**,** *register2***,** ?]]

説明

アセンブリ・オプティマイザに最適化させたいコードのセクションを区切るには、**.proc/.endproc** 疑似命令のペアを使用します。このセクションはプロシージャと呼ばれます。セクションの先頭で .proc を使用し、セクションの最後で .endproc を使用してください。このようにして、最適化したいアセンブリ・コードのセクションを関数のように区切ることができます。この疑似命令は、ペアで使用しなければなりません。.proc を使用するときは、必ず .endproc を対応させてください。.proc 疑似命令を使用する際には、ラベルを指定してください。1 つのリニア・アセンブリ・ファイル内に複数のプロシージャをもつことができます。

どのレジスタがリブ･インであるかを指定するには .proc 疑似命令でオプションの *register* パラメータを使用し、またプロシージャごとにどのレジスタがリブ・アウトであるかを指定するには .endproc 疑似命令でオプションの register パラメータを使用してください。*register* はシンボリック・レジスタでも、マシン・レジスタでもかまいません。たとえば次のとおりです。

```
   .PROC x, A5, y, B7
...
   .ENDPROC y
```

値がプロシージャの前で定義され、プロシージャからの入力として使用される場合、その値はリブ・インです。値がプロシージャの前またはプロシージャ内で定義され、プロシージャからの出力として使用される場合、その値はリブ・アウトです。.endproc 疑似命令でレジスタを指定しない場合、リブ・アウトのレジスタはないものと想定されます。

プロシージャ内のコードだけが最適化されます。アセンブリ・オプティマイザは、プロシージャ外にあるすべてのコードを出力ファイルに複写し、それを変更しません。

例

.proc および .endproc が使用されるブロック・ムーブ例は、次のとおりです。

```
move   .proc A4, B4, B0
       .no_mdep

loop:
       LDW*B4++, A1
       MVA1, B1
       STWB1, *A4++
       ADD-4, B0, B0
 [B0]  Bloop
       .endproc
```

.proc または .cproc（4-17 ページと 4-24 ページを参照）領域では、次のタイプの命令は使用できません。

- スタック・ポインタ（レジスタ B15）を参照する命令は、.proc や .cproc 領域では許されていません。アセンブリ・オプティマイザは、.proc や .cproc 領域内で一時的な値を保管するために、スタック・スペースを割り当てることができます。この記憶域を割り当てるために、スタック・ポインタはこの領域に入ると減らされ、この領域から出ると増やされます。スタック・ポインタは領域に入るときに値を変更できるので、アセンブリ・オプティマイザは、スタック・ポインタ・レジスタを参照するコードを許していません。

- .proc や .cproc 領域の出口プロトコルをバイパスできないようにするため、.proc や .cproc 領域では間接分岐は許されていません。間接分岐の例は、次のとおりです。

```
   BB4<=  illegal
```

- .proc や .cproc 領域の出口プロトコルをバイパスできないようにするため、.proc や .cproc 領域内で定義されていないラベルへの直接分岐は許されていません。.proc 領域外の直接分岐の例は、次のとおりです。

```
   .proc
   ...
   B outside<= illegal
   .endproc
outside:
```

- .proc 疑似命令に関連付けられていないラベルへの直接分岐は許されていません。リニア・アセンブリ関数の先頭に分岐を戻す必要がある場合は、.call 疑似命令を使用します。.proc 領域のラベルへの直接分岐の例は、次のとおりです。

```
_func:.proc
   ...
   B _func<= illegal
   ...
   .endproc
```

- .if/.endif ループは、.proc または .cproc 領域の完全に内部に配置するか、または外部に配置する必要があります。.if/.endif ループの一部を .proc または .cproc 領域の内部に配置し、.if/.endif ループの他の部分を .proc または .cproc 領域の外部に配置することは許されていません。次に、有効な .if/.endif ループの 2 つの例を示します。最初のループは .cproc 領域の外部にあり、2 番目のループは .proc 領域の内部にあります。

```
.if
.cproc
...
.endproc
.endif

.proc
.if
....
.endif
.endproc
```

次に、一部が .cproc または .proc の内部にあり、一部が .cproc または .proc の外部にある .if/.endif ループの 2 つの例を示します。

```
.if
.cproc
.endif
.endproc

.proc
.if
...
.else
.endproc
.endif
```

.reg　変数の宣言

構文　**.reg** *variable*$_1$ [**,** *variable*$_2$**,**?]

説明　**.reg** 疑似命令を使用すると、レジスタに保管されている値に対して記述名を使用できます。アセンブリ・オプティマイザは、その値で動作する命令に対して選択された機能ユニットとレジスタの用途とが一致するように、ユーザに代わってレジスタを選択します。

.reg 疑似命令はプロシージャ内でのみ有効です。つまり、.proc と .endproc 疑似命令のペア、または .cproc と .endproc 疑似命令のペアの範囲内でのみ有効です。

レジスタ・ペアの明示的な宣言はオプションです。アセンブリ・オプティマイザは、レジスタがペアで使用される場合に派生します。レジスタ・ペアを宣言する例を次に示します。

```
.reg   A6, A7
```

例 1

この例では 4-24 ページのブロック・ムーブの例と同じコードを使用しますが、.reg 疑似命令を使用します。

```
move   .cproc dst, src, cnt

       .reg tmp1, tmp2

loop:
       LDW     *src++, tmp1
       MV      tmp1, tmp2
       STW     tmp2, *dst++
       ADD     -4, cnt, cnt
[cnt]  B       loop

       .endproc
```

この例と 4-24 ページの .proc 例との相違点に注目してください。.reg を使用して宣言されたシンボリック・レジスタは、マシン・レジスタとして割り当てられます。

例 2

次の例のコードは無効です。.reg 疑似命令により定義された変数を .proc 疑似命令と一緒に使用できないからです。

```
move   .proc dst, src, cnt    ; WRONG:You cannot use a
       .reg dst, src, cnt     ; variable with .proc
```

この例は、次のように訂正できます。

```
move   .cproc dst, src, cnt
```

例 3

次の例のコードは無効です。.reg 疑似命令により定義された変数を、定義されたプロシージャの外では使用できないからです。

```
move   .proc A4
       .reg tmp

       LDW*A4++, tmp
       MVtmp, B5

       .endproc

       MV tmp, B6; WRONG:tmp is invalid outside of
                 ; the procedure
```

.rega/.regb　レジスタの直接分割

構文

.rega *variable*$_1$ [**,** *variable*$_2$,?]
.regb *variable*$_1$ [**,** *variable*$_2$,?]

説明

レジスタは 2 つの疑似命令を使用して直接分割できます。**.rega** 疑似命令は、変数名を A サイドのレジスタに制約するために使用します。**.regb** 疑似命令は、変数名を B サイドのレジスタに制約するために使用します。たとえば次のとおりです。

```
.REGA y
.REGB u, v, w

MV   x, y
LDW *u, v:w
```

.rega と .regb 疑似命令はプロシージャ内でのみ有効です。つまり、.proc と .endproc 疑似命令のペア、または .cproc と .endproc 疑似命令のペアの範囲内でのみ有効です。

.rega または .regb 疑似命令を使用して変数を宣言する場合、.reg 疑似命令を使用して変数を宣言する必要はありません。

命令を分割して間接的にレジスタを分割する古い方法は、引き続き使用できます。サイドおよび機能ユニット指定子も命令で引き続き使用できます。ただし、機能ユニット指定子（.L/.S/.D/.M）とクロス・パス情報は無視されます。サイド指定子は、対応する変数名（ある場合）の分割化の制約に変換されます。たとえば次のとおりです。

```
MV  .1X     z, y     ; translated to .REGA y
LDW .D2T2 *u, v:w    ; translated to .REGB u, v, w
```

.reserve　レジスタの予約

構文

.reserve [*register*$_1$ [**,** *register*$_2$, ?]]

説明

.reserve 疑似命令は、アセンブリ・オ プティマイザが、.proc または .cproc 領域内で、指定された *register* を使用しないようにします。

予約済みのレジスタが .proc または .cproc 領域内で明示的に割り当てられる場合には、アセンブリ・オプティマイザもそのレジスタを使用できます。たとえば、レジスタ A7 が .reserve リスト内にある場合であっても、変数 tmp1 をレジスタ A7 に割り当てることができます。これは、A7 が ADD 命令で明示的に定義されたからです。

```
.cproc
.reserve  a7
.reg      tmp1

....
ADD       a6, b4, a7
....

.endproc
```

注 : レジスタ A4 と A5 の予約

.call 文を含む .cproc 領域内では、A4 と A5 を .reserve 文で指定することはできません。呼び出し規則は、A4 と A5 を .call 文の戻りレジスタとして使用するように指定しています。

例 1

この例の .reserve は、アセンブリ・オプティマイザが変数 tmp1 ～ tmp5 に対して A10 ～ A13 または B10 ～ B13 を使用しないことを保証します。

```
test   .proc a4, b4
    .reg  tmp1, tmp2, tmp3, tmp4, tmp5
    .reserve a10, a11, a12, a13, b10, b11, b12, b13

    .....

    .endproc a4
```

例 2

使用可能なレジスタ・プールが過度に制限されている場合、アセンブリ・オプティマイザは効率の良くないコードを生成する可能性があります。その上、使用可能なレジスタ・プールが、割り当てができないほど制約され、エラー・メッセージが生成される可能性もあります。たとえば、次のコードはエラーを生成します。これは、条件付きレジスタがすべて予約されているにもかかわらず、変数 tmp に条件付きレジスタが必要だからです。

```
    .cproc ...
    .reserve a1,a2,b0,b1,b2
    .reg tmp

    ....
[tmp] ....
    ....

    .endproc
```

.return　C 呼び出し可能プロシージャに値を戻す

構文

.return [*argument*]

説明

.return 疑似命令関数は、C/C++ コードの return 文と同じ働きをします。これは、C/C++ 呼び出し規則に従って、戻り値用の適切なレジスタにオプションの argument を入れます（8.4 節「関数の構造と呼び出し規則」（8-19 ページ）を参照）。

オプションの *argument* には、次の意味があります。

- 引数なしは、C/C++ コードの void 関数と同じように戻り値がない .cproc 領域を暗黙指定します。
- 1 つの引数は、C/C++ コードの int 関数と同じように 32 ビットの戻り値がある .cproc 領域を暗黙指定します。

- ❑ 形式が hi:lo であるレジスタ・ペアは、C/C++ コードの long/long long/double 関数と同じように 40 ビットの long 型の戻り値、64 ビットの long long 型の戻り値、または 64 ビットの double 型の戻り値をもつ .cproc 領域を暗黙指定します。

.return 疑似命令に対する引数は、シンボリック・レジスタ名であっても、マシン・レジスタ名であってもかまいません。

.cproc 領域内のすべての return 文は、戻り値のタイプが一貫していなければなりません。同じ .cproc 領域で、.return arg を .return hi:lo と組み合わせることはできません。

.return 疑似命令は無条件です。条件付き .return を実行する場合には、.return の周辺で単に条件付き分岐を使用してください。アセンブリ・オプティマイザがその分岐を除去し、適切な条件付きコードを生成します。たとえば条件 cc が真である場合に戻すには、次のように return をコード化します。

```
[!cc] Baround
   .return
around:
```

例 次の例では、シンボリック・レジスタ名 `tmp`、およびマシン・ レジスタ `A5` を `.return` 引数として使用します。

```
   .cproc ...
   .reg tmp

   ...
   .return tmp<= legal symbolic name
   ...
   .return a5<= legal actual name
```

.trip トリップ・カウント値の指定

構文 *label* **.trip** *minimum value*, [*maximum value*[, *factor*]]

説明 **.trip** 疑似命令は、トリップ・カウントの値を指定します。トリップ・カウントはループが繰り返す回数です。.trip 疑似命令は、プロシージャ内でのみ有効です。.trip 疑似命令パラメータについて、次に説明します。

label	label はループの先頭を表します。これは必須パラメータです。
minimum value	ループが反復できる最小回数。これは必須パラメータであり、デフォルトは 1 です。
maximum value	ループが反復できる最大回数。これはオプション・パラメータです。

factor

ループが反復できる回数を決定するために、*minimum value* および *maximum value* と一緒に使用される係数。次の例では、ループは 8 の倍数（8 ～ 48）回、実行されます。

```
loop:  .trip 8, 48, 8
```

factor が 2 であるとループは常に偶数回実行され、コンパイラが 1 度展開できることを示します。この結果、パフォーマンスが向上する場合があります。

maximum value が指定されている場合、factor はオプションです。

どのループにも .trip 疑似命令を指定する必要があるわけではありませんが、ループが何回か繰り返すことが分かっている場合は .trip を指定しなければなりません。一般に、これは（minimum value が非常に小さい場合を除いて）冗長なループが生成されず、コード・サイズと実行時間を節減することを意味します。

ループが呼び出されるたびに必ず同じ回数実行されることが分かっている場合は、maximum value も定義してください（この場合、maximum value は minimum value と同じ値です）。これでコンパイラはループを展開できるようになったので、パフォーマンスが向上します。

割り込み柔軟性オプション（-mi*n*）を指定してコンパイルしようとする場合、.trip maximum value を使用すると、コンパイラはループが実行できる最大サイクル数を決定できます。その後、コンパイラはその値を -mi オプションによって指定されたしきい値と比較します。詳細は、2.11 節「割り込み柔軟性オプション（-mi オプション）」（2-43 ページ）を参照してください。

トリップ・カウントがパフォーマンスを最大に保つためにループをパイプライン化できる大きさであることを、アセンブリ・オプティマイザが保証できない場合、同じループのパイプライン化されたバージョンとパイプライン化されていないバージョンが生成されます。これにより、ループのどちらかが冗長ループになります。パイプライン化されたループまたはパイプラインされていないループは、トリップ・カウントと、並列して実行できるループの反復回数との間の比較に基づいて実行されます。トリップ・カウントが並行反復数より大きいか等しい場合、パイプライン化されたループが実行されます。それ以外の場合は、パイプライン化されていないループが実行されます。冗長ループの詳細は、3.3 節「冗長なループ」（3-16 ページ）を参照してください。

例　次の .trip 疑似命令は、w_vecsum ルーチンが呼びされると、ループが 16、24、32、40 または 48 回実行されることを指定します。

```
w_vecsum:.cproc  ptr_a, ptr_b, ptr_c, weight, cnt
            .reg    ai, bi, prod, scaled_prod, ci
            .no_mdep

loop:.trip 16, 48, 8
            ldh     *ptr_a++, ai
            ldh     *ptr_b++, bi
            mpy     weight, ai, prod
            shr     prod, 15, scaled_prod
            add     scaled_prod, bi, ci
            sth     ci, *ptr_c++
  [cnt]     sub     cnt, 1, cnt
  [cnt]     b       loop
            .endproc
```

.volatile　メモリ参照を揮発性として宣言する

構文　**.volatile** *symbol*$_1$ [, *symbol*$_2$,?]

説明　**.volatile** 疑似命令を使用して、メモリ参照を volatile として指定できます。揮発性ロードおよびストアは削除されません。揮発性ロードおよびストアは、他の揮発性ロードおよびストアとの関連で並べ換えられません。

.volatile 疑似命令が割り込み時に変更されるメモリ位置を参照する場合、-mil オプションを使用してコンパイルして、揮発性メモリの位置を参照するすべてのコードの割り込みを可能にします。

例　st および ld メモリ参照は揮発性として指定されます。

```
   .volatile st, ld

   STW  W, *X{st}          ; volatile store
   STW  U, *V
   LDW *Y{ld}, Z           ; volatile load
```

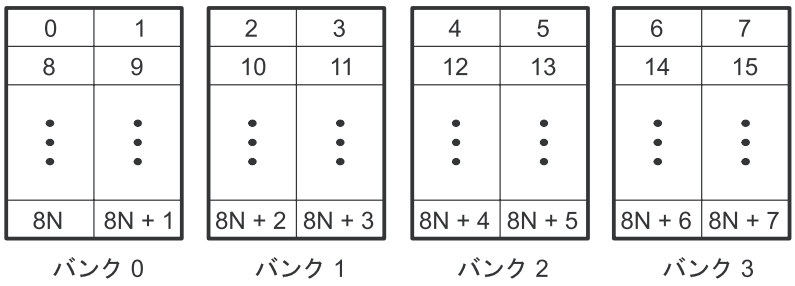
4.5 アセンブリ・オプティマイザを使用したメモリ・バンク競合の回避方法

C6000 ファミリーの内部メモリは、デバイスごとに異なります。ご使用の特定デバイス内のメモリ・スペースを判別する場合には、該当するデバイスのデータ・シートを参照してください。この節では、メモリ・バンクの競合を回避するためのコードの書き方を説明します。

大部分の C6000 デバイスは、図 4–1 に示されているようにインターリーブド・メモリ・バンク体系を使用しています。この図の中の各番号は、バイト・アドレスを表しています。アドレス 0 からのロード・バイト（LDB）命令は、バンク 0 にバイト 0 をロードします。アドレス 0 からのロード・ハーフワード（LDH）は、バンク 0 にあるバイト 0 と 1 にハーフワード値をロードします。アドレス 0 からのロード・ワード（LDW）は、バンク 0 と 1 にバイト 0 ～ 3 をロードします。

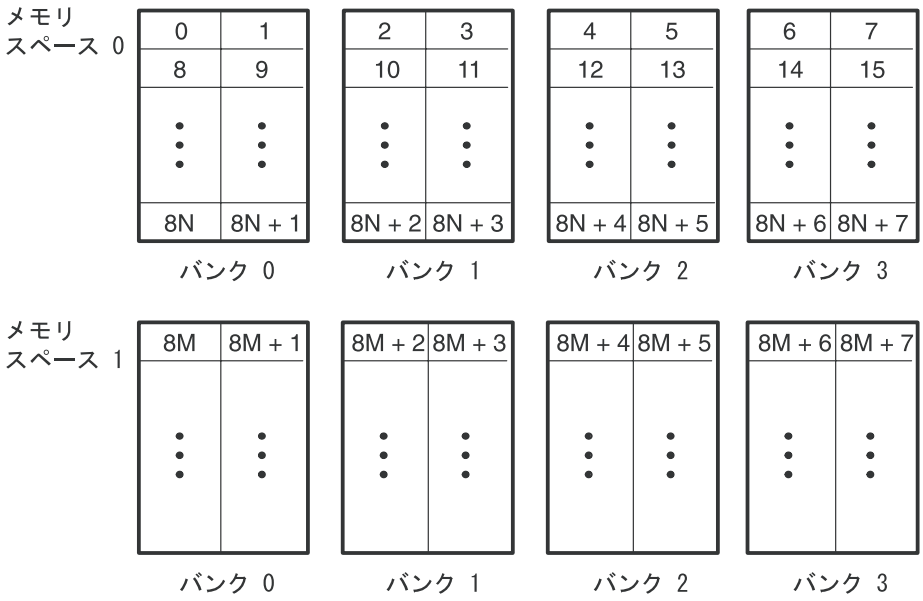
各バンクは単一ポート・メモリなので、1 サイクル当たり各バンクへのアクセスは 1 つしか許可されません。所定のサイクル内で 1 つのバンクに対して 2 つのアクセスがあると、メモリ待ちが発生し、1 サイクルの間すべてのパイプライン操作が停止すると同時に 2 番目の値がメモリから読み取られます。同じバンクにアクセスしない限り、メモリ待ちを発生させることなく 1 サイクル当たり 2 つのメモリ操作が可能です。

図 4–1. 4 バンク・インターリーブド・メモリ



複数のメモリ・スペースがあるデバイスの場合（図 4–2）、1 つのメモリ・スペース内のバンク 0 へのアクセスは、別のメモリ・スペース内のバンク 0 へのアクセスと衝突することはありません。パイプラインの停止も起きません。

図 4–2.　2 つのメモリ・スペースがある 4 バンク・インターリーブド・メモリ



4.5.1　メモリ・バンク競合の防止方法

アセンブリ・オプティマイザは、メモリ操作にはバンク競合がないことを前提としています。2 つのメモリ動作でループ反復中に何らかのバンク競合が起きていると判定すると、その 2 つの動作を並列してスケジュールすることはありません。アセンブリ・オプティマイザは、ソフトウェア・パイプライン化を試みているループに対してのみメモリ・バンク競合をチェックします。

メモリ・バンクの解析に必要な情報は、ベース、オフセット、ストライド、幅、および反復デルタです。幅は、メモリ・アクセスのタイプによって暗黙的に決定されます ('C6400 と 'C6700 の場合は、バイト、ハーフワー ド、ワード、またはダブルワード)。反復デルタは、ソフトウェア・パイプライン用のスケジュールを作成するときにアセンブリ・オプティマイザが判別します。ベース、オフセット、ストライドは、ロード命令とストア命令または .mptr 疑似命令、あるいはその両方によって指定されます。

リニア・アセンブリでの LD(B/BU)(H/HU)(W) または ST(B/H/W) 操作には、.mptr 疑似命令を使用すると、暗黙的なメモリ・バンク情報を関連付けることができます。.mptr 疑似命令は、アセンブリ・オプティマイザが 2 つのメモリ操作でバンク競合が起きているかどうかを自動的に判別できるようにする情報をレジスタに関連付けます。アセンブリ・オプティマイザが、2 つのメモリ操作にメモリ・バンク競合が起きていると判別すると、その 2 つの操作を並列してスケジュールすることはありません。構文は次のとおりです。

.mptr *register*, *base*+*offset*, *stride*

たとえば次のとおりです。

```
.mptr a_0,a+0,16
.mptr a_4,a+4,16

LDW *a_0++[4], val1  ; base=a, offset=0, stride=16
LDW *a_4++[4], val2  ; base=a, offset=4, stride=16

.mptr dptr,D+0,8

LDH *dptr++, d0      ; base=D, offset=0, stride=8
LDH *dptr++, d1      ; base=D, offset=2, stride=8
LDH *dptr++, d2      ; base=D, offset=4, stride=8
LDH *dptr++, d3      ; base=D, offset=6, stride=8
```

この例では、どのメモリ・アクセスの後でも dptr のオフセットが更新されます。オフセットが更新されるのは、ポインタが定数によって修正される場合だけです。これは、プリ／ポストインクリメント／デクリメント・アドレス・モードの場合だけ発生します。

.mptr 疑似命令については、4-21 ページを参照してください。

例 4-4 は、ソフトウェア・パイプライン化されているループから抽出されるロードとストアを示しています。

例 4-4. メモ リ・バンク情報を指定するロード命令とストア命令

```
        .mptr   Ain,IN,-16
        .mptr   Bin,IN-4,-16

        .mptr   Aco,COEF,16
        .mptr   Bco,COEF+4,16

        .mptr   Aout,optr+0,4
        .mptr   Bout,optr+2,4

        LDW           *Ain--[2],Ain12           ; IN(k-i) & IN(k-i+1)
        LDW           *Bin--[2],Bin23           ; IN(k-i-2) & IN(k-i-1)
        LDW           *Ain--[2],Ain34           ; IN(k-i-4) & IN(k-i-3)
        LDW           *Bin--[2],Bin56           ; IN(k-i-6) & IN(k-i-5)

        LDW           *Bco++[2],Bco12           ; COEF(i) & COEF(i+1)
        LDW           *Aco++[2],Aco23           ; COEF(i+2) & COEF(i+3)
        LDW           *Bco++[2],Bin34           ; COEF(i+4) & COEF(i+5)
        LDW           *Aco++[2],Ain56           ; COEF(i+6) & COEF(i+7)

        STH           Assum,*Aout++[2]          ; *oPtr++ = (r >> 15)
        STH           Bssum,*Bout++[2]          ; *oPtr++ = (i >> 15)
```

4.5.2 メモリ・バンクの競合を回避する内積例

例 4-5 の C コードは、内積関数を実現しています。C6000 が 1 つの 32 ビット・レジスタ内の 2 つの 16 ビット・データ項目で演算できる機能を利用するために、内部ループが一度展開されます。LDW 命令は、2 つの連続した短い値をロードするために使用されます。例 4-6 内のリニア・アセンブリ命令は、dotp のループ・カーネルを実現します。例 4-7 は、アセンブリ・オプティマイザにより判別されたループ・カーネルを示しています。

このループ・カーネルの場合、配列 a[] と b[] に関連した次の 2 つの制約事項があります。

- LDW を使用しているので、この配列はワード境界で始まるように位置合わせが必要です。
- メモリ・バンクの競合を避けるために、一方の配列はバンク 0 から開始し、もう一方の配列はバンク 2 から開始する必要があります。同じバンクから開始するとどのサイクルでもメモリ・バンク競合が発生し、ループは、メモリ・バンクが停止するので、すべてのサイクルごとにではなく 2 サイクルごとに結果を計算します。たとえば次のとおりです。

  バンク競合あり

```
     MVK   0, A0
  || MVK   8, B0
     LDW   *A0, A1
  || LDW   *B0, B1
```

  バンク競合なし

```
     MVK   0, A0
  || MVK   4, B0
     LDW   *A0, A1
  || LDW   *B0, B1
```

例 4-5. 内積用の C コード

```
int dotp(short a[], short b[])
{
    int sum0 = 0, sum1 = 0, sum, i;
    for (i = 0; i < 100/2; i+= 2)
    {
        sum0 += a[i] * b[i];
        sum1 += a[i + 1] * b[i + 1];
    }
    return sum0 + sum1;
}
```

例 4-6. 内積用のリニア・アセンブリ

```
_dotp:.cproc  a, b
        .reg   sum0, sum1, i
        .reg    val1, val2, prod1, prod2
        MVK     50,i ; i = 100/2
        ZERO    sum0 ; multiply result = 0
        ZERO    sum1 ; multiply result = 0
loop:.trip 50
        LDW     *a++,val1        ; load a[0-1]  bank0
        LDW     *b++,val2        ; load b[0-1]  bank2
        MPY     val1,val2,prod1  ; a[0] * b[0]
        MPYH    val1,val2,prod2  ; a[1] * b[1]
        ADD     prod1,sum0,sum0  ; sum0 += a[0] * b[0]
        ADD     prod2,sum1,sum1  ; sum1 += a[1] * b[1]
    [i] ADD     -1,i,i           ; i--
    [i] B       loop             ; if (!i) goto loop
        ADD     sum0,sum1,A4     ; compute final result
        .return A4
        .endproc
```

例 4-7. 内積ソフトウェア・パイプライン・カーネル

```
L2:         ; PIPED LOOP KERNEL

            ADD    .L2     B7,B4,B4           ; |14| <0,7>  sum0 += a[0]*b[0]
||          ADD    .L1     A5,A0,A0           ; |15| <0,7>  sum1 += a[1]*b[1]
||          MPY    .M2X    B6,A4,B7           ; |12| <2,5>  a[0] * b[0]
||          MPYH   .M1X    B6,A4,A5           ; |13| <2,5>  a[1] * b[1]
|| [ B0]   B      .S1     L2                 ; |18| <5,2>  if (!i) goto loop
|| [ B0]   ADD    .S2     0xffffffff,B0,B0   ; |17| <6,1>  i--
||          LDW    .D2T2   *B5++,B6           ; |10| <7,0>  load a[0-1] bank0
||          LDW    .D1T1   *A3++,A4           ; |11| <7,0>  load b[0-1] bank2
```

配列と他のメモリ・オブジェクトとの位置合わせ方法を完全に制御することは、常に可能なわけではありません。これが特に当てはまるのは、ポインタが関数に渡され、関数が呼び出されるたびに、そのポインタに対する位置合わせが異なる場合です。この問題の解決方法は、メモリのヒットがない内積ルーチンを作成することです。これにより、配列が異なるメモリ・バンクを使用する必要がなくなります。

内積ループ・カーネルが一度展開される場合、4 つの LDW 命令がそのループ・カーネル内で実行されます。配列 a と b のバンク位置合わせについて何も分かっていない（ワードの位置合わせを除いて）ことを前提とすると、配列アクセスについての安全な前提事項は、a[0-1] が a[2-3] と競合しないこと、および b[0-1] が b[2-3] と競合しないことだけです。例 4-8 は、展開されたループ・カーネルを示しています。

例 4-8.　メモリ・バンクの競合を防止するように展開された内積用のリニア・アセンブリの内積

```
_dotp2: .cproc  a_0, b_0
        .reg    a_4, b_4, sum0, sum1, i
        .reg    val1, val2, prod1, prod2

        ADD     4,a_0,a_4
        ADD     4,b_0,b_4
        MVK     25,i          ; i = 100/4
        ZERO    sum0          ; multiply result = 0
        ZERO    sum1          ; multiply result = 0

        .mptr   a_0,a+0,8
        .mptr   a_4,a+4,8
        .mptr   b_0,b+0,8
        .mptr   b_4,b+4,8

loop:   .trip   25
        LDW     *a_0++[2],val1     ; load a[0-1] bankx
        LDW     *b_0++[2],val2     ; load b[0-1] banky
        MPY     val1,val2,prod1    ; a[0] * b[0]
        MPYH    val1,val2,prod2    ; a[1] * b[1]
        ADD     prod1,sum0,sum0    ; sum0 += a[0] * b[0]
        ADD     prod2,sum1,sum1    ; sum1 += a[1] * b[1]

        LDW     *a_4++[2],val1     ; load a[2-3] bankx+2
        LDW     *b_4++[2],val2     ; load b[2-3] banky+2
        MPY     val1,val2,prod1    ; a[2] * b[2]
        MPYH    val1,val2,prod2    ; a[3] * b[3]
        ADD     prod1,sum0,sum0    ; sum0 += a[2] * b[2]
        ADD     prod2,sum1,sum1    ; sum1 += a[3] * b[3]

    [i] ADD     -1,i,i             ; i--
    [i] B       loop               ; if (!0) goto loop

        ADD     sum0,sum1,A4 ; compute final result
        .return A4
        .endproc
```

この目的は、次の命令が並行しているソフトウェア・パイプラインを見付けることです。

```
     LDW *a0++[2],val1 ; load a[0-1]  bankx
||   LDW *a2++[2],val2 ; load a[2-3]  bankx+2

     LDW *b0++[2],val1 ; load b[0-1]  banky
||   LDW *b2++[2],val2 ; load b[2-3]  banky+2
```

例 4-9. 内積ソフトウェア・パイプライン・カーネル から展開された内積カーネル

```
L2:    ; PIPED LOOP KERNEL

   [ B1]  SUB    .S2     B1,1,B1       ; <0,8>
||        ADD    .L2     B9,B5,B9      ; |21| <0,8>  ^  sum0 += a[0] * b[0]
||        ADD    .L1     A6,A0,A0      ; |22| <0,8>  ^  sum1 += a[1] * b[1]
||        MPY    .M2X    B8,A4,B9      ; |19| <1,6>  a[0] * b[0]
||        MPYH   .M1X    B8,A4,A6      ; |20| <1,6>  a[1] * b[1]
|| [ B0]  B      .S1     L2            ; |32| <2,4>  if (!i) goto loop
|| [ B1]  LDW    .D1T1   *A3++(8),A4   ; |24| <3,2>  load a[2-3] bankx+2
|| [ A1]  LDW    .D2T2   *B6++(8),B8   ; |17| <4,0>  load a[0-1] bankx

   [ A1]  SUB    .S1     A1,1,A1       ; <0,9>
||        ADD    .L2     B5,B9,B5      ; |28| <0,9>  ^  sum0 += a[2] * b[2]
||        ADD    .L1     A6,A0,A0      ; |29| <0,9>  ^  sum1 += a[3] * b[3]
||        MPY    .M2X    A4,B7,B5      ; |26| <1,7>  a[2] * b[2]
||        MPYH   .M1X    A4,B7,A6      ; |27| <1,7>  a[3] * b[3]
|| [ B0]  ADD    .S2     -1,B0,B0      ; |31| <3,3>  i--
|| [ A1]  LDW    .D2T2   *B4++(8),B7   ; |25| <4,1>  load b[2-3] banky+2
|| [ A1]  LDW    .D1T1   *A5++(8),A4   ; |18| <4,1>  load b[0-1] banky
```

例 4-8 で .mptr 擬似命令を指定しないと、a[0-1] と b[0-1] のロードが並行してスケジュールされ、a[2-3] と b[2-3] のロードが並行してスケジュールされます。この結果、すべてのサイクルでメモリの競合が発生する確率は 50 パーセントになります。しかし、例 4-9 に表示されているループ・カーネルには、メモリ・バンク競合はありません。

例 4-6 では、.mptr 疑似命令を使用して、a と b が異なるベースを指すように指定しても、アセンブリ・オプティマイザは、1 サイクル・ループ・カーネルのスケジュールを見付けません。これは、常にメモリ・バンクの競合があるからです。しかし、2 サイクル・ループ・カーネルのスケジュールを見つけます。

4.5.3 インデックス・ポインタのメモリ・バンク競合

インデックス付きメモリ・アクセスのメモリ・バンク競合を判別する際には、場合によっては 1 組のメモリ・アクセスが常に競合するか、または決して競合しないことを指定しなければなりません。これを行うには、ストライドに 0 を指定した .mptr 疑似命令を使用します。

ストライドに 0 を指定すると、反復デルタに関係なくメモリ・アクセス相互間に一定の関係があることを示します。本来、アセンブリ・オプティマイザは、ベース、オフセット、幅だけを使用してメモリ・バンクの競合を判別します。ストライドはオプションであり、デフォルトでは 0 に解釈されることに注目してください。

例 4-10 では、どのメモリ・アクセスが競合し、どれが競合しないかを指定するのに、.mptr 擬似命令が使用されています。

例 4-10. インデックス付きポインタ用の .mptr の使用方法

```
    .mptr  a,RS
    .mptr  b,RS
    .mptr  c,XY
    .mptr  d,XY+2
    LDW     *a++[i0a],A0  ; a and b always conflict with each other
    LDW     *b++[i0b],B0  ;
    STH     A1,*c++[i1a]    ; c and d never conflict with each other
    STH     B2,*d++[i1b]    ;
```

4.5.4 メモリ・バンク競合アルゴリズム

アセンブリ・オプティマイザは、次のプロセスを使用して、2 つのメモリ・アクセス命令にメモリ・バンク競合があるかどうかを判別します。

1) どちらかのアクセスにメモリ・バンク情報がない場合は、競合することはありません。

2) 両方のアクセスに同じベースを指定していない場合は、競合が起きます。

3) オフセット、ストライド、アクセス幅、および反復デルタを使用して、メモリ・バンク競合が発生するかどうかを判別します。アセンブリ・オプティマイザは、アクセス・パターンの単純な分析を使用し、同じ相対バンクにアクセスするかどうかを判別します。ストライドとオフセットの値は、常にバイト単位で表示されます。

   反復デルタは、ソフトウェア・パイプラインでスケジュールされるメモリ参照のループ反復の差です。たとえば 3 つの命令 A、B、C があり、1 サイクル・カーネルのあるソフトウェア・パイプラインがあるとすると、A と C の反復デルタは 2 です。

```
A
B       A
C       B       A
        C       B
                C
```

4.6 メモリ・エイリアスの明確化

2 つの命令が同じメモリ位置にアクセスできるときに、メモリ・エイリアス指定が生じます。このようなメモリ参照は「曖昧である」と呼ばれます。メモリ・エイリアスの明確化は、こうした曖昧さが発生しない期間を判別するプロセスです。2 つのメモリ参照が曖昧であるかどうかを判別できない場合、曖昧であるとみなします。これは、2 つの命令の間にメモリ依存関係があることと同じ意味です。

命令間の依存関係は、ソフトウェア・パイプライン・スケジュールを含めて、命令のスケジュールに制約を加えます。一般に、依存関係が少なければ少ないほど、スケジュールを選択する際の自由が広がり、最終的なスケジュールのパフォーマンスが向上します。

4.6.1 アセンブリ・オプティマイザによるメモリ参照の処理方法（デフォルト）

アセンブリ・オプティマイザは、すべてのメモリ参照が常にエイリアス指定される、つまり常に互いに依存しているものと見なします。この前提は、すべての入力にとって無難です。これにより、可能なメモリ・エイリアスの処理方法を完全に制御できます。

場合によっては、この前提では控え目過ぎになります。このような場合には、前提のメモリ・エイリアスによる余分な命令の依存関係により、アセンブリ・オプティマイザが発する命令スケジュールの並列性が低く、パフォーマンスが十分でない場合があります。こうした場合に対処するために、アセンブリ・オプティマイザは 1 つのオプションと 2 つの疑似命令を備えています。

4.6.2 メモリ参照を処理するための -mt オプションの使用方法

アセンブリ・オプティマイザでは、-mt オプションはメモリ参照が互いに依存しないことを意味します。コンパイラとは -mt オプションの意味が異なります。コンパイラは、-mt オプションをいくつかの特定のケースのメモリ・エイリアス指定が発生しないことが保証されていると解釈します。-mt オプションの使用方法については、3.7.2 項「エイリアス技法を使用しないことを示す -mt オプションの使用」（3-26 ページ）を参照してください。

4.6.3 .no_mdep 疑似命令の使用方法

ユーザは、.(c)proc 関数内の任意の位置に .no_mdep 疑似命令を指定できます。この疑似命令を使用する場合は、その関数内でメモリ依存関係がないことを保証します。

注：メモリ依存関係の例外

-mt および .no_mdep 方法のどちらの場合も、アセンブリ・オプティマイザは、ユーザが .mdep 疑似命令を使用して特定したメモリ依存関係を認識します。

4.6.4 .mdep 疑似命令を使用した特定のメモリ依存関係の識別

.mdep 疑似命令を使用すると、特定のメモリ依存関係を識別できます。つまり、各メモリ参照に名前の注釈を付け、それらの名前を .mdep 疑似命令で使用して実際の依存関係を指定します。メモリ参照に注釈を付けるには、アセンブリ・コード内のメモリ参照のすぐ隣に情報を追加する必要があります。メモリ参照の直後に、次の情報を指定してください。

{*symbol***}**

このシンボルには、アセンブリ・シンボルと同じ構文の制約事項があります（シンボルの詳細は、TMS320C6000 Assembly Language Tools User's Guide を参照してください）。シンボリック・レジスタと同じネーム・スペース内にあります。シンボリック・レジスタとメモリ参照の注釈付けに同じ名前を使用できません。

例 4-11. メモリ参照の注釈付け

```
LDW    *p1++ {ld1}, inp1   ;name memory reference "ld1"
;other code ...
STW    outp2, *p2++ {st1}  ;name memory reference "st1"
```

上記の例で、特定のメモリ依存関係を指定する疑似命令は次のとおりです。

.mdep ld1, st1

これは、ld1 がメモリ・ロケーション X にアクセスする際は、コードが実行し始めてからしばらくして必ず st1 もロケーション X にアクセスする可能性があることを意味します。これは、これらの 2 つの命令間に依存関係を追加するのと同じ意味です。ソフトウェア・パイプラインの場合、これらの 2 つの命令は同じ順序のままでなければなりません。ld1 参照は、常に st1 参照の前に指定する必要があります。また、これらの命令は並列してスケジュールできません。

ld1 から st1 への疑似命令の方向に注目することが重要です。反対の方向 st1 から ld1 は、暗黙的に指定されません。ソフトウェア・パイプラインの場合、どの ld1 も st1 の前に指定する必要がありますが、反復 n+1 からの ld1 を反復 n からの st1 の前にスケジュールしても全く問題はありません。

例 4-12 はソフトウェア・パイプラインを示しています。ここでは、2 つの異なる反復に属する命令は別々のカラムに入れてあります。実際の命令シーケンスでは、同じ行にある命令は同時に並行して実行されます。

例 4-12. .mdep ld1, st1 を使用したソフトウェア・パイプライン

```
iteration n       iteration n+1
-------------     -------------
LDW { ld1 }

...               LDW { ld1 }

STW { st1 }       ...

                  STW { st1 }
```

反復 n+1 の ld1 が読み取るべき値を反復 n の st1 が書き込む可能性があるため、そのスケジュールが機能しない場合は、st1 から ld1 への依存関係に注目する必要があります。

```
.mdep   st1, ld1
```

両方の疑似命令が一緒に、例 4-13 に示されているソフトウェア・パイプラインを強制実行します。

例 4-13. .mdep st1, ld1 および .mdep ld1, st1 を使用したソフトウェア・パイプライン

```
iteration n       iteration n+1
-------------     -------------
LDW { ld1 }

...

STW { st1 }

                  LDW { ld1 }

                  ...

                  STW { st1 }
```

インデックス・アドレッシングの *+base[index] は、メモリ・アクセスが同じ位置にアクセスする場合を除いて、一般にメモリ・アクセスの相対的な順序が分かっていないアドレッシング・モードの適切な例です。このケースを正しくモデル化するには、両方向の依存関係に注目し、両方の疑似命令を使用しなければなりません。

```
.mdep   ld1, st1
.mdep   st1, ld1
```

4.6.5 メモリ・エイリアス例

次の例では、.mdep および .no_mdep 疑似命令を使用したメモリ・エイリアスを示しています。

❑ 例 1

.mdep r1, r2 疑似命令は、LDW が STW の前でなければならないことを宣言します。この場合、src と dst は同じ配列をポイントする可能性があります。

```
fn:    .cproc      dst, src, cnt
       .reg        tmp
       .no_mdep
       .mdep       r1, r2

       LDW         *src{r1}, tmp
       STW         cnt, *dst{r2}

       .return     tmp
       .endproc
```

❑ 例 2

この例では、.mdep r2, r1 は、STW が LDW の前に指定されなければならないことを示しています。コード内で STW が LDW の後ろにあるので、依存関係はループ反復にまたがります。STW 命令は、次の反復の LDW 命令が読み取る値を書き込みます。この場合は 6 サイクルの繰り返しが作成されます。

```
fn:    .cproc     dst, src, cnt
       .reg       tmp
       .no_mdep
       .mdep      r2, r1

LOOP:  .trip      100
       LDW        *src++{r1}, tmp
       STW        tmp, *dst++{r2}
[cnt]  SUB        cnt, 1, cnt
[cnt]  B          LOOP

       .endproc
```

注 : メモリ依存関係／バンク競合

メモリ・エイリアスの明確化のトピックとメモリ・バンク競合の処理を混同しないでください。どちらもメモリ参照と命令のスケジュール上のメモリ参照の影響を取り扱うため同じように見えますが、エイリアスの明確化は正確さの問題であり、バンク競合はパフォーマンスの問題です。命令のスケジュール上、メモリの依存関係はバンク競合よりも幅広い影響を与えます。これらの 2 つのトピックを別のものとして取り扱うことが最良です。

第 5 章

C/C++コードのリンク

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。
- ❑ コンパイルとリンクを 1 ステップで実行できます。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・ライブラリの取り込み方法、初期化モデルの指定方法、プログラムをメモリに割り振る方法についても説明します。リンカの詳細は、TMS320C6000 Assembly Language Tools User's Guide を参照してください。

項目 **ページ**

5.1	コンパイラを使用してリンカを起動する方法（-z オプション）	5-2
5.2	リンカ・オプション	5-5
5.3	リンク・プロセスの制御方法	5-8

5.1 コンパイラを使用してリンカを起動する方法（-z オプション）

この節では、プログラムをコンパイルまたはアセンブリした後で、独立したステップとして、あるいはコンパイル・ステップの一部としてリンカを起動する方法を説明します。

5.1.1 独立したステップで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。

cl6x -z {**-c**|**-cr**} *filenames* [*options*] [**-o** *name*.**out**] **-l** *library* [*lnk.cmd*]

cl6x -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伝えるオプションです。cl6x -z を使用する場合は、**-c** または **-cr** を使用する必要があります。-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、コマンド行では **-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、5.2 節「リンカ・オプション」（5-5 ページ）を参照してください）。

-o *name.out* 出力ファイル名を指定します。

-l *library* （小文字の L）は、C/C++ ランタイム・サポート関数、浮動小数点算術関数、またはリンカ・コマンド・ファイルを含んだ適切なアーカイブ・ライブラリを識別します。C/C++ コードをリンクする場合は、ランタイム・サポート・ライブラリを使用しなければなりません。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・ライブラリを指定した場合、このパラメータは必要ありません。

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。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。詳細は、TMS320C6000 Assembly Language Tools User's Guide を参照してください。

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います。

```
cl6x -z -c prog1 prog2 prog3 -o prog.out -l rts6200.lib
```

5.1.2 コンパイル・ステップの一部で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・ステップの一部で実行する場合の一般的な構文を、次に示します。

cl6x *filenames* [*options*] **-z** {**-c**|**-cr**} *filenames* [*options*] [**-o** *name*.out] **-l** *library* [*lnk.cmd*]

-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、コマンド行ではすべてのファイル・オプションとコンパイラ・オプションに続いて指定する必要があります。

コマンド行の -z の後ろに指定した引数は、すべてリンカへ渡されます。それらの引数には、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、またはライブラリを指定できます。これらの引数は、5.1.1 項「独立したステップでリンカを起動する方法」（5-2 ページ）で説明したものと同じです。

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。これらの引数には、C/C++ ソース・ファイル、アセンブリ・ファイル、リニア・アセンブリ・ファイル、またはコンパイラ・オプションを指定できます。これらの引数は、2.2 節「C/C++ コンパイラの起動方法」（2-4 ページ）で説明したものと同じです。

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。

```
cl6x prog1.c prog2.c prog3.c -z -c -o prog.out -l rts6200.lib
```

注：リンカにおいて引数を処理する順序

リンカが引数を処理する順序は重要です。コンパイラは、次の順序で引数をリンカに渡します。

1） コマンド行から入力されたオブジェクト・ファイル名

2） コマンド行の -z オプションの後ろにある引数

3） C_OPTION または C6X_C_OPTION 環境変数の -z オプションの後ろにある引数

5.1.3　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 または C6X_C_OPTION 環境変数の中で -z オプションを指定しているとき、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。ロード時に変数を自動初期化する場合には、-z オプションの後ろに -cr リンカ・オプシ ョンを指定します。

5.2 リンカ・オプション

-z オプションの後ろ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、その効果についての詳しい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。出力モジュールには、特殊なリンカ・シンボル、オプション・ヘッダ、およびすべてのシンボル参照が含まれます。再配置情報は保持されます。
--args=*size*	ローダがコマンド行からプログラムへ引数を渡すために使用するメモリを割り当てます。リンカは、初期化されていない .args セクションに *size* バイトを割り当てます。 _ _c_args_ _ シンボルは、.args セクションのアドレスを含んでいます。
-b	シンボリック・デバッグ情報のマージを抑止します。リンカは、通常デバッグ用に C プログラムをコンパイルしたときに生成されるシンボリック・デバッグ情報の重複エントリを保持します。
-c	実行時に変数を自動初期化します。詳細は、8.8.4 項（8-56 ページ）を参照してください。
-cr	ロード時に変数を初期化します。詳細は、8.8.5 項（8-57 ページ）を参照してください。
-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ホールを満たすデフォルト埋め込み値を設定します。*fill_value* は 32 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。グローバル・シンボルは実質的には隠されます。これにより、（別のファイルの）同じ名前の外部シンボルを一意のものとして扱うことができます。

-heap *size*
ヒープ・サイズ（動的メモリ割り当て）を *size* で指定したバイト数に設定し、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 1K バイトです。

-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 個まで指定できます。

-**j**
.clink アセンブラ疑似命令で設定されている条件付きリンクを抑止します。デフォルトでは、すべてのセクションは無条件にリンクされます。

-l *libraryname*
（小文字の L）は、アーカイブ・ライブラリ・ファイルまたはリンカ・コマンド・ファイルの名前をリンカへの入力として指定します。*libraryname* はアーカイブ・ライブラリの名前であり、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。

-m *filename*
ホール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。

-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**
各未解決参照をそのシンボルの定義を含む最初のライブラリによって解決します。

-**q**
静的な実行（見出しの抑止）を要求します。

-**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て、小さい出力セクションを作成します。

-stack *size*
C/C++ システム・スタック・サイズを *size* で指定したバイト数に設定し、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 バイトです。

--trampolines
呼び出された宛先の範囲外にリンクする各呼び出しに対してトランポリン・コード・セクションを生成します。トランポリン・コード・セクションには、呼び出される元のアドレスへの透過的な長い分岐を実行する命令のシーケンスが含まれます。呼び出された関数の範囲外の各呼び出し命令は、トランポリンにリダイレクトされます。

-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。これにより、ライブラリの検索とシンボルを定義するメンバの組み込みが強制的に行われます。

-w
リンカが、SECTIONS 疑似命令で定義された対応する出力セクションをもたない 1 つ以上の入力セクションを検出したとき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、これ以上参照を解決できなくなるまでライブラリの再読み取りを続けます。

--xml_link_info *file*
XML リンク情報ファイルを生成します。このオプションを使うと、リンカはリンクの結果に関する詳細情報を含む適格な XML ファイルを生成します。このファイルの情報には、リンカ生成マップ・ファイルで現在作成されている情報がすべて入っています。

リンカ・オプションの詳細は、TMS320C6000 Assembly Language Tools User's Guide の「Linker Description」の章を参照してください。

5.3 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。

- ❏ コンパイラのランタイム・サポート・ライブラリを含める
- ❏ 初期化モデルを指定する
- ❏ プログラムをメモリに割り当てる方法を決定する

この節では、これらの要件を制御する方法について説明し、標準的なデフォルトのリンカ・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、TMS320C6000 Assembly Language Tools User's Guide の「Linker Description」を参照してください。

5.3.1 ランタイム・サポート・ライブラリとのリンク方法

すべての C/C++ プログラムはブートストラップ・ルーチンと呼ばれるプログラムを初期化し、実行するコードとリンクする必要があります。このルーチンは *boot.obj* オブジェクト・モジュールとも呼ばれています。ランタイム・サポート・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。-l リンカ・オプションを使用して、使用する C6000 ランタイム・サポート・ライブラリ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 や C6X_C_DIR 環境変数に注目するように伝えます。-l オプションを使用するには、次のようにコマンド行に入力します。

cl6x -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。ライブラリの後ろにオブジェクト・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、ライブラリが未解決の参照を導入し、複数のライブラリにそれに対する定義がある場合、未解決の参照を導入した同じライブラリの定義が使用されます。リンカがこの定義を含むコマンド行の最初のライブラリの定義を使用するようにしたい場合は、-priority オプションを使用します。

5.3.2 実行時の初期化

すべての C/C++ プログラムはブートストラップ・ルーチンと呼ばれるプログラムを初期化し、実行するコードとリンクする必要があります。このルーチンは boot.obj オブジェクト・モジュールとも呼ばれています。C/C++ プログラムは、実行を開始するときは最初に boot.obj を実行します。boot.obj モジュールには、ランタイム環境を初期化するためのコードとデータが入っています。リンク時に -c を使用し、適切なランタイム・サポート・ライブラリをインクルードすると、リンカは自動的に boot.obj を抽出してリンクします。

以下にリストしたアーカイブ・ライブラリには、C/C++ ランタイム・サポート関数が入っています。

rts6200.lib	rts6400.lib	rts6700.lib
rts6200e.lib	rts6400e.lib	rts6700e.lib

boot.obj モジュールには、ランタイム環境を初期化するためのコードとデータが入っています。このモジュールは次の作業を実行します。

1) スタック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) *.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 オプションを使用している場合）。
3) すべてのグローバル・コンストラクタを呼び出します（*.pinit*）。
4) main を呼び出します。
5) main が戻ったときに exit を呼び出します。

ランタイム・サポート・オブジェクト・ライブラリには boot.obj が入っています。次の作業を実行できます。

- ❏ アーカイバを使用してライブラリから boot.obj を 抽出し、そのモジュールを直接リンクできます。
- ❏ 入力ファイルとして、適切なランタイム・サポート・ライブラリを指定できます（-c または -cr オプションを使用すると、リンカは boot.obj を自動的に抽出します）。

サンプルのブートストラップ・ルーチンは、ランタイム・サポート・ライブラリの boot.obj に入っている _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ランタイム・サポートの追加の関数については、第 9 章で説明しています。これらの関数には、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

ランタイム・サポート・ライブラリに含まれている重要な関数の 1 つに _c_int00 があります。_c_int00 シンボルは boot.obj の開始点です。-c または -cr リンカ・オプションを使用する場合、_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。プログラムをリセットから実行する場合は、プロセッサが最初に boot.obj を実行するように、リセットが _c_int00 への分岐を指定するように設定する必要があります。

5.3.3 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もつグローバル C++ 変数は、コンストラクタに対して、プログラムの初期化時に呼び出されるように要求します。またデストラクタに対して、プログラムの終了時に呼び出されるように要求します。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、テーブル内に発生した順に起動されます。

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。

8.8.3 項「初期化テーブル」（8-53 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。

5.3.4 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。8.8.3 項「初期化テーブル」（8-53 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。

- グローバル変数は実行時に初期化されます。-c リンカ・オプションを使用してください（8.8.4 項「実行時の変数の自動初期化」（8-56 ページ）を参照）。
- グローバル変数はロード時に初期化されます。-cr リンカ・オプションを使用してください（8.8.5 項「ロード時の変数の初期化」（8-57 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、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伝えます。

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・オプションを使用する場合は、-z オプションの後に指定しなければなりません。（5.1 節「コンパイラを使用してリンカを起動する方法（-z オプション）」（5-2 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 を使用すると _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通知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。

5.3.5 セクションをメモリ内のどこに割り振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振られます。

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基本的な 2 種類のセクションがあります。表 5–1 は、それらのセクションをまとめたものです。

表 5–1. 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
.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
.const	明示的に初期化され、文字列リテラルを含むグローバル変数と静的な const 変数
.pinit	始動時に呼び出されるコンストラクタのテーブル
.switch	長い switch 文用のジャンプ・テーブル
.text	実行可能なコードと定数

（b）初期化されないセクション

名前	内容
.bss	グローバル変数および静的変数
.far	far 宣言されたグローバル変数と静的変数
.stack	スタック
.sysmem	malloc 関数（heap）用のメモリ

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振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。.text は例外として、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を内部プログラム・メモリ内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、8.1.1 項「セクション」（8-2 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。セクションをメモリに割り振る方法の詳細は、TMS320C6000 Assembly Language Tools User's Guide のリンカの章を参照してください。

5.3.6 リンカ・コマンド・ファイルの例

例 5–1 は、C プログラムをリンクする典型的なリンカ・コマンド・ファイルを示しています。この例のコマンド・ファイルは lnk.cmd という名前で、いくつかのリンカ・オプションのリストを示しています。

-c リンカに対して、実行時に自動初期化を使用するように指示します。

-heap リンカに対して、C ヒープ・サイズを 0x2000 バイトに設定するように指示します。

-stack リンカに対して、スタック・サイズを 0x0100 バイトに設定するように指示します。

-l リンカに対して、入力にアーカイブ・ライブラリ・ファイル rts6200.lib を使用するように指示します。

このプログラムをリンクするには、以下の構文を使用します。

cl6x -z *object_file(s)* **-o** *outfile* **-m** *mapfile* **lnk.cmd**

使用するシステムで機能させるには、MEMORY 疑似命令と、可能ならば SECTIONS 疑似命令に修正を加える必要があります。これらの疑似命令の詳細は、TMS320C6000 Assembly Language Tools User's Guide を参照してください。

例 5–1. リンカ・コマンド・ファイルの例

```
-c
-heap 0x2000
-stack 0x0100
-l rts6200.lib

MEMORY
{
    VECS:o = 00000000h       l = 00400h /* reset & interrupt vectors     */
    PMEM:o = 00000400h       l = 0FC00h /* intended for initialization   */
    BMEM:o = 80000000h       l = 10000h /* .bss, .sysmem, .stack, .cinit */
}

SECTIONS
{
    vectors     >       VECS
    .text       >       PMEM
    .data       >       BMEM
    .stack      >       BMEM
    .bss        >       BMEM
    .sysmem     >       BMEM
    .cinit      >       BMEM
    .const      >       BMEM
    .cio        >       BMEM
    .far        >       BMEM
}
```

5.3.7 関数のサブセクションを使用する方法（-mo コンパイラ・オプション）

リンカは、コードを実行可能ファイルの中に置くときに、1 つのソース・ファイルに入っている関数のすべてを 1 つのグループとして割り当てます。つまり、ファイル内のどのような関数であっても実行可能プログラムにリンクされる必要がある場合には、そのファイル内の関数すべてがリンクされます。このことは、あるファイルが多数の関数を含んでいるが、実行可能プログラム用に必要な関数は少ないという場合には、望ましいことではありません。

このような状況は、1 つのファイルに複数の関数が含まれているが、アプリケーションはそれらの関数のサブセットを必要としているだけというライブラリにおいて存在する可能性があります。符号付き除算ルーチンと符号なし除算ルーチンをもつライブラリ .obj ファイルがその一例です。アプリケーションが符号付き除算のみを要求する場合には、リンク用に要求されるのは符号付き除算ルーチンだけです。デフォルトでは、符号付きルーチンと符号なしルーチンは同じ .obj ファイル内に存在するので、どちらもリンクされます。

-mo コンパイラ・オプションは、ファイル内の各関数をファイル自身のサブセクションに配置することによりこの問題を改善します。したがって、アプリケーション上で参照される関数だけが最終の実行可能プログラムにリンクされます。この結果、コード・サイズ全体を縮小することができます。

ただし、-mo コンパイラ・オプションを使用することにより、すべてまたはほとんどすべての関数が参照される場合には、コード・サイズ全体が増大することを承知しておいてください。これは、コードを含んでいるセクションはすべて、C6000 の分岐メカニズムをサポートするために 32 バイトの境界に位置合わせされる必要があるからです。-mo オプションを使用しないと、ソース・ファイル内のすべての関数は通常、位置合わせされた共通セクション内に指定されます。-mo を使用すると、ソース・ファイル内で定義された各関数は固有のセクション内に指定されます。それぞれの固有セクションは、位置合わせを要求します。ファイル内のすべての関数がリンク用に要求される場合、コード・サイズは、個別のサブセクションに対する追加の位置合わせ埋め込みによって増加する可能性があります。

したがって -mo コンパイラ・オプションは、どれか 1 つの実行可能プログラム上でファイル内の限られた数の関数だけが正常に使用される場所で、ライブラリと一緒に使用すると便利です。

-mo オプションの代替は、自身のファイル内にそれぞれの関数を置くことです。

第 6 章

スタンドアロン・シミュレータの使用方法

TMS320C6000 スタンドアロン・シミュレータは、実行可能な COFF .out ファイルをロードし、実行します。C 入出力ライブラリと同時に使用する場合は、スタンドアロン・シミュレータは、スクリーンへの標準出力を含めてすべての C 入出力関数をサポートします。

スタンドアロン・シミュレータを使用すると、clock 関数を使用してプログラムについての統計情報を収集できます。スタンドアロン・シミュレータのその他の利点は、バッチ・ファイルで使用できること、およびコード生成ツールに含まれていることです。

本章では、スタンドアロン・シミュレータの起動方法について説明します。また、C コードの例とスタンドアロン・シミュレータの結果も記載します。

項目 **ページ**

6.1 スタンドアロン・シミュレータの起動方法 6-2
6.2 スタンドアロン・シミュレータのオプション 6-4
6.3 ローダを使用してプログラムに引数を渡す方法 6-6
6.4 スタンドアロン・シミュレータのプロファイル機能の使用方法 6-8
6.5 シミュレートするシリコン・リビジョンの選択方法（-rev オプション） 6-9
6.6 スタンドアロン・シミュレータの例 6-10

6.1 スタンドアロン・シミュレータの起動方法

この節では、スタンドアロン・シミュレータを起動して実行可能な COFF .out ファイルをロードし、実行する方法を示します。次の構文は、スタンドアロン・シミュレータを起動するための一般的な構文です。

load6x [*options*] *filename.out*

load6x スタンドアロン・シミュレータを起動するコマンドです。

options スタンドアロン・シミュレータの動作および .out ファイルの処理に影響を与えるオプションです。このオプションは、コマンド行の任意の場所に指定できます（オプションについては、6.2 節「スタンドアロン・シミュレータのオプション」（6-4 ページ）を参照してください。）

filename.out スタンドアロン・シミュレータにロードされる .out ファイルの名前を指定します。.out ファイルは、実行可能な COFF ファイルでなければなりません。

スタンドアロン・シミュレータは C6200、C6400、および C6700 用のファイルを実行できます。浮動小数点または固定小数点 .out ファイルの種類を指定するには、オプションは不要です。スタンドアロン・シミュレータは、.out ファイル内の COFF フラグを読み取って、ターゲット・バージョンを判別します。

スタンドアロン・シミュレータの起動時に生成される見出しは、.out ファイルをロードし実行するために使用する値（メモリ・マップ、シリコン・リビジョン、load6x の高速バージョンまたは低速バージョンなど）を定義します。例 6–1 は、見出しの 2 つの例を記載しています。

例 6–1. スタンドアロン・シミュレータの見出しの例

(a) オプションを指定せずに起動されたファイル clock.out

```
load6x clock.out
TMS320C6x Standalone Simulator Version X.X
Copyright (c) 1989-2000 by Texas Instruments Incorporated
OPTIONS -- C6xxx Simulator
OPTIONS -- REVISION 2
OPTIONS -- MAP 1 *** DEFAULT MEMORY MAPPING ***
NOTE    :For details on above options please refer to the readme.1st
Loading t.out
 174 Symbols loaded
Done
Interrupt to abort . . .
Hello, world
Time = 133 cycles
NORMAL COMPLETION:9873 cycles
```

(b) -a オプションを指定して起動されたファイル clock.out

```
load6x -a clock.out
TMS320C6x Standalone Simulator Version X.X
Copyright (c) 1989-2000 by Texas Instruments Incorporated
OPTIONS -- C6xxx Memory Hierarchy Modeling Simulator
OPTIONS -- REVISION 2
OPTIONS -- MAP 1 *** DEFAULT MEMORY MAPPING ***
WARNING :Ensure that map modes for linker.cmd file and load6x are same!!
NOTE    :For details on above options please refer to the readme.1st
Loading t.out
 174 Symbols loaded
Done
Interrupt to abort . . .
Hello, world
Time = 7593 cycles
NORMAL COMPLETION:98705 cycles
```

6.2 スタンドアロン・シミュレータのオプション

スタンドアロン・シミュレータを制御するオプションと、その効果についての説明を以下に示します。

-a データ・メモリのバンク競合検査を有効にします。

-b .bss セクション（データ）内のすべてのメモリを 0 で初期設定します。C 言語では、初期設定されていないすべての静的記憶クラスの変数は、プログラムの先頭で 0 に初期設定されます。コンパイラは初期設定されていない変数を設定しないので、-b オプションを使用するとこれらの変数を初期設定できます。

-d[d] 詳細モードを有効にします。低レベルでの入出力を記述する内部状況メッセージを出力します。さらに詳細な情報については、-dd を使用してください。

-f *value* .bss セクション（データ）内のすべてのメモリを指定した *value* で初期設定します。この *value* は、32 ビットの定数（最高 8 桁の 16 進数字）です。たとえば `load6x -f 0xabcdabcd` は、.bss セクションを 16 進値 abcdabcd で埋め込みます。

-g プロファイル・モードを有効にします。スタンドアロン・シミュレータでプロファイルを動作させるには、少なくともシンボリック・デバッグ情報のデフォルト・レベルでソース・ファイルをコンパイルする必要があります。詳細は、6.4 節「スタンドアロン・シミュレータのプロファイル機能の使用方法」（6-8 ページ）を参照してください。

-h スタンドアロン・シミュレータに使用可能なオプションのリストを出力します。

-i 実行不可能な場合でも、出力ファイルをロードします。

-map *value* メモリ・マップを選択します。`value` は、メモリ・マップ 0 の場合（内部プログラム・メモリが 0x1400000 で始まる場合）は 0、メモリ・マップ 1 の場合は 1 にすることができます。デフォルトでは、メモリ・マップ 1 が使用されます。-q オプションを使用しない場合、load6x の見出しには、選択されたメモリ・マップがリスト表示されます。

-o *xxx* 全体のタイムアウトを *xxx* 分に設定します。ロードされたプログラムが *xxx* 分後に終了しない場合、スタンドアロン・シミュレータは処理を打ち切ります。

-q 静的な実行（見出しの抑止）を要求します。

-r *xxx* ロード時に *xxx* バイト分、すべてのセクションを再配置します。再配置の詳細は、TMS320C6000 Assembly Language Tools User's Guide のリンカの章を参照してください。

-rev *value* シミュレートするシリコン・リビジョンを選択します。*value* はリビジョン 2 の場合は 2、リビジョン 3 の場合は 3 にすることができます。デフォルトでは、リビジョン 2 のシリコンがシミュレートされます。詳細は、6.5 節「シミュレートするシリコン・リビジョンの選択方法（-rev オプション）」（6-9 ページ）を参照してください。

-s タイム・スタンプを出力します。

-t *xxx* タイムアウトを *xxx* 秒に設定します。*xxx* 秒間、入出力イベントが発生しない場合、スタンドアロン・シミュレータは処理を打ち切ります。入出力イベントにはシステム呼び出しが含まれます。

-z 各内部入出力エラーの発生後、一時停止します。EOF の場合は一時停止しません。

6.3 ローダを使用してプログラムに引数を渡す方法

通常、コマンド行ツールでは、コマンド行からプログラムに引数を渡すことができます。次に例を示します。

例 6–2. コマンド行から引数を渡す方法

```
cl6x -a -b -c -d file.c
```

C は argc および argv 引数を通じて main に伝えるという、引数とプログラムが通信する標準的なメカニズムを備えています。C では、次のように main 関数を 2 つの引数をとるように宣言することができます。

```
int main(int argc, char *argv[])
```

コマンド名を含めて、例 6–2 には次の 6 つの引数があります。

- ❏ cl6x
- ❏ -a
- ❏ -b
- ❏ -c
- ❏ -d
- ❏ file.c

コマンド行の引数の数は argc に格納されます。引数を含む文字列へのポインタの配列は argv に格納されます。

6.3.1 引数が影響を及ぼすプログラムを判別する方法

ローダおよびランタイム環境では、ターゲット・プログラムがコマンド行から直接実行された場合と同じように、コマンド行の引数をロードするプログラムに渡すことができます。引数を渡す構文は次のとおりです。

load6x [*options*] *filename.out* [*options*]

オブジェクト・ファイル名の前にあるコマンド行オプションは、ローダの引数として扱われます。ローダは、オブジェクト・ファイル名の後にあるコマンド行オプションをロードするプログラムに対するコマンド行として扱います。たとえば次のとおりです。

```
load6x -q -x my_program.out -a -b -c
```

この例では、-q および -x は load6x の引数であり、-a、-b、および -c はロードするプログラム（ターゲット上の）に渡す引数です。したがって、この場合に *my_program* をロードするときは、ブート・コードで argc と argv 値を作成し、これらを main 関数に渡す必要があります。argc 値が 4 となり、argv が 4 つの文字列 my_program.out、-a、-b、および -c へのポインタの配列となります。

6.3.2 引数を格納するターゲット・メモリの予約方法（--args リンカ・オプション）

ブート・コードで情報を main に渡せるようにするには、ホストの load6x プログラムの引数をターゲット・システムに渡す必要があります。このためには、文字列と文字列へのポインタへの配列を格納するターゲット・メモリが必要です。リンカの --args=*size* オプションは、ローダがメモリを使用して argv 配列および argc 変数のすべての内容を格納できるように、ターゲット上にメモリを割り当てることをリンカに指示します。

ローダのコマンド行で渡されるすべての引数に対応できるように、十分な大きさの *size* を指定する必要があります。例 6–2 の場合、C6x には 4 つの引数と 4 つの文字列があります。C 標準では、argv はすべての正当な引数の後に追加 NULL ポインタをもつ必要があると規定されています。例 6–2 の場合、次のとおり合計 48 バイトが必要になります。

5 つのポインタの配列	20 バイト
文字列 my_program.out、-a、-b、および -c 用の空間	24 バイト
argc 用の空間	4 バイト

--args を指定しない場合、または --args を通じて割り当てられる空間が十分にない場合、ローダは必要な推奨サイズを含む警告メッセージを出力します。

6.4 スタンドアロン・シミュレータのプロファイル機能の使用方法

-g オプションを指定して load6x を起動すると、スタンドアロン・シミュレータはプロファイル・モードで動作します。ソース・ファイルは、少なくともシンボリック・デバッグ情報のデフォルト・レベルでコンパイルする必要があります。プロファイルの結果は、Code Composer Studio デバッガのプロファイラで指定される結果に似ています。プロファイルの結果は、.out ファイルと同じ名前で拡張子 .vaa をもつテキスト・ファイルに保管されます。

たとえば file.vaa と呼ばれるプロファイル情報ファイルを作成するには、次のように入力します。

```
load6x -g file.out
```

例 6–3 は、内積ルーチンの 3 種類のバージョンを実行し、各ルーチンの結果を出力しています。

例 6–3. 内積ルーチンのプロファイル

```
load6x -q -g t.out
val1 = 11480
val2 = 11480
val3 = 11480

<t.vaa>
Program Name:/c6xcode/t.out
Start Address:  01409680 main, in line 46, "/c6xcode/t.c"
Stop Address:   014001c0   exit
Run Cycles:     17988
Profile Cycles: 17988
BP Hits:        61

Name      Count Inclusive Incl-Max Exclusive Excl-Max  Address  Size   Full Name
dot_prod1     1      1024     1024      1024     1024 01409c20   168 _dot_prod1
dot_prod2     1       842      842       842      842 01409b20   232 _dot_prod2
main          1     17980    17980       125      125 01409680   576       _main
dot_prod3     1        89       89        89       89 01498ce0   124 _dot_prod3
```

6.5 シミュレートするシリコン・リビジョンの選択方法（-rev オプション）

新しいシリコン・リビジョンオプションを使用すると、スタンドアロン・シミュレータは C6000 シリコンのリビジョン 2 と 3 の両方をサポートできます。デフォルトでは、スタンドアロン・シミュレータはリビジョン 2 シリコンをシミュレートします。

load6x -rev *value* `file`.out

有効な値は、リビジョン 2 シリコンを選択する場合は 2 であり、リビジョン 3 シリコンを選択する場合は 3 です。リビジョン 3 シリコンでは内部データ・メモリが 2 つのメモリ・スペース（0x8000000-0x80007fff および 0x800800-0x800ffff）に分割されているので、その半分にアクセスしようとする場合、同じメモリ・バンクにアクセスできます。たとえば次のとおりです。

```
     MVK    .S2    0x80000000, B5
     MVKH   .S2    0x80000000, B5
     MVK    .S1    0x80008000, A5
     MVKH   .S1    0x80008000, A5
     LDW    .D2    *B5, B6
||   LDW    .D1    *A5, A6
```

この例では、並行した LDW 命令がリビジョン 3 シリコンではメモリ・バンクの競合を引き起こしませんが、リビジョン 2 シリコンでは競合を引き起こします。

リビジョン 3 シリコンについて 2 つのメモリ・スペースをもつインターリーブド・メモリの例については、図 4–2（4-34 ページ）を参照してください。

-q オプションを使用しない場合、load6x の見出しには、選択されたシリコン・リビジョンが表示されます。

6.6 スタンドアロン・シミュレータの例

スタンドアロン・シミュレータの典型的な用途は、clock 関数を含むコードを実行し、コードの実行に必要なサイクル数を検出することです。データを画面に表示するには、printf 文を使用してください。例 6–4 は、これを実現するための C コードの例を示しています。

例 6–4. Clock 関数を使用した C コード

```
#include <stdio.h>
#include <time.h>
main()
{
    clock_t start;
    clock_t overhead;
    clock_t elapsed;
    /* Calculate the overhead from calling clock() */
    start    = clock();
    overhead = clock() - start;
    /* Calculate the elapsed time */
    start   = clock();
    puts("Hello, world");
    elapsed = clock() - start - overhead;
    printf("Time = %ld cycles\n", (long)elapsed);
}
```

例 6–4 のコードをコンパイルし、リンクする場合は、コマンド行で次のテキストを入力してください。-z オプションはリンカを起動し、-o リンカ・オプションは出力ファイルの名前を指定します。

```
cl6x clock.c -z -l lnk60.cmd -o clock.out
```

その結果作成される実行可能な COFF ファイルでスタンドアロン・シミュレータを実行するには、次のように入力します。

```
load6x clock.out
```

例 6–5. 例 6–4 のコンパイルとリンク後のスタンドアロン・シミュレータの結果

```
TMS320C6x Standalone Simulator   Version x.xx
Copyright (c) 1989-2000 Texas Instruments Incorporated
Interrupt to abort . . .
Hello, world
Time = 3338 cycles
NORMAL COMPLETION:11692 cycles
```

第 7 章

TMS320C6000 C/C++ 言語の実装

TMS320C6000 C/C++ コンパイラは、C プログラム言語を標準化する目的で ISO（米国規格協会）の委員会で制定された C/C++ 言語規格をサポートしています。

C6000 によりサポートされる C++ 言語は、一部の例外付きで ISO/IEC 14882-1998 規格により定義されています。

項目 **ページ**

7.1	TMS320C6000 C の特性	7-2
7.2	TMS320C6000 C++ の特性	7-5
7.3	データ型	7-6
7.4	キーワード	7-7
7.5	レジスタ変数およびパラメータ	7-16
7.6	asm 文	7-17
7.7	プラグマ疑似命令	7-18
7.8	リンク名の生成	7-33
7.9	静的変数とグローバル変数の初期化方法	7-34
7.10	ISO C 言語モードの変更	7-36

7.1 TMS320C6000 C の特性

ISO C は、カーニハンとリッチーによる The C Programming Language（初版）で述べられた事実上の標準 C 規格に代わるものです。この ISO 規格は、米国規格協会（American National Standard for Information Systems）の Programming Language C X3.159-1989 で説明されています。The C Programming Language の第 2 版は ISO 標準に基づくものであり、本書でも参照されています。ISO C は、最近のさまざまな C コンパイラが備えている言語拡張機能の多くを取り込み、以前に仕様が定まっていなかった言語の多数の特性を正式な仕様としています。

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 言語のいくつかの事項について記載しています。これらの機能は、効率性や実用性の理由で、各標準コンパイラの間でも異なる可能性があります。この節では、これらの機能が C6000 C/C++ コンパイラにどのように実装されているかについて説明します。

以下に、これらの事項のすべてを取り上げ、それぞれに対する C6000 C/C++ の動作について説明します。それぞれの説明には、さらに詳しい情報への参照も含まれています。参照の多くは、正式な ISO 規格、またはカーニハンとリッチー（K&R）による The C Programming Language の第 2 版に対するものです。

7.1.1 識別子と定数

- ❑ 識別子に関しては、すべての文字が有効です。また、各識別子ごとに大文字と小文字が区別されています。これらの特性は、内部および外部を問わずすべての識別子に適用されます。 (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。 (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。 (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'`(ISO 3.1.3.4, K&R A2.5.2)

7.1.2 データ型

- ❑ データ型表記方法については、7.3 節「データ型」（7-6 ページ）を参照してください。 (ISO 3.1.2.5, K&R A4.2)

- ❑ size_t 型（*sizeof* 演算子の結果）は unsigned int です。 (ISO 3.3.3.4, K&R A7.4.8)

- ❑ ptrdiff_t 型（ポインタ減算の結果）は int です。 (ISO 3.3.6, K&R A7.7)

7.1.3 変換

- ❑ float から int への変換では、0 までの切り捨てが行われます。 (ISO 3.2.1.3, K&R A6.3)

- ❑ ポインタと整数は、自由に変換できます。 (ISO 3.3.4, K&R A6.6)

7.1.4 式

- ❑ 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。スラッシュ記号（/）は商を求めるために使用し、パーセント記号（%）は剰余を求めるために使用します。たとえば、次のとおりです。

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

(ISO 3.3.5, K&R A7.6)

符号付きの剰余演算は、被除数（第 1 オペランド）の符号をとります。

- ❑ 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。 (ISO 3.3.7, K&R A7.8)

7.1.5 宣言

- ❏ *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。詳細は、7.5 節「レジスタ変数およびパラメータ」(7-16 ページ) を参照してください。(ISO 3.5.1, K&R A2.1)

- ❏ 構造体のメンバは、ワードにパックされます。 (ISO 3.5.2.1, K&R A8.3)

- ❏ 整数として定義されたビット・フィールドは、符号付きです。ビット・フィールドはワードにパックされ、ワード境界をまたがることはありません。ビット・フィールドのパックの詳細は、8.2.2 項「ビット・フィールド」(8-15 ページ) を参照してください。 (ISO 3.5.2.1, K&R A8.3)

- ❏ interrupt キーワードは、引数を持たない void 関数にのみ適用できます。interrupt キーワードの詳細は、7.4.3 項「interrupt キーワード」(7-10 ページ) を参照してください。

7.1.6 プリプロセッサ

- ❏ プリプロセッサは、サポートされていない #pragma 疑似命令をすべて無視します。(ISO 3.8.6, K&R A12.8)

  以下のプラグマが、サポートされています。

  - CODE_SECTION
  - DATA_ALIGN
  - DATA_MEM_BANK
  - DATA_SECTION
  - FUNC_CANNOT_INLINE
  - FUNC_EXT_CALLED
  - FUNC_INTERRUPT_THRESHOLD
  - FUNC_IS_PURE
  - FUNC_IS_SYSTEM
  - FUNC_NEVER_RETURNS
  - FUNC_NO_GLOBAL_ASG
  - FUNC_NO_IND_ASG
  - INTERRUPT
  - MUST_ITERATE
  - NMI_INTERRUPT
  - PROB_ITERATE
  - STRUCT_ALIGN
  - UNROLL

  プラグマの詳細は、7.7 節「プラグマ疑似命令」(7-18 ページ) を参照してください。

7.2 TMS320C6000 C++ の特性

TMS320C6000 コンパイラは、ISO/IEC 14882:1998 規格で定義されている 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。
- C のライブラリ機能用の、次の C++ ヘッダは含まれていません。
  - <clocale>
  - <csignal>
  - <cwchar>
  - <cwctype>
- 次の C++ ヘッダは、C++ 標準のライブラリ・ヘッダ・ファイルにのみ含まれています。
  - <new>
  - <typeinfo>
  - <ciso646>
- 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。
- 実行時型情報（RTTI）は、デフォルトで無効にされます。RTTI は、-rtti コンパイラ・オプションを使用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。
- この規格の [tesp.res] および [temp.dep] に説明されているように、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。
- 関数の typedef 型は、member 関数の cv 修飾子を含めることができません。
- 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。

7.3 データ型

表 7–1 は、C6000 コンパイラの各スカラ・データ型のサイズ、表現、および範囲をリストにしたものです。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。詳細は、9.3.6 項「制限値（float.h/cfloat と limits.h/climits）」(9-19 ページ）を参照してください。

表 7–1. TMS320C6000 C/C++ のデータ型

			範囲	
型	**サイズ**	**表現**	**最小**	**最大**
char、signed char	8 ビット	ASCII	-128	127
unsigned char	8 ビット	ASCII	0	255
short	16 ビット	2 の補数	-32768	32767
unsigned short	16 ビット	2 進	0	65535
int、signed int	32 ビット	2 の補数	-2147483648	2147483647
unsigned int	32 ビット	2 進	0	4294967295
long、signed long	40 ビット	2 の補数	-549755813888	549755813887
unsigned long	40 ビット	2 進	0	1099511627775
long long、signed long long	64 ビット	2 の補数	-9 223 372 036 854 775 808	9223 372 036 854775807
unsigned long long	64 ビット	2 進	0	18446744073709551615
enum	32 ビット	2 の補数	-2147483648	2147483647
float	32 ビット	IEEE 32 ビット	1.175494e-38†	3.40282346e+38
double	64 ビット	IEEE 64 ビット	2.22507385e-308†	1.79769313e+308
long double	64 ビット	IEEE 64 ビット	2.22507385e-308†	1.79769313e+308
ポインタ、参照、データ・メンバを指すポインタ	32 ビット	2 進	0	0xFFFFFFFF

† 数値は最小精度です。

7.4 キーワード

C6000 C/C++ コンパイラは、標準の const、register、restrict、および volatile キーワードをサポートしています。さらに、C6000 C/C++ コンパイラは C/C++ 言語を拡張して cregister、interrupt、near、および far キーワードをサポートしています。

7.4.1 const キーワード

TMS320C6000 C/C++ コンパイラは、ISO 規格キーワードの *const* をサポートしています。このキーワードを使用すると、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最適化し、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、それらの値を変更しないようにすることができます。

オブジェクトを far const として定義した場合、その .const セクションはそのオブジェクトに記憶域を割り当てます。const のデータ記憶域割り当て規則には、次の 2 つの例外があります。

- オブジェクトの定義で volatile キーワードも指定した場合（たとえば volatile const int x など）、volatile キーワードは RAM に割り当てられるものと見なされます（プログラムは const volatile オブジェクトを変更しなくても、プログラムの外部にある何かが変更する可能性があります）。

- オブジェクトに（スタックで割り当てられた）自動記憶域がある場合

どちらの場合も、オブジェクトの記憶域は const キーワードを使用しなかった場合と同じものになります。

const キーワードを定義の中に置く位置が重要です。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;
```

const キーワードを使用すると、大きな定数テーブルを定義し、それらのテーブルをシステム ROM 内に割り当てることができます。たとえば、ROM テーブルを割り当てるには次のような定義を使用できます。

```
far const int digits[] = {0,1,2,3,4,5,6,7,8,9};
```

7.4.2 cregister キーワード

C6000 コンパイラは、C/C++ 言語を拡張して cregister キーワードを追加しています。これにより、制御レジスタの高水準言語からのアクセスを可能にしています。

オブジェクトに cregister キーワードを使用する場合、コンパイラは、そのオブジェクトの名前を C6000 の標準制御レジスタのリストと比較します（表 7–2 を参照）。名前が一致する場合、コンパイラはその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 7–2. 有効な制御レジスタ

レジスタ	説明
AMR	アドレッシング・モード・レジスタ
CSR	コントロール・ステータス・レジスタ
FADCR	（C6700 のみ）浮動小数点加算コンフィギュレーション・レジスタ
FAUCR	（C6700 のみ）浮動小数点補助コンフィギュレーション・レジスタ
FMCR	（C6700 のみ）浮動小数点乗算コンフィギュレーション・レジスタ
GFPGFR	（C6400 のみ）ガロア体多項式生成関数レジスタ
ICR	割り込みクリア・レジスタ
IER	割り込みイネーブル・レジスタ
IFR	割り込みフラグ・レジスタ
IRP	割り込みリターン・ポインタ
ISR	割り込みセット・レジスタ
ISTP	割り込みサービス・テーブル・ポインタ
NRP	ノンマスカブル割り込みリターン・ポインタ

cregister キーワードは、ファイル・スコープ内でしか使用できません。cregister キーワードは、関数の境界内の宣言では使用できません。整数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、または構造体や共用体オブジェクトでは使用できません。

cregister キーワードは、オブジェクトが volatile であることを暗黙指定しません。参照される制御レジスタが volatile である（つまり、何らかの外部制御によって変更できる）場合、オブジェクトは volatile キーワードも同時に宣言しなければなりません。

表 7–2 内の制御レジスタを使用する場合には、各レジスタを次のように宣言しなければなりません。c6x.h ヘッダ・ファイルは、次のようにすべての制御レジスタを定義します。

extern cregister volatile unsigned int *register*;

レジスタをいったん宣言すると、レジスタ名を直接使用できます。IFR は読み取り専用であることに注意してください。TMS320C6000 CPU and Instruction Set Reference Guide を参照してください。

制御レジスタを宣言し使用している例については、例 7–1 を参照してください。

例 7–1. 制御レジスタの定義と使用

```
extern cregister volatile unsigned int AMR;
extern cregister volatile unsigned int CSR;
extern cregister volatile unsigned int IFR;
extern cregister volatile unsigned int ISR;
extern cregister volatile unsigned int ICR;
extern cregister volatile unsigned int IER;
extern cregister volatile unsigned int FADCR;
extern cregister volatile unsigned int FAUCR;
extern cregister volatile unsigned int FMCR;
main()
{
   printf("AMR = %x\n", AMR);
}
```

7.4.3 interrupt キーワード

C6000 コンパイラは、C/C++ 言語を拡張して interrupt キーワードを追加します。これは、関数を割り込み関数として扱うことを指定します。

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出口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出口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だけ 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 は C/C++ のエントリ・ポイントです。この名前は、システム・リセットの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。

（-ps コンパイラ・オプションを使用して）厳密 ISO モードのコードを作成する場合は、代替キーワード _ _interrupt を使用してください。

7.4.4 near キーワードと far キーワード

C6000 C/C++ コンパイラは、C/C++ 言語を拡張し、near および far キーワードを追加します。これらは、グローバル変数と静的変数にアクセスする方法、および関数を呼び出す方法を指定します。

構文上、near および far キーワードは記憶クラス修飾子として扱われます。これらのキーワードは、記憶クラス指定子および型の前、後、または間に指定できます。near と far の例外として、2 つの記憶クラス修飾子を 1 つの宣言内で一緒に使用することはできません。次の例は、他の記憶クラス修飾子を使用した near と far の正しい組み合わせです。

```
far static int x;
static near int x;
static int far x;
far int foo();
static far int foo();
```

7.4.4.1 near データ・オブジェクトと far データ・オブジェクト

グローバルおよび静的データ・オブジェクトにアクセスする方法には、次の 2 通りがあります。

near キーワード コンパイラは、データ・ページ・ポインタと相対させてデータ項目にアクセスできるものと想定します。たとえば次のとおりです。

```
LDW   *+dp(_address),a0
```

far キーワード コンパイラは dp を介してデータ項目にアクセスできません。これが必要なのは、プログラム・データの合計量が、DP から許容されるオフセット（32K）より大きい場合です。たとえば次のとおりです。

```
MVKL  _address,a1
MVKH  _address,a1
LDW   *a1,a0
```

一度変数を far に定義すると、他の C ファイルまたはヘッダ内のこの変数に対するすべての外部参照にも、far キーワードが含まれていなければなりません。これは、near キーワードにも当てはまります。しかし、far キーワードがどこにも使用されない場合は、コンパイラまたはリンカ・エラーが表示されます。near キーワードをどこにも使用しなくても、データ・アクセス時間は長くなるだけです。

デフォルトでは、コンパイラはスモールメモリ・モデル・コードを生成します。つまり、実際に far に宣言されている場合を除き、どのデータ・オブジェクトも near に宣言された場合と同じように処理されます。オブジェクトが near に宣言される場合、データ・ページ・ポインタ（DP、B14 です）からの相対的なオフセット・アドレッシングを使用してロードされます。DP は .bss セクションの先頭を指します。

DATA_SECTION プラグマを使用すると、オブジェクトは far 変数として指定され、これを変更することはできません。このオブジェクトを別のファイルで参照する場合、別のソース・ファイルでこのオブジェクトを宣言するときに *extern far* を使用する必要があります。変数が .bss セクション内にない可能性があるので、これにより変数へのアクセスが保証されます。詳細については、7.7.4 項「DATA_SECTION プラグマ」（7-22 ページ）を参照してください。

注 : アセンブリ・コード内でグローバル変数を定義する方法
.usect 疑似命令を使用してアセンブリ・コードでグローバル変数を定義するか（変数が .bss セクションに割り当てられていない場合）、または #pragma DATA_SECTION 疑似命令を使用して独立したセクションに変数を割り当てる場合、および C コード内の変数を参照したい場合、変数を extern far として宣言する必要があります。これにより、コンパイラがデータ・ページ・ポインタを使用して変数に不正にアクセスするコードを生成しようとするのを防ぎます。

7.4.4.2 near 関数呼び出しと far 関数呼び出し

関数呼び出しは、次の 2 通りの方法のどちらかで起動できます。

near キーワード コンパイラは、呼び出しの宛先が、呼び出し側から ± 1 M ワード以内にあるものと想定します。ここで、コンパイラは PC 相対分岐命令を使用します。

```
B     _func
```

far キーワード コンパイラは、呼び出しの宛先が ± 1 M ワード以内にないことをユーザから指示されます。

```
MVKL  _func,a1
MVKH  _func,a1
B     a1
```

デフォルトでは、コンパイラはスモールメモリ・モデル・コードを生成します。つまり、実際に far に宣言されている場合を除いて、どの関数呼び出しも near に宣言された場合と同じように処理されます。

7.4.4.3 ランタイム・サポート関数の呼び出し方法の制御（-mr オプション）

-mr*n* オプションは、ランタイム・サポート関数を呼び出す方法を次のとおり制御します。

-mr0 ランタイム・サポート・データと呼び出しは near になります。

-mr1 ランタイム・サポート・データと呼び出しは far になります。

デフォルトでは、ランタイム・サポート関数は、ユーザ自身がコード化する通常の関数と同じ規則を使用して呼び出されます。-ml オプションを使用してラージメモリ・モデルのいずれかを有効にしない場合には、これらの呼び出しは near になります。-mr0 オプションを指定すると、-ml オプションの設定にかかわらずランタイム・サポート関数の呼び出しは near になります。-mr0 オプションは特別な状況に使用するものであり、通常は必要ありません。-mr1 オプションを指定すると、-ml オプションの設定にかかわらずランタイム・サポート関数の呼び出しは far になります。

-mr オプションは、ランタイム・サポート関数を呼び出す方法だけを処理します。far 方式を使用して関数を呼び出すということは、その関数がオフチップ・メモリ内になければならないという意味ではありません。単に、それらの関数を、呼び出された位置より離れた場所に置くことができることを意味しているだけです。

デフォルトでは、すべてのランタイム・サポート・データは far として定義されます。

7.4.4.4 ラージ・モデル・オプション（-ml）

ラージ・モデル・コマンド行オプションは、near と far のデフォルトの前提事項を変更します。near および far 修飾子は、常にデフォルトを上書きします。

-ml*n* オプションは、ラージ・メモリ・モデル・コードを 4 つのレベル（-ml0、-ml1、-ml2、および -ml3）で生成します。

-ml/-ml0 集合データ（構造体／配列）は、far にデフォルト解釈されます。

-ml1 すべての呼び出しは、far にデフォルト解釈されます。

-ml2 すべての集合データと呼び出しは、far にデフォルト解釈されます。

-ml3 すべての呼び出しとすべてのデータは、far にデフォルト解釈されます。

レベルを指定しないと、すべてのデータと関数は near にデフォルト解釈されます。near データはデータ・ページ・ポインタを使用してアクセスすると効率が上がり、near 呼び出しは PC 相対分岐を使用して実行すると効率が上がります。

static および extern データが多すぎて .bss セクションの先頭から 15 ビット分スケールされたオフセット内に収まらない場合、または呼び出された関数が呼び出し側から ± 1 M ワード以上離れている場合、これらのオプションを使用してください。これらの状態が発生すると、リンカがエラー・メッセージを発行します。

オブジェクトが far に宣言されると、そのアドレスはレジスタにロードされ、コンパイラはそのレジスタの間接ロードを行います。-ml*n* オプションの詳細は、2-16 ページを参照してください。

ラージ・メモリ・モデルとスモール・メモリ・モデルとの相違点の詳細は、8.1.5 項「メモリ・モデル」（8-6 ページ）を参照してください。

7.4.5 restrict キーワード

コンパイラがメモリの依存関係を判別するのに役立つように、ポインタ、参照、または配列を restrict キーワードで修飾できます。restrict キーワードは、ポインタ、参照、配列に適用できる型修飾子です。restrict キーワードを使用すると、ポインタ宣言のスコープ内でポインタにより指されるオブジェクトにはそのポインタしかアクセスできないことを、プログラマが保証します。この保証に違反すると、プログラムの定義が解除されます。この方法は、コードの特定のセクションをコンパイラが最適化するのに役立ちます。それは、エイリアス指定情報が判別しやすいからです。

例 7–2 では、restrict キーワードは、メモリ内でオーバーラップするオブジェクトを指すポインタ a と b によって関数 func1 が呼び出されないようにコンパイラに通知します。これにより、a と b を介するアクセスが競合しないことを保証します。つまり、あるポインタを介する書き込みが他のポインタからの読み取りに影響を与えることができないことを意味します。restrict キーワードの正確な規則は、ISO C 規格の 1999 バージョンに記載されています。

例 7–2. ポインタと restrict 型修飾子との使用

```
void func1(int * restrict a, int * restrict b)
{
  /* func1's code here */
}
```

例 7–3 は、配列を関数に渡す際に restrict キーワードを使用する方法を示しています。ここでは配列 c と d はオーバーラップせず、しかも c と d は同じ配列を指しません。

例 7–3. 配列と restrict 型修飾子との使用

```
void func2(int c[restrict], int d[restrict])
{
  int i;

  for(i = 0; i < 64; i++)
  {
    c[i] += d[i];
    d[i] += 1;
  }
}
```

7.4.6 volatile キーワード

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。volatile キーワードを使用して修飾された変数は、（レジスタと異なり）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、ループ内で位置が 0xFF として読み込まれるのを待ちます。

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

この例で、*ctrl はループ不変式です。そのため、ループは最適化により削除され、単独のメモリ読み取りになります。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;
```

ここでは *ctrl ポインタは、割り込みフラグなどのハードウェア位置を参照することを目的としています。

7.5 レジスタ変数およびパラメータ

TMS320C6000 C/C++ コンパイラは、-o オプションを使用するかどうかに応じてレジスタ変数（register キーワードにより定義された変数）の処理方法を変更します。

❑ 最適化を実行するコンパイル

コンパイラは register 定義を無視し、レジスタを最も効率的に使用できる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。

❑ 最適化を実行しないコンパイル

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。コンパイラは、レジスタ変数の割り当てに使用するのと同じレジスタのセットを、一時的な式の結果を割り当てるのに使用します。

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定義できます。それ以外の型（たとえば配列）のオブジェクトにレジスタを指定しても、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。コンパイラは、レジスタ型のパラメータをスタックではなくレジスタに複写します。このため、関数内でのパラメータへのアクセスが高速になります。

詳細は、8.3 節「レジスタ規則」（8-17 ページ）を参照してください。

7.6 asm 文

TMS320C6000 C/C++ コンパイラは、C6000 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++ 言語の拡張機能である *asm* 文によるものです。asm 文は、C/C++ では不可能なハードウェア機能へのアクセスを可能にします。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 .byte 疑似命令を挿入できます。

```
asm("STR: .byte \"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、この行の先頭にはラベル、空白、タブ、あるいはコメント（* または ;）ができます。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、TMS320C6000 Assembly Language Tools User's Guide 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。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便利な機能です。

注 : asm 文で 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、挿入したコード内や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す疑似命令も、問題となる場合があります。

asm 文と一緒に最適化を使用する場合は特に注意してください。コンパイラは asm 文を除去することはありませんが、asm 文の近くのコードの順序を大幅に再配置し、好ましくない結果を引き起こす可能性があります。

7.7 プラグマ疑似命令

プラグマ疑似命令はコンパイラに対して、特定の関数、オブジェクト、またはコード・セクションの処理方法を指示します。C6000 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_ALIGN
- ❏ DATA_MEM_BANK
- ❏ DATA_SECTION
- ❏ FUNC_CANNOT_INLINE
- ❏ FUNC_EXT_CALLED
- ❏ FUNC_INTERRUPT_THRESHOLD
- ❏ FUNC_IS_PURE
- ❏ FUNC_IS_SYSTEM
- ❏ FUNC_NEVER_RETURNS
- ❏ FUNC_NO_GLOBAL_ASG
- ❏ FUNC_NO_IND_ASG
- ❏ INTERRUPT
- ❏ MUST_ITERATE
- ❏ NMI_INTERRUPT
- ❏ PROB_ITERATE
- ❏ STRUCT_ALIGN
- ❏ UNROLL

上記プラグマの大部分が関数に適用されます。DATA_MEM_BANK を除き、引数 *func* および *symbol* を関数本体の内部で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、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。

関数またはシンボルに適用されるプラグマの場合、プラグマの構文は C と C++ で異なります。C では、最初の引数として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。

7.7.1 CODE_SECTION プラグマ

CODE_SECTION プラグマは、*section name* という名前のセクションに *symbol* 用のスペースを割り当てます。

このプラグマの C での構文は、次のとおりです。

#pragma CODE_SECTION (*symbol*, **"***section name***");**

このプラグマの C++ での構文は、次のとおりです。

#pragma CODE_SECTION ("*section name***");**

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。

例 7–4 は、CODE_SECTION プラグマの使用例です

例 7–4. CODE_SECTION プラグマの使用方法

(a) C ソース・ファイル

```
#pragma CODE_SECTION(fn, "my_sect")

int fn(int x)
{
    return x;
}
```

(b) 生成されるアセンブリ・コード

```
   .sect  "my_sect"
   .global  _fn

;******************************************************************************
;* FUNCTION NAME:_fn                                                          *
;*                                                                            *
;*   Regs Modified     :SP                                                    *
;*   Regs Used         :A4,B3,SP                                              *
;*   Local Frame Size  :0 Args + 4 Auto + 0 Save = 4 byte                     *
;******************************************************************************
_fn:
;** ---------------------------------------------------------------------------
-*
           RET     .S2     B3                ; |6|
           SUB     .D2     SP,8,SP           ; |4|
           STW     .D2T1   A4,*+SP(4)        ; |4|
           ADD     .S2     8,SP,SP           ; |6|
           NOP             2
           ; BRANCH OCCURS                   ; |6|
```

7.7.2 DATA_ALIGN プラグマ

DATA_ALIGN プラグマは、*symbol* を位置合わせ境界に位置合わせします。位置合わせ境界は、シンボルのデフォルト位置合わせ値または *constant* の値（バイト単位）の最大値です。constant は、2 の累乗でなければなりません。

このプラグマの C での構文は、次のとおりです。

```
#pragma DATA_ALIGN (symbol, constant);
```

このプラグマの C++ での構文は、次のとおりです。

```
#pragma DATA_ALIGN (constant);
```

7.7.3 DATA_MEM_BANK プラグマ

DATA_MEM_BANK プラグマは、シンボルまたは変数を指定された C6000 内部データ・メモリ・バンク境界に位置合わせします。*constant* は、変数を開始する特定のメモリ・バンクを指定します（メモリ・バンクのグラフィック表記については、図 4–1（4-33 ページ）を参照）。*constant* の値は、C6000 デバイスに応じて異なります。

C6200　C6200 デバイスには 4 つのメモリ・バンク（0、1、2、および 3）があり、*constant* は 0 または 2 にすることができます。

C6400　C6400 デバイスには 8 つのメモリ・バンクがあり、*constant* は 0、2、4、または 6 にすることができます。

C6700　C6700 デバイスには 8 つのメモリ・バンクがあり、*constant* は 0、2、4、または 6 にすることができます。

このプログラマの C での構文は、次のとおりです。

```
#pragma DATA_MEM_BANK (symbol, constant);
```

このプラグマの C++ での構文は、次のとおりです。

```
#pragma DATA_MEM_BANK (constant);
```

グローバル変数とローカル変数は、どちらも DATA_MEM_BANK プラグマを使用して位置合わせできます。DATA_MEM_BANK プラグマは、位置合わせされるローカル変数が入っている関数内に指定しなければなりません。*symbol* は、
DATA_SECTION プラグマ内のパラメータとしても使用できます。

最適化が有効の場合、ツールはローカル変数の値を保管するためにスタックを使用するかもしれませんし、使用しないかもしれません。

DATA_MEM_BANK プラグマを使用すると、*symbol* 型のサイズのデータを保持できる任意のデータ・メモリ・バンク上にデータを位置合わせすることができます。これは、手書きのアセンブリ・コード内におけるメモリ・バンクの競合を回避するために、ゼロを埋め込んだり、コード内で埋め込みに対処する等の特定の方法でデータを位置合わせする必要がある場合に便利です。

このプラグマではデータを正しいバンクに位置合わせするのに埋め込みが使用されるので、データ・メモリ内で使用されるスペース量が少し増えます。

C6200 の場合、例 7–5 のコードは配列 x が、4 または c（16 進数）で終わるアドレスから始まり、配列 y が 4 または c で終わるアドレスから始まることを保証します。配列 y の位置合わせは、そのスタックの配置に影響を与えます。配列 z は .z_sect セクションに置かれ、0 または 8 で終わるアドレスから始まります。

例 7–5. DATA_MEM_BANK プラグマの使用方法

```
#pragma DATA_MEM_BANK (x, 2);
short x[100];
#pragma DATA_MEM_BANK (z, 0);
#pragma DATA_SECTION (z, ".z_sect");
short z[100];
void main()
{
    #pragma DATA_MEM_BANK (y, 2);
    short y[100];
    ...
}
```

7.7.4 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。

このプログラマの C での構文は、次のとおりです。

#pragma DATA_SECTION (*symbol*, **"***section name***");**

このプラグマの C++ での構文は、次のとおりです。

#pragma DATA_SECTION ("*section name***");**

DATA_SECTION プラグマは、.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。DATA_SECTION プラグマを使用してグローバル変数を割り当て、C コード内でその変数を参照する場合には、変数を extern far として宣言する必要があります。

例 7–6 は、DATA_SECTION プラグマの使用例です。

例 7–6. 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) C++ ソース・ファイル

```
char bufferA[512];
#pragma DATA_SECTION("my_sect")
char bufferB[512];
```

(c) アセンブリ・ソース・ファイル

```
        .global _bufferA
        .bss    _bufferA,512,4
        .global _bufferB
_bufferB:.usect  "my_sect",512,4
```

7.7.5 FUNC_CANNOT_INLINE プラグマ

FUNC_CANNOT_INLINE プラグマはコンパイラに対して、指定した関数をインライン展開できないことを指示します。このプラグマで指定した関数では、inline キーワードなどを使用して指定したインライン展開がすべて無効になります。自動インライン展開もまた、このプラグマを使用すると無効にされます。2.10 節「インライン関数展開の使用方法」（2-38 ページ）を参照してください。

このプラグマは必ず、保持する関数の宣言や参照より前に置かなければなりません。

このプログラマの C での構文は、次のとおりです。

```
#pragma FUNC_CANNOT_INLINE (func);
```

このプラグマの C++ での構文は、次のとおりです。

```
#pragma FUNC_CANNOT_INLINE;
```

C では、引数 *func* はインライン展開できない関数の名前です。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。

7.7.6 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実施します。このような最適化を実施した場合、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・コードから呼び出される C/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/C++ 関数、またはそれらの C/C++ 関数に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/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、保持する関数の宣言や参照より前に置かなければなりません。

このプログラマの C での構文は、次のとおりです。

```
#pragma FUNC_EXT_CALLED (func);
```

このプラグマの C++ での構文は、次のとおりです。

```
#pragma FUNC_EXT_CALLED;
```

C では、引数 *func* は削除されたくない関数の名前です。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++ プログラム用のシステム・リセット割り込みに予約されている _c_int00 という名前を除き、割り込みの名前（引数 *func*）は命名規則に従っていなくても構いません。

プログラムレベルの最適化を使用する場合、FUNC_EXT_CALLED プラグマは特定のオプションと一緒に使用しなければならない場合があります。3.6.2 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」（3-22 ページ）を参照してください。

7.7.7 FUNC_INTERRUPT_THRESHOLD プラグマ

コンパイラは、関数内のしきい値サイクルの間、ソフトウェア・パイプライン・ループに対する割り込みを禁止にできます。これは 1 つの関数に対する -mi オプションを実現します（2.11 節「割り込み柔軟性オプション（-mi オプション）」（2-43 ページ）を参照）。FUNC_INTERRUPT_THRESHOLD プラグマは、-mi*n* コマンド行オプションを常に上書きします。0 より小さいしきい値は、関数が割り込まれないことを意味します。これは無限大の割り込みしきい値と同じ働きをします。

このプログラマの C での構文は、次のとおりです。

#pragma FUNC_INTERRUPT_THRESHOLD (*func***,** *threshold***);**

このプラグマの C++ での構文は、次のとおりです。

#pragma FUNC_INTERRUPT_THRESHOLD (*threshold***);**

次の例は、異なるしきい値の使用例を示しています。

❏ `#pragma FUNC_INTERRUPT_THRESHOLD (foo, 2000)`

関数 foo( ) は、少なくとも 2,000 サイクルごとに割り込み可能になる必要があります。

❏ `#pragma FUNC_INTERRUPT_THRESHOLD (foo, 1)`

関数 foo( ) は、常に割り込み可能でなければなりません。

❏ `#pragma FUNC_INTERRUPT_THRESHOLD (foo, -1)`

関数 foo( ) は、決して割り込まれません。

7.7.8 FUNC_IS_PURE プラグマ

FUNC_IS_PURE プラグマはコンパイラに対して、指定した関数に副次作用がないことを指定します。これにより、コンパイラは次のことができます。

- その関数の値が必要ない場合、その関数の呼び出しを削除します。
- 重複した関数を削除します。

このプラグマは必ず、その関数の宣言や参照より前に置かなければなりません。

このプログラマの C での構文は、次のとおりです。

```
#pragma FUNC_IS_PURE (func);
```

このプラグマの C++ での構文は、次のとおりです。

```
#pragma FUNC_IS_PURE;
```

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。

7.7.9 FUNC_IS_SYSTEM プラグマ

FUNC_IS_SYSTEM プラグマは、指定した関数の動作が、ISO 規格が定義している同じ名前の関数の動作と同じであることをコンパイラに対して指定します。

このプラグマは必ず、保持する関数の宣言や参照より前に置かなければなりません。

このプログラマの C での構文は、次のとおりです。

```
#pragma FUNC_IS_SYSTEM (func);
```

このプラグマの C++ での構文は、次のとおりです。

```
#pragma FUNC_IS_SYSTEM;
```

C では、引数 *func* は ISO 関数として扱う関数の名前です。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。

7.7.10 FUNC_NEVER_RETURNS プラグマ

FUNC_NEVER_RETURNS プラグマはコンパイラに対して、関数が決して呼び出し側に戻らないことを指定します。

このプラグマは必ず、保持する関数の宣言や参照より前に置かなければなりません。

このプログラマの C での構文は、次のとおりです。

```
#pragma FUNC_NEVER_RETURNS (func);
```

このプラグマの C++ での構文は、次のとおりです。

```
#pragma FUNC_NEVER_RETURNS;
```

C では、引数 *func* は戻らない関数の名前です。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。

7.7.11 FUNC_NO_GLOBAL_ASG プラグマ

FUNC_NO_GLOBAL_ASG プラグマはコンパイラに対して、関数が、名前付きのグローバル変数への代入を行わず asm 文も含んでいないことを指定します。

このプラグマは必ず、保持する関数の宣言や参照より前に置かなければなりません。

このプログラマの C での構文は、次のとおりです。

```
#pragma FUNC_NO_GLOBAL_ASG (func);
```

このプログラマの C での構文は、次のとおりです。

```
#pragma FUNC_NO_GLOBAL_ASG;
```

C では、引数 *func* は代入しない関数の名前です。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。

7.7.12 FUNC_NO_IND_ASG プラグマ

FUNC_NO_IND_ASG プラグマはコンパイラに対して、関数がポインタによる代入を行わず asm 文も含んでいないことを指定します。

このプラグマは必ず、保持する関数の宣言や参照より前に置かなければなりません。

このプログラマの C での構文は、次のとおりです。

```
#pragma FUNC_NO_IND_ASG (func);
```

このプラグマの C++ での構文は、次のとおりです。

```
#pragma FUNC_NO_IND_ASG;
```

C では、引数 *func* は代入しない関数の名前です。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。

7.7.13 INTERRUPT プラグマ

INTERRUPT プラグマを使用すると、C コードで割り込みを直接処理できるようになります。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。

このプログラマの C での構文は、次のとおりです。

```
#pragma INTERRUPT (func);
```

このプラグマの C++ での構文は、次のとおりです。

```
#pragma INTERRUPT;
```

関数のコードは、IRP（割り込み戻りポインタ）を介して戻ります。

C プログラム用のシステム・リセット割り込みに予約されている _c_int00 という名前を除き、割り込みの名前（引数 *func*）は命名規則に従っていなくても構いません。

7.7.14 MUST_ITERATE プラグマ

MUST_ITERATE プラグマはコンパイラに対して、ループの特定のプロパティを指定します。これらのプロパティが常に真であることを、ユーザは保証します。MUST_ITERATE プラグマを使用すると、ループが特定の回数実行されることを保証できます。UNROLL プラグマをループに適用するときはいつでも、MUST_ITERATE を同じループに適用する必要があります。MUST_ITERATE プラグマの 3 番目の引数 multiple は最も重要であり、常に指定しなければなりません。

さらに、MUST_ITERATE プラグマはできるだけ高い頻度で他のすべてのループにも適用すべきです。これは、このプラグマを通じて提 供される情報（特に最小反復回数）が、最善のループとループ変換（すなわ ち、ソフトウェア・パイプラインとネストされたループ変換）をコンパイラ が選択する際に役立つからです。また、コンパイラがコード・サイズを縮小するのにも役立つからです。

MUST_ITERATE プラグマと、このプラグマが適用される for、while、または do-while ループとの間に、文を入れることはできません。しかし、UNROLL や PROB_ITERATE などの他のプラグマは MUST_ITERATE プラグマとループとの間に入れることができます。

7.7.14.1 MUST_ITERATE プラグマの構文

このプラグマの C および C++ での構文は、次のとおりです。

#pragma MUST_ITERATE (*min, max, multiple***);**

引数 *min* および *max* は、プログラマが保証する最小と最大のトリップ・カウントです。トリップ・カウントはループが繰り返す回数です。ループのトリップ・カウントは、*multiple*（倍数）によって均等に分割できなければなりません。引数はすべてオプションです。たとえばトリップ・カウントが 5 以上である場合は、引数リストを次のように指定できます。

```
#pragma MUST_ITERATE(5);
```

しかし、トリップ・カウントが 5 の倍数（ゼロ以外）である場合、このプラグマは次のようになります。

```
#pragma MUST_ITERATE(5, , 5); /* Note the blank field for max */
```

場合によっては、コンパイラが展開を実行するために、ユーザが min および multiple を指定する必要があります。これは、ループが実行する反復回数をコンパイラが簡単に判別できない（つまりループの出口条件が複雑である）場合に特に当てはまります。

MUST_ITERATE プラグマを通じて倍数を指定するときに、トリップ・カウントが倍数で均等に分割できない場合は、プログラムの結果の定義が解除されます。また、トリップ・カウントが最小値より小さいか、指定された最大値より大きい場合も、プログラムの結果の定義が解除されます。

min が指定されない場合は、ゼロが使用されます。max が指定されない場合は、可能な限り最大の数値が使用されます。複数の MUST_ITERATE プラグマが同じループに指定されている場合、最も小さい max と最も大きい min が使われます。

7.7.14.2 MUST_ITERATE を使ってループのコンパイラの知識を拡張する方法

MUST_ITERATE プラグマを使用すると、ループが特定回数実行されることを保証できます。次の例は、ループが正確に 10 回実行されることが保証されることをコンパイラに伝えます。

```
#pragma MUST_ITERATE(10,10);
for(i = 0; i < trip_count; i++)  { ...
```

この例では、コンパイラはプラグマを使用しない場合でもソフトウェア・パイプライン・ループを生成しようとします。しかし、MUST_ITERATE がこのようなループに指定されない場合、コンパイラはループを迂回するコードを生成し、反復回数が 0 になる可能性に対して処理します。プラグマを指定すると、コンパイラはループが少なくとも 1 回反復し、ループ迂回コードを削除できることが分かります。

MUST_ITERATE はトリップ・カウントの係数とトリップ・カウントの範囲を指定するのにも使用できます。たとえば次のとおりです。

```
#pragma MUST_ITERATE(8, 48, 8);
for(i = 0; i < trip_count; i++)  { ...
```

次の例は、ループが 8 回～ 48 回実行され、trip_count 変数が 8 の倍数（8、16、24、32、40、48）であることをコンパイラに伝えます。multiple 引数により、コンパイラはループを展開できます。

また、複雑な境界にループする MUST_ITERATE を使うことも考慮する必要があります。次の例では、

```
for(i2 = ipos[2]; i2 < 40; i2 += 5)  { ...
```

コンパイラは、実行時に行われる反復の正確な回数を決定するために、除算関数呼び出しを生成する必要があります。コンパイラはこの動作を行いません。この場合、MUST_ITERATE を使用して、ループを常に 8 回実行するように指定すると、コンパイラはソフトウェア・パイプライン・ループを生成しようとします。

```
#pragma MUST_ITERATE(8, 8);
for(i2 = ipos[2]; i2 < 40; i2 += 5)  { ...
```

7.7.15 NMI_INTERRUPT プラグマ

NMI_INTERRUPT プラグマを使用すると、C コードでノンマスカブル割り込みを直接処理できるようになります。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。

このプログラマの C での構文は、次のとおりです。

#pragma NMI_INTERRUPT (*func*);

このプラグマの C++ での構文は、次のとおりです。

#pragma NMI_INTERRUPT;

関数用に生成されるコードは、interrupt キーワードまたは INTERRUPT プラグマで宣言された関数の場合の IRP と異なり、NRP により戻されます。

C プログラム用のシステム・リセット割り込みに予約されている _c_int00 という名前を除き、割り込みの名前（関数）は命名規則に従っていなくても構いません。

7.7.16 PROB_ITERATE プラグマ

PROB_ITERATE プラグマはコンパイラに対して、ループの特定のプロパティを指定します。一般的な場合にこれらのプロパティが当てはまることを、ユーザは表明します。PROB_ITERATE プラグマは、最善のループとループ変換（つまりソフトウェア・パイプラインとネストされたループ変換）をコンパイラが選択する際に役立ちます。PROB_ITERATE が有効なのは、MUST_ITERATE プラグマが使用されない場合、または PROB_ITERATE パラメータが MUST_ITERATE パラメータより制約が多い場合だけです。

PROB_ITERATE プラグマと、このプラグマが適用される for、while、または do-while ループとの間に、文を入れることはできません。しかし、UNROLL や MUST_ITERATE などの他のプラグマは、PROB_ITERATE プラグマとループとの間に入れることができます。

このプラグマの C および C++ での構文は、次のとおりです。

#pragma PROB_ITERATE (*min, max*);

この場合、min と max は一般的な場合におけるループの最小トリップ・カウントと最大トリップ・カウントです。トリップ・カウントはループが繰り返す回数です。どちらの引数もオプションです。

たとえば、PROB_ITERATE が、大部分の場合 8 回繰り返し実行される（ただし、場合によっては 8 回より多く、または少なく実行される可能性がある）ループに適用できるものとします。

```
#pragma PROB_ITERATE(8, 8);
```

最小予想トリップ・カウントだけ（たとえば 5）が分かっている場合、このプラグマは次のようになります。

```
#pragma PROB_ITERATE(5);
```

最大予想トリップ・カウントだけ（たとえば 10）が分かっている場合、このプラグマは次のようになります。

```
#pragma PROB_ITERATE(, 10);  /* Note the blank field for min */
```

7.7.17 STRUCT_ALIGN プラグマ

STRUCT_ALIGN プラグマは DATA_ALIGN とほぼ同じですが、STRUCT_ALIGN は構造体、共用体型、型定義に適用でき、その型から作成される任意のシンボルにより継承されます。STRUCT_ALIGN プラグマは、C でのみサポートされています。

このプラグマの構文は、次のとおりです。

#pragma STRUCT_ALIGN (*type***,** *constant expression***);**

このプラグマは、名前が指定された型または名前が指定された型定義の基本型の位置合わせが、少なくとも式の位置合わせと等しいことを保証します（この位置合わせは、コンパイラの必要に応じて、これより大きくなる場合があります。）位置合わせは、2 の累乗でなければなりません。*type* は、型または型定義の名前でなければなりません。型である場合は、構造体タグか共用体タグのどちらかでなければなりません。型定義である場合は、その基本型が構造体タグか共用体タグのどちらかでなければなりません。

ISO C は型定義が単なる型のエイリアス（つまり構造体）であることを宣言するので、このプラグマは構造体、構造体の型定義、またはそれらから派生される任意の型定義に適用でき、基本型のすべてのエイリアスに影響を与えます。

次の例は、ページ境界上ですべての st_tag 構造体変数の位置合わせをします。

```
typedef struct st_tag
{
  int   a;
  short b;
} st_typedef;

#pragma STRUCT_ALIGN (st_tag, 128);
```

STRUCT_ALIGN を基本型（int、short、float）または変数と一緒に使用すると、エラーが発生します。

7.7.18 UNROLL プラグマ

UNROLL プラグマはコンパイラに対して、ループが展開される回数を指定します。UNROLL プラグマは、コンパイラが C6400 ファミリーで SIMD 命令を利用するのに役立ちます。また、非展開ループでソフトウェア・パイプライン・リソースの使用効率を改善することが必要な場合にも便利です。

プラグマが指定したループの展開を実行するには、オプティマイザを起動する必要があります（-o1、-o2、または -o3 の使用）。コンパイラには、このプラグマを無視するオプションがあります。

UNROLL プラグマと、このプラグマが適用される for、while、または do-while ループとの間に、文を入れることはできません。しかし、MUST_ITERATE や PROB_ITERATE などの他のプラグマは、UNROLL プラグマとループとの間に入れることができます。

C および C++ のどちらの場合も、このプラグマの構文は次のとおりです。

#pragma UNROLL (*n*);

可能であれば、元のループのコピーが *n* 個あるように、コンパイラがループを展開します。コンパイラが展開するのは、係数 *n* による展開が安全であると判断できる場合だけです。ループが展開される確率を上げるために、コンパイラは次のような特定のプロパティを認識している必要があります。

- ループは *n* の倍数回繰り返します。この情報は、MUST_ITERATE プラグマ内の multiple 引数を介してコンパイラに指定できます。
- ループの最小反復回数
- ループの最大反復回数

場合によってコンパイラは、コードを解析して自身でこの情報を取得できます。しかし、コンパイラが前提条件を控え目にし過ぎるので、展開時に必要以上のコードを生成する場合があります。これが、展開しない原因にもなることがあります。

さらに、ループが終了する時期を決定するメカニズムが複雑である場合、コンパイラはループのこれらの特性を判別できない可能性があります。こうした場合には、MUST_ITERATE プラグマを使用してループの特性をコンパイラに指示しなければなりません。

#pragma UNROLL(1); を指定すると、ループが展開されません。自動ループ展開もこの場合は実行されません。

同じループに対して複数の UNROLL プラグマが指定される場合には、どの UNROLL プラグマが使用されるかは未定義です。

7.8 リンク名の生成

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スコープにより異なります。オブジェクトや C 関数の場合、下線（_）が識別子名の前に付きます。C++ 関数の前にも下線が付きますが、関数名はさらに修正されます。

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。マングリングが行われるのは C++ コードだけです。使用されるマングリング・アルゴリズムは、The 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。

たとえば、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__Fparmcodes
```

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
```

その結果のアセンブリ・コードは次のとおりです。

```
___foo__Fi
```

foo のリンク名は ___foo__Fi です。これは、foo が int 型の 1 つの引数をとる関数であることを示しています。確認とデバッグに役立つように、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・ユーティリティが備えられています。詳細は、第 11 章「C++ ネーム・デマングラ」を参照してください。

7.9 静的変数とグローバル変数の初期化方法

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）変数と静的変数は、プログラムが実行し始める前に 0 に初期化されなければならないと定めています。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。

7.9.1 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。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:fill = 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（プログラムではなく）のサイズが大幅に増えるという望ましくない効果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。上記の方法では .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、TMS320C6000 Assembly Language Tools User's Guide の「Linker description」を参照してください。

7.9.2 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ない *const* 型の静的およびグローバル変数は、0 に事前初期化されないのでその他の静的およびグローバル変数と似ています（7.9 節「静的変数とグローバル変数の初期化方法」(7-34 ページ) で説明する理由と同じです)。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、リンカは ROM の .const セクションを配置するためにも使えます。

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、以下の C コードを参照してください。

```
#pragma DATA_SECTION (var, “.mysect”);
   const int zero=0;
```

これは、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
```

7.10 ISO C 言語モードの変更

-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 ISO モード
- ❏ K&R C モード
- ❏ 緩和 ISO モード
- ❏ 厳密 ISO モード

デフォルトは、標準 ISO モードです。標準 ISO モードでは、大部分の ISO 違反にエラーが発行されます。しかし、厳密な ISO 違反（厳密な 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、ISO C と矛盾するものであっても有効です。

C++ コードの場合、ISO モードは、サポートされる最新の研究報告書を指定します。K&R C モードは、C++ コードには適用されません。

7.10.1 K&R C との互換性（-pk オプション）

ISO C/C++ 言語は、カーニハンとリッチーの「The C Programming Language」で定義された、事実上の C 標準のスーパーセットです。ISO 対応でない他のコンパイラ用に書かれたプログラムの大半は、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、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。

C6000 ISO C/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）が用意されています。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 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 ISO 規則を緩和するだけのオプションです。

ISO C と K&R C とで特に異なる点を以下に示します。

- 符号なし型をより広範囲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 では結果の型は広い型の符号なしバージョンであり、ISO では結果の型は広い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 mod）、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 */
```

- 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 */
```

- 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 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 */
```

- 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 */
```

  ISO では、この 2 つの定義はオブジェクト 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。

- 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 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 */
```

- 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定義できます。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コ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           /* illegal unless -pk used */
```

7.10.2 厳密 ISO モードと緩和 ISO モードの有効化（-ps および -pr オプション）

厳密 ISO モードでコンパイルする場合は、-ps オプションを使用してください。このモードでは、非 ISO 機能が使用されるとエラー・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 ISO モードで発生する）またはエラー・メッセージ（厳密 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 ISO 違反を無視させる場合には -pr オプションを使用してください。緩和 ISO モードでは、コンパイラは、ISO C 規格の拡張機能が ISO 規格の C と対立する場合であってもそれを受け入れます。

7.10.3 組み込み C++ モードの有効化（-pe オプション）

コンパイラは、組み込み C++ のコンパイルをサポートします。このモードでは、組み込みシステムでサポートするには価値が低すぎるか、コストがかかり過ぎる C++ の一部の機能は除去されます。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
- mutable キーワード
- 多重継承
- 仮想継承

組み込み C++ の標準定義では、ネームスペースおよび宣言の使用がサポートされていません。しかし、C6000 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。

第 8 章

ランタイム環境

本章では、TMS320C6000 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。また、C/C++ コードに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**

8.1	メモリ・モデル	8-2
8.2	オブジェクトの表記	8-8
8.3	レジスタ規則	8-17
8.4	関数の構造と呼び出し規則	8-19
8.5	アセンブリ言語と C および C++ 言語間のインターフェイス	8-23
8.6	割り込み処理	8-46
8.7	ランタイム・サポート算術ルーチン	8-48
8.8	システムの初期化	8-51

8.1 メモリ・モデル

C6000 コンパイラは、コードとデータのサブブロックに分割されている連続した 1 つのブロックとしてメモリを扱います。C プログラムが生成するコードまたはデータの各サブブロックは、それ自身の連続したメモリ空間内に配置されます。コンパイラは、32 ビットの全アドレス空間がターゲット・メモリで使用可能であるものと想定します。

注 : リンカはメモリ・マップを定義する

リンカは（コンパイラではなく）メモリ・マップを定義し、コードと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、使用可能なメモリのタイプ、コードまたはデータ用に使用できないロケーション（ホール）、または入出力あるいは制御用に確保されたロケーションについては何も考慮しません。コンパイラは、リンカが適切なメモリ空間にコードとデータを割り当てることができるように再配置可能なコードを生成します。

たとえば、リンカを使用して、グローバル変数をオンチップ RAM に割り当て、実行可能コードを外部 ROM に割り当てることができます。コードまたはデータの各ブロックを個別にメモリに割り当てることができますが、この方法は一般的ではありません（例外としてメモリ・マップド I/O がありますが、物理メモリ・ロケーションを C/C++ ポインタ型でアクセスできます）。

8.1.1 セクション

コンパイラは、セクションと呼ばれる、コードとデータが入った再配置可能ブロックを生成します。これらのセクションは、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。セクションとセクションの割り振り方法については、TMS320C6000 Assembly Language Tools User's Guide で COFF 情報を参照してください。

C6000 コンパイラは、次のセクションを作成します。

- **初期化されたセクション**には、データおよび実行可能なコードが含まれます。C/C++ コンパイラは、次の初期化されたセクションを作成します。
  - **.cinit セクション** には、変数および定数を初期化するためのテーブルが含まれます。
  - **.const セクション**は、C/C++ 修飾子 *const* により定義された文字列リテラル、浮動小数点定数、およびデータを含みます（その定数が *volatile* としても定義されている場合は除く）。
  - **.switch セクション**には、大きな switch 文用のジャンプ・テーブルが含まれます。
  - **.text セクション**には、すべての実行可能コードが含まれます。

- **初期化されないセクション**は、メモリ（通常 RAM）に空間を確保します。プログラムは、この空間を実行時に変数の作成と保存に使用できます。本コンパイラは、次の 4 つの初期化されないセクションを生成します。
  - **.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空間を確保します。-c リンカ・オプションを指定する場合、プログラムの始動時に、C ブート・ルーチンが .cinit セクション（ROM に入っている場合がある）からデータをコピーし、それを .bss セクションに保存します。コンパイラは、グローバル・シンボル $bss を定義し、.bss セクションの開始アドレスの値を $bss に割り当てます。
  - **.far セクション**は、far と宣言されるグローバル変数と静的変数用の空間を確保します。
  - **.stack セクション**は、システム・スタック用のメモリを割り当てます。このメモリは関数に引数を渡し、ローカル変数を割り当てます。
  - **.sysmem セクション**は、動的メモリの割り当て用の空間を確保します。この確保された空間は malloc、calloc、および realloc 関数により使用されます。C/C++ プログラムでこれらの関数を使用しない場合、コンパイラは .sysmem セクションを作成しません。

注 : プログラム・メモリではコードだけを使用する

.text を除き、初期化されたセクションと初期化されないセクションを内部プログラム・メモリに割り当てることはできません。

アセンブラは、デフォルト・セクション .text、.bss、.data を作成します。しかし、C/C++ コンパイラは .data セクションを使用しません。追加のセクションを作成するようにコンパイラに指示するには、CODE_SECTION および DATA_SECTION プラグマを使用してください（7.7.1 項「CODE_SECTION プラグマ」（7-19 ページ）および 7.7.4 項「DATA_SECTION プラグマ」（7-22 ページ）を参照）。

8.1.2 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。

- ❏ 関数の戻りアドレスを保管します。
- ❏ ローカル変数を割り当てます
- ❏ 関数に引数を渡します。
- ❏ 一時的な結果を保存します。

ランタイム・スタックは、上位のアドレスから下位のアドレスへと増大します。コンパイラは、B15 レジスタを使用してこのスタックを管理します。B15 は、スタック上にある次の未使用の位置を指すスタック・ポインタ（SP）です。

リンカはスタック・サイズを設定し、グローバル・シンボル _ _STACK_SIZE を作成し、バイト単位のスタック・サイズをそれに割り当てます。デフォルトのスタック・サイズは 0x400（1024）バイトです。リンカ・コマンドで -stack オプションを使用すると、リンク時にスタック・サイズを変更できます。-stack オプションの詳細は、5.2 節「リンカ・オプション」（5-5 ページ）を参照してください。

システムの初期化時に、SP は、.stack セクションの最後の直前にある 8 バイトの位置合わせされたアドレス（最高数値のアドレス）に設定されます。スタックの位置は .stack セクションが割り当てられる位置に応じて異なるので、スタックの実際のアドレスはリンク時に決定されます。

C/C++ 環境は、関数の入り口で SP（レジスタ B15）を自動的に減らして、その関数の実行に必要なすべての空間を確保します。スタック・ポインタは関数の出口で増分され、関数へ入る前の状態にスタックを戻します。アセンブリ言語ルーチンと C/C++ プログラムとのインターフェイスを取る場合には、必ずスタック・ポインタを関数へ入る前の状態に戻してください。スタックとスタック・ポインタの詳細は、8.4 節「関数の構造と呼び出し規則」（8-19 ページ）を参照してください。

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・オーバーフローを検出する方法がありません。スタック・オーバーフローがシミュレータに障害を引き起こすように、マップされていないメモリ空間の直後のアドレスに .stack セクションの先頭を配置します。これにより、スタック・オーバーフローの問題を簡単に検出できるようになります。スタックが増大できるように十分な空間を確保してください。

8.1.3 動的なメモリ割り当て

動的メモリ割り当ては、C 言語の標準な部分にはありませんが、C6000 コンパイラに付属のランタイムサポート・ライブラリには、実行時に変数にメモリを動的に割り当てることができるいくつかの関数（malloc、calloc、realloc など）が含まれています。

メモリは、.sysmem セクションで定義されたグローバル・プール（ヒープ）から割り当てられます。.sysmem セクションのサイズは、-heap *size* オプションとリンカ・コマンドを使用して設定できます。リンカはグローバル・シンボル _ _SYSMEM_SIZE も作成し、このシンボルにバイト単位で表したヒープ・サイズを割り当てます。デフォルトのサイズは 0x400 バイトです。-heap オプションの詳細は、5.2 節「リンカ・オプション」（5-5 ページ）を参照してください。

動的に割り当てられるオブジェクトは、直接的にはアドレス指定されません（常にポインタを使用してアクセスされます）。また、メモリ・プールは別のセクション（.sysmem）に入っています。したがって動的メモリ・プールのサイズを制限するものは、システム内の使用可能メモリの量だけです。.bss セクション内の空間を節約するには、グローバルまたは静的な配列を定義するのでなくヒープから大きな配列を割り当てます。たとえば、

```
struct big table[100];
```

という定義の代わりにポインタを使用し、次のような malloc 関数を呼び出すことができます。

```
struct big *table
table = (struct big *)malloc(100*sizeof (struct big));
```

8.1.4 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、.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、これらのテーブルのデータ（ROM 内の）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。詳細は、8.8 節「システムの初期化」（8-51 ページ）を参照してください。

8.1.5 メモリ・モデル

コンパイラは、.bss セクションがメモリに割り当てられる方法に影響を与える、2 つのメモリ・モデルをサポートします。どちらのモデルも、.text セクションまたは .cinit セクションのサイズに制限を与えません。

- **スモール・メモリ・モデル**（デフォルト）では、.bss セクション全体が 32K バイト（32,768 バイト）のメモリ内に収まらなければなりません。つまり、プログラム内のすべての静的データとグローバル・データ用の空間の合計が 32K バイト未満でなければなりません。コンパイラは、.bss セクションの先頭を指すように、データ・ページ・ポインタ（DP。これは B14 です）を実行時の初期化中に設定します。その後コンパイラは、DP を変更することなく直接アドレッシングを使用して、.bss 内のすべてのオブジェクト（グローバル変数と静的変数、および定数テーブル）にアクセスできるようになります。

- **ラージ・メモリ・モデル**は .bss セクションのサイズを制限しません。静的データとグローバル・データに使用できる空間には制限がありません。ただし、コンパイラが .bss に保存されていないグローバル・オブジェクトまたは静的オブジェクトにアクセスする場合は、最初にオブジェクトのアドレスをレジスタにロードしてからグローバル・データ項目にアクセスする必要があります。このため 2 つの追加アセンブリ命令が生成されます。

  たとえば次のコンパイラ生成アセンブリ言語は、MVKL および MVKH 命令を使用してグローバル変数 _x を A0 レジスタに移動してから、A0 を指すポインタを使用して B0 レジスタをロードします。

  ```
  MVKL_x, A0
  MVKH_x, A0
  LDW*A0, B0
  ```

  ラージ・メモリ・モデルを使用する場合は、-ml*n* オプションを指定してコンパイラを起動してください。-ml*n* オプションの詳細は、7.4.4.4 項「ラージ・モデル・オプション（-ml）」（7-13 ページ）を参照してください。

グローバル変数と静的変数の記憶域割り当ての詳細は、7.4.4 項「near キーワードと far キーワード」（7-11 ページ）を参照してください。

8.1.6 位置に依存しないデータ

near グローバル・データと静的データは .bss セクションに保存されます。プログラムのすべての near データは、32K バイトのメモリ内に収まらなければなりません。この制限は、near データへのアクセスに使用されるアドレッシング・モードによるものです。near データは、DP（B14）データ・ページ・ポインタからの 15 ビット符号なしオフセットに制限されます。

一部のアプリケーションでは、near データの別々のインスタンスをもつ複数のデータ・ページが必要な場合があります。たとえば、マルチチャネル・アプリケーションには、同じプログラムの複数のコピーが異なるデータ・ページで動作している場合があります。この機能は C6000 コンパイラのメモリ・モデルによりサポートされ、位置に依存しないデータと呼ばれます。

位置に依存しないデータとは、すべての near データのアクセスがデータ・ページ（DP）ポインタと関連し、実行時に DP が変更できることを意味します。コンパイラが位置に依存しないデータを実現する領域には、次の 3 つがあります。

1） near 直接メモリ・アクセス

```
STW  B4,*DP(_a)

.global _a
.bss    _a,4,4
```

すべての near 直接アクセスは DP と相対的なものになります。

2） near 間接メモリ・アクセス

```
MVK (_a - $bss),A0
ADD DP,A0,A0
```

この式（_a - $bss）は、.bss セクションの先頭からのシンボル _a のオフセットを計算します。コンパイラは、生成されたアセンブリ・コード内でグローバル $bss を定義します。$bss の値は .bss セクションの開始アドレスです。

3） 初期化された near ポインタ

初期化された near ポインタ値用の .cinit レコードは、.bss セクションの先頭からのオフセットとして保存されます。グローバル変数の自動初期化時に、データ・ページ・ポインタがこれらのオフセットに加算されます（8.8.3 項「初期化テーブル」（8-53 ページ）を参照）。

8.2 オブジェクトの表記

この節では、各種データ・オブジェクトのサイズ指定、位置合わせ、アクセスの方法を説明します。

8.2.1 データ型の記憶域

表 8–1 は、各種データ型のレジスタとメモリの記憶域を示しています。

表 8–1. レジスタとメモリ内のデータ表記

データ型	レジスタ記憶域	メモリ記憶域
char	レジスタのビット 0 ～ 7	8 ビット境界に位置合わせされた 8 ビット
unsigned char	レジスタのビット 0 ～ 7	8 ビット境界に位置合わせされた 8 ビット
short	レジスタのビット 0 ～ 15	16 ビット境界に位置合わせされた 16 ビット
unsigned short	レジスタのビット 0 ～ 15	16 ビット境界に位置合わせされた 16 ビット
int	レジスタ全体	32 ビット境界に位置合わせされた 32 ビット
unsigned int	レジスタ全体	32 ビット境界に位置合わせされた 32 ビット
enum	レジスタ全体	32 ビット境界に位置合わせされた 32 ビット
float	レジスタ全体	32 ビット境界に位置合わせされた 32 ビット
long	偶数／奇数レジスタ・ペアのビット 0 ～ 39	64 ビット境界に位置合わせされた 64 ビット
unsigned long	偶数／奇数レジスタ・ペアのビット 0 ～ 39	64 ビット境界に位置合わせされた 64 ビット
long long	偶数／奇数レジスタ・ペア	64 ビット境界に位置合わせされた 64 ビット
unsigned long long	偶数／奇数レジスタ・ペア	64 ビット境界に位置合わせされた 64 ビット
double	偶数／奇数レジスタ・ペア	64 ビット境界に位置合わせされた 64 ビット
long double	偶数／奇数レジスタ・ペア	64 ビット境界に位置合わせされた 64 ビット
struct	メンバは、個々の型で求められるとおりに保存されます。	最大のメンバ型の境界に位置合わせされた 8 ビットの倍数。メンバは、個々の型で求められるとおりに保存され、位置合わせされます。
配列	メンバは、個々の型で求められるとおりに保存されます。	メンバは、個々の型で求められるとおりに保存されます。C64x の場合 64 ビット境界に位置合わせされ、32 ビット以下のすべての型は 32 ビット境界に位置合わせされ、C62x および C67x の 32 ビットを超えるすべての型は 64 ビット境界に位置合わせされます。構造体内のすべての配列は、配列の各要素の型に基づいて位置合わせされます。

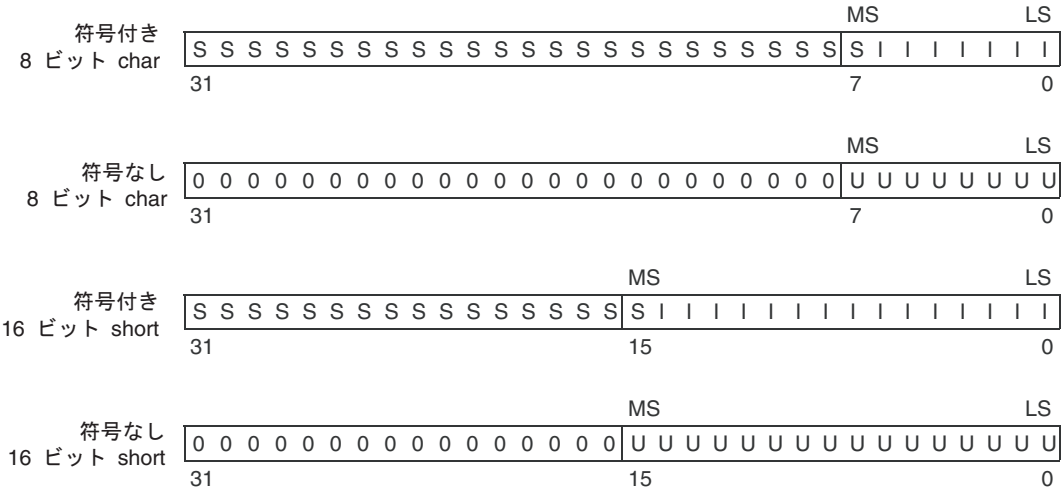
表 8–1. レジスタとメモリ内のデータ表記（続き）

データ型	レジスタ記憶域	メモリ記憶域
データ・メンバへのポインタ	レジスタのビット 0 ～ 31	32 ビット境界に位置合わせされた 32 ビット
メンバ関数へのポインタ	コンポーネントは、個々の型で求められるとおりに保存されます。	32 ビット境界に位置合わせされた 64 ビット

8.2.1.1 char データ型と short データ型（符号付きと符号なし）

char または unsigned char データ型は、1 バイトとしてメモリに保存され、レジスタのビット 0 ～ 7 にロードされ、そこから保存されます（表 8–1 を参照）。short または unsigned short として定義されるオブジェクトは、2 バイトとしてメモリに保存され、レジスタのビット 0 ～ 15 にロードされ、そこから保存されます（表 8–1 を参照）。ビッグエンディアン・モードでは、2 バイト・オブジェクトをレジスタにロードする際には、メモリの最初のバイト（つまり下位アドレス）をレジスタのビット 8 ～ 15 に移動し、メモリの 2 番目のバイトをビット 0 ～ 7 に移動します。リトルエンディアン・モードでは、2 バイト・オブジェクトをレジスタにロードする際には、メモリの最初のバイト（つまり下位アドレス）をレジスタのビット 0 ～ 7 に移動し、メモリの 2 番目のバイトをビット 8 ～ 15 に移動します。

図 8–1. char および short のデータ保存形式

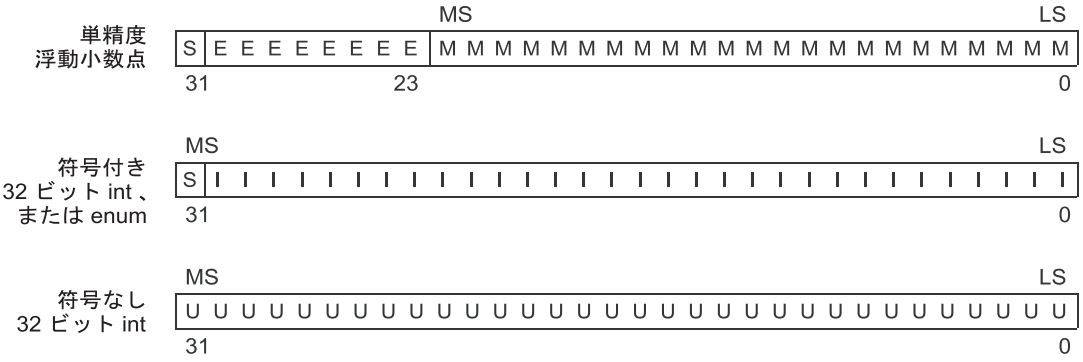


凡例： S = 符号　　MS = 最大重み
I = 符号付き整数　　LS = 最小重み
U = 符号なし

8.2.1.2 enum、float、および int データ型（符号付きと符号なし）

int、unsigned int、enum、および float データ型は、32 ビット・オブジェクトとしてメモリ内に保存されます（表 8–2 を参照）。これらの型のオブジェクトはレジスタのビット 0 ～ 31 にロードされ、そこから保存されます。ビッグエンディアン・モードでは、4 バイト・オブジェクトをレジスタにロードする際には、メモリの最初のバイト（つまり下位アドレス）をレジスタのビット 24 ～ 31 に移動し、メモリの 2 番目のバイトをビット 16 ～ 23 に移動し、3 番目のバイトをビット 8 ～ 15 に移動し、4 番目のバイトをビット 0 ～ 7 に移動します。リトルエンディアン・モードでは、4 バイト・オブジェクトをレジスタにロードする際には、メモリの最初のバイト（つまり下位アドレス）をレジスタのビット 0 ～ 7 に移動し、2 番目のバイトをビット 8 ～ 15 に移動し、3 番目のバイトをビット 16 ～ 23 に移動し、4 番目のバイトをビット 24 ～ 31 に移動します。

図 8–2. 32 ビット・データの保存形式



凡例：

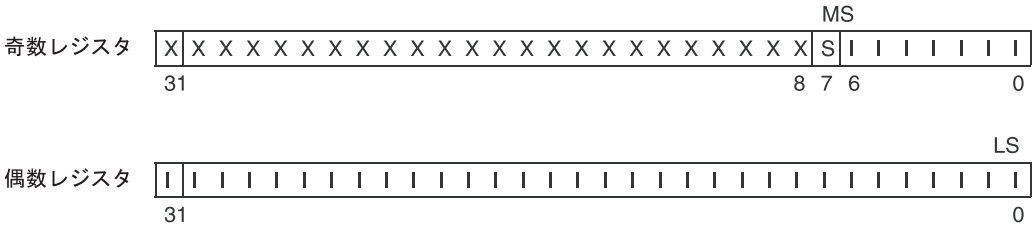
S = 符号　E = 指数　MS = 最大重み
M = 小数部　I = 符号付き　LS = 最小重み
U = 符号なし整数

8.2.1.3 long データ型（符号付きと符号なし）

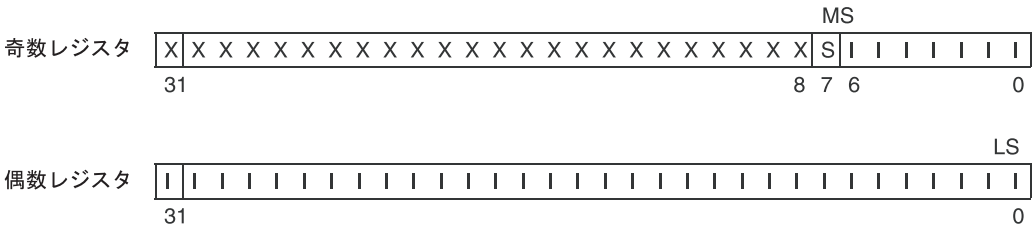
long および unsigned long データ型は、奇数／偶数のペアのレジスタに保存され（図 8–3 を参照）、奇数レジスタ : 偶数レジスタ（たとえば A1:A0）の形式で常にペアで参照されます。リトルエンディアン・モードでは、下位アドレスが偶数レジスタにロードされ、上位アドレスが奇数レジスタにロードされます。データが位置 0 からロードされる場合、0 の位置のバイトが偶数レジスタの最下位バイトです。ビッグエンディアン・モードでは、上位アドレスが偶数レジスタにロードされ、下位アドレスが奇数レジスタにロードされます。データが位置 0 からロードされる場合、0 の位置のバイトは奇数レジスタの最上位バイトですが、これは無視されます。

図 8–3. 40 ビット・データの保存形式

（a）符号付き 40 ビット long



（b）符号なし 40 ビット long



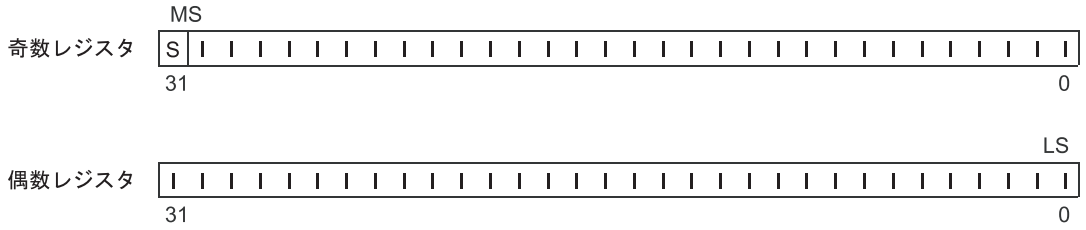
凡例： S = 符号　I = 符号付き整数　MS = 最大重み
U = 符号なし整数　X = 未使用　LS = 最小重み

8.2.1.4 long long データ型（符号付きと符号なし）

long long および unsigned long long データ型は、奇数／偶数のペアのレジスタに保存され（図 8–4 を参照）、奇数レジスタ : 偶数レジスタ（たとえば A1:A0）の形式で常にペアで参照されます。リトルエンディアン・モードでは、下位アドレスが偶数レジスタにロードされ、上位アドレスが奇数レジスタにロードされます。データが位置 0 からロードされる場合、0 の位置のバイトが偶数レジスタの最下位バイトです。ビッグエンディアン・モードでは、上位アドレスが偶数レジスタにロードされ、下位アドレスが奇数レジスタにロードされます。データが位置 0 からロードされる場合、0 の位置のバイトが奇数レジスタの最上位バイトです。

図 8–4. 64 ビット・データの保存形式

（a）符号付き 64 ビット long



（b）符号なし 64 ビット long

凡例： S = 符号　I = 符号付き整数　MS = 最大重み
U = 符号なし整数　X = 未使用　LS = 最小重み

8.2.1.5 double データ型と long double データ型

double および long double データ型は奇数／偶数のペアのレジスタに保存され（図 8–5 を参照）、奇数レジスタ : 偶数レジスタ（たとえば A1:A0）の形式で常にペアで参照されます。奇数メモリ・ワードには、符号ビット、指数、および小数部の最大重み部分が入っています。偶数メモリ・ワードには小数部の最小重み部分が入っています。リトルエンディアン・モードでは、下位アドレスは偶数レジスタにロードされ、上位アドレスは奇数レジスタにロードされます。ビッグエンディアン・モードでは、上位アドレスが偶数レジスタにロードされ、下位アドレスが奇数レジスタにロードされます。リトルエンディアン・モードでは、コードが位置 0 からロードされる場合、0 の位置にあるバイトが偶数レジスタの最下位バイトです。ビッグエンディアン・モードでは、コードが位置 0 からロードされる場合は、0 の位置にあるバイトが奇数レジスタの最上位バイトです。

図 8–5. 倍精度浮動小数点データの保存形式

凡例： S = 符号　　MS = 最大重み
M = 小数部　　LS = 最小重み
E = 指数

8.2.1.6 構造体と配列

ネストされた構造体は、自身がもつ最大の型により必要とされる境界に位置合わせされます。たとえば、ネストされた構造体の最大の型が short 型の場合、ネストされた構造体は 2 バイト境界上で位置合わせされます。ネストされた構造体の最大の型が long、unsigned long、double、または long double の場合、ネストされた構造体は 8 バイト境界上に位置合わせされます。

構造体は常に、最大の要素型のサイズの倍数のメモリを確保します。たとえば、構造体に int、unsigned int、または float が入っている場合は、4 バイトの倍数の記憶域をメモリ内に確保します。構造体のメンバは、個々のオブジェクトである場合と同様に保存されます。

C64x の場合、配列は 8 バイト境界上に位置合わせされ、C62x または C67x の場合、4 バイト（32 ビット以下のすべての要素型の場合）または 8 バイトバイト境界上のいずれかに位置合わせされます。配列の要素は、個々のオブジェクトである場合と同じように保存されます。

8.2.1.7　データ・メンバ型を指すポインタ

データ・メンバ・オブジェクトを指すポインタは、unsigned int（32 ビット）整数型のようにメモリ内に保存されます。その値は、クラス内のデータ・メンバに対するバイト・オフセット + 1 です。ゼロの値は、データ・メンバへの NULL ポインタを表すために予約されています。

8.2.1.8　メンバ関数型を指すポインタ

メンバ関数オブジェクトを指すポインタは、3 つのメンバを備えた構造体として保存されます。

```
struct {
   short int d;
   short int i;
   union {
      void (*f) ();
      int 0;

   }
};
```

パラメータ d は、このポインタ用にクラス・オブジェクトの先頭に追加されるオフセットです。パラメータ i は、1 オフセットされる、仮想関数テーブルへのインデックスです。このインデックスにより、NULL ポインタを表示できます。関数が仮想でない場合、その値は -1 です。パラメータ f は、i がゼロである時に、メンバ関数が仮想でない場合にそのメンバ関数を指すポインタです。このゼロは、クラス・オブジェクト内の仮想関数ポインタに対するオフセットです。

8.2.2 ビット・フィールド

ビット・フィールドは、1 バイト内にパックされる唯一のオブジェクトです。つまり、2 つのビット・フィールドを同じバイトに保存できます。ビット・フィールドのサイズは 1 ～ 32 ビットですが、4 バイト境界にまたがることはありません。

ビッグエンディアン・モードの場合、ビット・フィールドは、定義された順に最大重みビット（MSB）から最小重みビット（LSB）までのレジスタにパックされます。ビット・フィールドは、最大重みバイト（MSbyte）から最小重みバイト（LSbyte）までのメモリにパックされます。リトルエンディアン・モードの場合、ビット・フィールドは定義された順に LSB から MSB までのレジスタにパックされ、LSbyte から MSbyte までのメモリにパックされます（図 8–6 を参照）。

図 8–6 は、次のビット・フィールド定義を使用したビット・フィールドのパッキングを示しています。

```
struct {
     int A:7
     int B:10
     int C:3
     int D:2
     int E:9
}x;
```

A0 はフィールド A の最小重みビットを表し、A1 は次の最小重みビットを表し、以下同様に表しています。また、メモリ内にビット・フィールドを保存するのは、ビット単位の転送ではなくバイト単位の転送により行われます。

図 8–6. ビッグエンディアン形式とリトルエンディアン形式のビット・フィールド・パッキング

凡例： X = 未使用
MS = 最大重み
LS = 最小重み

8.2.3 文字列定数

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″;
  ```

  初期化指定子として使用されると、文字列は、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、8.8 節「システムの初期化」（8-51 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。たとえば次の行は、文字列 abc、および終了 0 バイトを定義します（ラベル SL5 はその文字列を指します）。

  ```
    .sect ″.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、このラベルにより文字列式を参照します。

  文字列は .const セクション（ほとんどの場合 ROM 内の）に格納され、共用されるので、プログラムが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。

  ```
  const char*a = ″abc″
  a[1] = ′x′;   /* Incorrect!*/
  ```

8.3 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とは、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を理解し、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。表 8–2 は、コンパイラによる TMS320C6000 レジスタの使用方法をまとめたものです。

表 8–2 のレジスタは、コンパイラがレジスタ変数と一時的な式の結果に割り当てるために使用できます。コンパイラが必要な型のレジスタを割り振ることができない場合、スピルが行われます。スピルとは、レジスタの内容をメモリに移動して、別の目的用にそのレジスタを解放するプロセスです。

double、long、long long、または long double 型のオブジェクトは奇数と偶数のレジスタ・ペアに割り当てられ、常にレジスタ・ペア（たとえば、A1:A0）として参照されます。奇数レジスタには、符号ビット、指数、および小数部の最大重み部分が入っています。偶数レジスタには、小数部の最小重み部分が入っています。最初の引数が double、long、long long、または long double である場合は、最初の引数を渡すために A4 レジスタが A5 と一緒に使用されます。2 番目のパラメータには B4 と B5 がこれに当てはまり、以下同様に当てはまります。引数を渡すレジスタと戻りレジスタの詳細については、8.4 節「関数の構造と呼び出し規則」を参照してください。

表 8–2. レジスタの用途

レジスタ	関数の保存	特殊な用途	レジスタ	関数の保存	特殊な用途
A0	親	--	B0	親	--
A1	親	--	B1	親	--
A2	親	--	B2	親	--
A3	親	構造体レジスタ（戻された構造体を指すポインタ）	B3	親	戻りレジスタ（戻される先のアドレス）
A4	親	引数 1 または戻り値	B4	親	引数 2
A5	親	引数 1 または戻り値（double、long、および long long の場合は、A 4 と一緒に）	B5	親	引数 2（double、long、および long long の場合は、B4 と一緒に）
A6	親	引数 3	B6	親	引数 4
A7	親	引数 3（double、long、および long long の場合は、A6 と一緒に）	B7	親	引数 4（double、long、および long long の場合は、B6 と一緒に）
A8	親	引数 5	B8	親	引数 6
A9	親	引数 5（double、long、および long long の場合は、A8 と一緒に）	B9	親	引数 6（double、long、および long long の場合は、B8 と一緒に）
A10	子	引数 7	B10	子	引数 8
A11	子	引数 7（double、long、および long long の場合は、A10 と一緒に）	B11	子	引数 8（double、long、および long long の場合は、B10 と一緒に）
A12	子	引数 9	B12	子	引数 10
A13	子	引数 9（double、long、および long long の場合は、A12 と一緒に）	B13	子	引数 10（double、long、および long long の場合は、B12 と一緒に）
A14	子	--	B14	子	データ・ページ・ポインタ（DP）
A15	子	フレーム・ポインタ（FP）	B15	子	スタック・ポインタ（SP）
A16 ～ A31	親	C64x のみ	B16 ～ B31	親	C64x のみ

8.4 関数の構造と呼び出し規則

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。

8.4.1 関数の呼び出し方法

関数（親関数）は、別の関数（子関数）を呼び出すときに次の作業を実行します。

1） 関数に渡される引数が、レジスタまたはスタックに置かれます。

引数が関数に渡される場合、最初の 10 個までの引数は、レジスタ A4、B4、A6、B6、A8、B8、A10、B10、A12、B12 に置かれます。long、long long、double、または long double が渡される場合は、レジスタのペア A5:A4、B5:B4、A7:A6 などに置かれます。

残りの引数は、すべてスタックに置かれます（つまりスタック・ポインタが次に空いている位置を指し、SP + *offset* が 11 番めの引数を指します）。スタックに置かれる引数は、そのサイズに適した値に位置合わせされなければなりません。プロトタイプで宣言されていない引数で、int のサイズ より小さい引数は、int として渡されます。float である引数がプロトタイプで宣言されていない場合は、double として渡されます。

構造体引数は、構造体のアドレスとして渡されます。ローカル・コピーを作成するのは、呼び出し先関数です。

関数が、可変数の引数を指定して呼び出されることを示す省略符号を指定して宣言される場合は、規則がやや変更されます。明示的に宣言された最後の引数がスタック上で渡されるので、そのスタック・アドレスが、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。

図 8–7 は、レジスタ引数の規則を示しています。

図 8–7. レジスタ引数の規則

```
int func1(int a,  int b,    int c);
A4        A4      B4        A6

int func2(int a,  float b,  int *c,  struct A d,  float e,  int f,  int g);
A4        A4      B4        A6       B6           A8        B8      A10

int func3(int a,  double b, float c, long double d);
A4        A4      B5:B4     A6       B7:B6

/* NOTE: The following function has a variable number of arguments */

int vararg(int a, int b,    int c,   int d,       ...);
A4        A4      B4        A6       stack       ...

struct A func4(   int y);
A3                A4
```

2) 呼び出し元の関数は、レジスタ A0 ～ A9 と B0 ～ B9（C64x の場合、A16 ～ A31 と B16 ～ B31）の値が呼び出し後に必要な場合は、その値をスタックに入れることによりそれらのレジスタを保管する必要があります。

3) 呼び出し元（親）が、関数（子）を呼び出します。

4) 呼び出し元は戻った後、スタック・ポインタに加算することにより引数に必要なスタック空間を再利用します。このステップが必要なのは、C/C++ コードからコンパイルされなかったアセンブリ・プログラムだけです。これは、C/C++ コンパイラが関数の先頭ですべての呼び出しに必要なスタック空間を割り当て、関数の最後で空間を割り当て解除するからです。

8.4.2 呼び出し先関数の対応方法

呼び出し先関数（子関数）は、次の作業を実行します。

1) 呼び出し先関数（子）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空間をスタック上に割り当てます。この割り当ては関数の先頭で一度行われ、フレーム・ポインタ（FP）の割り当てが含まれる場合があります。

   フレーム・ポインタは、スタックから引数を読み取り、レジスタのスピル命令を処理するのに使用されます。引数がスタック上に置かれている場合、またはフレーム・サイズが 128K バイトを超える場合、フレーム・ポインタ（A15）は次のように割り当てられます。

   a) 元の A15 がスタック上に保管されます。

   b) 新しいフレーム・ポインタが、現在の SP（B15）に設定されます。

c） SP を定数分減らして、フレームが割り当てられます。

d） A15（FP）も B15（SP）も、この関数内の他の場所で減らされません。

上記の条件に合致しない場合、フレーム・ポインタ（A15）は割り当てられません。この状態で、フレームは、レジスタ B15（SP）から定数を差し引くことにより割り当てられます。レジスタ B15（SP）は、この関数内の他の部分では減らされません。

2） 呼び出し先関数が他の関数を呼び出す場合は、戻りアドレスをスタック上に保管する必要があります。それ以外の場合は戻りレジスタ（B3）に置かれたままであり、次の関数呼び出しにより上書きされます。

3） A10 ～ A15 または B10 ～ B15 のレジスタを呼び出し先関数が修正する場合は、それらを他のレジスタまたはスタック上に保管しなければなりません。呼び出し先関数は、他のレジスタを保管せずに変更できます。

4） 呼び出し先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・コピー用の空間をスタック上に割り当てて、構造体を渡されたポインタからローカル・コピーにコピーしなければなりません。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。

構造体引数を受け入れる関数を宣言する場合は、（追加の構造体引数が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）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） 呼び出し先関数は、関数のコードを実行します。

6） 呼び出し先関数が integer、pointer、または float 型を戻す場合、戻り値は A4 レジスタに置かれます。関数が double、long double、または long long 型を戻す場合、値は A5:A4 レジスタ・ペアに置かれます。

関数が構造体を戻す場合、呼び出し元はその構造体用の空間を割り当ててから、A3 の中で呼び出し先関数に構造体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、呼び出し側は構造体をどこに戻せばよいかを上手に呼び出し先関数に知らせることができます。たとえば s = f(x) 文で、s が構造体であり f が構造体を戻す関数である場合、呼び出し元は実際に f(&s, x) として呼び出しを行うことができます。そうすると、関数 f は戻りの構造体を直接 s の中にコピーし、この代入は自動的に実行されます。

呼び出し元が戻りの構造体値を使用しない場合は、アドレス値 0 を最初の引数として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、（追加の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。

7） ステップ 3 で保存された A10 ～ A15 または B10 ～ B15 のレジスタが、すべて復元されます。

8） A15 がフレーム・ポインタ（FP）として使用された場合、A15 の元の値がスタックから復元されます。ステップ 1 で関数に割り当てられた空間は、レジスタ B15（SP）に定数を加算することにより関数の終わりで再利用されます。

9） 関数は、戻りレジスタ（B3）の値または戻りレジスタの保存値にジャンプすることで戻ります。

8.4.3 引数とローカル変数へのアクセス方法

関数は、スタックの最上部を指すレジスタ A15（FP）またはレジスタ B15（SP）を介して、間接的にそのスタック引数およびレジスタ変数以外のローカル変数にアクセスします。スタックは下位アドレスに向かって伸長するので、関数のローカル・データと引数データは、FP または SP から正のオフセットでアクセスされます。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に対するスタック引数用に確保された領域は、その関数の始めで FP または SP から差し引かれた定数よりも小さいオフセットでアクセスされます。

この関数に渡されるスタック引数は、その関数の始めでレジスタ FP または SP から差し引かれた定数以上のオフセットでアクセスされます。最適化が行われる場合、またはレジスタ引数が register キーワードを使用して定義される場合には、コンパイラはそのレジスタ引数を元のレジスタに保存しようとします。それ以外の場合、引数はスタックにコピーされ、今後の割り当て用にそれらのレジスタを解放します。

ローカル変数、一時記憶域、およびスタック引数へのアクセスに使用されるのが FP であるか SP であるかについては、8.4.2 項「呼び出し先関数の対応方法」（8-20 ページ）を参照してください。C/C++ システム・スタックの詳細は、8.1.2 項「C/C++ システム・スタック」（8-4 ページ）を参照してください。

8.5 アセンブリ言語と C および C++ 言語間のインターフェイス

アセンブリ言語を C/C++ コードと一緒に使用する方法は、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（8.5.1 項を参照）。
- ❑ 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（8.5.2 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」（8-26 ページ）を参照）。
- ❑ 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（8.5.8 項「インライン・アセンブリ言語の使用方法」（8-43 ページ）を参照）。
- ❑ アセンブリ言語の変数と定数を C/C++ ソースの中で使用します（8.5.9 項「アセンブリ言語変数に C/C++ からアクセスする方法」（8-44 ページ）を参照）。

8.5.1 C/C++ コードでのアセンブリ言語モジュールの使用方法

8.4 節「関数の構造と呼び出し規則」（8-19 ページ）で定義された呼び出し規則、および 8.3 節「レジスタ規則」（8-17 ページ）で定義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 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。

- ❑ 関数が C/C++ で作成されているかアセンブリ言語で作成されているかにかかわらず、すべての関数が 8.3 節「レジスタ規則」（8-17 ページ）に概要が記載されているレジスタ規則に従わなければなりません。
- ❑ レジスタ A10 ～ A15、B3、B10 ～ B15 を保存しなければ なりません。A3 の保存が必要な場合もあります。スタックを通常どおりに使用する場合は、スタックを明示的に保存する必要はありません。つまり、関数が終了する前にプッシュしたすべてのものをポップする限り、関数内でスタックを自由に使用できます。その他のすべてのレジスタは、内容を保存せずに自由に使用できます。
- ❑ 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、8.6 節「割り込み処理」（8-46 ページ）を参照してください。

- アセンブリ言語から C/C++ 関数を呼び出す場合は、8.4.1 項「関数の呼び出し方法」（8-19 ページ）に説明されているように、指定されたレジスタを引数と一緒にロードし、残りの引数をスタックに入れます。

  C/C++ コンパイラが保存するのは A10 ～ A15 および B10 ～ B15 だけであることに注意してください。C/C++ 関数は他のレジスタを変更できます。関数の呼び出し前に内容を保存する必要のあるレジスタをスタックに保管し、関数が戻った後にそれらを復元できます。

- 関数は、C/C++ 宣言に従って値を正しく戻さなければなりません。整数および 32 ビット浮動小数点（float）値は、A4 で戻されます。double、long double、long、および long long は A5:A4 で戻されます。構造体は、A3 のアドレスにコピーすることにより戻されます。

- アセンブリ・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。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 に入れてテーブルを壊すと、予期できない結果が生じます。

- コンパイラは、すべての外部オブジェクトにリンク名を割り当てます。したがってアセンブリ言語コードを作成する場合は、コンパイラが割り当てたものと同じリンク名を使用しなければなりません。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は、下線（_）で始めることはできません。詳細は、7.8 節「リンク名の生成」（7-33 ページ）を参照してください。

- 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、すべてアセンブリ言語修飾子内で .def または .global 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宣言され、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にアクセスする場合には、C/C++ オブジェクトをアセンブリ言語モジュールの中で .ref または .global 疑似命令によって宣言します。これにより、リンカが解決できる未宣言の外部参照が作成されます。

例 8-1 は、asmfunc アセンブリ言語関数を呼び出す main C++ 関数を示しています。asmfunc 関数は引数を 1 つ必要とし、それを gvar C++ グローバル変数に加算し、その結果を戻します。

例 8–1. アセンブリ言語関数を C/C++ から呼び出す方法

(a)C プログラム

```
extern "C" {
extern int asmfunc(int a);/* declare external as function*/
int gvar = 4;             /* 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 _asmfunc
   .global _gvar
_asmfunc:
   LDW     *+b14(_gvar),A3
   NOP     4
   ADD     a3,a4,a3
   STW     a3,*b14(_gvar)
   MV      a3,a4
   B       b3
   NOP     5
```

例 8–1 の C++ プログラムでは、戻りの型が int なので、asmfunc の外部宣言は省略してもかまいません。C/C++ 関数のように、アセンブリ関数が非整数値を戻すか非整数パラメータを渡す場合にのみ、そのアセンブリ関数を宣言しなければなりません。

注：SP の意味構造

スタック・ポインタは常に 8 バイト境界に位置合わせする必要があります。これは、C コンパイラおよびランタイムサポート・ライブラリのシステム初期化コードによって自動的に実行されます。C またはリニア・アセンブリ・ソースで定義された関数を呼び出す手書きのアセンブリ・コードもスタック上に 8 の倍数の空間を確保する必要があります。

注 : スタックの割り当て

コンパイラがスタックのダブルワード位置合わせを保証し、スタック・ポインタ（SP）がスタック空間の次に空いているロケーションを指す場合でも、そのロケーションには 1 個の 32 ビット・ワードを格納する空間しか保証されません。呼び出された関数がダブルワードを格納する空間を割り当てる必要があります。

8.5.2 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

C6000 C/C++ コンパイラは、多くの組み込み関数を認識します。組み込み関数を使用すると、C/C++ では他の方法によって表現できないような扱いにくいアセンブリ文の意味を表現できます。組み込み関数は関数のように使用され、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。

```
int x1, x2, y;
y = _sadd(x1, x2);
```

表 8–3 に掲載されている組み込み関数は、すべての C6000 デバイスに対して使用できます。これらは示された C6000 アセンブリ言語命令に対応します。詳細は、TMS320C6000 CPU and Instruction Set Reference Guide を参照してください。

注 : C とアセンブリ言語の組み込み関数命令

組み込み関数に正確に対応するアセンブリ言語命令をコンパイラが使用しない場合があります。このような場合でも、プログラムの意味は変わりません。

C64x 固有の組み込み関数の一覧は、表 8–4 を参照してください。C67x 固有の組み込み関数の一覧は、表 8–5 を参照してください。

表 8–3. TMS320C6000 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 **_abs(**int *src*)**;** int **_labs(**long *src*)**;**	**ABS**	src の飽和絶対値を戻します。
int **_add2(**int *src1*, int *src2*)**;**	**ADD2**	src1 の上位 16 ビットと下位 16 ビットを src2 の上位 16 ビットと下位 16 ビットに加算し、結果を戻します。下位 16 ビットの加算からのオーバーフローは、上位 16 ビットの加算に影響を与えません。
ushort & **_amem2(**void **ptr*)**;**	**LDHU** **STHU**	2 バイトを位置合わせしてメモリにロードし、保存します。†
const ushort & **_amem2_const(** const void **ptr*)**;**	**LDHU**	2 バイトを位置合わせしてメモリからロードします。†
uint & **_amem4(**void **ptr*)**;**	**LDW** **STW**	4 バイトを位置合わせしてメモリにロードし、保存します。†
const uint & **_amem4_const(** const void **ptr*)**;**	**LDW**	4 バイトを位置合わせしてメモリからロードします。†
double & **_amemd8(**void **ptr*)**;**	**LDW/LDW** **STW/STW**	8 バイトを位置合わせしてメモリにロードし、保存します。†‡ C64x の場合、_amemd は、他の C6000 デバイスで使用する場合と異なるアセンブリ命令に対応します。詳細は、表 8–4 を参照してください。
const double & **_amemd8_const(** const void **ptr*)**;**	**LDDW**	8 バイトを位置合わせしてメモリからロードします。†‡
uint **_clr(**uint *src2,* uint *csta*, uint *cstb*)**;**	**CLR**	src2 内の指定されたフィールドをクリアします。クリアするフィールドの先頭のビットと最後のビットは、それぞれ csta と cstb により指定されます。
uint **_clrr(**uint *src2*, int *src1*)**;**	**CLR**	src2 内の指定されたフィールドをクリアします。クリアするフィールドの先頭のビットと最後のビットは、src1 の下位 10 ビットにより指定されます。
ulong **_dtol(**double *src*)**;**		double のレジスタ・ペア src を unsigned long のレジスタ・ペアとして解釈し直します。

† 詳細は、TMS320C6000 Programmer's Guide を参照してください。
‡ 8 バイトのデータ数量の操作方法については、8.5.3 項「位置合わせされていないデータと 64 ビット値の使用方法」（8-36 ページ）を参照してください。

表 8–3. TMS320C6000 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 **_ext(**int *src2,* uint *csta*, uint *cstb***);**	**EXT**	src2 内の指定されたフィールドを抽出し、32 ビットに符号拡張します。この抽出は、左シフトの後に符号付き右シフトによって実行されます。csta と cstb は、それぞれ左シフトと右シフトの量です。
int **_extr(**int *src2*, int *src1***)**	**EXT**	src2 内の指定されたフィールドを抽出し、32 ビットに符号拡張します。この抽出は、左シフトの後に符号付き右シフトによって実行されます。左シフトと右シフトの量は、src1 の下位 10 ビットで指定されます。
uint **_extu(**uint *src2,* uint *csta*, uint *cstb***);**	**EXTU**	src2 内の指定されたフィールドを抽出し、32 ビットにゼロ拡張します。この抽出は、左シフトの後に符号なし右シフトにより実行されます。csta と cstb は、それぞれ左シフトと右シフトの量です。
uint **_extur(**uint *src2*, int *src1***);**	**EXTU**	src2 内の指定されたフィールドを抽出し、32 ビットにゼロ拡張します。この抽出は、左シフトの後に符号なし右シフトにより実行されます。左シフトと右シフトの量は、src1 の下位 10 ビットで指定されます。
uint **_ftoi(**float *src***);**		float 内のビットを、unsigned int として解釈し直します。たとえば次のとおりです。 _ftoi (1.0) == 1065353216U
uint **_hi(**double *src***);**		double レジスタ・ペアの上位（奇数）レジスタを戻します。
double **_itod(**uint *src2*, uint *src1***)**		2 つの unsigned int 値を解釈し直して、新しい double レジスタ・ペアを作成します。ここで src2 は上位（奇数）レジスタ、src1 は下位（偶数）レジスタです。
float **_itof(**uint *src***);**		unsigned int のビットを float として解釈し直します。たとえば次のとおりです。 _itof (0x3f800000)==1.0
long long **_itoll(**uint *src2*, uint *src1***)**		2 つの unsigned int 値を解釈し直して、新しい long long レジスタ・ペアを作成します。ここで src2 は上位（奇数）レジスタ、src1 は下位（偶数）レジスタです。

† 詳細は、TMS320C6000 Programmer's Guide を参照してください。
‡ 8 バイトのデータ数量の操作方法については、8.5.3 項「位置合わせされていないデータと 64 ビット値の使用方法」（8-36 ページ）を参照してください。

表 8–3. TMS320C6000 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
uint **_lo(**double *src***);**		double レジスタ・ペアの下位（偶数）レジスタを戻します。
uint **_lmbd(**uint s*rc1,* uint *src2***);**	**LMBD**	src1 の LSB の値 0 か 1 を、src2 の左端から検索します。ビットが検出されるまでのビット数を戻します。
double **_ltod(**long *src***);**		long のレジスタ・ペア src を double のレジスタ・ペアとして解釈し直します。
int **_mpy(**int *src1***,** int *src2***);**	**MPY**	src1 の下位 16 ビットと src2 の下位 16 ビットを乗算し、その結果を戻します。値は、符号付きでも符号なしでも構いません。
int **_mpyus(**uint *src1***,** int *src2***);**	**MPYUS**	
int **_mpysu(**int *src1***,** uint *src2***);**	**MPYSU**	
uint **_mpyu(**uint *src1***,** uint *src2***);**	**MPYU**	
int **_mpyh(**int *src1***,** int *src2***);**	**MPYH**	src1 の上位 16 ビットと src2 の上位 16 ビットを乗算し、その結果を戻します。値は、符号付きでも符号なしでも構いません。
int **_mpyhus(**uint *src1***,** int *src2***);**	**MPYHUS**	
int **_mpyhsu(**int *src1***,** uint *src2***);**	**MPYHSU**	
uint **_mpyhu(**uint *src1***,** uint *src2***);**	**MPYHU**	
int **_mpyhl(**int *src1***,** int *src2***);**	**MPYHL**	src1 の上位 16 ビットと src2 の下位 16 ビットを乗算し、その結果を戻します。値は、符号整数でも符号なしでも構いません。
int **_mpyhuls(**uint *src1***,** int *src2***);**	**MPYHULS**	
int **_mpyhslu(**int *src1***,** uint *src2***);**	**MPYHSLU**	
uint **_mpyhlu(**uint *src1***,** uint *src2***);**	**MPYHLU**	
int **_mpyhl(**int *src1***,** int *src2***);**	**MPYLH**	src1 の下位 16 ビットと src2 の上位 16 ビットを乗算し、その結果を戻します。値は、符号付きでも符号なしでも構いません。
int **_mpyluhs(**uint *src1***,** int *src2***);**	**MPYLUHS**	
int **_mpylshu(**int *src1***,** uint *src2***);**	**MPYLSHU**	
uint **_mpylhu(**uint *src1***,** uint *src2***);**	**MPYLHU**	
void **_nassert(**int**);**		コードを生成しません。assert 関数で宣言された式が真であることを、オプティマイザに伝えます。これは、どの最適化が有効であるかのヒントをオプティマイザに提供します。
uint **_norm(**int *src2***);** uint **_lnorm(**long *src2***);**	**NORM**	src2 の最初の非冗長符号ビットまでのビット数を戻します。

† 詳細は、TMS320C6000 Programmer's Guide を参照してください。
‡ 8 バイトのデータ数量の操作方法については、8.5.3 項「位置合わせされていないデータと 64 ビット値の使用方法」（8-36 ページ）を参照してください。

表 8–3. TMS320C6000 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 **_sadd(**int *src1,* int *src2***);** long **_lsadd(**int *src1,* long *src2***);**	**SADD**	src1 を src2 に加算し、結果を飽和させます。結果を戻します。
int **_sat(**long *src2***);**	**SAT**	40 ビット long を 32 ビット signed int に変換し、必要に応じて飽和させます。
uint **_set(**uint *src2,* uint *csta*, uint *cstb***);**	**SET**	src2 内の指定されたフィールドをすべて 1 に設定し、src2 の値を戻します。設定するフィールドの先頭のビットと最後のビットは、それぞれ csta と cstb により指定されます。
unit **_setr(**unit *src2*, int *src1***);**	**SET**	src2 内の指定されたフィールドをすべて 1 に設定し、src2 の値を戻します。設定するフィールドの先頭のビットと最後のビットは、src1 の下位 10 ビットにより指定されます。
int **_smpy (**int *src1*, int *sr2***);** int **_smpyh (**int *src1*, int *sr2***);** int **_smpyhl (**int *src1*, int *sr2***);** int **_smpylh (**int *src1*, int *sr2***);**	**SMPY** **SMPYH** **SMPYHL** **SMPYLH**	src1 と src2 を乗算し、結果を 1 だけ左にシフトし、結果を戻します。結果 が 0x80000000 の場合、結果を 0x7FFFFFFF に飽和させます。
int **_sshl (**int *src2*, uint *src1***);**	**SSHL**	src1 の内容分 src2 を左にシフトし、結果を 32 ビットに飽和させ、結果を戻します。
int **_ssub (**int *src1,* int *src2***);** long **_lssub (**int *src1,* long *src2***);**	**SSUB**	src2 を src1 から減算し、結果を飽和させ、その結果を戻します。
uint **_subc (**uint *src1,* uint *src2***);**	**SUBC**	条件付きの減算除算ステップ
int **_sub2 (**int *src1*, int *src2***);**	**SUB2**	src2 の上位 16 ビットと下位 16 ビットを src1 の上位ビットと下位ビットから減算し、結果を戻します。下位 16 ビット分の減算での借りは、上位 16 ビット分の減算に影響を与えません。

† 詳細は、TMS320C6000 Programmer's Guide を参照してください。
‡ 8 バイトのデータ数量の操作方法については、8.5.3 項「位置合わせされていないデータと 64 ビット値の使用方法」（8-36 ページ）を参照してください。

表 8–4 に示す組み込み関数は、C64x デバイスに対してのみ使用できます。これらの組み込み関数は、C6000 アセンブリ言語命令に対応します。詳細は、TMS320C6000 CPU and Instruction Set Reference Guide を参照してください。

C6000 の汎用組み込み関数の一覧は、表 8–3 を参照してください。C67x 固有の組み込み関数の一覧は、表 8–5 を参照してください。

表 8–4. TMS320C64x 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 **_abs2(**int *src***);**	**ABS2**	2 つの 16 ビット値ごとの絶対値を計算します。
int **_add4(**int *src1***,** int *src2***);**	**ADD4**	パックされた 4 つの 8 ビット数のペアに対して、2 の補数の加算を実行します。
long long & **_amem8(**void **ptr***)**;	**LDDW** **STDW**	8 バイトを位置合わせしてメモリにロードし、保存します。
const long long & **_amem8_const(** const void **ptr***)**;	**LDDW**	8 バイトを位置合わせしてメモリからロードします。‡
double & **_amemd8(**void **ptr***)**;	**LDDW** **STDW**	8 バイトを位置合わせしてメモリにロードし、保存します。†‡ C64x の場合、_amemd は、他の C6000 デバイスで使用する場合と異なるアセンブリ命令に対応します。表 8–3 を参照してください。
const double & **_amemd8_const(** const void **ptr***)**;	**LDDW**	8 バイトを位置合わせしてメモリからロードします。†‡
int **_avg2(**int *src1***,** int *src2***);**	**AVG2**	2 つの符号付き 16 ビット値のペアごとに、平均を計算します。
uint **_avgu4(**uint**,** uint**);**	**AVGU4**	4 つの符号付き 8 ビット値のペアごとに、平均を計算します。
uint **_bitc4(**uint *src***);**	**BITC4**	src 内の 4 つの 8 ビット値ごとに、"1" ビットの数が戻り値内の対応する位置に書き込まれます。
uint **_bitr(**uint *src***);**	**BITR**	ビットの順序を逆にします。
int **_cmpeq2(**int *src1***,** int *src2***);**	**CMPEQ2**	2 つの 16 ビット値のペアごとに、同等比較を実行します。比較の結果は、戻り値の LSB 2 ビットにパックされます。
int **_cmpeq4(**int *src1***,** int *src2***);**	**CMPEQ4**	4 つの 8 ビット値のペアごとに、同等比較を実行します。比較の結果は、戻り値の LSB 4 ビットにパックされます。
int **_cmpgt2(**int *src1***,** int *src2***);**	**CMPGT2**	2 つの符号付き 16 ビット値の各ペアを比較します。結果は、戻り値の LSB 2 ビットにパックされます。

† 詳細は、TMS320C6000 Programmer's Guide を参照してください。
‡ 8 バイトのデータ数量の操作方法については、8.5.3 項「位置合わせされていないデータと 64 ビット値の使用方法」（8-36 ページ）を参照してください。

表 8–4. TMS320C64x 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
uint _**cmpgtu4(**uint *src1*, uint *src2*);	**CMPGTU4**	4 つの 8 ビット値のペアごとに比較します。結果は、戻り値の LSB 4 ビットにパックされます。
uint _**deal(**uint *src*);	**DEAL**	src の奇数ビットと偶数ビットが抽出され、2 つの別々の 16 ビット値になります。
int _**dotp2(**int *src1*, int *src2*); long _**ldotp2(**int *src1*, int *src2*);	**DOTP2** **LDOTP2**	src1 と src2 内の符号付き下位 16 ビット値の積を、src1 と src2 内の符号付き上位 16 ビット値の積に加算します。 _lo および _hi 組み込み関数を使用して、64 ビット整数の結果のそれぞれ半分にアクセスする必要があります。
int _**dotpn2(**int *src1*, int *src2*);	**DOTPN2**	src1 と src2 内の符号付き下位 16 ビット値の積を、src1 と src2 内の符号付き上位 16 ビット値の積から減算します。
int _**dotpnrsu2(**int *src1*, uint *src2*);	**DOTPNRSU2**	src1 と src2 内の符号なし下位 16 ビット値の積を、src1 と src2 内の符号付き上位 16 ビット値の積から減算します。2 の 15 乗が加算され、結果は符号付きで右に 16 シフトされます。
int _**dotprsu2(**int *src1*, uint *src2*);	**DOTPRSU2**	最初の符号付き 16 ビット値のペアの積を、2 番目の符号なし 16 ビット値のペアに加算します。2 の 15 乗が加算され、結果は符号付きで右に 16 シフトされます。
int _**dotprsu4(**int *src1*, uint *src2*); uint _**dotpu4(**uint *src1*, uint *src2*);	**DOTPRSU4** **DOTPU4**	4 つの src1 と src2 内の 8 ビット値のペアごとに、src1 の 8 ビット値 が src2 の 8 ビット値に乗算されます。4 つの積が合計されます。
int _**gmpy4(**int *src1*, int *src2*);	**GMPY4**	src1 内の 4 つの値と src2 内の 4 つの並列値とのガロア域乗算を実行します。この 4 つの積は、戻り値にパックされます。

int _**max2 (**int *src1*, int *src2*);
int _**min2(**int *src1*, int *src2*);
uint _**maxu4(**uint *src1*, uint *src2*);
uint _**minu4(**uint *src1*, uint *src2*);
	MAX2
	MIN2
	MAX4
	MINU4
		戻り値内の対応する位置に、値の各ペアの大きい値と小さい値を入れます。値は、16 ビットの符号付き値でも 8 ビットの符号なし値でも構いません。

† 詳細は、TMS320C6000 Programmer's Guide を参照してください。
‡ 8 バイトのデータ数量の操作方法については、8.5.3 項「位置合わせされていないデータと 64 ビット値の使用方法」（8-36 ページ）を参照してください。

表 8–4. TMS320C64x 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
ushort & **_mem2(**void * *ptr***);**	**LDB/LDB** **STB/STB**	2 バイトを位置合わせなしにメモリにロードし、保存します。†
const ushort & **_mem2_const(** const void **ptr***);*	**LDB/LDB**	2 バイトを位置合わせなしにメモリにロードします。†
uint & **_mem4(**void * *ptr***);**	**LDNW** **STNW**	4 バイトを位置合わせなしにメモリにロードし、保存します。†
const uint & **_mem4_const(** const void **ptr***);*	**LDNW**	4 バイトを位置合わせなしにメモリからロードします。†
long long & **_mem8(**void * *ptr***);**	**LDNDW** **STNDW**	8 バイトを位置合わせなしにメモリにロードし、保存します。†
const long long & **_mem8_const(** const void **ptr***);*	**LDNDW**	8 バイトを位置合わせなしにメモリからロードします。†
double & **_memd8(**void * *ptr***)**	**LDNDW** **STNDW**	8 バイトを位置合わせなしにメモリにロードし、保存します。†‡
const double & **_memd8_const(** const void * *ptr***)**	**LDNDW**	8 バイトを位置合わせなしにメモリからロードします。†‡
double **_mpy2(**int *src1***,** int *src2***);**	**MPY2**	src1 と src2 内の下位と下位、上位と上位の 16 ビット値の積を戻します。
double **_mpyhi(**int *src1***,** int *src2***);** double **_mpyli(**int *src1***,** int *src2***);**	**MPYHI** **MPYLI**	符号付きの 16 ビットと 32 ビットの乗算を行います。結果は、戻された double の下位 48 ビットに入ります。src1 の上位または下位の 16 ビットを使用できます。
int **_mpyhir(**int *src1***,** int *src2***);** int **_mpylir(**int *src1***,** int *src2***);**	**MPYHIR** **MPYLIR**	符号付きの 16 ビットと 32 ビットの乗算を行います。結果は、15 ビット分右シフトされます。src1 の上位または下位の 16 ビットを使用できます。
double **_mpysu4 (**int *src1***,** uint *src2***);** double **_mpyu4 (**uint *src1***,** uint *src2***);**	**MPYSU4** **MPYU4**	src1 と src2 内の 8 ビット数量ごとに、8 ビット x 8 ビットの乗算を行います。4 つの 16 ビットの結果が double にパックされます。結果は、符号付きでも符号なしでも構いません。
int **_mvd (**int *src2***);**	**MVD**	乗算器パイプラインを使用して、4 サイクルでデータを src2 から戻り値に移動します。
uint **_pack2 (**uint *src1***,** uint *src2***);** uint **_packh2 (**uint *src1***,** uint *src2***);**	**PACK2** **PACKH2**	src1 と src2 の下位／上位のハーフワードが、戻り値の上位と下位のハーフワードに入れられます。

† 詳細は、TMS320C6000 Programmer's Guide を参照してください。
‡ 8 バイトのデータ数量の操作方法については、8.5.3 項「位置合わせされていないデータと 64 ビット値の使用方法」（8-36 ページ）を参照してください。

表 8–4. TMS320C64x 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
uint **_packh4 (**uint *src1*, uint *src2*);	**PACKH4**	戻り値にバイトを交互にパックします。高位バイトまたは低位バイトをパックできます。
uint **_packl4 (**uint *src1*, uint *src2*);	**PACKL4**	
uint **_packhl2 (**uint *src1*, uint *src2*);	**PACKHL2**	src1 の上位／下位のハーフワードが、戻り値の上位のハーフワードに入れられます。src2 の下位／上位のハーフワードが、戻り値の下位のハーフワードに入れられます。
uint **_packlh2 (**uint *src1*, uint *src2*);	**PACKLH2**	
uint **_rotl (**uint *src1*, uint *src2*);	**ROTL**	src1 の量だけ、src2 を左に回転します。
int **_sadd2(**int *src1*, int *src2*);	**SADD2**	src1 と src2 内の 16 ビット値のペア間で飽和加算を実行します。src1 値 は、符号付きでも符号なしでも構いません。
int **_saddus2(**uint *src1*, int *src2*);	**SADDUS2**	
uint **_saddu4(**uint *src1*, uint *src2*);	**SADDU4**	4 つの src1 と src2 内の 8 ビット符号なし整数値のペア間で飽和加算を実行します。
uint **_shfl (**uint *src2*);	**SHFL**	src2 の下位 16 ビットが偶数ビット位置に置かれ、src の上位 16 ビットが 奇数ビット位置に置かれます。
uint **_shlmb (**uint *src1*, uint *src2*);	**SHLMB**	src2 を 1 バイト分左または右にシフトし、src1 の最大または最小重みバイトが、最小または最大重みバイト位置にマージされます。
uint **_shrmb (**uint *src1*, uint *src2*);	**SHRMB**	
int **_shr2 (**int *src1*, uint *src2*);	**SHR2**	src2 内の 16 ビット数量ごとに、その数量が、src1 のビット数分、右に算術または論理シフトされます。src2 には、符号付き値または符号なし値が入ります。
uint **shru2 (**uint *src1*, uint *src2*);	**SHRU2**	
double **_smpy2 (**int *src1*, int *sr2*);	**SMPY2**	パックされた符号付きの 16 ビット値のペア間で 16 ビット乗算を実行し、1 ビット分左にシフトし、double 結果に飽和させます。
int **_spack2 (**int *src1*, int *sr2*);	**SPACK2**	2 つの符号付き 32 ビット値が 16 ビット値に飽和され、戻り値にパックされます。
uint **_spacku4 (**int *src1*, int *sr2*);	**SPACKU4**	4 つの符号付き 16 ビット値が 8 ビット値に飽和され、戻り値にパックされます。
int **_sshvl (**int *src2*, int *src1*);	**SSHVL**	src2 を左／右に src1 ビットだけシフトします。シフトされた値が MAX_INT より大きいか、MIN_INT より小さい場合、結果を飽和させます。
int **_sshvr (**int *src2*, int *src1*);	**SSHVR**	
int **_sub4 (**int *src1*, int *src2*);	**SUB4**	4 つのパックされた 8 ビット値のペア間で、2 の補数の減算を実行します。

† 詳細は、TMS320C6000 Programmer's Guide を参照してください。

‡ 8 バイトのデータ数量の操作方法については、8.5.3 項「位置合わせされていないデータと 64 ビット値の使用方法」（8-36 ページ）を参照してください。

表 8–4. TMS320C64x 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 **_subabs4 (**int *src1*, int *src2*);	**SUBABS4**	4 つのパックされた 8 ビット値のペアごとに、差の絶対値を計算します。
uint **_swap4 (**uint *src*);	**SWAP4**	各 16 ビット値内のバイトのペアを交換します（エンディアン・スワップ）。
uint **_unpkhu4 (**uint *src*);	**UNPKHU4**	2 つの上位符号なし 8 ビット値を、パックされた符号なし 16 ビット値にアンパックします。
uint **_unpklu4 (**uint *src*);	**UNPKLU4**	2 つの下位符号なし 8 ビット値を、パックされた符号なし 16 ビット値にアンパックします。
uint **_xpnd2 (**uint *src*);	**XPND2**	src のビット 1 と 0 が、それぞれ結果の上位ハーフワードと下位ハーフワードに複写されます。
uint **_xpnd4 (**uint *src*);	**XPND4**	src のビット 3 ～ 0 が、結果のバイト 3 ～ 0 に複写されます。

† 詳細は、TMS320C6000 Programmer's Guide を参照してください。
‡ 8 バイトのデータ数量の操作方法については、8.5.3 項「位置合わせされていないデータと 64 ビット値の使用方法」（8-36 ページ）を参照してください。

表 8–5 に示す組み込み関数は、C67x デバイスに対してのみ使用できます。これらの組み込み関数は、C6000 アセンブリ言語命令に対応します。詳細は、TMS320C6000 CPU and Instruction Set Reference Guide を参照してください。

C6000 の汎用組み込み関数の一覧は、表 8–3 を参照してください。C64x 固有の組み込み関数の一覧は、表 8–4 を参照してください。

表 8–5. TMS320C67x 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 **_dpint(**double *src*);	**DPINT**	CSR レジスタにより設定された丸めモードを使用して、64 ビット double を 32 ビット signed int に変換します。
double **_fabs(**double *src*);	**ABSDP**	src の絶対値を戻します。
float **_fabsf(**float *src*);	**ABSSP**	
double **_mpyid (**int *src1*, int *src2*);	**MPYID**	符号付き整数の乗算を行います。結果はレジスタ・ペアに置かれます。
double **_rcpdp(**double *src*);	**RCPDP**	64 ビット double の逆数近似値を計算します。
float **_rcpsp(**float *src*);	**RCPSP**	32 ビット float の逆数近似値を計算します。

表 8–5. TMS320C67x 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
double **_rsqrdp(**double *src***);**	**RSQRDP**	64 ビット double 平方根の逆数の近似値を計算します。
float **_rsqrsp(**float *src***);**	**RSQRSP**	32 ビット float の平方根の逆数の近似値を計算します。
int **_spint (**float**);**	**SPINT**	CSR レジスタにより設定された丸めモードを使用して、32 ビット float を 32 ビット signed int に変換します。

8.5.3 位置合わせされていないデータと 64 ビット値の使用方法

C64x ファミリーは、_mem8、_memd8、および _mem4 組み込み関数の使用により、64 ビット値と 32 ビット値の位置合わせのないロードと保存をサポートします。64 ビット double から 2 つの 32 ビット部分を抽出するには、_lo および _hi 組み込み関数が便利です。例 8-2 は、_lo、_hi、_mem8、および _memd8 組み込み関数の使用方法を示しています。

例 8–2. _lo および _hi 組み込み関数の使用方法

```
void load_longlong_unaligned(void *a, int *high, int *low)
{
  double d = _memd8(a);

  *high = _hi(d);
  *low  = _lo(d);
}

void store_longlong_unaligned(void *a, int high, int low)
{
  double d  = _itod(high, low);
  _mem8d(a) = d;
}
```

例 8–3. _lo および _hi 組み込み関数を long long 関数と一緒に使用する方法

```
void alt_load_longlong_unaligned(void *a, int *high, int *low)
{
  long long p = _mem8(a);

  *high = p >> 32;
  *low  = (unsigned int) p;
}

void alt_store_longlong_unaligned(void *a, int high, int low)
{
  long long p = _itoll(high, low);
  _mem8(a)    = p;
}
```

8.5.4 MUST_ITERATE と _nassert を使用して SIMD を使用可能にし、ループについてのコンパイラの知識を拡張する方法

MUST_TERATE と _nassert を使用すると、ループが特定の回数実行されることを保証できます。

次の例は、ループが正確に 10 回実行されることが保証されることをコンパイラに伝えます。

```
#pragma MUST_ITERATE(10,10);
for (i = 0; i < trip_count; i++) { ...
```

また、MUST_ITERATE はトリップ・カウントの係数とトリップ・カウントの範囲を指定するのにも使用できます。たとえば次のとおりです。

```
#pragma MUST_ITERATE(8,48,8);
for (i = 0; i < trip; i++) { ...
```

次の例は、ループが 8 回～ 48 回実行され、トリップ変数が 8 の倍数（8、16、24、32、40、48）であることをコンパイラに伝えます。コンパイラはこの情報をすべて使用すれば、-min*n* オプションを使用して割り込みが *n* 回ごとに発生することが指定されている場合であっても、展開のパフォーマンスを向上させることにより最善のループを生成できるようになります。

TMS320C6000 Programmer's Guide では、C/C++ コードを改良する方法の 1 つが、32 ビット・レジスタの上位および下位に保存されている 16 ビット・データで操作する際にワード・アクセスを使用することであると説明しています。int ポインタへのキャストを使用した例が、_mpyh のような特定の命令を使用するための組み込み関数の使用とともに示されています。これを自動化するには、_nassert(); 組み込み関数を使用して、16 ビット short 配列が 32 ビット（ワード）境界で位置合わせされることを指定します。

次の 2 つの例は、同じアセンブリ・コードを生成します。

❑ **例 1**

```
int dot_product(short *x, short *y, short z)
{
    int *w_x = (int *)x;
    int *w_y = (int *)y;
    int sum1 = 0, sum2 = 0, i;
    for (i = 0; i < z/2; i++)
    {
        sum1 += _mpy(w_x[i], w_y[i]);
        sum2 += _mpyh(w_x[i], w_y[i]);
    }
    return (sum1 + sum2);
}
```

❑ **例 2**

```
int dot_product(short *x, short *y, short z)
{
    int sum = 0, i;

    _nassert (((int)(x) & 0x3) == 0);
    _nassert (((int)(y) & 0x3) == 0);
    #pragma MUST_ITERATE(20, , 4);
    for (i = 0; i < z; i++) sum += x[i] * y[i];
    return sum;
}
```

注 : _nassert の C++ 構文

C++ コードでは、_nassert は標準ネームスペースの一部です。したがって、正確な構文は std::_nassert() となります。

8.5.5 データを位置合わせする方法

次のコードでは、_nassert が、f() の呼び出しごとに ptr が 8 バイト境界に位置合わせされることをコンパイラに伝えます。このようなアサーションにより、コンパイラが SIMD（Single Instruction/Multiple Data）とも呼ばれる 1 つの命令で複数のデータ値を処理するコードを生成する可能性があります。

```
void f(short *ptr)
{
   _nassert((int) ptr % 8 == 0)

   ; a loop operating on data accessed by ptr
}
```

次の項で、ptr が参照するデータを確実に位置合わせするために使用できる方法を説明します。f() を呼び出すコードを使用する場合は、常にこれらの方法のいずれかを使用する必要があります。

8.5.5.1 配列の基本アドレス

ptr などの引数には、通常配列のベース・アドレスが渡されます。たとえば次のとおりです。

```
short buffer[100];

...

f(buffer);
```

C64x デバイスをコンパイルする場合、このような配列は 8 バイト境界に自動的に位置合わせされます。C62x または C67x デバイスをコンパイルする場合、このような配列は 4 バイト境界に自動的に位置合わせされ、基本型が要求する場合は 8 バイト境界に位置合わせされます。これは、配列がグローバル、静的、またはローカルにかかわらず当てはまります。上記のそれぞれのデバイスで SIMD 最適化を実現するために必要になるのは、この自動位置合わせだけです。しかし、通常の場合、コンパイラは ptr が正しく位置合わせされた配列のアドレスを保持することを保証できないので、_nassert を組み込む必要があります。

配列の基本アドレスを常に ptr などのポインタに渡す場合は、次のマクロを使用して、この事実を反映することができます。

```
#if   defined(_TMS320C6400)
   #define ALIGNED_ARRAY(ptr) _nassert((int) ptr % 8 == 0)
#elif defined(_TMS320C6200) || defined(_TMS320C6700)
   #define ALIGNED_ARRAY(ptr) _nassert((int) ptr % 4 == 0)
#else
   #define ALIGNED_ARRAY(ptr) /* empty */
#endif

void f(short *ptr)
{
   ALIGNED_ARRAY(ptr);

   ; a loop operating on data accessed by ptr
}
```

どのような C6x デバイスを構築する場合でも、あるいは他のターゲットにコードをポートする場合でも、このマクロは動作します。

8.5.5.2 配列のベースからのオフセット

稀には配列からのオフセットのアドレスを渡す場合があります。たとえば次のとおりです。

```
f(&buffer[3]);
```

このコードは位置合わせされないアドレスを ptr に渡し、その結果 _nassert() にコード化された前提に対する違反が発生します。この違反への直接的な対処方法はありません。このような状況はできる限り回避してください。

8.5.5.3 動的なメモリ割り当て

通常の動的メモリ割り当てでは、バッファのアドレスの位置合わせは保証されません。たとえば次のとおりです。

```
buffer = malloc(100 * sizeof(short));
```

代わりに memalign() で位置合わせ 8 を使用する必要があります。たとえば次のとおりです。

```
buffer = memalign(8, 100 * sizeof(short));
```

BIOS メモリ割り当てルーチンを使用している場合は、以下の構文を使用して、位置合わせ係数を最後の引数として渡す必要があります。

buffer = **MEM_alloc(***segid***,** 100 * sizeof(short), **8);**

BIOS メモリ割り当てルーチンおよび特に *segid* パラメータの詳細は、「TMS320C6000 DSP/BIOS」に関する Help を参照してください。

8.5.5.4 構造体またはクラスのメンバ

構造体またはクラスのメンバである配列は、配列の基本型が要求する場合にのみ位置合わせされます。8.5.5.1 項「配列の基本アドレス」（8-39 ページ）で説明される自動位置合わせは行われません。

例 8–4. 構造体の配列

```
struct s
{
   ...
   short buf1[50];
   ...
} g;

...

f(g.buf1);
```

例 8–5. クラスの配列

```
class c
{
   public :
      short buf1[50];
      void mfunc(void);
   ...
};

void c::mfunc()
{
   f(buf1);
   ...
}
```

構造体またはクラスの配列を位置合わせする最も簡単な方法は、配列の直前に希望する位置合わせを要求するスカラを宣言することです。このため、8 バイト位置合わせを行いたい場合は、long または double を使用します。4 バイト位置合わせを行いたい場合は、int または float を使用します。たとえば次のとおりです。

```
struct s
{
      long  not_used;   /* 8-byte aligned      */
      short buffer[50]; /* also 8-byte aligned */
   ...
};
```

複数の配列を連続して宣言し、所定の位置合わせを保持したい場合は、希望する位置合わせの倍数であっても、バイトで測定された配列のサイズを保持することによって実行できます。たとえば次のとおりです。

```
struct s
{
   long  not_used;/* 8-byte aligned      */
   short buf1[50];/* also 8-byte aligned */
   short buf2[50];/* 4-byte aligned      */
   ...
};
```

buf1 のサイズが 50 * 2 = 100 バイト（2 バイト／ short）で、100 が 8 の倍数ではなく 4 の倍数のため、buf2 は 4 バイト境界上にのみ位置合わせされます。buf1 に 52 個の要素を埋め込むことで、buf2 を 8 バイト境界上に位置合わせします。

構造体またはクラス内では、8 以上の配列の位置合わせを強制する方法はありません。SIMD 最適化のためには、これは必要ありません。

注 : プログラム・レベルの最適化による位置合わせ

ほとんどの場合、プログラム・レベルの最適化（3.6 節「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）」（3-20 ページ）を参照）を実施するには、-pm -o3 オプションを使用してコンパイラを起動させ、すべてのソース・ファイルをコンパイルする必要があります。この結果、コンパイラがすべてのソース・コードを一度に確認し、それ以外の場合にはほとんど行われない最適化が可能になります。たとえば、このような最適化時には、関数 f() に対するすべての呼び出しが配列のベース・アドレスを ptr に渡し、この結果 ptr は常に SIMD 最適化が適用できるように正しく位置合わせされます。この場合、_nassert() は必要ありません。コンパイラは、ptr が位置合わせされているに違いないと自動的に判断し、最適化された SIMD 命令を生成します。

8.5.6 SAT ビットの副次作用

飽和組み込み演算では、飽和が生じる場合に SAT ビットが定義されます。SAT ビットはコントロール・ステータス・レジスタ（CSR）にアクセスすることにより、C/C++ コードから設定したり消去したりできます。コンパイラは、SAT ビットにアクセスするコードを生成するために次のステップを実行します。

1） SAT ビットは、関数呼び出しまたは関数の戻りにより未定義になります。これは、CSR 内の SAT ビットが有効であり、関数呼び出しまたは関数の戻りまで C/C++ コード内で読み取ることができることを意味しています。

2） 関数内のコードが CSR にアクセスすると、コンパイラは SAT ビットがその関数全体に存在するものと想定します。これは、次のことを意味します。

- SAT ビットは、ソフトウェア・パイプライン化されたループの周辺で割り込みをディセーブルにするコードによって保持される。
- 飽和された命令は、見込みで実行できない。

3） 割り込みサービス・ルーチンが SAT ビットを変更する場合、このルーチンは CSR を保存し、復元しなければなりません。

8.5.7 IRP と AMR 規則

コンパイラには IRP と AMR 制御レジスタに関して行う特定の前提事項があります。これらの前提事項は、次のとおりすべてのプログラムで強制的に実行する必要があります。

1） 関数の呼び出しまたは関数からの戻りでは、AMR を 0 に設定する必要があります。関数は AMR を保存し、復元する必要はありませんが、関数から戻る前に AMR を 0 にする必要があります。

2） 割り込みを有効にする場合、またはすべての割り込みで SAVE_AMR および STORE_AMR マクロを使用する必要がある場合は、AMR を 0 に設定する必要があります（8.6.2 項を参照）。

3） IRP は割り込みが無効の場合にのみ安全に変更することができます。

4） IRP を一時レジスタとして使用する場合は、IRP の値を保存し、復元する必要があります。

8.5.8 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 1 行を挿入できます。一連の asm 文を使用すれば、コードを介在させることなくコンパイラ出力の中にアセンブリ言語の連続した行を挿入できます。詳細は、7.6 節「asm 文」（7-17 ページ）を参照してください。

asm 文は、コンパイラ出力の中にコメントを挿入する場合に便利です。次に示すように、単にアセンブリ・コードの文字列の前にセミコロン（;）を付けるだけです。

```
asm”;*** this is an assembly language comment”);
```

注 : asm 文の使用方法

asm 文を使用する場合は、以下の点に注意してください。

- ❏ C/C++ 環境を破壊しないよう細心の注意を払ってください。コンパイラは挿入された命令のチェックや、分析を行いません。
- ❏ C/C++ コードにジャンプまたはラベルを挿入してはなりません。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、予期できない結果が生じる可能性があるからです。
- ❏ asm 文を使用する際に、C/C++ 変数の値を変更してはなりません。コンパイラはこのような文を検証しないからです。asm 文はそのままアセンブリ・コードに挿入され、この文の効果がわからない場合には、問題を引き起こす可能性があります。
- ❏ asm 文を使用して、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なりません。
- ❏ C コードでアセンブリ・マクロを作成し、.g（デバッグ）オプションを使用してコンパイルしてはなりません。C 環境のデバッグ情報とアセンブリ・マクロの展開は互換性がありません。

8.5.9 アセンブリ言語変数に C/C++ からアクセスする方法

アセンブリ言語で定義された変数または定数に 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。これを実現するために使用できるいくつかの方法があり、どの方法を使用するかは、項目が定義されている場所と方法により異なります。つまり、.bss セクション内で定義されている変数か、.bss セクション内で定義されていない変数か、あるいは定数かどうかです。

8.5.9.1 アセンブリ言語のグローバル変数にアクセスする方法

.bss セクションまたは .usect で名前を付けられたセクションに入っている、初期化されない変数にアクセスするのは簡単です。

1) .bss または .usect 疑似命令を使用して変数を定義します。

2) .usect を使用すると変数は .bss 以外のセクションに定義されるので、C で far として宣言する必要があります。

3) .def または .global 疑似命令を使用して、その定義を外部定義にします。

4) アセンブリ言語の中で、名前の前に下線を付けます。

5) C/C++ の中で、その変数を *extern* として宣言し、通常の方法でアクセスします。

例 8-6 は、.bss の中で定義された変数にアクセスする方法を示しています。

例 8–6. C から アセンブリ言語変数にアクセスする方法

(a)C プログラム

```
extern int var1;        /* External variable       */
far extern int var2;    /* External variable       */
var1 = 1;               /* Use the variable        */
var2 = 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    _var1,4,4    ; Define the variable
       .global var1         ; Declare it as external

_var2  .usect  ”mysect”,4,4 ; Define the variable
       .global _var2        ; Declare it as external
```

8.5.9.2 アセンブリ言語定数へのアクセス

.set、.def、および .global 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。または、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を判断できません。

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であるならば、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 8-7 に示すように、キャストと #define を使用できます。

例 8–7. 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 8-7 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。

8.6 割り込み処理

この節の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、また 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に、簡単に asm 文で組み込んだりアセンブリ言語関数を呼び出したりすることで実現できます。

8.6.1 割り込み時のレジスタの保存方法

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。割り込みサービス・ルーチンが C/C++ で作成される場合は、コンパイラがレジスタの保存を処理します。

8.6.2 C/C++ 割り込みルーチンの使用方法

C/C++ 割り込みルーチンは、ローカル変数とレジスタ変数を使用できる点で他の C/C++ 関数と似ています。しかし、このルーチンは引数を指定せずに宣言しなければならず、また void も戻さなければなりません。C/C++ 割り込みルーチンは、ローカル変数用にスタック上に最高 32K を割り当てることができます。たとえば次のとおりです。

```
interrupt void example (void)
{
...
}
```

C/C++ 割り込みルーチンが他の関数を呼び出さない場合は、割り込みハンドラが定義しようとするレジスタだけが保存され、復元されます。しかし C/C++ 割り込みルーチンが他の関数を呼び出す場合、これらの関数は割り込みハンドラが使用しない確認不能レジスタを変更する場合があります。このため、このルーチンは、他の関数が呼び出される場合は使用可能なすべてのレジスタを保存します。割り込みは、割り込み戻りポインタ（IRP）に分岐されます。割り込み処理関数を直接呼び出さないでください。

割り込みは、interrupt プラグマまたは interrupt キーワードを使用することにより、C/C++ 関数で直接処理できます。詳細は、7.7.13 項「INTERRUPT プラグマ」（7-27 ページ）および 7.4.3 項「interrupt キーワード」（7-10 ページ）を参照してください。

ユーザは、割り込みの内部で AMR 制御レジスタおよび CSR 内の SAT ビットを正しく処理する必要があります。デフォルトでは、コンパイラは AMR と SAT ビットを保存または復元するための追加の処理は行いません。SAT ビットと AMR レジスタを処理するためのマクロは、c6x.h ヘッダ・ファイル内に入っています。

たとえば、何らかの手書きのアセンブリ・コードでサーキュラ・アドレッシングを使用するとします（つまり AMR が 0 ではない）。この手書きのアセンブリ・コードは、C コードの割り込みサービス・ルーチンに割り込むことができます。この C コードの割り込みサービス・ルーチンは、AMR が 0 に設定されているものと見なします。C 割り込みサービス・ルーチン内で AMR を正しく保存し復元するには、ローカルの符号なし整数型の一時変数を定義し、C 割り込みサービス・ルーチンの先頭と最後で SAVE_AMR および RESTORE_AMR マクロを呼び出さなければなりません。

例 8–8. AMR と SAT の処理

```
#include <c6x.h>

interrupt void interrupt_func()
{
   unsigned int temp_amr;
   /* define other local variables used inside interrupt */

   /* save the AMR to a temp location and set it to 0 */
   SAVE_AMR(temp_amr);

   /* code and function calls for interrupt service routine */
   ...

   /* restore the AMR for you hand assembly code before exiting */

   RESTORE_AMR(temp_amr);
}
```

C 割り込みサービス・ルーチンで SAT ビットの保存または復元が必要な場合（つまり、何らかの飽和された算術演算も実行する C 割り込みサービス・ルーチンに割り込むときに、飽和された算術演算を実行していた場合）、SAVE_SAT および RESTORE_SAT マクロを使用して上記の例とほぼ同じ方法で実行できます。

8.6.3 アセンブリ言語割り込みルーチンの使用方法

コンパイラと同じレジスタ規則に従うと、アセンブリ言語コードで割り込みを処理できます。すべてのアセンブリ関数のように、割り込みルーチンはスタックを使用し、グローバル C/C++ 変数にアクセスし、C/C++ 関数を正常に呼び出すことができます。C/C++ 関数を呼び出す際は、必ず表 8–2 に掲載されているすべてのレジスタが保存されていることを確認してください。C/C++ 関数がそれらのレジスタを変更する可能性があるからです。

8.7 ランタイム・サポート算術ルーチン

ランタイムサポート・ライブラリには、C6000 命令セットが提供しない C/C++ 算術演算（たとえば整数除算、整数剰余、および浮動小数点の演算）用の算術ルーチンを提供する複数のアセンブリ言語関数が含まれています。

これらのルーチンは、標準 C/C++ 呼び出しシーケンスに従っています。コンパイラは、必要に応じてそれらを自動的に追加します。ただし、ユーザー・プログラムから直接呼び出されるように意図されていません。

これらの関数のソース・コードは、ソース・ライブラリ rts.src 内に入っています。ソース・コードには、関数の演算を説明するコメントが付いています。算術関数を抽出、検査、変更することはできますが、本章で概要が説明されている呼び出し規則とレジスタ保存規則に必ず従ってください。表 8–6 は、算術演算に使用されるランタイムサポート関数をまとめています。

表 8–6. ランタイムサポート算術関数のまとめ

型	関数	説明
float	_cvtdf (double)	double を float に変換
int	_fixdi (double)	double を signed int に変換
long	_fixdli (double)	double を long に変換
long long	_fixdlli (double)	double を long long に変換
uint	_fixdu (double)	double を unsigned int に変換
ulong	_fixdul (double)	double を unsigned long に変換
ulong long	_fixdull (double)	double を unsigned long long に変換
double	_cvtfd (float)	float を double に変換
int	_fixfi (float)	float を signed int に変換
long	_fixfli (float)	float を long に変換
long long	_fixflli (float)	float を long long に変換
uint	_fixfu (float)	float を unsigned int に変換
ulong	_fixful (float)	float を unsigned long に変換
ulong long	_fixfull (float)	float を unsigned long long に変換

表 8–6. ランタイムサポート算術関数のまとめ（続き）

型	関数	説明
double	_fltid (int)	signed int を double に変換
float	_fltif (int)	signed int を float に変換
double	_fltud (uint)	unsigned int を double に変換
float	_fltuf (uint)	unsigned int を float に変換
double	_fltlid (long)	signed long を double に変換
float	_fltlif (long)	signed long を float に変換
double	_fltuld (ulong)	unsigned long を double に変換
float	_fltulf (ulong)	unsigned long を float に変換
double	_fltllid (long long)	signed long long を double に変換
float	_fltllif (long long)	signed long long を float に変換
double	_fltulld (ulong long)	unsigned long long を double に変換
float	_fltullf (ulong long)	unsigned long long を float に変換
double	_absd (double)	double 絶対値
float	_absf (float)	float 絶対値
long	_labs (long)	long 絶対値
long long	_llabs (long long)	long long 絶対値
double	_negd (double)	double 負数値
float	_negf (float)	float 負数値
long long	_negll (long)	long long 負数値
long long	_llshl (long long)	long long 左シフト
long long	_llshr (long long)	long long 右シフト
ulong long	_llshru (ulong long)	unsigned long long 右シフト
double	_addd (double, double)	double 加算
double	_cmpd (double, double)	double 比較
double	_divd (double, double)	double 除算

表 8–6. ランタイムサポート算術関数のまとめ（続き）

型	関数	説明
double	_mpyd (double, double)	double 乗算
double	_subd (double, double)	double 減算
float	_addf (float, float)	float 加算
float	_cmpf (float, float)	float 比較
float	_divf (float, float)	float 除算
float	_mpyf (float, float)	float 乗算
float	_subf (float, float)	float 減算
int	_divi (int, int)	signed int 除算
int	_remi (int, int)	signed int 剰余
uint	_divu (uint, uint)	unsigned int 除算
uint	_remu (uint, uint)	unsigned int 剰余
long	_divli (long, long)	signed long 除算
long	_remli (long, long)	signed long 剰余
ulong	_divul (ulong, ulong)	unsigned long 除算
ulong	_remul (ulong, ulong)	unsigned long 剰余
long long	_divlli (long long, long long)	signed long long 除算
long long	_remlli (long long, long long)	signed long long 剰余
ulong long	_mpyll(ulong long, ulong long)	unsigned long long 乗算
ulong long	_divull (ulong long, ulong long)	unsigned long long 除算
ulong long	_remull (ulong long, ulong long)	unsigned long long 剰余

8.8 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・ルーチンが c_int00 と呼ばれる関数を使用して行います。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 rts.src の boot.c という名前の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には、c_int00 関数へ分岐するか、呼び出します。しかし、この関数は通常、ハードウェアのリセットにより指定されます。ユーザは、c_int00 関数を別のオブジェクト・モジュールにリンクしなければなりません。これを自動で行うには、-c または -cr リンカ・オプションを使用し、標準ランタイムサポート・ライブラリ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。しかしリンカは、リセット時に c_int00 に自動的に向けるようにハードウェアを設定しません（TMS320C6000 CPU and Instruction Set Reference Guide を参照）。

c_int00 関数は、次の作業を実行して環境を初期化します。

1） .stack と呼ばれるセクションをシステム・スタック用に定義し、初期のスタック・ポインタをセットアップします。

2） 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。変数をロード時に初期化する場合（-cr オプション）は、プログラムが実行される前にローダがこのステップを実行します（これはブート・ルーチンでは実行されません）。詳細については、8.8.1 項「変数の自動初期化」を参照してください。

3） 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、ブート・ルーチンでは C/C++ 環境を正しく初期化するために、上記の操作を必ず実行しなければなりません。

C6000 コード生成ツールに付属の標準ランタイムサポート・ライブラリのリストは、9.1 節「ライブラリ」（9-2 ページ）を参照してください。

8.8.1 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、.cinit と呼ばれる特殊なセクションに、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

ISO C では、明示的に初期化されないグローバル変数と静的変数は、プログラムを実行する前に 0 に設定されます。C6000 C/C++ コンパイラは、初期化されない変数の事前初期化は行いません。初期値 0 をもつ変数は、明示的に初期化する必要があります。

最も簡単な方法は、プログラムの実行を開始する前に、スタンドアロン・シミュレータで -b オプションを使用して .bss セクションをクリアする方法です。

ROM に焼き込むコードで、これらの方法を使用することはできません。

グローバル変数は、実行時またはロード時のどちらかに自動的に初期化されます。詳細は、8.8.4 項「実行時の変数の自動初期化」（8-56 ページ）および 8.8.5 項「ロード時の変数の初期化」（8-57 ページ）を参照してください。また、7.9 節「静的変数とグローバル変数の初期化方法」（7-34 ページ）も参照してください。

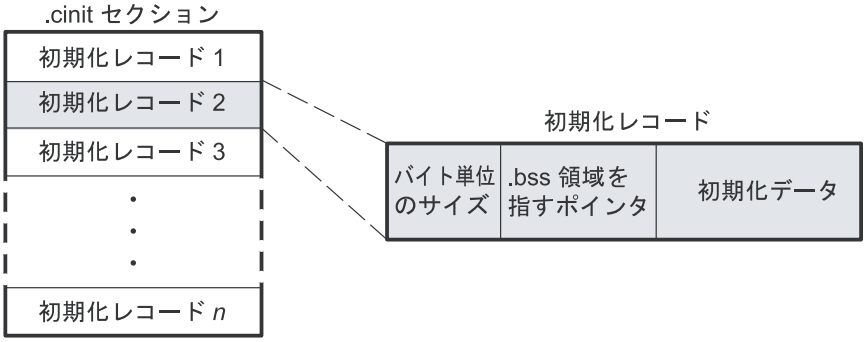
8.8.2 グローバル・コンストラクタ

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に、main () の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・アドレスの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。

8.8.3 初期化テーブル

.cinit セクションのテーブルは、可変サイズの初期化レコードで構成されます。自動初期化が必要な変数は、.cinit セクションにレコードをもっています。図 8–8 は、.cinit セクションの形式と初期化レコードを示しています。

図 8–8. .cinit セクション内の初期化レコードの形式



初期化レコードのフィールドには、次の情報が含まれています。

- 初期化レコードの最初のフィールドは、初期化データのサイズ（バイト数）です。
- 2 番目のフィールドは .bss セクション内の領域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。
- 3 番目のフィールドには、変数を初期化するために .bss セクションにコピーされるデータが入っています。

自動初期化が必要な各変数には、.cinit セクションに初期化レコードがあります。

例 8-9（a）は、C で定義された初期化されるグローバル変数を示しています。例 8-9（b）は、対応する初期化テーブルを示しています。.cinit:c は .cinit セクション内のサブセクションであり、すべてのスカラ・データが入っています。このサブセクションは初期化時に 1 つのレコードとして処理されるので、.cinit セクションの全体のサイズを最小限に抑えます。

例 8–9. 初期化テーブル

(a) C で定義された初期化される変数

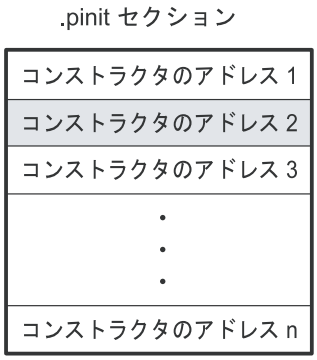
```
int x;
short i = 23;
int *p = &x;
int a[5] = {1,2,3,4,5};
```

(b) (a) で定義された変数の初期化される情報

```
        .global _x
        .bss    _x,4,4

        .sect   ".cinit:c"
        .align  8
        .field          (CIR   $)   8, 32
        .field          _i+0,32
        .field         23,16                    ; _i @ 0

        .sect   ".text"
        .global _i
_i:     .usect  ".bss:c",2,2

        .sect   ".cinit:c"
        .align  4
        .field         _x,32                    ; _p @ 0

        .sect   ".text"
        .global _p
_p:     .usect  ".bss:c",4,4

        .sect   ".cinit"
        .align  8
        .field          IR_1,32
        .field          _a+0,32
        .field         1,32                     ; _a[0] @ 0
        .field         2,32                     ; _a[1] @ 32
        .field         3,32                     ; _a[2] @ 64
        .field         4,32                     ; _a[3] @ 96
        .field         5,32                     ; _a[4] @ 128
IR_1:   .set    20

        .sect   ".text"
        .global _a
        .bss    _a,20,4
;***************************************************************
;* MARK THE END OF THE SCALAR INIT RECORD IN CINIT:C            *
;***************************************************************

CIR:    .sect   ".cinit:c"
```

.cinit セクションに含まれるのは、この形式の初期化テーブルだけでなければなりません。アセンブリ言語モジュールを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。

図 8–9.　.pinit セクション内の初期化レコードの形式



-c や -cr オプションを使用すると、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。

同様に -c または -cr リンカ・オプションにより、リンカはすべての C/C++ モジュールのすべての .pinit セクションをまとめてリンクし、結合した .pinit セクションの最後にヌル・ワードを付けます。ブート・ルーチンは、ヌル・コンストラクタ・アドレスを検出したときに、グローバル・コンストラクタ・テーブルの終わりが分かります。

const で修飾された変数の初期化方法は異なります。7.4.1 項「const キーワード」（7-7 ページ）を参照してください。

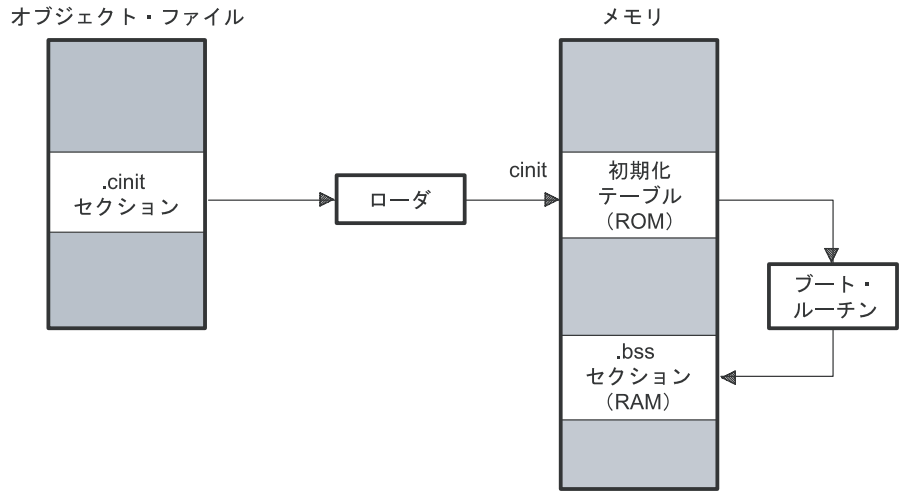
8.8.4 実行時の変数の自動初期化

実行時の変数の自動初期化は、自動初期化のデフォルト方式です。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。

この方式を使用すると、.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内にロードされます。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 にコピーできます。

図 8–10 は実行時の自動初期化を示しています。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 8–10. 実行時の自動初期化



8.8.5 ロード時の変数の初期化

ロード時の変数の初期化を使用すると、ブート時間が短くなり、初期化テーブルにより使用され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。

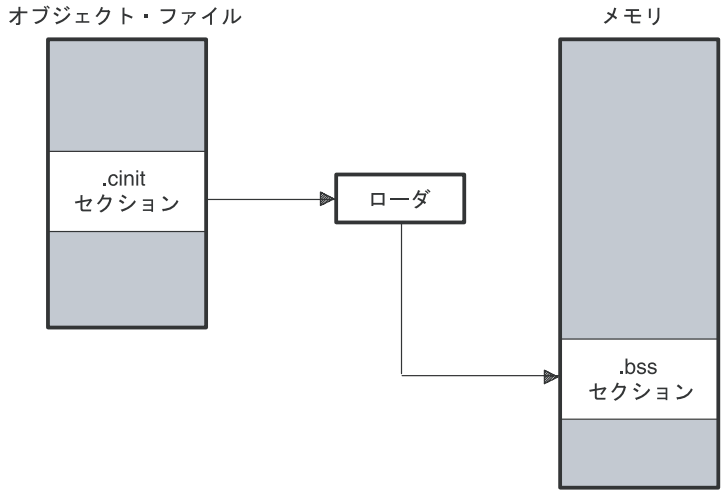
-cr リンカ・オプションを使用すると、リンカは .cinit セクションのヘッダ内に STYP_COPY ビットを設定します。これにより、.cinit セクションを メモリ内にロードしないことをローダに指示します（.cinit セクションはメモリ・マップ内の空間を占有しません）。また、リンカは cinit シンボルを -1 に設定します（通常では cinit は初期化テーブルの先頭を指します）。これにより、初期化テーブルがメモリ内に存在しないことをブート・ルーチンに示します。したがって、ブート時に実行時の初期化は行われません。

ロード時の初期化を使用するには、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 セクションをメモリ内にコピーしないことを認識するように、.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する。
- ❏ 初期化テーブルのフォーマットを理解する。

図 8–11 は、ロード時の変数の初期化を示しています。

図 8–11. ロード時の初期化



第 9 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（入出力、動的メモリ割り当て、文字列操作、三角関数など）の中には、C/C++ 言語自体の一部ではないものがあります。しかし、ISO C 標準は、これらの作業を実行する 1 組のランタイムサポート関数を定義します。TMS320C6000 C/C++ コンパイラは、特殊条件や地域別の項目（居住地の言語、国名、文化圏に応じて決まるプロパティ）を処理する機能を除き、ISO 標準ライブラリに完全に準拠しています。ISO 標準ライブラリを使用すると、関数は、C++ プログラムと整合がとれて移植性も高くなります。

ISO 固有の関数に加えて、TMS320C6000 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と直接 C 言語入出力要求を行うルーチンが組み込まれています。

コード生成ツールに含まれているライブラリ作成ユーティリティを使用すると、カスタマイズされたランタイムサポート・ライブラリを作成できます。このユーティリティの使用方法については、第 10 章「ライブラリ作成ユーティリティ」を参照してください。

項目 **ページ**

9.1	ライブラリ	9-2
9.2	C 入出力関数	9-4
9.3	ヘッダ・ファイル	9-16
9.4	ランタイムサポート関数とマクロのまとめ	9-29
9.5	ランタイム・サポート関数とマクロの解説	9-41

9.1 ライブラリ

以下のライブラリが TMS320C6000 C/C++ コンパイラに添付されています。

- ❏ rts6200.lib、rts6400.lib、および rts6700.lib — リトルエンディアン C/C++ コード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
- ❏ rts6200e.lib、rts6400e.lib、および rts6700e.lib — ビッグエンディアン C/C++ コードで使用する ランタイムサポート・オブジェクト・ライブラリ
- ❏ rts.src — ランタイムサポート・ソース・ライブラリ。ランタイムサポート・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。

ランタイムサポート・ライブラリには、シグナルと地域の項目を含む関数は入っていません。次のものが入っています。

- ❏ ISO C/C++ 標準ライブラリ
- ❏ C 入出力ライブラリ
- ❏ ホスト・オペレーティング・システムに I/O（入出力）を提供する低レベルのサポート関数
- ❏ 組み込み算術ルーチン
- ❏ システム始動ルーチン _c_int00
- ❏ C/C++ からの特定の命令へのアクセスを可能にする関数とマクロ

-mr オプションを使用すると、ニア・コールまたはファー・コールに関して、ランタイムサポート関数の呼び出し方法を制御できます。詳細は、7.4.4.3 項「ランタイム・サポート関数の呼び出し方法の制御（-mr オプション）」（7-12 ページ）を参照してください。

9.1.1 コードとオブジェクト・ライブラリとのリンク方法

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、入出力関数と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最後に指定すべきです。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、リンカが解決できる参照がなくなるまで各ライブラリの検索を強制的に繰り返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のに必要なライブラリ・メンバだけを組み込みます。リンクの詳細は、TMS320C6000 Assembly Language Tools User's Guide を参照してください。

C、C++、C と C++ の混在したプログラムは、同じランタイムサポート・ライブラリを使用できます。C と C++ の両方から呼び出したり参照したりできるランタイム・サポート関数および変数は、同じリンケージをもっています。

9.1.2 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。

```
ar6x x rts.src atoi.c strcpy.c
```

関数を修正するには、先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。たとえば次のとおりです。

```
cl6x -options atoi.c strcpy.c           ;recompile
ar6x r rts6200.lib atoi.obj strcpy.obj  ;rebuild library
```

また、rts6200.lib を作成し直すのではなく、この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、TMS320C6000 Assembly Language Tools User's Guide を参照してください。

9.1.3 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 mk6x 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、以下のコマンドによって、最適化済みのランタイム・サポート・ライブラリを作成できます。

```
mk6x --u -O2 -x rts.src -l rts.lib
```

--u オプションは、ヘッダ・ファイルをソース・アーカイブから抽出する代わりに、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に mk6x ユーティリティに指示します。-O2 オプションを使用すると、こ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両立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 10 章「ライブラリ作成ユーティリティ」を参照してください。

9.2 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。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増えます。

入出力関数を使用するには、C 入出力関数を参照するモジュールごとにヘッダ・ファイル stdio.h、または C++ コードの場合は cstdio を組み込んでください。

たとえば、main.c ファイル内に次の C プログラムが指定されていると想定します。

```
#include <stdio.h>

main()
{
   FILE *fid;

   fid = fopen("myfile","w");
   fprintf(fid,"Hello, world\n");
   fclose(fid);

   printf("Hello again, world\n");
}
```

次のコンパイラ・コマンドを発行すると、コンパイルおよび 'C6200 ランタイム・サポート・リトルエンディアン・ライブラリとのリンクが行われ、ファイル main.out が作成されます。

```
cl6x main.c -z -heap 400 -l rts6200.lib -o main.out
```

main.out を実行すると、

```
Hello, world
```

がファイルに出力され、

```
Hello again, world
```

がホストの stdout ウィンドウに出力されます。

正しく作成されたデバイス・ドライバを使用すると、ライブラリは、ユーザ指定のデバイス上で入出力を実行する機能も提供します。

注 : C 入出力バッファの障害

C 入出力バッファ用の十分な空間がヒープ上にないと、ファイル上でのバッファ付きの操作は失敗します。printf() の呼び出しが原因不明で失敗する場合は、これが理由かもしれません。ヒープのサイズを確認してください。ヒープ・サイズを設定するには、リンク時に -heap オプションを使います（5-6 ページを参照）。

9.2.1 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、高レベル、低レベル、デバイス・レベルの層に分かれています。

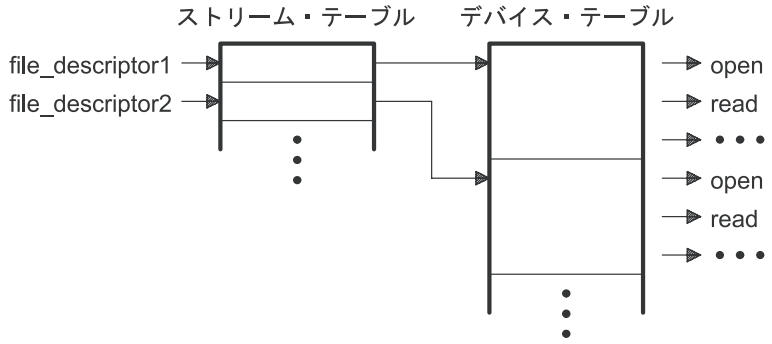
高レベル関数は、ストリーム入出力ルーチン（printf、scanf、fopen、getchar など）の標準 C ライブラリです。これらのルーチンは入出力要求を、低レベル・ルーチンによって処理される 1 つまたは複数の入出力コマンドにマップします。

低レベル・ルーチンは、基本入出力関数（open、read、write、close、lseek、rename、unlink）から構成されています。これらの低レベル・ルーチンは、高レベル関数と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デバイス・レベル・ドライバの間のインターフェイスを提供します。

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ているので、ストリーム上で実行する入出力コマンドで、デバイス・レベルのルーチンが正しく実行されます。

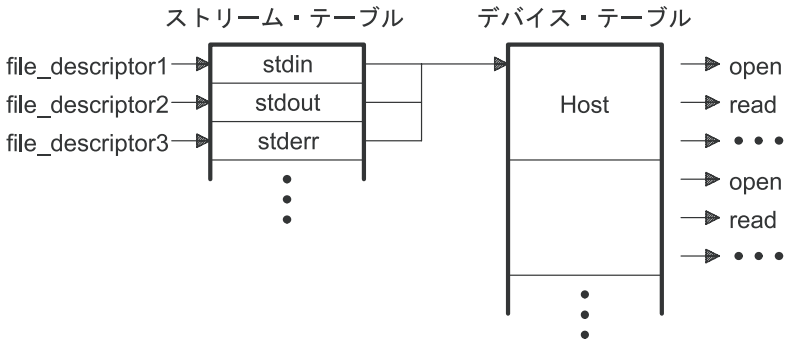
データ構造は、図 9–1 に示すように相互作用します。

図 9–1. 入出力関数での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デバイス・ドライバを指します。

図 9–2. ストリーム・テーブル内の最初の 3 つのストリーム



次のレベルのものは、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的にマップします。ランタイム・サポート・ライブラリには、デバッガが稼働中のホスト上で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を取るようにデバイス・レベルのルーチンを作成する際の仕様を、以下に示します。各関数は、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。

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)(),
        int (*dclose)(),
        int (*dread)(),
        int (*dwrite)(),
        fpos_t (*dlseek)(),
        int (*dunlink)(),
        int (*drename)());
```

定義される場所　rts.src 内の 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、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、形式 ***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。名前は 8 文字に制限されます。
- ❏ *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 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 9.2.1 項「低レベル入出力実装の概要」（9-5 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。TMS320C6000 デバッガが実行されるホストのデバイス・ドライバは、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、次の操作が実行されます。

- ❏ デバイス *mydevice* をデバイス・テーブルに追加する。
- ❏ 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。
- ❏ そのファイルに文字列 *Hello, world* を出力する。
- ❏ 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(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, char *buffer, unsigned count);
extern int my_lseek(int fno, long offset, int origin);
extern int my_unlink(char *path);
extern int my_rename(char *old_name, 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);
}
```

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>

int std::close(int file_descriptor);
```

説明

close 関数は、*file_descriptor* に関連し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している低レベル・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

戻り値は、次のいずれかです。

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ファイル位置標識を *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、デバイス・レベル・ドライバがオープンされたファイルまたはデバイスに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、次のマクロのいずれかによって戻される値にし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) ファイルの終わり

戻り値

戻り値は、次のいずれかです。

#	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(const char *path, unsigned flags, int file_descriptor);
```

C++ の構文

```
#include <cstdio>
#include <file.h>

int std::open(const char *path, unsigned flags, int file_descriptor);
```

説明

open 関数は、入出力用に *path* で指定したデバイスまたはファイルをオープンします。

- *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を操作する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  (0x0200)  /* open with file create */
O_TRUNC  (0x0400)  /* 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、これらのパラメータを無視できる場合もあります。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を阻止します。

- *file_descriptor* は、オープンされたファイルまたはデバイスに関連している低レベル・ルーチンによって割り当てられるストリーム番号です。

  次に使用可能な file_descriptor（3 ～ 20）は、新しくオープンされた各デバイスに割り当てられます。finddevice() 関数を使用してデバイス構造体を戻し、このポインタを使用して同じポインタの _stream 配列を検索できます。
  file_descriptor の番号は _stream 配列のもう一つのメンバです。

戻り値

この関数は、以下の値のどれかを戻します。

≠-1	成功した場合
-1	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* で指定した数の文字を、*file_descriptor* に関連し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している低レベル・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、読み取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(const char *old_name, const char *new_name);
```

C++ の構文

```
#include <cstdio>
#include <file.h>

int std::rename(const char *old_name, const 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(const char *path);
```

C++ の構文

```
#include <cstdio>
#include <file.h>

int std::unlink(const 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。

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, const char *buffer, unsigned count);
```

C++ の構文

```
#include <cstdio>
#include <file.h>

int write(int file_descriptor, const char *buffer, unsigned count);
```

説明

write 関数は、*count* で指定した数の文字を、*buffer* から *file_descriptor* に関連しているデバイスまたはファイルに書き込みます。

- *file_descriptor* は、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。この番号は、オープンされたファイルまたはデバイスに関連しています。
- *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、書き込まれた文字数
-1	失敗した場合

9.2.2 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を、次に示します。

1） 9.2.1 項「低レベル入出力実装の概要」（9-5 ページ）で説明されているデバイス・レベル関数を定義します。

> **注 : 固有の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・ルーチンで使用します。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 にユーザ作成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/cstdio の中にあるマクロ _NDEVICE で定義されます。デバイスに対応する構造体も stdio.h/cstdio の中に定義されていて、次のフィールドから構成されています。

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。詳しくは、9-7 ページの add_device 関数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。*devicename*:*filename* を fopen() への最初の引数として使います。

次のプログラムは、C 入出力用デバイスの追加方法と使用方法を示しています。

```
#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 long my_lseek(int fno, long offset, int origin);
extern int  my_unlink(const char *path);
extern int  my_rename(const char *old_name, 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);
}
```

9.3 ヘッダ・ファイル

各ランタイムサポート関数は、ヘッダ・ファイル内で宣言されます。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

以下は、ISO C ランタイム・サポート関数を宣言するヘッダ・ファイルを示しています。

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

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	cstddef	exception	

さらに、TI の提供する他の関数のために以下のヘッダ・ファイルが含まれています。

c6x.h	cpy_tbl.h	file.h	gsm.h	linkage.h

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 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。

C6000C/C++ コンパイラに組み込まれているヘッダ・ファイルについては、9.3.1 項「診断メッセージ（assert.h/cassert）」（9-17 ページ）から 9.3.21 項「実行時の型情報（typeinfo）」（9-28 ページ）までを参照してください。9.4 節「ランタイムサポート関数とマクロのまとめ」（9-29 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。

9.3.1 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 は NDEBUG の定義は行いません）。assert.h/cassert を組み込むときに NDEBUG をマクロ名としてすでに定義してある場合、assert は無効になり何も実行されません。NDEBUG が定義されない場合、assert は有効となります。

assert.h/cassert ヘッダは NASSERT マクロを参照します（assert.h/cassert は NASSERT の定義は行いません）。assert.h/cassert を組み込むときに NASSERT をマクロ名としてすでに定義してある場合、assert は _nassert と同じ働きをします。_nassert 組み込み関数はコードを生成せず、assert で宣言された式が真であることをコンパイラに伝えます。この関数は、どの最適化が有効であるかのヒントをコンパイラに提供します。NASSERT が定義されない場合、assert は通常有効です。

_nassert 組み込み関数は、ポインタに特定の位置合わせがあることを保証するためにも使用できます。詳細は、8.5.4 項「MUST_ITERATE と _nassert を使用して SIMD を使用可能にし、ループについてのコンパイラの知識を拡張する方法」（8-37 ページ）を参照してください。

assert 関数は、表 9–3（a）（9-30 ページ）に示しています。

9.3.2 文字の判別と変換（ctype.h/cctype）

ctype.h/cctypeヘッダは、文字の型をテストし、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。文字判別関数の名前の形式は、is*xxx*（*isdigit* など）です。

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、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。マクロのほうが、同じ機能をもつ関数より処理速度が高速です。副次作用がある引数が渡される場合は、対応する関数を使用してください。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操作に展開されます。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。

文字の判別と変換関数は、表 9–3（b）（9-30 ページ）に示しています。

9.3.3 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 変数はライブラリ関数のエラーを示します。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）
- ❏ ENOENT - パス・エラーの場合（パスが存在しない）
- ❏ EFPOS - シーク・エラーの場合（ファイル位置エラー）

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。

9.3.4 低レベル入出力関数（file.h）

file.h ヘッダは、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を宣言します。

C6000 に対する入出力の実現方法は、9.2 節「C 入出力関数」（9-4 ページ）を参照してください。

9.3.5 高速マクロ／静的インライン関数（gsm.h）

gsm.h ヘッダ・ファイルには、高速マクロ、および GSM ボコーダの基本的な ETSI 算術演算を定義するための静的インライン関数定義が入っています。

9.3.6 制限値（float.h/cfloat と limits.h/climits）

float.h/cfloat と limits.h/climits ヘッダは、TMS320C6000 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 9–1 と表 9–2 は、これらのマクロ、およびその制限値のリストを示しています。

表 9–1. 整数型の範囲に関する制限値を指定するマクロ（limits.h/climits）

マクロ	値	説明
CHAR_BIT	8	type char のビット数
SCHAR_MIN	-128	signed char の最小値
SCHAR_MAX	127	signed char の最大値
UCHAR_MAX	255	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	2147483647	int の最大値
UINT_MAX	4294967295	unsigned int の最大値
LONG_MIN	(-LONG_MAX - 1)	long int の最小値
LONG_MAX	549755813887	long int の最大値
ULONG_MAX	1099511627775	unsigned long int の最大値
LLONG_MIN	(-LLONG_MAX - 1)	long long int の最小値
LLONG_MAX	9 223 372 036 854 775 807	long long int の最大値
ULLONG_MAX	18 446 744 073 709 551 615	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 9–2. 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/cfloat）

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 15 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 53 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 -1021 -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
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	128 1024 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、最大の負の整数
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1.19209290e-07 2.22044605e-16 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 2.22507386e-308 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 1.79769313e+308 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 -307 -307	10 を整数乗にした値が正規化した 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
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	38 308 308	10 を整数乗にした値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

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/cfloat ヘッダ・ファイルを参照してください。

9.3.7 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、複数のマシンを通じて一貫し、オペレーティング・システムおよび他の実装の特徴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 は様々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 に提供された型として、自由に整数型の typedef を実行できます。一貫して inttypes.h ヘッダを使用することにより、プラットフォームを超えたユーザ・プログラムの移植性が高くなります。

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 型の値を表すのに充分な大きさの int 型です。
- *uintmax_t* 型 - unsigned int 型の値を表すのに充分な大きさの 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。例えば、符号付き整数の 3 つの fprintf マクロは、PRId32、PRIdLEAST32、および PRIdFAST32 です。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* および *strtoumax* 関数。文字列の最初の部分は、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。

9.3.8 代替表現（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	!		

9.3.9 near または far としての関数呼び出し（linkage.h）

linkage.h ヘッダは、ランタイム・サポート・ライブラリのコードおよびデータにアクセスする方法を決定するマクロを宣言します。_FAR_RTS マクロの値に応じて、ランタイム・サポート関数の呼び出しをユーザ・デフォルトの near、または far に強制的に設定するために、_CODE_ACCESS マクロが設定されます。_FAR_RTS マクロは、-mr コンパイラ・オプションの使用に従って設定されます。

_DATA_ACCESS マクロは、常に far に設定されます。_IDECL マクロはインライン関数の宣言方法を決定します。

関数またはデータを定義するすべてのヘッダ・ファイルは、#include <linkage.h> を宣言します。関数は _CODE_ACCESS を使用して変更されます。たとえば次のとおりです。

```
extern _CODE_ACCESS void    exit(int _status);
```

データは _DATA_ACCESS を使用して変更されます。たとえば次のとおりです。

```
extern _DATA_ACCESS unsigned char _ctypes_[];
```

9.3.10 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定義します。これらの関数は、表 9–3（c）（9-31 ページ）に掲載されています。この算術関数は double 型または float 型の引数を要求し、それぞれ double 型または float 型の値を戻します。指定されている場合を除き、すべての三角関数はラジアンで表記される角度を使用します。

math.h/cmath ヘッダは、HUGE_VAL という名前の 1 つのマクロも定義します。算術関数は、このマクロを使用し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、代わりに HUGE_VAL を戻します。

math.h/cmath ヘッダには、ソース・ファイルで _TI_ENHANCED_MATH_H シンボルを定義した場合に使用できる拡張算術関数が含まれています。
_TI_ENHANCED_MATH_H シンボルを定義した場合は、HUGE_VALF シンボルが可視になります。HUGE_VALF は、HUGE_VAL の float 版です。

すべての math.h/cmath 関数の場合、領域と範囲のエラーは、必要に応じて errno を EDOM または ERANGE に設定して処理されます。関数の入出力値は、最も近い有効値に丸められます。

9.3.11 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とマクロを定義し、関数を宣言します。次のものが含まれます。

- 配列型 *jmp_buf* は、呼び出し環境の復元に必要な情報の保存に適しています。
- *setjmp* マクロは、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。
- *longjmp* 関数は、jmp_buf 引数を使用してプログラム環境を復元するために使用します。

非ローカル・ジャンプのマクロと関数は、表 9–3 (d)（9-34 ページ）に掲載されています。

9.3.12 可変引数（stdarg.h/cstdarg）

関数の中には、型が異な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呼びます。stdarg.h/cstdarg ヘッダでは、可変引数関数の使用に役立つマクロと 1 つの型を宣言しています。

- マクロは *va_start*、*va_arg*、および *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、および 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。この時、関数は、実際に渡された引数の数と型を認識します。引数が正しく処理されるために、可変引数関数に対する呼び出しに、関数のプロトタイプに対する可視性があることを確認する必要があります。可変引数関数は、表 9–3（e）（9-34 ページ）に掲載されています。

9.3.13 標準定義（sttdef.h/cstddef）

sttdef.h/cstddef ヘッダは、これらの型とマクロを定義します。

- *ptrdiff_t* 型は、2 つのポインタの減算の結果のデータ型の符号付き整数型です。
- *size_t* 型は、*sizeof* 演算子のデータ型である符号なし整数型です。
- *NULL* マクロは、ヌル・ポインタ定数（0）に展開されます。
- *offsetof(type, identifier)* マクロは、*size_t* 型をもつ整数に展開されます。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（バイト単位）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。

9.3.14 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た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。型は以下のカテゴリで定義されます。

- 厳密な幅を持つ整数型 — 符号付き整数型の int*N*_t と符号なし整数型の uint*N*_t があります。
- 最小の幅を持つ整数型 — 符号付き整数型の int_least*N*_t と符号なし整数型の unit_least*N*_t があります。
- 最も速度の遅い最小の幅を持つ整数型 — 符号付き整数型の int_fast*N*_t と符号なし整数型の uint_fast*N*_t があります。
- ポインタを格納可能な整数型 — 符号付き整数型の *intptr_t* と符号なし整数型の *uintptr_t* があります。
- 最も大きな幅を持つ整数型 — 符号付き整数型の *intmax_t* と符号なし整数型の *uintmax_t* があります。

stdint.h の提供する各符号付き型は、上限または下限を指定するマクロです。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 を持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 を持つ符号なし整数定数に展開されます。

この例は、最低 16 ビットで保持できる最小の整数を使用する、stdint.h で定義され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。

9.3.15 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 つの構造体および複数の関数を定義します。型と構造体は次のとおりです。

- *size_t* 型は、*sizeof* 演算子のデータ型である符号なし整数型です。本来は stddef.h/cstddef に定義されていました。
- *fpos_tI* 型は、ファイル内のすべての位置を独自に指定できる符号付き整数型です。
- *FILE* 型は、ストリームを制御するために必要なすべての情報を記録する構造体型です。

マクロは、次のとおりです。

- *NULL* マクロは、ヌル・ポインタ定数（0）に展開されます。本来は stddef.h/cstddef に定義されていました。すでに定義されている場合、再定義は行われません。
- *BUFSIZ* マクロは、setbuf() が使用するバッファのサイズに展開されます。
- *EOF* マクロは、ファイル終わり位置を示すマーカです。
- *FOPEN_MAX* マクロは、一度に開くことができるファイルの最大数に展開されます。
- *FILENAME_MAX* マクロは、最長ファイル名の文字数単位の長さに展開されます。
- *L_tmpnam* マクロは、tmpnam() が生成できるファイル名の最長文字列に展開されます。
- *SEEK_CUR*、*SEEK_SET*、および *SEEK_END* マクロは、ファイルの位置（それぞれ、現在位置、ファイル開始位置、ファイル終わり位置）を示すために展開されます。
- *TMP_MAX* マクロは、tmpnam() が生成できる一意的なファイル名の最大数に展開されます。
- *stderr*、*stdin*、および *stdout* マクロは、それぞれ、標準エラー、入力、および出力ファイルへのポインタです。

入出力関数は、表 9–3（f）（9-34 ページ）に示しています。

9.3.16 汎用ユーティリティ（stdlib.h/cstdlib）

stdlib.h/cstdlib ヘッダは 1 つのマクロと 2 つの型を定義し、多くの関数を宣言します。マクロは RAND_MAX と呼ばれ、rand() 関数によって戻される最大の値を戻します。型は次のとおりです。

- *div_t* 型は div 関数が戻す値の構造体の型です。
- *ldiv_t* 型は ldiv 関数が戻す値の構造体の型です。

関数は、次のとおりです。

- 文字列変換関数は、文字列を数値表現に変換します。
- 検索およびソート関数は、配列の検索とソートを行います。
- シーケンス生成関数は、疑似乱数シーケンスを生成し、シーケンスの開始点を選択できます。
- プログラム出口関数は、プログラムを正常終了または異常終了させます。
- 整数算術は、C 言語の標準部分としては提供されていません。

汎用ユーティリティ関数は、表 9–3（g）（9-37 ページ）に示しています。

9.3.17 文字列関数（string.h/cstring）

string.h/cstring ヘッダは、文字配列（文字列）に関して次の作業を実行する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 では、すべての文字列の最後は 0（ヌル）文字です。str*xxx* という名の文字列関数は、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の形式は mem*xxx* 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。文字列関数は、表 9–3（h）（9-38 ページ）に示しています。

9.3.18 時間関数（time.h/ctime）

time.h/ctime ヘッダは 1 つのマクロと複数の型を定義し、日時を操作する関数を宣言します。時刻は次のように表されます。

- *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。このような表しかたの時間を詳細時刻と呼びます。この構造体には、次のメンバがあります。

```
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 (0 and up)    */
int   tm_wday;      /* days since Saturday (0-6)      */
int   tm_yday;      /* days since January 1 (0-365)   */
int   tm_isdst;     /* daylight savings time flag     */
```

時刻は、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。

時間関数とマクロは、表 9–3（i）（9-40 ページ）に掲載されています。

地方時は、地域や季節の変動に合わせて調整できます。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 _tz というこの型の変数を宣言します。実行時に、または tmzone.c を編集して初期設定を変更することで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。

すべての time.h/ctime 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 形式）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。

注 : ユーザ固有の clock 関数の記述

clock 関数は、スタンドアロン・シミュレータ（load6x）で動作します。load6x 環境で使用する場合、clock() はサイクルの正確なカウントを戻します。HLL デバッガで使用する場合、clock 関数は -1 を戻します。

ホスト固有の clock 関数を作成できます。また、clock() が戻す値（クロックの目盛り値）を CLOCKS_PER_SEC で割って値を秒数で表すことができるようにするには、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。

9.3.19 例外処理（exception と stdexcept）

例外処理はサポートされていません。exception と stdexcept のインクルード・ファイル（C++ 専用）は空です。

9.3.20 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。

9.3.21 実行時の型情報（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。

9.4 ランタイムサポート関数とマクロのまとめ

表 9–3 は、TMS320C6000 ANSI/ISO C/C++ コンパイラが提供するランタイムサポート・ヘッダ・ファイルを要約して示します（アルファベット順）。説明されている関数の大部分は、ISO 規格に従ってその規格内に定められているとおりに動作します。

表 9–3 に掲載されている関数とマクロの詳細は、9.5 節「ランタイム・サポート関数とマクロの解説」(9-41 ページ）を参照してください。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。

肩付きの数字は、指数を示すために次の説明で使用されています。たとえば、x^y は x の y 乗に相当します。

表 9–3. 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 expr);	診断メッセージをプログラムに挿入します。	9-46

（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。	9-69
int **isalpha**(int c);	英字 ASCII 文字かどうか、c をテストします。	9-69
int **isascii**(int c);	ASCII 文字かどうか、c をテストします。	9-69
int **iscntrl**(int c);	制御文字かどうか、c をテストします。	9-69
int **isdigit**(int c);	数字かどうか、c をテストします。	9-69
int **isgraph**(int c);	スペース以外の印字文字かどうか、c をテストします。	9-69
int **islower**(int c);	ASCII の英小文字かどうか、c をテストします。	9-69
int **isprint**(int c);	印刷可能な ASCII 文字かどうか、c をテストします。	9-69
int **ispunct**(int c);	ASCII の句読点文字かどうか、c をテストします。	9-69
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。	9-69
int **isupper**(int c);	ASCII の英大文字かどうか、c をテストします。	9-69
int **isxdigit**(int c);	16 進数字かどうか、c をテストします。	9-69
int **toascii**(int c);	c を有効な ASCII 値にマスクします。	9-107
int **tolower**(int char c);	c が大文字の場合は、小文字に変換します。	9-108
int **toupper**(int char c);	c が小文字の場合は、大文字に変換します。	9-108

注： ctype.h/cctype 内の関数は、-pi オプションが使用されている場合を除き、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。	9-42
float **acosf**(float x);	x のアーク・コサインを戻します。	9-42
double **acosh**(double x);	x のハイパボリック・アーク・コサインを戻します。†	9-42
float **acoshf**(float x);	x のハイパボリック・アーク・コサインを戻します。†	9-42
double **acot**(double x);	x のアーク・コタンジェントを戻します。†	9-43
double **acot2**(double x, double y);-	x/y のアーク・コタンジェントを戻します。†	9-43
float **acot2f**(float x, float y);	x/y のアーク・コタンジェントを戻します。†	9-43
float **acotf**(float x);	x のアーク・コタンジェントを戻します。†	9-43
double **acoth**(double x);	x のハイパボリック・アーク・コタンジェント を戻します。†	9-44
float **acothf**(float x);	x のハイパボリック・アーク・コタンジェント を戻します。†	9-44
double **asin**(double x);	x のアーク・サインを戻します。	9-45
float **acinf**(float x);	x のアーク・サインを戻します。	9-45
double **asinh**(double x);	x のハイパボリック・アーク・サインを戻します。†	9-45
float **asinhf**(float x);	x のハイパボリック・アーク・サインを戻します。†	9-45
double **atan**(double x);	x のアーク・タンジェントを戻します。	9-47
double **atan2**(double y, double x);	y/x のアーク・タンジェントを戻します。	9-47
float **atan2f**(float y, float x);	y/x のアーク・タンジェントを戻します。	9-47
float **atanf**(float x);	x のアーク・タンジェントを戻します。	9-47
double **atanh**(double x);	x のハイパボリック・アーク・タンジェ ントを戻します。†	9-48
float **atanhf**(float x);	x のハイパボリック・アーク・タンジェ ントを戻します。†	9-48
double **ceil**(double x);	x 以上の最小の整数を戻します。-pi が使用されている場合を除き、インライン展開します。	9-51
float **ceilf**(float x);	x 以上の最小の整数を戻します。-pi が使用されている場合を除き、インライン展開します。	9-51
double **cos**(double x);	x のコサインを戻します。	9-53
float **cosf**(float x);	x のコサインを戻します。	9-53
double **cosh**(double x);	x のハイパボリック・コサインを戻します。	9-53
float **coshf**(float x);	x のハイパボリック・コサインを戻します。	9-53
double **cot**(double x);	x のコタンジェントを戻します。†	9-54

† 拡張算術関数です。この関数のアクセス方法は、9.3.10 項「浮動小数点算術（math.h/cmath）」（9-22 ページ）を参照してください。

（c）浮動小数点算術関数（math.h/cmath）（続き）

関数	説明	ページ
float **cotf**(float x);	x のコタンジェントを戻します。†	9-54
double **coth**(double x);	x のハイパボリック・コタンジェントを戻します。†	9-54
float **cothf**(float x);	x のハイパボリック・コタンジェントを戻します。†	9-54
double **exp**(double x);	e^x を戻します。	9-57
double **exp10**(double x);	10.0^x を戻します。†	9-58
float **exp10f**(float x);	10.0^x を戻します。†	9-58
double **exp2**(double x);	2.0^x を戻します。†	9-58
float **exp2f**(float x);	2.0^x を戻します。†	9-58
float **expf**(float x);	e^x を戻します。	9-57
double **fabs**(double x);	x の絶対値を戻します。	9-59
float **fabsf**(float x);	x の絶対値を戻します。	9-59
double **floor**(double x);	x 以下の最大の整数を戻します。-pi が使用されている場合を除き、インライン展開します。	9-61
float **floorf**(float x);	x 以下の最大の整数を戻します。-pi が使用されている場合を除き、インライン展開します。	9-61
double **fmod**(double x, double y);	x/y の正確な浮動小数点数の剰余を戻します。	9-62
float **fmodf**(float x, float y);	x/y の正確な浮動小数点数の剰余を戻します。	9-62
double **frexp**(double value, int *exp);	$.5 \le \|f\| < 1$ であり、かつ、値が $f \times 2^{exp}$ に等しくなるように f と exp を戻します。	9-65
float **frexpf**(float value, int *exp);	$.5 \le \|f\| < 1$ であり、かつ、値が $f \times 2^{exp}$ に等しくなるように f と exp を戻します。	9-65
double **ldexp**(double x, int exp);	$x \times 2^{exp}$ を戻します。	9-71
float **ldexpf**(float x, int exp);	$x \times 2^{exp}$ を戻します。	9-71
double **log**(double x);	x の自然対数を戻します。	9-72
double **log10**(double x);	底が 10 の x の対数を戻します。	9-72
float **log10f**(float x);	底が 10 の x の対数を戻します。	9-72
double **log2**(double x);	底が 2 の x の対数を戻します。†	9-73
float **log2f**(float x);	底が 2 の x の対数を戻します。†	9-73
float **logf**(float x);	x の自然対数を戻します。	9-72

† 拡張算術関数です。この関数のアクセス方法は、9.3.10 項「浮動小数点算術（math.h/cmath）」（9-22 ページ）を参照してください。

（c）浮動小数点算術関数（math.h/cmath）（続き）

関数	説明	ページ
double **modf**(double value, double *ip);	値を、符号付き整数と符号付き小数に分けます。	9-80
float **modff**(float value, float *ip);	値を、符号付き整数と符号付き小数に分けます。	9-80
double **pow**(double x, double y);	x^y を戻します。	9-81
float **powf**(float x, float y);	x^y を戻します。	9-81
double **powi**(double x, int y);	x^i を戻します。†	9-81
float **powif**(float x, int y);	x^i を戻します。†	9-81
double **round**(double x);	最も近い整数に丸められた x を戻します。†	9-86
float **roundf**(float x);	最も近い整数に丸められた x を戻します。†	9-86
double **rsqrt**(double x);	x の逆平方根を戻します。†	9-87
float **rsqrtf**(float x);	x の逆平方根を戻します。†	9-87
double **sin**(double x);	x のサインを戻します。	9-90
float **sinf**(float x);	x のサインを戻します。	9-90
double **sinh**(double x);	x のハイパボリック・サインを戻します。	9-90
float **sinhf**(float x);	x のハイパボリック・サインを戻します。	9-90
double **sqrt**(double x);	x の負でない平方根を戻します。	9-91
float **sqrtf**(float x);	x の負でない平方根を戻します。	9-91
double **tan**(double x);	x のタンジェントを戻します。	9-105
float **tanf**(float x);	x のタンジェントを戻します。	9-105
double **tanh**(double x);	x のハイパボリック・タンジェントを戻します。	9-106
float **tanhf**(float x);	x のハイパボリック・タンジェントを戻します。	9-106
double **trunc**(double x);	0 に切り捨てられた x を戻します。†	9-108
float **truncf**(float x);	0 に切り捨てられた x を戻します。†	9-108

† 拡張算術関数です。この関数のアクセス方法は、9.3.10 項「浮動小数点算術（math.h/cmath）」（9-22 ページ）を参照してください。

（d）非ローカル・ジャンプ・マクロと関数（setjmp.h/csetjmp）

関数またはマクロ	説明	ページ
int **setjmp**(jmp_buf env);	longjmp が使用するために呼び出し環境を保管します。これはマクロです。	9-88
void **longjmp**(jmp_buf env, int _val);	jmp_buf 引数を使用して、以前に保管された環境を復元します。	9-88

（e）可変引数マクロ（stdarg.h/cstdarg）

マクロ	説明	ページ
type **va_arg**(va_list, 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。	9-109
void **va_end**(va_list);	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。	9-109
void **va_start**(va_list, parmN);	可変引数リスト内の第 1 オペランドを指すよう ap を初期化します。	9-109

（f） C 入出力関数（stdio.h/cstdio）

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。	9-7
void **clearerr**(FILE *_fp);	_fp が指すストリームの EOF 標識とエラー標識を消去します。	9-52
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。	9-59
int **feof**(FILE *_fp);	_fp が指すストリームの EOF 標識をテストします。	9-59
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。	9-60
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。	9-60
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。	9-60
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。	9-60
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。	9-61

(f) C 入出力関数（stdio.h/cstdio）（続き）

関数	説明	ページ
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。	9-62
int **fprintf**(FILE *_fp, const char *_format, ...);	_fp が指すストリームに書き込みます。	9-63
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。	9-63
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。	9-63
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。	9-64
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。	9-65
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。	9-66
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。	9-66
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。	9-66
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。	9-67
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。	9-67
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。	9-67
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。	9-68
char ***gets**(char *_ptr);	stdin を入力ストリームとして使用して、fgets() と同じ機能を実行します。	9-68
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。	9-80
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。	9-82
int **putc**(int _x, FILE *_fp);	fputc() と同じ機能を実行するマクロです。	9-82

(f) C 入出力関数（stdio.h/cstdio）（続き）

関数	説明	ページ
int **putchar**(int _x);	fputc() を呼び出し、stout を出力ストリームとして使用するマクロです。	9-82
int **puts**(const char *_ptr);	_ptr が指す文字列を stdout に書き込みます。	9-83
int **remove**(const char *_file);	_file が指す名前をもつファイルを、その名前では使用できないようにします。	9-85
int **rename**(const char *_old, const char *_new);	_old が指す名前をもつファイルを、_new が指す名前で認識されるようにします。	9-85
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。	9-86
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。	9-87
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。	9-87
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。	9-89
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。	9-91
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。	9-91
FILE ***tmpfile**(void);	一時ファイルを作成します。	9-107
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。	9-107
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。	9-109
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。	9-110
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。	9-110
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。	9-111

（g）汎用関数（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。	9-41
int **abs**(int i);	値の絶対値を戻します。インライン展開します。	9-41
int **atexit**(void (*fun)(void));	プログラム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。	9-48
double **atof**(const char *st);	文字列を浮動小数点値に変換します。-pi が使用されている場合を除き、インライン展開します。	9-49
int **atoi**(const char *st);	文字列を整数に変換します。	9-49
long **atol**(const char *st);	文字列を長整数値に変換します。-pi が使用されている場合を除き、インライン展開します。	9-49
long long **atoll**(const char *st);	文字列を倍長整数値に変換します。-pi が使用されている場合を除き、インライン展開します。	9-49
void ***bsearch**(const void *key, 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。	9-50
void ***calloc**(size_t num, size_t size);	size バイトごとに num 個のオブジェクトのメモリの割り当てとクリアを行います。	9-51
div_t **div**(int numer, int denom);	numer を denom で除算し、商と剰余を生成します。	9-56
void **exit**(int status);	プログラムを正常終了させます。	9-57
void **free**(void *packet);	malloc、calloc、または realloc によって割り当てられたメモリ空間を解放します。	9-64
char ***getenv**(const char *_string)	_string に関連した変数の環境情報を戻します。	9-68
long **labs**(long i);	i の絶対値を戻します。インライン展開します。	9-41
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。	9-41
ldiv_t **ldiv**(long numer, long denom);	numer を denom で除算します。	9-56
lldiv_t **lldiv**(long long numer, long long denom);	numer を denom で除算します。	9-56
int **lltoa**(long long val, char *buffer);	val を等価の文字列に変換します。	9-73
int **ltoa**(long val, char *buffer);	val を等価の文字列に変換します。	9-74
void ***malloc**(size_t size);	size バイトのオブジェクトにメモリを割り当てます。	9-74
void ***memalign**(size_t alignment, size_t size);	位置合わせバイト境界に位置合わせされた size バイトのオブジェクトに、メモリを割り当てます。	9-75

（g）汎用関数（stdlib.h/cstdlib）（続き）

関数	説明	ページ
void **minit**(void);	malloc、calloc、または realloc によって以前に割り当てられたメモリすべてをリセットします。	9-78
void **qsort**(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const 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。	9-83
int **rand**(void);	0 ～ RAND_MAX の範囲の整数の疑似ランダム列を戻します。	9-84
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。	9-84
void **srand**(unsigned int seed);	乱数発生ルーチンをリセットします。	9-84
double **strtod**(const char *st, char **endptr);	文字列を浮動小数点値に変換します。	9-103
long **strtol**(const char *st, char **endptr, int base);	文字列を長整数に変換します。	9-103
long long **strtoll**(const char *st, char **endptr, int base);	文字列を倍長整数に変換します。	9-103
unsigned long **strtoul**(const char *st, char **endptr**,** int base**);**	文字列を符号なし長整数に変換します。	9-103
unsigned long long **strtoull**(const char *st, char **endptr**,** int base**);**	文字列を符号なし倍長整数に変換します。	9-103

（h）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 の最初の出現を検出します。-pi が使用されている場合を除き、インライン展開します。	9-75
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。-pi が使用されている場合を除き、インライン展開します。	9-76
void ***memcpy**(void *s1, const void *s2, register size_t n);	n 個の文字を s1 から s2 にコピーします。	9-76
void ***memmove**(void *s1, const void *s2, size_t n);	n 個の文字を s1 から s2 に移動します。	9-77
void ***memset**(void *mem, int ch, size_t length);	mem の先頭の length 個の文字に ch の値をコピーします。-pi が使用されている場合を除き、インライン展開します。	9-77
char ***strcat**(char *string1, const char *string2);	string2 を string1 の末尾に付加します。	9-92
char ***strchr**(const char *string, int c);	string を検索して、文字 c の最初の出現を検出します。-x が使用されている場合は、インライン展開します。	9-93

（h）文字列関数（string.h/cstring）（続き）

関数	説明	ページ
int **strcmp**(register const char *string1, register const char *s2);	文字列を比較し、以下の値のどれかを戻します。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 が使用されている場合は、インライン展開します。	9-93
int **strcoll**(const char *string1, const char *string2);	文字列を比較し、以下の値のどれかを戻します。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。	9-93
char ***strcpy**(register char *dest, register const char *src);	文字列 src を dest にコピーします。-pi が使用されている場合を除き、インライン展開します。	9-94
size_t **strcspn**(register const char *string, const char *chs);	全体が chs 内にない文字から構成される string の先頭部分の長さを戻します。	9-95
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。	9-95
size_t **strlen**(const char *string);	文字列の長さを戻します。	9-97
char ***strncat**(char *dest, const char *src, register size_t n);	最高で n 個の文字を src から dest に付加します。	9-98
int **strncmp**(const char *string1, const char *string2, size_t n);	2 つの文字列の最高 n 個の文字を比較します。-pi が使用されている場合を除き、インライン展開します。	9-99
char ***strncpy**(register char *dest, register const char *src, register size_t n);	最高で n 個の文字を src から dest にコピーします。-pi が使用されている場合を除き、インライン展開します。	9-100
char ***strpbrk**(const char *string, const char *chs);	string の中で chs のいずれかの文字の最初の出現を検出します。	9-101
char ***strrchr**(const char *string, int c);	string の中の文字 c の最後の出現を検出します。-pi が使用されている場合を除き、インライン展開します。	9-101
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。	9-102
char ***strstr**(register const char *string1, const char *string2);	string1 を検索して string2 の最初の出現を検出します。	9-102
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。	9-104
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。	9-105

(i) 時間サポート関数（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。	9-44
clock_t **clock**(void);	使用されたプロセッサ時間を判定します。	9-52
char ***ctime**(const time_t *timer);	カレンダ時を地方時に変換します。	9-55
double **difftime**(time_t time1, time_t time0);	2 つのカレンダ時の差を戻します。	9-55
struct tm ***gmtime**(const time_t *timer);	地方時をグリニッジ標準時に変換します。	9-69
struct tm ***localtime**(const time_t *timer);	time_t の値を詳細時刻に変換します。	9-71
time_t **mktime**(struct tm *tptr);	詳細時刻を time_t の値に変換します。	9-79
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。	9-96
time_t **time**(time_t *timer);	現在のカレンダ時を戻します。	9-106

9.5 ランタイム・サポート関数とマクロの解説

この節では、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 コードでのみ表示されています。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 名前空間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 内の 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);
 }
```

9-57 ページの exit 関数を参照してください。

abs/labs/llabs 絶対値

C の構文

```
#include   <stdlib.h>

int abs(int i);
long labs(long i);
long long llabs(long long i);
```

C++ の構文

```
#include   <cstdlib>

int std::abs(int i);
long std::labs(long i);
long long std::llabs(long long i);
```

定義される場所 rts.src 内の 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 i の絶対値を戻します。
- ❏ llabs 関数は倍長整数 i の絶対値を戻します。

acos/acosf　アーク・コサイン

C の構文

```
#include   <math.h>

double acos(double x);
float acosf(float x);
```

C++ の構文

```
#include   <cmath>

double std::acos(double x);
float std::acosf(float x);
```

定義される場所 rts.src 内の acos.c と acosf.c

説明 acos および acosf 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 Pi_Over_2;

Pi_Over_2 = acos(-1.0)  /* Pi  */
          + acos( 0.0)  /* Pi/2 */
          + acos( 1.0); /* 0.0  */
```

acosh/acoshf　ハイパボリック・アーク・コサイン

C の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double acosh(double x);
float acoshf(float x);
```

C++ の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::acosh(double x);
float std::acoshf(float x);
```

定義される場所 rts.src 内の acosh.c と acoshf.c

説明 acosh および acoshf 関数は、浮動小数点引数 x のハイパボリック・アーク・コサインを戻します。x の 範囲は [1, 無限大] とします。戻り値は ≥ 0.0 です。

acot/acotf 極アーク・コタンジェント

C の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double acot(double x);
float acotf(float x);
```

C++ の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::acot(double x);
float std::acotf(float x);
```

定義される場所 rts.src 内の acot.c と acotf.c

説明 acot および acotf 関数は、浮動小数点引数 x のアーク・コタンジェントを戻します。戻り値は範囲 [0, π/2] のラジアンの角度です。

例

```
double realval, radians;

realval = 0.0;
radians = acotf(realval);     /* return value = Pi/2 */
```

acot2/acot2f デカルト・アーク・コタンジェント

C の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double acot2(double x, double y);-
float acot2f(float x, float y);
```

C++ の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::acot2(double x, double y);
float std::acot2f(float x, float y);
```

定義される場所 rts.src 内の acot2.c と acot2f.c

説明 acot2 および acot2f 関数は、x/y 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。

acoth/acothf　ハイパボリック・アーク・コタンジェント

C の構文

```
#define _TI_ENHANCED_MATH_H 1
#include  <math.h>

double acoth(double x);
float acothf(float x);
```

C++ の構文

```
#define _TI_ENHANCED_MATH_H 1
#include  <cmath>

double std::acoth(double x);
float std::acothf(float x);
```

定義される場所　rts.src 内の acoth.c と acothf.c

説明　acoth および acothf 関数は、浮動小数点引数 x のハイパボリック・アーク・コタンジェントを戻します。x の大きさは ≥ 0 でなければなりません。

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 内の 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、9.3.18 項「時間関数（time.h/ctime）」（9-27 ページ）を参照してください。

asin/asinf　アーク・サイン

C の構文

```
#include  <math.h>

double asin(double x);
float asinf(float x);
```

C++ の構文

```
#include  <cmath>

double std::asin(double x);
float std::asinf(float x);
```

定義される場所 rts.src 内の asin.c と asinf.c

説明 asin および asinf 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 π/2 */
```

asinh/asinhf　ハイパボリック・アーク・サイン

C の構文

```
#define _TI_ENHANCED_MATH_H 1
#include  <math.h>

double asinh(double x);
float asinhf(float x);
```

C++ の構文

```
#define _TI_ENHANCED_MATH_H 1
#include  <cmath>

double std::asinh(double x);
float std::asinhf(float x);
```

定義される場所 rts.src 内の asinh.c と asinhf.c

説明 asinh および asinhf 関数は、浮動小数点数 x のハイパボリック・アーク・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。

assert　診断情報挿入マクロ

C の構文

#include <assert.h>

void **assert**(int expr);

C++ の構文

#include <cassert>

void **std::assert**(int expr);

定義される場所　assert.h/cassert（マクロとして定義）

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のときに便利です。

- ❏ 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デバイスに書き込み、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 マクロを定義するヘッダ・ファイルは、別のマクロである 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)
```

assert マクロを定義するヘッダ・ファイルは、別のマクロである NASSERT 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 NASSERT をマクロ名として定義した場合、assert マクロは、あたかも _nassert 組み込み関数を呼び出すマクロであるかのように動作します。

例

この例では、整数 i を別の整数 j で割ります。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;
```

atan/atanf　極アーク・タンジェント

C の構文

#include <math.h>

double **atan**(double x);
float **atanf**(float x);

C++ の構文

#include <cmath>

double **std::atan**(double x);
float **std::atanf**(float x);

定義される場所 rts.src 内の atan.c と atanf.c

説明 atan および atanf 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);     /* radians = 0.0 */
```

atan2/atan2f　デカルト・アーク・タンジェント

C の構文

#include <math.h>

double **atan2**(double y, double x);
float **atan2f**(float y, float x);

C++ の構文

#include <cmath>

double **std::atan2**(double y, double x);
float **std::atan2f**(float y, float x);

定義される場所 rts.src 内の atan2.c と atan2f.c

説明 atan2 および atan2f 関数は、x/y 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 = 0.0, rvalv = 1.0, radians;

radians = atan2(rvalu, rvalv);   /* radians = 0.0 */
```

atanh/atanhf ハイパボリック・アーク・タンジェント

C の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double atanh(double y, double x);
float atanhf(float x);
```

C++ の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::atanh(double y, double x);
float std::atanhf(float x);
```

定義される場所 rts.src 内の atanh.c と atanhf.c

説明 atanh および atanhf 関数は、浮動小数点引数 x のハイパボリック・アーク・タンジェントを戻します。戻り値の範囲は [-1.0, 1.0] です。

atexit exit() に呼び出される関数の登録

C の構文

```
#include   <stdlib.h>

int atexit(void (*fun)(void));
```

C++ の構文

```
#include   <cstdlib>

int std::atexit(void (*fun)(void));
```

定義される場所 rts.src 内の 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。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。

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 内の atof.c、atoi.c、atol.c、および atoll.c

説明　上記の 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、オプションの符号を表す *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。

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。

bsearch　配列の検索

C の構文

#include　<stdlib.h>

void ***bsearch**(const void *key, const void *base, size_t nmemb, size_t size,
　　int (*compar)(const void *, const void *));

C++ の構文

#include　<cstdlib>

void ***std::bsearch**(const void *key, const void *base, size_t nmemb,
　　size_t size, int (*compar)(const void *, const void *));

定義される場所　rts.src 内の bsearch.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）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

例

```
int list[10] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };

 int intcmp(const void *ptr1, const void *ptr2)
 {
    return *(int*)ptr1 - *(int*)ptr2;
 }
```

calloc　メモリの割り当てとクリア

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 内の memory.c

説明

calloc 関数は、num 個のオブジェクトのそれぞれに size バイト（size は符号なし整数または size_t）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・プールまたは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 2K バイト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 -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）を指定してリンカを起動します（8.1.3 項「動的なメモリ割り当て」（8-5 ページ）を参照）。

例

この例では、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 */
```

ceil/ceilf　切り上げ

C の構文

#include <math.h>

double **ceil**(double x);
float **ceilf**(float x);

C++ の構文

#include <cmath>

double **std::ceil**(double x);
float **std::ceilf**(float x);

定義される場所 rts.src 内の ceil.c と ceilf.c

説明

ceil および ceilf 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 float ceil();
float answer

answer = ceilf(3.1415);   /* answer = 4.0  */
answer = ceilf(-3.5);     /* answer = -3.0 */
```

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 内の 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消去します。

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 内の clock.c

説明　clock 関数は、使用したプロセッサ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使用プロセッサ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 関数を記述してください**
>
> clock 関数は、スタンドアロン・シミュレータ（load6x）で動作します。load6x 環境で使用する場合、clock() はサイクルの正確なカウントを戻します。HLL デバッガで使用する場合、clock 関数は -1 を戻します。
>
> ホスト固有の clock 関数を作成できます。また、clock() が戻す値（クロックの目盛り数）を CLOCKS_PER_SEC で除算して、値を秒数で表すことができるようにするには、クロックの単位に基づいて CLOCKS_PER_SEC マクロを定義する必要もあります。

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、9.3.18 項「時間関数（time.h/ctime）」（9-27 ページ）を参照してください。

cos/cosf　コサイン

C の構文

```
#include  <math.h>

double cos(double x);
float cosf(float x);
```

C++ の構文

```
#include  <cmath>

double std::cos(double x);
float std::cosf(float x);
```

定義される場所　rts.src 内の cos.c と cosf.c

説明　cos および cosf 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。引数の値が大き過ぎると、有意性のある結果がほとんど得られないか、全く得られなくなる場合があります。

例

```
double radians, cval;
radians = 0.0;
cval = cos(radians);  /* cval = 0.0 */
```

cosh/coshf　ハイパボリック・コサイン

C の構文

```
#include  <math.h>

double cosh(double x);
float coshf(float x);
```

C++ の構文

```
#include  <cmath>

double std::cosh(double x);
float std::coshf(float x);
```

定義される場所　rts.src 内の cosh.c と coshf.c

説明　cosh および coshf 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（errno は EDOM の値に設定されます）。これらの関数は $(e^x + e^{-x}) / 2$ と等価ですが、計算速度が増し、正確度も増します。

例

```
double x, y;

x = 0.0;
y = cosh(x);   /* return value = 1.0 */
```

cot/cotf　極コタンジェント

C の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double cot(double x);
float cotf(float x);
```

C++ の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::cot(double x);
float std::cotf(float x);
```

定義される場所　rts.src 内の cot.c と cotf.c

説明　cot および cotf 関数は、浮動小数点引数 x のコタンジェントを戻します。x を 0.0 にすることはできません。x が 0.0 である場合、errno は EDOM の値に設定され、この関数は最大正数を戻します。

coth/cothf　ハイパボリック・コタンジェント

C の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double coth(double x);
float cothf(float x);
```

C++ の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::coth(double x);
float std::cothf(float x);
```

定義される場所　rts.src 内の coth.c と cothf.c

説明　coth および cothf 関数は、浮動小数点引数 x のハイパボリック・コタンジェントを戻します。戻り値の大きさは ≥ 1.0 でなければなりません。

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 内の 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、9.3.18 項「時間関数（time.h/ctime）」（9-27 ページ）を参照してください。

difftime　時間差

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 内の 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、9.3.18 項「時間関数（time.h/ctime）」（9-27 ページ）を参照してください。

div/ldiv/lldiv 除算

C の構文

```
#include <stdlib.h>

div_t div(int numer, int denom);
ldiv_t ldiv(long numer, long denom);
lldiv_t lldiv(long long numer, long long denom);
```

C++ の構文

```
#include <cstdlib>

div_t std::div(int numer, int denom);
ldiv_t std::ldiv(long numer, long denom);
lldiv_t std::lldiv(long long numer, long long denom);
```

定義される場所 rts.src 内の div.c

説明

これら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  quot;           /*  quotient */
   long  rem;            /* remainder  */
} ldiv_t;
```

- l ldiv 関数は、倍長整数の除算を行います。入力する引数は倍長整数です。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。

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 内の 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（9-41 ページ の abort 関数を参照してください）。

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。

exp/expf　指数

C の構文

```
#include  <math.h>

double exp(double x);
float expf(float x);
```

C++ の構文

```
#include  <cmath>

double std::exp(double x);
float std::expf(float x);
```

定義される場所　rts.src 内の exp.c と expf.c

説明

exp および expf 関数は、実数 x の指数値を戻します。戻り値は e の x 乗です。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 = approx 7.38 (e*e, e is 2.17828)... */
```

exp10/exp10f 指数

C の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double exp10(double x);
float exp10f(float x);
```

C++ の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::exp10(double x);
float std::exp10f(float x);
```

定義される場所 rts.src 内の exp10.c と exp10f.c

説明 exp10 および exp10f 関数は 10 の x 乗を戻します。x は実数です。x の値が大きすぎると範囲エラーになります。

exp2/exp2f 指数

C の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double exp2(double x);
float exp2f(float x);
```

C++ の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::exp2(double x);
float std::exp2f(float x);
```

定義される場所 rts.src 内の exp2.c と exp2f.c

説明 exp2 および exp2f 関数は 2 の x 乗を戻します。x は実数です。x の値が大きすぎると範囲エラーになります。

fabs/fabsf 絶対値

C の構文

```
#include  <math.h>

double fabs(double x);
float fabsf(float x);
```

C++ の構文

```
#include  <cmath>

double std::fabs(double x);
float std::fabsf(float x);
```

定義される場所 rts.src 内の fabs.c

説明 fabs および fabsf 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 */
```

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 内の 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したファイルをクローズします。

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 内の 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。

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 内の 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。

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 内の 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。

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 内の 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。

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 内の 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に、pos が指すオブジェクトを格納します。

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 内の 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。

floor/floorf　切り捨て

C の構文

```
#include  <math.h>

double floor(double x);
float floorf(float x);
```

C++ の構文

```
#include  <cmath>

double std::floor(double x);
float std::floorf(float x);
```

定義される場所　rts.src 内の floor.c と floorf.c

説明　floor および floorf 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 */
```

fmod/fmodf 浮動小数点の剰余

C の構文

#include <math.h>

double **fmod**(double x, double y);
float **fmodf**(float x, float y);

C++ の構文

#include <cmath>

double **std::fmod**(double x, double y);
float **std::fmodf**(float x, float y);

定義される場所 rts.src 内の fmod.c と fmodf.c

説明

fmod および fmodf 関数は、x を y で割った剰余の浮動小数点数を戻します。y==0 のとき、この関数は 0 を戻します。

この関数は x - trunc(x / y) y と数学上等しくなりますが、同じように作成された C 式とは等しくありません。たとえば、fmod(x, 3.0) は、小整数 x > 0.0 の場合は、0.0、1.0、または 2.0 です。x が x / y が正確に表示できないほど大きくなる場合、fmod(x, 3.0) は引き続き正しい答えを出しますが、C 式はすべての x の値に対して 0.0 を戻します。

例

```
double x, y, r;

x = 11.0;
y = 5.0;
r = fmod(x, y);         /* fmod returns 1.0 */
```

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 内の 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。

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 内の fprint.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、ストリームへの書き込み方法が記述されています。

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 内の 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。

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 内の 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。

fread　ストリームの読み込み

C の構文　#include　<stdio.h>

size_t **fread**(void *_ptr, size_t _size, size_t _count, FILE *_fp);

C++ の構文　#include　<cstdio>

size_t **std::fread**(void *_ptr, size_t _size, size_t _count, FILE *_fp);

定義される場所　rts.src 内の 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。

free　メモリの解放

C の構文　#include　<stdlib.h>

void **free**(void *packet);

C++ の構文　#include　<cstdlib>

void **free**(void *packet);

定義される場所　rts.src 内の memory.c

説明　free 関数は、malloc、calloc、realloc の呼び出しにより割り当てられた（packet 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。割り当てられていない空間を解放しようとすると、関数は動作せずに戻ります。詳細は、8.1.3 項「動的なメモリ割り当て」（8-5 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       */
```

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 内の fopen.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、再度 _fp をこのファイルに関連付けます。_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。

frexp/frexpf　小数部と指数部

C の構文

```
#include  <math.h>

double frexp(double value, int *exp);
float frexpf(float value, int *exp);
```

C++ の構文

```
#include  <cmath>

double std::frexp(double value, int *exp);
float std::frexpf(float value, int *exp);
```

定義される場所 rts.src 内の frexp.c と frexpf.c

説明 frexp および frexpf 関数は、浮動小数点数を正規化した小数部 (f) と 2 の整数乗に分けます。これらの関数は、value = $f \times 2^{exp}$, $0.5 \le |f| < 1.0$ となる値 f を戻します。指数は、exp が指す整数に保存されます。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 */
```

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 内の 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。

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 内の 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。位置は _ptrname に指定されます。バイナリ・ファイルの場合、_ptrname から _offset の位置に標識を設定します。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。

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);
```

定義される場所　rts.src 内の 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() からの値を指定する必要があります。

ftell　現行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 内の 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。

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 内の fwrite.c

説明　fwrite 関数は _ptr が指すメモリからデータ・ブロックを取り出し、_fp が指すストリームに書き込みます。

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 内の 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。

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 内の 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。

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 内の trgdrv.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。

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 内の 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。可能ならば、gets ではなく、fgets() を使用してください。

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 内の 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、9.3.18 項「時間関数（time.h/ctime）」（9-27 ページ）を参照してください。

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 内の is*xxx*.c と ctype.c
また ctype.h/cctype でも定義されています（マクロとして定義）。

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、マクロは関数よりも処理速度が高速です。

labs/llabs

9-41 ページの abs/labs/llabs を参照してください。

ldexp/ldexpf

2 の累乗による乗算

C の構文

```
#include   <math.h>

double ldexp(double x, int exp);
float ldexpf(float x, int exp);
```

C++ の構文

```
#include   <cmath>

double std::ldexp(double x, int exp);
float std::ldexpf(float x, int exp);
```

定義される場所 rts.src 内の ldexp.c と ldexpf.c

説明 ldexp および ldexpf 関数は、浮動小数点数 x に 2^{exp} を掛け、$x \times 2^{exp}$ を戻します。指数部 *exp* は負でも正でも指定できます。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 */
```

ldiv/lldiv

9-56 ページの div/ldiv/lldiv 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 内の 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、9.3.18 項「時間関数（time.h/ctime）」（9-27 ページ）を参照してください。

log/logf　自然対数

C の構文

```
#include  <math.h>

double log(double x);
float logf(float x);
```

C++ の構文

```
#include  <cmath>

double std::log(double x);
float std::logf(float x);
```

定義される場所　rts.src 内の log.c と logf.c

説明　log および logf 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。

例

```
float x, y;

x = 2.718282;
y = logf(x);          /* y = approx 1.0 */
```

log10/log10f　常用対数

C の構文

```
#include  <math.h>

double log10(double x);
float log10f(float x);
```

C++ の構文

```
#include  <cmath>

double std::log10(double x);
float std::log10f(float x);
```

定義される場所　rts.src 内の log10.c と log10f.c

説明　log10 および log10f 関数は、実数 x の底が 10 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 = log10f(x);        /* y = approx 1.0 */
```

log2/log2f　底が 2 の対数

C の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double log2(double x);
float log2f(float x);
```

C++ の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::log2(double x);
float std::log2f(float x);
```

定義される場所　rts.src 内の log2.c と log2f.c

説明　log2 および log2f 関数は、実数 x の底が 2 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 = 2.0;
y = log2f(x);          /* y = approx 1.0 */
```

longjmp　9-88 ページ の setjmp/longjmp を参照してください。

lltoa　倍長整数の ASCII への変換

C の構文

プロトタイプはありません。

```
int lltoa(long long val, char *buffer);
```

C++ の構文

プロトタイプはありません。

```
int std::lltoa(long long val, char *buffer);
```

定義される場所　rts.src 内の lltoa.c

説明　lltoa 関数は標準外（非 ISO）関数で、互換性を維持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。この関数は、rts.src の中でプロトタイプ化されていません。lltoa 関数は、倍長整数 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。lltoa 関数は、バッファに書き込まれた文字数を戻します。

ltoa　長整数の ASCII への変換

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 内の ltoa.c

説明　ltoa 関数は標準外（非 ISO）関数で、互換性を維持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。この関数は、rts.src の中でプロトタイプ化されていません。ltoa 関数は、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。

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　<stdlib.h>

void ***std::malloc**(size_t size);

定義される場所　rts.src 内の memory.c

説明　malloc 関数は size バイトの空間をオブジェクトに割り当て、その空間へ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 2K バイト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 -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）を指定してリンカを起動します。詳細は、8.1.3 項「動的なメモリ割り当て」（8-5 ページ）を参照してください。

memalign　ヒープの位置合わせ

C の構文

#include<stdlib.h>

void ***memalign**(size_t alignment, size_t _size);

C++ の構文

#include <stdlib.h>

void ***std::memalign**(size_t alignment, size_t _size);

定義される場所　rts.src 内の memory.c

説明

memalign 関数は、ANSI/ISO 規格の malloc 関数と同様の動作をしますが、alignment バイト境界に位置合わせされたメモリ・ブロックへのポインタを戻す点が異なっています。したがって、_size が 128 で alignment が 16 の場合、memalign は、16 バイト境界に位置合わせされた 128 バイトのメモリ・ブロックへのポインタを戻します。

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 ***std::memchr**(const void *cs, int c, size_t n);

定義される場所　rts.src 内の memchr.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。

memcmp　メモリの比較

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 内の memcmp.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。

memcpy　メモリ・ブロック・コピー — オーバーラップ非対応

C の構文

```
#include  <string.h>

void *memcpy(void *s1, const void *s2, register size_t n);
```

C++ の構文

```
#include  <cstring>

void *std::memcpy(void *s1, const void *s2, register size_t n);
```

定義される場所　rts.src 内の memcpy.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。

memmove　メモリ・ブロック・コピー — オーバーラップ対応

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 内の memmove.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。

memset　メモリ内での値のコピー

C の構文

#include <string.h>

void ***memset**(void *mem, int ch, size_t length);

C++ の構文

#include <cstring>

void ***std::memset**(void *mem, int ch, size_t length);

定義される場所 rts.src 内の memset.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。

minit　動的なメモリ・プールのリセット

C の構文　プロトタイプはありません。

void **minit**(void);

C++ の構文　プロトタイプはありません。

void **std::minit**(void);

定義される場所　rts.src 内の memory.c

説明　minit 関数は、malloc、calloc、realloc 関数の呼び出しによって割り当てられていたすべての空間をリセットします。

minit 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 2K バイト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 -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）を指定してリンカを起動します。詳細は、8.1.3 項「動的なメモリ割り当て」（8-5 ページ）を参照してください。

注：minit の実行後、以前に割り当てられたオブジェクトは使用できなくなります

minit 関数の呼び出しにより、ヒープ内のすべてのメモリ空間は再び使用可能になります。以前に割り当てたオブジェクトはすべてなくなります。アクセスを試みるのはやめてください。

mktime　カレンダ時への変換

C の構文

#include <time.h>

time_t **mktime**(register struct tm *tptr);

C++ の構文

#include <ctime>

time_t **std::mktime**(register struct tm *tptr);

定義される場所 rts.src 内の mktime.c

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。tptr 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、9.3.18 項「時間関数（time.h/ctime）」（9-27 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 */
```

modf/modff　符号付き整数と符号付き小数

C の構文

```
#include <math.h>

double modf(double value, double *ip);
float modff(float value, float *ip);
```

C++ の構文

```
#include <cmath>

double std::modf(double value, double *ip);
float std::modff(float value, float *ip);
```

定義される場所 rts.src 内の modf.c と modff.c

説明 modf および modff 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 = -10.125;

fpart = modf(value, &ipart);

/* After execution, ipart contains -10.0, */
/* and fpart contains -.125.               */
```

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 内の perror.c

説明 perror 関数は、_s の文字列とエラー番号をマップして得られた文字列を結合し、エラー・メッセージを出力します。

pow/powf　ベキ乗

C の構文

```
#include  <math.h>

double pow(double x, double y);
float powf(float x, float y);
```

C++ の構文

```
#include  <cmath>

double std::pow(double x, double y);
float std::powf(float x, float y);
```

定義される場所　rts.src 内の pow.c と powf.c

説明　pow および powf 関数は、x の y 乗を戻します。これらの pow 関数は exp(y × log(x)) と数学上等価ですが、計算速度が増し、正確度も増します。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示できない場合は、範囲エラーになります。

例

```
double x, y, z;

x = 2.0;
y = 3.0;
y = pow(x, y);          /* return value = 8.0 */
```

powi/powif　整数のベキ乗

C の構文

```
#define _TI_ENHANCED_MATH_H 1
#include  <math.h>

double powi(double x, int y);
float powif(float x, int y);
```

C++ の構文

```
#define _TI_ENHANCED_MATH_H 1
#include  <cmath>

double std::powi(double x, int y);
float std::powif(float x, int y);
```

定義される場所　rts.src 内の powi.c と powif.c

説明　powi および powif 関数は、x^i を戻します。これらの powi 関数は pow（x, (double) i）と数学上等価ですが、計算速度が増し、正確度はほぼ同じです。x = 0 かつ i ≤ 0 の場合、または x が負で i 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示できない場合は、範囲エラーになります。

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 内の 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。

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　<cstdio>

int **std::putc**(int _x, FILE *_fp);

定義される場所　rts.src 内の fputc.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。

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 内の fputc.c

説明　putchar 関数は、標準出力デバイスに文字を書き込みます。

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 内の fputs.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。

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)(const void *, const void *));
```

C++ の構文

```
#include   <cstdlib>

void std::qsort(void *base, size_t nmemb, size_t size,
        int (*compar)(const void *, const void *));
```

定義される場所　rts.src 内の qsort.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

例

```
int list[10] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };

 int intcmp(const void *ptr1, const void *ptr2)
 {
     return *(int*)ptr1 - *(int*)ptr2;
 }
```

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 内の rand.c

説明

2 つの関数が共に機能して、疑似乱数シーケンスを生成します。

- ❏ rand 関数は、0 から RAND_MAX までの範囲で疑似乱整数を戻します。
- ❏ rand 関数に対する後の呼び出しで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 rand で生成されます。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。

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 内の 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 によってバイト単位で指定したサイズに変更します。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・プールまたは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 2K バイト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 -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）を指定してリンカを起動します。詳細は、8.1.3 項「動的なメモリ割り当て」（8-5 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 内の 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。

rename　ファイル名の変更

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 内の lowlev.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 が指す新しい名前になります。

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>

int rewind(register FILE *_fp);
```

C++ の構文

```
#include  <cstdlib>

int std::rewind(register FILE *_fp);
```

定義される場所　rts.src 内の 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。

round/roundf　最も近い整数への丸め

C の構文

```
#define _TI_ENHANCED_MATH_H 1
#include  <math.h>

double std::round(double x);
float std::roundf(float x);
```

C++ の構文

```
#define _TI_ENHANCED_MATH_H 1
#include  <cmath>

double round(double x);
float roundf(float x);
```

定義される場所　rts.src 内の round.c と roundf.c

説明　round および roundf 関数は、最も近い整数に丸められた x に相当する浮動小数点数を戻します。x が 2 つの整数から等距離にある場合、偶数値が戻されます。

例

```
float x, y, u, v, r, s, o, p;

x = 2.65;
y = roundf(x);          /* y = 3  */

u = -5.28
v = roundf(u);          /* v = -5 */

r = 3.5
s = roundf(s);          /* s = 4 */

o = 6.5
p = roundf(o);          /* p = 6.0 */
```

rsqrt/rsqrtf　逆平方根

C の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double rsqrt(double x);
float rsqrtf(float x);
```

C++ の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::rsqrt(double x);
float std::rsqrtf(float x);
```

定義される場所　rts.src 内の rsqrt.c と rsqrtf.c

説明　rsqrt および rsqrtf 関数は実数 x の逆平方根を戻します。rsqrt(x) 関数は 1.0 / sqrt(x) と数学上等価ですが、計算速度がかなり増し、正確度はほぼ同じです。引数が負の場合は領域エラーになります。

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 内の 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。

setbuf　ストリームのバッファの指定

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 内の 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。値は戻りません。

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 内の setjmp.asm

説明

setjmp.h 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とマクロを定義し、関数を宣言します。

- **jmp_buf** 型は、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です。
- **setjmp** マクロは、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の戻りの場合、setjmp マクロは 0 を戻します。longjmp 関数の呼び出しの結果として戻る場合、setjmp マクロは 0 以外の値を戻します。
- **longjmp** 関数は、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。

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);
   . . .
}
```

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   <stdio.h>

int setvbuf(register FILE *_fp, register char *_buf, register int _type,
        register size_t _size);
```

C++ の構文

```
#include   <cstdio>

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 内の 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の定義と関連付けを行います。_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。

sin/sinf　サイン

C の構文

```
#include  <math.h>

double sin(double x);
float sinf(float x);
```

C++ の構文

```
#include  <cmath>

double std::sin(double x);
float std::sinf(float x);
```

定義される場所 rts.src 内の sin.c と sinf.c

説明 sin および sinf 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。引数の値が大き過ぎると、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;     /* sin returns sval        */

radian = 3.1415927;
sval = sin(radian);      /* sin returns approx -1.0 */
```

sinh/sinhf　ハイパボリック・サイン

C の構文

```
#include  <math.h>

double sinh(double x);
float sinhf(float x);
```

C++ の構文

```
#include  <cmath>

double std::sinh(double x);
float std::sinhf(float x);
```

定義される場所 rts.src 内の sinh.c と sinhf.c

説明 sinh および sinhf 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。これらの関数は $(e^x - e^{-x}) / 2$ と等価ですが、計算速度が増し、正確度も増します。

例

```
double x, y;

x = 0.0;
y = sinh(x);      /* y = 0.0 */
```

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 内の 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。

sqrt/sqrtf　平方根

C の構文　#include <math.h>

double **sqrt**(double x);
float **sqrtf**(float x);

C++ の構文　#include <cmath>

double **std::sqrt**(double x);
float **std::sqrtf**(float x);

定義される場所　rts.src 内の sqrt.c と sqrtf.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 */
```

srand　9-84 ページの rand/srand 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 <stdio.h>

int **sscanf**(const char *_str, const char *_fmt, ...);

C++ の構文　#include <cstdio>

int **std::sscanf**(const char *_str, const char *_fmt, ...);

定義される場所　rts.src 内の 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 が指す文字列は、ストリームの読み込み方法を記述します。

strcat　文字列の連結

C の構文

```
#include  <string.h>

char *strcat(char *string1, const char *string2);
```

C++ の構文

```
#include  <cstring>

char *std::strcat(char *string1, const char *string2);
```

定義される場所　rts.src 内の strcat.c

説明　strcat 関数は、string2 のコピー（終了ヌル文字を含む）を string1 の末尾に追加します。string2 の先頭の文字は、もとは 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。string1 は、文字列全体を入れられる大きさでなければなりません。

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"                                    */
```

strchr　文字の最初の出現の検出

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 内の strchr.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。

strcmp/strcoll　文字列の比較

C の構文

#include <string.h>

int **strcmp**(const char *string1, register const char *string2);
int **strcoll**(const char *string1, const char *string2);

C++ の構文

#include <cstring>

int **std::strcmp**(const char *string1, register const char *string2);
int **std::strcoll**(const char *string1, const char *string2);

定義される場所　rts.src 内の strcmp.c と strcoll.c

説明

strcmp 関数と strcoll 関数は、string2 と string1 を比較します。これらの関数は等価です。どちらも ISO C との互換性に対応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 execute        */
   }
if (strcoll(stra, strc) == 0)
   {
      /* statements here execute also      */
   }
```

strcpy　文字列のコピー

C の構文

#include　<string.h>

char ***strcpy**(register char *dest, register const char *src);

C++ の構文

#include　<cstring>

char ***std::strcpy**(register char *dest, register const char *src);

定義される場所

rts.src 内の strcpy.c

説明

strcpy 関数は、src（終了ヌル文字を含む）を dest 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 dest 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"             */
```

strcspn　不一致の文字数の検出

C の構文　#include　<string.h>

size_t **strcspn**(register const char *string, const char *chs);

C++ の構文　#include　<cstring.h>

size_t **std::strcspn**(register const char *string, const char *chs);

定義される場所　rts.src 内の strcspn.c

説明　strcspn 関数は、全体が chs 内にない文字から構成される string の最初の部分の長さを戻します。string の中の先頭の文字が 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 */
```

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 内の strerror.c

説明　strerror 関数は文字列「string error」を戻します。この関数は ISO との互換性に対応するための関数です。

strftime　時間の書式

C の構文

```
#include  <time.h>

size_t *strftime(char *out, size_t maxsize, const char *format,
        const struct tm *time);
```

C++ の構文

```
#include  <ctime>

size_t *std::strftime(char *out, size_t maxsize, const char *format,
        const struct tm *time);
```

定義される場所　rts.src 内の strftime.c

説明　strftime 関数は format 文字列に基づいて（time が指す）時間を書式化し、書式化された時間を文字列 out に戻します。out には、最高 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。次のリストは、有効な文字と、それぞれの展開内容を示したものです。

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 進数で表した日（0 ～ 31）
%H	10 進数で表した時刻（24 時間制）（00 ～ 23）
%I	10 進数で表した時刻（12 時間制）（01 ～ 12）
%j	10 進数で表した日（001 ～ 366）
%m	10 進数で表した月（01 ～ 12）
%M	10 進数で表した分（00 ～ 59）
%p	各地域ごとの午前や午後の呼び方
%S	10 進数で表した秒（00 ～ 59）
%U	10 進数で表したその年の週番号（週の初日は日曜日）（00 ～ 52）
%x	日付の表記
%X	時間の表記
%y	10 進数で表した世紀を表す最初の 2 桁を除いた年（00 ～ 99）

文字	展開内容
%Y	10 進数で表した世紀を表す最初の 2 桁を付けた年
%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、9.3.18 項「時間関数（time.h/ctime）」（9-27 ページ）を参照してください。

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.h>

size_t std::strlen(const char *string);
```

定義される場所　rts.src 内の strlen.c

説明　strlen 関数は、string の長さを戻します。C では、文字列は値が 0 の文字（ヌル文字）で終了します。戻っ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  */
```

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 ***strncat**(char *dest, const char *src, size_t n);

定義される場所　rts.src 内の strncat.c

説明

strncat 関数は、src の最大 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 の先頭の文字上に上書きされます。strncat 関数は結果にヌル文字を付けます。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"             */;
```

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 内の strncmp.c

説明

strncmp 関数は、string2 の最大 n 個の文字を string1 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 not?";
size_t size = 4;

if (strcmp(stra, strb, size) > 0)
   {
      /* statements here execute       */
   }
if (strcomp(stra, strc, size) == 0)
   {
      /* statements here execute also  */
   }
```

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 内の strncpy.c

説明　strncpy 関数は、最高で 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 の終わりには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[100] = "she is the one mother warned you of";
char strb[100] = " he is";
char strc[100] = "I am the one father warned you of";
char strd[100] = "oops";
int length = 5;

strncpy (stra,strb,length);

/* stra--> " he is the one mother warned you of\0" */;
/* strb--> " he is\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 is\0"                              */;
/* strc--> "I am the one father warned you of\0"   */;
/* strd--> "oops\0"                                */;

strncpy (stra,strd,length);

/* stra--> "oops\0"                                */;
/* strb--> " he is\0"                              */;
/* strc--> "I am the one father warned you of\0"   */;
/* strd--> "oops\0"                                */;
```

strpbrk　一致する文字の検出

C の構文

#include <string.h>

char ***std::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 内の strpbrk.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 was not me";
char *strb = "wave";
char *a;

a = strpbrk (stra,strb);
```

この例の後では、*a は was の中の w を指します。

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 内の strrchr.c

説明

strrchr 関数は、string において最後に現れる 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';
```

この例のあとでは、*b は文字列の終わりに近い zs の中の z を指します。

strspn　一致する文字数の検出

C の構文

```
#include  <string.h>

size_t strspn(register const char *string, const char *chs);
```

C++ の構文

```
#include  <cstring>

size_t std::strspn(register const char *string, const char *chs);
```

定義される場所　rts.src 内の strspn.c

説明　strspn 関数は、chs にある文字だけで構成された string の先頭の部分の長さを戻します。string の中の先頭の文字が 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 = strspn(stra,strb);      /* length = 3 */
length = strspn(stra,strc);      /* length = 0 */
```

strstr　一致する文字列の検出

C の構文

```
#include  <string.h>

char *strstr(register const char *string1, const char *string2);
```

C++ の構文

```
#include  <cstring>

char *std::strstr(register const char *string1, const char *string2);
```

定義される場所　rts.src 内の strstr.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、最初の文字列の what の中の w を指しています。

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 std::strtoll(const char *st, char **endptr, int base);
unsigned long std::strtoul(const char *st, char **endptr, int base);
unsigned long long std::strtoull(const char *st, char **endptr, int base);
```

定義される場所　rts.src 内の strtod.c、strtol.c、strtoll.c、strtoul.c、および strtoull.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、もう 1 つの引数 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。変換後の文字列がオーバーフローになると、この関数は、±HUGE_VAL を戻します。変換後の文字列がアンダーフローになると、この関数は 0 を戻します。変換後の文字列がオーバーフローやアンダーフローになると、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*]

- strtoul 関数は、文字列を符号なし長整数に変換します。文字列は次の形式で指定します。

  [*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。文字列は次の形式で指定します。

  [*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、オプションの符号を示す sign、さらに数値の整数部を示す digits が続きます。その後は数値の小数部が続き、オプションの符号を示す sign をもつ指数部が続きます。

認識できない文字が初めて出現した位置で、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。

strtok 文字列のトークンへのブレーク

C の構文

```
#include  <string.h>

char *std::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 内の 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。呼び出しのたびに次のトークンへのポインタが戻ります。

例 次の例の strtok の最初の呼び出しの後、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"     */
```

strxfrm　文字の変換

C の構文

#include <string.h>

size_t **strxfrm**(register char *to, register const char *from, register size_t n);

C++ の構文

#include <cstring>

size_t **std::strxfrm**(register char *to, register const char *from,
register size_t n);

定義される場所 rts.src 内の strxfrm.c

説明 strxfrm 関数は、from が指す n 文字を to が指す n 文字に変換します。

tan/tanf　タンジェント

C の構文

#include <math.h>

double **tan**(double x);
float **tanf**(float x);

C++ の構文

#include <cmath>

double **std::tan**(double x);
float **std::tanf**(float x);

定義される場所 rts.src 内の tan.c と tanf.c

説明 tan および tanf 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。引数の値が大き過ぎると、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);            /* y = approx 1.0 */
```

tanh/tanhf　ハイパボリック・タンジェント

C の構文

```
#include  <math.h>

double tanh(double x);
float tanhf(float x);
```

C++ の構文

```
#include  <cmath>

double std::tanh(double x);
float std::tanhf(float x);
```

定義される場所　rts.src 内の tanh.c と tanhf.c

説明　tanh および tanhf 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 */
```

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 内の 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、9.3.18 項「時間関数（time.h/ctime）」（9-27 ページ）を参照してください。

注 : time 関数はターゲット・システムに固有です

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。

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 内の tmpfile.c

説明　tmpfile 関数は、一時ファイルを作成します。

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 内の tmpnam.c

説明　tmpnam 関数は、有効なファイル名を示す文字列を生成します。

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 内の toascii.c

説明　toascii 関数は、下位の 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。

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 内の tolower.c と toupper.c

説明 これら 2 つの関数は、単独の英字 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。

trunc/truncf 0 への切り捨て

C の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double trunc(double x);
float truncf(float x);
```

C++ の構文

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::trunc(double x);
float std::truncf(float x);
```

定義される場所 rts.src 内の trunc.c と truncf.c

説明 trunc および truncf 関数は、0 の方向に最も近い整数に丸められた x に相当する浮動小数点数を戻します。

例

```
float x, y, u, v;

x = 2.35;
y = truncf(x);          /* y = 2 */

u = -5.65;
v = truncf(v);          /* v = -5 */
```

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, register FILE *_fp);
```

C++ の構文

```
#include  <cstdlib>

int std::ungetc(int _c, register FILE *_fp);
```

定義される場所　rts.src 内の 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。

va_arg/va_end/va_start　可変引数マクロ

C の構文

```
#include  <stdarg.h>

typedef  char *va_list;
         type va_arg(va_list  _ap, _type);
         void va_end(va_list  _ap);
         void va_start(va_list  _ap, parmN);
```

C++ の構文

```
#include  <cstdarg>

typedef  char *std::va_list;
         type std::va_arg(va_list  _ap, _type);
         void std::va_end(va_list  _ap);
         void std::va_start(va_list  _ap, parmN);
```

定義される場所　rts.src 内の stdarg.h

説明　関数の中には、型が変化する可変数の引数で呼び出されるものがあります。このような関数（可変引数関数）では、次のマクロを使用して、実行時に引数リストを調べることができます。_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 に対する連続呼び出し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 の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                    */
}
```

vfprintf　ストリームへの書き込み

C の構文

#include <stdio.h>

int **vfprintf**(FILE *_fp, const char *_format, va_list _ap);

C++ の構文

#include <cstdio>

int **std::vfprintf**(FILE *_fp, const char *_format, va_list _ap);

定義される場所

rts.src 内の vfprintf.c

説明

vffprintf 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。

vprintf　標準出力への書き込み

C の構文

#include <stdio.h>

int **vprintf**(const char *_format, va_list _ap);

C++ の構文

#include <cstdio>

int **std::vprintf**(const char *_format, va_list _ap);

定義される場所

rts.src 内の 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。

vsprintf　ストリームの書き込み

C の構文

```
#include  <stdio.h>

int vsprintf(char *_string, const char *_format, va_list _ap);
```

C++ の構文

```
#include  <cstdio>

int std::vsprintf(char *_string, const char *_format, va_list _ap);
```

定義される場所　rts.src 内の 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。

第10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、多くの構成や互いに互換性を維持する必要のないオプションでコードをコンパイルできます。個々のランタイムサポート・ライブラリで可能なすべての組み合わせを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。

アーカイバと mk6x ユーティリティを使用して、各自のランタイムサポート・ライブラリを作成できます。mk6x ユーティリティについては、この章で説明します。アーカイバについては、TMS320C6000 Assembly Language Tools User's Guide を参照してください。

項目 **ページ**

10.1 標準ランタイムサポート・ライブラリ 10-2
10.2 ライブラリ作成ユーティリティの起動方法 10-3
10.3 ライブラリ作成ユーティリティのオプション 10-4
10.4 オプションのまとめ 10-5

10.1 標準ランタイムサポート・ライブラリ

C6000 コード生成ツールに付属するランタイムサポート・ライブラリは、以下のコマンドを用いて生成されています。

コマンド	コメント
`mk6x -o -ml2 --RTS rts.src -l rts6200.lib`	基本、C6200
`mk6x -o -ml2 --RTS -me rts.src -l rts6200e.lib`	基本、C6200、ビッグエンディアン
`mk6x -o -ml2 --RTS -mv6400 rts.src -l rts6400.lib`	基本、C6400
`mk6x -o -ml2 --RTS -mv6400 -me rts.src -l rts6400e.lib`	基本、C6400、ビッグエンディアン
`mk6x -o -ml2 --RTS -mv6700 rts.src -l rts6700.lib`	基本、C6700
`mk6x -o -ml2 --RTS -mv6700 -me rts.src -l rts6700e.lib`	基本、C6700、ビッグエンディアン

各ライブラリ用の基本のオプション・セットは、次のとおりです。

- ❏ 最適化レベル 2（-o または -o2 オプション）を適用します。
- ❏ グローバルな構造体および配列は far データとしてアクセスします。関数呼び出しには far コールを使用します。（-ml2 オプション）
- ❏ rts.src に含まれている TI が独自に開発したソース・コードに基づく C++ ランタイムサポート・ライブラリのコンパイルを有効化します。（--RTS オプション）

10.2 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。

mk6x [*options*]*src_arch1* [**-l***obj.lib1*][*src_arch2* [**-l***obj.lib2*]]**...**

mk6x 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、10.2 節および 10.4 節を参照してください）。

src_arch ソース・アーカイブ・ファイルの名前です。mk6x 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、mk6x 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。

mk6x 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またはアセンブル、あるいはその両方を行うため、各ソース・ファイルに対してコンパイラ・プログラムを実行します。次にすべてのオブジェクト・ファイルが収集されて、1 つのオブジェクト・ライブラリが作成されます。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 C6X_C_OPTION、C_OPTION、C6X_C_DIR、および C_DIR は無視されます。

10.3 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、およびコンパイラ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。以下のオプションは、ライブラリ作成ユーティリティにだけ適用します。

--c ソース・アーカイブに含まれる C ソース・ファイルをライブラリから抽出します。ユーティリティの実行後は、これらをカレント・ディレクトリに残します。

--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・ディレクトリに残されます。ツールに付属している rts.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 デフォルトのオプションを使用して、独自の C++ ソース・コードをランタイムサポート・ライブラリにコンパイルします。標準 C6000 ランタイム・サポート・ライブラリの独自のバージョンを作成する場合は、このオプションが必要です。

--u オブジェクト・ライブラリの作成時にソース・アーカイブのヘッダ・ファイルを使用しません。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、各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）。

10.4 オプションのまとめ

ライブラリ作成ユーティリティで使用できるその他のオプションは、コンパイラとアセンブラで使用されるオプションに直接対応しています。表 10–1 にこのようなオプションのリストを示します。これらのオプションの詳細は、「ページ」の欄に示されているページを参照してください。

表 10–1. オプションとその機能のまとめ

(a) コンパイラを制御するオプション

オプション	機能	ページ
-D*name*[=*def*]	*name* を事前に定義します。	2-15
-g	シンボリック・デバッグを有効にします。	2-18
-U*name*	*name* を未定義にします。	2-17

(b) マシン固有のオプション

オプション	機能	ページ
-ma	変数にエイリアスが設定されていることを前提とします。	3-25
-mb	バージョン 4.0 のツールまたは C6200/C6700 オブジェクト・コードの配列位置合わせ制限と互換性があるように C6400 コードをコンパイルします。	2-45
-mc	加法の浮動小数点演算の並べ換えを防止します。	3-28
-me	ビッグエンディアン形式でオブジェクト・コードを作成します。	2-16
-mh*n*	見込み実行ができるようになります。	3-14
-mi*n*	割り込みしきい値を定義します。	2-43
-ml*n*	near 前提事項と far 前提事項を 4 つのレベル（-ml0、-ml1、-ml2、-ml3）で変更します。	2-16
-mo	関数のサブセクションを有効にします。	5-13
-ms*n*	コード・サイズを 4 つのレベル（-ms0、-ms1、-ms2、-ms3）で制御します。	3-17
-mt	特定のエイリアス技法が使用されないことを指定します。	3-26
-mu	ソフトウェアのパイプライン化を取り止めます。	3-5
-mv*n*	ターゲット CPU バージョンを選択します。	2-17

表 10–1. オプションとその機能のまとめ（続き）

（c）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を抑止します（ただし、-o3 最適化は自動インラインを実行し続けます）。	2-40
-pk	コードを K&R 互換コードにします。	7-36
-pr	緩和モードを可能にします。厳密な ISO 違反を無視します。	7-38
-ps	厳密な ISO モードを有効にします（K&R C ではなく、C/C++ の場合）。	7-38

（d）診断を制御するパーサのオプション

オプション	機能	ページ
-pdr	注釈（軽い警告）を発行します。	2-33
-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

（e）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います。	3-2
-O1	-O0 最適化と、さらにローカル最適化を行います。	3-2
-O2（または -O）	-O1 最適化と、さらにグローバル最適化を行います。	3-3
-O3	-O2 最適化と、さらにファイル最適化を行います。mk6x により -oI0 と -op0 が自動的に設定されることに注意してください。	3-3

（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維持します。	2-24

（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[.]*new 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。	2-21
-eo[.]*new 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。	2-21

第11章

C++ ネーム・デマングラ

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、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実現しています。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内の名前のようにマングルされた名前を調べる場合は、C++ ソース・コード内で、マングルされた名前を対応する名前と関連付けるのが難しい場合があります。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。

以下のトピックにより、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**

11.1 C++ ネーム・デマングラの起動方法 11-2
11.2 C++ ネーム・デマングラのオプション 11-2
11.3 C++ ネーム・デマングラの使用例 11-3

11.1 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ための構文は、次のとおりです。

dem6x [*options*][*filenames*]

dem6x C++ ネーム・デマングラを起動するコマンドです。

options ネーム・デマングラの動作に影響を与えるオプションです。このオプションは、コマンド行の任意の場所に指定できます（オプションについては、11.2 節「C++ ネーム・デマングラのオプション」（11-2 ページ）を参照してください）。

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、dem6x 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 オプションを使用できます。

11.2 C++ ネーム・デマングラのオプション

次のオプションは、C++ ネーム・デマングラだけに適用されます。

-h C++ ネーム・デマングラ・オプションのオンライン要約を提供するヘルプ画面を出力します。

-o *file* 標準出力ではなく、指定されたファイルに出力します。

-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）。

11.3 C++ ネーム・デマングラの使用例

例 11–1 は、C++ プログラム例、およびその結果 C6000 コンパイラによって出力されるアセンブリを示しています。例 11–1（b）では、すべてのリンク名がマングルされます。つまり、そのシグニチャ情報が名前にエンコードされます。

例 11–1. 名前のマングリング

（a）calories_in_a_banana の C コード

```
class banana {

public:

    int calories(void);

    banana();

    ~banana();

};

int calories_in_a_banana(void)
{
    banana x;

    return x.calories();
}
```

（b）calories_in_a_banana のアセンブリ出力

```
_calories_in_a_banana__Fv:
;** --------------------------------------------------------------------------*
           CALL    .S1     ___ct__6bananaFv  ; |10|
           STW     .D2T2   B3,*SP--(16)      ; |9|
           MVKL    .S2     RL0,B3            ; |10|
           MVKH    .S2     RL0,B3            ; |10|
           ADD     .S1X    8,SP,A4           ; |10|
           NOP             1
RL0:; CALL OCCURS                            ; |10|
           CALL    .S1     _calories__6bananaFv ; |12|
           MVKL    .S2     RL1,B3            ; |12|
           ADD     .S1X    8,SP,A4           ; |12|
           MVKH    .S2     RL1,B3            ; |12|
           NOP             2
RL1:; CALL OCCURS                            ; |12|
           CALL    .S1     ___dt__6bananaFv  ; |13|
           STW     .D2T1   A4,*+SP(4)        ; |12|
           ADD     .S1X    8,SP,A4           ; |13|
           MVKL    .S2     RL2,B3            ; |13|
           MVK     .S2     0x2,B4            ; |13|
           MVKH    .S2     RL2,B3            ; |13|
RL2:; CALL OCCURS                            ; |13|
           LDW     .D2T1   *+SP(4),A4        ; |12|
           LDW     .D2T2   *++SP(16),B3      ; |13|
           NOP             4
           RET     .S2     B3                ; |13|
           NOP             5
           ; BRANCH OCCURS                   ; |13|
```

C++ ネーム・デマングラを実行すると、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。

```
% dem6x calories_in_a_banana.asm
```

例 11–2 は、その結果を示しています。例 11–1 のリンク名 `___ct__6bananaFv`、`_calories__6bananaFv`、および `___dt__6bananaFv` がデマングルされています。

例 11–2. C++ ネーム・デマングラ実行後の結果

```
_calories_in_a_banana():
           CALL    .S1     banana::banana()  ; |10|
           STW     .D2T2   B3,*SP--(16)      ; |9|
           MVKL    .S2     RL0,B3            ; |10|
           MVKH    .S2     RL0,B3            ; |10|
           ADD     .S1X    8,SP,A4           ; |10|
           NOP             1
RL0:; CALL OCCURS                            ; |10|
           CALL    .S1     banana::_calories() ; |12|
           MVKL    .S2     RL1,B3            ; |12|
           MVKH    .S2     RL1,B3            ; |12|
           ADD     .S1X    8,SP,A4           ; |12|
           NOP             2
RL1:; CALL OCCURS                            ; |12|
           CALL    .S1     banana::~banana()  ; |13|
           STW     .D2T1   A4,*+SP(4)        ; |12|
           ADD     .S1X    8,SP,A4           ; |13|
           MVKL    .S2     RL2,B3            ; |13|
           MVK     .S2     0x2,B4            ; |13|
           MVKH    .S2     RL2,B3            ; |13|
RL2:; CALL OCCURS                            ; |13|
           LDW     .D2T1   *+SP(4),A4        ; |12|
           LDW     .D2T2   *++SP(16),B3      ; |13|
           NOP             4
           RET     .S2     B3                ; |13|
           NOP             5
           ; BRANCH OCCURS                   ; |13|
```

付録 A

用語集

数

3 文字符号系列： ある意味をもつ 3 文字シーケンス（ISO 646-1983 Invariant Code Set によって定義されている）。これらの文字は C の文字セットでは表現できませんが、1 つの文字に展開されます。たとえば、??' という 3 文字符号は ^ に展開されます。

A

ANSI： 米国規格協会を参照

B

.bss セクション： 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/C++ オプティマイザ：「オプティマイザ」を参照

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。コンパイル、アセンブル、および任意のリンクを 1 ステップで実行できるユーティリティ。コンパイラは、コンパイラ（パーサ、オプティマイザ、コード・ジェネレータを含む）、アセンブラ、およびリンカを 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。

COFF： 共通オブジェクト・ファイル・フォーマットを参照

D

.data セクション： 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。

H

Hex 変換ユーティリティ： COFF ファイルを受け取り、そのファイルを EPROM プログラマにロードできる標準 ASCII 16 進数フォーマットの 1 つに変換するプログラム

I

ISO： 国際標準化機構（ISO）。各国の標準化機関の国際的な連合体で、自主的な国際工業規格を設定します。

K

K&R C： カーニハンとリッチーの C。「The C Programming Language（K&R）」の初版で定義された、事実上の標準規格。以前の ANSI 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。

T

.text セクション： デフォルトの COFF セクションの 1 つ。.text セクションは実行可能なコードを含んだ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。アーカイバを使用すると、新しいメンバを追加するだけでなく、アーカイブ・ライブラリのメンバを削除、抽出、または置換することができます。

アーカイブ・ライブラリ： アーカイバによって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んだソース・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コードに換えます。

アセンブリ・オプティマイザ： リニア・アセンブリ・コードを最適化するソフトウェア・プログラム。リニア・アセンブリ・コードは、レジスタが割り当てられたりスケジュールされたりしないアセンブリ・コードです。入力ファイルのいずれかに拡張子 .sa がある場合、アセンブリ・オプティマイザはコンパイラ・プログラム cl6x により自動的に起動されます。

い

インターリスト機能： 元の C/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挿入する機能。C/C++ の文は、それに相当するアセンブリ命令の後ろに挿入されます。

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 1 つの名前付きオブジェクトを指す場合などに行われます。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。

エイリアスの明確化： 2 つ のポインタ式が同じ位置を指すことができない場合を判定する技法。これにより、コンパイラはこれらの式を自由に最適化できるようになります。

エピローグ： スタックの復元および戻りを行う関数のコード部分。「パイプライン・ループ・エピローグ」も参照

エミュレータ： TMS320C6000 の機能をエミュレートするハードウェア開発システム

エントリ・ポイント： ターゲット・メモリでの実行開始点

お

オブジェクト・ファイル： 機械語オブジェクト・コードを含む、アセンブルまたはリンクされたファイル

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たアーカイブ・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

オプティマイザ： C プログラムの実行速度を向上させ、サイズを縮小するソフトウェア・ツール。「アセンブリ・オプティマイザ」も参照

オペランド： アセンブリ言語命令、アセンブラ疑似命令、またはマクロ疑似命令の引数。これにより、命令または疑似命令で実行される操作に対して情報が提供されます。

か

カーネル： パイプライン・ループ・プロローグとパイプライン・ループ・エピローグの間のソフトウェア・パイプライン・ループの本体

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、その定義または宣言が異なるプログラム・モジュールで行われるシンボル

環境変数： ユーザが定義して文字列に割り当てるシステム・シンボル。多くの場合、環境変数はバッチ・ファイル（.cshrc など）に組み込まれます。

関数のインライン展開： 関数のコードを呼び出し点に挿入するプロセス。これにより関数呼び出しのオーバーヘッドが軽減され、オプティマイザは周囲のコードとのコンテキストで関数の最適化を実行できます。

間接呼び出し： 1 つの関数が別の関数を呼び出す関数呼び出し。呼び出し先の関数アドレスを渡すことにより行われます。

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

疑似命令： 特別な目的をもつ複数のコマンド。ソフトウェア・ツールの動作や機能を制御します。

共通オブジェクト・ファイル・フォーマット（COFF）： AT&T の開発した標準に従って構成されるオブジェクト・ファイル・システム。これらのファイルはメモリ空間に再配置できます。

く

組み込み関数： 関数と同じように使用して、アセンブリ言語コードを生成する演算子。この関数を使用しないと C では表現できないか、またはコード化するために多くの時間と労力が必要になる場合があります。

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。

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。

こ

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た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： リンカまたは Hex 変換ユーティリティのオプションが入っており、リンカまたは Hex 変換ユーティリティ用の入力ファイルを指定するファイル

コメント： ソース・ファイルを文書化したり、読みやすくするためのソース文（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・ファイルには何の効果も及ぼしません。

コンパイラ：「C/C++ コンパイラ」を参照

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こと

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

実行時の自動初期化： 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。リンカは、-c オプションを指定して起動された場合にこの方法を使用します。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： プログラムの実行の前に、C のグローバル変数（.cinit セクションに保持されている）を初期化すること

出力セクション： リンクされた実行可能モジュールの中の、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できる、リンクされた実行可能オブジェクト・ファイル

冗長なループ： 同じループの 2 つのバージョン。1 つはソフトウェア・パイプライン・ループで、もう 1 つはパイプラインなしループです。冗長ループは、TMS320C6000 のツールが、トリップ・カウントに対して最大パフォーマンスのためにループをパイプラインするに十分な大きさを保証できないときに生成されます。

初期化されたセクション： 実行可能コードまたはデータを含む 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、または .sect 疑似命令で構成されます。

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。

シンボリック・デバッグ： シンボル情報を保持して、シミュレータやエミュレータなどのデバッグ・ツールがこの情報を使用できるようにするソフトウェア・ツールの機能

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： ファイルで定義して使用されるシンボルについての情報を含んだ COFF オブジェクト・ファイルの部分

す

スタンドアロン・シミュレータ： 実行可能な COFF .out ファイルをロードし、実行するソフトウェア・ツール。C 入出力ライブラリと同時に使用する場合は、スタンドアロン・シミュレータは、スクリーンへの標準出力を含めてすべての C 入出力関数をサポートします。

スタンドアロン・プリプロセッサ： マクロ、#include ファイル、および条件付きコンパイルを独立したプログラムとして展開するソフトウェア・ツール。命令の解析を含む統合的な前処理も実行します。

せ

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。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： コードまたはデータの再配置可能なブロック。最終的にはメモリ・マップ内に連続した空間を占めます。

セクション・ヘッダ： COFF オブジェクト・ファイルの一部分。そのファイルのセクションに関する情報が含まれます。各セクションには専用のヘッダがあり、そこ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。

そ

ソース・ファイル： C/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。

ソフトウェア・パイプライン： ループからの命令をスケジュールしてループ実行の複数の反復を並列に実行できるようにする C/C++ オプティマイザとアセンブリ・オプティマイザの使用する技法

た

ターゲット・システム： 開発されたオブジェクト・コードを実行するシステム

代入文： 値を指定して変数 を初期化する文

ち

直接呼び出し： ある関数が、関数の名前を使用して別の関数を呼び出す関数呼び出し

て

定数： 値を変更できない型

と

統合プリプロセッサ： パーサが組み込まれている C/C++ プリプロセッサで、高速コンパイルが可能です。また、前処理を独立して行ったり、前処理リストを生成することもできます。

動的メモリ割り当て： いくつかの関数（malloc、calloc、realloc など）によって使用される技法。このメモリ割り当てでは、実行時に変数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。

トリップ・カウント： 1 つのループが終了する前に実行する回数

ね

ネーム・マングリング： 関数名を関数の引数や戻り型に関連する情報でエンコードするコンパイラ固有の機能

は

バイト： ANSI C によると、1 文字を含めることのできる最小アドレス可能単位

パーサ： ソース・ファイルの読み取り、前処理機能の実行、構文のチェック、および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する中間ファイルを生成するソフトウェア・ツール

パイプラインループ・エピローグ： ソフトウェアパイプライン・ループのパイプラインの後段階のコード部分。「エピローグ」も参照

パイプラインループ・プロローグ： ソフトウェアパイプライン・ループのパイプラインの前段階のコード部分。「プロローグ」も参照

ひ

ビッグエンディアン： 1 つのワード内で、バイトの番号を左から右の順で付けていくアドレス方式。1 つのワード内の上位のバイトほど、アドレス番号が小さくなります。エンディアンの順序はハードウェア固有のものであり、リセット時に決定されます。「リトルエンディアン」も参照

ふ

ファイル・レベルの最適化： 最適化のレベルの 1 つ。コンパイラは、ファイル全体に関する情報を使用してコードを最適化します（コンパイラがプログラム全体に関する情報を使用してコードを最適化する、プログラムレベルの最適化とは反対の意味をもちます）。

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

プッシュ： スタック上のデータ・オブジェクトを一次記憶装置に格納する操作

プラグマ： コンパイラに対して、特定の文の処理方法について指示を与えるプリプロセッサ疑似命令

プリプロセッサ： マクロ定義の解釈、マクロの展開、ヘッダ・ファイルの解釈、条件付きコンパイルの解釈、およびプリプロセッサ疑似命令の処理を行うソフトウェア・ツール

プログラムレベルの最適化： すべてのソース・ファイルが 1 つの中間ファイルにコンパイルされるときに適用される高度なレベルの最適化。コンパイラはプログラム全体を参照できるので、プログラムレベルの最適化では、ファイルレベルの最適化ではほとんど適用されないいくつかの最適化が実行されます。

ブロック： 中括弧でまとめられた一連の宣言または文

プロローグ： スタックを設定する関数のコード部分。「パイプラインループ・プロローグ」も参照

分割化： データ・パスを各命令に割り当てるプロセス

へ

米国規格協会（ANSI）： 自主的な工業規格を設定する組織

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

ポップ： スタックからデータ・オブジェクトを検索する操作

ま

マクロ： 命令として使用でき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： マクロ呼び出しに代わってソース文をコードに挿入するプロセス

マクロ呼び出し： マクロを起動すること

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を示します。

め

明確化：「エイリアスの明確化」を参照

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。

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 ANSI 標準関数

ランタイムサポート・ライブラリ： ランタイムサポート関数のソースが格納さ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： ユーザのプログラムで機能しなければならないランタイム・パラメータ。これらのパラメータは、メモリおよびレジスタの規則、スタックの編成、関数呼び出し規則、およびシステムの初期化により定義されます。

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）への効果が記述されています。

リトルエンディアン： 1 つのワード内で、バイトの番号を右から左の順に付けていくアドレス指定方式。1 つのワード内で上位のバイトほど、アドレス番号が大きくなります。エンディアンの順序はハードウェア固有のものであり、リセット時に決定されます。「ビッグエンディアン」も参照

リニア・アセンブリ： レジスタが割り当てられたり、スケジュールされたりしないアセンブリ・コード。アセンブリ・オプティマイザ用の入力として使用されます。リニア・アセンブリ・ファイルは .sa 拡張子をもっています。

リブ･アウト： ある手順の中で定義される値。その手順からの出力として使用されます。

リブ・イン： ある手順の前に定義される値。その手順への入力として使用されます。

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、デバイスにより実行できます。

る

ループの展開： 小さなループを展開してループの各反復をコードで表示する最適化。これを使用するとコード･サイズは大きくなりますが、コードの効率が向上します。

ろ

ローダ： 実行可能モジュールをシステム・メモリにロードするデバイス

ロード時の初期化： C/C++ コードをリンクする場合、リンカが使用する自動初期化の方法。リンカは、ユーザが -cr オプションを指定して起動した場合にこの方法を使用します。この方法では、実行時でなくロード時に変数が初期化されます。

わ

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

*、リニア・アセンブリ・ソース内の　4-11
>> 記号　2-35
-@ コンパイラ・オプション　2-15

数

2 の累乗による乗算　9-71
3 文字符号系列
　定義　A-1

A

-a スタンドアロン・シミュレータ・オプション　6-4
-a リンカ・オプション　5-5
-aa アセンブラ・オプション　2-23
abort 関数　9-41
.abs 拡張子　2-19
abs 関数　9-41
-abs リンカ・オプション　5-5
-ac アセンブラ・オプション　2-23
acos 関数　9-42
acosf 関数　9-42
acosh 関数　9-42
acoshf 関数　9-42
acot 関数　9-43
acot2 関数　9-43
acot2f 関数　9-43
acotf 関数　9-43
acoth 関数　9-44
acothf 関数　9-44
-ad アセンブラ・オプション　2-23
add_device 関数　9-7
-ahc アセンブラ・オプション　2-23
-ahc アセンブラ・オプションを使用したファイルのコピー　2-23
-ahi アセンブラ・オプション　2-23
-ahi アセンブラ・オプションを使用したファイルのインクルード　2-23
-al アセンブラ・オプション　2-23
alt.h パス名　2-28
ANSI
　C
　　K&R C との互換性　7-36
　　TMS320C6000 C の相違点　7-2
　定義　A-8
-apd アセンブラ・オプション　2-23
-api アセンブラ・オプション　2-23
-ar リンカ・オプション　5-5
--args リンカ・オプション　5-5, 6-7
-as アセンブラ・オプション　2-24
ASCII 文字列変換関数　9-49
asctime 関数　9-44
asin 関数　9-45
asinf 関数　9-45
asinh 関数　9-45
asinhf 関数　9-45
.asm 拡張子　2-19
asm 文
　最適化されたコード　3-28
　使用方法　8-43
　説明　7-17
assert 関数　9-46
assert.h ヘッダ
　関数のまとめ　9-30
　説明　9-17
atan 関数　9-47
atan2 関数　9-47
atan2f 関数　9-47
atanf 関数　9-47
atanh 関数　9-48
atanhf 関数　9-48
atexit 関数　9-48
atof 関数　9-49
atoi 関数　9-49
atol 関数　9-49
atoll 関数　9-49
-au アセンブラ・オプション　2-24
-ax アセンブラ・オプション　2-24

B

-b スタンドアロン・シミュレータ・オプション　6-4
-b リンカ・オプション　5-5
_BIG_ENDIAN マクロ　2-26

boot.obj 5-8, 5-10
bsearch 関数 9-50
.bss セクション
説明 8-3
定義 A-1
メモリ内への割り振り 5-11
BUFSIZE マクロ 9-25

C

C 10-1
.C 拡張子 2-19
.c 拡張子 2-19
C 言語の識別子 7-2
C 言語の宣言 7-4
C 言語の特性 7-2 ～ 7-4
式 7-3
識別子 7-2
宣言 7-4
定数 7-2
データ型 7-3
プラグマ 7-4
変換 7-3
-c コンパイラ・オプション 2-15, 5-4
C とアセンブリをインターフェイスする 8-23
--c ライブラリ作成ユーティリティ・オプション 10-4
-c リンカ・オプション 5-2, 5-4, 5-10
C++ 言語の特性 7-5
C++ ネーム・デマングラ
説明 1-7
オプション 11-2
起動方法 11-2
説明 11-1
例 11-3
C/C++ 言語
const キーワード 7-7
cregister キーワード 7-8
far キーワード 7-11
interrupt キーワード 7-10
near キーワード 7-11
restrict キーワード 7-14
volatile キーワード 7-15
アセンブラ定数にアクセスする方法 8-45
アセンブラのグローバル変数にアクセスする方法 8-44
アセンブラ文の配置 8-43
アセンブラ変数にアクセスする方法 8-44
アセンブリによるインターリスト 2-46
グローバル・コンストラクタとデストラクタ 5-10
プラグマ疑似命令 7-18
C/C++ コードのコンパイル
オプティマイザによる 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 2-16
前処理後 2-29
C/C++ コンパイラ
説明 1-3
定義 A-1
C/C++ 呼び出し可能プロシージャに値を戻す 4-29
C6X_C_DIR 環境変数 2-27, 2-29
.call アセンブリ・オプティマイザ疑似命令 4-15
calloc 関数 9-78
解放 9-64
説明 9-51
動的メモリ割り当て 8-5
cassert ヘッダ
関数のまとめ 9-30
説明 9-17
C_C6X_OPTION 2-25
cctype ヘッダ
関数のまとめ 9-30
説明 9-17
C_DIR 環境変数 2-27, 2-29
ceil 関数 9-51
ceilf 関数 9-51
cerrno ヘッダ 9-18
cfloat ヘッダ 9-19
.cinit セクション
アセンブリ・モジュールの使用 8-24
自動初期化での使用 5-10
説明 8-2
メモリ内への割り振り 5-11
_c_int00 説明 5-10
.circ アセンブリ・オプティマイザ疑似命令 4-17
ciso646 ヘッダ 9-22
cl6x -z コマンド 5-2
cl6x コマンド 2-4, 5-3
clearerr 関数 9-52
clearerrf 関数 9-52
climits ヘッダ 9-19
CLK_TCK マクロ 9-27
clock 関数 9-52
CLOCKS_PER_SEC マクロ 9-52
説明 9-27
clock_t データ型 9-27
close 入出力関数 9-9
cmath ヘッダ
関数のまとめ 9-31
説明 9-22
_CODE_ACCESS マクロ 9-22
CODE_SECTION プラグマ 7-19
COFF
定義 A-1, A-4
_ _COMPILER_VERSION_ _ マクロ 2-27

const キーワード　7-7
.const セクション　5-11
　説明　8-2
--consultant コンパイラ・オプション　2-15
C_OPTION　2-25
cos 関数　9-53
cosf 関数　9-53
cosh 関数　9-53
coshf 関数　9-53
cot 関数　9-54
cotf 関数　9-54
coth 関数　9-54
cothf 関数　9-54
.cproc アセンブリ・オプティマイザ疑似命令　4-17
-cr リンカ・オプション　5-2, 5-10
cregister キーワード　7-8
csetjmp ヘッダ
　関数とマクロのまとめ　9-34
　説明　9-23
cstdarg ヘッダ
　説明　9-23
　マクロのまとめ　9-34
cstdio ヘッダ
　関数のまとめ　9-34
　説明　9-25
cstdlib ヘッダ
　関数のまとめ　9-37
　説明　9-26
cstring ヘッダ
　関数のまとめ　9-38
　説明　9-26
ctime 関数　9-55
ctime ヘッダ
　関数のまとめ　9-40
　説明　9-27
ctype.h ヘッダ
　関数のまとめ　9-30
　説明　9-17

D

-d コンパイラ・オプション　2-15
-d スタンドアロン・シミュレータ・オプション　6-4
data セクション
　定義　A-1
_DATA_ACCESS マクロ　9-22
DATA_ALIGN プラグマ　7-20
DATA_MEM_BANK プラグマ　7-20
DATA_SECTION プラグマ　7-22
_ _DATE_ _ マクロ　2-27
difftime 関数　9-55
div 関数　9-56
div_t データ型　9-26
DP（データ・ページ・ポインタ）　7-11
DWARF デバッグ・フォーマット　2-18

E

-e リンカ・オプション　5-5
-ea コンパイラ・オプション　2-21
-ec コンパイラ・オプション　2-21
EDOM マクロ　9-18
EFPOS マクロ　9-18
-el コンパイラ・オプション　2-21
.endproc アセンブリ・オプティマイザ疑似命令
　4-17, 4-24
ENOENT マクロ　9-18
-eo コンパイラ・オプション　2-21
EOF クリア関数　9-52
EOF テスト関数　9-59
EOF マクロ　9-25
-ep コンパイラ・オプション　2-21
EPROM プログラマ　1-4
ERANGE マクロ　9-18
errno.h ヘッダ　9-18
exception インクルード・ファイル　9-28
exit 関数
　abort 関数　9-41
　atexit　9-48
　exit 関数　9-57
exp 関数　9-57
exp10 関数　9-58
exp10f 関数　9-58
exp2 関数　9-58
exp2f 関数　9-58
expf 関数　9-57

F

-f スタンドアロン・シミュレータ・オプション　6-4
-f リンカ・オプション　5-5
-fa コンパイラ・オプション　2-20
fabs 関数　9-59
fabsf 関数　9-59
far キーワード　7-11
.far セクション
　説明　8-3
　メモリ内への割り振り　5-11
_FAR_RTS マクロ　9-22
-fb コンパイラ・オプション　2-22
-fc コンパイラ・オプション　2-20
fclose 関数　9-59

feof 関数　9-59
ferror 関数　9-60
-ff コンパイラ・オプション　2-22
fflush 関数　9-60
-fg コンパイラ・オプション　2-20
fgetc 関数　9-60
fgetpos 関数　9-60
fgets 関数　9-61
FILE データ型　9-25
file.h ヘッダ　9-18
FILENAME_MAX マクロ　9-25
_ _FILE_ _ マクロ　2-27
-fl コンパイラ・オプション　2-20
float.h ヘッダ　9-19
floor 関数　9-61
floorf 関数　9-61
fmod 関数　9-62
fmodf 関数　9-62
-fo コンパイラ・オプション　2-20
fopen 関数　9-62
FOPEN_MAX マクロ　9-25
-fp コンパイラ・オプション　2-20
fpos_t データ型　9-25
fprintf 関数　9-63
fputc 関数　9-63
fputs 関数　9-63
-fr コンパイラ・オプション　2-22
fread 関数　9-64
free 関数　9-64
freopen 関数　9-65
frexp 関数　9-65
frexpf 関数　9-65
-fs コンパイラ・オプション　2-22
fscanf 関数　9-66
fseek 関数　9-66
fsetpos 関数　9-66
-ft コンパイラ・オプション　2-22
ftell 関数　9-67
FUNC_CANNOT_INLINE プラグマ　7-23
FUNC_EXT_CALLED プラグマ
　-pm オプションで使用　3-22
　説明　7-23
FUNC_INTERRUPT_THRESHOLD プラグマ　7-24
FUNC_IS_PURE プラグマ　7-25
FUNC_IS_SYSTEM プラグマ　7-25
FUNC_NEVER_RETURNS プラグマ　7-26
FUNC_NO_GLOBAL_ASG プラグマ　7-26
FUNC_NO_IND_ASG プラグマ　7-27
fwrite 関数　9-67

G

-g コンパイラ・オプション　2-18
-g スタンドアロン・シミュレータ・オプション　6-4
-g リンカ・オプション　5-5
getc 関数　9-67
getchar 関数　9-68
getenv 関数　9-68
gets 関数　9-68
gmtime 関数　9-69
gsm.h ヘッダ　9-18

H

-h C++ ネーム・デマングラ・オプション　11-2
-h スタンドアロン・シミュレータ・オプション　6-4
--h ライブラリ作成ユーティリティ・オプション　10-4
-h リンカ・オプション　5-5
-heap リンカ・オプション　5-6, 9-74
-help コンパイラ・オプション　2-15
Hex 変換ユーティリティ
　説明　1-4
　定義　A-2
HUGE_VAL マクロ　9-22

I

-i コンパイラ・オプション　2-16, 2-28
-i スタンドアロン・シミュレータ・オプション　6-4
-i リンカ・オプション　5-6
_IDECL マクロ　9-22
#include
　ファイル
　　検索するディレクトリの追加　2-16
　　検索パスの指定　2-27
　プリプロセッサ疑似命令　2-27
　　インクルードされるファイルのリスト作成　2-30
inline キーワード　2-40
_INLINE プリプロセッサ・シンボル　2-40
_INLINE マクロ　2-26
interrupt キーワード　7-10
INTERRUPT プラグマ　7-27
int_fastN_t 整数型　9-24
int_leastN_t 整数型　9-24
intmax_t 整数型　9-24
INTN_C マクロ　9-24
intN_t 整数型　9-24
intprt_t 整数型　9-24
isalnum 関数　9-69
isalpha 関数　9-69
isascii 関数　9-69

iscntrl 関数　9-69
isdigit 関数　9-69
isgraph 関数　9-69
islower 関数　9-69
ISO
　TMS320C6000 の相違点
　　標準 C　7-2
　　標準 C++　7-5
　定義　A-2
　標準の概要　1-5
iso646.h ヘッダ　9-22
isprint 関数　9-69
ispunch 関数　9-69
isspace 関数　9-69
isupper 関数　9-69
isxdigit 関数　9-69
isxxx 関数　9-17

J

-j リンカ・オプション　5-6
jmp_buf データ型　9-23

K

-k コンパイラ・オプション　2-16
--k ライブラリ作成ユーティリティ・オプション　10-4
K&R C
　ANSI C との互換性　7-36
　関連文献　vi
　定義　A-2
K&R C との互換性　7-36

L

-l ライブラリ作成ユーティリティ・オプション　10-3
-l リンカ・オプション　5-2, 5-8
labs 関数　9-41
_LARGE_MODEL マクロ　2-26
_LARGE_MODEL_OPTION マクロ　2-26
ldexp 関数　9-71
ldexpf 関数　9-71
ldiv 関数　9-56
ldiv_t データ型　9-26
limits.h ヘッダ　9-19
_ _LINE_ _ マクロ　2-27
linkage.h ヘッダ　9-22
_LITTLE_ENDIAN マクロ　2-26
lldiv 関数　9-56
lltoa 関数　9-73
load6x　6-2
localtime 関数　9-71
log 関数　9-72
log10 関数　9-72
log10f 関数　9-72
log2 関数　9-73
log2f 関数　9-73
logf 関数　9-72
long long 除算　9-56
longjmp 関数　9-23, 9-88
lseek 入出力関数　9-10
L_tmpnam マクロ　9-25
ltoa 関数　9-74

M

-m リンカ・オプション　5-6
-ma コンパイラ・オプション　3-25
malloc 関数　9-78
　解放　9-64
　動的メモリ割り当て　8-5
　メモリの割り当て　9-74
.map アセンブリ・オプティマイザ疑似命令　4-20
-map スタンドアロン・シミュレータ・オプション　6-4
math.h ヘッダ
　関数のまとめ　9-31
　説明　9-22
-mb コンパイラ・オプション　2-45
.mdep アセンブリ・オプティマイザ疑似命令　4-21, 4-44
-me コンパイラ・オプション　2-16
memalign 関数　9-75
memchr 関数　9-75
memcmp 関数　9-76
memcpy 関数　9-76
memmove 関数　9-77
memset 関数　9-77
-mh コンパイラ・オプション　3-15
-mi コンパイラ・オプション　2-43
minit 関数　9-78
mk6x　10-3, 11-2
mktime 関数　9-79
-ml コンパイラ・オプション　2-16
-mo コンパイラ・オプション　5-13
modf 関数　9-80
modff 関数　9-80

.mptr アセンブリ・オプティマイザ疑似命令　4-21
-mr コンパイラ・オプション　7-12
-ms コンパイラ・オプション　3-17
-mt コンパイラ・オプション　3-26, 3-27, 4-43
MUST_ITERATE プラグマ　7-28
-mv コンパイラ・オプション　2-16
-mw コンパイラ・オプション　2-16

N

-n コンパイラ・オプション　2-16
_nassert 組み込み関数　8-37
NASSERT マクロ　9-17
NDEBUG マクロ　9-17, 9-46
near 位置に依存しないデータ　8-7
near キーワード　7-11
new ヘッダ　9-28
new_handler 型　9-28
.nfo 拡張子　3-19
NMI_INTERRUPT プラグマ　7-30
.no_mdep アセンブリ・オプティマイザ疑似命令　4-43, 4-23
NULL マクロ　9-24, 9-25

O

-o C++ ネーム・デマングラ・オプション　11-2
-o コンパイラ・オプション　3-2
-o スタンドアロン・シミュレータ・オプション　6-4
-o リンカ・オプション　5-6
.obj 拡張子　2-19
offsetof マクロ　9-24
-oi コンパイラ・オプション　3-29
-ol コンパイラ・オプション　3-18
-on コンパイラ・オプション　3-19
-op コンパイラ・オプション　3-21
open 入出力関数　9-11

P

-pdel コンパイラ・オプション　2-33
-pden コンパイラ・オプション　2-33
-pdf コンパイラ・オプション　2-33
-pdr コンパイラ・オプション　2-33
-pds コンパイラ・オプション　2-33
-pdse コンパイラ・オプション　2-33
-pdsr コンパイラ・オプション　2-33
-pdsw コンパイラ・オプション　2-33
-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 コンパイラ・オプション　7-38
perror 関数　9-80
-pi コンパイラ・オプション　2-39
.pinit セクション　5-11
-pk コンパイラ・オプション　7-36, 7-38
-pm コンパイラ・オプション　3-20
pow 関数　9-81
powf 関数　9-81
powi 関数　9-81
powif 関数　9-81
-ppl コンパイラ・オプション　2-30
-ppa コンパイラ・オプション　2-29
-ppc コンパイラ・オプション　2-30
-ppd コンパイラ・オプション　2-30
-ppi コンパイラ・オプション　2-30
-ppo コンパイラ・オプション　2-29
-pr コンパイラ・オプション　7-38
#pragma 疑似命令　7-4
.pref アセンブリ・オプティマイザ疑似命令　4-23
printf 関数　9-82
-priority リンカ・オプション　5-6
PROB_ITERATE プラグマ　7-30
.proc アセンブリ・オプティマイザ疑似命令　4-24
--profile:breakpt コンパイラ・オプション　2-18
-ps コンパイラ・オプション　7-38
ptrdiff_t　7-3
ptrdiff_t データ型　9-24
putc 関数　9-82
putchar 関数　9-82
puts 関数　9-83
-px コンパイラ・オプション　2-35

Q

-q コンパイラ・オプション　2-16
-q スタンドアロン・シミュレータ・オプション　6-4
--q ライブラリ作成ユーティリティ・オプション　10-4
qsort 関数　9-83

R

-r スタンドアロン・シミュレータ・オプション　6-5
-r リンカ・オプション　5-6
rand 関数　9-84
RAND_MAX マクロ　9-26
read 入出力関数　9-12

realloc 関数　8-5, 9-78
　解放　9-64
　ヒープ・サイズの変更　9-84
.reg アセンブリ・オプティマイザ疑似命令　4-26
.rega アセンブリ・オプティマイザ疑似命令　4-28
.regb アセンブリ・オプティマイザ疑似命令　4-28
register 記憶クラス　7-4
remove 関数　9-85
rename 関数　9-85
rename 入出力関数　9-12
.reserve アセンブリ・オプティマイザ疑似命令　4-28
restrict キーワード　7-14
.return アセンブリ・オプティマイザ疑似命令　4-29
-rev スタンドアロン・シミュレータ・オプション　6-5
rewind 関数　9-86
round 関数　9-86
roundf 関数　9-86
rsqrt 関数　9-87
rsqrtf 関数　9-87
--rts ライブラリ作成ユーティリティ・オプション　10-4

S

-s オプション
　コンパイラ　2-46
　リンカ　5-6
.s 拡張子　2-19
-s コンパイラ・オプション　2-17
-s スタンドアロン・シミュレータ・オプション　6-5
.sa 拡張子　2-19
SAT ビットの副次作用　8-42
scanf 関数　9-87
SEEK_CUR マクロ　9-25
SEEK_END マクロ　9-25
SEEK_SET マクロ　9-25
setbuf 関数　9-87
setjmp マクロ　9-23, 9-88
setjmp.h ヘッダ
　関数とマクロのまとめ　9-34
　説明　9-23
set_new_handler 関数　9-28
setvbuf 関数　9-89
SIMD
　_nassert を使用した使用可能化　8-37
sin 関数　9-90
sinf 関数　9-90
sinh 関数　9-90
sinhf 関数　9-90
size_t　7-3
size_t データ型　9-24, 9-25
_SMALL_MODEL マクロ　2-27
sprintf 関数　9-91
sqrt 関数　9-91
sqrtf 関数　9-91
srand 関数　9-84
-ss コンパイラ・オプション　2-17, 3-30
sscanf 関数　9-91
STABS デバッグ・フォーマット　2-18
.stack セクション
　説明　8-3
　メモリ内への割り振り　5-11
-stack リンカ・オプション　5-7
_ _STACK_SIZE
　使用方法　8-4
stdarg.h ヘッダ
　説明　9-23
　マクロのまとめ　9-34
_ _STDC_ _ マクロ　2-27
stddef.h ヘッダ　9-24
stden マクロ　9-25
stdin マクロ　9-25
stdexcept インクルード・ファイル　9-28
stdint.h ヘッダ　9-24
stdio.h ヘッダ
　関数のまとめ　9-34
　説明　9-25
stdlib.h ヘッダ
　関数のまとめ　9-37
　説明　9-26
stdout マクロ　9-25
strcat 関数　9-92
strchr 関数　9-93
strcmp 関数　9-93
strcoll 関数　9-93
strcpy 関数　9-94
strcspn 関数　9-95
strerror 関数　9-95
strftime 関数　9-96
string.h ヘッダ
　関数のまとめ　9-38
　説明　9-26
strlen 関数　9-97
strncat 関数　9-98
strncmp 関数　9-99
strncpy 関数　9-100
strpbrk 関数　9-101
strrchr 関数　9-101
strspn 関数　9-102
strstr 関数　9-102
strtod 関数　9-103
strtok 関数　9-104
strtol 関数　9-103
strtoll 関数　9-103
strtoul 関数　9-103

strtoull 関数　9-103
STRUCT_ALIGN プラグマ　7-31
struct_tm データ型　9-27
strxfrm 関数　9-105
STYP_CPY フラグ　5-11
.switch セクション
　説明　8-2
　メモリ内への割り振り　5-11
--symdebug
　coff コンパイラ・オプション　2-18
　dwarf コンパイラ・オプション　2-18
　none コンパイラ・オプション　2-18
　skeletal コンパイラ・オプション　2-18
.sysmem セクション
　説明　8-3
　メモリ内への割り振り　5-11
_SYSMEM_SIZE　8-5

T

-t スタンドアロン・シミュレータ・オプション　6-5
tan 関数　9-105
tanf 関数　9-105
tanh 関数　9-106
tanhf 関数　9-106
.text セクション
　説明　8-2
　定義　A-2
　メモリ内への割り振り　5-11
_TI_ENHANCED_MATH_H シンボル　9-23
time 関数　9-106
_ _TIME_ _ マクロ　2-27
time.h ヘッダ
　関数のまとめ　9-40
　説明　9-27
time_t データ型　9-27
tmpfile 関数　9-107
TMP_MAX マクロ　9-25
tmpnam 関数　9-107
_TMS320C6200 マクロ　2-26
_TMS320C6400 マクロ　2-26
_TMS320C6700 マクロ　2-26
_TMS320C6X マクロ　2-26
toascii 関数　9-107
tolower 関数　9-108
toupper 関数　9-108
--trampolines リンカ・オプション　5-7
.trip アセンブリ・オプティマイザ疑似命令　4-30
trunc 関数　9-108
truncf 関数　9-108
type_info 構造体　9-28
typeinfo ヘッダ　9-28

U

-u C++ ネーム・デマングラ・オプション　11-2
-u コンパイラ・オプション　2-17
--u ライブラリ作成ユーティリティ・オプション　10-4
-u リンカ・オプション　5-7
uint_fastN_t 符号なし整数型　9-24
uint_leastN_t 符号なし整数型　9-24
uintmax_t 符号なし整数型　9-24
UINTN_C マクロ　9-24
uintN_t 符号なし整数型　9-24
uintprt_t 符号なし整数型　9-24
ungetc 関数　9-109
unlink 入出力関数　9-13
UNROLL プラグマ　7-32

V

-v C++ ネーム・デマングラ・オプション　11-2
--v ライブラリ作成ユーティリティ・オプション　10-4
va_arg マクロ　9-23, 9-109
va_end マクロ　9-23, 9-109
va_list データ型　9-23
va_start マクロ　9-23, 9-109
vfprintf 関数　9-110
.volatile アセンブリ・オプティマイザ疑似命令　4-32
volatile キーワード　7-15
vprintf 関数　9-110
vsprintf 関数　9-111

W

-w リンカ・オプション　5-7
write 入出力関数　9-13

X

-x リンカ・オプション　5-7
--xml_link_info リンカ・オプション　5-7

Y

y/x の逆タンジェント　9-47

Z

-z コンパイラ・オプション　2-4, 2-17
　　-c コンパイラ・オプションで上書き　5-4
-z スタンドアロン・シミュレータ・オプション　6-5

あ

アーカイバ
　　説明　1-3
　　定義　A-2
アーカイブ・ライブラリ
　　定義　A-2
　　リンク　5-8
アーク
　　コサイン関数　9-42
　　コタンジェント
　　　極関数　9-43
　　　デカルト関数　9-43
　　　ハイパボリック関数　9-44
　　サイン関数　9-45
　　タンジェント
　　　極関数　9-47
　　　デカルト関数　9-47
　　　ハイパボリック関数　9-48
アセンブラ
　　オプションのまとめ　2-13
　　コンパイラによる制御　2-23
　　説明　1-3
　　定義　A-2
アセンブリ言語
　　C/C++ コードによるインターリスト　2-46
　　アクセス方法
　　　グローバル変数　8-44
　　　定数　8-45
　　　変数　8-44
　　埋め込み方法　7-17
　　組み込み関数を使用して呼び出す　8-26
　　コードをインターフェイスする　8-23
　　出力の保存　2-16
　　含む　8-43
　　モジュールのインターフェイス　8-23
　　割り込みルーチン　8-47
アセンブリ・オプティマイザ
　　疑似命令のまとめ　4-13
　　起動方法　4-4
　　使用方法　4-1
　　説明　1-3
　　定義　A-2
アセンブリ・オプティマイザ疑似命令
　　.call　4-15
　　.circ　4-17
　　.cproc　4-17
　　.endproc　4-17, 4-24
　　.map　4-20
　　.mdep　4-21
　　.mptr　4-21
　　.no_mdep　4-23
　　.pref　4-23
　　.proc　4-24
　　.reg　4-26
　　.rega　4-28
　　.regb　4-28
　　.reserve　4-28
　　.return　4-29
　　.trip　4-30
　　.volatile　4-32
アセンブリ・ソース・デバッグ　2-18
アセンブリ・リスト・ファイルの作成　2-23

い

位置合わせされていないデータと 64 ビット値の使用方法　8-36
一時ファイル作成関数　9-107
位置に依存しないデータ　8-7
インターリスト・ユーティリティ
　　オプティマイザでの使用　3-30
　　コンパイラによる起動　2-17, 2-46
　　説明　1-3
　　定義　A-3
インライン
　　アセンブリ言語　8-43
　　関数展開　2-38
　　関数の宣言　2-40
　　組み込み演算子　2-38
　　自動展開　3-29
　　制約事項　2-42
　　定義制御　2-40
　　保護されない定義制御　2-39
　　無効化　2-39

え

エイリアス指定
　　定義　A-3

エイリアス指定技法　3-25
　関数からのアドレスの戻り　3-25
　グローバル変数へのアドレスの割り当て　3-25
　特定の技法を使用しないことを示す　3-26
エイリアスの明確化
　説明　3-38
　定義　A-3
エスケープ・シーケンス　7-2, 7-37
エピローグ
　定義　A-3
エピローグの縮小　3-14
　見込み実行　3-14
エピローグを縮小する　3-14
　見込み実行　3-14
エピローグを除去する
　積極的に　3-15
エミュレータ
　定義　A-3
エラー
　errno.h ヘッダ・ファイル　9-18
　標識関数　9-52
　マップ関数　9-80
　メッセージ
　　オプションによる処理　2-34
　　⇒診断メッセージも参照
　　プリプロセッサ　2-26
　メッセージ・マクロ　9-30
エラー・テスト関数　9-60
エントリ・ポイント
　定義　A-3

お

オブジェクトの格納関数　9-60
オブジェクト・ファイル
　定義　A-3
オブジェクト・ライブラリ
　コードとのリンク方法　9-2
　定義　A-3
オプション
　C++ ネーム・デマングラ　11-2
　アセンブラ　2-23
　規則　2-5
　コンパイラ・シェルのまとめ　2-6
　診断　2-11, 2-33
　スタンドアロン・シミュレータ　6-4
　定義　A-3
　プリプロセッサ　2-10, 2-29
　ライブラリ作成ユーティリティ　10-4
　リンカ　5-5
オプティマイザ
　オプションのまとめ　2-12
　コンパイラ・オプションによる起動　3-2
　説明　1-3
　定義　A-3
オペランド
　定義　A-3

か

カーネル
　説明　3-4
　定義　A-3
開発フローのダイヤグラム　1-2
外部シンボル
　定義　A-3
外部宣言　7-37
書き込み関数
　fprintf　9-63
　fputc　9-63
　fputs　9-63
　printf　9-82
　putc　9-82
　putchar　9-82
　puts　9-83
　sprintf　9-91
　ungetc　9-109
　vfprintf　9-110
　vprintf　9-110
　vsprintf　9-111
拡張子
　abs　2-19
　asm　2-19
　C　2-19
　c　2-19
　cc　2-19
　cpp　2-19
　cxx　2-19
　nfo　3-19
　obj　2-19
　s　2-19
　sa　2-19, 4-4
　指定方法　2-21
可変引数マクロ
　使用法　9-109
　説明　9-23
　のまとめ　9-34
加法の浮動小数点演算の並べ換えの防止　3-28
カレンダ時
　ctime 関数　9-55
　difftime 関数　9-55
　mktime 関数　9-79

カレンダ時（続き）
time 関数 9-106
説明 9-27
環境情報関数 9-68
環境変数
C6X_C_DIR 2-27, 2-29
C_DIR 2-27, 2-29
定義 A-4
関数
アルファベット順の参照 9-41
インライン展開 2-38, 3-42
インライン展開の定義 A-4
構造 8-19
サブセクション 5-13
汎用ユーティリティ 9-26, 9-37
プロトタイプ
-pk オプションの効果 7-36
呼び出し
.call アセンブリ・オプティマイザ疑似命令を使用 4-15
規則 8-19
スタックの使用 8-4
通常の呼び出しのバイパス 9-23
呼び出し先の関数の責務 8-20
呼び出し元の関数の責務 8-19
間接呼び出し
定義 A-4
関連文献 v, vi
緩和 ANSI モード 7-38
緩和 ISO モード 7-38

き

キーワード
const 7-7
cregister 7-8
far 7-11
inline 2-40
interrupt 7-10
near 7-11
restrict 7-14
volatile 7-15
記憶クラス
定義 A-4
疑似命令
アセンブリ・オプティマイザ 4-13
定義 A-4
疑似乱整数生成関数 9-84
規則
関数呼び出し 8-19
表記の iv
レジスタ 8-17
起動方法
C++ ネーム・デマングラ 11-2
コンパイラ 2-4
スタンドアロン・シミュレータ 6-2
ライブラリ作成ユーティリティ 10-3
リンカ
コンパイラを使用 5-2
逆平方根関数 9-87
強度換算の最適化 3-43
切り上げ関数 9-51
切り捨て関数 9-108

く

組み込み C++ モード 7-38
組み込み関数
アセンブリ言語文を呼び出すための使用方法 8-26
演算子のインライン展開 2-38
定義 A-4
グリニッジ標準時関数 9-69
グレゴリオ暦 9-27
グローバル変数
C/C++ からアセンブラ変数にアクセスする方法 8-44
確保された空間 8-2
自動初期化 8-52
初期化 7-34
グローバル・コンストラクタとデストラクタ 5-10
グローバル・シンボル
定義 A-4
クロスリファレンス・ユーティリティ 1-4
クロスリファレンス・リスト
アセンブラで生成 2-24
コンパイラで生成 2-35
定義 A-4

け

警告メッセージ 2-31
検索 9-50
厳密 ANSI モード 7-38
厳密 ISO モード 7-38

こ

構造体
定義 A-4

構造体のメンバ　7-4
コード・サイズの縮小　3-5, 3-17
コード・ジェネレータ
　定義　A-4
コサイン関数　9-53
コストに基づいたレジスタ割り当ての最適化　3-36
コタンジェント
　極関数　9-54
　ハイパボリック関数　9-54
コマンド・ファイル
　コマンド行への追加　2-15
　定義　A-4
　リンカ　5-12
コメント
　定義　A-4
　リニア・アセンブリ　4-6
　リニア・アセンブリ・ソース・コード内　4-11
コンパイラ
　オプション
　　アセンブラ　2-13
　　オプティマイザ　2-12
　　型チェック　2-9
　　規則　2-5
　　コンパイラ　2-6
　　出力ファイル　2-8
　　シンボリック・デバッグ　2-7
　　入力ファイル　2-8
　　入力ファイル拡張子　2-7
　　パーサ　2-10
　　非推奨　2-24
　　プロファイル作成　2-7
　　まとめ　2-6
　　リンカ　2-14
　オプティマイザ　3-2
　概要　1-5, 2-2
　起動方法　2-4
　コンパイラ・コンサルタント・アドバイス・ツール　2-15
　使用頻度の高いオプション　2-15
　診断オプション　2-33
　診断メッセージ　2-31
　セクション　5-11
　説明　2-1
　定義　A-5
　プリプロセッサ・オプション　2-29
コンパイラにより生成されるリンク名　7-33

さ

最適化
　インライン展開　3-42
最適化（続き）
　エイリアスの明確化　3-38
　強度換算　3-43
　コストに基づいたレジスタ割り当て　3-36
　式の簡略化　3-41
　情報ファイル・オプション　3-19
　制御フローの簡略化　3-38
　データ・フロー　3-41
　ファイル・レベル
　　説明　3-18
　　定義　A-7
　プログラム・レベル
　　説明　3-20
　　定義　A-8
　分岐　3-38
　誘導変数　3-43
　リスト　3-35
　ループの循環　3-44
　ループ不変コードの移動　3-44
　レジスタのターゲッティング　3-44
　レジスタのトラッキング　3-44
　レジスタ変数　3-44
　レベル　3-2
　レベルの制御方法　3-21
最適化コード
　デバッグ　3-33
　プロファイル作成　3-33
最適化されたコードのプロファイル方法　3-34
再配置
　定義　A-5
サイン関数　9-90
三角関数　9-22
算術演算　8-48

し

シェル・プログラム
　⇒コンパイラを参照
時間関数
　asctime 関数　9-44
　clock 関数　9-52
　ctime 関数　9-55
　difftime 関数　9-55
　gmtime 関数　9-69
　localtime 関数　9-71
　mktime 関数　9-79
　strftime 関数　9-96
　time 関数　9-106
　説明　9-27
　のまとめ　9-40

式　7-3
　C 言語　7-3
　簡略化　3-41
　定義　A-5
式テスト関数　9-46
指数関数
　exp 関数　9-57
　exp10 関数　9-58
　exp10f 関数　9-58
　exp2 関数　9-58
　exp2f 関数　9-58
　expf 関数　9-57
　説明　9-22
システムの初期化
　初期化テーブル　8-53
　説明　8-51
システムの制約
　_SYSMEM_SIZE　8-5
システム・スタック　8-4
事前初期化された変数
　グローバルと静的　7-34
自然対数関数　9-72
実行可能モジュール
　定義　A-5
実行時の初期化
　変数　8-5
　リンク・プロセス　5-9
実装が定義する動作　7-2
自動初期化
　型　5-10
　実行時
　　説明　8-56
　　定義　A-5
　初期化テーブル　8-53
　定義　A-5
　変数の　8-5, 8-52
シフト　7-3
ジャンプ関数　9-34
ジャンプ・マクロ　9-34
ジャンプ（非ローカル）関数　9-88
出力
　セクション
　　定義　A-5
　ファイル・オプションのまとめ　2-8
　ファイルの概要　1-6
　モジュール
　　定義　A-5
小数部と指数部の関数　9-65
冗長なループ
　説明　3-16
　定義　A-5
剰余　7-3
常用対数関数　9-72, 9-73
初期化
　型　5-10
　変数の　7-34
　　実行時　8-5
　　ロード時　8-5
　ロード時
　　説明　8-57
　　定義　A-10
初期化されたセクション
　説明　8-2
　定義　A-5
　メモリ内の割り当て　5-11
初期化されないセクション
　定義　A-5
　メモリ内の割り当て　5-11
　リスト　8-3
初期化テーブル　8-53
除算　7-3
除算関数　9-56
診断メッセージ
　assert 関数　9-46
　エラー　2-31
　警告　2-31
　形式　2-31
　制御方法　2-33
　生成　2-33
　説明　2-31, 9-17
　その他のメッセージ　2-35
　致命的エラー　2-31
　注釈　2-31
　抑止　2-33
診断メッセージの制御方法　2-33
診断メッセージの抑止方法　2-33
進捗情報の抑止　2-16
シンボリック・デバッグ
　DWARF フォーマットの使用　2-18
　STABS フォーマットの使用　2-18
　定義　A-5
　無効化　2-18
シンボル
　大文字小文字の区別　2-23
　定義　A-5
シンボル・テーブル
　定義　A-5
　ラベルの作成　2-24

す

スタック
　確保された空間　8-3
　ポインタ　8-4

スタンドアロン・シミュレータ　6-1 ～ 6-10
　オプション　6-4
　起動　6-2
　定義　A-6
　引数を格納するためにターゲット・メモリを確保する方法　6-7
　プログラムに引数を渡す方法　6-6 ～ 6-7
　プロファイリング　6-8
スタンドアロン・シミュレータのプロファイル機能　6-8
スタンドアロン・プリプロセッサ　A-6
スモール・メモリ・モデル　8-6

せ

制御フローの簡略化　3-38
制限レジスタ
　C/C++ からのアクセス方法　7-8
制限値
　整数型　9-19
　浮動小数点型　9-20
整数の除算　9-56
生成方法
　#include ファイルのリスト　2-30
　シンボリック・デバッグ疑似命令　2-18
　リンク名　7-33
静的変数
　初期化　7-34
　定義　A-6
セクション
　.bss　8-3
　.cinit　8-2
　.const　8-2
　.far　8-3
　.stack　8-3
　.switch　8-2
　.sysmem　8-3
　.text　8-2
　コンパイラが作成する　5-11
　初期化されない　8-3
　初期化される　8-2
　説明　8-2
　定義　A-6
　定義済みヘッダ　A-6
　メモリの割り当て　5-11
絶対値
　abs/labs 関数　9-41
　fabs 関数　9-59
　fabsf 関数　9-59
絶対リスト
　作成方法　2-23
宣言
　.circ 疑似命令によるサーキュラ・アドレッシング　4-17
　揮発性としてのメモリ参照　4-32
　リニア・アセンブリの変数　4-26

そ

ソース・ファイル
　拡張子　2-20
　定義　A-6
ソフトウェア開発ツールの概要　1-2
ソフトウェア・パイプライン化
　C コード　3-4
　アセンブリ・オプティマイザ・コード　4-4
　情報　3-5
　説明　3-4
　定義　A-6
　無効化　3-5

た

ターゲット・システム
　定義　A-6
代入文
　定義　A-6
タンジェント関数　9-105, 9-106

ち

地方時
　カレンダ時の地方時への変換　9-55
　詳細時刻から地方時への変換　9-79
　説明　9-27
注釈　2-31
直接呼び出し
　定義　A-6

て

底が 10 の対数　9-72
底が 2 の対数　9-73
定義
　リニア・アセンブリ内の C/C++ 呼び出し可能関数　4-17
　リニア・アセンブリのプロシージャ　4-24

定数
C 言語 7-2
C/C++ からアセンブラ定数にアクセスする方法 8-45
定義 A-6
文字定数内のエスケープ・シーケンス 7-37
文字列 7-37, 8-16
定数を定義解除する 2-17, 2-24
ディレクトリ
include ファイル 2-16, 2-28
include ファイルの代替 2-28
指定方法 2-22
低レベル入出力関数 9-18
データ型
C 言語 7-3
clock_t 9-27
div_t 9-26
FILE 9-25
fpos_t 9-25
jmp_buf 9-23
ldiv_t 9-26
ptrdiff_t 9-24
size_t 9-24, 9-25
struct_tm 9-27
time_t 9-27
va_list 9-23
記憶域 8-8
char データ型と short データ型（符号付きと符号なし） 8-9
double と long double（符号付きと符号なし） 8-13
enum、float、および int データ型（符号付きと符号なし） 8-10
long long データ型（符号付きと符号なし） 8-11, 8-12
構造体と配列 8-13
データ・メンバへのポインタ 8-14
メンバ関数へのポインタ 8-14
メモリ内の保存方法 8-8
リスト 7-6
データ・オブジェクト表記 8-8
データ・フローの最適化 3-41
データ・ブロックの書き込み関数 9-67
データ・ページ・ポインタ（DP） 7-11
デバイス
関数 9-7
追加方法 9-14
デバッグ
最適化コード 3-33

と

統合プリプロセッサ
定義 A-7
動的メモリ割り当て
説明 8-5
定義 A-7
トークン 9-104
トリップ・カウント
説明 3-16
定義 A-7

な

夏時間 9-27

に

入出力
関数
close 9-9
lseek 9-10
open 9-11
read 9-12
rename 9-12
unlink 9-13
write 9-13
バッファのフラッシュ 9-60
関数のまとめ 9-34
実装の概要 9-5
説明 9-4
低レベル定義 9-18
デバイスの追加方法 9-14
入出力定義 9-18
入出力バッファのフラッシュ関数 9-60
入力ファイル
オプションのまとめ 2-8
デフォルトの拡張子 2-19
デフォルトの拡張子の変更 2-21
ファイル名の解釈方法の変更 2-20
拡張子
オプションのまとめ 2-7

ね

ネーム・マングリング
定義 A-7

は

パーサ
　オプションのまとめ　2-10
　定義　A-7
バイト
　定義　A-7
バイトの最初の出現を検出する関数　9-75
ハイパボリック算術関数
　説明　9-22
　ハイパボリック・アーク・コサイン関数　9-42
　ハイパボリック・アーク・コタンジェント関数　9-44
　ハイパボリック・アーク・サイン関数　9-45
　ハイパボリック・アーク・タンジェント関数　9-48
　ハイパボリック・コサイン関数　9-53
　ハイパボリック・コタンジェント関数　9-54
　ハイパボリック・サイン関数　9-90
　ハイパボリック・タンジェント関数　9-106
パイプラインループ・エピローグ
　定義　A-7
　説明　3-4
パイプラインループ・プロローグ
　説明　3-4
　定義　A-7
配列
　検索関数　9-50
　ソート関数　9-83
配列をソートする関数　9-83
パック・データの最適化に関して　2-45
バッファ
　指定関数　9-87
　定義および関連付け関数　9-89
パラメータ
　レジスタ・パラメータのコンパイル　7-16
汎用ユーティリティ関数
　minit　9-78
汎用レジスタ
　32 ビット・データ　8-9, 8-10
　40 ビット・データ　8-11
　64 ビット・データ　8-12
　ハーフワード　8-9
　倍精度浮動小数点データ　8-13

ひ

ヒープ
　位置合わせ関数　9-75
　確保された空間　8-3
　説明　8-5
ヒープ・サイズ関数　9-84
引数
　アクセス方法　8-22
ビッグエンディアン
　作成　2-16
　定義　A-7
ビット・フィールド　7-4
　サイズと型　7-38
　割り当て　8-15
表記規則　iv
非ローカル・ジャンプ関数　9-34
非ローカル・ジャンプ関数とマクロ
　説明　9-88
　のまとめ　9-34

ふ

ファイル
　インクルード　2-23
　コピー　2-23
　除去関数　9-85
　名前を変更する関数　9-85
ファイル位置取得関数　9-67
ファイル位置の設定関数
　fseek 関数　9-66
　fsetpos 関数　9-66
ファイル位置標識設定関数　9-86
ファイル名
　拡張子規則　2-20
　指定方法　2-19
　生成関数　9-107
ファイルレベルの最適化　3-18
　定義　A-7
ファイル・オープン関数　9-62, 9-65
ファイル・クローズ関数　9-59
符号付き整数関数と符号付き小数関数　9-80
符号なし
　定義　A-7
プッシュ
　定義　A-7
浮動小数点
　関数のまとめ　9-31 ～ 9-33
　算術関数　9-22
　剰余関数　9-62
プラグマ
　定義　A-8
プラグマ疑似命令　7-18
　CODE_SECTION　7-19
　DATA_ALIGN　7-20
　DATA_MEM_BANK　7-20
　DATA_SECTION　7-22
　FUNC_CANNOT_INLINE　7-23

プラグマ疑似命令（続き）
FUNC_EXT_CALLED 7-23
FUNC_INTERRUPT_THRESHOLD 7-24
FUNC_IS_PURE 7-25
FUNC_IS_SYSTEM 7-25
FUNC_NEVER_RETURNS 7-26
FUNC_NO_GLOBAL_ASG 7-26
FUNC_NO_IND_ASG 7-27
INTERRUPT 7-27
MUST_ITERATE 7-28
NMI_INTERRUPT 7-30
PROB_ITERATE 7-30
STRUCT_ALIGN 7-31
UNROLL 7-32
プリプロセッサ
C 言語の疑似命令 7-4
_INLINE シンボル 2-40
エラー・メッセージ 2-26
オプション 2-29
シンボル 2-26
制御方法 2-26
定義 A-8
定数 name の事前定義 2-15
プログラム終了関数
abort 関数 9-41
atexit 関数 9-48
exit 関数 9-57
プログラムレベルの最適化
実施 3-20
制御方法 3-21
定義 A-8
プロセッサ時間関数 9-52
ブロック
コピー関数
メモリをオーバーラップしない 9-76
メモリをオーバーラップする 9-77
定義 A-8
メモリの割り当て 5-11
プロローグ
定義 A-8
プロローグの縮小 3-14
見込み実行 3-14
プロローグを縮小する 3-14
見込み実行 3-14
分割化
定義 A-8
分岐の最適化 3-38
文献 v, vi

へ

平方根関数 9-91
ヘッダ・ファイル
typeinfo ヘッダ 9-28
assert.h ヘッダ 9-17
cassert ヘッダ 9-17
cctype ヘッダ 9-17
cerrno ヘッダ 9-18
cfloat ヘッダ 9-19
ciso646 9-22
climits ヘッダ 9-19
cmath ヘッダ 9-22
csetjmp ヘッダ 9-23
cstdarg ヘッダ 9-23
cstdio ヘッダ 9-25
cstdlib ヘッダ 9-26
cstring ヘッダ 9-26
ctime ヘッダ 9-27
ctype.h ヘッダ 9-17
errno.h ヘッダ 9-18
file.h ヘッダ 9-18
float.h ヘッダ 9-19
gsm.h ヘッダ 9-18
iso646.h 9-22
limits.h ヘッダ 9-19
linkage.h ヘッダ 9-22
math.h ヘッダ 9-22
new ヘッダ 9-28
setjmp.h ヘッダ 9-23
stdarg.h ヘッダ 9-23
stddef.h ヘッダ 9-24
stdint.h 9-24
stdio.h ヘッダ 9-25
stdlib.h ヘッダ 9-26
string.h ヘッダ 9-26
time.h ヘッダ 9-27
リスト 9-16
ヘルプ関数の位置合わせ 9-75
変換 7-3
ASCII への関数 9-107
C 言語 7-3
大文字小文字関数 9-108
時間から文字列への変換 9-44
説明 9-17
長整数を ASCII へ 9-74
倍長整数を ASCII に 9-73
文字列から数値へ 9-49
変数
C/C++ からアセンブラ変数にアクセスする方法 8-44
自動初期化 8-52
初期化
グローバル 7-34
静的 7-34
定義 A-8
レジスタ変数のコンパイル 7-16

変数（続き）
ローカル変数へのアクセス方法　8-22
変数をレジスタにセットで割り当てる　4-23
変数をレジスタに割り当てる　4-20

ほ

ポインタの組み合わせ　7-37
保護されない定義制御のインライン展開　2-39
ポップ
定義　A-8

ま

前処理リスト・ファイル
#line 疑似命令付き生成　2-30
アセンブリ依存行　2-23
アセンブリ・インクルード・ファイル　2-23
コメント付き生成　2-30
生情報の生成　2-36
マクロ
BUFSIZ　9-25
CLK_TCK　9-27
CLOCKS_PER_SEC　9-27, 9-52
_CODE_ACCESS　9-22
_DATA_ACCESS　9-22
EOF　9-25
_FAR_RTS　9-22
FILENAME_MAX　9-25
FOPEN_MAX　9-25
HUGE_VAL　9-22
_IDECL　9-22
INTN_C　9-24
L_tmpnam　9-25
NASSERT　9-17
NDEBUG　9-17, 9-46
NULL　9-24, 9-25
offsetof　9-24
RAND_MAX　9-26
SEEK_CUR　9-25
SEEK_END　9-25
SEEK_SET　9-25
setjmp　9-23, 9-88
stden　9-25
stdin　9-25
stdout　9-25
TMP_MAX　9-25
UINTN_C　9-24
va_arg　9-109
va_end　9-109
va_start　9-109
マクロ（続き）
アルファベット順の参照　9-41
事前定義名　2-26
定義　A-8
展開　2-26
マクロ定義
定義　A-8
マクロ展開
定義　A-8
マクロ呼び出し
定義　A-8
マップ・ファイル
定義　A-8
マルチバイト文字　7-2
丸め関数　9-86

み

見出しの抑止　2-16

む

無効化
最適化情報ファイル　3-19
自動インライン展開　3-29
条件付きリンク　5-6
シンボリック・デバッグ　2-18
シンボリック・デバッグ情報のマージ　5-5
ソフトウェア・パイプライン化　3-5
定義制御のインライン展開　2-40
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の最適化　3-34
リンク　5-4

め

メモリ依存関係　4-43, 4-44
例外　4-43
メモリ解放関数　9-64
メモリ管理関数
calloc　9-51
free　9-64
malloc　9-74
minit　9-78
realloc　9-84

メモリ参照
アセンブリ・オプティマイザによるデフォルトの処理 4-43
注釈付け 4-44
メモリ内での値のコピー関数 9-77
メモリ比較関数 9-76
メモリ割り当て
セクション 5-11
メモリの割り当てとクリア関数 9-51
メモリ割り当て関数 9-74
メモリ・エイリアス指定 4-43
例 4-46
メモリ・エイリアスの明確化 4-43
メモリ・バンク 4-33
.mptr による競合の防止 4-21
メモリ・バンク体系（インターリーブド） 4-33
4 バンク・メモリ 4-33
2 つのメモリ空間がある 4-34
メモリ・プール
malloc 関数 9-74
確保された空間 8-3
メモリ・マップ
定義 A-8
メモリ・モデル
スタック 8-4
スモール・メモリ・モデル 8-6
セクション 8-2
説明 8-2
動的メモリ割り当て 8-5
変数の初期化 8-5
ラージ・メモリ・モデル 8-6

も

文字
一致関数
strpbrk 9-101
strrchr 9-101
strspn 9-102
エスケープ・シーケンス 7-37
型テスト関数 9-69
検出関数 9-93
セット 7-2
不一致関数 9-95
変換関数
説明 9-17
のまとめ 9-30
文字数 9-105
文字列定数 8-16
読み込み関数
単一文字 9-60
複数文字 9-61
文字列関数 9-26, 9-38
一致 9-102
コピー 9-100
トークンへの分割 9-104
長さ 9-97
比較
任意の文字数 9-99
文字列全体 9-93
変換 9-103
文字列エラー 9-95
文字列定数 7-37
文字列のコピー関数 9-94
文字列の比較関数
任意の文字数 9-99
文字列全体 9-93
文字列の連結関数
任意の文字数 9-98
文字列全体 9-92

ゆ

ユーティリティ
概要 1-7

よ

呼び出し規則
関数の呼び出し方法 8-19
引数とローカル変数へのアクセス方法 8-22
呼び出し先関数の対応方法 8-20
レジスタの使用状況 8-18
読み込み
ストリーム関数
標準入力からの 9-87
文字列 9-66, 9-91
文字列から配列へ 9-64
文字関数
単一文字 9-60
次の文字関数 9-67, 9-68
複数文字 9-61
読み込み関数 9-68

ら

ラージ・メモリ・モデル 2-16, 8-6
ライブラリ
ランタイム・サポート 9-2

ライブラリ作成ユーティリティ
オプション 10-4
オプションのオブジェクト・ライブラリ 10-3
コンパイラとアセンブラ・オプション 10-5 ～ 10-6
説明 1-4
ラベル
大文字小文字の区別
-ac コンパイラ・オプション 2-23
定義 A-9
保持 2-24
乱整数関数 9-84
ランタイム環境
C とアセンブリ言語間のインターフェイス 8-23
関数呼び出し規則 8-19
スタック 8-4
定義 A-9
はじめに 8-1
メモリ・モデル
自動初期化中 8-5
セクション 8-2
動的メモリ割り当て 8-5
レジスタ規則 8-17
割り込み処理
説明 8-46
レジスタの保存方法 7-10
ランタイム・サポート A-9
関数
概要 9-1
定義 A-9
まとめ 9-29 ～ 9-40
標準ライブラリ 10-2
マクロの一覧 9-29 ～ 9-40
ライブラリ
C コードとのリンク 5-2, 5-8
説明 9-2
ライブラリ作成ユーティリティ 1-4, 10-1
ランタイム・サポートのオフチップへの配置
関数呼び出しの制御 7-12, 8-51

り

リスト・ファイル
クロスリファレンスの作成 2-24
定義 A-9
プリプロセッサによる生成 2-36
リスト・ファイル内のシンボルのクロスリファレンス 2-24
リトルエンディアン
定義 A-9
ビッグに変更 2-16
リニア・アセンブリ
アセンブリ・オプティマイザ疑似命令 4-13
記述 4-4
機能ユニットの指定 4-6
説明 4-1
ソース・コメント 4-6
定義 A-9
レジスタ指定方法 4-8
レジスタの指定 4-6
リニア・アセンブリ内の機能ユニットの指定方法 4-6
リニア・アセンブリ内のトリップ・カウントの指定 4-30
リニア・アセンブリ内のレジスタの予約 4-28
リニア・アセンブリのレジスタの指定 4-6
リニア・アセンブリのレジスタの直接分割 4-28
リニア・アセンブリ・ソース内 4-11
リンカ
オプション 5-5
オプションのまとめ 2-14
起動方法 2-17
コマンド・ファイル 5-12
コンパイラを使用してリンカを起動 5-2
コンパイル・ステップの一部として 5-3
独立したステップ 5-2
制御方法 5-8
説明 1-3
定義 A-9
無効化 5-4
抑止 2-15
リンク
C/C++ コード 5-1
C6400 コードと C6200/C6700/ 旧世代 C6400 オブジェクト・コード 2-45
オブジェクト・ライブラリ 9-2
ランタイム・サポート・ライブラリ 5-8

る

累乗関数 9-81
ループ
_nassert を使用したコンパイラの知識の拡張 8-37
最適化 3-43
冗長 3-16
ソフトウェア・パイプライン化 3-4
ループの循環の最適化 3-44
ループの展開
定義 A-9
ループ不変最適化 3-44

れ

例外処理　9-28
レジスタ
　C/C++ からの制限レジスタへのアクセス方法　7-8
　規則　8-17
　リニア・アセンブリの分割化　4-8
　リブアウト　4-24
　リブイン　4-24
　割り当て　8-17
　割り込み時の使用　8-46
　割り込み時の保存方法　7-10
レジスタ変数
　コンパイル　7-16
　最適化　3-44
レジスタ・パラメータ
　コンパイル　7-16

ろ

ローカル変数
　アクセス方法　8-22
ローダ
　定義　A-9
　リンカと一緒に使用する　7-34
ローダを使用して引数を渡す方法　6-6
ロー・リスト・ファイル
　-pl オプションによる生成　2-36
　識別子　2-36
ロー・リスト・ファイル内の診断識別子　2-37

わ

ワイルドカード
　使用　2-19
割り当て
　定義　A-10
割り込み
　柔軟性オプション　2-43
　処理
　　説明　8-46
　　レジスタの保存方法　7-10
割り込み時のレジスタの保存方法　7-10

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/

TMS320C6000
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 3 版 2005 年 7 月
第 2 版 2001 年 12 月
第 1 版 2000 年 6 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）

Made in Japan
’05.07

日本テキサス・インスツルメンツ株式会社

SPRU419B