TEXAS INSTRUMENTS

TMS320C55x
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*

SPRU281 翻訳版

JAJU027 WAS SPRUE45

TMS320C55x
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：SPRU281F
2003年 3月

2005 年 9 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
- ❏ オプティマイザ
- ❏ ライブラリ作成ユーティリティ
- ❏ C++ ネーム・デマングラ

TMS320C55x™ C/C++ コンパイラは、国際標準化機構（ISO）準拠の標準 C および C++ コードを受け入れ、TMS320C55x デバイス用のアセンブリ言語ソース・コードを生成します。コンパイラは、1989 バージョンの C 言語をサポートします。

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

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

表記規則

本書では、次の表記規則を使用します。

- ❏ TMS320C55x デバイスは C55x として参照されます。
- ❏ プログラム・リスト、プログラム例、および対話表示は、タイプライタの活字に似た特殊な活字（`special typeface`）で示してあります。例は、強調のため、ボールド（**`bold version`**）で示してあります。対話表示についても、ユーザが入力するコマンドとシステムが表示する項目（プロンプト、コマンド出力、エラー・メッセージなど）と区別するために、ボールド（**`bold version`**）で示しています。

C コードの例を次に示します。

```
#ifdef NDEBUG
#define assert(ignore) ((void)0)
#else
#define assert(expr) ((void)((_expr) ? 0 :                    \
       (printf("Assertion failed, ("#_expr"), file %s,       \
       line %d\n, __FILE__, __LINE__),                       \
       abort () )))
#endif
```

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

  cl55 [*options*] [*filenames*] [*-z* [*link_options*] [*object files*]]

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

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

- 大括弧（[ ]）は、オプションパラメータを特定します。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。

  cl55 [*options*] [*filenames*] [*-z* [*link_options*] [*object files*]]

  cl55 コマンドには、いくつかのオプション・パラメータがあります。

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

  cl55 -z {**-c** | **-cr**} *filenames* [-o *name.out*] **-l** *libraryname*

当社発行の関連文献

以下の文献は、TMS320C55x および関連サポート・ツールについて説明しています。当社の刊行物を入手するには、タイトルと文献番号（表紙に記載）をご確認の上、プロダクト・インフォメーション・センター（PIC）、www.tij.co.jp/PIC/ にお問い合わせください。

TMS320C55x Assembly Language Tools User's Guide（文献番号 SPRU280）は、TMS320C55x 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。

TMS320C55x DSP CPU Reference Guide（文献番号 SPRU371）は、TMS320C55x DSP の CPU のアーキテクチャ、レジスタ、および動作について解説しています。

TMS320C55x DSP Peripherals Reference Guide（文献番号 SPRU317）は、周辺インターフェイス、および TMS320C55x DSP で使用できる関連ハードウェアについて解説しています。

TMS320C55x DSP Algebraic Instruction Set Reference Guide（文献番号 SPRU375）は、TMS320C55x DSP 代数命令をそれぞれ解説しています。また、命令セット、命令コード、およびニーモニック命令セットへのクロスリファレンスについてまとめています。

TMS320C55x DSP Mnemonic Instruction Set Reference Guide（文献番号 SPRU374）は、TMS320C55x DSP ニーモニック命令をそれぞれ解説しています。また、命令セット、命令コード、および代数命令セットへのクロスリファレンスについてまとめています。。

TMS320C55x DSP Programmer's Guide（文献番号 SPRU376）は、TMS320C55x DSP のための C およびアセンブリ・コードの最適化方法を解説し、DSP の特殊機能および特殊命令を使うコードの記述方法を説明しています。

TMS320C55x Technical Overview（文献番号 SPRU393）は、TMS320C55x DSP の概要を説明します。TMS320C5000™ DSP プラットフォームで生成される固定小数点 DSP の最新情報です。旧世代と同様、このプロセッサはパフォーマンスが高く、低消費電力での動作に最適化されています。本書は CPU アーキテクチャ、低消費電力での機能向上、および組み込み用のエミュレーション機能について解説しています。

Code Composer User's Guide（文献番号 SPRU328）は、Code Composer 開発環境を使用して組み込み用のリアルタイム DSP アプリケーションを作成し、デバッグする方法について解説しています。

関連文献

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。

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

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

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

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

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

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

The Annotated C++ Reference Manual、Margaret A. Ellis および Bjarne Stroustup との共著、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 との共著、Prentice-Hall, Englewood Cliffs（New Jersey）刊行（1988）

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

商標

Code Composer Studio、TMS320C55x、C55x、TMS320C54x、および C54x は Texas Instruments の商標です。

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

MCS-86 は Intel Corporation の商標です。

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

Tektronix は Tektronix, Inc の商標です。

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

目次

1 はじめに 1-1
TMS320C55x のソフトウェア開発ツール、特に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 規格 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 コンパイラおよび Code Composer Studio 1-8

2 C/C++ コンパイラの使用方法 2-1
コンパイラの操作方法を説明します。ソース・ファイルをコンパイル、アセンブル、リンクするコンパイラの起動についての方法を含んでいます。コンパイラ・オプション、コンパイラ・エラー、および差し込み機能について考察します。

2.1 コンパイラについて 2-2
2.2 C/C++ コンパイラの起動方法 2-4
2.3 オプションによるコンパイラの動作の変更 2-5
2.3.1 使用頻度の高いオプション 2-17
2.3.2 デバイス・バージョンの選択方法（-v オプション） 2-21
2.3.3 シンボリック・デバッグおよびプロファイルのオプション 2-23
2.3.4 ファイル名の指定方法 2-24
2.3.5 コンパイラによるファイル名の解釈方法の変更（-fa、-fc、-fg、-fo、および -fp オプション） 2-24
2.3.6 コンパイラによるファイル名の拡張子の解釈方法とファイル作成時の拡張子の付け方の変更（-ea、-ec、-eo、-ep、および -es オプション） 2-25
2.3.7 ディレクトリの指定方法 2-26
2.3.8 アセンブラを制御するオプション 2-27
2.3.9 非推奨オプション 2-30
2.4 環境変数の使用方法 2-31
2.4.1 ディレクトリの指定方法（C_DIR および C55x_C_DIR） 2-31
2.4.2 デフォルトのコンパイラ・オプションの設定方法（C_OPTION および C55X_C_OPTION） 2-31

2.5 プリプロセッサの制御方法 2-33
2.5.1 事前定義マクロ名 2-33
2.5.2 #include ファイル検索パス 2-34
2.5.3 前処理リスト・ファイルの生成方法（-ppo オプション） 2-35
2.5.4 前処理後のコンパイルの続行方法（-ppa オプション） 2-36
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） 2-36
2.5.6 行の制御情報付き前処理リスト・ファイルの生成（-ppl オプション） 2-36
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） 2-36
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） 2-36
2.6 診断メッセージの概要 2-37
2.6.1 診断の制御方法 2-39
2.6.2 診断抑止オプションの使用方法 2-40
2.6.3 その他のメッセージ 2-41
2.7 クロスリファレンス・リスト情報の生成（-px オプション） 2-42
2.8 ロー・リスト・ファイルの生成方法（-pl オプション） 2-43
2.9 インライン関数展開の使用方法 2-45
2.9.1 組み込み演算子のインライン展開 2-45
2.9.2 自動インライン展開 2-45
2.9.3 保護されない定義制御インライン展開 2-46
2.9.4 保護されたインライン展開と _INLINE プリプロセッサ・シンボル 2-46
2.9.5 インライン展開の制約事項 2-48
2.10 インターリストの使用方法 2-49

3 コードの最適化方法 3-1

インライン展開やループ展開等の機能を含む C/C++ コードの最適化を説明します。オプティマイザ使用時に行われる最適化の種類も説明します。

3.1 オプティマイザの起動方法 3-2
3.2 ファイルレベルの最適化の実行（-O3 オプション） 3-3
3.2.1 ファイルレベルの最適化の制御（-ol オプション） 3-3
3.2.2 最適化情報ファイルの作成（-on オプション） 3-4
3.3 プログラムレベルの最適化の実行（-pm および -O3 オプション） 3-5
3.3.1 プログラムレベルの最適化の制御（-op オプション） 3-5
3.3.2 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 3-7
3.4 最適化コード内で asm 文を使用する場合の注意 3-9
3.5 最適化コード内でエイリアスが設定された変数へのアクセス 3-10
3.6 自動インライン展開（-oi オプション） 3-11
3.7 インターリスト・ユーティリティをオプティマイザと組み合わせて使用する方法 3-12
3.8 最適化されたコードのデバッグ 3-14
3.8.1 最適化されたコードのデバッグ（-g、--symdebug:dwarf、--symdebug:coff および -O オプション） 3-14
3.8.2 最適化されたコードのプロファイル方法 3-15

3.9 実行できる最適化の種類 3-16
3.9.1 コストに基づいたレジスタ割り当て 3-17
3.9.2 エイリアスの明確化 3-17
3.9.3 分岐の最適化と制御フローの簡略化 3-17
3.9.4 データ・フローの最適化 3-19
3.9.5 式の簡略化 3-19
3.9.6 関数のインライン展開 3-21
3.9.7 誘導変数と強度換算 3-22
3.9.8 ループ不変コードの移動 3-22
3.9.9 ループの循環 3-22
3.9.10 自動インクリメント・アドレッシング 3-22
3.9.11 リピート・ブロック 3-23
3.9.12 テール結合 3-24

4 C/C++ コードのリンク方法 4-1
リンクをコンパイル・ステップと独立または一部の手順としてリンクを行う方法と、C/C++ コードのリンクする特定要求事項に合わせる方法を説明します。

4.1 リンカの起動方法（-z オプション） 4-2
4.1.1 独立したステップでリンカを起動する方法 4-2
4.1.2 コンパイル・ステップの一部としてリンカを起動する方法 4-3
4.1.3 リンカを無効にする方法（-c オプション） 4-4
4.2 リンカ・オプション 4-5
4.3 リンク・プロセスの制御方法 4-8
4.3.1 ランタイムサポート・ライブラリのリンク方法 4-8
4.3.2 実行時の初期化 4-9
4.3.3 割り込みベクトルによる初期化 4-9
4.3.4 グローバル変数の構築 4-10
4.3.5 初期化のタイプの指定方法 4-10
4.3.6 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 4-11
4.3.7 リンカ・コマンド・ファイルの例 4-12

5 TMS320C55x C/C++ 言語 5-1
ISO C 標準に帰属するコンパイラの特性を考察します。

5.1 TMS320C55x C の特性 5-2
5.1.1 識別子と定数 5-2
5.1.2 データ型 5-2
5.1.3 変換 5-3
5.1.4 式 5-3
5.1.5 宣言 5-3
5.1.6 プリプロセッサ 5-4
5.2 TMS320C55x C++ の特性 5-5
5.3 データ型 5-6

5.4 キーワード 5-8
5.4.1 const キーワード 5-8
5.4.2 ioport キー ワード 5-9
5.4.3 interrupt キー ワード 5-12
5.4.4 onchip キー ワード 5-12
5.4.5 restrict キーワード 5-13
5.4.6 volatile キーワード 5-14
5.5 レジスタ変数およびパラメータ 5-15
5.6 asm 文 5-16
5.7 プラグマ疑似命令 5-17
5.7.1 CODE_SECTION プラグマ 5-18
5.7.2 C54X_CALL および C54X_FAR_CALL プラグマ 5-19
5.7.3 DATA_ALIGN プラグマ 5-21
5.7.4 DATA_SECTION プラグマ 5-22
5.7.5 FAR プラグマ 5-23
5.7.6 FUNC_CANNOT_INLINE プラグマ 5-24
5.7.7 FUNC_EXT_CALLED プラグマ 5-24
5.7.8 FUNC_IS_PURE プラグマ 5-25
5.7.9 FUNC_IS_SYSTEM プラグマ 5-26
5.7.10 FUNC_NEVER_RETURNS プラグマ 5-26
5.7.11 FUNC_NO_GLOBAL_ASG プラグマ 5-27
5.7.12 FUNC_NO_IND_ASG プラグマ 5-27
5.7.13 INTERRUPT プラグマ 5-28
5.7.14 MUST_ITERATE プラグマ 5-28
5.7.15 UNROLL プラグマ 5-30
5.8 リンク名の生成 5-32
5.9 静的変数とグローバル変数の初期化方法 5-33
5.9.1 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 5-33
5.9.2 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 5-34
5.10 ISO C 言語モードの変更方法（-pk、-pr、および -ps オプション） 5-35
5.10.1 K&R C との互換性（-pk オプション） 5-35
5.10.2 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） 5-37
5.10.3 組み込み C++ モードの有効化（-pe オプション） 5-37
5.11 コンパイラの限界 5-38

6 ランタイム環境 6-1
コンパイラがどのように C55x アーキテクチャを使用しているかに関する技術情報を含みます。メモリ、レジスタ、関数呼び出し規則、そしてシステム初期化について説明します。C/C++ プログラムにアセンブリ言語をインターフェイスするために必要な情報を提供します。

6.1 メモリ 6-2
6.1.1 スモール・メモリ・モデル 6-2
6.1.2 ラージ・メモリ・モデル 6-3
6.1.3 ヒュージ・メモリ・モデル 6-3
6.1.4 セクション 6-4
6.1.5 C/C++ システム・スタック 6-6
6.1.6 動的なメモリ割り当て 6-7
6.1.7 変数の初期化 6-8
6.1.8 静的変数とグローバル変数へのメモリ割り当て方法 6-9
6.1.9 フィールド／構造体の位置合わせ 6-9
6.2 文字列定数 6-11
6.3 レジスタ規則 6-12
6.3.1 ステータス・レジスタ 6-14
6.4 関数の構造と呼び出し規則 6-16
6.4.1 関数の呼び出し方法 6-17
6.4.2 呼び出し先関数の対応方法 6-20
6.4.3 引数とローカルへのアクセス方法 6-21
6.5 アセンブリ言語と C/C++ 言語間のインターフェイス 6-22
6.5.1 C/C++ コードでのアセンブリ言語モジュールの使用方法 6-22
6.5.2 アセンブリ言語変数に C/C++ からアクセスする方法 6-24
6.5.3 インライン・アセンブリ言語の使用方法 6-27
6.5.4 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 6-28
6.6 割り込み処理 6-38
6.6.1 割り込みについての一般的なポイント 6-38
6.6.2 割り込みエントリ上でのコンテキストの保存方法 6-38
6.6.3 C/C++ 割り込みルーチンの使用方法 6-39
6.6.4 割り込みのための組み込み関数 6-39
6.7 P2 リザーブド・モードにおけるデータの拡張アドレッシング 6-40
6.7.1 拡張メモリ内へのデータの配置 6-41
6.7.2 拡張メモリにおけるデータ・オブジェクトの全アドレス取得方法 6-41
6.7.3 far データ・オブジェクトへのアクセス方法 6-42
6.8 拡張メモリ内における .const セクション 6-43
6.9 システムの初期化 6-45
6.9.1 変数の自動初期化 6-45
6.9.2 グローバル・オブジェクト・コンストラクタ 6-46
6.9.3 初期化テーブル 6-46
6.9.4 実行時の変数の自動初期化 6-49
6.9.5 ロード時の変数の初期化 6-50

7 ランタイムサポート関数 7-1

C/C++ コンパイラやマクロ、関数、型宣言によって含まれるライブラリやヘッダ・ファイルを説明します。ヘッダによるランタイムサポート関数の分類によるまとめとアルファベット順のまとめを提供します。

7.1 ライブラリ 7-2
7.1.1 ライブラリ関数の修正方法 7-3
7.1.2 さまざまなオプションによるライブラリの作成方法 7-3
7.2 C 入出力関数 7-4
7.2.1 低レベル入出力実装の概要 7-6
7.2.2 C 入出力用デバイスの追加方法 7-7
7.3 ヘッダ・ファイル 7-16
7.3.1 診断メッセージ（assert.h/cassert） 7-17
7.3.2 文字の判別と変換（ctype.h/cctype） 7-17
7.3.3 エラー報告（errno.h/cerrno） 7-18
7.3.4 拡張アドレッシング関数（extaddr.h） 7-18
7.3.5 低レベル入出力関数（file.h） 7-18
7.3.6 制限値（float.h/cfloat と limits.h/climits） 7-19
7.3.7 整数型のフォーマット変換（inttypes.h） 7-21
7.3.8 代替スペリング（iso646.h/ciso646） 7-22
7.3.9 浮動小数点算術（math.h/cmath） 7-22
7.3.10 非ローカル・ジャンプ（setjmp.h/csetjmp） 7-22
7.3.11 可変引数（stdarg.h/cstdarg） 7-23
7.3.12 標準定義 （stddef.h/cstddef） 7-23
7.3.13 整数型（stdint.h） 7-24
7.3.14 入出力関数（stdio.h/cstdio） 7-25
7.3.15 汎用ユーティリティ（stdlib.h/cstdlib） 7-26
7.3.16 文字列関数（string.h/cstring） 7-27
7.3.17 時間関数（time.h/ctime） 7-27
7.3.18 例外処理（exception と stdexcept） 7-28
7.3.19 動的メモリ管理（new） 7-28
7.3.20 ランタイム型情報（typeinfo） 7-28
7.4 ランタイムサポート関数およびマクロのまとめ 7-29
7.5 ランタイムサポート関数およびマクロについて 7-39

8 ライブラリ作成ユーティリティ 8-1
コードをコンパイルするのに使用されるオプションに対するランタイムサポート・ライブラリのカスタム化を行うユーティリティを説明します。このユーティリティを用いてヘッダ・ファイルをディレクトリに入れたり、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。

8.1 ライブラリ作成ユーティリティの起動方法 8-2
8.2 ライブラリ作成ユーティリティのオプション 8-3
8.3 オプションのまとめ 8-4

9 C++ ネーム・デマングラ 9-1
C++ ネーム・デマングラの説明と起動および使用方法について解説します。

9.1 C++ ネーム・デマングラの起動方法 9-2
9.2 C++ ネーム・デマングラのオプション 9-2
9.3 C++ ネーム・デマングラの使用例 9-3

10 用語集 A-1

図

図 1–1　TMS320C55x ソフトウェア開発のフロー 1-2
図 2–1　C/C++ コンパイラの概要 2-3
図 6–1　関数の呼び出し中におけるスタックの使用 6-16
図 6–2　組み込み関数ヘッダ・ファイル gsm.h 6-36
図 6–3　ファイル extaddr.h 6-40
図 6–4　.cinit セクション内の初期化レコードの形式 6-46
図 6–5　.pinit セクション内の初期化レコードの形式 6-49
図 6–6　実行時の自動初期化 6-50
図 6–7　ロード時の初期化 6-51
図 7–1　入出力関数でのデータ構造の相互作用 7-6
図 7–2　ストリーム・テーブル内の最初の 3 つのストリーム 7-7

表

表 2–1　コンパイラ・オプションのまとめ 2-6
表 2–2　コンパイラ下位互換オプションのまとめ 2-30
表 2–3　事前定義マクロ名 2-33
表 2–4　ロー・リスト・ファイルの識別子 2-43
表 2–5　ロー・リスト・ファイル診断識別子 2-43
表 3–1　-O3 と組み合わせて使用できるオプション 3-3
表 3–2　-ol オプションのレベルの選択方法 3-3
表 3–3　-on オプションのレベルの選択方法 3-4
表 3–4　-op オプションのレベルの選択方法 3-6
表 3–5　-op オプションを使用する場合の特別な注意事項 3-6
表 4–1　コンパイラが作成するセクション 4-11
表 5–1　TMS320C55x C/C++ データ型 5-6
表 6–1　セクションおよびメモリ配置のまとめ 6-6
表 6–2　レジスタの使用および保存の規則 6-13
表 6–3　ステータス・レジスタ・フィールド 6-14
表 6–4　C55x C/C++ コンパイラの組み込み関数（加算、減算、否定演算、絶対値） 6-30
表 6–5　C55x C/C++ コンパイラの組み込み関数（乗算、シフト） 6-31
表 6–6　C55x C/C++ コンパイラの組み込み関数（丸め処理、飽和、ビットカウント、極値） 6-32
表 6–7　C55x C/C++ コンパイラの組み込み関数（副次作用のある算術演算） 6-33
表 6–8　C55x C/C++ コンパイラの組み込み関数（非算術演算） 6-34
表 6–9　ETSI サポート関数 6-35
表 7–1　整数型の範囲に関する制限値を指定するマクロ（limits.h） 7-19
表 7–2　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） 7-20
表 7–3　ランタイムサポート関数およびマクロのまとめ 7-30
表 8–1　オプションとその機能のまとめ 8-4

例

例 2–1 inline キーワードの使用方法 2-46
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 2-47
例 2–3 差し込み後のアセンブリ言語ファイル 2-50
例 3–1 -O2 および -os オプションを使用してコンパイルした 例 2–3 の関数 3-12
例 3–2 -O2、-os、および -ss オプションを使用してコンパイルした 例 2–3 の関数 3-13
例 3–3 制御フローの簡略化と複写伝播 3-18
例 3–4 データ・フローの最適化と式の簡略化 3-20
例 3–5 関数のインライン展開 3-21
例 3–6 自動インクリメント・アドレッシング、ループ不変コードの移動、および強度換算 3-23
例 3–7 テール結合 3-24
例 4–1 リンカ・コマンド・ファイル 4-13
例 5–1 I/O 空間内のポインタの宣言方法 5-9
例 5–2 I/O 空間内のデータを示すポインタの宣言方法 5-10
例 5–3 I/O 空間内のデータを示す ioport ポインタの宣言方法 5-11
例 5–4 ポインタと restrict 型修飾子の使用 5-13
例 5–5 配列と restrict 型修飾子の使用 5-13
例 5–6 CODE_SECTION プラグマの使用方法 5-18
例 5–7 DATA_SECTION プラグマの使用方法 5-22
例 5–8 FAR プラグマの使用方法 5-23
例 6–1 “Var” のフィールド／構造体の位置合わせ 6-10
例 6–2 レジスタ引数の規則 6-19
例 6–3 C からのアセンブリ言語関数の呼び出し方法 6-24
例 6–4 変数に C からアクセスする方法 6-25
例 6–5 .bss で定義されていない変数に C からアクセスする方法 6-26
例 6–6 C からアセンブリ言語定数にアクセスする方法 6-27
例 6–7 Far オブジェクトへのアクセスのためのイディオム 6-42
例 6–8 データの拡張アドレッシング 6-43
例 6–9 初期化変数および初期化テーブル 6-47
例 9–1 名前のマングリング 9-3
例 9–2 C++ ネーム・デマングラ実行後の結果 9-4

注

-v オプションを使用した場合と使用しない場合の違い 2-21
-vP2 オプションを使った P2 リザーブド・モードのコンパイル 2-22
関数のインライン展開によりコード・サイズが大きく増大する可能性があります 2-45
-O3 の最適化とインライン展開 3-11
インライン展開とコード・サイズ 3-11
パフォーマンスとコード・サイズに与える影響 3-13
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 3-14
プロファイル・ポイント 3-15
リンカにおいて引数を処理する順序 4-4
_c_int00 シンボル 4-9
ブート・ローダ 4-11
セクションの割り振り 4-12
C55x の Byte は 16 ビット 5-6
long long は 40 ビット 5-7
asm 文により C/C++ 環境が損なわれるのを防ぐ 5-16
Code Composer Studio でインストールしたランタイム・ライブラリを修正 5-21
リンカのメモリ・マップ定義 6-2
データの配置 6-3
ヒュージ・メモリ・モデルに無い機能 6-4
.stack および .sysstack セクションの配置 6-7
C55x 呼び出し規則 6-18
asm 文の使用方法 6-28
変数の初期化方法 6-46
一意的な関数名の使用方法 7-7
ユーザ固有の clock 関数の記述 7-28
ユーザ固有の clock 関数の記述 7-46
C55x の char がホストのバイトと異なる場合の fread の使用方法 7-57
minit の後では以前に割り当てられているオブジェクトは使用不可能 7-71
time 関数はターゲットシステム固有 7-95

第 1 章

はじめに

TMS320C55x™ は、オプティマイジング（最適化）C/C++ コンパイラ、アセンブラ、リンカ、および各種ユーティリティなど、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。アセンブラとリンカの詳細は、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

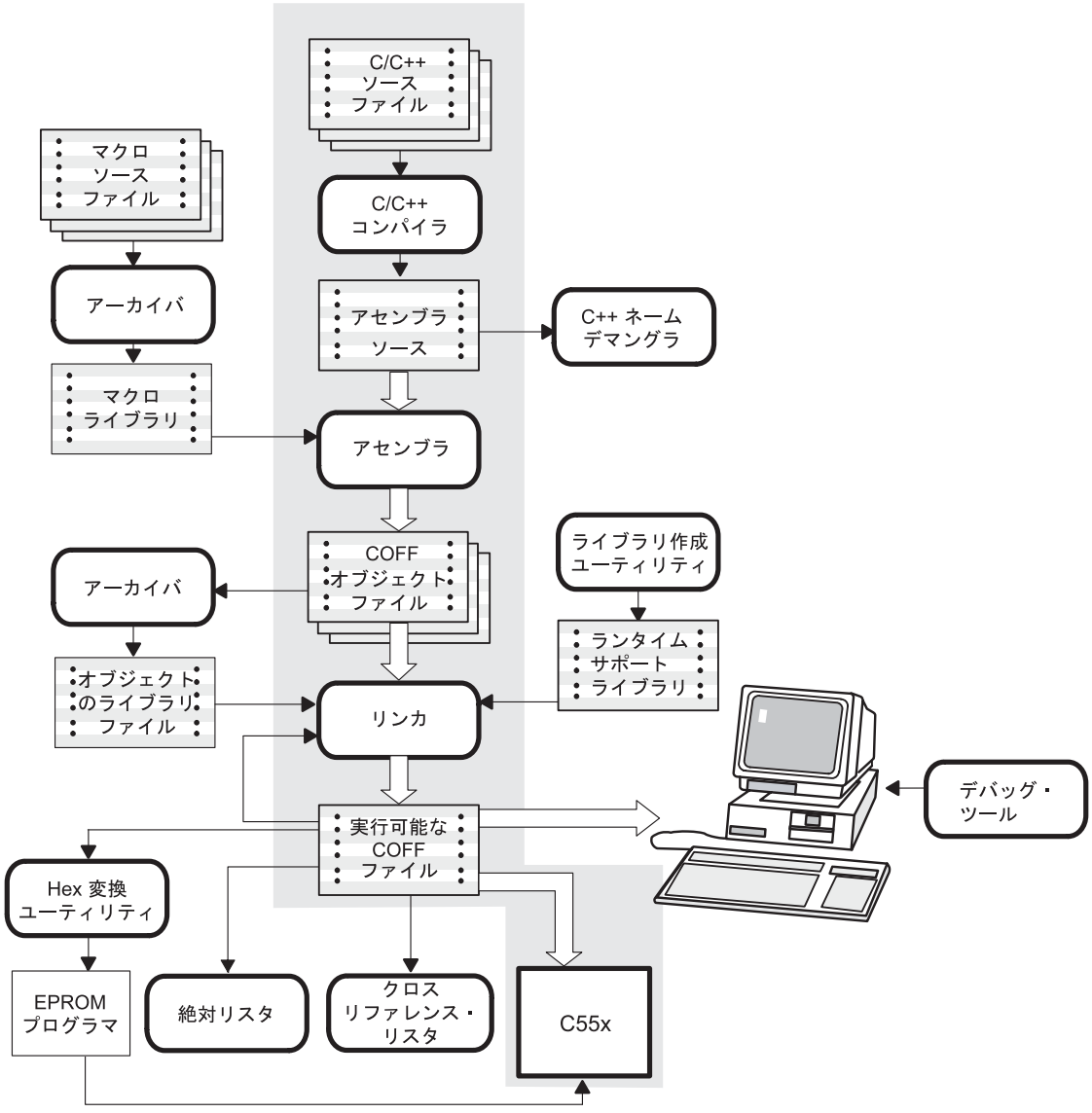
項目 **ページ**

1.1 ソフトウェア開発ツールの概要 1-2
1.2 C/C++ コンパイラの概要 1-5
1.3 コンパイラおよび Code Composer Studio 1-8

1.1 ソフトウェア開発ツールの概要

図 1–1 は、C55x 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. TMS320C55x ソフトウェア開発のフロー



以下のリストは、図 1–1 で説明されているツールについて説明しています。

- **C/C++ コンパイラ**は C/C++ ソース・コードを受け入れ、C55x アセンブリ言語ソース・コードを作成します。**オプティマイザ**は、コンパイラの一部です。このオプティマイザは、コードを変更して C/C++ プログラムの効率を高めます。

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

- **アセンブラ**は、アセンブリ言語ソース・コードを機械語のオブジェクト・ファイルに変換します。TMS320C55x ツールには、2 つのアセンブラが含まれています。ニーモニック・アセンブラは C54x および C55x のニーモニック・アセンブリ・ソース・ファイルを受け入れます。代数アセンブラは C55x の代数アセンブリ・ソース・ファイルを受け入れます。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、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

- **リンカ**は、オブジェクト・ファイルを結合して、1 つの実行可能なオブジェクト・モジュールを作成します。また、リンカは実行可能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 4 章「C/C++ コードのリンク方法」を参照してください。リンカの詳細は、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

- **アーカイバ**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。アセンブラの使用方法については、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

- **ライブラリ作成ユーティリティ**を使って、独自にカスタマイズしたランタイムサポート・ライブラリを作成できます（第 8 章「ライブラリ作成ユーティリティ」を参照してください）。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・コードとして提供されています。

  ランタイムサポート・ライブラリには、C55x コンパイラによってサポートされている ANSI/ISO 標準の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。詳細については、第 7 章「ランタイムサポート関数」を参照してください。

- C55x デバッガは、実行可能な COFF ファイルを入力として受け入れますが、大半の EPROM プログラマはそれらのファイルを受け入れません。**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、または Tektronix オブジェクト・フォーマットに変換します。変換後のファイルは、EPROM プログラマにダウンロードできます。Hex 変換ユーティリティの使用方法については、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

- **絶対リスタ**は、リンクされたオブジェクト・ファイルを入力として受け入れ、.abs ファイルを出力として作成します。これらの .abs ファイルをアセンブルし、相対的でなく絶対的なアドレスを含むリストを作成できます。絶対リスタがない場合、このようなリストの作成は冗長で、多くの手動操作が必要となります。絶対リスタの使用方法については、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

- **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等が表示されています。クロスリファレンス・リスタの使用方法については、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

- **C++ ネーム・デマングラ**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 9 章「C++ ネーム・デマングラ」を参照してく ださい。

- この開発プロセスの主な目的は、TMS320C55x 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。

  - 命令に正確なソフトウェア・シミュレータ
  - 拡張開発システム（XDS510™）エミュレータ

  これらのツールは、Code Composer Studio からアクセスされます。詳細は、Code Composer Studio ユーザーズ・ガイドを参照してください。

1.2 C/C++ コンパイラの概要

C55x C/C++ コンパイラは、標準 ANSI/ISO C/C++ プログラムを C55x 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 規格

次の特徴は ISO 規格に関するものです。

- **ISO 規格 C**

  C55x C/C++ コンパイラは ISO C 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 C 規格は ANSI C 規格の代わりで、同じです。

- **ISO 規格 C++**

  C55x C/C++ コンパイラは ISO C++ 規格により定義され、Ellis および Stroustrup の The Annotated C++ Reference Manual (ARM) に記述された ISO C++ 規格をサポートしています。コンパイラは、組み込み C++ もサポートします。

  サポートされていない C++ の機能については、5.2 節「TMS320C55x C++ の特性」（5-5 ページ）を参照してください

- **ISO 規格ランタイムサポート**

  本コンパイラ・ツールには、完全なランタイム・ライブラリが標準装備されています。すべてのライブラリ関数は、ISO 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。

  C++ 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。

  詳細については、第 7 章「ランタイムサポート関数」を参照してください。

1.2.2 出力ファイル

次の特徴は、コンパイラによって作成される出力ファイルに関するものです。

- **アセンブリ・ソース出力**

  本コンパイラにより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、C/C++ ソース・ファイルから生成したコードを確認することができます。

- **COFF オブジェクト・ファイル**

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

- **EPROM プログラマ・データ・ファイル**

  スタンドアロン型の組み込みアプリケーションの場合は、本コンパイラを使用して、すべてのコードと初期設定データを ROM に配置することができます。この結果、C/C++ コードはリセットで実行できるようになります。コンパイラによる COFF ファイル出力は、Hex 変換ユーティリティを使うことにより、EPROM プログラマ・データ・ファイルに変換できます。変換ユーティリティの詳細は、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

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**

  コンパイラ・ツールを使用すると、プログラムのコンパイル、アセンブル、およびリンクを 1 ステップで実行することができます。詳細は、2.2 節「C/C++ コンパイラの起動方法」（2-4 ページ）を参照してください。

- **柔軟なアセンブリ言語インターフェイス**

  本コンパイラの呼び出し規則は単純化しているので、相互に呼び出すアセンブリ関数や C 関数を記述することができます。詳細は、第 6 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」（2-33 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、C55x に固有の最適化では、C55x 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。

- **ライブラリ作成**ユーティリティは、コンパイラ・ユーティリティの重要な機能です。ライブラリ作成ユーティリティを使用すると、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 8 章「ライブラリ作成ユーティリティ」を参照してください。

- **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。詳細は、第 9 章「C++ ネーム・デマングラ」を参照してください。

1.3 コンパイラおよび Code Composer Studio

Code Composer Studio は、コード生成ツールを使うためのグラフィカル・インターフェイスを提供します。

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すべての情報を記録しています。プロジェクトは以下を記録します。

- ❏ 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、およびリンカのオプション
- ❏ インクルード・ファイルの依存関係

Code Composer Studio を使ってプロジェクトをビルドすると、適切なコード生成ツールが起動され、プログラムのコンパイル、アセンブル、およびリンクが行われます。

コンパイラ、アセンブラ、およびリンカのオプションは、Code Composer Studio の「Build Options」ダイアログで指定できます。このダイアログには、ほぼすべて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、ダイアログ上部に表示される編集可能なテキスト・ボックスに直接オプションを入力することにより、指定できます。

本書は、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を説明しています。Code Composer Studio についての詳細は、Code Composer Studio ユーザーズ・ガイドを参照してください。Code Composer Studio のコード生成ツール・オプションの設定については、「コード生成ツールのオンライン・ヘルプ」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 TMS320C55x™ の実行できるコードに変換します。実行可能なオブジェクト・ファイルを作成するに は、ソース・ファイルのコンパイル、アセンブル、リンクを実行する必要があります。これらすべてのステップは、コンパイラ cl55 を使って同時に実行されます。本章では、cl55 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、オプティマイザ、インライン関数展開機能 、およびインターリストについても説明します。

項目 **ページ**

2.1 コンパイラについて 2-2
2.2 C/C++ コンパイラの起動方法 2-4
2.3 オプションによるコンパイラの動作の変更 2-5
2.4 環境変数の使用方法 2-31
2.5 プリプロセッサの制御方法 2-33
2.6 診断メッセージの概要 2-37
2.7 クロスリファレンス・リスト情報の生成（-px オプション） 2-42
2.8 ロー・リスト・ファイルの生成方法（-pl オプション） 2-43
2.9 インライン関数展開の使用方法 2-45
2.10 インターリストの使用方法 2-49

2.1 コンパイラについて

コンパイラ cl55 を使用すると、コンパイルとアセンブル、さらに必要に応じてリンクを 1 つのステップで実行できます。コンパイラは、次の処理を 1 つ以上のソース・モジュールに対して実行します。

- **コード・ジェネレータ**は、パーサおよびオプティマイザから成り、C/C++ のソース・コードを取り込んで、C55x アセンブリ言語ソース・コードを生成します。

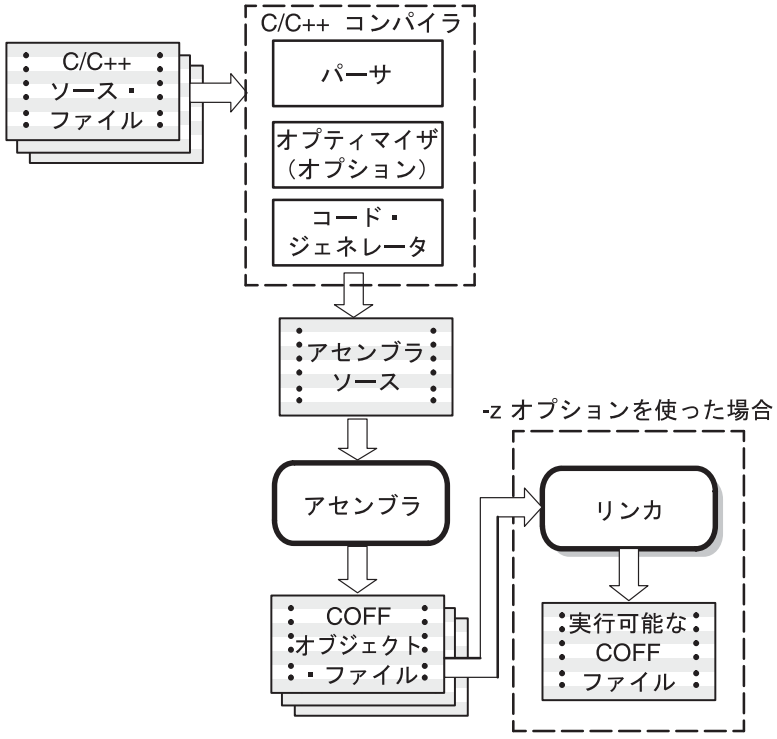
  C ファイルと C++ ファイルを 1 つのコマンドでコンパイルできます。コンパイラは、ファイル名拡張子を使用して両者を区別します（詳細は 2.3.4 項「ファイル名の指定方法」を参照してください。

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

デフォルトでは、コンパイラはリンク手順を実行しません。-z コンパイラ・オプションを使用することにより、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++ コンパイラの概要



アセンブラとリンカの詳細は、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

2.2 C/C++ コンパイラの起動方法

コンパイラを起動するには、次のように入力します。

cl55 [*options*] [*filenames*] [**-z** [*link_options*] [*object files*]]

cl55	コンパイラとアセンブラを実行させるコマンドです。
options	コンパイラによる入力ファイルの処理方法に影響を与えるオプションです（オプションは表 2–1（2-6 ページ）にリストされています)。
filenames	1 つまたは複数の C/C++ ソース・ファイル、アセンブリ・ソース・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 4 章「C/C++ コードのリンク方法」を参照してください。
link_options	リンク・プロセスの制御オプションです。
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。

cl55 への引数は、コンパイラ・オプション、リンカ・オプション、およびファイルの 3 つのタイプのいずれかです。-z リンカ・オプションは、リンクが実行されるというシグナルです。-z リンカ・オプションが使われる場合、コンパイラ・オプションは -z リンカ・オプションの前、他のリンカ・オプションは -z リンカ・オプションの後に指定する必要があります。ソース・コード・ファイル名は、-z リンカ・オプションの前に配置する必要があります。追加オブジェクト・ファイル名は、-z リンカ・オプションの後に配置する必要があります。

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

```
cl55 symtab.c file.c seek.asm -z -llnk.cmd -lrts55.lib
```

このコマンドを入力すると、次のように出力されます。

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

2.3 オプションによるコンパイラの動作の変更

オプションは、コンパイラの動作を制御します。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。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は詳細に説明しています。オプションに関する最新のまとめを参照したい場合は、コマンド行でパラメータを指定せずに cl55 と入力してください。

コンパイラ・オプションには次の規則が適用されます。

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

シェル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 C55X_C_OPTION 環境変数を使用します。C_OPTION 環境変数についての詳細は、2.4.2 項「デフォルトのコンパイラ・オプションの設定方法（C_OPTION および C55X_C_OPTION）」（2-31 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）とその構文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示されています。必要に応じて参照してください。

表 2–1. コンパイラ・オプションのまとめ

(a) コンパイラを制御するオプション

オプション	機能	ページ
-@=*filename*	コマンド行の延長として、ファイルの内容を解釈します。	2-17
-b	補足情報ファイルを生成します。	2-17
-c	リンクを使用不可にします（-z の無効化）。	2-17、4-4
--consultant	コンパイラ・コンサルタント情報を生成します。	2-18
-D=*name*[=*def*]	name を事前に定義します。	2-18
-h	オプションおよび使用方法を表示します。	--
-I=*directory*	#include 検索パスを定義します。	2-18、2-35
-k	.asm ファイルを保持します。	2-18
-n	コンパイルのみを行います。	2-19
-q	複数の進捗メッセージの出力を抑止します（静的実行）。	2-19
-qq	すべての進捗メッセージの出力を抑止します(静的実行)。	2-19
-s	オプティマイザのコメントをアセンブリ・ソース文に差し込みます。オプティマイザのコメントがない場合は、C/C++ のソースをアセンブリ・ソース文に差し込みます。	2-20
-ss	C/C++ のソースをアセンブリ文に差し込みます。	2-20、3-12
-U=*name*	name を未定義にします。	2-20
--verbose	見出しおよび関数進捗情報を表示します。	2-20
-version	ツールのバージョン番号を表示します。	2-20
-z	リンクの実行を有効にします。	2-20

表 2-1. コンパイラ・オプションのまとめ（続き）

(b) シンボリック・デバッグおよびプロファイルを制御するオプション

オプション	機能	ページ
-g	シンボリック・デバッグを有効にします（--symdebug:dwarf と同等）。	2-23、3-14
--profile:breakpt	ブレーク・ポイントに基づいたプロファイルの実行を有効にします。	2-23
--profile:power	累乗プロファイルの実行を有効にします。	2-23
--symdebug:dwarf	DWARF デバッグ・フォーマットを使ったシンボリック・デバッグを有効にします（-g と同等）。	2-23、3-14
--symdebug:none	すべてのシンボリック・デバッグを無効にします。	2-23
--symdebug:skeletal	最適化を妨げない最小限のシンボリック・デバッグを有効にします（デフォルトの動作）。	2-23

注： 非推奨シンボリック・デバッグ・オプションについては、2.3.9 項（2-30 ページ）を参照してください。

(c) ファイル作成時に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=[.]*ext*	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。	2-25
-ec=[.]*ext*	C ソース・ファイルのデフォルトの拡張子を設定します。	2-25
-eo=[.]*ext*	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。	2-25
-ep=[.]*ext*	C++ ソース・ファイルのデフォルトの拡張子を設定します。	2-25
-es=[.]*ext*	アセンブリ・リスト・ファイルのデフォルトの拡張子を設定します。	2-25

表 2-1. コンパイラ・オプションのまとめ（続き）

(d) ファイルおよびディレクトリ名を指定するオプション

オプション	機能	ページ
-fa=*filename*	拡張子に関係なく、*filename* をアセンブリ・ソース・ファイルとして識別します。デフォルトでは、コンパイラは .asm ファイルをアセンブリ・ソース・ファイルとして扱 います。	2-24
-fc=*filename*	拡張子に関係なく、*filename* を C ソース・ファイルとして識別します。デフォルトでは、コンパイラは .c ファイルを C ソース・ファイルとして扱 います。	2-24
-fg	すべての C ファイルを C++ ファイルとして処理させます。	2-24
-fo=*filename*	拡張子に関係なく、*filename* をオブジェクト・コード・ファイルとして識別します。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。	2-24
-fp=*filename*	拡張子に関係なく、*filename* を C++ ファイルとして識別します。デフォルトでは、コンパイラは .C、.cpp、.cc または .cxx ファイルを C++ ファイルとして扱います。	2-24

(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。	2-26
-ff=*directory*	アセンブリ・リスト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。	2-26
-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。	2-26
-fs=*directory*	アセンブリ・ファイルのディレクトリを指定します。	2-26
-ft=*directory*	一時ファイルのディレクトリを指定します。	2-26

表 2-1. コンパイラ・オプションのまとめ（続き）

(f) パーサを制御するオプション

オプション	機能	ページ
-pc	マルチバイト文字のサポートを有効にします。	--
-pe	組み込み C++ モードを有効にします。	5-37
-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)。	2-46
-pk	K&R 互換性を許容します。	5-35
-pl	ロー・リスト・ファイルを生成します。	2-43
-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。	3-5
-pn	組み込み関数を無効にします。	--
-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。	5-37
-ps	厳密な ISO モード（C/C++ に対してであり、K&R C に対してではない）を可能にします。	5-37
-px	クロスリファレンス・リスト・ファイルを生成します。	2-42
-rtti	ランタイム型情報（RTTI）を有効にし、オブジェクトの型を実行時に決定できるようにします。	5-5

(g) 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。	2-36
-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。	2-36
-ppd[=*file*]	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。	2-36
-ppi[=*file*]	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。	2-36
-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、行の制御情報（#line 疑似命令）と一緒に前処理された出力を書き込みます。	2-36
-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。	2-35

表 2-1. コンパイラ・オプションのまとめ（続き）

(h) 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)。	2-39
-pden	診断の識別子を、そのテキストと一緒に表示します。	2-39
-pdf	診断情報ファイルを生成します。	2-39
-pdr	注釈（軽い警告）を発行します。	2-39
-pds=*num*	*num* の識別診断を抑止します。	2-39
-pdse=*num*	*num* の識別診断をエラーに分類します。	2-39
-pdsr=*num*	*num* の識別診断を注釈に分類します。	2-39
-pdsw=*num*	*num* の識別診断を警告に分類します。	2-39
-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。	2-40
-pdw	警告診断を抑止します（エラーは発行されます)。	2-40

表 2-1. コンパイラ・オプションのまとめ（続き）

(i) C55x 固有のオプション

オプション	機能	ページ
-call=*value*	代替 C55x 関数呼び出し規則を選択します。	2-17
-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。	3-10
-mb	すべてのデータ・メモリがオンチップにあるよう指定します。	2-18
-mc	通常 .const セクションに配置される定数を、読み取り専用の初期化された静的変数として処理できるようにします。	2-18
-mg	C55x 代数アセンブリ・ファイルを受け入れます。	2-18
-ml	ラージ・メモリ・モデルを使用します。	6-3
-mn	-g が無効化した最適化を有効にします。	3-14
-mo	ファイル内の関数ごとのコードを、.clink 疑似命令でマークされた独立したサブセクションに配置します。	2-19
-mr	コンパイラがハードウェアのブロックリピート、ローカルリピート、およびリピート命令を生成するのを防ぎます。-O2 または -O3 も指定されたときのみ有効です。	2-19
-ms	最小限のコード空間を最適化します。	2-19
--nomacx	ソースが出力の場合、マクロの展開を防ぎます。	2-19
--ptrdif_t_16	ptrdiff_t を int（16 ビット）に変更します。	2-19
-v=*device*[:*revision*]	コンパイラが、C55x に最適なコードおよび任意に指定された改訂番号を生成できるようにします。	2-20

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 または **-O=0**	レジスタの使用状況を最適化します。	3-2
-O1 または **-O=1**	-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=2** 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。	3-2
-O3 または **-O=3**	-O2 による最適化を行い、さらにファイルに対して最適化を行います。	3-2
-oi=*size*	自動インライン展開サイズを設定します（-O3 のみ）。	3-11
-ol0 または **-ol=0**	（小文字の L）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。	3-3
-ol1 または **-ol=1**	（小文字の L）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。	3-3
-ol2 または **-ol=2**	（小文字の L）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。	3-3
-on0 または **-on=0**	オプティマイザ情報ファイルの生成を抑止します。	3-4
-on1 または **-on=1**	オプティマイザ情報ファイルを作成します。	3-4
-on2 または **-on=2**	冗長なオプティマイザ情報ファイルを作成します。	3-4
-op0 または **-op=0**	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。	3-5
-op1 または **-op=1**	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。	3-5
-op2 または **-op=2**	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）。	3-5
-op3 または **-op=3**	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。	3-5
-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。	3-12

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。	2-27
-ac	アセンブリ・ソース・ファイルで大文字と小文字を区別します。	2-27
-ad=*name*	*name* シンボルを設定します。	2-27
-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。	2-27
-ahi=*filename*	アセンブリ・モジュールのファイルを組み込みます。	2-27
-al	アセンブリ・リスト・ファイルを生成します。	2-27
-apd	前処理を行います。アセンブリ依存性のみをリスト表示します。	2-27
-api	前処理を行います。組み込まれた #include ファイルのみをリスト表示します。	2-27
-ar=*num*	*num* の識別するアセンブラ注釈を抑止します。	2-27
-as	シンボル・テーブルにラベルを格納します。	2-28
-ata	このソース・ファイルの実行中に、ARMS ステータス・ビットが有効になると表明します。	2-28
-atb	アセンブラに、パラレル・バス・コンフリクト・エラーを警告として処理させます。	2-28
-atc	このソース・ファイルの実行中に、CPL ステータス・ビットが有効になると表明します。	2-28
-ath	アセンブラに、サイズよりスピードの C54x 命令をエンコードさせます。	2-28
-atl	このソース・ファイルの実行中に、C54CM ステータス・ビットが有効になると表明します。	2-28
-atn	C54x の遅延した分岐または呼び出し命令の遅延スロットに位置する NOP を削除します。	2-28
-atp	アセンブリ命令プロファイル・ファイル（.prf）を生成します。	2-28
-ats	(ニーモニック・アセンブリのみ)。リテラル・シフト・カウント上の # を任意選択にします。	2-28
-att	このソース・ファイルの実行中に、SST ステータス・ビットが無効になると表明します。	2-28

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（続き）

オプション	機能	ページ
-atv	アセンブラに、特定の可変長命令の最大形式を使用させます。	2-28
-atw	(代数アセンブラのみ)。アセンブラ警告メッセージを抑止します。	2-29
-au=*name*	事前定義された定数の *name* を未定義にします。	2-29
-aw	パイプライン・コンフリクト警告を有効にします。	2-29
-ax	クロスリファレンス・ファイルを生成します。	2-29
--purecirc	C54x 固有のサーキュラ・アドレッシングのみが使われることをアセンブラに通知します。	2-29

(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	絶対出力を生成します。	4-5
-abs	絶対リスト・ファイルを作成します。	2-17
-ar	再配置可能で実行可能な出力を生成します。	4-5
--args=*size*	メモリがローダによって使われるよう位置合わせし、引数を渡します。	4-5
-b	シンボリック・デバッグ情報のマージを無効にします。	4-5
-c	実行時に変数を自動初期化します。	4-5
-cr	リセット時に変数を自動初期化します。	4-5
-e=*global_symbol*	プログラム・エントリ・ポイントを定義します。	4-5
-f=*fill_value*	ホールへの埋め込み値を出力セクションに定義します。	4-5
-g=*global_symbol*	*global_symbol* のグローバルを維持します（-h の無効化）。	4-5
-h	グローバル・シンボルを静的にします。	4-5
-heap=*size*	ヒープ・サイズ（バイト数）を設定します。	4-5
-help	オプションおよび使用状況を表示します。	4-5
-l=*directory*	ライブラリ検索パスを定義します。	4-6
-j	条件付きリンクを無効にします。	4-6

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（続き）

オプション	機能	ページ
-k	入力セクションに指定された位置合わせフラグを無視します。	4-6
-l=*filename*	ライブラリ名を提供します。	4-6
-m=*filename*	マップ・ファイル名を指定します。	4-6
-o=*filename*	出力ファイル名を指定します。	4-6
-priority	ライブラリの代替検索メカニズムを提供します。	4-6
-q	進捗メッセージの出力を抑止します（静的実行）。	4-6
-r	再配置可能な出力を生成します。	4-6
-s	シンボル・テーブルを除去します。	4-6
-stack=*size*	1 次スタック・サイズ（バイト数）を設定します。	4-6
-sysstack=*size*	2 次システム・スタック・サイズ（バイト数）を設定します。	4-6
-u=*symbol*	シンボルを未定義にします。	4-6
-w	未定義の出力セクションが作成された場合にメッセージを表示します。	4-6
-x	ライブラリの再読み取りを強制します。	4-7
--xml_info_file=*file*	詳細なリンク情報ファイルを生成します。	4-7

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。コメントを埋め込む場合は、コマンド・ファイルの行頭に # か ; を入力します。
コマンド行内では、スペースやハイフンの埋め込まれたファイル名やオプション・パラメータは、引用符で囲む必要があります（例：“this-file.obj”）。

-abs -z オプションの後で使用すると、絶対リスト・ファイルを作成します。絶対リスタに .out ファイルを指定するためには、使用しているリンカ・コマンド・ファイルで既に -o を使っている場合でも、-z の後で -o オプションを使う必要があることに注意してください。

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

-c リンカを抑止し、リンクの実行を指定した -z オプションを無効にします。このオプションは、C_OPTION または C55X_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、4.1.3 項「リンカを無効にする方法（-c オプション）」（4-4 ページ）参照してください。

-call=*value* コンパイラに、特定の呼び出し規則との互換性を強制します。C55x コンパイラの初期バージョンは、もっと非効率的な呼び出し規則を使っていました。新しい呼び出し規則は、これらの非効率性を改善しています。-call オプションは、以前の規則を使っているアセンブリ・コードを呼び出すまたは呼び出される既存のコードとの互換性をサポートしています。
-call=c55_compat を使うと、コンパイラは強制的に以前の呼び出し規則と互換性のあるコードを生成します。-call=c55_new を使うと、コンパイラは強制的に新しい呼び出し規則を使います。
コンパイラは、デフォルトでは新しい規則を使います。P2 リザーブド・モードにコンパイルする場合は例外です。この場合は、デフォルトが以前の規則に設定されています。単独の実行可能なファイル内では、1 つの呼び出し規則だけが使えます。リンカはこの規則を強制します。

--consultant コンパイラ・コンサルタント・アドバイス・ツールを介して、コンパイル時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・ツールの詳細については、「TMS320C55x Code Composer Studio のオンライン・ヘルプ」を参照してください。

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

-I=*directory* コンパイラが #include ファイルを検索するディレクトリのリストに *directory* を追加します。複数のディレクトリを定義するこのオプションは、複数回使用できます。各 -I オプションとの間は必ず空白で区切ってください。ディレクトリ名を指定しないと、プリプロセッサは -I オプションを無視します。詳細は、2.5.2.1 項「-ɪ オプションによる #include ファイル検索パスの変更」（2-35 ページ）を参照してください。

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

-mb すべてのデータ・メモリがオンチップにあるよう指定します。このオプションにより、コンパイラは C55x デュアル MAC 命令の使用状況を最適化できます。リンカ・コマンド・ファイルは、必ずこのようなデータすべてをオンチップ・メモリに配置する必要があります。

-mc 通常 .const セクションに配置される定数を、読み取り専用の初期化された静的変数として処理できるようにします。これは、小さなメモリ・モデルを使用する場合や、P2 リザーブド・モードのハードウェア上で使用する場合に便利です。これにより、ランタイムの値を保持するために使われた空間がまだプログラムの使用する単独のデータ・ページにある場合、定数値を拡張メモリにロードできるようになります。

-mg デフォルトでは、コンパイラおよびアセンブラはニーモニック・アセンブリを使用しません。コンパイラはニーモニック・アセンブリ出力を生成し、アセンブラはニーモニック・アセンブリ入力ファイルを受け入れるだけです。このオプションを指定すると、コンパイラおよびアセンブラは代数アセンブリを使用します。代数アセンブリ入力ファイルをアセンブルするか、代数 asm 文を含む C コードをコンパイルするには、このオプションを使う必要があります。代数およびニーモニックのソース・コードは、単独のソース・ファイルに混在できません。

-mo ファイル内の関数ごとのコードを、独立したサブセクションに配置します。関数が割り込みでない場合、サブセクションは .clink 疑似命令を使って条件付きリンク用にマークされます。

すべてまたはほぼすべての関数が参照され、関数が互いに複数の参照を持っている場合、-mo オプションを使うと、コード・サイズ全体が大きくなる場合があります。これは、同じファイル内の他の関数を呼び出す関数が、その呼び出し関数が短い呼び出し命令の範囲内にあると保証できないため、もっと長い呼び出し命令を使う必要があるからです。

-mr コンパイラがハードウェアのブロックリピート、ローカルリピート、およびリピート命令を生成するのを防ぎます。このオプションは、-O2 または -O3 が指定されたときのみ有効です。

-ms スピードではなく、コード空間を最適化します。

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

--nomacx ソースが出力の場合、マクロの展開を防ぎます。

--ptrdiff_t_16 prtdiff_t 型を int（16 ビット）に変更します。ptrdiff_t 型は stddef.h または cstddef ヘッダで定義されます。これは、2 つのポインタの減算結果のデータ型を示す signed int 型です。このオプションにより、ptrdiff_t 型が int であることが保証されます。

-q cl55 を起動し、ツールを実行する際、コンパイル出力メッセージを抑止します。コンパイル出力メッセージは、角括弧（[]）内の C/C++ メッセージです。

-qq cl55 を起動し、ツールを実行する際、すべての出力メッセージを抑止します。

-s
インターリスト機能を起動します。この機能では、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・ソースに差し込むことができます。オプティマイザを起動している場合は（-O*n* オプションを指定）、オプティマイザのコメントがコンパイラのアセンブリ言語出力に差し込まれます。これにより、コードを大幅に再配置できます。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。オプティマイザのインターリスト機能についての詳細は、3.7 節「インターリスト・ユーティリティをオプティマイザと組み合わせて使用する方法」（3-12 ページ）を参照してください。

-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。このオプションとともにオプティマイザが起動している場合（-O*n* オプション）、コードが大幅に再編成される場合があります。詳細は、2.10 節「インターリストの使用方法」（2-49 ページ）を参照してください。

-U=*name*
事前定義された定数の *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。

-v=*device*
[**:***revision*]
命令が生成されるプロセッサを決定します。詳細は、2.3.2 項「デバイス・バージョンの選択方法（-v オプション）」（2-21 ページ）を参照してください。

--verbose
コンパイル中に見出しを表示します。コンパイラ・バージョン番号および著作権情報が含まれます。

-version
ツールのバージョン番号を表示します。

-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。このオプションとパラメータは、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、4.1 節「リンカの起動方法（-z オプション）」（4-2 ページ）を参照してください。

2.3.2 デバイス・バージョンの選択方法（-v オプション）

-v オプションを使うと、生成するコードを実行するために使うターゲット・デバイスを指示できます。*-vdevice*[*:revision*] オプションを使うと、命令を生成する必要のあるターゲットが指定できます。*device* パラメータは、通常 TMS320C55x 製品番号の最後の 4 桁です。たとえば、TMS320VC5510 DSP デバイスを指定するには -v5510 を使います。*rev* 番号のない *-vdevice* オプションは、そのデバイスの現行の全シリコン・リビジョンで実行されるコードを生成します。しかしリビジョン番号（-v5510:1 など）を指定すると、コンパイラがそのリビジョンに対してさらに最適なコードを生成できる場合があります。
-v オプションを複数回使用して、複数のデバイスとリビジョン番号を指定することができます。特定の組み合わせは許可されていない場合があります。

-v オプションを使用した場合と使用しない場合の違い

-v オプションを使用しない場合、コンパイラは既知の TMS320C55x デバイスすべてで実行されるコードを生成します。しかし、すべてのデバイスで実行されるコードは、どのデバイスにも最適ではありません。-v を注意深く使用することにより、使用していないデバイスにおけるハードウェアの欠点への対策および検出を避けることができます。

-vdevice:0 オプションは、デバイスのオリジナルのハードウェア仕様に従ってコードを生成します。

リビジョン指定子は、必要に応じて複数桁にもできます。1.1 のリビジョン番号は、*-vdevice*:1.1 として指定されます。

この情報により、ある機能が指定されたすべてのターゲットにある場合、ツールは特定のデバイスやリビジョンでその機能を使用可能にできます。同様に、ハードウェアの欠点対策や検出は、それらが指定のターゲットに適応する場合にのみ行われます。

正確なリビジョンを指定しなくても、デバイスのコードを安全に生成できます。たとえば、-v5509 を使うと、C5509 のすべてのバージョンに互換性のある最適なコードを生成できます。製品の正確なリビジョンにさらに最適化したい場合、特定のリビジョン番号を知り、-v オプションを使って指定する必要があります。

異なる製品モデルに異なるデバイス・リビジョンやデバイスを使用している製品、または製品群でソフトウェアを実行させたい場合、複数のデバイス・リビジョン（たとえば -v5510:2.1 -v5510:2.2 -v5509）を指定することにより、特定のハードウェアに適したコードを作成できます。

一般的に、-v を使って一連の特定デバイスへのコード生成を制限すると、コードの実行状況が改善されます。

デバイスおよびリビジョンは、現在以下のようにサポートされています。

デバイス	リビジョン
core	0、1、1.0、1.1、1.x、2、2.0、2.1、2.2、3、3.0
5510	0、0.0、1、1.0、1.0A、1.1、1.1A、1.2、1.x、2、2.0、2.1、2.2、2.x、3、3.0
5501	-、1、1.0
5502	-、1、1.0
5509	A、B、C、D、E
5561	-、1、1.0、2、2.0、2.1、2.x
DA250	1、1.0、1.1、1.x、2、2.0、2.05、2.1、2.x、3、3.0
DA255	1、1.0
OMAP1510	
OMAP5910	
P2	
P2+	

デバイス core は実際のデバイスではありませんが、コードが C55x コア CPU の特定のリビジョンに対して生成されることを示す方法として使われます。上記に表示されていないデバイスに対しては、デバイスにどの C55x コアが使われるかを調べ、適切な -vcore:REV オプションを使用する必要があります。-vcore:0 は、C55x ハードウェア仕様向けにコードを生成するために使われる場合があります。つまり、シリコン固有の例外に対処する必要があります。

注 : -vP2 オプションを使った P2 リザーブド・モードのコンパイル

-vP2 オプションを使って P2 リザーブド・モードのコンパイルを行う場合、コンパイラ出力とインライン・アセンブリの両方に代数アセンブリ・コードが使われます。生成されたコードには、P2 および P2+ のリザーブド・モードにおけるシリコンの例外に対する対策が含まれます。-vP2+ を使って P2 リザーブド・モードのコンパイルを行う場合、P2+ シリコンの例外に対する対策だけが含まれます。独立したランタイム・ライブラリである *rts552.lib* が、P2 および P2+ リザーブド・モードのシリコン・デバイスに提供されます。

2.3.3 シンボリック・デバッグおよびプロファイルのオプション

シンボリック・デバッグまたはプロファイルの選択に使用するオプションを次に示します。

-g または --symdebug:dwarf
C/C++ ソースレベル・デバッガが使用する疑似命令を生成します。これにより、アセンブラでのアセンブリ・ソース・デバッグが可能になります。ただし多くのコード・ジェネレータによる最適化はデバッガを混乱させるため、-g オプションにより抑止されます。-g オプションを -o オプションと一緒に使用すると、デバッグと互換性のある最適化を最大にできます（3.8 節「最適化されたコードのデバッグ」（3-14 ページ）を参照）。
DWARF デバッグ・フォーマットの詳細については、DWARF デバッグ情報フォーマットの仕様（1992-1993, UNIX International, Inc）を参照してください。

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

--profile:power
累乗プロファイラの命令コードを作成する累乗プロファイルを有効にします。

--symdebug:none
すべてのシンボリック・デバッグ出力を無効にします。このオプションは、デバッグおよび大半のパフォーマンス解析機能を妨げるため、推奨しません。

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

廃止されたシンボリック・デバッグ・オプションについては、2.3.9 項（2-30 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/C++ ソース・ファイル、アセンブリ・ソース・ファイル、またはオブジェクト・ファイルを指定できます。シェルは、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、.cpp、.cxx、または .cc	C++ ソース
.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

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

ファイル名拡張子の規則により、C および C++ ファイルのコンパイル、およびアセンブリ・ファイルのアセンブルを 1 つのコマンドで行うことが可能になります。

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

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

```
cl55 *.c
```

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

コンパイラがファイル名を解釈する方法は、オプションによって変更できます。コンパイラが認識できない拡張子を使用している場合は、-fa、-fc、-fo、および -fp オプションを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
-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 asmbly というアセンブリ言語ソース・ファイルがある場合、次のように -fa と -fc オプションを指定することにより、ファイル・タイプを正しく解釈させることができ ます。

```
cl55 -fc=file.s -fa=asmbly
```

ワイルドカードの指定とともに -f オプションは使用できません。

-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。ファイル名拡張子の規則についての詳細は、2.3.4 項（2-24 ページ）を参照してください。

2.3.6 コンパイラによるファイル名の拡張子の解釈方法とファイル作成時の拡張子の付け方の変更（-ea、-ec、-eo、-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。コマンド行では、これらのオプションは適応するファイル名の前に付ける必要があります。ど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。

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=*.ext*	アセンブリ・ソース・ファイルの場合
-ec=*.ext*	C ソース・ファイルの場合
-eo=*.ext*	オブジェクト・ファイルの場合
-ep=*.ext*	C++ ソース・ファイルの場合
-es=*.ext*	アセンブリ・リスト・ファイルの場合

拡張子の長さは 9 文字まで有効です。

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

```
cl55 -ea .rrr -eo .o fit.rrr
```

2.3.7 ディレクトリの指定方法

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

-fb*directory* 絶対リスト・ファイルに格納する宛先ディレクトリを指定します。デフォルトでは、オブジェクト・ファイルと同じディレクトリが使用されます。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。

```
cl55 -fb d:\object ...
```

-ff*directory* アセンブリ・リストとクロスリファレンス・リスト・ファイルの宛先ディレクトリを指定します。デフォルトでは、オブジェクト・ファイルのディレクトリと同じディレクトリが使用されます。アセンブリ・リスト（-al）オプションやクロスリファレンス・リスト（-ax）オプションを使わずにこのオプションを使うと、-al オプションが指定されたようにシェルが動作します。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。

```
cl55 -ff d:\object ...
```

-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。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。

```
cl55 -fr d:\object ...
```

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

```
cl55 -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。

```
cl55 -ft d:\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わない場合、大文字と小文字は区別されます（これがデフォルトです）。

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

-ahc *filename*
-hc オプションを使ってアセンブラを起動し、指定のファイルをアセンブリ・モジュールにコピーします。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 オプションを使ってアセンブラを起動し、指定のファイルをアセンブリ・モジュールにインクルードします。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呼び出し、アセンブリ・リスト・ファイルを生成します。

-apd
アセンブリ・ファイルのための前処理を実行します。ただし前処理された出力を書き込むのではなく、標準 make ユーティリティへの入力に適した依存行のリストを書き込みます。このリストは、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・ファイルのための前処理を実行します。ただし前処理された出力を書き込むのではなく、#include 疑似命令で組み込まれるファイルのリストを書き込みます。このリストは、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。

-ar*num*
num の識別するアセンブラ注釈を抑止します。注釈は、警告より厳しくないアセンブラ・メッセージです。*num* に値を指定しない場合、すべての注釈が抑止されます。

-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。

-ata （ARMS モード）このソース・ファイルの実行中に、ARMS ステータス・ビットが有効になるとアセンブラに通知します。デフォルトでは、アセンブラはビットが無効になることを前提としています。

-atb アセンブラに、パラレル・バス・コンフリクト・エラーを警告として処理させます。

-atc （CPL モード）このソース・ファイルの実行中に、CPL ステータス・ビットが有効になるとアセンブラに通知します。これにより、アセンブラは SP 相対のアドレッシング構文の使用を強制します。デフォルトでは、アセンブラはビットが無効になることを前提としています。

-ath C54x ファイルの移植時に、アセンブラが小さいコードより速いコードを生成するようにします。デフォルトでは、アセンブラは小さいコード・サイズにエンコードしようとします。

-atl （C54x 互換モード）このソース・ファイルの実行中に、C54CM ステータス・ビットが有効になるとアセンブラに通知します。デフォルトでは、アセンブラはビットが無効になることを前提としています。

-atn アセンブラに C54x の遅延した分岐または呼び出し命令の遅延スロットに位置する NOP を削除させます。

-atp 拡張子 .prf のアセンブリ命令プロファイル・ファイルを生成します。ファイルの内容は、アセンブリ・コードで使われる各種命令の使用回数です。

-ats （ニーモニック・アセンブリのみ）。リテラル・シフト・カウント・オペランドを # 文字を始めるという仕様を緩めます。これにより、ニーモニック・アセンブラの初期バージョンと互換性が生じます。このオプションを使って # を省略すると、新しい構文への変更を勧める警告が発行されます。

-att この移植された C54x ソース・ファイルの実行中に、SST ステータス・ビットが無効になるとアセンブラに通知します。デフォルトでは、アセンブラはビットが有効になることを前提としています。

-atv アセンブラに、特定の可変長命令の最大（P24）形式を使用させます。デフォルトでは、アセンブラはすべての可変長命令を最も小さいコード・サイズに解決しようとします。

-atw (代数アセンブリのみ) アセンブラ警告メッセージを抑止します。

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

-aw パイプライン・コンフリクト警告を有効にします。

-ax アセンブラに、リスト・ファイル内にシンボリック・クロスリファレンスを作成させます。

--purecirc C54x ファイルが C54x サーキュラ・アドレッシングを使うこと（C55x リニア／サーキュラ・モード・ビットは使わないこと）をアセンブラに通知します。

アセンブラの詳細は、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

2.3.9 非推奨オプション

一部のコンパイラ・オプションは非推奨になります。コンパイラは引き続きこれらのオプションを受け入れますが、その使用は推奨しません。今後リリースされるツールは、これらのオプションをサポートしません。表 2–2 に、非推奨オプションとそれぞれの説明をまとめています。

表 2–2. コンパイラ下位互換オプションのまとめ

オプション	機能
-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。
-gpp‡	累乗プロファイルの実行を有効にします。
-gt	代替 STABS デバッグ・フォーマットを使ったシンボリック・デバッグを有効にします（--symdebug:coff と同等）。
-gw†	DWARF デバッグ・フォーマットを使ったシンボリック・デバッグを有効にします。
--symdebug:coff	代替 STABS デバッグ・フォーマットを使ったシンボリック・デバッグを有効にします（-gt と同等）。このオプションは、古いデバッガやカスタム・ツールを使ったデバッグを可能にするために必要な場合があります。これらは DWARF フォーマットは読み込みません。
--symdebug:profile_coff	代替 STABS デバッグ・フォーマットを使い、シンボリック・デバッグを使った最適なコードの関数レベル・プロファイルを有効にします。

† --symdebug:dwarf オプションに置き換えられます。
‡ --profile:power オプションに置き換えられます。

2.4 環境変数の使用方法

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

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

2.4.1 ディレクトリの指定方法（C_DIR および C55x_C_DIR）

コンパイラは、C55X_C_DIR および C_DIR 環境変数を使い、#include ファイルを含む代替ディレクトリ名を指定します。コンパイラは、まず C55X_C_DIR 環境変数を検索した後、読み込み、処理します。この変数が検出されない場合、コンパイラは C_DIR 環境変数を読み込んで処理します。#include ファイルのディレクトリを指定するには、これらのコマンドの 1 つを使って C_DIR を設定します。

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

環境変数は、システムを再起動するか変数をリセットするまで保持されます。

2.4.2 デフォルトのコンパイラ・オプションの設定方法（C_OPTION および C55X_C_OPTION）

C55X_C_OPTION または C_OPTION 環境変数を使用して、コンパイラ、アセンブラ、リンカのデフォルトのオプションを設定すると便利です。この方法で設定すると、コンパイラを実行するたびに これらの変数に設定したデフォルトのオプションまたは入力ファイル名が使用されます。

C_OPTION 環境変数によりデフォルト・オプションを設定する方法は、同じオプションや入力ファイル（またはその両方）を使用して連続してコンパイラを実行する場合に便利です。コマンド行と入力ファイル名を読み込んだ後、コンパイラはまず C55X_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。C55X_C_OPTION が検出されない場合、コンパイラは C_OPTION 環境変数を読み込んで処理します。

次の表は、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=-qs -z
```

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

```
cl55    *.c               ; compiles and links
cl55    -c *.c            ; only compiles
cl55    *.c -z lnk.cmd    ; compiles/linksusing.cmdfile
cl55    -c *.c -z lnk.cmd ; only compiles (-c overrides -z)
```

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

2.5 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 C55x プリプロセッサの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。C55x 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、一目瞭然のエラー・メッセージを生成します。エラーが発生した行番号とファイル名が、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
__TMS320C55X__	常に定義されます。
__DATE__ †	*mm dd yyyy* 形式でコンパイル時刻に展開します。
__FILE__ †	カレント・ソース・ファイル名に展開します。
__LARGE_MODEL__	-ml コンパイラ・オプションが指定されると 1 に展開します。指定されない場合は未定です。
__LINE__ †	カレント行番号に展開します。
__TIME__ †	*hh:mm:ss* 形式でコンパイル時刻に展開します。
__TI_COMPILER_VERSION__	カレント・コンパイラ・バージョン番号を示す整数値に展開します。たとえば、バージョン 1.20 は、0012000 として示されます。
_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。

† ANSI/ISO 規格で定義

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

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

この場合は、次のような行に変換されます。

```
printf ("%s %s" , "13:58:17", "Jan 14 1999");
```

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) C55X_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。
  1) -I オプションで指定されたディレクトリ
  2) C55X_C_DIR または 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「-I オプションによる #include ファイル検索パスの変更」（2-35 ページ）を参照してください。C_DIR 環境変数の使用法については、2.4.1 項「ディレクトリの指定方法（C_DIR および C55x_C_DIR）」（2-31 ページ）を参照してください。

2.5.2.1 -I オプションによる #include ファイル検索パスの変更

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

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

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

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

alt.h の完全なパス名を次のように仮定します。

Windows	c:\tools\files\alt.h
UNIX	/tools/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
Windows	`cl55 -Ic:\tools\files source.c`
UNIX	`cl55 -I/tools/files source.c`

2.5.3 前処理リスト・ファイルの生成方法（-ppo オプション）

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

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

2.5.4 前処理後のコンパイルの続行方法（-ppa オプション）

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

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

-ppc オプションはコメントの除去を除いてすべての前処理機能を実行し、.pp 拡張子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 行の制御情報付き前処理リスト・ファイルの生成（-ppl オプション）

デフォルトでは、前処理された出力ファイルにはプリプロセッサ疑似命令は入っていません。#line 疑似命令を出力させたい場合は、-ppl オプションを使用してください。-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、行の制御情報（#line 疑似命令）と一緒に前処理済み出力を書き込みます。

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

-ppd オプションは前処理だけを実行します。ただし前処理された出力を書き込むのではなく、標準 make ユーティリティへの入力に適した従属行のリストを書き込みます。このリストは、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。以下の例のように、必要に応じて出力ファイル名を指定できます。

```
cl55 -ppd=make.pp file.c
```

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。以下の例のように、必要に応じて出力ファイル名を指定できます。

```
cl55 -ppi=include.pp file.c
```

2.6 診断メッセージの概要

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

"*file.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。

- **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。
- **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。
- **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)。
- **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（^ シンボル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、エラーがどのコード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

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

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

-pden 診断の数値識別子を、そのテキストと一緒に表示します。診断抑止オプション（-pds、-pdse、-pdsr、-pdsw）に指定する引数を見つけるために、このオプションを使用します。
また、このオプションは診断が自由裁量であるかどうかも表示します。自由裁量の診断とは、重大度を無効にできる診断です。自由裁量の診断には接尾部 -D が含まれ、自由裁量でない診断には接尾部がありません 。詳細については、2.6 節「診断メッセージの概要」を参照してください。

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

-pdr 注釈（軽い警告）を発行します。注釈はデフォルトでは抑止されます。

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

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

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

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

-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
```

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 #112-D: statement is unreachable
"err.c", line 12: warning #112-D: statement is unreachable
```

次に、診断識別子 112 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。

2.6.3 その他のメッセージ

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

たとえば次のとおりです。

```
cl55 -j
>> WARNING: invalid option -j (ignored)
>> ERROR: no source files
```

2.7 クロスリファレンス・リスト情報の生成（-px オプション）

-px シェル・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（.crl）が生成されます。このファイルには、ソース・ファイル内の識別子に対する参照情報が含まれています （-px オプションは -ax とは別です。-ax はコンパイラ・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・ファイル内の情報は、次の形式で表示されます。

sym-id　name　X　filename　line number　column number

sym-id　各識別子に固有に割り当てられた整数

name　識別子の名前

X　次の値のいずれか 1 つ

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

filename　ソース・ファイル

line number　ソース・ファイル内の行番号

column number　ソース・ファイル内のカラム番号

2.8 ロー・リスト・ファイルの生成方法（-pl オプション）

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

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

ロー・リスト・ファイル内の各ソース行 は、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 句）
L	ソース位置の変更。これは次の形式で表示されます。 L *line number filename key* この *line number* は、ソース・ファイル内の行番号です。*key* が表示されるのは、変更の原因が include ファイルの開始／終了の場合だけです。*key* の値は、次のとおりです。 1 = include ファイルの開始 2 = include ファイルの終了

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

表 2–5. ロー・リスト・ファイル診断識別子

診断識別子	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

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

2.9 インライン関数展開の使用方法

インライン関数を呼び出すと、その関数の C 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
- ❏ 保護される inline キーワードを使用した定義制御インライン展開

注：関数のインライン展開によりコード・サイズが大きく増大する可能性があります

展開関数 inline を使用するとコード・サイズが増大します。特に複数の場所で呼び出される関数をインライン展開する場合、増大します。関数のインライン展開は、呼び出される回数が少ない関数や、小さい関数に適しています。

2.9.1 組み込み演算子のインライン展開

C55x には多数の組み込み演算子があります。コンパイラは組み込み演算子を効率のよいコード（通常 1 つの命令）に置き換えます。インライン展開は、オプティマイザを使用するかどうかに関係なく自動的に行われます。

組み込み関数の詳細、および組み込み関数のリストについては、6.5.4 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」（6-28 ページ）を参照してください。

インライン展開できるその他の関数は以下のとおりです。

- ❏ abs
- ❏ labs
- ❏ fabs
- ❏ assert
- ❏ _nassert
- ❏ memcpy

2.9.2 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、3.6 節「自動インライン展開（-oi オプション）」（3-11 ページ）を参照してください。

2.9.3 保護されない定義制御インライン展開

inline キーワードは、標準の呼び出し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、または -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、関数呼び出しが呼び出し先関数の中のコードで置き換えられます。

例 2–1. inline キーワードの使用方法

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

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

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

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

- ❏ 関数の static inline バージョンのプロトタイプを作成します。その後、その関数の代替の静的でない外部とリンクされるバージョンのプロトタイプを作成します。例 2–2 に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・ファイルでのもので、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、-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   v x.xx                                                         */
/* Copyright (c) 2002 Texas Instruments Incorporated                         */
/*****************************************************************************/

; . . .
#ifndef   _SIZE_T
#define   _SIZE_T
typedef   unsigned int size_t;
#endif

#ifdef    _INLINE
#define   __INLINE static inline
#else
#define   __INLINE
#endif

__INLINE 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
#undef    __INLINE
#endif
```

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（続き）

(b) strlen.c

```
/*****************************************************************************/
/*  strlen v x.xx                                                            */
/*  Copyright (c) 2002  Texas Instruments Incorporated                       */
/*****************************************************************************/
#undef _INLINE
#include <string.h>

size_t strlen(const char *string)
{
   size_t n = (size_t)  1;
   const char *s = string   1;

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

2.9.5 インライン展開の制約事項

自動インライン展開と定義制御インライン展開の両方に対して、どの関数をインライン展開できるかに関していくつかの制約事項があります。ローカルな静的変数や引数の変数番号をもつ関数はインライン展開されませんが、static inline として宣言された関数は展開されます。static inline として宣言された関数の場合、ローカルな静的変数が存在しても展開が行われます。また、再帰関数やノンリーフ関数に対してはインライン展開の深さが制限されます。さらに、インライン展開は小さい関数や呼び出し箇所が少ない関数に対してのみ使用してください（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、インライン展開には不適格です。

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

2.10 インターリストの使用方法

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

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

```
cl55 -ss function.c
```

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

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

例 2–3 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。オプティマイザとともにインターリストを使用する方法の詳細は、3.7 節「インターリスト・ユーティリティをオプティマイザと組み合わせて使用する方法」（3-12 ページ）を参照してください。

例 2–3. 差し込み後のアセンブリ言語ファイル

```
       .global   _main
;
;      3 | void main (void)

;
;*************************************************************************
;* FUNCTION NAME        :   _main                                        *
;* Stack Frame          :   Compact (No Frame Pointer, w/ debug)         *
;* Total Frame Size     :   3 words                                      *
;*                          (1 return address/alignment)                *
;*                          (2 function parameters)                     *
;*************************************************************************
_main:
        AADD #-3, SP
;
;      5 | printf("Hello World\n");
;
          MOV #SL1, *SP(#0)
          CALL      #_printf  ;
                              ; call occurs [#_printf];
          AADD #3, SP
          RET                 ; return occurs
;************************************************************************
;* STRINGS                                                              *
;************************************************************************
       .sect  ".const"
       .align 1
       SL1:   .string   "Hello World",10,0
;************************************************************************
;* UNDEFINED EXTERNAL REFERENCES                                        *
;************************************************************************
       .global   _printf
```

第 3 章

コードの最適化方法

コンパイラ・ツールには最適化プログラムが含まれ、ループの簡略化、文と式の再配置、およびレジスタへの変数の割り当てなどを実行し、C/C++ プログラムの実行速度を向上させたりプログラム・サイズを縮小させたりします。

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

項目 **ページ**

3.1	オプティマイザの起動方法	3-2
3.2	ファイルレベルの最適化の実行（-O3 オプション）	3-3
3.3	プログラムレベルの最適化の実行（-pm および -O3 オプション）	3-5
3.4	最適化コード内で asm 文を使用する場合の注意	3-9
3.5	最適化コード内でエイリアスが設定された変数へのアクセス	3-10
3.6	自動インライン展開（-oi オプション）	3-11
3.7	インターリスト・ユーティリティをオプティマイザと組み合わせて使用する方法	3-12
3.8	最適化されたコードのデバッグ	3-14
3.9	実行できる最適化の種類	3-16

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。オプティマイザでは最適なコードを得るために高レベルな最適化が実行されます。

最適化を起動する最も簡単な方法は、cl55 コマンド行で -O*n* オプションを指定する方法です。*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。
- インライン関数として宣言された関数の呼び出しを展開します。

❑ **-O1**

-O0 のすべての最適化の他に、以下の最適化を実行します。

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

❑ **-O2**

-O1 のすべての最適化の他に、以下の最適化を実行します。

- ループの最適化を行います。
- グローバルな共通部分式を除去します。
- 不要でグローバルな代入を除去します。
- 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。
- 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、詳細については、3.2 節「ファイルレベルの最適化の実行（-O3 オプション）」（3-3 ページ）を参照してください。

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

3.2 ファイルレベルの最適化の実行（-O3 オプション）

-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*	3-3
最適化情報ファイルを作成する。	-on*n*	3-4
複数のソース・ファイルをコンパイルする。	-pm	3-5

3.2.1 ファイルレベルの最適化の制御（-ol オプション）

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

表 3–2. -ol オプションのレベルの選択方法

ソース・ファイルの状況 …	使用するオプション
標準ライブラリ関数と同じ名前の関数を宣言し、変更している。	-ol0
標準ライブラリ関数と同じライブラリ関数の定義を含んでいるが、標準ライブラリで宣言された関数を変更していない。	-ol1
標準ライブラリ関数を変更しないが、コマンド・ファイルや環境変数で -ol0 オプションまたは -ol1 オプションを使用する。-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

3.2.2 最適化情報ファイルの作成（-on オプション）

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

表 3–3. -on オプションのレベルの選択方法

状況 …	**使用するオプション**
情報ファイルを必要としないが、コマンド・ファイルや環境変数で -on1 オプションまたは -on2 オプションを使用している。 -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

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

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

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

コンパイラが適用しているプログラムレベルの最適化を知るには、-on2 オプションを使用して情報ファイルを作成します。詳細は、3.2.2 項「最適化情報ファイルの作成（-on オプション）」（3-4 ページ）を参照してください。

3.3.1 プログラムレベルの最適化の制御（-op オプション）

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

表 3–4.　-op オプションのレベルの選択方法

モジュールの状況	使用するオプション
他のモジュールから呼び出される関数と、他のモジュール内で変更されるグローバル変数がある。	-op0
他のモジュールから呼び出される関数はないが、他のモジュール内で変更されるグローバル変数がある。	-op1
他のモジュールから呼び出される関数も、他のモジュール内にあ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ている main を持つ関数がない。	-op0 に戻る
-op1 または -op2	割り込み関数が定義されていない。	-op0 に戻る
-op1 または -op2	FUNC_EXT_CALLED プラグマによって特定されている関数がない。	-op0 に戻る
-op3	任意の条件	-op3 のまま残る

-pm と -O3 を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、3.3.2 項「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」（3-7 ページ）を参照してください。

3.3.2 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（5.7.7 項「FUNC_EXT_CALLED プラグマ」（5-24 ページ）を参照）を配置します。

プログラムの中にアセンブリ関数が指定されている場合に使用できる別の方法は、-op オプションを -pm オプションおよび -O3 オプションと組み合わせて使用することです（3.3.1 項「プログラムレベルの最適化の制御（-op オプション）」（3-5 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++ 変数の変更もしません。

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、3.3.1 項を参照してください。

-pm -O3 オプションだけを指定してコンパイルすると、コンパイラ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 volatile キーワードを付加し、-pm -O3 -op2 を指定してコンパイルします（詳細については、5.4.6 項「volatile キーワード」（5-14 ページ）を参照してください）。

-op オプションについては、3.3.1 項を参照してください。

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

解決策　割り込みによって変更される可能性がある C 変数に volatile キーワードを追加します。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・ポイント関数とともにプラグマを使います。そうしないと、コンパイラは先頭に FUNC_EXT_CALL プラグマを指定していないエントリ・ポイント関数を除去します。
- -pm -O3 -op3 を指定してコンパイルします。FUNC_EXT_CALL プラグマを指定しないので、-op3 オプションを使用しなければなりません。このオプションは -op2 ほど強力ではなく、最適化があまり効率的でない場合もあります。

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

FUNC_EXT_CALLED についての詳細は 5.7.7 項（5-24 ページ）を、-op オプションについては 3.3.1 項を参照してください。

3.4 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。オプティマイザは、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。

3.5 最適化コード内でエイリアスが設定された変数へのアクセス

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

コンパイラは、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使用する場合は、コードを最適化する際に -ma シェル・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;
}
```

3.6 自動インライン展開（-oi オプション）

-O3 オプションを使用して最適化すると、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。-oi*size* は、size しきい値を指定します。この *size* しきい値より大きい関数は、自動的にインライン展開されることはありません。-oi*size* オプションは次の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。オプティマイザは、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 オプションを使用しない場合でも、コンパイラは非常に小さい関数をインライン展開します。

注 : -O3 の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 オプションは他の最適化をオンにします。-O3 の最適化を希望するが、自動インライン展開は希望しない場合は、-oi0 と -O3 オプションを使用します。

注 : インライン展開とコード・サイズ

展開関数 inline を使用するとコード・サイズが増大します。特に複数の場所に呼び出された関数をインライン展開する場合、増大します。関数のインライン展開は、少ない場所および小さい関数からのみ呼び出される関数に適しています。インライン展開によるコード・サイズの増大を防ぐには、-oi0 および -pi オプションを使います。これらのオプションを使うと、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。

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

オプティマイザを実行する場合（-O*n* オプション）、-os オプションと -ss オプションを指定すると、インターリスト・ユーティリティの出力を制御できます。

- -os オプションを使用すると、コンパイラのコメントがアセンブリ・ソース文に差し込まれます。
- -ss オプションと -os オプションを一緒に使用すると、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。

最適化により -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、オプティマイザはコードの中にコメントを挿入し、そのコメントにはオプティマイザ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、オプティマイザは再構築後の C/C++ の文を書き込みます。これらの文は、実行されている動作の正確な C/C++ 構文を反映しない場合があります。

例 3–1 は、最適化（-O2）と -os オプションによってコンパイルした 例 2–3（2-50 ページ）の関数を示しています。アセンブリ・ファイルの中で、アセンブリ・コードにコンパイラのコメントが差し込まれています。

例 3–1. -O2 および -os オプションを使用してコンパイルした 例 2–3 の関数

```
_main:
;** 5  ------------------ printf((char *)"Hello, world\n");
;** 5  ------------------ return;
       AADD #-3, SP
       MOV #SL1, *SP(#0)
       CALL #_printf
                        ; call occurs [#_printf]
       AADD #3, SP
       RET                ;return occurs
```

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

例 3–2 は、最適化（-O2）と -ss および -os オプションによってコンパイルした例 2–3（2-50 ページ）の関数を示しています。アセンブリ・ファイルの中で、アセンブリ・コードにコンパイラのコメントと C ソースが差し込まれています。

例 3–2. -O2、-os、および -ss オプションを使用してコンパイルした 例 2–3 の関数

```
_main:
;** 5  -----------------  printf((char *)"Hello, world\n");
;**    -----------------   return;
         AADD #-3, SP
;----------------------------------------------------------------
------------------
;   5 | printf("Hello, world\n");
;----------------------------------------------------------------
------------------
         MOV #SL1, *SP(#0)
         CALL #_printf
                        ; call occurs [#_printf]
         AADD #3, SP
         RET                ;return occurs
```

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・サイズによくない影響を及ぼします。

3.8 最適化されたコードのデバッグ

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

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

最適化されたコードをデバッグするには、-o 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。

シンボリック・デバッグを使用して最も最適化したコードを生成したい場合は、-mn オプションを使います。-mn オプションは --symdebug:dwarf や --symdebug:coff によって無効となった最適化を有効にします。しかし、-mn オプションを使うとデバッガの機能の信頼性が低くなります。

注：シンボリック・デバッグ・オプション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、パフォーマンスが低下し、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。プロファイルが推奨されない場合に --symdebug:dwarf または --symdebug:coff を使用してください。

3.8.2 最適化されたコードのプロファイル方法

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

ブレイク・ポイント・ベースのプロファイラがある場合、-O オプションとともに --profile:breakpt オプションを使います。--profile:breakpt オプションを使うと、ブレイク・ポイント・ベースのプロファイラを使う場合に、誤った動作を引き起こす最適化を無効にします。

累乗プロファイラがある場合、-O オプションとともに --profile:power オプションを使います。--profile:power オプションは、累乗プロファイラの計器コードを作成します。

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

注 : プロファイル・ポイント

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

3.9 実行できる最適化の種類

TMS320C55x 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。最適化は、コンパイラ全体を通じて様々なレベルで発生します。

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

コンパイラによって行われる最適化の例は次のとおりです。

最適化	ページ
コストに基づいたレジスタ割り当て	3-17
エイリアスの明確化	3-17
分岐の最適化と制御フローの簡略化	3-17
データ・フローの最適化	3-19
❑ 複写伝播	
❑ 共通部分式の除去	
❑ 冗長な代入の除去	
式の簡略化	3-19
ランタイムサポート・ライブラリ関数のインライン展開	3-21
誘導変数の最適化と強度換算	3-22
ループ不変コードの移動	3-22
ループの循環	3-22
自動インクリメント・アドレッシング	3-22
リピート・ブロック	3-23

3.9.1 コストに基づいたレジスタ割り当て

コンパイラは最適化が有効にされた場合、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。

3.9.2 エイリアスの明確化

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

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

3.9.3 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、分岐条件や冗長条件を除去します。到達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でき、分岐の必要が全くなくなります。

例 3–3 では、この単純な有限ステートマシンの例では、switch 文および state 変数が完全に最適化された後に一連の簡素化された条件付き分岐が残ります。

例 3–3. 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;
      }
}
```

(b) C コンパイラ出力

```
; opt55 -O3 control.if control.opt
;*****************************************************************************
;* FUNCTION NAME:_fsm                                                        *
;*****************************************************************************
_fsm:
       MOV *AR3+, AR1
       BCC L2, AR1!=#0
L1:
       MOV *AR3+, AR1
       BCC L2, AR1==#0
       MOV *AR3+, AR1
       BCC L1, AR1==#0
L2:
       MOV *AR3+,AR1
       BCC L2, AR1==#0

       RET
```

3.9.4 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。オプティマイザは、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。この値は、別の変数、定数、または共通部分式の場合もあります。その結果、定数の畳み込みや共通部分式の除去、または変数全体の除去にも十分に活用できます。例 3–3（3-18 ページ）および例 3–4（3-20 ページ）を参照してください。

- **共通部分式の除去**

  複数の式から同じ値が得られる場合、コンパイラは値を一度だけ計算し、それを保存し、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（それ以降、別の代入があるまで、または関数が終了するまで次の 参照がない変数）への不要な代入が生じます。オプティマイザは、このような不要な代入を除去します（例 3–4 を参照）。

3.9.5 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 3–4 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ます。

例 3–4. 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);
   returnb;
}
```

(b) C コンパイラ出力

```
; opt55 -O2 data.if data.opt
;*************************************************************
;* FUNCTION NAME:_simplify                                   *
;*************************************************************
_simplify:
       MOV T0, HI(AC0)
       MPYK #5,AC0,AC0
       MOV AC0, T0
       RET
```

3.9.6 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 3–5 を参照）。

例 3–5 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 3–5. 関数のインライン展開

(a) C ソース

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

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

(b) C コンパイラ出力

```
; opt55 -O3 inline.if inline.opt


   .sect ".text"
   .global _main
;*************************************************************
;* FUNCTION NAME:_main                                       *
;*************************************************************
_main:
       MOV #12,T0
       RET
```

3.9.7 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。for 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、ループ制御変数をすべて除去することができます。

3.9.8 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。

3.9.9 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から除去されます。

3.9.10 自動インクリメント・アドレッシング

*p++ の形式のポインタ式に、コンパイラは有効な C55x 自動インクリメント・アドレッシング・モードを使います。以下のようにループ内の配列を通じてコードが進むと、

```
for (i = 0; i < N; ++i) a[i]...
```

多くの場合、ループの最適化により、オート・インクリメントされたレジスタ変数ポインタを通じて、配列参照は間接的な参照に変換されます（例 3–6 を参照してください）。

この最適化は、特に C55x アーキテクチャのために設計されています。該当するデバイスのみに適用されるコード・セクションで機能するため、一般的な C コードには適用されません。

例 3–6. 自動インクリメント・アドレッシング、ループ不変コードの移動、および強度換算

(a) C ソース

```
int a[10], b[10];
void scale(int k)
{
   int i;
   for (i= 0; i < 10; ++i)
   a[i]  = b[i] * k;
}
```

(b) C コンパイラ出力

```
;***************************************************************
;* FUNCTION NAME:_scale                                        *
;***************************************************************
_scale:
         MOV #(_b & 0xffff), AR2
         MOV #(_a & 0xffff), AR3
         MOV #9, BRC0
         RPTBLOCAL L2-1
         MPYM *AR2+, T0, AC0
         MOV AC0, *AR3+
L2:
         RET
```

3.9.11 リピート・ブロック

C55x は、RPTB（リピート・ブロック）命令を使ってゼロオーバーヘッド・ループをサポートします。オプティマイザがない場合、コンパイラがカウンタの制御するループを検出し、効率的なリピート形式を使ってこれらを生成します。反復回数は、定数または式のいずれも可能です。

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、リピート・ブロックを生成できます。強度換算は、配列参照を効率的なポインタ・オート・インクリメントにします。

3.9.12 テール結合

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

例 3–7. テール結合

(a) C ソース

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

   return a;
}
```

(b) C コンパイラ出力

```
_main:
      push (DR2)
      DR2 = AR1
      if  (DR2>=#0)  goto L3
      AC0 = DR2
      AC0 = -AC0
      goto L6
L3:
      if  (DR2==#0)  goto L5
      AR1 = DR2
      call  #_f
      AR1 = AC0
      DR2 = DR2 + AR1
      goto L7
L5:
      AR1 = DR2
      call  #_g
L6:
      DR2 = AC0
      AR1 = DR2
      call  #_f
      DR1 = AC0
      AC0 = DR2
      AC0 = AC0 + (DR1 * #3)
      DR2 = AC0
L7:
      AC0 = DR2
      DR2 = pop()
      return
```

第 4 章

C/C++ コードのリンク方法

TMS320C55x™ C/C++ コンパイラとアセンブリ言語ツールを使用してユーザ・プログラムをリンクするには、次の 2 つの方法があります。

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

本章では、それぞれの方法によるリンカの起動方法について説明します。また、C/C++ コードのリンクに関する特別な要件、たとえばランタイムサポート・ライブラリの取り込み法、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、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

項目 **ページ**

4.1	リンカの起動方法（-z オプション）	4-2
4.2	リンカ・オプション	4-5
4.3	リンク・プロセスの制御方法	4-8

4.1 リンカの起動方法（-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ルした後で、独立したステップとして、またはコンパイラの一部としてリンカを起動する方法を示しています。

4.1.1 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。

cl55 -z {**-c**|**-cr**} *filenames* [*options*] [**-o** *name.out*] [*lnk.cmd*] **-l** *library*

cl55 -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。cl55 -z を使用する場合は、-c または -cr を必ず使用しなければなりません。-c オプションを指定すると、実行時に変数の自動初期化を行います。-cr オプションを指定すると、ロード時に変数の自動初期化を行います。

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

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

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

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。

-l *libraryname* （小文字の L）C/C++ ランタイムサポート関数、および浮動小数点算術関数を含んだ適切なアーカイブ・ライブラリを識別します（-l オプションは、そのファイルがアーカイブ・ライブラリであることをリンカに伝えます）。コンパイラに添付されているライブラリを使用しても、または独自のランタイムサポート・ライブラリ を作成しても構いません。リンカ・コマンド・ファイルにランタイムサポート・ライブラリを指定すると、このパラメータは不要です。

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。詳細については、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

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。

```
cl55 -z -c prog1 prog2 prog3 -o prog.out -l rts55.lib
```

4.1.2 コンパイル・ステップの一部としてリンカを起動する方法

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

cl55 [*options*] *filenames* **-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、コマンド行ですべてのファイル・オプションとコンパイラ・オプションの一番後ろに入力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、4.1.1 項「独立したステップでリンカを起動する方法」（4-2 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。これらの引数は C/C++ ソース・ファイル、アセンブリ・ファイル、コンパイラ・オプションのいずれかを指定します。これらの引数についての詳細は、2.2 節「C/C++ コンパイラの起動方法」（2-4 ページ）を参照してください。

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。

```
cl55 prog1.c prog2.c prog3.c -z -c -o prog.out -l rts55.lib
```

注：リンカにおいて引数を処理する順序

1） 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。

2） コマンド行から入力されたオブジェクト・ファイル名

3） コマンド行の -z オプションの後ろにある引数

4） C_OPTION または C55X_C_OPTION 環境変数の -z オプションの後ろにある引数

4.1.3 リンカを無効にする方法（-c オプション）

-c 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 C55X_C_OPTION 環境変数の中で -z オプションを指定しているとき、およびコマンド行でリンクを選択的に無効にする場合に特に便利です。

-c リンカ・オプションは、-c 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。

4.2 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、その影響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。絶対リスタに .out ファイルを指定するためには、使用しているリンカ・コマンド・ファイルで既に -O を使っている場合でも、-z の後で -O オプションを使う必要があります。

-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。

--args=*size* ローダが使用するメモリを割り当て、ローダのコマンド行からプログラムへ引数を渡します。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_ _ symbol 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無効にします。

-c 実行時に変数を自動初期化します。

-cr ロード時に変数を自動初期化します。

-e=*global_symbol* 出力モジュールの 1 次エントリ・ポイントを指定する *global_symbol* を定義します。

-f=*fill_value* 出力セクション内の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 16 ビットの定数です。

-g=*global_symbol* グローバル・シンボルが -h リンカ・オプションにより静的シンボルにされていても、*global_symbol* をグローバル・シンボルとして定義します。

-h すべてのグローバル・シンボルを静的シンボルにします。

-heap=*size* ヒープ・サイズ（動的メモリ割り当て）を *size* で指定したバイト数に設定し、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 2000 バイトです。

--help すべてのオプションとその使用方法について、ヘルプ画面を表示します。

-I=*directory*
ライブラリ検索アルゴリズムを変更し、デフォルト位置を検索する前に *directory* で指定したディレクトリを検索します。このオプションは -l リンカ・オプションの前に指定しなければなりません。ディレクトリ名は、オペレーティング・システムの規則に従ったものでなければなりません。**-I** オプションは 128 まで指定できます。

-j
条件付きリンクを無効にします。

-k
入力セクションの位置合わせフラグを無視します。

-l=*filename*
（小文字の L）アーカイブ・ライブラリ・ファイルの名前をリンカへの入力として指定します。*filename* はアーカイブ・ライブラリ名のため、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
ライブラリの代替検索メカニズムを提供します。-priority を使うと、該当するシンボルの定義を含むコマンド行の最初のライブラリで、未解決の参照を解決できます。

-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ます。

-stack=*size*
1 次の 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 1000 バイトです。

-sysstack *size*
2 次システム・スタック・サイズを *size* で指定したバイト数に設定し、2 次スタック・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 1000 バイトです。

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

-w
未定義の出力セクションが作成された場合にメッセージを表示します。

-x	ライブラリの再読み取りを強制実行します。後方参照を解決します。
--xml_link_info=*file*	詳細な 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、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルの「リンカの説明」の章を参照してください。

4.3 リンク・プロセスの制御方法

C/C++ プログラムをリンクするときには、リンカの起動方法に関係なく次の特別な要件があります。

- ❏ コンパイラのランタイムサポート・ライブラリを組み込む。
- ❏ 初期化モデルを設定する。
- ❏ 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、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルの「リンカの説明」の章を参照してください。

4.3.1 ランタイムサポート・ライブラリのリンク方法

C/C++ プログラムは、すべてランタイムサポート・ライブラリにリンクしなければなりません。このライブラリには、標準の C/C++ 関数、および本コンパイラ が C/C++ 環境を管理するために使用する関数も組み込まれています。使用するランタイムサポート・ライブラリを指定するには、-l リンカ・オプションを使う必要があります。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 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。

```
cl55 -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。

4.3.2 実行時の初期化

bootstrap routine と呼ばれるプログラムを初期化および実行するには、C/C++ プログラムすべてをコードとリンクする必要があります。bootstrap routine は、以下のタスクを実行します。

1) テータス・レジスタおよびコンフィギュレーション・レジスタを設定します。

2) スタックおよび 2 次システム・スタックを設定します。

3) .cinit ランタイム初期化テーブルを処理し、グローバル変数を自動的に初期化します（-c オプションを使用している場合）。

4) すべてのグローバル・コンストラクタを呼び出します（.pinit）。

5) main を呼び出します。

6) main が戻ったときに exit を呼び出します。

rts55.lib の boot.obj にある bootstrap routine の例は、_c_int00 です。*entry point* は、通常 bootstrap routine の開始アドレスに設定されます。

ライブラリに含まれている追加のランタイムサポート関数については、第 7 章「ランタイムサポート関数」で説明しています。これらの関数には、ANSI/ISO C 標準ランタイムサポートが含まれています。

注 : _c_int00 シンボル

-c または -cr リンカ・オプションを使う場合、_c_int00 は自動的にプログラムのエントリ・ポイントとして定義されます。

4.3.3 割り込みベクトルによる初期化

プログラムがロード時から実行を開始する場合、_c_int00 への分岐にリセット・ベクトルを設定する必要があります。これにより、boot.obj はライブラリからロードされ、プログラムが正しく初期化されます。サンプルの割り込みベクトルは、rts55.lib の vectors.obj にあります。C55x の場合、ベクトルの最初の数行は次のとおりです。

```
        .def  _Reset
        .ref  _c_int00
_Reset:.ivec _c_int00, USE_RETA
```

4.3.4 グローバル変数の構築

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

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

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

6.9.3 項「初期化テーブル」（6-46 ページ）では、.pinit テーブルのフォーマットについて説明しています。

4.3.5 初期化のタイプの指定方法

C/C++ コンパイラは、グローバル変数を自動的に初期化するためにデータ・テーブルを作成します。6.9.3 項「初期化テーブル」（6-46 ページ）では、これらのテーブルのフォーマットについて説明しています。これらのテーブルは .cinit という名前付きセクションに含まれています。初期化テーブルは、次のどちらかの方法で使用されます。

- ❏ 実行時に変数を自動初期化します。グローバル変数は実行時に初期化されます。-c リンカ・オプションを使用してください（6.9.4 項「実行時の変数の自動初期化」（6-49 ページ）を参照）。
- ❏ ロード時に変数を初期化します。グローバル変数はロード時に初期化されます。-c リンカ・オプションを使用してください（6.9.5 項「ロード時の変数の初期化」（6-50 ページ）を参照）。

C/C++ プログラムをリンクするときには、-c と -cr のどちらかのオプションを指定する必要があります。これらのオプションはリンカに対して、自動初期化を実行時に行うかロード時に行うかを選択するように伝えます。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 オプションを使う場合は、-z オプションに続けて使う必要があります（4.1 節「リンカの起動方法（-z オプション）」（4-2 ページ）を参照してください）。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。

注 : ブート・ローダ
ローダは C/C++ コンパイラ・ツールの一部に含まれないことに注意してください。ローダとしてのソース・デバッガとともに C55x シミュレータまたはエミュレータを使います。

4.3.6 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 *sections* と呼ばれ、さまざまなシステム構成に対応するように、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 4–1 は、それらのセクションをまとめたものです。

表 4–1. コンパイラが作成するセクション

（a）初期化されたセクション

名前	内容
.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たグローバル変数と静的定数変数および文字列リテラル
.pinit	グローバル・オブジェクト・コンストラクタのテーブル
.text	実行可能なコード
.switch	switch 文のテーブル

表 4–1. コンパイラが作成するセクション（続き）

（b）初期化されないセクション

名前	内容
.bss	グローバル変数および静的変数
.cio	C 入出力バッファ
.stack	1 次スタック
.sysstack	2 次システム・スタック
.sysmem	malloc 関数用のメモリ

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。

注 : セクションの割り振り
セクションを割り振る際、.stack および .sysstack セクションは必ず同じ 64K ワード・データ・ページにします。また、コード・セクションのみがページ境界をまたがることができます。

一般に、初期化されたセクションは ROM または RAM 内にリンクされ、初期化されないセクションは RAM 内にリンクされます。コンパイラが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、6.1.4 項「セクション」（6-4 ページ）を参照してください。リンカは、セクションを割り当てるための MEMORY 疑似命令と SECTIONS 疑似命令を備えています。メモリへのセクション割り振りの詳細は、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルの「リンカの説明」の章を参照してください。

4.3.7 リンカ・コマンド・ファイルの例

例 4–1 は、C/C++ プログラムをリンクする典型的なリンカ・コマンド・ファイルを示しています。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・オプションを示しています。プログラムをリンクするには、次のように入力します。

```
cl55 -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。これらのオプションの詳細は、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルの「リンカの説明」の章を参照してください。

例 4–1. リンカ・コマンド・ファイル

```
 stack  0x2000      /* PRIMANY STACK SIZE            */
 sysstack  0x1000   /* SECONDARY STACK SIZE          */
 heap   0x2000      /* HEAP AREA SIZE                */
 c                 /* Use C linking conventions: auto init vars at run time*/

 u _Reset           /* Force load of reset interrupt handler  */


/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY
{
 PAGE 0:  /*      Unified Program/Data Address Space      */
   RAM (RWIX): origin = 0x000100, length = 0x01FF00 /* 128Kb page of RAM */
   ROM (RIX) : origin = 0x020100, length = 0x01FF00 /* 128Kb page of ROM */
   VECS (RIX): origin = 0xFFFF00, length = 0x000100 /* 256 byte int vector */

 PAGE 2: /*        64K word I/O Address Space         */
   IOPORT (RWI) : origin = 0x000000, length = 0x020000
}
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
   .text      > ROM  PAGE 0    /* CODE                          */
 /* These sections must be on same physical memory page when */
 /* small memory model is used                                */
   .data      > RAM  PAGE 0    /* INITIALIZED VARS              */
   .bss       > RAM  PAGE 0    /* GLOBAL & STATIC VARS          */
   .const     > RAM  PAGE 0    /* CONSTANT DATA                 */
   .sysmem    > RAM  PAGE 0    /* DYNAMIC MEMORY (malloc)       */
   .stack     > RAM  PAGE 0    /* PRIMARY SYSTEM STACK          */
   .sysstack  > RAM  PAGE 0    /* SECONDARY SYSTEM STACK        */
   .cio       > RAM  PAGE 0    /* C I/O BUFFERS                 */
 /* The .switch, .cinit, and .pinit sections may be on any    */
 /* physical memory page when small memory model is used      */
   .switch    > RAM  PAGE 0    /* SWITCH STATEMENT TABLES       */
   .cinit     > RAM  PAGE 0    /* AUTOINITIALIZATION TABLES     */
   .pinit     > RAM  PAGE 0    /* INITIALIZATION FN TABLES      */

   .vectors   > VECS PAGE 0    /* INTERRUPT VECTORS             */

   .ioport    > IOPORT PAGE 2  /* GLOBAL & STATIC IO VARS       */
}
```

第 5 章

TMS320C55x C/C++ 言語

TMS320C55x™ 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（国際標準化機構）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。

C55x のサポートする C/C++ 言語は ISO/IEC 14882-1998 C++ 規格により定義され、The Annotated C++ Reference Manual (ARM) に記述されています。さらに、ISO/IEC 14882-1998 C++ 規格の多くの拡張機能が実装されています。

項目 **ページ**

5.1	TMS320C55x C の特性	5-2
5.2	TMS320C55x C++ の特性	5-5
5.3	データ型	5-6
5.4	キーワード	5-8
5.5	レジスタ変数およびパラメータ	5-15
5.6	asm 文	5-16
5.7	プラグマ疑似命令	5-17
5.8	リンク名の生成	5-32
5.9	静的変数とグローバル変数の初期化方法	5-33
5.10	ISO C 言語モードの変更方法（-pk、-pr、および -ps オプション）	5-35
5.11	コンパイラの限界	5-38

5.1 TMS320C55x 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。ISO C 基準は、International Standard ISO/IEC 9899 (1989)-Programming languages-C (C Standard) に記載されています。

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 言語の特定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 C55x C/C++ コンパイラにどのように実装されているかを説明しています。

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 C55x C/C++ コンパイラの動作を説明しています。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、C のための 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。

5.1.1 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 6.1.2, 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 5.2.1, 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 6.1.3.4, 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 6.1.3.4, K&R A2.5.2）

5.1.2 データ型

- ❑ データ型の表記方法については、5.3 節 を参照してください。（ISO 6.1.2.5, K&R A4.2）
- ❑ size_t 型（*sizeof* 演算子の結果）は unsigned int です。（ISO 6.3.3.4, K&R A7.4.8）
- ❑ ptrdiff_t 型（ポインタ減算の結果）は int です。（ISO 6.3.6, K&R A7.7）

5.1.3 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 6.2.1.3, K&R A6.3）

- ❏ 結果の型が元の値を保持するのに充分なほど大きければ、ポインタおよび整数は自由に変換できます。（ISO 6.3.4, K&R A6.6）

5.1.4 式

- ❏ 2 つの符号付き整数を除算をしたとき、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 6.3.5, 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 6.3.7, K&R A7.8）

5.1.5 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。
  （ISO 6.5.1, K&R A2.1）

- ❏ 構造体のメンバは、ワードにパックされます。（ISO 6.5.2.1, 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。
  （ISO 6.5.2.1, 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。詳細は、5.4.3 項（5-12 ページ）を参照してください。
  （TI C 拡張機能）

5.1.6 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 6.8.6, K&R A12.8）

以下のプラグマはサポートされます。

- CODE_SECTION
- C54X_CALL
- C54X_FAR_CALL
- DATA_ALIGN
- DATA_SECTION
- FAR - extaddr.h にのみ使用される
- FUNC_CANNOT_INLINE
- FUNC_EXT_CALLED
- FUNC_IS_PURE
- FUNC_IS_SYSTEM
- FUNC_NEVER_RETURNS
- FUNC_NO_GLOBAL_ASG
- FUNC_NO_IND_ASG
- INTERRUPT
- MUST_ITERATE
- UNROLL

プラグマの詳細は、5.7 節（5-17 ページ）を参照してください。

5.2 TMS320C55x C++ の特性

C55x コンパイラは、ISO/IEC 14882-1998 C++ 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。特に、iostream ライブラリはサポートされていません。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。
- 例外処理はサポートされていません。
- ランタイム型情報（RTTI）は、デフォルトで無効にされます。RTTI を使うと、オブジェクトの型をランタイム時に決定できます。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。
- 含まれている C++ 標準ライブラリ・ヘッダ・ファイルは <typeinfo> および <new> だけです。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。
- 次に示す C のライブラリ機能用の C++ ヘッダは含まれていません。
  - <clocale>
  - <csignal>
  - <cwchar>
  - <cwctype>
- reinterpret_cast 型は、クラス同士に関連がない場合、ある 1 つのクラスのメンバに対するポインタは他のクラスのメンバに対するポインタをキャストすることができません。
- 2 つのフェーズの名前をテンプレートでバインドすることは、（この規格の [temp.res] および [temp.dep] に説明されているように）できません。
- テンプレートのパラメータは実装されていません。
- テンプレート用の export キーワードは実装されていません。
- 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。

5.3 データ型

表 5–1 は、C55x 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、7.3.6 項「制限値（float.h/cfloat と limits.h/climits）」（7-19 ページ）を参照してください。

表 5–1. TMS320C55x C/C++ データ型

型	サイズ	表現	最小	最大
char、signed char	16 ビット	ASCII	-32 768	32 767
unsigned char	16 ビット	ASCII	0	65 535
short、signed short	16 ビット	2 の補数	-32 768	32 767
unsigned short	16 ビット	2 進	0	65 535
int、signed int	16 ビット	2 の補数	-32 768	32 767
unsigned int	16 ビット	2 進	0	65 535
long、signed long	32 ビット	2 の補数	-2 147 483 648	2 147 483 647
unsigned long	32 ビット	2 進	0	4 294 967 295
long long	40 ビット	2 の補数	-549 755 813 888	549 755 813 887
unsigned long long	40 ビット	2 進	0	1 099 511 627 775
enum	16 ビット	2 の補数	-32 768	32 767
float	32 ビット	IEEE 32 ビット	1.175 494e-38	3.40 282 346e+38
double	32 ビット	IEEE 32 ビット	1.175 494e-38	3.40 282 346e+38
long double	32 ビット	IEEE 32 ビット	1.175 494e-38	3.40 282 346e+38
pointers (data) small memory mode large memory mode	 16 ビット 23 ビット	 2 進	 0	 0xFFFF 0x7FFFFF
pointers (function)	24 ビット	2 進	0	0xFFFFFF

注 : C55x の Byte は 16 ビット

ISO C 定義により、演算子のサイズはオブジェクトの保存に必要なバイト数を出します。さらに ISO は、sizeof が char に適用される場合、結果が 1 になることを定めています。C55x char が 16 ビットなので（個別のアドレスを可能にするために）、1 バイトも 16 ビットです。このため、予期しない結果になることがあります。たとえば sizeof (int) == 1 (not 2) となります。C55x のバイトおよびワードは同じ（16 ビット）です。

注 : long long は 40 ビット

long long データ型は、ISO/IEC 9899 C 規格に応じて実装されます。しかし、C55x コンパイラはこのデータ型を 64 ビットでなく 40 ビットとして実装します。フォーマットされた I/O 関数とともに “ll” 長さ修飾子を使い（printf および scanf 等）、long long 変数を出力または読み込みます。たとえば次のとおりです。

```
printf("%lld\n", (long long)global);
```

5.4 キーワード

C55x C/C++ コンパイラは、標準の const、および volatile キーワードをサポートしています。さらに、C55x C/C++ コンパイラは C/C++ 言語を拡張して interrupt、ioport、および restrict キーワードをサポートしています。

5.4.1 const キーワード

C55x C/C++ コンパイラは、ISO 規格のキーワード *const* をサポートしています。このキーワードを使用すると、特定のデータ・オブジェクトに対する記憶域の割り当てをより細かく制御できます。const 修飾子を変数または配列の定義に適用すると、そ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 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）。
- ❏ オブジェクトが auto（関数スコープ）の場合。

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ことが重要です。たとえば、次の最初の文は、変数 int を指す定数ポインタ p を定義します。2 番目の文は、定数 int を指す変数ポインタ q を定義します。

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

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。

```
const int digits [] = {0,1,2,3,4,5,6,7,8,9};
```

5.4.2 ioport キー ワード

C55x プロセッサには I/O のための 2 番目のメモリ空間があります。コンパイラは、C/C++ 言語を拡張して ioport（または __ioport）キーワードを追加しています。これにより、I/O アドレッシング・モードをサポートしています。

ioport 型修飾子は、配列、構造体、共用体、および列挙法などを含む標準の型修飾子とともに使用できます。また、const および volatile 型修飾子とともに使用することもできます。配列とともに使う場合、ioport は配列型そのものでなく、配列の要素を修飾します。構造体のメンバは、ioport データへのポインタでなければ ioport を使って修飾できません。

ioport 型修飾子は、グローバル変数または静的変数にのみ適用できます。ローカルの変数は、ポインタ宣言でなければ ioport を使って修飾できません。たとえば次のとおりです。

```
void foo (void)
{
     ioport int i;  /* invalid */
     ioport int *j;  /* valid */
```

ioport を使って修飾されたポインタを宣言する際、修飾子の配置される場所によって宣言の意味が異なることに注意してください。I/O 空間が 16 ビットでアドレス可能であるため、ラージ・メモリ・モデル内でも I/O 空間へのポインタは常に 16 ビットです。

直接の ioport ポインタ引数で printf() を使えないことに注意してください。その代わり、printf() 内のポインタ引数は、次の例に示すとおり、“void *” に変換する必要があります。

```
ioport int *p;
printf("%p\n", (void*)p);
```

例 5–1 の宣言では、データ・メモリのオブジェクトを示す I/O 空間にポインタを配置しています。

例 5–1. I/O 空間内のポインタの宣言方法

(a) C ソース

```
int * ioport ioport_pointer; /* ioport pointer */
int i;
int j;

void foo (void)
{
    ioport_pointer = &i;
    j = *ioport_pointer;
}
```

例 5–1. I/O 空間内のポインタの宣言方法（続き）

(b) コンパイラ出力

```
_foo:
    MOV #_i,port(#_ioport_pointer); store addr of #i
                                      ; (I/O memory)
    MOV port(#_ioport_pointer),AR3    ; load address of #i
                                      ; (I/O memory)
    MOV *AR3,AR1            ; indirectly load value of #i
    MOV AR1,*abs16(#_j)     ; store value of #i at #j
    RET
```

typedef が例 5–1 で使われるとすると、ポインタ宣言は次のようになります。

```
typedef int *int_pointer;
ioport int_pointer ioport_pointer; /* ioport pointer */
```

例 5–2 では、I/O 空間内でデータを示すポインタを宣言しています。このポインタは、ラージ・メモリ・モデル内でも 16 ビットです。

例 5–2. I/O 空間内のデータを示すポインタの宣言方法

(a) C ソース

```
/* pointer to ioport data: */
ioport int * ptr_to_ioport;
ioport int i;

void foo (void)
{
    int j;
    i = 10;
    ptr_to_ioport = &i;
    j = *ptr_to_ioport;
}
```

(b) コンパイラ出力

```
_foo:
    MOV #_i,*abs16(#_ptr_to_ioport)   ; store address of #i
    MOV *abs16(#_ptr_to_ioport),AR3
    AADD #-1, SP
    MOV #10,port(#_i)       ; store 10 at #i (I/O memory)
    MOV *AR3,AR1
    MOV AR1,*SP(#0)
    AADD #1,SP
    RET
```

例 5–3 では、I/O 空間内でデータを示す ioport ポインタを宣言しています。

例 5–3. I/O 空間内のデータを示す ioport ポインタの宣言方法

(a) C ソース

```
/* ioport pointer to ioport data: */
ioport int * ioport iop_ptr_to_ioport;
ioport int i;
ioport int j;

void foo (void)
{
   i = 10;
   iop_ptr_to_ioport = &i;
   j = *iop_ptr_to_ioport;
}
```

(b) コンパイラ出力

```
_foo:
   MOV #10,port(#_i)       ; store 10 at #i (I/O memory)
   MOV #_i,port(#_iop_ptr_to_ioport) ; store address of
                                     ; #i (I/O memory)
   MOV port(#_iop_ptr_to_ioport),AR3 ; load addr of #i
   MOV *AR3, AR1           ; load #i
   MOV AR1,port(#_j)       ; store 10 in #j (I/O memory)
   RET
```

5.4.3 interrupt キー ワード

C55x コンパイラは、関数を割り込み関数として扱うよう指定する interrupt （または __interrupt）キーワードを追加することによって、C/C++ 言語を拡張しています。

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

void を戻すように定義した、パラメータをもたない関数に 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、スタックを自由に使用できます。たとえば次のとおりです。

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

    ...
}
```

c_int00 が C/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 シェル・オプションを使用して）厳密な 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。

5.4.4 onchip キー ワード

onchip（または __onchip）キーワードは、コンパイラに、ポインタがデュアル MAC 命令に対しオペランドとして使用される可能性があるデータを指していることを通知します。onchip パラメータを使って関数に渡されるデータ、または最終的に onchip 式によって参照されるデータは、外部メモリでなくオンチップ・メモリにリンクする必要があります。データを適切にリンクできなかった場合、BB データ・バスを介して外部メモリに参照され、バス・エラーとなります。

```
onchip int x[100];  /* array declaration   */
onchip int *p;      /* pointer declaration */
```

-mb シェル・オプションは、すべてのデータ・メモリがオンチップにあるよう指定します。

5.4.5 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 5–4 において、restrict キーワードは、関数 func1 が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 および b を使って呼び出されることは決してないことをコンパイラに通知するために使われています。ユーザは、a および b を介したアクセスが決して競合しないことを約束しています。つまり、1 つのポインタを介した書き込みは他のポインタからの読み込みに影響を与える可能性はありません。restrict キーワードの正確な意味構造は、ISO C 規格の 1999 バージョンに記載されています。

例 5–4. ポインタと restrict 型修飾子の使用

```
void func1(int * restrict a, int * restrict b)
{
   int i
   for(i=0;i<64;i++)*a++=*b++;
}
```

例 5–5 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 5–5. 配列と restrict 型修飾子の使用

```
void func2(int c[restrict], int d[restrict])
{
   int i;
   for(i = 0; i < 64; i++)
   {
     c[i] += d[i];
     d[i] += 1;
   }
}
```

5.4.6 volatile キーワード

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

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

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

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

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

5.5 レジスタ変数およびパラメータ

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

- **オプティマイザとともにコンパイル**

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

- **オプティマイザを使わずにコンパイル**

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

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

スカラ型のオブジェクト（整数、浮動小数点、ポインタ）は、レジスタ変数として宣言できます。それ以外の型のオブジェクトに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。レジスタ・パラメータは、スタックではなくレジスタに複写されます。この動作により、関数内でのパラメータへのアクセスが高速になります。

レジスタ変数の詳細については、6.3 節「レジスタ規則」（6-12 ページ）を参照してください。

5.6 asm 文

TMS320C55x C/C++ コンパイラは、C55x 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（または __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 .string 疑似命令を挿入できます。

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

挿入するコードは正しいアセンブリ言語文でなければなりません。通常のアセンブリ言語文同様に、引用符の中のコードの先頭には、ラベル、空白、タブ、あるいはコメント（* または ;）がきます。コンパイラはこの文字列のチェックはしません。エラーがある場合はアセンブラが検出します。アセンブリ言語文の詳細は、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

これらの asm 文は、通常の C/C++ 文の構文上の制約を受けません。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 文により C/C++ 環境が損なわれるのを防ぐ

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、挿入したコードの中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 文とともにオプティマイザを使用する場合は特に注意してください。オプティマイザが asm 文を削除することはありませんが、周辺のコードの順序を大きく再配置する可能性があり、予期せぬ結果を招くことがあります。

5.7 プラグマ疑似命令

プラグマ疑似命令は、コンパイラのプリプロセッサに関数の処理方法を指示します。C55x 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
- ❏ C54X_CALL
- ❏ C54X_FAR_CALL
- ❏ DATA_ALIGN
- ❏ DATA_SECTION
- ❏ FAR - extaddr.h にのみ使用される
- ❏ FUNC_CANNOT_INLINE
- ❏ FUNC_EXT_CALLED
- ❏ FUNC_IS_PURE
- ❏ FUNC_IS_SYSTEM
- ❏ FUNC_NEVER_RETURNS
- ❏ FUNC_NO_GLOBAL_ASG
- ❏ FUNC_NO_IND_ASG
- ❏ INTERRUPT
- ❏ MUST_ITERATE
- ❏ UNROLL

関数本体の内部で、引数 *func* および *symbol* を定義または宣言することはできません。プラグマは関数の本体の外部で指定しなければなりません。しかも、*func* 引数または *symbol* 引数のすべての宣言、定義、または参照の前に置かなければなりません。これらの規則に従わなかった場合、コンパイラは警告を発します。

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、プラグマを適用しようとするオブジェクトまたは関数の名前を最初の引数として指定しなければなりません。C++ では名前は省略されます。プラグマは、その後に続くオブジェクトまたは関数の宣言に適用されます。

関数にプラグマのマークを付けると、ユーザはコンパイラに、すべての状況においてその関数がプラグマの仕様を満たしていることを表明します。関数がこれらの仕様を満たしていない場合、コンパイラは予期せぬ動作をします。

5.7.1 CODE_SECTION プラグマ

CODE_SECTION プラグマは、*section name* というセクションに *symbol* のためのスペースを割り当てます。

C での構文は次のとおりです。

#pragma CODE_SECTION (*symbol, "section name"***) [;]**

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 5–6 は CODE_SECTION プラグマの使用例です。

例 5–6. CODE_SECTION プラグマの使用方法

(a) C ソース・ファイル

```
#pragma CODE_SECTION(funcA,“codeA”)
int funcA(int a)

{
    int i;
    return (i = a);
}
```

(b) アセンブリ・ソース・ファイル

```
       .sect “codeA”
       .global _funcA

;*********************************************************
;* FUNCTION NAME:_funcA                                  *
;*********************************************************
_funcA:
       RET
```

(c) C++ ソース・ファイル

```
#pragma CODE_SECTION("codeB")
int i_arg(int x) { return 1; }
int f_arg(float x) { return 2; }
```

(d) アセンブリ・ソース・ファイル

```
          .sect   "codeB"
_i_arg__Fi:
          MOV #1,T0
          RET
          .sect   ".text"
_f_arg__Ff:
          MOV #2,T0
          RET
```

5.7.2 C54X_CALL および C54X_FAR_CALL プラグマ

C54X_CALL および C54X_FAR_CALL プラグマを使うと、C55x C コードから masm55 を使って移植された C54x アセンブリ関数への呼び出しを行えます。これらのプラグマは、C54x および C55x のランタイム環境の違いを処理します。

C での構文は次のとおりです。

#pragma C54X_CALL (*asm_function*) [;]

#pragma C54X_FAR_CALL (*asm_function*) [;]

C++ での構文は次のとおりです。

#pragma C54X_CALL

#pragma C54X_FAR_CALL

function は、どうしても C55x に修正されない C54x C コンパイラで動作する C54x アセンブリ関数です。このプラグマは C 関数に適用できません。

アセンブリ関数への宣言や呼び出しの前に、適切なプラグマを置かなければなりません。したがって、ヘッダ・ファイルに指定するのが最も適しています。

C54X_FAR_CALL は、FCALL で呼び出す必要のある C54x アセンブリ関数を呼び出す時だけ使用してください。

C54x アセンブリ関数への他の呼び出しには C54X_CALL を使ってください。これには _ _far_mode シンボルを使う呼び出しが含まれます。CALL または FCALL のいずれを使うかにより、アセンブル時間を決めます。このような呼び出しは C55x では必ず CALL を使います。

コンパイラがこれらのプラグマの 1 つを検出した場合、以下のようになります。

1） 一時的に C54x 呼び出し規則および移植された C54x アセンブリ・コードに必要なランタイム環境を採用します。C54x ランタイム環境の詳細は、TMS320C54x 最適化 C コンパイラ ユーザーズ・マニュアルを参照してください。

2） 指定のアセンブリ関数を呼び出します。

3） 結果をキャプチャします。

4） 元の C55x 呼び出し規則およびランタイム環境に戻ります。

これらのプラグマは、C コードを呼び出す C54x アセンブリ関数をサポートしません。この場合、C54x および C55x のランタイム環境の違いに対処するには、アセンブリ・コードを変更する必要があります。詳細は、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルの「C54x システムを C55x システムにマージ」の章を参照してください。

C55x で実行するために移植された C54x アセンブリ・コードは、32 ビット・スタック・モードを使う必要があります。しかし、C55x ランタイム・ライブラリのリセット・ベクトル（vectors.asm 内）は、スタックを高速リターン・モードに設定します。コードで C54X_CALL または C54X_FAR_CALL プラグマを使う場合、リセット・ベクトルを次のように変更する必要があります。

1） rts.src から vectors.asm を抽出します。rts.src ファイルは、lib サブディレクトリに位置しています。

```
ar55 -x rts.src vectors.asm
```

2） vectors.asm で、次の行を変更します。

```
_Reset:.ivec _c_int00, USE_RETA
```

以下のように変更します。

```
_Reset:.ivec _c_int00, C54X_STK
```

3） vectors.asm をアセンブルし直します。

```
asm55 vectors.asm
```

4） 新しいファイルをオブジェクトおよびソース・ライブラリに配置します。

```
ar55 -r rts55.lib vectors.obj
ar55 -r rts55x.lib vectors.obj
ar55 -r rts.src vectors.asm
```

注：Code Composer Studio でインストールしたランタイム・ライブラリを修正

Code Composer Studio でインストールしたソースおよびオブジェクト・ライブラリを修正するには、これらのファイルの読み取り専用の属性をオフにする必要があります。Windows のエクスプローラで適切なランタイム・ライブラリ・ファイルを選択し、プロパティ・ダイアログを開き、読み取り専用チェックボックスのチェックを外します。

これらのプラグマは以下では使えません。

- ❏ 間接呼び出し。
- ❏ （-ml シェル・オプションで）C55x ラージ・メモリ・モデルにコンパイルされたファイル。ラージ・メモリ・モデルのデータ・ポインタは 23 ビットです。スタック上に渡されるとき、ポインタは 2 ワードを占有します。これはこれらのポインタが 1 ワードを占有する関数と矛盾します。

C55x C コンパイラは、C54x C コンパイラのグローバル・レジスタ機能をサポートしないことに注意してください。

5.7.3 DATA_ALIGN プラグマ

DATA_ALIGN プラグマは、*symbol* をアライメント境界に位置合わせします。境界は定数のワード値です。たとえば、定数 4 は 64 ビットの位置合わせを指定します。constant は 2 の累乗でなければなりません。

C での構文は次のとおりです。

#pragma DATA_ALIGN (*symbol, constant*) [;]

C++ での構文は次のとおりです。

#pragma DATA_ALIGN (*constant*) [;]

5.7.4 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。これは、.bss セクションとは別の領域内にデータ・オブジェクトをリンクする場合に便利です。

C での構文は次のとおりです。

#pragma DATA_SECTION (*symbol, "section name"***) [;]**

C++ での構文は次のとおりです。

#pragma DATA_SECTION (*"section name"***) [;]**

例 5–7 は DATA_SECTION プラグマの使用例です。

例 5–7. 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,512,0,0
          .global _bufferB
_bufferB: .usect  "my_sect",512,0,0
```

5.7.5 FAR プラグマ

FAR プラグマを使うと、ファイル・スコープで宣言されたデータ・オブジェクトの全 23 ビット・アドレスを取得できます。プラグマは、extaddr.h ヘッダにのみ使われ、P2 リザーブド・モードでのみ使用可能です。

C での構文は次のとおりです。

```
#pragma FAR (x);
```

C++ での構文は次のとおりです。

```
#pragma FAR;
```

プラグマは、例5–8 に示すように、適用するシンボルの定義の直前に来る必要があります。

このプラグマはファイル・スコープで宣言されたオブジェクトにのみ適用できます。一度適用されると、指定のオブジェクトのアドレスを取得することにより、オブジェクトの全アドレスである 23 ビット値が作成されます。この値は FARPTR（つまり unsigned long）にキャストし、ランタイムサポート関数に渡すことができます。構造化オブジェクトでは、フィールドのアドレス取得によっても全アドレスを作成できます。

例 5–8. FAR プラグマの使用方法

```
#pragma FAR(x)
int x;           /* ok */

#pragma FAR(z)
int y;          /* error - z's definition must  */
int z;          /*   immediately follow #pragma */
```

5.7.6 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。

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。

C での構文は次のとおりです。

```
#pragma FUNC_CANNOT_INLINE (func) [;]
```

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。詳細は、2.9 節「インライン関数展開の使用方法」（2-45 ページ）を参照してください。

5.7.7 FUNC_EXT_CALLED プラグマ

-pm オプションを使う場合、コンパイラはプログラムレベルの最適化を使用します。このような最適化を使用した場合、コンパイラは main から直接または間接に呼び出されない関数を除去します。main の代わりに手書きのアセンブリ・コードにより呼び出される 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、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。

C での構文は次のとおりです。

```
#pragma FUNC_EXT_CALLED (func) [;]
```

C++ での構文は次のとおりです。

```
#pragma FUNC_EXT_CALLED [;]
```

C では、引数 *func* は削除し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、引数 *func* の名前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 プラグマとともに特定のオプションを使用しなければならない場合があります。詳細は、3.3.2 項「C 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」（3-7 ページ）を参照してください。

5.7.8 FUNC_IS_PURE プラグマ

FUNC_IS_PURE プラグマはオプティマイザに対して、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、オプティマイザ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。

副次作用のある関数でこのプラグマを使うと、オプティマイザはその副次作用を除去できます。

C での構文は次のとおりです。

#pragma FUNC_IS_PURE (*func*) [;]

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。

5.7.9 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になるようにオプティマイザに対して指定します。

このプラグマは、ISO 規格に記載されている関数（strcmp または memcpy など）でのみ使用できます。これによりコンパイラは、関数の ISO 実装が修正されていないことを前提とすることができます。コンパイラは実装に関して仮説を立てることができます。たとえば、関数の使用するレジスタについての仮説を立てることができます。

修正した ISO 関数でこのプラグマを使ってはいけません。

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。

C での構文は次のとおりです。

```
#pragma FUNC_IS_SYSTEM (func)  [;]
```

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。

5.7.10 FUNC_NEVER_RETURNS プラグマ

FUNC_NEVER_RETURNS プラグマはオプティマイザに対して、あらゆる状況におい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。たとえば、無限にループする関数、exit() を呼び出す関数、またはプロセッサを停止させる関数などは、決してその呼び出し側に戻りません。プラグマが関数にマークを付けると、コンパイラはその関数のための関数エピローグ（スタックを展開するため）を生成しません。

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。

C での構文は次のとおりです。

```
#pragma FUNC_NEVER_RETURNS (func) [;]
```

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。

5.7.11 FUNC_NO_GLOBAL_ASG プラグマ

FUNC_NO_GLOBAL_ASG プラグマはオプティマイザに対して、関数が指定した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、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。

C での構文は次のとおりです。

#pragma FUNC_NO_GLOBAL_ASG (*func*) [;]

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。

5.7.12 FUNC_NO_IND_ASG プラグマ

FUNC_NO_IND_ASG プラグマはオプティマイザに対して、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、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。

C での構文は次のとおりです。

#pragma FUNC_NO_IND_ASG (*func*) [;]

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。

5.7.13 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ます。C では、引数 *func* は関数の名前です。C++ では、このプラグマは次に宣言される関数に適用されます。

C での構文は次のとおりです。

```
#pragma INTERRUPT (func) [;]
```

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。

5.7.14 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。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。このプラグマの提供する情報は、コンパイラがループのためにハードウェア・ループ（ローカルリピートまたはブロックリピート）を生成できるかどうかを決定するのに役立ちます。また、プラグマはコンパイラが不要なコードを削除するのにも役立ちます。

一般的に、ループの境界が複雑すぎる場合、コンパイラはハードウェア・ループ（ローカルリピートまたはブロックリピート）を使えません。しかし、正確な回数のループの反復で MUST_ITERATE を指定すると、コンパイラはハードウェア・ループを使ってパフォーマンスを向上し、コード・サイズを縮小できる場合があります。

また、ループの反復の最小回数が分かれば、このプラグマは便利です。MUST_ITERATE を介して最小回数を指定すると、コンパイラは、反復回数が 0 の場合にハードウェア・ループを迂回するコードを除去できます。詳細については、5.7.14.1 項「MUST_ITERATE を使ってループのコンパイラの理解を拡張する方法」(5-29 ページ) を参照してください。

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。この場合、MUST_ITERATE プラグマの 3 番目の引数 *multiple* は必ず指定する必要があります。

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 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。

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* 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); /* A 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 *multiple* を指定するときに、トリップ・カウントが *multiple* 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。

5.7.14.1 MUST_ITERATE を使ってループのコンパイラの理解を拡張する方法

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、trip_count が 16 ビット変数の場合、コンパイラはプラグマがなくてもハードウェア・ループを生成します。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。

trip_count が 32 ビット変数の場合、コンパイラはハードウェア・ループを自動的に生成することができません。この場合、MUST_ITERATE を使って反復回数の上限の指定を検討します。上限が 16 ビット値の場合（BRC0、BRC1、または CSR に対応）、コンパイラはハードウェア・ループを生成できます。

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++) { ...
```

この例は、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) { ...
```

コンパイラが、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 を使ってループが必ず 8 回実行されるよう指定すると、コンパイラはハードウェア・ループを生成できます。

```
#pragma MUST_ITERATE(8,8);
for(i2 = ipos[2]; i2 < 40; i2 += 5) { ...
```

5.7.15 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。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 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++ での構文は、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。

次のプラグマを指定すると、

```
#pragma UNROLL(1);
```

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。

5.8 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有効範囲により異なります。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。

Mangling 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__F parmcodes
```

ここで *parmcodes* は、func のパラメータ型をエンコードする一連の文字です。

次の単純な C++ ソース・ファイルがあるとします。

```
int foo(int i);   //global C++ function
```

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、foo が int 型の単独の引数を取る関数であることを示しています。検索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 9 章「C++ ネーム・デマングリング」を参照してください。

5.9 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静的変数およびグローバル（外部）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。

5.9.1 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大するという望ましくないこと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、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルのリンカの説明を参照してください。

5.9.2 const 型修飾子を使った静的変数とグローバル変数の初期化方法

明示的に初期化されていない型 *const* の静的変数およびグローバル変数は、他の静的変数およびグローバル変数に似ています。これらは 0 に事前初期化されないためです（5.9 節「静的変数とグローバル変数の初期化方法」の説明と同じ理由により）。たとえば次のとおりです。

```
const int zero;        /*  may not be initialized to 0          */
```

しかし、const グローバル変数および静的変数は .const と呼ばれるセクションで宣言および初期化されているため、初期化方法が異なります。たとえば次のとおりです。

```
const int zero = 0          /*      guaranteed to be 0*/
```

これは .const セクションのエントリに対応します。

```
        .sect     .const
_zero
        .word     0
```

この機能は、大きな定数テーブルを宣言する場合に特に便利です。システム起動時に時間もスペースも無駄にならず、テーブルを初期化できるからです。さらに、リンカは .const セクションを ROM に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;
```

上の例は、以下のアセンブリ・コードにコンパイルされます。

```
   .sect .mysect
_zero
   .word 0
```

5.10 ISO C 言語モードの変更方法（-pk、-pr、および -ps オプション）

-pk、-pr、および -ps オプションを使用すると、C/C++ コンパイラがソース・コードを解釈する方法を指定できます。ソース・コードは、次のモードでコンパイルできます。

- ❏ 標準 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。

K&R C モードは、C++ コードには適用されません。

5.10.1 K&R C との互換性（-pk オプション）

ISO C 言語は、基本的にカーニハンとリッチーの The C Programming Language で定義された、C 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。

C55x 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）が付いています。このオプションにより言語の意味上の規則を一部変更し、古いコードとの互換性に対応しています。一般には、-pk オプションの目的は、K&R C よりも厳しくなっている 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。

- 符号なし型をよりワイドな符号付き型に拡張することについて、整数拡張規則が変更になりました。K&R C では結果の型はワイド型の符号なしバージョンであり、ISO では結果の型はワイド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宣言できます。たとえば次のとおりです。

```
struct s
{
   short f :2;        /* illegal unless -pk used */
};
```

- K&R 構文では、列挙型定数リスト内に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 */
```

5.10.2 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 と矛盾する場合であってもそれを受け入れます。

5.10.3 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。このモードではあまり重要でないか、組み込みシステムでサポートするには、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、C55x 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。

5.11 コンパイラの限界

C55x C/C++ コンパイラがさまざまなホスト・システムをサポートし、これらのシステムの一部には制限があるため、コンパイラは極端に大きいまたは複雑なソース・ファイルを正しくコンパイルできない場合があります。一般的に、そのようなシステムの制限を越えると、コンパイルを続行できなくなるため、コンパイラはエラー・メッセージを出力した直後に終了します。システムの制限を越えないようにするために、プログラムを簡略化してください。

一部のシステムでは、500 文字を超えるファイル名を許可していません。ファイル名は必ず 500 文字より短くしてください。

コンパイラには任意の制限はありませんが、ホスト・システムで使用可能なメモリ容量により制限があります。PC のような小さなホスト・システムでは、オプティマイザがメモリ不足になる場合があります。その場合、オプティマイザは停止し、シェルがコード・ジェネレータでファイルのコンパイルを続けます。その結果、ファイルが最適化されずにコンパイルされます。オプティマイザは一度に 1 つの関数をコンパイルするため、この現象の原因の大半はソース・モジュール内の大きな関数や極端に複雑な関数です。この問題を修正するには、オプションを以下のようにします。

- ❏ 疑わしいモジュールを最適化しない。
- ❏ 問題を発生させた関数を識別し、小さな関数に分解する。
- ❏ モジュールから関数を抽出し、最適化せずにコンパイルできる独立したモジュールに配置して、残りの関数を最適化できるようにする。

第 6 章

ランタイム環境

本章では、TMS320C55x™ 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**

6.1	メモリ	6-2
6.2	文字列定数	6-11
6.3	レジスタ規則	6-12
6.4	関数の構造と呼び出し規則	6-16
6.5	アセンブリ言語と C/C++ 言語間のインターフェイス	6-22
6.6	割り込み処理	6-38
6.7	P2 リザーブド・モードにおけるデータの拡張アドレッシング	6-40
6.8	拡張メモリ内における .const セクション	6-43
6.9	システムの初期化	6-45

6.1 メモリ

C55x 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、24 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 : リンカのメモリ・マップ定義

コンパイラでなくリンカがメモリ・マップを定義し、コードおよびデータをターゲット・メモリに割り当てます。コンパイラは、使用できるメモリの型について、コードまたはデータに使用できない位置（ホール）や I/O または制御のために確保されている位置については、何も想定していません。コンパイラは、リンカがコードおよびデータを適切なメモリ空間に割り当てられる再配置可能なコードを作成します。

たとえば、リンカを使用して、グローバル変数を高速内部 RAM に割り当てたり、実行可能なコードを外部 ROM に割り当てたりできます。各コード・ブロックまたはデータ・ブロックをそれぞれメモリに割り当てることができますが、これは一般的な方法ではありません（この例外はメモリ・マッピングされた I/O です。ただし、C/C++ ポインタ型で物理メモリ位置にアクセスできます）。

コンパイラは小さなスモール・メモリ・モデルとラージ・メモリ・モデルをサポートします。これらのメモリ・モデルは、メモリへのデータの配置およびアクセスの方法に影響します。

6.1.1 スモール・メモリ・モデル

スモール・メモリ・モデルを使うと、コードおよびデータのサイズはラージ・メモリ・モデルの使用時より若干小さくなります。しかし、プログラムは指定のサイズおよびメモリ配置制限を満たす必要があります。

スモール・メモリ・モデルでは、次のセクションすべてを 64K ワードのサイズのメモリ 1 ページ内に収める必要があります。

- .bss および .data セクション（すべて静的およびグローバル・データ）
- .stack および .sysstack セクション（1 次および 2 次システム・スタック）
- .sysmem セクション（動的メモリ空間）
- .const セクション

.text セクション（コード）、.switch セクション（スイッチ文）、または .cinit/.pinit セクション（変数の自動初期化）のサイズまたは配置についての制限はありません。

スモール・モデルでは、コンパイラは 16 ビット・データ・ポインタを使ってデータにアクセスします。XAR*n* レジスタの上位 7 ビットは、.bss セクションを含むページをポイントするよう設定されています。これらはプログラムの実行中は同じ値に設定されたままです。

6.1.2 ラージ・メモリ・モデル

ラージ・メモリ・モデルは、制限のないデータ配置をサポートします。ラージ・メモリ・モデルを使用するには、-ml シェル・オプションを使います。

ラージ・モデルでは、

- ❏ データ・ポインタは 23 ビットで、メモリへの格納時 2 ワードを占有します。
- ❏ .stack および .sysstack セクションは同じページにある必要があります。

注 : データの配置
コード・セクションのみがページ境界をまたがることができます。セクションの他のタイプは 1 ページに収めなければなりません。

ラージ・メモリにコードをコンパイルする場合、rts55x.lib ランタイム・ライブラリとリンクする必要があります。アプリケーションではすべてのファイルに同じメモリ・モデルを使う必要があることに注意してください。リンカはラージ・メモリ・モデルとスモール・メモリ・モデルのコードの混在は受け入れません。

スモール・モデルでは、コードは 7 ビット MDPxx レジスタもすべて 0 に初期化し、すべてのデータ・アクセスが 0 ページになるようにします。

6.1.3 ヒュージ・メモリ・モデル

ヒュージ・モデルでは、オブジェクトは最大メモリ空間の限界である 8MB まで、必要な大きさにできます。ヒュージ・メモリ・モデルの利点を活かすためには、C ソース・ファイルをコンパイルし直し、手書きのコードのアセンブリ・ファイルをアセンブルし直し、標準ライブラリ関数にライブラリ rts55h.lib を使用する必要があります。C およびアセンブリ・コードの両方で、事前定義されたマクロ _ _HUGE_MODEL_ _ を使い、モデルに応じて条件付きでコードをコンパイルできます。しかし、size_t および ptrdiff_t 型を適切に使っている場合、C コードで _ _HUGE_MODEL を使う必要はありません。

ヒュージ・モデルでは、size_t および ptrdiff_t はともに 32 ビット型です。サイズの変更は、これらの型のメンバを含む構造体およびこれらの型をパラメータまたは戻り値型としてもっている関数に影響を与えます（主に memcpy、sizeof、および malloc）。ランタイムサポート・ライブラリのバイナリ・イメージに依存している場合（コプロセッサと構造体を共有するなど）、こうした構造体および関数への変更はコードに影響を与える可能性があります。

ヒュージ・モデルのパフォーマンスは、ラージ・モデルによく似ていますが、非効率的であるという点に気付きます。C55x ゼロ・オーバーヘッドが構成要素（RPT、RPTB）をループすると 16 ビット・ループ・カウントだけが許容されるため、size_t 値で反復するループが実行され、条件付き分岐は非効率的になります。しかし、16 ビットより少ない定数カウントでインライン展開できるライブラリ文字列関数への呼び出し（memcpy など）は、RPT を使って行われます。

ptrdiff_t は 32 ビット値になるため、D ユニット・アキュミュレータでポインタ比較を行う必要があります。このコードは、スモール・モデルで実行可能な以前の A ユニットより遅く、若干大きくなります。これはラージ・モデルおよびヒュージ・モデルでは避けられない点です。

ヒュージ・モデルで生成されるオブジェクト・ファイルは、ラージ・モデルおよびスモール・モデルの両方と互換性がなく、また CPU の 3.0 以前の リビジョンとも互換性がありません。コンパイラおよびリンカは、ヒュージ・モデルには CPU のリビジョン 3.0 を使うことを強制し、スモール・モデル・オブジェクトおよびラージ・モデル・オブジェクトをリンクしないようにします。

コンパイラおよびリンカは、CPU のリビジョン 1 および 2 に最適化されたオブジェクト・ファイルが CPU のリビジョン 3 にのみ最適化されたオブジェクト・ファイルと混在するのを防ぎます。デフォルトでは、CPU の特定のリビジョンに最適化することはなく、最大限に互換性のあるオブジェクト・ファイルを生成します。しかし、旧バージョンのツールで生成されたオブジェクト・ファイルは、CPU のリビジョン互換性をチェックしません。

リンカは標準コンパイラの生成したセクション（6.1.4 項を参照）内のオブジェクトが、スモール・メモリ・モデルおよび CPU のリビジョン 1 および 2 で実行できるコードのページ境界をまたぐことを防ぎます。ページ境界をまたごうとするセクションを命名するエラーが発行されます。

注：ヒュージ・メモリ・モデルに無い機能

動的メモリ割り当てシステムはまだアップデートされていないため、オブジェクト・サイズが柔軟ではありません。この機能は、今後のツール・リリース時に検討します。

グローバル・オブジェクト初期化指定子は、2^14 以上の文字オブジェクトには対応しません。これは初期化体系の制限事項であり、改善については現在検討中です。

選択したメモリ・モデルで許容された以上のオブジェクトを宣言した場合、コンパイラは警告を発行しません。

6.1.4 セクション

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。COFF セクションの詳細は、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルの「共通オブジェクトファイル・フォーマットについて」の章を参照してください。

セクションの基本型には 2 つあります。

- ❏ **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。
  - ■ **.pinit セクション** には、実行時にグローバル・オブジェクト・コンストラクタを呼び出すためのテーブルが含まれます。
  - ■ **.const セクション** には、C/C++ 修飾子 *const* で定義される文字列定数およびデータが含まれます（定数が *volatile* としても定義されていない場合）。
  - ■ **.switch セクション**には、スイッチ文のためのテーブルが含まれます。
  - ■ .**text セクション**には、実行可能なコードすべてが含まれます。
- ❏ **初期化されないセクション**は、メモリ（通常 RAM）にスペースを確保します。プログラムは、この空間を実行時に変数の作成と格納に使用できます。本コンパイラは、以下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スペースを確保します。ブート時およびロード時に、C ブート・ルーチンまたはローダは .cinit セクション（ROM に入っている場合もある）からデータをコピーし、それを使って .bss の変数を初期化します。
  - ■ **.stack セクション**は、システム・スタックにメモリを割り当てます。このメモリは変数を渡し、ローカル記憶域に使います。
  - ■ **.sysstack セクション**は、2 次システム・スタックにメモリを割り当てます。

    .stack および .sysstack セクションは同じページにある必要があることに注意してください。
  - ■ **.sysmem セクション**は、動的メモリの割り当てのためにスペースを確保します。この空間は 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。
  - ■ **.cio セクション**は、C I/O をサポートします。このスペースは、ラベル _ _CIOBUF_ とのバッファとして使われます。C I/O が実行されると（printf、scanf など）、バッファが作成されます。これには、C I/O コマンドから戻るデータと、実行されるストリーム入出力の型の内部 C I/O コマンド（および必要なパラメータ）が含まれます。C I/O を使用するためには、.cio セクションをリンカ・コマンド・ファイルに割り当てる必要があります。

コード・セクションのみがページ境界をまたがることができます。他のタイプのセクションは 1 ページに収めなければなりません。

アセンブラは .data という追加のセクションを作成します。C/C++ コンパイラはこのセクションを使用しません。

リンカは、様々なモジュールから個々のセクションを取得し、同じ名前のセクションを組み合わせます。その結果出力された、8 つのセクションと各セクションのメモリにおける適切な配置を表 6–1 に示します。これらの出力セクションは、システムの仕様に合わせて、アドレス空間のどこにでも配置できます。

表 6–1. セクションおよびメモリ配置のまとめ

セクション	メモリの型	セクション	メモリの型
.bss	RAM	.data	ROM または RAM
.cinit	ROM または RAM	.text	ROM または RAM
.pinit	ROM または RAM	.stack	RAM
.cio	RAM	.sysstack	RAM
.const	ROM または RAM	.sysmem	RAM

メモリへのセクションの割り当ての詳細は、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルの「共通オブジェクトファイル・フォーマットについて」の章を参照してください。

6.1.5 C/C++ システム・スタック

C/C++ コンパイラは、スタックを使用して以下の作業を実行します。

- ローカル変数を割り当てます。
- 関数に引数を渡します。
- プロセッサ・ステータスを保存します。

ランタイム・スタックは、単独の連続したメモリ・ブロックに割り当てられ、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、ハードウェア・スタック・ポインタ（SP）を使用してスタックを管理します。

コードはランタイム・スタックがオーバーフローしているかどうかをチェックしません。スタックが割り当てられたメモリ空間の上限を超えると、スタックがオーバーフローします。スタックに充分なメモリを割り当てるようにします。

C55x は 2 次システム・スタックもサポートします。C54x との互換性のために、1 次ランタイム・スタックは、アドレスの下位 16 ビットを保持します。2 次システム・スタックは、C55x 戻りアドレスの上位 8 ビットを保持します。コンパイラは、2 次スタック・ポインタ（SP）を使用して 2 次システム・スタックを管理します。

両方のスタックのサイズはリンカによって設定されます。リンカはグローバル・シンボル _ _STACK_SIZE および _ _SYSSTACK_SIZE も作成し、このシンボルにバイト単位で表したスタックのサイズをそれぞれ割り当てます。デフォルトのスタック・サイズは 1000 バイトです。デフォルトの 2 次システム・サイズも 1000 バイトです。リンカコマンド行で -stack または -sysstack オプションを使い、オプションの直後の定数としてサイズを指定することにより、リンク時に両方のスタックのサイズを変更できます。

注：.stack および .sysstack セクションの配置

.stack および .sysstack セクションは同じページにある必要があります。

6.1.6 動的なメモリ割り当て

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。動的な割り当ては、標準ランタイムサポート関数により提供されます。

メモリは、.sysmem セクションで定義されたグローバル・プールまたはヒープから割り当てられます。.sysmem セクションのサイズは、リンカ・コマンドで -heap *size* オプション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。デフォルトは 2000 バイトです。-heap オプションの詳細は、4.2 節「リンカ・オプション」（4-5 ページ）を参照してください。

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、ヒープ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));
```

6.1.7 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、6.9 節「システムの初期化」（6-45 ページ）を参照してください。

6.1.8 静的変数とグローバル変数へのメモリ割り当て方法

C/C++ プログラムで宣言されたすべての外部変数または静的変数に、固有の連続した空間が割り当てられます。リンカは空間のアドレスを決定します。コンパイラは、これらの変数の空間が複数のワードに割り当てられるようにします。それにより、各変数はワード境界に割り当てられます。

C/C++ コンパイラはグローバル変数がデータ・メモリに割り当てられることを予想します（.bss にそのための空間を確保します）。同じモジュールで宣言された変数は、単独の連続したメモリ・ブロックに割り当てられます。

6.1.9 フィールド／構造体の位置合わせ

コンパイラが構造体にスペースを割り当てる場合、構造体のメンバすべてを保持するのに必要なだけのワードを割り当てます。

構造体に 32 ビット（long）のメンバがある場合、long は 2 ワード（32 ビット）境界に割り当てられます。このことにより、long を位置合わせし、構造体の sizeof 値が偶数値になるようにするには、構造体の前、内部、または最後に埋め込みが必要な場合があります。

非フィールド型はすべてワード境界に位置合わせされます。フィールドには必要なだけのビット数が割り当てられます。隣接したフィールドは、1 つのワードの隣接したビットに埋め込まれます。しかし複数のワードにまたがることはありません。フィールドが次のワードにまたがってしまう場合は、フィールド全体が次のワードに配置されます。フィールドは出現順にパックされます。構造体ワードの最上位ビットが最初に埋められます。

例 6-1 に、C コードの定義と “var” のメモリ・レイアウトを示します。

例 6-1.　“Var” のフィールド／構造体の位置合わせ

（a）“var” の C コードの定義

```
struct example {
    char c;
    long l;
    int bf1:1;
    int bf2:2;
    int bf3:3;
    int bf4:4;
    int bf5:5;
    int bf6:6;
};
```

（b）“var” のメモリ・レイアウト

6.2 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。初期化の詳細は、6.9 節「システムの初期化」（6-45 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。次の例は、文字列 abc および終了バイトを定義します。ラベル SL5 はその文字列を指します。

  ```
        .const
  SL5:  .string   "abc", 0
  ```

  文字列ラベルの形式は SL*n* です。*n* はコンパイラが割り当てる番号であり、これによりラベルは一意になります。この番号は、1 から順に 1 ずつ増分して各文字列を定義します。ソース・モジュールで使用する文字列の定義は、すべてコンパイラ出力のアセンブリ言語モジュールの最後に配置されます。

  ラベル SL*n* は文字列定数のアドレスを表します。コンパイラは、このラベルにより式の文字列を参照します。

  文字列は text セクション（ほとんどの場合 ROM 内の）に格納され、共用される場合があるので、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。

  ```
  char *a = "abc";
  a[1] = 'x';           /* Incorrect! */
  ```

6.3 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 6–2 に、レジスタの使用方法および保存方法を示します。親関数は、関数呼び出しを行う関数です。子関数は、呼び出される関数です。呼び出しを通じて値を保存する方法の詳細については、6.4 節「関数の構造と呼び出し規則」（6-16 ページ）を参照してください。

コンパイルされたコードで使用されていないレジスタは、この表には載っていません。レジスタの使用について表内に指定されていない場合、レジスタはコンパイルされたコード内で各自の判断で使用されます。

表 6–2. レジスタの使用および保存の規則

レジスタ	保存する関数	使用
AC[0-3]	親	16 ビット、32 ビットまたは 40 ビットのデータ、または 24 ビット・コード・ポインタ
(X)AR[0-4]	親	16 ビットまたは 23 ビット・ポインタ、または 16 ビットのデータ
(X)AR[5-7]	子	16 ビットまたは 23 ビット・ポインタ、または 16 ビットのデータ
BK03、BK47、BKC	親	--
BRC0、BRC1	親	--
BRS1	親	--
BSA01、BSA23、BSA45、BSA67、BSAC	親	--
(X)CDP	親	‡
CFCT	子	--
CSR	親	--
(X)DP	子	‡ ラージ・メモリ・モデルでは未使用
MDP、MDP05、MDP67	子（P2 リザーブド・モード）	--
PC	該当なし	--
REA0, REA1	親	--
RETA	子	--
RPTC	親	--
RSA0、RSA1	親	--
SP	該当なし †	--
SSP	該当なし	--
ST[0-3]_55	6.3.1 項 を参照してください。	--
T0、T1	親	16 ビット・データ
T2、T3	子	16 ビット・データ
TRN0、TRN1	親	--

† SP は、スタックにプッシュしたすべてのものは戻る前にポップするという規則により保存されます。
‡ スモール・メモリ・モデルでは、アドレッシング・レジスタ（XARn、XCDP、および XDP）のすべての拡張子ビットは、子が保存し、プログラム・データのあるページのページ番号を含むと見なされます。

6.3.1 ステータス・レジスタ

表 6–3 に、ステータス・レジスタ・フィールドを示します。

推定値の欄には、以下の条件のいずれかを満たす値が含まれます。

- ❏ 関数にエントリする時または関数から戻る時に、コンパイラが該当するフィールドに対して予測する値。
- ❏ アセンブリ関数が C/C++ コードから呼び出される時に、その関数が期待する値。
- ❏ C/C++ コードへの戻り時または呼び出し時に、アセンブリ関数が設定する必要のある値。これが不可能な場合、アセンブリ関数は C/C++ 関数では使えません。

この欄のダッシュ（-）は、コンパイラが特定の値を期待しないことを示します。

修正の欄は、コンパイラの生成したコードがこのフィールドを修正するかどうかを示します。

ランタイム初期化コード（boot.asm）は、前提とする値を設定します。boot.asm の詳細は、6.9 節「システムの初期化」（6-45 ページ）を参照してください。

表 6–3. ステータス・レジスタ・フィールド

（a）ST0_55

フィールド	名前	推定値	修正
ACOV[0-3]	オーバーフロー検出	-	あり
CARRY	キャリー	-	あり
TC[1-2]	テスト制御	-	あり
DP[07-15]	データ・ページ・レジスタ	-	なし

表 6–3. ステータス・レジスタ・フィールド（続き）

(b) ST1_55

フィールド	名前	推定値	修正
BRAF	ブロック・リピート・アクティブ・フラッグ	-	なし
CPL	コンパイラ・モード	1	なし
XF	外部フラグ	-	なし
HM	保持モード	-	なし
INTM	割り込みモード	-	なし
M40	計算モード（D ユニット）	0	あり †
SATD	飽和モード（D ユニット）	0	あり
SXMD	符号拡張子モード（D ユニット）	1	なし
C16	デュアル 16 ビット算術モード	0	なし
FRCT	小数モード	0	あり
54CM	C54x 互換モード	0	あり ‡
ASM	アキュミュレータ・シフト・モード	-	なし

† 40 ビット算術の使用時（long long データ型）
‡ プラグマ C54X_CALL の使用時

(c) ST2_55

フィールド	名前	推定値	修正
ARMS	AR モード	1	なし
DBGM	デバッグ・イネーブル・マスク	-	なし
EALLOW	エミュレーション・アクセス・イネーブル	-	なし
RDM	端数処理モード	0	あり
CDPLC	CDP リニア / サーキュラ・コンフィギュレーション	0	あり
AR[0-7]LC	AR[0-7] リニア / サーキュラ・コンフィギュレーション	0	あり

(d) ST3_55

フィールド	名前	推定値	修正
CAFRZ	キャッシュ・フリーズ	-	なし
CAEN	キャッシュ・イネーブル	-	なし
CACLR	キャッシュ・クリア	-	なし
HINT	ホスト割り込み	-	なし
CBERR	CPU バス・エラー・フラグ	-	なし
MPNMC	マイクロプロセッサ／マイクロコンピュータ・モード	-	なし
SATA	飽和モード（A ユニット）	0	あり
CLKOFF	CLKOUT ディスエーブル	-	なし
SMUL	乗算飽和モード	1	あり
SST	保存飽和モード	-	なし

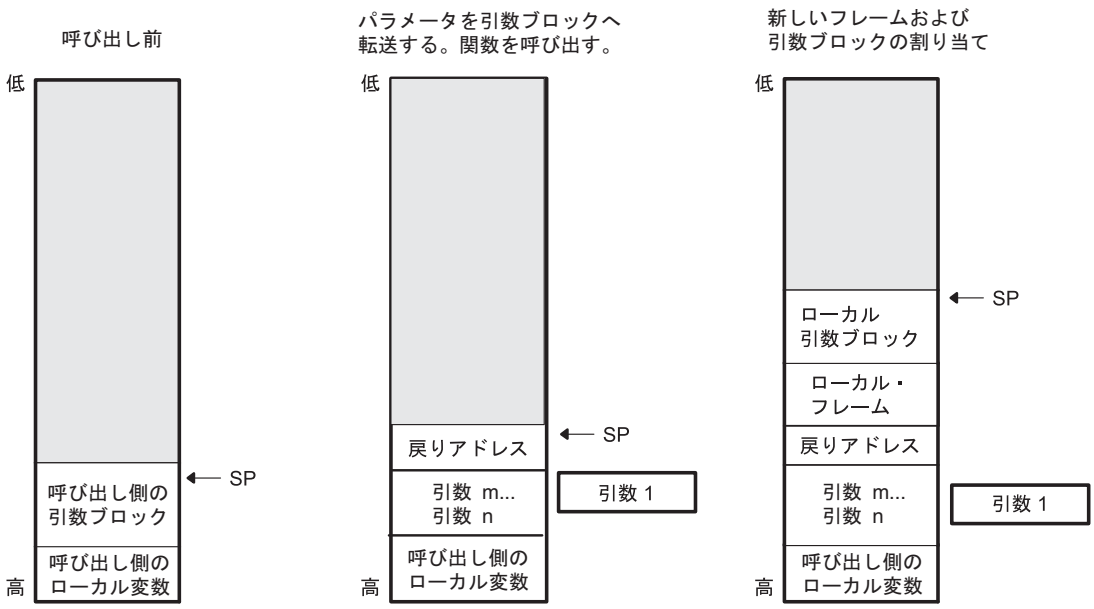
6.4 関数の構造と呼び出し規則

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 関数を呼び出すか C 関数によって呼び出されるすべての関数は、以下の規則に従わなければなりません。これらの規則に従わない場合は C/C++ 環境が損なわれ、プログラムが異常終了する恐れがあります。

図 6–1 に、典型的な関数の呼び出しを示します。この例では、パラメータは関数に渡され、関数はローカル変数を使用して他の関数を呼び出します。レジスタでは最大 10 のパラメータが渡されます。さらにこの例では、ローカル・フレームの割り当ておよび呼び出された関数の引数ブロックも示しています。スタックは、関数を呼び出す前に 32 ビット（偶数の 16 ビット・ワード）の境界に位置合わせされます。親関数は 16 ビットのプログラム・カウンタの戻り値をプッシュします。

「引数ブロック」とは、他の関数に引数を渡すために使われるローカル・フレームの部分を指します。パラメータは、スタックにプッシュされるのではなく、引数ブロックの中へ転送され関数に渡されます。ローカル・フレームおよび引数ブロックは同時に割り当てられます。

図 6–1. 関数の呼び出し中におけるスタックの使用



6.4.1 関数の呼び出し方法

関数（親関数）は、別の関数を呼び出すときに次の作業を実行します。C ルーチンは、分岐（B）命令ではなく、呼び出し（CALL）命令を使ってアクセスしなければなりません。

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。

a） 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、明示的に宣言された最後の引数がスタック上に渡され、残りの引数がその後に続きます。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。

最後の明示的な引数の前で宣言する引数は、以下に示す規則に基づきます。

b） 一般的に引数が関数に渡される場合、コンパイラは特定のクラスにこれを割り当てます。その後、そのクラスに応じて可能であればレジスタに配置します。コンパイラは 3 つのクラスを使用します。

- データ・ポインタ（int *、long * など）
- 16 ビット・データ（char、short、int）
- 32 ビット・データ（long、float、double、関数ポインタ）または 40 ビット（long long）

引数がデータ型へのポインタの場合、それはデータ・ポインタと見なされます。引数が 16 ビット・レジスタ内に収まる場合、それは 16 ビット・データと見なされます。16 ビットに収まらない場合、32 ビット・データと見なされます。40 ビット・データは下位 32 ビットだけでなく、レジスタ全体を使って渡されます。

c） 2 ワード（32 ビット）以下の構造体は、32 ビット・データ引数と同じように処理されます。使用できる場合は、レジスタに渡されます。

d） 2 ワードより大きな構造体は参照によって渡されます。コンパイラは構造体のアドレスをポインタとして渡します。このポインタは、データ・ポインタ引数と同じように処理されます。

e） 呼び出された（子）関数が構造体または共用体の値を戻す場合、親関数はそのサイズの構造体にローカル・スタック上のスペースを割り当てます。そして親関数は、そのスペースのアドレスを、隠れた最初の引数として呼び出された関数に渡します。このポインタは、データ・ポインタ引数と同じように処理されます。実際、関数の呼び出しは、

```
struct s result = fn(x, y);
```

から

`fn(&result, x, y);` に変換されます。

f) 引数をプロトタイプにリストするために、引数をレジスタに順番に割り当てます。これらは、クラスに応じ、以下のレジスタに以下に示す順序で配置されます。たとえば、最初の 32 ビット・データ引数は AC0 に配置されます。2 番目の 32 ビット・データ引数は AC 1 に配置される、と言う具合です。

引数クラス	レジスタへの割り当て
データ・ポインタ （16 または 23 ビット）	(X)AR0、(X)AR1、(X)AR2、(X)AR3、 (X)AR4
16 ビット・データ	T0、T1、AR0、AR1、AR2、AR3、AR4
32 ビット・データまたは 40 ビット・データ	AC0、AC1、AC2

AR*x* レジスタは、データ・ポインタおよび 16 ビット・データに重複します。たとえば T0 および T1 が 16 ビット・データ引数を保持し、AR0 は既にデータ・ポインタ引数を保持している場合、3 番目の 16 ビット・データ引数は AR1 に配置されます。例として、例 6-2 の 2 番目のプロトタイプを参照してください。

適切なタイプのレジスタが使用できない場合、引数はスタック上に渡されます。

注：C55x 呼び出し規則

上記はデフォルトの呼び出し規則 *c55_std* です。これより非効率的な *c55_compat* を使用している場合、呼び出し規則では最大 3 つの引数までしかレジスタに渡せません。最初の引数は AR1 または AC0 のいずれかになります。16 ビット・データまたはデータ・ポインタの場合、AR1 になります。そうでない場合は AC0 です。同様に、2 番目の引数は AR2 または AC1 となり、3 番目の引数は AR3 または AC2 となります。

g) スタック上に渡された引数は以下のように処理されます。スタックはまず偶数境界上に位置合わせされます。そして各引数は引数の型に合わせてスタック上に位置合わせされます（long、long long、float、double、コード・ポインタ、およびラージ・モデル・データ・ポインタは偶数境界です。int、short、char、ioport ポインタ、およびスモール・モデル・データ・ポインタは位置合わせされません）。必要に応じて埋め込みを挿入し、引数を正しく位置合わせします。

2) 子関数は、保存エントリ・レジスタすべてを保存します（T2、T3、AR5-AR7）。しかし、親関数は呼び出しの後に必要な値をスタックにプッシュすることにより、他のレジスタの値を保存する必要があります。

3) 親関数が、関数を呼び出します。

4） 親関数が戻り値を収集します。

- 短いデータ値は T0 に戻されます。
- 長いデータ値は AC0 に戻されます。
- データ・ポインタ値は (X)AC0 に戻されます。
- 子関数が構造体を戻す場合、構造体はステップ 1 の通りに割り当てられたスペースのローカル・スタックにあります。

レジスタ引数規則の例は、例 6-2 のとおりです。例にあるレジスタは、ステップ 1 から 4 の規則に従って割り当てられます。

例 6-2. レジスタ引数の規則

```
struct big { long x[10]; };
struct small { int x; };
T0      T0      AC0     AR0
int fn(int i1, long l2, int *p3);
AC0     AR0     T0      T1      AR1
long fn(int *p1, int i2, int i3, int i4);
AR0             AR1
struct big fn(int *p1);
T0      AR0             AR1
int fn(struct big b, int *p1);
AC0             AR0
struct small fn(int *p1);
T0              AC0     AR0
int fn(struct small b, int *p1);
T0          stack       stack...
int printf(char *fmt, ...);
        AC0     AC1     AC2     stack   T0
void fn(long l1, long l2, long l3, long l4, int i5);
        AC0     AC1     AC2     AR0     AR1
void fn(long l1, long l2, long l3, int *p4, int *p5,
        AR2     AR3     AR4     T0      T1
        int *p6, int *p7, int *p8, int i9, int i10);
```

6.4.2 呼び出し先関数の対応方法

呼び出し先関数は、以下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スペースをスタック上に割り当てます。この割り当ては、関数の最初で一度発生します。

2) 子関数が保存エントリ・レジスタ（T2、T3、AR5-AR7）を変更する場合、スタックまたは未使用レジスタのいずれかに値を保存する必要があります。呼び出し先関数は、値を保管せずに他のレジスタ（ACx、Tx、ARx）を変更できます。

3) 子関数が構造体引数を要求する場合、関数は構造体を指すポインタを受け取ります。呼び出し先関数内で構造体への書き込みが行われる場合は、その構造体のローカル・コピー用のスペースをスタック上に割り当てなければなりません。構造体のローカル・コピーは、渡されたポインタから作成する必要があります。構造体に書き込みが行われない場合は、ポインタ引数を通して間接的に呼び出し先関数で参照することができます。

4) 子関数は、関数のコードを実行します。

5) 子関数が値を戻す場合、長いデータ値は AC0 に、短いデータ値は T0 に、データ・ポインタは (X)AR0 にそれぞれ配置します。

   子関数が構造体を戻す場合、親関数はその構造体用のスペースを割り当ててから、(X)AR0 内のこのスペースにポインタを渡します。構造体を戻すには、呼び出し先関数が、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。

   親関数が戻りの構造体値を使用しない場合は、アドレス値 0 を (X)AR0 に渡すことができます。この 0 は子関数に対して、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、(呼び出し元が最初の引数を正しく設定するように）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）定義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。

6) 子関数はステップ 2 で保存したレジスタをすべて復元します。

7) 子関数はスタックを元の値に復元します。

8) 関数がリターンします。

6.4.3 引数とローカルへのアクセス方法

コンパイラは、コンパイラ・モード（ステータス・レジスタ ST1_55 の CPL ビットが 1 に設定された場合に選択される）を使って引数とローカルにアクセスします。このビットが設定されると、命令の dma フィールドの定数を SP に追加することにより、直接アドレッシング・モードがデータ・アドレスを計算します。たとえば次のとおりです。

```
MOV *SP(#8), T0
```

このアドレッシング・モードで使用できる最大のオフセットは 128 です。したがって、オブジェクトが SP から遠すぎてこのアクセス・モードを使用できない場合、コンパイラは関数 prolog で SP を AR6（FP）にコピーし、長いオフセット・アドレッシングを使ってデータにアクセスします。たとえば次のとおりです。

```
MOV SP, FP
...
MOV *FP(#130), AR3
```

6.5 アセンブリ言語と C/C++ 言語間のインターフェイス

アセンブリ言語を C/C++ コードと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（6.5.1 項を参照）。これは最も汎用性のある方法です。
- ❑ 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（6.5.2 項（6-24 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（6.5.3 項（6-27 ページ）を参照）。
- ❑ C/C++ ソースで組み込み関数を使用して、アセンブリ言語文を直接呼び出します（6.5.4 項（6-28 ページ）を参照）。

6.5.1 C/C++ コードでのアセンブリ言語モジュールの使用方法

6.3 節「レジスタ規則」（6-12 ページ）および 6.4 節「関数の構造と呼び出し規則」（6-16 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。

- ❑ 関数によって変更される専用のレジスタは、すべて保存する必要があります。専用のレジスタには次のものが含まれます。
  - ■ 子関数が保存したレジスタ（T2、T3、AR5、AR6、AR7）
  - ■ スタック・ポインタ（SP）

  SP を通常どおりに使用する場合は、明示的に保存する必要はありません。つまり、スタック上にプッシュされたものがすべて関数が戻る前にポップされる限り（つまり SP の保存）、アセンブリ関数は自由にスタックを使用できます。

  専用でないレジスタは、すべて最初に保存することなく自由に使用できます。
- ❑ コンパイルされた C55x コードから C54x アセンブリ関数を呼び出すには、C54X_CALL または C54X_FAR_CALL プラグマを使います。詳細は、5.7.2 項「C54X_CALL および C54X_FAR_CALL プラグマ」（5-19 ページ）を参照してください。
- ❑ 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、6.6 節「割り込み処理」（6-38 ページ）を参照してください。

- C/C++ 関数を呼び出す場合、専用のレジスタだけが保存されることに注意してください。C/C++ 関数は、他のどのレジスタの定数も変更できます。

- コンパイラは、スタックが起動時に偶数ワード・アドレスに初期化されることを前提としています。アセンブリ関数が C/C++ 関数を呼び出す場合、SP から奇数の数を引くことにより、SP を位置合わせする必要があります。同じ数を SP に加え、関数の最後でスペースを割り当て解除する必要があります。たとえば次のとおりです。

```
_func: AADD #-1, SP     ; aligns SP
       ...              ; body of function
       AADD #1, SP      ; deallocate stack space
       RET              ; return from asm function
```

- long および float は、最上位のワードが下位アドレスになるようにメモリに保存されます。

- 関数は、6.4.2 項「呼び出し先関数の対応方法」（6-20 ページ）で記載されたとおりに値を戻さなければなりません。

- アセンブリ言語モジュールは、グローバル変数の自動初期化以外の目的に .cinit セクションを使用することができません。boot.asm の C/C++ 始動ルーチンは、.cinit セクションがすべて初期化テーブルで構成されているものと見なします。それ以外の情報を .cinit に入れ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、すべての識別子の先頭に下線（_）を付けます。このネーム・スペースは、コンパイラによって予約されています。C/C++ からアクセスできる変数および関数の名前の前に _ を付けます。たとえば、x という C/C++ 変数は、アセンブリ言語では _x と呼ばれます。

  アセンブリ言語モジュール（1 つまたは複数）の中でのみ使用される識別子の場合は、下線で始めることはできません。

- アセンブリ言語内で宣言し C/C++ からアクセスまたは呼び出しが行われるオブジェクトや関数は、すべてアセンブラの中で .global 疑似命令を使用して宣言しなければなりません。これにより、シンボルが外部シンボルとして定義され、リンカはそのシンボルへの参照を解決できます。

  同様に、アセンブリ言語から C/C++ 関数またはオブジェクトにアクセスする場合には、C/C++ オブジェクトを .global によって宣言します。これにより、リンカが解決できる未定義の外部参照が作成されます。

- コンパイルされたコードは CPL（コンパイラ・モード）を 1 に設定して実行されるので、直接アドレッシングされたオブジェクトにアクセスする唯一の方法は、間接絶対モードを使う方法です。たとえば次のとおりです。

```
MOV *(#global_var),AR3 ; works with CPL = = 1
MOV global_var, AR3    ; does not work with CPL ==1
```

アセンブリ言語関数で CPL ビットを 0 に設定すると、コンパイルされたコードに戻る前に 1 に設定し直す必要があります。

例 6-3.　C からのアセンブリ言語関数の呼び出し方法

(a) C プログラム

```
extern void asmfunc(int *);
int global;

void func()
{
       int local = 5;
       asmfunc(&local);
}
```

(b) アセンブリ言語出力

```
_func:
         AADD #-1, SP
         MOV XSP, XAR0
         MOV #5, *SP(#0)
         CALL #_asmfunc
         AADD #1, SP
         RET
```

(c) asmfunc のためのアセンブリ言語ソース・コード

```
_asmfunc:
          MOV *AR0, AR1
          ADD *(#_global), AR1, AR1
          MOV AR1, *(#_global)
          RET
```

例 6-3 のアセンブリ言語コードでは、アセンブリ・コードで使われている C/C++ シンボル名の下線に注意してください。

6.5.2　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。この動作を実行するために使える方法は、定義する項目の場所と方法に応じて 3 とおりあります。.bss セクションで定義する変数、.bss セクションで定義しない変数、または定数です。

6.5.2.1 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） .global 疑似命令を使用して、その定義を外部定義にします。

3） アセンブリ言語の中で、名前の前に下線を付けます。

4） 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 6-4 は、.bss の中で定義された変数に C からアクセスする方法を示しています。

例 6-4. 変数に C からアクセスする方法

(a) アセンブリ言語プログラム

```
* Note the use of underscores in the following lines

   .bss      _var,1      ; Define the variable
   .global   _var        ; Declare it as external
```

(b) C プログラム

```
extern int var;    /* External variable              */
var = 1;           /* Use the variable               */
```

変数が必ず .bss セクションに欲しい訳ではありません。たとえば一般的な状況として、アセンブリ言語に定義された照合テーブルを RAM に入れたくない場合があります。この場合、オブジェクトへのポインタを定義し、間接的に C/C++ からアクセスする必要があります。

最初のステップはオブジェクトを定義することです。これを初期化されたセクションに入れると便利です（必ずしも入れる必要はありません）。オブジェクトの最初を示すグローバル・ラベルを宣言すると、オブジェクトを任意のメモリ空間にリンクできます。C/C++ でこれにアクセスするには、オブジェクトを *extern* として宣言し、その前に下線を付けてはなりません。すると、通常の方法でオブジェクトにアクセスできます。

例 6-5 は、.bss の中で 定義されていない変数にアクセスする方法を示しています。

例 6-5.　.bss で定義されていない変数に C からアクセスする方法

（c）C プログラム

```
extern float sine[];   /* This is the object              */
float *sine_p = sine;  /* Declare pointer to point to it  */
f = sine_p[4];         /* Access sine as normal array     */
```

（d）アセンブリ言語プログラム

```
   .global   _sine        ; Declare variable as external
   .sect     "sine_tab"   ; Make a separate section
_sine:                    ; The table starts here
   .float    0.0
   .float    0.015987
   .float    0.022145
```

6.5.2.2　アセンブリ言語定数へのアクセス

.set 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 6-6 に示すように、キャストと #define を使用できます。

例 6-6.　C からアセンブリ言語定数にアクセスする方法

(a) 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
```

(b) 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 */
```

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 6-6 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。

6.5.3　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。連続して asm 文を使用すれば、他のコードを間に入れることなくコンパイラ出力の中にアセンブリ言語の連続した行を挿入できます。詳細は、5.6 節「asm 文」(5-16 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++ 環境が損なわれないよう特に注意してください。コンパイラは挿入された命令をチェックまたは解析しません。
- ❑ ジャンプまたはラベルを C/C++ コードに挿入すると、コード・ジェネレータの使用するレジスタ・トラッキング・アルゴリズムを混乱させることにより、予期せぬ結果が生じる場合があります。
- ❑ asm 文を使用する際には、C/C++ 変数の値を変更してはなりません。
- ❑ asm 文を使って、アセンブリ環境を変更するアセンブラ疑似命令を挿入してはなりません。

6.5.4 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

コンパイラは、多くの組み込み関数を認識します。組み込み関数は、関数と同様に使われ、C/C++ では他の方法によって表現できないようなアセンブリ文を作成します。通常の関数で使用する場合と全く同じように、これらの組み込み関数でも C/C++ 変数を使用できます。組み込み関数は名前の前に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);
```

組み込み関数演算子の多くは、飽和をサポートします。算術演算の飽和中、オーバーフローしている式には、妥当な極値として、式が保持できる最大値または最小値のいずれかが与えられます。たとえば上記の式で x1==x2==INT_MAX の場合、式はオーバーフローし飽和しますが、y に INT_MAX の値が与えられます。次の命令を使い、飽和ビット ST1_SATD を設定することにより飽和は制御されます。

```
BSET ST1_SATD
BCLR ST1_SATD
```

飽和している算術演算と非飽和の算術演算を混在させるには、コンパイラはこのビットのオン・オフを切り替える必要があります。しかし、同じ操作で命令のブロックを認識することにより、このような命令のビット変更の回数を最小限に抑えます。効率を最大化するために、オーバーフローしている場合に値を飽和させる必要があるとき、およびオーバーフローが発生する可能性のあるときの操作のためだけに、飽和組み込み演算子を使います。ループ繰り返しカウンタにこれらを使用しないでください。

コンパイラは、結合しやすい加算と乗算アキュミュレート組み込み関数をサポートします。これらの結合組み込み関数の前には “_a_” が付きます。コンパイラは、結合組み込み関数を含む算術演算を並べ替えることができ、より効率的なコードを作成できます。

たとえば次のとおりです。

```
int x1, x2, x3, y;
y = _a_sadd(x1, _a_sadd(x2, x3)); /* version 1 */
```

上記はコンパイラ内部で次のように並べ替えることができます。

```
y = _a_sadd(_a_sadd(x1, x2), x3); /* version 2 */
```

しかし、この並べ替えにより新しい順序の様々な場所で飽和が発生すると、式の値に影響する場合があります。たとえば、x1==INT_MAX、x2==INT_MAX、および x3==INT_MIN の場合、式のバージョン 1 は飽和せず、y は（INT_MAX-1）と等しくなります。しかしバージョンは飽和し、y は -1 になります。経験則からすると、すべてのデータの符号が同じであれば結合組み込み関数を安全に使うことができます。

乗算組み込み演算子の大半は、小数モード算術演算で動作します。したがって、オペランドは *Q15* 固定小数点値となり、結果は *Q31* 値となります。操作上、これは通常の乗算の結果を 1 つ左にシフトし、*Q31* 値に正規化することを意味しています。このモードは、小数モード・ビット ST1_FRCT によって制御されます。

表 6–7（6-33 ページ）の組み込み関数は特別で、値へのポインタおよび参照を受け取ります。引数は値でなく参照によって渡されます。これらの値は修正可能な値（たとえば定数や算術式でなく、変数）です。これらの組み込み関数は値を戻しません。参照によって渡された値を変更することにより、結果を作成します。これらの組み込み関数は C++ 参照構文に依存していますが、C++ の意味構造を使い、C コードで使用可能です。

組み込み関数の宣言は不要ですが、宣言はコンパイラとともに含まれているヘッダ・ファイル C55x.h に提供されます。

組み込み演算子の大半は、ヨーロッパ電気通信規格研究所（ETSI）のグローバル・システム・フォー・モバイル・コミュニケーションズ（GSM）に記載された基本的な DSP 関数の実行に便利です。これらの関数は、コンパイラとともに含まれる、ヘッダ・ファイル gsm.h で実行されてきました。ETSI GSM 関数についての詳細は、6.5.4.1 項「組み込み関数および ETSI 関数」（6-35 ページ）に説明があります。

表 6–4（6-30 ページ）から表 6–8（6-34 ページ）に、TMS320C55x C/C++ コンパイラの組み込み演算子をすべて示しています。それぞれの組み込み関数には各パラメータの型を示す関数プロトタイプがあります。引数の型がパラメータと一致しない場合、引数上で型の変換が実行されます。引数の順序が重要な場合、組み込み関数の入力引数の順序は、基本的なハードウェア命令の順序に一致します。その結果、アセンブリ言語のニーモニックが各命令に与えられます。MPY など一部の命令には、効率性により、SQR などの代替命令（特化された MPY）が生成される場合があります。それぞれの命令について簡単に説明しています。アセンブリ命令についての正確な定義については、TMS320C55x DSP ニーモニック命令セット リファレンス・ガイド（SPRU374）および TMS320C55x DSP 代数命令セット リファレンス・ガイド（SPRU375）を参照してください。

表 6–4. C55x C/C++ コンパイラの組み込み関数（加算、減算、否定演算、絶対値）

コンパイラ組み込み関数		アセンブリ命令	説明

int
int
long
long
long long
long long
	_sadd(int src1, int src2);
	_a_sadd(int src1, int src2);
	_lsadd(long src1, long src2);
	_a_lsadd(long src1, long src2);
	_llsadd(long long src1, long long src2);
	_a_llsadd(long long src1, long long src2);
		ADD	オペランドの飽和した合計を戻します。

int
long
long long
	_ssub(int src1, int src2);
	_lssub(long src1, long src2);
	_llssub(long long src1, long long src2);
		SUB	式の飽和した値を戻します（src1 - src2）。

int
long
long long
	_sneg(int src);
	_lsneg(long src);
	_llsneg(long long src);
		NEG	式の飽和した値を戻します（0 - src）。

int
long
long long
	_abss(int src);
	_labss(long src);
	_llabss(long long src);
		ABS	オペランドの飽和した絶対値を戻します。

表 6–5. C55x C/C++ コンパイラの組み込み関数（乗算、シフト）

コンパイラ組み込み関数		アセンブリ命令	説明
int long	_smpy(int src1, int src2); _lsmpy(int src1, int src2);	MPY	オペランドの飽和した小数モードの結果を戻します。
long	_lsmpyr(int src1, int src2);	MPYR	オペランドの飽和した小数モードの結果を戻し、組み込み関数 _sround を使用したように丸めます。
long long	_smac(long src1, int src2, int src3); _a_smac(long src1, int src2, int src3);	MAC	飽和した src1 と、src2 および src3 の小数モードの結果の合計を戻します。モード・ビット SMUL も設定されます。
long long	_smacr(long src1, int src2, int src3); _a_smacr(long src1, int src2, int src3);	MACR	飽和した src1 と、src2 および src3 の小数モードの結果の合計を戻します。合計は、組み込み関数 _sround を使用したように丸められます。モード・ビット SMUL も設定されます。
long long	_smas(long src1, int src2, int src3); _a_smas(long src1, int src2, int src3);	MAS	飽和した src1 と、src2 および src3 の小数モードの結果の差を戻します。モード・ビット SMUL も設定されます。
long long	_smasr(long src1, int src2, int src3); _a_smasr(long src1, int src2, int src3);	MASR	飽和した src1 と、src2 および src3 の小数モードの結果の差を戻します。合計は、組み込み関数 _sround を使用したように丸められます。モード・ビット SMUL も設定されます。
int long	_sshl(int src1, int src2); _lsshl(long src1, int src2);	SFTS	式の飽和した値を戻します（src1 << src2）。src2 が負の場合、右シフトが行われます。
int long	_shrs(int src1, int src2); _lshrs(long src1, int src2);	SFTS	式の飽和した値を戻します（src1 >> src2）。src2 が負の場合、左シフトが行われます。

int
long
long long
	_shl(int src1, int src2);
	_lshl(long src1, int src2);
	_llshl(long long src1, int src2);
		SFTS	式を戻します（src1 << src2）。src2 が負の場合、右シフトが行われます。飽和は実行されません。

表 6–6. C55x C/C++ コンパイラの組み込み関数（丸め処理、飽和、ビットカウント、極値）

コンパイラ組み込み関数		アセンブリ命令	説明
long	_round(long src);	ROUND	（無限に正の方向へ丸められた）飽和していない算術演算を使い、2^15 を加え下位 16 ビットをクリアすることにより丸められた src の値を戻します。Q31 の結果の上位 16 ビットは、Q15 の値として処理できます。
long long	_sround(long src); _rnd(long src);	ROUND	（無限に正の方向へ丸められた）飽和している算術演算を使い、2^15 を加え下位 16 ビットをクリアすることにより丸められた src の値を戻します。Q31 の結果の上位 16 ビットは、Q15 の値として処理できます。
long	_roundn(long src);	ROUND	飽和していない算術演算を使い、最も近い 2^16 の倍数に丸められ下位 16 ビットをクリアした src の値を戻します。偶数に丸められることにより、接続は切れます。Q31 の結果の上位 16 ビットは、Q15 の値として処理できます。
long	_sroundn(long src);	ROUND	飽和している算術演算を使い、最も近い 2^16 の倍数に丸められ下位 16 ビットをクリアした src の値を戻します。偶数に丸められることにより、接続は切れます。Q31 の結果の上位 16 ビットは、Q15 の値として処理できます。
int int	_norm(int src); _lnorm(long src);	EXP	src を 32 ビット長の値に正規化するために必要な左シフト・カウントを戻します。このカウントは負の場合があります。
long	_lsat(long long src);	SAT	飽和した src を 32 ビット長の値に戻します。src が既に long の許容する範囲内にあった場合、値は変更しません。範囲内でなかった場合、戻った値は LONG_MIN または LONG_MAX のいずれかです。
int	_count(unsigned long long src1, unsigned long long src2);	BCNT	式に設定されたビット数を戻します（src1 および src2）。

int
long
long long
	_max(int src1, int src2);
	_lmax(long src1, long src2);
	_llmax(long long src1, long long src2);
		MAX	src1 および src2 の最大値を戻します。

int
long
long long
	_min(int src1, int src2);
	_lmin(long src1, long src2);
	_llmin(long long src1, long long src2);
		MIN	src1 および src2 の最小値を戻します。

表 6–7. C55x C/C++ コンパイラの組み込み関数（副次作用のある算術演算）

コンパイラ組み込み関数	アセンブリ命令	説明

void _firs(int *, int *, int *, int&, long&);
void _firsn(int *, int *, int *, int&, long&);
	FIRSADD
	FIRSSUB
		以下に示す対応する命令を実行します。
		
		int *p1, *p2, *p3, srcdst1;
		long srcdst2;
		...
		_firs(p1, p2, p3, srcdst1, srcdst2);
		_firsn(p1, p2, p3, srcdst1, srcdst2);
		
		それぞれ以下のようになります。
		
		FIRSADD *p1, *p2, *p3, srcdst1, srcdst2
		FIRSSUB *p1, *p2, *p3, srcdst1, srcdst2
		
		モード・ビット SATD、FRCT、および M40 は 0 です。

void _lms(int *, int *, int&, long&);	LMS	以下に示す LMS 命令を実行します。 int *p1, *p2, srcdst1; long srcdst2; ... _lms (p1, p2, srcdst1, srcdst2); それぞれ以下のようになります。 LMS *p1, *p2, srcdst1, srcdst2 モード・ビット SATD、FRCT、RDM および M40 は 0 です。

void _abdst(int *, int *, int&, long&);
void _sqdst(int *, int *, int&, long&);
	ABDST
	SQDST
		以下に示す対応する命令を実行します。
		
		int *p1, *p2, srcdst1;
		long srcdst2;
		...
		_abdst(p1, p2, srcdst1, dst);
		_sqdst(p1, p2, srcdst1, dst);
		
		それぞれ以下のようになります。
		
		ABDST *p1, *p2, srcdst1, srcdst2
		SQDST *p1, *p2, srcdst1, srcdst2
		
		モード・ビット SATD、FRCT、および M40 は 0 です。

表 6–7. C55x C/C++ コンパイラの組み込み関数（副次作用のある算術演算）（続き）

コンパイラ組み込み関数	アセンブリ命令	説明
int _exp_mant(long, long&);	MANT:: NEXP	以下に示す MANT::NEXP 命令のペアを実行します。 int src, dst2; long dst1; ... dst2 = _exp_mant(src, dst1); 以下のようになります。 MANT src, dst1 ::NEXP src, dst2

void _max_diff_dbl(long, long,
long&, long&,
unsigned &);
void _min_diff_dbl(long, long,
long&, long&,
unsigned &);
	DMAXDIFF
	DMINDIFF
		以下に示す対応する命令を実行します。
		
		long src1, src2, dst1, dst2;
		int dst3;
		...
		_max_diff_dbl(src1, src2, dst1, dst2, dst3);
		_min_diff_dbl(src1, src2, dst1, dst2, dst3);
		
		それぞれ以下のようになります。
		
		DMAXDIFF src1, src2, dst1, dst2, dst3
		DMINDIFF src1, src2, dst1, dst2, dst3

表 6–8. C55x C/C++ コンパイラの組み込み関数（非算術演算）

コンパイラ組み込み関数	アセンブリ命令	説明
void _enable_interrupts(void); void _disable_interrupts(void);	BCLR ST1_INTM BSET ST1_INTM	割り込みを有効または無効にし、充分なサイクルが消費されて他のことが起こる前に変更が有効になるようにします。

6.5.4.1 組み込み関数および ETSI 関数

表 6–9 の関数は、ETSI GSM 関数をさらにサポートします。関数 L_add_c、L_sub_c、および L_sat は、GSM インライン・マクロにマップされます。表内の他の関数は、ランタイム関数です。

表 6–9. ETSI サポート関数

コンパイラ組み込み関数	説明
long L_add_c(long src1, long src2);	src1、src2、および キャリー・ビットを加算します。この関数は単独のアセンブリ命令にはマップされず、インライン関数にマップされます。
long L_sub_c(long src1, long src2);	src2 と符号ビットの論理否定を src1 から減算します。この関数は単独のアセンブリ命令にはマップされず、インライン関数にマップされます。
long L_sat(long src1);	オーバーフローが設定されると、L_add_c または L_sub_c より後のすべての結果を飽和します。
int crshft_r(int x, int y);	x を y の分だけ右シフトし、飽和で結果を丸めます。
long L_crshft_r(long x, int y);	x を y の分だけ右シフトし、飽和で結果を丸めます。
int divs(int x, int y);	飽和を使い、x を y で除算します。

図 6–2.　組み込み関数ヘッダ・ファイル gsm.h

```
#ifndef _GSMHDR
#define _GSMHDR
#include <linkage.h>
#define MAX_16 0x7fff
#define MIN_16 -32768
#define MAX_32 0x7fffffff
#define MIN_32 0x80000000
extern int Overflow;
extern int Carry;

#define L_add(a,b) (_lsadd((a),(b)))
#define L_sub(a,b) (_lssub((a),(b)))
#define L_negate(a) (_lsneg(a))
#define L_deposit_h(a) ((long)a<<16)
#define L_deposit_l(a) ((long)a)
#define L_abs(a) (_labss((a)))
#define L_mult(a,b) (_lsmpy((a),(b)))
#define L_mac(a,b,c) (_smac((a),(b),(c)))
#define L_macNs(a,b,c) (L_add_c((a),L_mult((b),(c))))
#define L_msu(a,b,c) (_smas((a),(b),(c)))
#define L_msuNs(a,b,c) (L_sub_c((a),L_mult((b),(c))))
#define L_shl(a,b) _lsshl((a),(b))
#define L_shr(a,b) _lshrs((a),(b))
#define L_shr_r(a,b) (L_crshft_r((a),(b)))
#define L_shift_r(a,b) (L_shr_r((a),-(b)))

#define abs_s(a) (_abss((a)))
#define add(a,b) (_sadd((a),(b)))
#define sub(a,b) (_ssub((a),(b)))
#define extract_h(a) ((unsigned)((a)>>16))
#define extract_l(a) ((int)a)
#define round(a) (short)(_rnd(a)>>16)
#define mac_r(a,b,c) (short)(_smacr((a),(b),(c))>>16)
#define msu_r(a,b,c) (short)(_smasr((a),(b),(c))>>16)
#define mult_r(a,b) (short)(_smpyr((a),(b))>>16)
#define mult(a,b) (_smpy((a),(b)))
#define norm_l(a) (_lnorm(a))
#define norm_s(a) (_norm(a))
#define negate(a) (_sneg(a))
#define shl(a,b) _sshl((a),(b))
#define shr(a,b) _shrs((a),(b))
#define shr_r(a,b) (crshft_r((a),(b)))
#define shift_r(a,b) (shr_r(a,-(b)))
#define div_s(a,b) (divs(a,b))
```

図 6–2.　組み込み関数ヘッダ・ファイル gsm.h（続き）

```
#ifdef __cplusplus
extern "C"
{
#endif /* __cplusplus */

int             crshft_r(int x, int y);
long            L_crshft_r(long x, int y);
int             divs(int x, int y);
_IDECL long     L_add_c(long, long);
_IDECL long     L_sub_c(long, long);
_IDECL long     L_sat(long);

#ifdef _INLINE
static inline long L_add_c (long L_var1, long L_var2)
{
    unsigned   long     uv1 = L_var1;
    unsigned   long     uv2 = L_var2;
    int        cin= Carry;
    unsigned   long     result =    uv1 + uv2 + cin;

    Carry = ((~result & (uv1 | uv2)) | (uv1 & uv2)) >> 31;
    Overflow = ((~(uv1 ^ uv2)) & (uv1 ^ result)) >> 31;

    if (cin && result == 0x80000000) Overflow = 1;
    return (long)result;
}

static inline long L_sub_c (long L_var1, long L_var2)
{
    unsigned   long     uv1=    L_var1;
    unsigned   long     uv2=    L_var2;
    int        cin= Carry;
    unsigned   long     result  =   uv1 + ~uv2 + cin;

    Carry=     ((~result & (uv1 | ~uv2)) | (uv1 & ~uv2)) >> 31;
    Overflow   =        ((uv1 ^ uv2) & (uv1 ^ result)) >> 31;

    if (!cin && result == 0x7fffffff) Overflow = 1;
    return (long)result;
}

static inline long L_sat (long L_var1)
{
    int cin    =        Carry;
    return !Overflow ?L_var1 :(Carry = Overflow = 0, 0x7fffffff+cin);
}
#endif /* !_INLINE */

#ifdef __cplusplus
} /* extern "C" */
#endif /* __cplusplus */
#endif /* !_GSMHDR */
```

6.6 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（システムがハードウェア・リセットによって初期化される場合、割り込みは無効です）。システムで割り込みが使用される場合は、必要となる割り込みの有効化やマスキングの処理はユーザが行わなければなりません。このような操作は、C/C++ 環境に影響を与えず、asm 文で組み込むことで簡単に実現できます。

6.6.1 割り込みについての一般的なポイント

割り込みルーチンは、グローバル変数へのアクセス、ローカル変数の割り当て、他の関数の呼び出しなど、他の関数によって実行されるタスクを実行できます。

割り込みルーチンを書く場合、以下の点に注意してください。

- ❏ 割り込みの特別なマスキングの処理はユーザが行わなければなりません。
- ❏ 割り込み処理ルーチンは引数を持てません。
- ❏ 割り込み処理ルーチンは、通常の C/C++ コードを使って呼び出すことはできません。
- ❏ 割り込み処理ルーチンは、単独の割り込みまたは複数の割り込みを処理できます。コンパイラは、特定の割り込みに特有のコードを生成しません。ただし、システム・リセット割り込みの c_int00 は除きます。c_int00 に入った場合、ランタイム・スタックが設定されていると想定することができないので、ローカル変数を割り当てることができず、ランタイム・スタック上に情報を保存できません。
- ❏ 割り込みルーチンを割り込みに関連付けるには、適切な割り込みベクトルに分岐を配置する必要があります。アセンブラとリンカを使い、.sect アセンブラ疑似命令を使い分岐命令の表を作成することにより、この操作を実行します。
- ❏ アセンブリ言語の中で、シンボル名の前に下線を付けることを忘れないこと。たとえば、c_int00 は _c_int00 としてください。
- ❏ 割り込みルーチンの最初で、偶数の（long が割り当てられた）アドレスにスタックを割り当てることはできません。コードを書いて、スタック・ポインタが正しく割り当てられるようにしなければなりません。

6.6.2 割り込みエントリ上でのコンテキストの保存方法

ステータス・レジスタなど、割り込みルーチンが使用するレジスタはすべて保存する必要があります。割り込みルーチンが他の関数を呼び出す場合、表 6–2（6-13 ページ）にあるレジスタはすべて保存しなければなりません。

6.6.3 C/C++ 割り込みルーチンの使用方法

割り込みは、interrupt キーワードを使うことにより、C/C++ 関数で直接処理できます。たとえば次のとおりです。

```
interrupt void isr()
{
  ...
}
```

interrupt キーワードを付けると、割り込みルーチンを定義できます。コンパイラがこれらのルーチンの 1 つを検出すると、その関数が割り込みトラップから起動され得るようなコードを生成します。必要なスタック割り当てを実行し、コンテキストの保存が行われます。この方法により、標準の C/C++ シグナル・メカニズムより多くの機能が使えます。これらは C55x ライブラリでは実行されません。これにより、シグナル関数の実装を妨げずに、これらの関数全体を C/C++ で書くことができるようになります。

6.6.4 割り込みのための組み込み関数

2 つの組み込み関数が C からの割り込みの有効化および無効化をサポートします。これらはファイル c55x.h で定義され、次のようになります。

```
void _enable_interrupts(void);
void _disable_interrupts(void);
```

これらの関数は、ST1 ステータス・レジスタの INTM ビットを有効化のためにクリアするか、無効化のために設定する命令にコンパイルされます。

一部のバージョンのハードウェアでは、INTM ビットの設定と割り込みが実際に無効となるまでに待ち時間があります。コンパイラは、ソース・コードに _disable_interrupts() が現れるポイントの適切なサイクル数分前に、INTM を設定することにより、この待ち時間を制御します（割り込みが無効化される実際のポイントは、コンパイラのアセンブリ出力の割り込みを無効化するコメントにより示されます）。

これらの命令を使うと、C コードで割り込みを有効化および無効化できるだけでなく、必要なタイミングをすべて自動的に処理できます。

6.7 P2 リザーブド・モードにおけるデータの拡張アドレッシング

ランタイム・ライブラリには、TMS320C55x の全アドレス空間へのデータの読み書きをサポートする関数が含まれています。これらの関数は、図 6–3 に示すとおり extaddr.h を介してアクセスできます。拡張メモリ・アドレスは、integer 型 FARPTR （unsigned long）の値で表されます。拡張アドレッシングコードの例は、例 6-8（6-43 ページ）に示しています。

P2 Restricted Mode のハードウェアを使う場合、C でアクセス可能なデータ・オブジェクトはすべて 64K ワード・データ・ページ・ゼロになくてはなりません。

リモート・データ・ページおよびページ・ゼロ間のデータのコピーをサポートするために、ランタイム関数が追加されてきました。これらの関数に渡される全 23 ビット・アドレスは、符号なし倍長整数値として表されます。

.cinit セクションおよび .switch セクションは、拡張メモリに配置できます。しかし、.const セクションは拡張メモリに配置できません。詳細は、6.8 節「拡張メモリ内における .const セクション」（6-43 ページ）を参照してください。

図 6–3. ファイル extaddr.h

```
/*************************************************************************/
/* extaddr.h                                                             */
/*************************************************************************/
/*************************************************************************/
/* Type of extended memory data address values                           */
/*************************************************************************/
  typedef unsigned long FARPTR;
/*************************************************************************/
/* Prototypes for Extended Memory Data Support Functions                 */
/*                                                                       */
/* far_peek       Read an int from extended memory address               */
/* far_peek_l     Read an unsigned long from extended memory address     */
/* far_poke       Write an int to extended memory address                */
/* far_poke_l     Write an unsigned long to extended memory address      */
/* far_memcpy      Block copy between extended memory addresses          */
/* far_near_memcpy  Block copy from extended memory address to page 0    */
/* near_far_memcpy  Block copy from page 0 to extended memory address    */
/*************************************************************************/
int far_peek(FARPTR);
unsigned long far_peek_l(FARPTR);
void far_poke(FARPTR, int);
void far_poke_l(FARPTR, unsigned long);
void far_memcpy(FARPTR, FARPTR, int);
void far_near_memcpy(void *, FARPTR, int);
void near_far_memcpy(FARPTR, void *, int);
```

6.7.1 拡張メモリ内へのデータの配置

グローバルおよび静的 C 変数は、拡張メモリに配置できます。これらを特定の名前の付いたセクションに配置するには、DATA_SECTION プラグマを使います。そして、リンカ制御ファイルを編集し、名前付きのセクションを拡張メモリに配置します。

6.7 節で説明している関数を使うことによってのみ、このような変数にアクセスできます。通常の C 式でこのような変数を使うと、予期せぬ結果が生じます。

拡張メモリに配置されたグローバルおよび静的変数は、通常通り C ソース・コードで初期化できます。これらの変数の自動初期化は、他のグローバルおよび静的変数と同じ方法でサポートされます。

6.7.2 拡張メモリにおけるデータ・オブジェクトの全アドレス取得方法

FAR プラグマを使うと、ファイル・スコープで宣言されたデータ・オブジェクトの全 23 ビット・アドレスを取得できます。プラグマは、下記に示すように、適用するシンボルの定義の直前に来る必要があります。

```
#pragma FAR(x)
int x;          /* ok */

#pragma FAR(z)
int y;         /* error - z's definition must  */
int z;         /*   immediately follow #pragma */
```

このプラグマはファイル・スコープで宣言されたオブジェクトにのみ適用できます。一度適用されると、指定のオブジェクトのアドレスを取得することにより、オブジェクトの全アドレスである 23 ビット値が作成されます。この値は FARPTR（つまり unsigned long）にキャストし、ランタイムサポート関数に渡すことができます。構造化オブジェクトでは、フィールドのアドレス取得によっても全アドレスを作成できます。

6.7.3 far データ・オブジェクトへのアクセス方法

far データ・オブジェクトのアクセスに対するサポートは、一般的ではありません。FAR プラグマでオブジェクトを宣言すると、コンパイラはそのアドレスを取得しなければどの構成要素に対しても far として処理しません（例 6-7 に示すとおりです）。たとえば、far ポインタ型がないと、far オブジェクトに直接アクセスできません（i = ary[10] のように）。コンパイラはこれらの違反を診断しません。

例 6-7. Far オブジェクトへのアクセスのためのイディオム

```
#include  <extaddr.h>

#pragma   DATA_SECTION(var, ".far_data")
#pragma   FAR(var)
int       var;

#pragma   DATA_SECTION(ary, ".far_data")
#pragma   FAR(ary)
int       ary[100];
          ...

void         func(void)
{
          /* Save pointer to ary */
          FARPTR aptr = (FARPTR) &ary;
          ...

          /* Load page zero from extended memory */
          i = far_peek((FARPTR) &var);
          far_near_copy(&data, aptr, 100);
          ...
```

6.8 拡張メモリ内における .const セクション

.cinit セクションおよび .switch セクションと異なり、.const セクションは、拡張メモリに配置できません。しかし、-mc シェル・オプションにより、通常 .const セクションに配置される定数は、読み取り専用の初期化された静的変数として処理できます。これにより、ランタイムの値を保持するために使われた空間がまだページ 0 にある場合、定数値を拡張メモリにロードできるようになります。

定数を保持するために使われるスペースは、.bss セクションに割り当てられます。定数を初期化するために使われる自動初期化レコードは、.cinit セクションに割り当てられます。自動初期化により、定数値は .bss セクションのある場所に配置されます。これはグローバルおよび静的変数の初期化の場合と同じです。

DATA_SECTION プラグマを介して名前付きセクションに明示的に配置された定数は、-mc オプションの影響を受けません。初期化のための .cinit レコードでなく名前付きセクションに配置されます。

例 6-8. データの拡張アドレッシング

```
#include  <extaddr.h>

/************************************************/
/* Variables to be placed in extended memory.     */
/************************************************/
#pragma   DATA_SECTION(ival,  ".far_data")
#pragma   FAR(ival)
int       ival;

#pragma   DATA_SECTION(lval,  ".far_data")
#pragma   FAR(lval)
unsigned  long lval;

#pragma   DATA_SECTION(a,     ".far_data")
#pragma   FAR(a)
int       a[10] =             {0,1,2,3,4,5,6,7,8,9};

#pragma   DATA_SECTION(b,     ".far_data")
#pragma   FAR(b)
int       b[10];

#pragma   DATA_SECTION(c,     ".far_data")
#pragma   FAR(c)
char      c[12] =             {"test  string"};
```

例 6–8. データの拡張アドレッシング（続き）

```
/************************************************/
/* Variable to be placed in memory page 0       */
/************************************************/
char   d[12];


void   main(void)
{
int    ilocal;

/* Get extended addresses of variables */
FARPTR iptr   =   (FARPTR)  &ival;
FARPTR lptr   =   (FARPTR)  &lval;

/************************************************/
/* Read and write variables in extended memory  */
/************************************************/

/* ival = 100 */
far_poke(iptr, 100);

/* ival += 10 */
ilocal = far_peek(iptr) + 10;
far_poke(iptr, ilocal);

/* lval = 0x7ffffffe */
far_poke_l(lptr, 0x7ffffffe);

/* lval += 1 */
far_poke_l(lptr, far_peek_l(lptr) + 1);

/************************************************/
/* Copy string from extended data memory to an  */
/* address in Page 0.                           */
/************************************************/
far_near_memcpy((void *) d, (FARPTR) &c, 12);

/************************************************/
/* Copy an array of integers from one extended  */
/* address to another extended address.         */
/************************************************/
far_memcpy((FARPTR) &b, (FARPTR) &a, 10);
}
```

6.9 システムの初期化

C/C++ プログラムを実行する前に、C/C++ ランタイム環境を作成する必要があります。この作業は、C/C++ ブート・ルーチンによって行われ、c_int00 と呼ばれる関数を使用します。このルーチンのソースは、ランタイムサポート・ソース・ライブラリ（rts.src）の boot.asm と呼ばれるモジュール内に入っています。

システムの実行を開始するために、リセット・ハードウェアによって _c_int00 関数を呼び出すことができます。_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 リンカ関数オプションを使用し、rts.src をリンカ入力ファイルの 1 つとして組み込みます。

C/C++ プログラムをリンクした場合、リンカは実行可能な出力モジュールのエントリ・ポイント値をシンボル _c_int00 に設定します。_c_int00 関数は、次の作業を実行して C/C++ 環境を初期化します。

1） スタックおよび 2 次システム・スタックを設定します。

2） グローバル変数を初期化するため、.cinit セクションおよび .pinit 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、6.9.1 項「変数の自動初期化」（6-45 ページ）を参照してください。

3） 関数 main を呼び出して、C/C++ プログラムの実行を開始します。

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。

ブート・ルーチンの詳細は、4.3.2 項「実行時の初期化」（4-9 ページ）を参照してください。

6.9.1 変数の自動初期化

事前に初期化されるように宣言された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、各ファイルの .cinit 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ていないグローバル変数および静的変数は、プログラム実行前に 0 に設定する必要があります。C/C++ コンパイラは、初期化されていない変数の事前初期化は実行しません。初期値が 0 でなければならない変数は、すべて明示的に初期化する必要があります。

最も簡単な方法は、.bss セクションのリンカ制御マップで、埋め込み値を 0 に設定することです（ROM に組み込まれたコードではこの方法は使えません）。

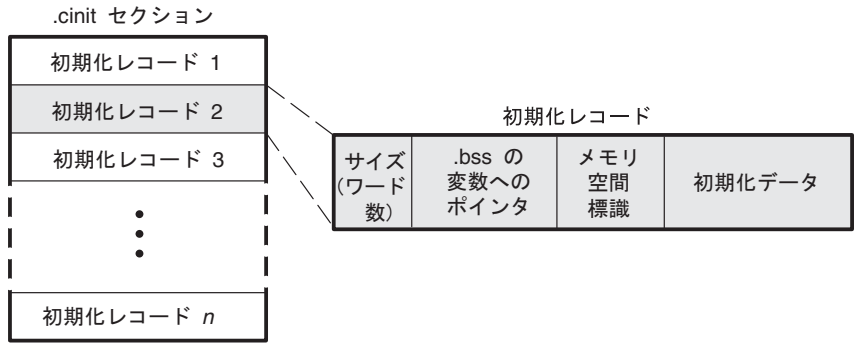
6.9.2 グローバル・オブジェクト・コンストラクタ

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。

6.9.3 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 6–4 は、.cinit セクションの形式と初期化レコードの形式を示しています。

図 6–4. .cinit セクション内の初期化レコードの形式



初期化レコードには、次の情報が含まれています。

- 最初のフィールド（ワード 0）には、初期化データのサイズ（ワード数）があります。

  ビット 14 または 15 は保存されます。初期化レコードの長さは 2^{13} ワードまで有効です。

- 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。このフィールドは 24 ビットで、16 ビットより大きいアドレスに適応します。
- 3 番目のフィールドは 8 ビットのフラグが含まれています。ビット 0 はメモリ空間標識のためのフラグです（I/O またはデータ）。他のビットは予約されています。
- 4 番目のフィールド（3 から *n* までのワード）には、変数を初期化するためにコピーされたデータが含まれています。

.cinit セクションには、初期化される各変数の初期化レコードが含まれています。例 6-9（a）は、C/C++ で定義された初期化される変数を示しています。例 6-9（b）は、対応する初期化テーブルを示しています。

例 6-9. 初期化変数および初期化テーブル

（a）C で定義された初期化される変数

```
int    i = 3;
long   x = 4;
float  f = 1.0;
char   s[] = “abcd”;
long   a[5] = { 1, 2, 3, 4, 5 };
```

例 6–9. 初期化変数および初期化テーブル（続き）

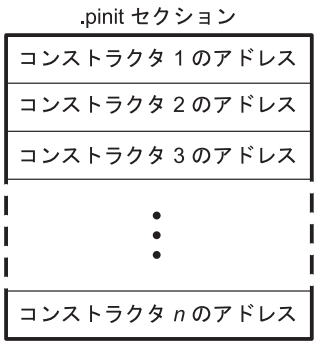
（b）（a）で定義された変数の初期化される情報

```
       .sect      ".cinit"      ; Initialization section
* Initialization record for variable i
       .field     1,16          ; length of data (1 word)
       .field     _i+0,24       ; address in .bss
       .field     0,8           ; signifies data memory
       .field     3,16          ; int is 16 bits
* Initialization record for variable x
       .field     2,16          ; length of data (2 words)
       .field     _l+0,24       ; address in .bss
       .field     0,8           ; data memory
       .field     4,32          ; long is 32 bits
* Initialization record for variable f
       .field     2,16          ; length of data (2 words)
       .field     _f+0,24       ; address in .bss
       .field     0,8           ; data memory
       .xlong     0x3f800000    ; float is 32 bits
* Initialization record for variable s
       .field     IR_1,16       ; length of data
       .field     _s+0,24       ; address in .bss
       .field     0,8           ; data memory
       .field     97,16         ; a
       .field     98,16         ; b
       .field     99,16         ; c
       .field     100,16        ; d
       .field     0,16          ; end of string
IR_1   .set       5             ; symbolic value gives
                                 ; count of elements
* Initialization record for variable a
       .field     IR_2,16       ; length of data
       .field     _a+0,24       ; address in .bss
       .field     0,8           ; data memory
       .field     1,32          ; beginning of array
       .field     2,32
       .field     3,32
       .field     4,32
       .field     5,32          ; end of array
IR_2   .set       10            ; size of array
```

.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 C/C++ プログラムにインターフェイスする場合は、.cinit セクションを他の目的に使用しないでください。

-c や -cr 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。

図 6–5.　.pinit セクション内の初期化レコードの形式



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 修飾子で修飾された変数の初期化方法は異なります。5.9.2 項「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」（5-34 ページ）を参照してください。

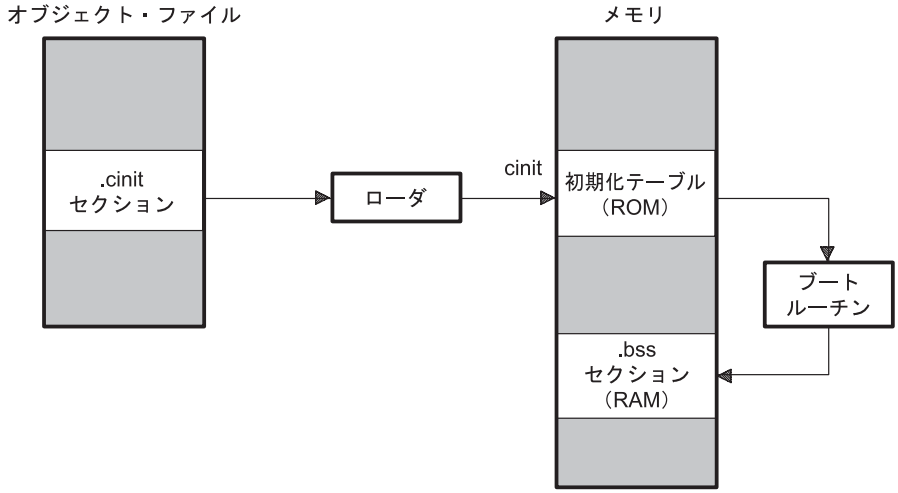
6.9.4　実行時の変数の自動初期化

実行時の変数の自動初期化は、自動初期化のデフォルト・モデルです。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（可能ならば ROM）にロードされ、グローバル変数は実行時に初期化されます。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 6–6 は実行時の自動初期化を説明しています。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 6–6. 実行時の自動初期化



6.9.5 ロード時の変数の初期化

ロード時の変数の自動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。

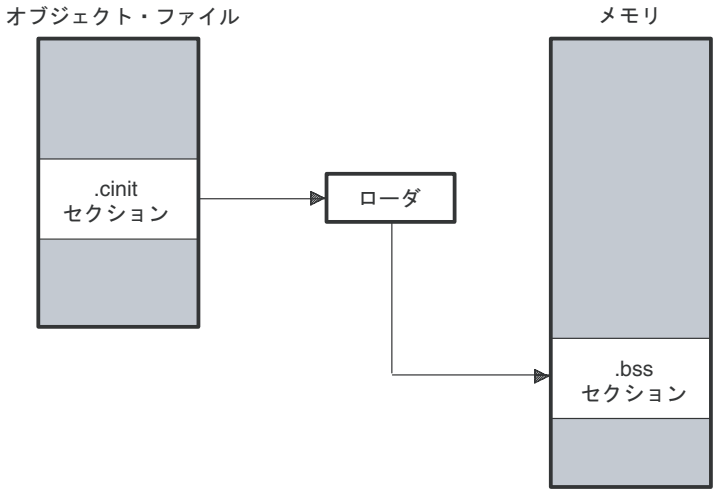
-cr リンカ・オプションを使用すると、リンカは .cinit セクションのヘッダ内に STYP_COPY ビットを設定します。これにより、.cinit セクションを メモリ内にロードしないことをローダに指示します（.cinit セクションは、メモリ・マップのスペースを占有しません）。また、リンカは cinit シンボルを -1 に設定します（通常では cinit は初期化テーブルの先頭を指します）。これにより、初期化テーブルがメモリ内に存在しないことをブート・ルーチンに示します。したがって、ブート時に実行時の初期化は行われません。

ロード時の自動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・ヘッダ内に STYP_COPY が設定されているかどうかを判別して、.cinit セクションをメモリ内にコピーしないことを認識する。
- ❏ 初期化テーブルのフォーマットを理解する。

図 6–7 は、自動初期化の RAM モデルを説明しています。

図 6–7.　ロード時の初期化



第 7 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、文字列の検索など）の中には、C/C++ 言語自体の一部ではないものがあります。ランタイムサポート関数は、C/C++ コンパイラに添付されていますが、これらの作業を実行する標準 ISO 関数です。

ランタイムサポート・ライブラリ rts.src に、他の関数およびルーチンのためのソースと同様、これらの関数のためのソースが入っています。基本的なオペレーティング・システム（信号など）を必要とするものを除き、すべての ISO 関数が提供されます。

カスタマイズしたランタイムサポート・ライブラリを作成できるコード生成ツールによって、ライブラリ作成ユーティリティが組み込まれます。ライブラリ作成ユーティリティの詳細は、第 8 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**

7.1 ライブラリ 7-2
7.2 C 入出力関数 7-4
7.3 ヘッダ・ファイル 7-16
7.4 ランタイムサポート関数およびマクロのまとめ 7-29
7.5 ランタイムサポート関数およびマクロについて 7-39

7.1 ライブラリ

以下のライブラリは、 TMS320C55x C/C++ コンパイラに添付されています。

- *rts55.lib* には、ISO のランタイムサポート・オブジェクト・ライブラリが含まれています。
- *rts55.lib* には、ラージ・メモリ・モデルのための ISO のランタイムサポート・オブジェクト・ライブラリが含まれています。
- *rts.src* には、ISO のランタイムサポート・ルーチンのためのソースが含まれています。

オブジェクト・ライブラリには、本章で説明している標準 C/C++ のランタイムサポート関数、浮動小数点ルーチン、およびシステム起動ルーチン _c_int00 が含まれています。オブジェクト・ライブラリは、rts.src ライブラリに入っている C/C++ およびアセンブリ・ソースから作成されます。

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もの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。リンカの詳細は、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルのリンカについての章を参照してください。

7.1.1 ライブラリ関数の修正方法

アーカイバを使用して rts.src 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。

```
ar55 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。

```
cl55 -options atoi.c strcpy.c      ;recompile
ar55 -r rts.src atoi.c strcpy.c    ;rebuild library
```

rts55.lib に再度作成するのでなく、このような方法で新しいライブラリを作成することもできます。アーカイバの詳細は、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルのアーカイバに関する説明の章を参照してください。

7.1.2 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 mk55 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。

```
mk55 --u -o2 rts.src -l rts55.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、mk55 ユーティリティに指示します。オプティマイザ（-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。ライブラリ作成の詳細は、第 8 章「ライブラリ作成ユーティリティ」を参照してください。

7.2 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（デバッガを使用）。たとえば、C55x プログラムで実行される printf 文は、デバッガ・コマンド・ウィンドウに表示されます。デバッグ・ツールとともに使用した場合には、ホスト上で入出力を実行できるので、コードのデバッグとテストに利用できるオプションの数が多くなります。

入出力関数を使用するには、関数を参照するモジュールごとにヘッダ・ファイル stdio.h を組み込んでください。

入出力関数を使用するには、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。

TI のデフォルトのリンカ・コマンド・ファイルを使わない場合は、ユーザのリンカ・コマンド・ファイルの .cio セクションを割り当てる必要があります。.cio セクションは、ランタイムおよびデバッガの使用するバッファを（label _ _CIOBUF_ で）提供します。C 入出力の型が実行されると（printf、scanf など）、バッファが作成され、_ _CIOBUF_ アドレスに配置されます。バッファには次の内容が含まれます。

- ❏ 実行されるストリーム入出力の型のための内部 C 入出力コマンド（および必要なパラメータ）。
- ❏ I/O コマンドから戻されるデータ。

バッファはデバッガによって読み込まれ、デバッガは適切な入出力コマンドを実行します。

たとえば、main.c ファイル内に次の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");
}
```

次のシェル・コマンドを発行すると、ファイル main.out のコンパイル、リンク、および作成が行われます。

```
cl55 main.c -z -heap 400 -l rts.lib -o main.out
```

SPARC ホストの C55x デバッガで main.out を実行すると、次の動作が行われます。

1） デバッガが起動されたディレクトリにファイル *myfile* を開く。

2） そのファイルに文字列 *Hello, world* を出力する。

3） ファイルをクローズする。

4） そのデバッガ・コマンド・ウィンドウに文字列 *Hello again, world* を出力する。

正しく作成されたデバイス・ドライバにより、C 入出力関数を使用して、ユーザ指定のデバイス上で入出力を実行できます。

C 入出力バッファのために充分なヒープ・サイズがない場合、ファイル上でバッファされた操作は実行されません。printf() への呼び出しが原因不明に失敗する場合は、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。

7.2.1 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 3 つの層に分類されます。高レベル、低レベル、およびデバイス・レベルです。

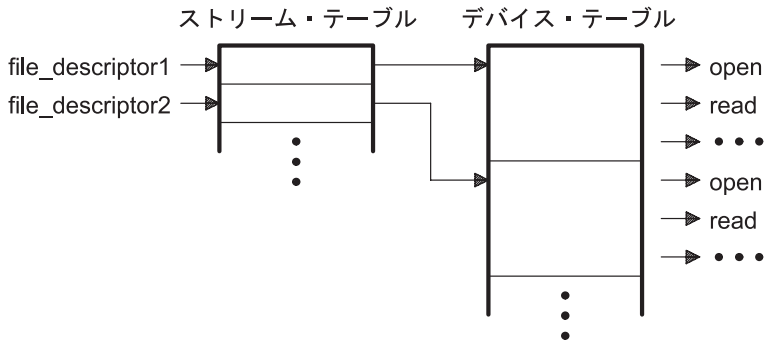
高レベル関数は、ストリーム入出力ルーチン（printf、scanf、fopen、getchar など）の標準 C ライブラリです。これらのルーチンは、低レベル・シェルによって処理される 1 つまたは複数の入出力コマンドに、入出力要求をマップします。

低レベル関数は、基本入出力関数（open、read、write、close、lseek、rename、unlink）から構成されています。指定したデバイス上で入出力コマンドを実際に実行する高レベル関数とデバイス・レベル・ドライバの間のインターフェイスが、これら低レベル関数によって提供されます。

また、ファイル記述子をデバイスに関連付けるストリーム・テーブルの定義と保守も、低レベル関数によって行われます。ストリーム・テーブルはデバイス・テーブルと相互作用しているので、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。

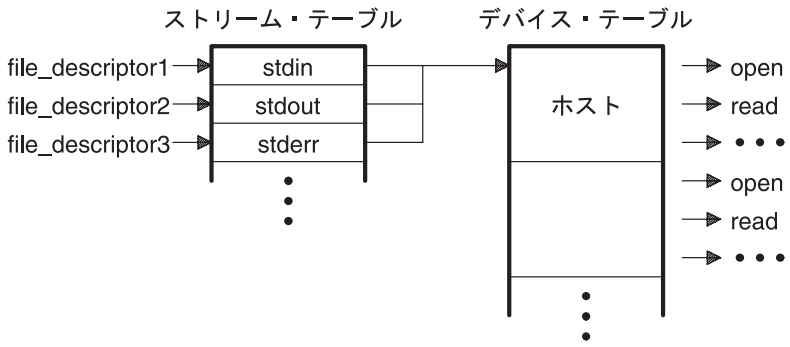
データ構造は、図 7–1 に示すように相互作用します。

図 7–1. 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 7–2. 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。C 入出力ライブラリには、デバッガが稼働中のホストで C 入出力を実行するために必要なデバイス・ドライバが入っています。

低レベルのルーチンとインターフェイス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、7-12 から 7-15 ページに示します。各関数が、必要に応じてそれぞれに固有のデータ構造を設定し保持するように作成する必要があります。何の処置も実行せず、ただ戻るように関数を定義する場合もあります。

7.2.2 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) 7.2.1 項（7-6 ページ）に示すとおり、デバイス・レベルの関数を定義します。

注：一意的な関数名の使用方法
関数名 open()、close()、read() などは、低レベルのルーチンで使用されています。作成するデバイス・レベルの関数には、他の名前を使ってください。

2) 低レベル関数 add_device() を使用して、device_table に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 file.h の中にあるマクロ _NDEVICE で定義されます。デバイスを表す構造体も lowlev.c の中に定義され、次のフィールドから構成されます。

name デバイス名を表す文字列

flags デバイスが複数のストリームをサポートするかどうかを指定するフラグ

function pointers 次のデバイス・レベル関数を指すポインタ

- ❏ close
- ❏ lseek
- ❏ open
- ❏ read
- ❏ rename
- ❏ write
- ❏ unlink

デバイス・テーブルの最初のエントリは、デバッガが稼働中のホスト・デバイスになるように事前定義されます。低レベル・ルーチン add_device() は、デバイス・テーブルの最初の空き位置を検出し、渡された引数でデバイス・フィールドを初期化します。add_device 関数の詳細は、7-10 ページを参照してください。

3） デバイスを追加した後に fopen() を呼び出してストリームをオープンし、このストリームをそのデバイスに関連付けます。*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>
#include <file.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 off_t  my_lseek(int fno, off_t 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);
}
```

add_device　デバイス・テーブルへのデバイスの追加

構文

```
#include < file.h >
int add_device(char *name,
        unsigned flags,
        int (*dopen)(),
        int (*dclose)(),
        int (*dread)(),
        int (*dwrite)(),
        off_t (*dlseek)(),
        int (*dunlink)(),
        int (*drename)());
```

定義される場所　lowlev.c in rts.src

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、書式 *devicename*:*filename* の文字列を第 1 引数にして、fopen() を使用してください。

- *name* は、デバイス名を示す文字列です。
- *flags* は、デバイス特性です。フラグは以下のとおりです。

  _SSA　一度に 1 つのストリームしかオープンできないことを示します。

  _MSA　複数のストリームをオープンできることを示します。

  フラグを stdio.h 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 7.2.1 項「低レベル入出力実装の概要」（7-6 ページ）に指定されているインターフェイスを使用してください。デバッガが実行されるホストのデバイス・ドライバは、C 入出力ライブラリに組み込まれています。

戻り値　この関数は、以下の値のどれかを戻します。

0	成功した場合
-1	失敗した場合

例　　この例では、以下の操作が実行されます。

- ❏ 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(const char *path, unsigned flags, int fno);
extern int my_close(int fno);
extern int my_read(int fno, char *buffer, unsigned count);
extern int my_write(int fno, const char *buffer, unsigned count);
extern int my_lseek(int fno, long offset, int origin);
extern int my_unlink(const char *path);
extern int my_rename(const char *old_name, const char *new_name);

main()
{

   FILE *fid;
   add_device("mydevice", _MSA, my_open, my_close, my_read, my_write, my_lseek,
                                my_unlink, my_rename);

   fid = fopen("mydevice:test","w");

   fprintf(fid,"Hello, world\n");

   fclose(fid);
}
```

close　入出力用ファイルまたはデバイスのクローズ

構文

#include < file.h >
int close(int *file_descriptor***);**

説明

close 関数は、*file_descriptor* に関連した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

構文

#include < file.h >
off_t lseek(int *file_descriptor*, **off_t** *offset*, **int** *origin***);**

説明

lseek 関数は、指定されたファイルの位置標識を
origin + *offset* に設定します。ファイル位置標識では、ファイルの先頭から文字単位で位置が測定されます。

- *file_descriptor* は、オープンされたファイルまたはデバイスにデバイス・レベル・ドライバが関連付けなければならない低レベル・ルーチンによって割り当てられるストリーム番号です。
- *offset* は、*origin* からの文字単位の相対位置を指示します。
- *origin* は、どの基本位置から *offset* を測定するかを指示するために使用します。*origin* は、次のいずれかのマクロによって戻される値でなければなりません。

  SEEK_SET　(0x0000) ファイルの先頭

  SEEK_CUR　(0x0001) ファイル位置標識の現行値

  SEEK_END　(0x0002) ファイルの最後

戻り値

戻り関数は、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

構文

#include < file.h >
int 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の扱われ方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  (0x0100)  /* open with file create */
O_TRUNC  (0x0200)  /* 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。

  次に使用可能な file_descriptor（3 から 20 の順）は、オープンされた新しい各デバイスに割り当てられます。finddevice() 関数を使ってデバイス構造体を戻すことができ、このポインタを使って同じポインタの _stream 配列を検索することができます。file_descriptor の番号は、_stream 配列の他のメンバです。

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

構文

#include < file.h >
int read(int *file_descriptor*, **char ***buffer*, **unsigned** *count*);**

説明

read 関数は、*count* で指定した文字の数を、*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、読みとられた文字が入るバッファのロケーションです。
- ❏ *count* は、デバイスまたはファイルから読みとる文字数です。

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　ファイルの再命名

構文

#include < file.h >
int 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	rename が成功した場合
0 以外	失敗した場合

unlink　ファイルの削除

構文

#include < file.h >
int 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 は削除するファイルのファイル名で、パス情報を含んでいます。

戻り値

この関数は、以下の値のどれかを戻します。

0　成功した場合

-1　失敗した場合

write　バッファへの文字の書き込み

構文

#include < file.h >
int write(int *file_descriptor*, **const char *** *buffer*, **unsigned** *count*);

説明

write 関数は、*count* で指定した文字の数を *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、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、書き込まれた文字が入るバッファのロケーションです。
- *count* は、デバイスまたはファイルに書き込む文字数です。

戻り値

この関数は、以下の値のどれかを戻します。

#　成功した場合に書き込まれた文字数

-1　失敗した場合

7.3 ヘッダ・ファイル

各ランタイムサポート関数は、*header file* 内に宣言されています。各ヘッダ・ファイルで宣言する内容は、次のとおりです。

- ❏ 一連の関連する関数（またはマクロ）
- ❏ 関数を使用するために必要な型
- ❏ 関数を使用するために必要なマクロ

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、提供され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。

c55x.h	cpy_tbl.h	file.h	gsm.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、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。

7.3.1 項「診断メッセージ （assert.h/cassert）」（7-17 ページ）から 7.3.20 項「ランタイム型情報（typeinfo）」（7-28 ページ）に、C/C++ コンパイラとともに含まれるヘッダ・ファイルについての説明があります。7.5 節「ランタイムサポート関数およびマクロについて」（7-39 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。

7.3.1 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 と呼ばれる、異なる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 は有効です。

assert.h ヘッダは NASSERT と呼ばれる、異なるマクロを参照します（assert.h は NASSERT の定義は行いません）。assert.h 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オプティマイザに伝えます。この関数は、どの最適化が有効であるかのヒントをオプティマイザに提供します。NASSERT が定義されない場合、assert は通常有効です。

assert 関数を 表 7–3（a）（7-30 ページ）に示します。

7.3.2 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。文字変換関数は、文字を小文字、大文字、あるいは ASCII に変換し、変換後の文字を戻します。文字判別関数の名前の形式は、**is***xxx*（*isdigit* など）です。文字変換関数の名前の形式は、**to***xxx*（*toupper* など）です。

ctype.h/cctype ヘッダには、これらと同じ処理を実行するマクロ定義も含まれています。マクロのほうが、同じ機能をもつ関数より処理速度が高速です。これらのマクロの 1 つに副次作用がある引数が渡される場合は、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。

文字判別関数および文字変換関数を 表 7–3（b）（7-30 ページ）に示します。

7.3.3 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 変数は算術関数のエラーを宣言します。無効なパラメータ値が関数に渡された場合、または定義されている範囲外の結果を関数が戻した場合は、算術関数でエラーが起きる可能性があります。このような場合、errno という変数は、次のマクロのいずれかの値に設定されます。

- EDOM - 領域エラーの場合（パラメータが不正）
- ERANGE - 範囲エラーの場合（結果が不正）
- 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/errno.cpp 内で定義されています。

7.3.4 拡張アドレッシング関数（extaddr.h）

extaddr.h ヘッダは、C55x の全アドレス空間へのデータの読み書きをサポートする関数を宣言します。拡張メモリ・アドレスは、integer 型 FARPTR（unsigned long）の値で表されます。

拡張アドレッシング関数を表 7–3（c）（7-31 ページ）に示します。

7.3.5 低レベル入出力関数（file.h）

入出力操作を実行するために使用する低レベル入出力関数が file.h ヘッダで宣言されます。7.2 節「C 入出力関数」に C55x に入出力を実装する方法を説明しています。

7.3.6 制限値（float.h/cfloat と limits.h/climits）

float.h/cfloat ヘッダと limits.h/climits ヘッダは、プロセッサ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 7–1 と表 7–2 は、これらのマクロおよび関連付けられた制限値のリストを示しています。

表 7–1. 整数型の範囲に関する制限値を指定するマクロ（limits.h）

マクロ	値	説明
CHAR_BIT	16	type char のビット数
SCHAR_MIN	-32 768	signed char の最小値
SCHAR_MAX	32 767	signed char の最大値
UCHAR_MAX	65 535	unsigned char の最大値
CHAR_MIN	-32 768	char の最小値
CHAR_MAX	32 767	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	-32 768	int の最小値
INT_MAX	32 767	int の最大値
UINT_MAX	65 535	unsigned int の最大値
LONG_MIN	-2147483648	long int の最小値
LONG_MAX	2147483647	long int の最大値
ULONG_MAX	4294967295	unsigned long int の最大値
LLONG_MIN	-549 755 813 888	long long int の最小値
LLONG_MAX	549 755 813 887	long long int の最大値
ULLONG_MAX	1 099 511 627 775	unsigned long long int の最大値
MB_LEN_MAX	1	マルチバイトの最大バイト数

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 7–2.　浮動小数点の範囲に関する制限値を指定するマクロ（float.h）

マクロ	値	説明
FLT_RADIX	2	指数表現の底や基数
FLT_ROUNDS	1	浮動小数点加算の端数処理モード
FLT_DIG DBL_DIG LDBL_DIG	6 6 6	float、double、または long double に対する精度の 10 進桁数
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	24 24 24	float、double、または long double の仮数部の底 FLT_RADIX の桁数
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	-125 -125 -125	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	128 128 128	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 1.19209290e-07 1.19209290e-07	$1.0 + x \neq 1.0$ となる float、double、または long double の正の最小値 x
FLT_MIN DBL_MIN LDBL_MIN	1.17549435e-38 1.17549435e-38 1.17549435e-38	float、double、または long double の正の最小値
FLT_MAX DBL_MAX LDBL_MAX	3.40282347e+38 3.40282347e+38 3.40282347e+38	float、double、または long double の正の最大値
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	-37 -37 -37	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 38 38	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 ヘッダ・ファイルを参照してください。

7.3.7 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、複数のマシン間で整合性を保持し、オペレーティング・システムおよび他の実装の特異性から独立した状態で定義します。inttypes.h ヘッダは、最大幅の整数を操作し、数値文字列を最大幅の整数に変換する関数を宣言します。

typedef を介して、inttypes.h はさまざまなサイズの整数型を定義します。標準 C 整数型として、また inttypes.h に用意されている型として、自由に整数型を型定義できます。一貫して inttypes.h ヘッダを使用することにより、複数のプラットフォーム間でユーザ・プログラムの移植性が高くなります。

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 integer 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 に用意されている型のそれぞれのサイズ用に、複数の fprintf および fscanf マクロがあります。たとえば、符号付き整数の 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);
```

- 型 intmax_t の整数の絶対値を計算する *imaxabs* 関数。
- strtol、strtoll、strtoul、および strtoull 関数と同等の *strtoimax* 関数および *strtoumax* 関数。文字列の最初の部分は、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。

7.3.8 代替スペリング（iso646.h/ciso646）

iso646.h/ciso646 ヘッダは、対応するトークンに拡張する以下の 11 のマクロを定義します。

マクロ	トークン	マクロ	トークン
and	&&	not_eq	!=
and_eq	&=	or	||
bitand	&	or_eq	|=
bitor	|	xor	^
compl	~	xor_eq	^=
not	!		

7.3.9 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。これらの算術関数は、倍精度浮動小数点引数を要求し、倍精度浮動小数点値を戻します。

math.h/cmath ヘッダは *HUGE_VAL* というマクロも定義します。算術関数はこのマクロを使って範囲外の値を表します。関数が浮動小数点値を戻すときに、その値が大きすぎて表現できない場合は、代りに HUGE_VAL を戻します。

7.3.10 非ローカル・ジャンプ（setjmp.h/csetjmp）

setjmp.h/csetjmp ヘッダは、通常の関数の呼び出しと復帰に関する規律をバイパスするための型、マクロ、および関数を定義します。次のものが含まれます。

- 配列型 *jmp_buf* - 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。
- マクロ *setjmp* - 後で longjmp 関数で使用できるように、呼び出し環境を jmp_buf 引数に保存します。
- 関数 *longjmp* - jmp_buf 引数を使用してプログラム環境を復元します。非ローカル jmp マクロおよび関数を 表 7–3（e）（7-32 ページ）に示します。

7.3.11 可変引数（stdarg.h/cstdarg）

関数の中には、型が変化する可変数の引数を持つものがあります。このような関数を可変引数関数といいます。stdarg.h/cstdarg ヘッダでは、可変引数関数の使用に役立つ 3 つのマクロと 1 つの型を宣言しています。

3 つのマクロとは、*va_start*、*va_arg*、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*、*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 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ステップ処理することができます。この時、マクロを使用している関数は、実際に渡された引数の数と型を認識します。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。可変引数関数を表 7–3（f）（7-32 ページ）に示します。

7.3.12 標準定義 （stddef.h/cstddef）

stddef.h/cstddef ヘッダは 2 つの型と 2 つのマクロを定義します。型は以下のとおりです。

- *ptrdiff_t* 型 - 2 つのポインタの減算結果のデータ型を示す signed int 型です。
- *size_t* 型 - *sizeof* 演算子のデータ型である unsigned int 型です。

マクロは以下のとおりです。

- *NULL* マクロ - ヌル・ポインタ定数（0）に展開されます。
- *offsetof（type、identifier）*マクロ - *size_t* 型の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。

7.3.13 整数型（stdint.h）

stdint.h ヘッダは、特定の幅の整数型を宣言し、対応するマクロのセットを定義します。また、他の標準ヘッダで定義されている型に対応する整数型の限界を指定するマクロを定義します。型は、以下のカテゴリで定義されます。

- 符号付きフォーム int*N*_t および符号なしフォーム uint*N*_t の幅が正確に決まっている整数型
- 符号付きフォーム int_least*N*_t および符号なしフォーム uint_least*N*_t の幅の最低値が決まっている整数型
- 符号付きフォーム int_fast*N*_t および符号なしフォーム uint_fast*N*_t の幅の最低値が決まっている最速の整数型
- ポインタ値を保持するのに充分な大きさの符号付き *intprt_t*、および符号なし *uintptr_t* の整数型
- どんな integer 型の値を表示するのにも充分な大きさの符号付き *intmax_t*、および符号なし *uintmax_t* の整数型

stdint.h に用意されている各符号付き型は、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、特定の値と型 int_least*N*_t をもつ符号付き整数定数に展開されます。符号なし UINT*N*_C(*value*) マクロは、特定の値と型 uint_least*N*_t をもつ符号なし整数定数に展開されます。

この例は、少なくとも 16 ビットを保持できる最小の整数を使用する stdint.h で定義され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。

7.3.14 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。型および構造体は以下のとおりです。

- *size_t* 型 - *sizeof* 演算子のデータ型である unsigned int 型です。元の宣言は stddef.h/cstddef にあります。
- *fpos_t* 型 - ファイル内のすべての場所を一意的に指定できる unsigned long 型です。
- *FILE* 構造体 - ストリームの制御に必要な情報すべてを記録します。

マクロは以下のとおりです。

- *NULL* マクロ - ヌル・ポインタ定数（0）に展開されます。元の宣言は stddef.h にあります。すでに定義されている場合、再定義は行われません。
- *BUFSIZ* マクロ - 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 - ファイルの終わりを示します。
- *FOPEN_MAX* マクロ - 一度にオープンできるファイルの最大数に展開されます。
- *FILENAME_MAX* マクロ - 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 - tmpnam() が 生成する最長のファイル名文字列に展開されます。
- *SEEK_CUR*、*SEEK_SET*、および *SEEK_END* マクロ - ファイル内の位置（現在位置、ファイル開始位置、ファイル終わり位置のそれぞれ）を示すために展開されます。
- *TMP_MAX* マクロ - 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* - 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。

入出力関数を表 7–3（g）（7-33 ページ）に示します。

7.3.15 汎用ユーティリティ（stdlib.h/cstdlib）

stdlib.h/cstdlib ヘッダはいくつかの関数、1 つのマクロ、および 2 つの型を宣言します。型は以下のとおりです。

- *div_t* - div 関数が戻す値の構造体の型。
- *ldiv_t* - ldiv 関数が戻す値の構造体の型。

マクロ *RAND_MAX* は、rand 関数が戻す最大乱数です。

ヘッダは、多くの共通のライブラリ関数も宣言します。

- 文字列変換関数 - 文字列を数値表現に変換します。
- 検索およびソート関数 - 配列の検索とソートを行います。
- シーケンス生成関数 - 疑似乱数シーケンスを生成し、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 - プログラムを正常終了または異常終了できます。
- 整数算術 - C 言語の標準な部分にはありません。

汎用ユーティリティ関数を表 7–3（h）（7-35 ページ）に示します。

7.3.16 文字列関数（string.h/cstring）

string.h/cstring ヘッダでは、文字配列（文字列）に関して以下の作業を実行するための標準関数を宣言します。

- ❏ 文字列の一部あるいは全体の移動やコピー
- ❏ 文字列の連結
- ❏ 文字列の比較
- ❏ 文字や他の文字列における文字列の検索
- ❏ 文字列の長さの検出

C 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は **mem***xxx* などです。

文字列の移動やコピーを行う関数を使用するときは、デスティネーションに結果を格納するだけの十分な大きさがあることを確認しておいてください。文字列関数を表 7–3 (i)（7-37 ページ）に示します。

7.3.17 時間関数（time.h/ctime）

time.h/ctime ヘッダでは、1 つのマクロ、いくつかの型、および日付と時刻を操作する関数を宣言します。時刻は 2 とおりの方法で表されます。

- ❏ *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                     */
int     tm_wday;        /* days since Saturday (0-6)            */
int     tm_yday;        /* days since January 1 (0-365)         */
int     tm_isdst;       /* daylight savings time flag           */
```

時間は、time_t または struct_tm のいずれの型で表される場合でも、次のように異なる計測基準で表すことができます。

- ❏ カレンダ時は、グレゴリオ暦で現在の日付と時刻を表します。
- ❏ 地方時は、特定のタイム・ゾーンに関して表されたカレンダ時です。

time 関数およびマクロを表 7–3（j）（7-38 ページ）に示します。

地方時は夏時間に調整できます。当然、地方時はタイム・ゾーンによって異なります。time.h/ctime ヘッダ は、tmzone という構造体型および _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。

time.h 内のすべての関数の基本となるのは、clock と time の 2 つのシステム関数です。time は現在の時刻（time_t 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

ホスト固有の clock 関数を書き込めます。また、clock() - クロックの目盛数 - が戻す値が CLOCKS_PER_SEC で割り、秒単位で値を作成できるよう、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。

7.3.18 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。

7.3.19 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。

7.3.20 ランタイム型情報（typeinfo）

C++ 専用の typeinfo ヘッダは、実行時に C++ 型情報を表すのに使用される type_info 構造体を定義します。

7.4 ランタイムサポート関数およびマクロのまとめ

表 7–3 に、TMS320C55x ISO C/C++ コンパイラの提供するランタイムサポート・ヘッダ・ファイルをアルファベット順にまとめています。説明されている関数の大部分は、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 7–3 に掲載されている関数とマクロは、7.5 節「ランタイムサポート関数およびマクロについて」（7-39 ページ）で詳しく説明されています。関数またはマクロの詳細は、指示されているページを参照してください。

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 7–3. 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。	7-41

（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。	7-63
int **isalpha**(int c);	英字 ASCII 文字かどうか、c をテストします。	7-63
int **isascii**(int c);	ASCII 文字かどうか、c をテストします。	7-63
int **iscntrl**(int c);	制御文字かどうか、c をテストします。	7-63
int **isdigit**(int c);	数字かどうか、c をテストします。	7-63
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。	7-63
int **islower**(int c);	ASCII の英小文字かどうか、c をテストします。	7-63
int **isprint**(int c);	印刷可能な ASCII 文字かどうか、c をテストします（スペースを含む）。	7-63
int **ispunct**(int c);	ASCII の句読点文字かどうか、c をテストします。	7-63
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。	7-63
int **isupper**(int c);	ASCII の英大文字かどうか、c をテストします。	7-63
int **isxdigit**(int c);	16 進数字かどうか、c をテストします。	7-63
char **toascii**(int c);	c を有効な ASCII 値にマスクします。	7-96
char **tolower**(int char c);	c が大文字の場合は、小文字に変換します。	7-97
char **toupper**(int char c);	c が小文字の場合は、大文字に変換します。	7-97

（c）拡張アドレッシング関数（extaddr.h）

関数	説明	ページ
extern int **far_peek**(FARPTR x);	拡張メモリ・アドレスから整数を読み込みます。	7-51
extern unsigned long **far_peek_l**(FARPTR x);	拡張メモリ・アドレスから unsigned long を読み込みます。	7-51
extern void **far_poke**(FARPTR x, int x);	拡張メモリ・アドレスに整数を書き込みます。	7-52
extern void **far_poke_l**(FARPTR x, unsigned long x);	拡張メモリ・アドレスに unsigned long を書き込みます。	7-52
extern void **far_memcpy**(FARPTR *s1, FARPTR *s2, int n);	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 個の整数をコピーします。	7-52
extern void **far_near_memcpy**(void *, FARPTR, int n);	拡張メモリ・アドレスからページ 0 に n 個の整数をコピーします。	7-52
extern void **near_far_memcpy**(FARPTR, void *, int n);	ページ 0 から拡張メモリ・アドレスに n 個の整数をコピーします。	7-52

（d）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。	7-40
double **asin**(double x);	x のアーク・サインを戻します。	7-41
double **atan**(double x);	x のアーク・タンジェントを戻します。	7-42
double **atan2**(double y, double x);	y/x のアーク・タンジェントを戻します。	7-42
double **ceil**(double x);	≥ x の条件を満たす最小の整数を戻します。-x が使用されている場合、インライン展開します。	7-45
double **cos**(double x);	x のコサインを戻します。	7-47
double **cosh**(double x);	x のハイパボリック・コサインを戻します。	7-47
double **exp**(double x);	e^x を戻します。	7-50
double **fabs**(double x);	x の絶対値を戻します。	7-51
double **floor**(double x);	≤ x の条件を満たす最大の整数を戻します。-x が使用されている場合、インライン展開します。	7-55
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。	7-55
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。	7-59
double **ldexp**(double x, int exp);	$x \times 2^{exp}$ を戻します。	7-64

（d）浮動小数点算術関数（math.h/cmath）（続き）

関数	説明	ページ
double **log**(double x);	x の自然対数を戻します。	7-65
double **log10**(double x);	底が 10 の x の自然対数を戻します。	7-66
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。	7-71
double **pow**(double x, double y);	x^y を戻します。	7-72
double **sin**(double x);	x のサインを戻します。	7-80
double **sinh**(double x);	x のハイパボリック・サインを戻します。	7-80
double **sqrt**(double x);	x の負でない平方根を戻します。	7-81
double **tan**(double x);	x のタンジェントを戻します。	7-94
double **tanh**(double x);	x のハイパボリック・タンジェントを戻します。	7-95

（e）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。	7-78
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。	7-78

（f） 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。	7-98
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。	7-98
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。	7-98

（g）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。	7-10
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。	7-46
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。	7-53
int **feof**(FILE *_fp);	_fp が指すストリームの EOF 標識をテストします。	7-53
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。	7-53
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。	7-54
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。	7-54
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に、pos が指すオブジェクトを保存します。	7-54
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。	7-55
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。	7-56
int **fprintf**(FILE *_fp, const char *_format, ...);	_fp が指すストリームに書き込みます。	7-56
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。	7-56
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。	7-57
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。	7-57
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。	7-58
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。	7-59

（g）C 入出力関数（stdio.h/cstdio）（続き）

関数	説明	ページ
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。	7-59
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。	7-60
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。	7-60
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。	7-60
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。	7-61
int **getchar**(void);	fgetc() を呼び出し、stdin を引数として指定します。	7-61
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。	7-62
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。	7-72
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。	7-72
int **putc**(int _x, FILE *_fp);	fputc() と同じように実行するマクロです。	7-73
int **putchar**(int _x);	fputc() を呼び出し、stdout を出力ストリームとして使用するマクロです。	7-73
int **puts**(const char *_ptr);	_ptr が指す文字列を stdout に書き込みます。	7-73
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。	7-76
int **rename**(const char *_old_name, const char *_new_name);	_old_name が指す名前をもつファイルを、_new_name が指す名前で認識されるようにします。	7-76
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。	7-77
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。	7-77
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。	7-77

（g）C 入出力関数（stdio.h/cstdio）（続き）

関数	説明	ページ
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。	7-79
int **snprintf**(char *_string, const char *_format, ...);	sprintf() と同じ機能を実行しますが、文字列に書き込む文字数の上限を格納します。	7-81
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。	7-81
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。	7-82
FILE ***tmpfile**(void);	一時ファイルを作成します。	7-96
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。	7-96
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。	7-97
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。	7-99
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。	7-99
int **vsnprintf**(char *_string, const char *_format, va_list _ap);	vsprintf() と同じ機能を実行しますが、文字列に書き込む文字数の上限を格納します。	7-100
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。	7-100

（h）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。	7-39
int **abs**(int i);	val の絶対値を戻します。-x0 が使用されている場合を除き、インライン展開します。	7-39
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。	7-43
double **atof**(const char *st);	文字列を浮動小数点値に変換します。-x が使用されている場合、インライン展開します。	7-43
int **atoi**(register const char *st);	文字列を整数に変換します。	7-43

（h）汎用関数（stdlib.h/cstdlib）（続き）

関数	説明	ページ
long **atol**(register const char *st);	文字列を倍長整数値に変換します。-x が使用されている場合、インライン展開します。	7-43
void ***bsearch**(register const void *key, register const void *base, size_t nmemb, size_t size, int (*compar)(const void *,const void *));	nmemb 個のオブジェクトの配列から、key が指すオブジェクトを検索します。	7-44
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。	7-45
div_t **div**(register int numer, register int denom);	numer を denom で除算し、商と剰余を生成します。	7-49
void **exit**(int status);	プログラムを正常終了させます。	7-50
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。	7-58
char ***getenv**(const char *_string)	_string に関連した変数の環境情報を戻します。	7-61
long **labs**(long i);	i の絶対値を戻します。-x0 が使用されている場合を除き、インライン展開します。	7-39
ldiv_t **ldiv**(register long numer, register long denom);	numer を denom で除算します。	7-49
int **ltoa**(long val, char *buffer);	val を等価の文字列に変換します。	7-66
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。	7-67
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。	7-71
void **qsort**(void *base, size_t nmemb, size_t size, int (*compar) ());	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。	7-74
int **rand**(void);	0 ～ RAND_MAX の範囲の整数の疑似乱数シーケンスを戻します。	7-75
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。	7-75
void **srand**(unsigned int seed);	乱数発生ルーチンをリセットします。	7-75
double **strtod**(const char *st, char **endptr);	文字列を浮動小数点値に変換します。	7-92
long **strtol**(const char *st, char **endptr, int base);	文字列を倍長整数に変換します。	7-92
unsigned long **strtoul**(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。	7-92

(i) 文字列関数（string.h/cstring）

関数	説明	ページ
void ***memchr**(const void *cs, int c, size_t n);	cs の先頭の n 個の文字の中で c の最初の出現を検出します。-x が使用されている場合、インライン展開します。	7-67
int **memcmp**(const void *cs, const void *ct, size_t n);	cs の先頭の n 個の文字を ct と比較します。-x が使用されている場合、インライン展開します。	7-68
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。	7-68
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。	7-69
void ***memset**(void *mem, register int ch, register size_t length);	mem の先頭の length 個の文字に ch の値をコピーします。-x が使用されている場合、インライン展開します。	7-69
char ***strcat**(char *string1, const char *string2);	string2 を string1 の末尾に付加します。	7-82
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。	7-83
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。	7-84
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。	7-84
char ***strcpy**(register char *dest, register const char *src);	文字列 src を dest にコピーします。-x が使用されている場合、インライン展開します。	7-85
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。	7-85
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。	7-86
size_t **strlen**(const char *string);	文字列の長さを戻します。	7-87
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。	7-88
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。-x が使用されている場合、インライン展開します。	7-89
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。-x が使用されている場合、インライン展開します。	7-89

(i) 文字列関数（string.h/cstring）（続き）

関数	説明	ページ
char ***strpbrk**(const char *string, const char *chs);	string の中で、chs のいずれかの文字が最初に現れる位置を検索します。	7-90
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。-x が使用されている場合、インライン展開します。	7-91
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。	7-91
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。	7-92
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。	7-93
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。	7-94

(j) 時間関数（time.h/cstring）

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。	7-40
clock_t **clock**(void);	使用されたプロセッサ時間を判定します。	7-46
char ***ctime**(const time_t *timer);	カレンダ時を地方時に変換します。	7-48
double **difftime**(time_t time1, time_t time0);	2 つのカレンダ時の差を戻します。	7-48
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。	7-62
struct tm ***localtime**(const time_t *timer);	time_t の値を詳細時刻に変換します。	7-65
time_t **mktime**(register struct tm *tptr);	詳細時刻を time_t の値に変換します。	7-69
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。	7-86
time_t **time**(time_t *timer);	現在のカレンダ時を戻します。	7-95

7.5 ランタイムサポート関数およびマクロについて

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 ネーム・スペースに導入されている点では異なっています。

abort 中止

構文

```
#include <stdlib.h>
void abort(void);
```

C++ の構文

```
#include <cstdlib>
void std::abort(void);
```

定義される場所 rts.src の exit.c

説明 abort 関数により、プログラムが終了します。

例

```
if (error_detected)
   abort ();
```

abs/labs/llabs 絶対値

構文

```
#include    <stdlib.h>
int abs(int j);
long labs(long i);
long long llabs (long long k);
```

C++ の構文

```
#include <cstdlib>
int std::abs(int j);
long std::labs(long i);
long long std::llabs(long long k);
```

定義される場所 rts.src の abs.c

説明 C/C++ コンパイラは、以下のように、整数の絶対値を戻す 2 つの関数をサポートしています。

- ❏ abs 関数は、整数 j の絶対値を戻します。
- ❏ labs 関数は倍長整数 i の絶対値を戻します。
- ❏ llabs 関数は、倍長整数 k の絶対値を戻します。

acos　アーク・コサイン

構文

#include <math.h>

double **acos**(double x);

C++ の構文

#include <cmath>

double **std::acos**(double x);

定義される場所　rts.src の acos.c

説明

acos 関数は、浮動小数点引数 x のアーク・コサインを戻します。x の範囲は、[-1.1] とします。戻り値は、範囲 [0,π] のラジアンの角度です。

例

```
double realval, radians;

realval = 1.0;
radians = acos(realval);
return (radians);            /* acos return π/2 */
```

asctime　文字列への内部時間

構文

#include　<time.h>

char *asctime(const struct tm *timeptr**);**

C++ の構文

#include <ctime>

char ***std::asctime**(const struct tm *timeptr);

定義される場所　rts.src の asctime.c

説明

asctime 関数は、詳細時刻を以下の形式の文字列に変換します。

```
Mon Jan 11 11:18:36 1988 \n\0
```

この関数は、変換された文字列へのポインタを戻します。

time.h ヘッダで宣言と定義が行われる関数と型の詳細は、7.3.17 項「時間関数（time.h/ctime）」（7-27 ページ）を参照してください。

asin　アーク・サイン

構文

#include <math.h>

double **asin**(double x);

C++ の構文

#include <cmath>

double **std::asin**(double x);

定義される場所　rts.src の asin.c

説明　asin 関数は、浮動小数点引数 x のアーク・サインを戻します。x の範囲は、[-1, 1] とします。戻り値は、範囲 [-π/2, π/2] のラジアンの角度です。

例

```
double realval, radians;
realval = 1.0;
radians = asin(realval);  /* asin returns π/2 */
```

assert　診断情報マクロの挿入

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　マクロとしての assert.h/cassert

説明　assert マクロは、式をテストします。式の値に基づき、メッセージを発行して実行を打ち切るか、実行を継続します。このマクロはデバックのときに便利です。

- ❏ 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に書き込み、実行を打ち切ります。
- ❏ expr が真の場合、assert マクロは何も実行しません。

assert マクロを宣言するヘッダ・ファイルは、別のマクロ、NDEBUG を参照します。assert.h ヘッダがソース・ファイルに組み込まれるときに NDEBUG をマクロ名として定義した場合、assert マクロは次のように定義されます。

#define assert(ignore)

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　極座標のアーク・タンジェント

構文

#include　<math.h>

double **atan**(double x);

C++ の構文

#include <cmath>

double **std::atan**(double x);

定義される場所　rts.src の atan.c

説明

atan 関数は、浮動小数点引数 x のアーク・タンジェントを戻します。戻り値は、範囲 [-π/2, π/2] のラジアンの角度です。

例

```
double realval, radians;

realval = 0.0;
radians = atan(realval);     /* return value = 0 */
```

atan2　デカルト座標のアーク・タンジェント

構文

#include　<math.h>

double **atan2**(double y, double x);

C++ の構文

#include <cmath>

double **std::atan2**(double y, double x);

定義される場所　rts.src の atan2.c

説明

atan2 関数は、y/x の逆タンジェントを戻します。この関数は、これらの引数の符号を使用して、戻り値の座標象限を判定します。どちらの引数も 0 にすることはできません。戻り値は、範囲 [-π, π] のラジアンの角度です。

例

```
double rvalu, rvalv;
double radians;

rvalu  = 0.0;
rvalv  = 1.0;
radians = atan2(rvalu, rvalv);   /* return value = 0 */
```

atexit　Exit () から呼び出される関数の登録

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 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　文字列から数値への変換

構文

```
#include    <stdlib.h>
double atof(const char *st);
int atoi(const char *st);
long atol(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);
```

定義される場所　rts.src の atof.c、atoi.c、および atol.c

説明

上記の 3 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 関数は文字列を倍長整数に変換します。引数 st は文字列を指します。文字列の形式は次のとおりです。

  [*space*] [*sign*] *digits*

space は、スペース（文字）、水平タブか垂直タブ、復帰、書式送り、あるいは改行文字で表します。*space* の後は、オプションの *sign*、さらに数値の整数部を示す *digits* が続きます。その後は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

構文

```
#include    <stdlib.h>

void *bsearch(register const void *key, register const void *base,
                    size_t nmemb, size_t size,
                    int (*compar)(const void *,const void *));
```

C++ の構文

```
#include <cstdlib>

void *std::bsearch(register const void *key, register const void *base,
                    size_t nmemb, size_t size,
                    int (*compar)(const void *,const void *));
```

定義される場所 rts.src の bsearch.c

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。

calloc　メモリの割り当ておよびクリア

構文

#include <stdlib.h>

void ***calloc**(size_t num, size_t size);

C++ の構文

#include <cstdlib>

void ***std::calloc**(size_t num, size_t size);

定義される場所　rts.src の memory.c

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 2000 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、6.1.6 項「動的なメモリ割り当て」（6-7 ページ）を参照してください。

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　切り上げ

構文

#include <math.h>

double **ceil**(double x);

C++ の構文

#include <cmath>

double **std::ceil**(double x);

定義される場所　rts.src の ceil.c

説明

ceil 関数は、x 以上の最小の整数を表す浮動小数点数を戻します。

例

```
extern double ceil();
double answer;
answer = ceil(3.1415);   /* answer = 4.0 */
answer = ceil(-3.5);     /* answer = -3.0 */
```

clearerr　EOF およびエラー標識のクリア

構文

```
#include    <stdio.h>
void clearerr(FILE *_fp);
```

C++ の構文

```
#include <cstdio>
void std::clearerr(FILE *_fp);
```

定義される場所　rts.src の clearerr

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

構文

```
#include    <time.h>
clock_t clock(void);
```

C++ の構文

```
#include <ctime>
clock_t std::clock(void);
```

定義される場所　rts.src の clock.c

説明　clock 関数は、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 関数の記述**
>
> clock 関数はホスト・システム固有の関数です。よって、ユーザ固有の clock 関数を記述する必要があります。また、clock() - クロックの目盛数 - が戻す値が CLOCKS_PER_SEC で割り、秒単位で値を作成できるよう、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。

cos　コサイン

構文

#include <math.h>

double **cos**(double x);

C++ の構文

#include <cstdio>

double **std::cos**(double x);

定義される場所　rts.src の cos.c

説明

cos 関数は、浮動小数点数 x のコサインを戻します。角度 x は、ラジアンで表します。引数の値が大き過ぎると、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 radians, cval;/* cos returns cval */
radians = 3.1415927;
cval = cos(radians);/* return value = -1.0 */
```

cosh　ハイパボリック・コサイン

構文

#include <math.h>

double **cosh**(double x);

C++ の構文

#include <cstdio>

double **std::cosh**(double x);

定義される場所　rts.src の cosh.c

説明

cosh 関数は、浮動小数点数 x のハイパボリック・コサインを戻します。引数の値が大きすぎると範囲エラーになります。

例

```
double x, y;

x = 0.0;
y = cosh(x);/* return value = 1.0 */
```

ctime　カレンダ時

構文

#include　<time.h>

char ***ctime**(const time_t *timer);

C++ の構文

#include <cstdio>

char ***std::ctime**(const time_ *timer);

定義される場所　rts.src の ctime.c

説明

ctime 関数は、カレンダ時（timer が指す時刻）を文字列の形式の地方時に変換します。これは、次のように指定しても同じ結果になります。

```
asctime(localtime(timer))
```

この関数は、asctime 関数が戻すポインタを戻します。

time.h ヘッダで宣言と定義が行われる関数と型の詳細は、7.3.17 項「時間関数（time.h/ctime）」（7-27 ページ）を参照してください。

difftime　時差

構文

#include　<time.h>

double **difftime**(time_t time1, time_t time0);

C++ の構文

#include <cstdio>

double **std::difftime**(time_t time1, time_t time0);

定義される場所　rts.src の difftime.c

説明

difftime 関数は、2 つのカレンダ時の差、time1 から time0 を引いた値を計算します。戻り値の単位は秒です。

time.h ヘッダで宣言と定義が行われる関数と型の詳細は、7.3.17 項「時間関数（time.h/ctime）」（7-27 ページ）を参照してください。

div/ldiv/lldiv 除算

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 <cstdio>

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 の div.c

説明

2 つの関数による整数の除算では、numer（分子）を denom（分母）で割った値が戻されます。これらの関数を使用すれば、1 回の演算で商と剰余の両方を得ることができます。

- div 関数は、整数の除算を行います。入力する引数は整数です。この関数は、商と剰余を型 div_t の構造体で戻します。構造体は、次のように定義します。

```
typedef struct
{
   int  quot;             /*  quotient     */
   int  rem;              /* remainder     */
} div_t;
```

- ldiv 関数は、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 ldiv_t の構造体で戻します。構造体は次のように定義します。

```
typedef struct
{
   long int  quot;        /*  quotient     */
   long int  rem;         /* remainder     */
} ldiv_t;
```

- lldiv 関数は、long long integer の除算を行います。入力する引数は long long integer 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 int quot, rem; } 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

構文

#include　<stdlib.h>

void **exit**(int status);

C++ の構文

#include <cstdio>

void **std::exit**(int status);

定義される場所　rts.src の exit.c

説明

exit 関数は、プログラムを正常に終了します。atexit 関数で登録された関数は、すべてその登録の順序とは逆の順序で呼び出されます。exit 関数は EXIT_FAILURE を値として使用できます（7-39 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　指数

構文

#include　<math.h>

double **exp**(double x);

C++ の構文

#include <cstdio>

double **std::exp**(double x);

定義される場所　rts.src の exp.c

説明

exp 関数は、実数 x の指数関数を戻します。戻り値は e^x です。x の値が大きすぎると範囲エラーになります。

例

```
double x, y;

x = 2.0;
y = exp(x);     /* y = 7.38, which is e**2.0 */
```

fabs　絶対値

構文

#include <math.h>

double **fabs**(double x);

C++ の構文

#include <cstdio>

double **std::fabs**(double x);

定義される場所

rts.src の fabs.c

説明

fabs 関数は、浮動小数点数 x の絶対値を戻します。

例

```
double x, y;

x = -57.5;
y = fabs(x);        /* return value = +57.5 */
```

far_peek / far_peek_l　拡張メモリ・アドレスからの読み込み

構文

#include <extaddr.h>

extern int **far_peek**(FARPTR x);
extern unsigned long **far_peek_l**(FARPTR x);

定義される場所

rts.src の extaddr.asm

説明

far_peek 関数は拡張メモリ・アドレスから整数を読み込みます。far_peek_l 関数は拡張メモリ・アドレスから unsigned long を読み込みます。

例

```
int ilocal;
int llocal;
FARPTR iptr = (FARPTR) &ival;
FARPTR lptr = (FARPTR) &lval;
far_poke(iptr, 100);
ilocal = far_peek(iptr) + 10;
far_poke_l(lptr, 0x7ffffffe);
llocal = far_peek_l(lptr) + 1;
```

far_poke / far_poke_l　拡張メモリ・アドレスへの書き込み

構文

#include <extaddr.h>;

extern void **far_poke**(FARPTR x, int x);
extern void **far_poke_l**(FARPTR x, unsigned long x);

定義される場所 rts.src の extaddr.asm

説明 far_poke 関数は拡張メモリ・アドレスに整数を書き込みます。far_poke 1 関数は拡張メモリ・アドレスに unsigned long を書き込みます。

例 far_peek のコード例を参照してください。

far_memcpy　拡張メモリ・アドレス間のメモリ・ブロック・コピー

構文

#include <extaddr.h>;

extern void **far_memcpy**(FARPTR *s1, FARPTR *s2, int n);

定義される場所 rts.src の extaddr.asm

説明 far_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 個の整数をコピーします。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。

例

```
int a[10] = {0,1,2,3,4,5,6,7,8,9};
int b[10];
far_memcpy((FARPTR) &b, (FARPTR) &a, 10);
```

far_near_memcpy / near_far_memcpy　拡張メモリからまたは拡張メモリへのメモリ・ブロック・コピー

構文

#include <extaddr.h>;

extern void **far_near_memcpy**(void *, FARPTR, int);
extern void **near_far_memcpy**(FARPTR, void *, int);

定義される場所 rts.src の extaddr.asm

説明 far_near_memcpy 関数は、拡張メモリ・アドレスからページ 0 に整数をコピーします。near_far_memcpy 関数は、ページ 0 から拡張メモリ・アドレスに整数をコピーします。

例
```
char c[12] = {“test string”};
char d[12];
far_near_memcpy((void *) d, (FARPTR) &c, 12);
```

fclose ファイルのクローズ

構文
#include <stdio.h>
int **fclose**(FILE *_fp);

C++ の構文
#include <cstdio>
int **std::fclose**(FILE *_fp);

定義される場所 rts.src の fclose.c

説明 fclose 関数は、_fp が指すストリームをフラッシュし、そのストリームに関連したファイルをクローズします。

feof EOF 標識のテスト

構文
#include <stdio.h>
int **feof**(FILE *_fp);

C++ の構文
#include <cstdio>
int **std::feof**(FILE *_fp);

定義される場所 rts.src の feof.c

説明 feof 関数は、_fp が指すストリームの EOF 標識をテストします。

ferror エラー標識のテスト

構文
#include <stdio.h>
int **ferror**(FILE *_fp);

C++ の構文
#include <cstdio>
int **std::ferror**(FILE *_fp);

定義される場所 rts.src の ferror.c

説明 ferror 関数は、_fp が指すストリーム のエラー標識をテストします。

fflush　入出力バッファのフラッシュ

構文
#include　<stdio.h>
int **fflush**(register FILE *_fp);

C++ の構文
#include <cstdio>
int **std::fflush**(register FILE *_fp);

定義される場所
rts.src の fflush.c

説明
fflush 関数は、_fp が指すストリームの入出力バッファをフラッシュします。

fgetc　次の文字の読み込み

構文
#include　<stdio.h>
int **fgetc**(register FILE *_fp);

C++ の構文
#include <cstdio>
int **std::fgetc**(register FILE *_fp);

定義される場所
rts.src の fgetc.c

説明
fgetc 関数は、_fp が指すストリーム内の次の文字を読み込みます。

fgetpos　オブジェクトの格納

構文
#include　<stdio.h>
int **fgetpos**(FILE *_fp, fpos_t *pos);

C++ の構文
#include <cstdio>
int **std::fgetpos**(FILE *_fp, fpos_t *pos);

定義される場所
rts.src の fgetpos.c

説明
fgetpos 関数は、_fp が指すストリームのファイル位置標識の現行値に、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

構文
#include <stdio.h>
char ***fgets**(char *_ptr, register int _size, register FILE *_fp);

C++ の構文
#include <cstdio>
char ***std::fgets**(char *_ptr, register int _size, register FILE *_fp);

定義される場所 rts.src の fgets.c

説明 fgets 関数は、指定した数の文字を、_fp が指すストリームから読み込みます。文字は、_ptr で指定された配列に入れます。読み込まれる文字の数は _size -1 です。

floor　切り捨て

構文
#include <math.h>
double **floor**(double x);

C++ の構文
#include <cstdio>
double **std::floor**(double x);

定義される場所 rts.src の floor.c

説明 floor 関数は、x 以下の最大の整数を表す浮動小数点数を戻します。

例
```
double answer;

answer = floor(3.1415);      /* answer =   3.0 */
answer = floor(-3.5);        /* answer =  -4.0 */
```

fmod　浮動小数点の剰余

構文
#include <math.h>
double **fmod**(double x, double y);

C++ の構文
#include <cstdio>
double **std::fmod**(double x, double y);

定義される場所 rts.src の fmod.c

説明 fmod 関数は、x を y で割った浮動小数点の剰余を戻します。y==0 の場合、関数は 0 を戻します。

例
```
double x, y, r;

x = 11.0;
y = 5.0;
r = fmod(x, y);              /* fmod returns 1.0 */
```

fopen　ファイルのオープン

構文

```
#include    <stdio.h>
FILE *fopen(const char *_fname, const char *_mode);
```

C++ の構文

```
#include <cstdio>
FILE *std::fopen(const char *_fname, const char *_mode);
```

定義される場所 rts.src の fopen.c

説明 fopen 関数は、_fname が指すファイルをオープンします。_mode が指す文字列にはファイルのオープン方法が記述されています。UNIX では、バイナリ読み出しには rb として、またはバイナリ書き込みには wb としてモードを指定します。

fprintf　ストリームの書き込み

構文

```
#include    <stdio.h>
int fprintf(FILE *_fp, const char *_format, ...);
```

C++ の構文

```
#include <cstdio>
int std::fprintf(FILE *_fp, const char *_format, ...);
```

定義される場所 rts.src の fprintf.c

説明 fprintf 関数は、_fp が指すストリームへの書き込みを行います。_format が指す文字列には、ストリームを書き込む方法が記述されています。

fputc　文字の書き込み

構文

```
#include    <stdio.h>
int fputc(int _c, register FILE *_fp);
```

C++ の構文

```
#include <cstdio>
int std::fputc(int _c, register FILE *_fp);
```

定義される場所 rts.src の fputc.c

説明 fputc 関数は、_fp が指すストリームに 1 文字を書き込みます。

fputs　文字列の書き込み

構文

```
#include    <stdio.h>
int fputs(const char *_ptr, register FILE *_fp);
```

C++ の構文

```
#include <cstdio> s
int std::fputs(const char *_ptr, register FILE *_fp);
```

定義される場所　rts.src の fputs.c

説明　fputs 関数 は、_fp が指すストリームに、_ptr が指す文字列を書き込みます。

fread　ストリームの読み込み

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 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。

> **注：C55x の char がホストのバイトと異なる場合の fread の使用方法**
>
> C55x の char がホストのバイトと異なる場合の入出力の処理方法については、8 ビットより多い char によるターゲット上のバイナリ・ファイルの読み書き（文献番号 SPRA757）を参照してください。

free　メモリの解放

構文

```
#include    <stdlib.h>
void free(void *ptr);
```

C++ の構文

```
#include <cstdlib>
void std::free(void *ptr);
```

定義される場所　rts.src の memory.c

説明　free 関数は、malloc、calloc、realloc の呼び出しにより割り当てられた（ptr が指す）メモリ空間を解放します。これにより、メモリ空間を再び利用できます。割り当て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、6.1.6 項「動的なメモリ割り当て」（6-7 ページ）を参照してください。

例　この例では 10 バイトを割り当ててから、解放します。

```
char *x;
x = malloc(10);              /*  allocate 10 bytes   */
free(x);                     /*  free 10 bytes       */
```

freopen　ファイルのオープン

構文

```
#include    <stdio.h>

FILE *freopen(const char *_fname, const char *_mode, register FILE *_fp);
```

C++ の構文

```
#include <cstdio>

FILE *std::freopen(const char *_fname, const char *_mode, register FILE *_fp);
```

定義される場所　rts.src の freopen.c

説明　freopen 関数は、_fname が指すファイルをオープンし、_fp が指すストリーム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　小数部と指数部

構文

#include <math.h>

double **frexp**(double value, int *exp);

C++ の構文

#include <cmath>

double **std::frexp**(double value, int *exp);

定義される場所　rts.src の frexp.c

説明

frexp 関数は、浮動小数点数を正規化した小数部と 2 の整数乗に分けます。関数は範囲 [1/2, 1] または 0 の値を戻すため、value = =x × 2^{exp} となります。frexp 関数は、累乗を exp が指す整数に保存します。value が 0 の場合、小数部、指数部ともに 0 が戻ります。

例

```
double fraction;
int exp;

fraction = frexp(3.0, &exp);
/* after execution, fraction is .75 and exp is 2 */
```

fscanf　ストリームの読み込み

構文

#include <stdio.h>

int **fscanf**(FILE *_fp, const char *_fmt, ...);

C++ の構文

#include <cstdio>

int **std::fscanf**(FILE *_fp, const char *_fmt, ...);

定義される場所　rts.src の fscanf.c

説明

fscanf 関数は、_fp が指すストリームからの読み込みを行います。_fmt が指す文字列には、ストリームの読み込み方法が記述されています。

fseek　ファイル位置標識の設定

構文

#include <stdio.h>

int **fseek**(register FILE *_fp, long _offset, int _ptrname);

C++ の構文

#include <cstdio>

int **std::fseek**(register FILE *_fp, long _offset, int _ptrname);

定義される場所　rts.src の fseek.c

説明

fseek 関数は、_fp が指すストリームのファイル位置標識を設定します。位置は _ptrname で指定されます。バイナリ・ファイルの場合、_offset を使用して _ptrname から標識の位置を設定します。テキスト・ファイルの場合、オフセット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

構文　#include　<stdio.h>

int **fsetpos**(FILE *_fp, const fpos_t *_pos);

C++ の構文　#include <cstdio>

int **std::fsetpos**(FILE *_fp, const fpos_t *_pos);

定義される場所　rts.src の fsetpos.c

説明　fsetpos 関数は、_fp が指すストリームのファイル位置標識を、_pos に設定します。ポインタ _pos の値は、同じストリームに対する fgetpos() からの値を指定する必要があります。

ftell　現行のファイル位置標識の取得

構文　#include　<stdio.h>

long **ftell**(FILE *_fp);

C++ の構文　#include <cstdio>

long **ftell**(FILE *_fp);

定義される場所　rts.src の ftell.c

説明　ftell 関数は、_fp が指すストリームのファイル位置標識の現行値を取得します。

fwrite　データ・ブロックの書き込み

構文　#include　<stdio.h>

size_t **fwrite**(const void *_ptr, size_t _size, size_t _count, register FILE *_fp);

C++ の構文　#include <cstdio>

size_t **std::fwrite**(const void *_ptr, size_t _size, size_t _count, register FILE *_fp);

定義される場所　rts.src の fwrite.c

説明　fwrite 関数は _ptr が指すメモリから、_fp が指すストリームにデータ・ブロックを書き込みます。

getc　次の文字の読み込み

構文
#include　<stdio.h>
int **getc**(FILE *_fp);

C++ の構文
#include <cstdio>
int **std::getc**(FILE *_fp);

定義される場所 rts.src の fgetc.c

説明 getc 関数は、_fp が指すファイル内の次の文字を読み込みます。

getchar　標準入力からの次の文字の読み込み

構文
#include　<stdio.h>
int **getchar**(void);

C++ の構文
#include <cstdio>
int **std::getchar**(void);

定義される場所 rts.src の fgetc.c

説明 getchar 関数は、標準入力デバイスから次の文字を読み込みます。

getenv　環境情報の取得

構文
#include　<stdlib.h>
char ***getenv**(const char *_string);

C++ の構文
#include <cstdlib>
char ***std::getenv**(const char *_string);

定義される場所 rts.src の trgdrv.c

説明 getenv 関数は、_string に関連した変数の環境情報を戻します。

gets　標準入力からの次行の読み込み

構文

#include　<stdio.h>

char ***gets**(char *_ptr);

C++ の構文

#include <cstdio>

char ***std::gets**(char *_ptr);

定義される場所　rts.src の fgets.c

説明

gets 関数は、標準入力デバイスから入力行を読み込みます。文字は、_ptr で指定された配列に入ります。

gmtime　グリニッジ標準時

構文

#include　<time.h>

struct tm ***gmtime**(const time_t *timer);

C++ の構文

#include <ctime>

struct tm ***std::gmtime**(const time_t *timer);

定義される場所　rts.src の gmtime.c

説明

gmtime 関数は、カレンダ時（timer が指す）をグリニッジ標準時で表される詳細時刻に変換します。

time.h ヘッダで宣言と定義が行われる関数と型の詳細は、7.3.17 項「時間関数（time.h/ctime）」（7-27 ページ）を参照してください。

isxxx　文字の判別

構文

#include　<ctype.h>

int **isalnum**(int c);int **islower**(int c);
int **isalpha**(int c);int **isprint**(int c);
int **isascii**(int c);int **ispunct**(int c);
int **iscntrl**(int c);int **isspace**(int c);
int **isdigit**(int c);int **isupper**(int c);
int **isgraph**(int c);int **isxdigit**(int c);

C++ の構文

#include <cctype>

int **std::isalnum**(int c);int **std::islower**(int c);
int **std::isalpha**(int c);int **std::isprint**(int c);
int **std::isascii**(int c);int **std::ispunct**(int c);
int **std::iscntrl**(int c);int **std::isspace**(int c);
int **std::isdigit**(int c);int **std::isupper**(int c);
int **std::isgraph**(int c);int **std::isxdigit**(int c);

定義される場所

rts.src の isxxx.c と ctype.c
またはマクロとして ctype.h/cctype にて定義

説明

これらの関数は 1 つの引数 c をテストし、それが 英字、英数字、数字、ASCII など特定の種類の文字であるかどうかを調べます。テストの結果が真の場合、この関数は 0 以外の値を戻します。テストの結果が偽の場合、この関数は 0 を戻します。文字判別関数には次のような関数があります。

isalnum　英数字 ASCII 文字を認識します（isalpha や isdigit が真となるすべての文字をテストします)。

isalpha　英字 ASCII 文字を認識します（islower や isupper が真となるすべての文字をテストします)。

isascii　ASCII 文字（0 ～ 127 の範囲の文字）を認識します。

iscntrl　制御文字（0 ～ 31 の範囲と 127 の ASCII 文字）を認識します。

isdigit　数字（0 ～ 9）を認識します。

isgraph　空白以外の文字を認識します。

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 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

7-39 ページの abs/labs を参照してください。

ldexp

2 の累乗による乗算

構文

#include <math.h>

double **ldexp**(double x, int exp);

C++ の構文

#include <cmath>

double **std::ldexp**(double x, int exp);

定義される場所 rts.src の ldexp.c

説明

ldexp 関数は、浮動小数点数と 2 の累乗を乗算し、$x \times 2^{exp}$ を戻します。*exp* は、負の値または正の値のどちらでも構いません。結果の値が大きすぎると、範囲エラーになります。

例

```
double result;

result = ldexp(1.5, 5);          /* result is 48.0 */
result = ldexp(6.0, -3);         /* result is 0.75 */
```

ldiv/lldiv

7-49 ページの div/ldiv を参照してください。

localtime 地方時

構文

```
#include   <time.h>
struct tm *localtime(const time_t *timer);
```

C++ の構文

```
#include <ctime>
struct tm *std::localtime(const time_t *timer);
```

定義される場所 rts.src の localtime.c

説明 localtime 関数は、カレンダ時（timer が指す）を地方時で表される詳細時刻に変換します。この関数は、変換された時刻へのポインタを戻します。

time.h ヘッダで宣言と定義が行われる関数と型の詳細は、7.3.17 項「時間関数（time.h/ctime）」（7-27 ページ）を参照してください。

log 自然対数

構文

```
#include   <math.h>
double log(double x);
```

C++ の構文

```
#include <cmath>
double std::log(double x);
```

定義される場所 rts.src の log.c

説明 log 関数は、実数 x の自然対数を戻します。x が負の場合は、領域エラーになります。x が 0 の場合は、範囲エラーになります。

例

```
float x, y;

x = 2.718282;
y = log(x);          /* Return value = 1.0 */
```

log10 常用対数

構文

#include <math.h>

double **log10**(double x);

C++ の構文

#include <cmath>

double **std::log10**(double x);

定義される場所 rts.src の log10.c

説明 log10 関数は、底が 10 の実数 x の対数を戻します。x が負の場合は、領域エラーになります。x が 0 の場合は、範囲エラーになります。

例

```
float x, y;

x = 10.0;
y = log(x);          /* Return value = 1.0 */
```

longjmp 7-78 ページの setjmp/longjmp を参照してください。

ltoa 倍長整数から ASCII へ

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 **ltoa**(long val, char *buffer);

定義される場所 rts.src 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、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。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

構文

#include　<stdlib.h>

void ***malloc**(size_t size);

C++ の構文

#include <cstdlib>

void ***std::malloc**(size_t size);

定義される場所　rts.src の memory.c

説明

malloc 関数は size 16 ビット・バイトの空間をオブジェクトに割り当て、その空間のポインタを戻します。malloc でパケットを割り当てることができない場合（つまりメモリ不足のとき）は、ヌル・ポインタ（0）が戻ります。この関数では、割り当てたメモリの内容変更はしません。

malloc が使用するメモリは、特別なメモリ・プール（ヒープ）内にあります。定数 __SYSMEM_SIZE により、ヒープのサイズは 2000 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、6.1.6 項「動的なメモリ割り当て」（6-7 ページ）を参照してください。

memchr　バイトの最初の出現を検出

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 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　メモリ比較

構文

#include　<string.h>

int **memcmp**(const void *cs, const void *ct, size_t n);

C++ の構文

#include <cstring>

int **std::memcmp**(const void *cs, const void *ct, size_t n);

定義される場所　rts.src の memcmp.c

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 - 非オーバーラップ

構文

#include　<string.h>

void ***memcpy**(void *s1, const void *s2, size_t n);

C++ の構文

#include <cstring>

void ***std:memcpy**(void *s1, const void *s2, size_t n);

定義される場所　rts.src の memcpy.c

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

構文

#include　<string.h>

void ***memmove**(void *s1, const void *s2, size_t n);

C++ の構文

#include <cstring>

void ***std::memmove**(void *s1, const void *s2, size_t n);

定義される場所　rts.src の memmove.c

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　メモリの値のコピー

構文

#include　<string.h>

void ***memset**(void *mem, register int ch, size_t length);

C++ の構文

#include <cstring>

void ***std::memset**(void *mem, register int ch, size_t length);

定義される場所　rts.src の memset.c

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。

mktime　カレンダ時への変換

構文

#include　<time.h>

time_t　***mktime**(struct tm *timeptr);

C++ の構文

#include <ctime>

time_t　***std::mktime**(struct tm *timeptr);

定義される場所　rts.src の mktime.c

説明

mktime 関数は、地方時で表された詳細時刻を対応するカレンダ時に変換します。timeptr 引数は、詳細時刻を保持する構造体を指します。

この関数では、tm_wday と tm_yday の元の値は無視されます。また、構造体内に設定する値の範囲を制限しません。時間の変換が正常に完了すると tm_wday と tm_yday は適切に設定され、構造体の他の構成要素には制限範囲内の値が設定されます。tm_mday の最終値は、tm_mon と tm_year が決定されるまで設定されません。

戻り値は型 time_t の値としてエンコードされます。カレンダ時を表現できない場合、この関数は値 -1 を戻します。

time.h ヘッダで宣言と定義が行われる関数と型の詳細は、7.3.17 項「時間関数（time.h/ctime）」（7-27 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 */
```

minit　動的なメモリ・プールのリセット

構文

プロトタイプはありません。

void **minit**(void);

定義される場所　rts.src 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 2000 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、6.1.6 項「動的なメモリ割り当て」（6-7 ページ）を参照してください。

注：minit の後では以前に割り当てられているオブジェクトは使用不可能

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が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なくなるため、アクセスを試みるのはやめてください。

modf　符号付き整数および符号付小数

構文

#include　<math.h>

double **modf**(double value, double *iptr);

C++ の構文

#include <cmath>

double **std::modf**(double value, double *iptr);

定義される場所　rts.src の modf.c

説明

modf 関数は、値を符号付き整数と符号付き小数に分けます。この 2 つの部分の符号は、入力した引数の符号と同じです。この関数は値の小数部を戻し、整数部を iptr で指定したオブジェクトに倍精度浮動小数点値として保存します。

例

```
double value, ipart, fpart;

value = -3.1415;

fpart = modf(value, &ipart);

/* After execution, ipart contains -3.0,    */
/* and fpart contains -0.1415.              */
```

perror　エラー番号のマップ

構文

```
#include    <stdio.h>
void perror(const char *_s);
```

C++ の構文

```
#include <cstdio>
void std::perror(const char *_s);
```

定義される場所　rts.src の perror.c

説明　perror 関数は、s 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。

pow　累乗

構文

```
#include    <math.h>
double pow(double x, double y);
```

C++ の構文

```
#include <cmath>
double std::pow(double x, double y);
```

定義される場所　rts.src の pow.c

説明　pow 関数は、x の y 乗を戻します。x = 0 かつ y ≤ 0 の場合、または x が負で y が整数でない場合は、領域エラーになります。結果の値が大きすぎて表示できない場合は、範囲エラーになります。

例

```
double x, y, z;

x = 2.0;
y = 3.0;
x = pow(x, y);          /* return value = 8.0 */
```

printf　標準出力への書き込み

構文

```
#include    <stdio.h>
int printf(const char *_format, ...);
```

C++ の構文

```
#include <cstdio>
int std::printf(const char *_format, ...);
```

定義される場所　rts.src の printf.c

説明　printf 関数は、標準出力デバイスへの書き込みを行います。_format が指す文字列には、ストリームを書き込む方法が記述されています。

putc　文字の書き込み

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 の putc.c

説明 putc 関数は、_fp が指すストリームに 1 文字を書き込みます。

putchar　標準出力への文字の書き込み

構文

```
#include    <stdio.h>
int putchar(int _x);
```

C++ の構文

```
#include <cstdio>
int std::putchar(int _x);
```

定義される場所 rts.src の putchar.c

説明 putchar 関数は、標準出力デバイスに 1 文字を書き込みます。

puts　標準出力への書き込み

構文

```
#include    <stdio.h>
int puts(const char *_ptr);
```

C++ の構文

```
#include <cstdio>
int std::puts(const char *_ptr);
```

定義される場所 rts.src の puts.c

説明 puts 関数は、_ptr が指す文字列を標準出力デバイスに書き込みます。

qsort　配列のソート

構文

#include　<stdlib.h>

void **qsort**(void *base, size_t nmemb, size_t size, int (*compar) ());

C++ の構文

#include <cstdlib>

void **std::qsort**(void *base, size_t nmemb, size_t size, int (*compar) ());

定義される場所

rts.src の qsort.c

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。

rand/srand　乱整数

構文

```
#include    <stdlib.h>

int rand(void);
void srand(unsigned int seed);
```

C++ の構文

```
#include <cstdlib>

int std::rand(void);
void std::srand(unsigned int seed);
```

定義される場所　rts.src の rand.c

説明

2 つの関数がともに機能して、疑似乱数シーケンスを生成します。

- ❏ rand 関数は、0 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

構文

```
#include    <stdlib.h>

void *realloc(void *packet, size_t size);
```

C++ の構文

```
#include <cstdlib>

void *std::realloc(void *packet, size_t size);
```

定義される場所　rts.src の memory.c

説明

realloc 関数は、packet が指す割り当て済みのメモリのサイズを、size 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。

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 2000 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、6.1.6 項「動的なメモリ割り当て」（6-7 ページ）を参照してください。

remove　ファイルの除去

構文

```
#include    <stdio.h>
int remove(const char *_file);
```

C++ の構文

```
#include <cstdio>
int std::remove(const char *_file);
```

定義される場所　rts.src の remove.c

説明　remove 関数は、_file が指すファイルをその名前では使用できないようにします。

rename　ファイルの再命名

構文

```
#include    <stdio.h>
int rename(const char *old_name, const char *new_name);
```

C++ の構文

```
#include <cstdio>
int std::rename(const char *old_name, const char *new_name);
```

定義される場所　rts.src の rename.c

説明　rename 関数は、old_name が指すファイルの名前を変更します。新しい名前は、new_name が指しています。

rewind　ファイルの先頭へのファイル位置標識の設定

構文　#include　<stdio.h>

void **rewind**(register FILE *_fp);

C++ の構文　#include <cstdio>

void **std::rewind**(register FILE *_fp);

定義される場所　rts.src の rewind.c

説明　rewind 関数は、_fp が指すストリームのファイル位置標識を、ファイルの先頭に設定します。

scanf　標準入力からのストリームの読み込み

構文　#include　<stdio.h>

int **scanf**(const char *_fmt, ...);

C++ の構文　#include <cstdio>

int **std::scanf**(const char *_fmt, ...);

定義される場所　rts.src の fscanf.c

説明　scanf 関数は、標準入力デバイスからストリームを読み込みます。_fmt が指す文字列には、ストリームの読み込み方法が記述されています。

setbuf　ストリーム用のバッファの指定

構文　#include　<stdio.h>

void **setbuf**(register FILE *_fp, char *_buf);

C++ の構文　#include <cstdio>

void **std::setbuf**(register FILE *_fp, char *_buf);

定義される場所　rts.src の setbuf.c

説明　setbuf 関数は、_fp が指すストリームに使用されるバッファを指定します。_buf をヌルに設定すると、バッファリングはオフになります。値は戻りません。

setjmp/longjmp 非ローカル・ジャンプ

構文

#include <setjmp.h>

int **setjmp**(jmp_buf env)
void **longjmp**(jmp_buf env, int _val)

C++ の構文

#include <csetjmp>

int **std::setjmp**(jmp_buf env)
void **std::longjmp**(jmp_buf env, int _val)

定義される場所 rts.src の setjmp.asm

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、マクロ、および関数を定義します。

- **jmp_buf** 型 - 呼び出し環境の復元に必要な情報の保存に適している配列型です。
- **setjmp** マクロ - 後で longjmp 関数で使用できるように、呼び出し環境を jmp_buf 引数に保存します。

  直接の呼び出しからの戻りの場合、setjmp マクロは 0 を戻します。呼び出しから longjmp 関数へ戻す場合、setjmp マクロは 0 以外の値を戻します。
- **longjmp** 関数 - setjmp マクロの一番最後の呼び出しにより jmp_buf 引数に保存された環境を復元します。setjmp マクロが呼び出されなかった場合や setjmp マクロが異常終了した場合には、longjmp の動作は予測できません。

  longjmp が完了した後、対応する setjmp の呼び出しにより _val で指定した値が戻った場合と同様に、プログラムは引き続き実行されます。たとえ _val が 0 でも、longjmp 関数は、setjmp に値 0 を戻すことはありません。_val が 0 の場合、setjmp マクロは値 1 を戻します。

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

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 の setvbuf.c

説明

setvbuf 関数は、_fp が指すストリームに使用されるバッファを定義し、関連付けます。_buf をヌルに設定すると、バッファが割り当てられます。_buf でバッファを指定すると、そのバッファがストリームに使用されます。_size はバッファのサイズを指定します。_type は、バッファリングのタイプを次のように指定します。

_IOFBF	完全なバッファリングが行われます。
_IOLBF	行バッファリングが行われます。
_IONBF	バッファリングは行われません。

sin　サイン

構文

#include　<math.h>

double **sin**(double x);

C++ の構文

#include <cmath>

double **std::sin**(double x);

定義される場所　rts.src の sin.c

説明

sin 関数は、浮動小数点数 x のサインを戻します。角度 x は、ラジアンで表します。引数の値が大き過ぎると、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;     /* sval is returned by sin */

radian = 3.1415927;
sval = sin(radian);      /* -1 is returned by sin  */
```

sinh　ハイパボリック・サイン

構文

#include　<math.h>

double **sinh**(double x);

C++ の構文

#include <cmath>

double **std::sinh**(double x);

定義される場所　rts.src の sinh.c

説明

sinh 関数は、浮動小数点数 x のハイパボリック・サインを戻します。引数の値が大きすぎると、範囲エラーになります。

例

```
double x, y;

x = 0.0;
y = sinh(x);      /* return value = 0.0 */
```

snprintf　上限付きのストリームの書き込み

構文

#include　<stdio.h>

int **snprintf**(char _string, size_t n, const char *_format, ...);

C++ の構文

#include <cstdio>

int **std::snprintf**(char _string, size_t n, const char *_format, ...);

定義される場所　rts.src の snprintf.c

説明

snprintf 関数は、最高で n 個の文字を _string が指す配列へ書き込みます。_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。文字列に上限が設定されていない場合に書き込まれたであろう文字数を返します。

sprintf　ストリームの書き込み

構文

#include　<stdio.h>

int **sprintf**(char _string, const char *_format, ...);

C++ の構文

#include <cstdio>

int **std::sprintf**(char _string, const char *_format, ...);

定義される場所　rts.src の sprintf.c

説明

sprintf 関数は、_string が指す配列への書き込みを行います。_format が指す文字列には、ストリームを書き込む方法が記述されています。

sqrt　平方根

構文

#include　<math.h>

double **sqrt**(double x);

C++ の構文

#include <cmath>

double **std::sqrt**(double x);

定義される場所　rts.src の sqrt.c

説明

sqrt 関数は、実数 x の非負数の平方根を戻します。この引数が負の場合は領域エラーになります。

例

```
double x, y;

x = 100.0;
y = sqrt(x);      /* return value = 10.0 */
```

srand

7-75 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

構文

#include <stdio.h>

int **sscanf**(const char *str, const char *format, ...);

C++ の構文

#include <cstdio>

int **std::sscanf**(const char *str, const char *format, ...);

定義される場所 rts.src の sscanf.c

説明 sscanf 関数 は、str が指す文字列からの読み込みを行います。_format が指す文字列は、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

構文

#include <string.h>

char ***strcat**(char *string1, char *string2);

C++ の構文

#include <cstring>

char ***std::strcat**(char *string1, char *string2);

定義される場所 rts.src の strcat.c

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。

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　文字の最初の出現を検出

構文

#include <string.h>

char ***strchr**(const char *string, int c);

C++ の構文

#include <cstring>

char ***std::strchr**(const char *string, int c);

定義される場所

rts.src の strchr.c

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

構文

#include　<string.h>

int **strcmp**(const char *string1, const char *string2);
int **strcoll**(const char *string1, const char *string2);

C++ の構文

#include <cstring>

int **std::strcmp**(const char *string1, const char *string2);
int **std::strcoll**(const char *string1, const char *string2);

定義される場所

rts.src の strcmp.c

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 will be executed    */
   }
if (strcoll(stra, strc) == 0)
   {
      /* statements here will be executed also */
   }
```

strcpy　文字列のコピー

構文

#include <string.h>

char ***strcpy**(char *dest, const char *src);

C++ の構文

#include <cstring>

char ***std::strcpy**(char *dest, const char *src);

定義される場所

rts.src の strcpy.c

説明

strcpy 関数は、s2（終了ヌル文字を含む）を s1 にコピーします。オーバーラップする文字列をコピーした場合の関数の動作は予測できません。この関数は s1 へのポインタを戻します。

例

次の例で、*a および *b が指す文字列は、2 つの独立した別々のメモリの位置です。コメントの中の \0 の表記は、ヌル文字を表します。

```
char a [] = "The quick black fox";
char b []= " jumps over ";

/* a --> "The quick black fox\0"         */
/* b --> " jumps over \0"                */

strcpy(a,b);

/* a --> " jumps over \0"                */
/* b --> " jumps over \0"                */
```

strcspn　不一致文字数の検出

構文

#include <string.h>

size_t **strcspn**(const char *string, const char *chs);

C++ の構文

#include <cstring>

size_t **std::strcspn**(const char *string, const char *chs);

定義される場所

rts.src の strcspn.c

説明

strcspn 関数は、全体が chs 内にない文字から構成される string の先頭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

構文

```
#include   <string.h>
char *strerror(int errno);
```

C++ の構文

```
#include <cstring>
char *std::strerror(int errno);
```

定義される場所　rts.src の strerror.c

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　時間の書式化

構文

```
#include   <time.h>
size_t *strftime(char *s, size_t maxsize, const char *format,
        const struct tm *timeptr);
```

C++ の構文

```
#include <ctime>
size_t *std::strftime(char *s, size_t maxsize, const char *format,
        const struct tm *timeptr);
```

定義される場所　rts.src の strftime.c

説明　strftime 関数は format 文字列に基づいて（timeptr が指す）時間を書式化し、書式化された時間を文字列 s で戻します。s には、最高 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（0 ～ 31）で表した日付
%H	10 進数（00 ～ 23）で表した時刻（24 時間制）

文字	展開内容
%I	10 進数（01 ～ 12）で表した時刻（12 時間制）
%j	10 進数（001 ～ 366）で表した日付
%m	10 進数（01 ～ 12）で表した月
%M	10 進数（00 ～ 59）で表した分
%p	各地域ごとの午前や午後の呼び方
%S	10 進数（00 ～ 50）で表した秒
%U	10 進数（00 ～ 52）で表したその年の週番号（週の初日は日曜日）
%x	日付の表記
%X	時間の表記
%y	10 進数（00 ～ 99）で表した年の下 2 桁
%Y	10 進数で表した年
%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 ヘッダで宣言と定義が行われる関数と型の詳細は、7.3.17 項「時間関数（time.h/ctime）」（7-27 ページ）を参照してください。

strlen　文字列の長さの検出

構文

```
#include   <string.h>
size_t strlen(const char *string);
```

C++ の構文

```
#include <cstring>
size_t std::strlen(const char *string);
```

定義される場所　rts.src の strlen.c

説明　strlen 関数は文字列の長さを戻します。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

構文

#include　<string.h>

char ***strncat**(char *dest, const char *src, size_t n);

C++ の構文

#include <cstring>

char ***std::strncat**(char *dest, const char *src, size_t n);

定義される場所　rts.src の strncat.c

説明

strncat 関数は、s2 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

構文

#include <string.h>

int **strncmp**(const char *string1, const char *string2, size_t n);

C++ の構文

#include <cstring>

int **std::strncmp**(const char *string1, const char *string2, size_t n);

定義される場所

rts.src の strncmp.c

説明

strncmp 関数は、s2 の最大 n 個の文字を s1 と比較します。この関数は、以下の値のどれかを戻します。

< 0	*string1 が *string2 より小さいとき。
0	*string1 が *string2 と等しいとき。
> 0	*string1 が *string2 より大きいとき。

例

```
char *stra = "why ask why";
char *strb = "just do it";
char *strc = "why not?";
size_t size = 4;

if (strcmp(stra, strb, size) > 0)
   {
      /* statements here will get executed */
   }
if (strcomp(stra, strc, size) == 0)
   {
      /* statements here will get executed also */
   }
```

strncpy　文字列のコピー

構文

#include <string.h>

char ***strncpy**(char *dest, const char *src, size_t n);

C++ の構文

#include <cstring>

char ***std::strncpy**(char *dest, const char *src, size_t n);

定義される場所

rts.src 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 []= "she's the one mother warned you of";
char strb []= " he's";
char strc []= "I am the one father warned you of";
char strd []= "oops";
int length = 5;

strncpy (stra,strb,length);

/* stra--> " he's the one mother warned you of\0" */;
/* strb--> " he's\0"                              */;
/* strc--> "I am the one father warned you of\0"  */;
/* strd--> "oops\0"                               */;

strncpy (stra,strc,length);

/* stra--> "I am  the one mother warned you of\0" */;
/* strb--> " he's\0"                              */;
/* strc--> "I am the one father warned you of\0"  */;
/* strd--> "oops\0"                               */;

strncpy (stra,strd,length);

/* stra--> "oops\0"                               */;
/* strb--> " he's\0"                              */;
/* strc--> "I am the one father warned you of\0"  */;
/* strd--> "oops\0"                               */;
```

strpbrk　一致する文字の検出

構文

#include　<string.h>
char ***strpbrk**(const char *string, const char *chs);

C++ の構文

#include <cstring>
char ***std::strpbrk**(const char *string, const char *chs);

定義される場所

rts.src の strpbrk.c

説明

strpbrk 関数は、string の中で、chs のいずれかの文字が最初に現れる位置を検索します。一致する文字を検出すると、strpbrk はその文字を指すポインタを戻します。その文字が存在しない場合は、ヌル・ポインタ（0）を戻します。

例

```
char *stra = "it wasn't me";
char *strb = "wave";
char *a;

a = strpbrk (stra,strb);
```

この例の後では、*a は was't の中の w を指します。

strrchr 文字の最後の出現を検出

構文

#include <string.h>

char ***strrchr**(const char *string, int c);

C++ の構文

#include <cstring>

char ***std::strrchr**(const char *string, int c);

定義される場所 rts.src の strrchr.c

説明

strchr 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

構文

#include <string.h>

size_t ***strspn**(const char *string, const char *chs);

C++ の構文

#include <cstring>

size_t ***std::strspn**(const char *string, const char *chs);

定義される場所 rts.src の strspn.c

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

構文

#include <string.h>
char ***strstr**(const char *string1, const char *string2);

C++ の構文

#include <cstring>
char ***std::strstr**(const char *string1, const char *string2);

定義される場所

rts.src の strstr.c

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/strtolstrtoul/strtoll/strtoull　文字列から数値への変換

構文

#include <stdlib.h>

double **strtod**(const char *st, char **endptr);
long **strtol**(const char *st, char **endptr, int base);
unsigned long **strtoul**(const char *st, char **endptr, int base);
long long **strtoll**(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);
unsigned long **std::strtoul**(const char *st, char **endptr, int base);
long long **std::strtoll**(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 の strtod.c, strtol.c, strtoul.c

説明

上記の 3 つの関数は、ASCII 文字列を数値に変換します。それぞれの関数の引数 st は元の文字列を指します。引数 endptr はポインタを指します。これらの関数は、変換された文字列の後にある最初の文字を指すようにこのポインタを設定します。整数への変換を行う関数には、3 番目の引数 base もあります。この引数は、どの底で文字列を変換するかを関数に指示します。

- ❏ strtod 関数は、文字列を浮動小数点値に変換します。文字列の形式は次のとおりです。

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

  この関数は、変換後の文字列を戻します。元の文字列が空のときや、その形式が正しくないときは、この関数は 0 を戻します。変換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、文字列を符号なし倍長整数に変換します。文字列の形式は次のとおりです。

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

space は、水平タブか垂直タブ、スペースバー、復帰、書式送り、改行を組み合わせて示します。space の後は、オプションの sign、さらに数値の整数部を示す digits が続きます。その後は数値の小数部が続き、オプションの sign をもつ指数部が続きます。

認識できない文字が初めて出現した位置で、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　文字列からトークンへのブレーク

構文

```
#include   <string.h>
char *strtok(char *str1, const char *str2);
```

C++ の構文

```
#include <cstdio>
char *std::strtok(char *str1, const char *str2);
```

定義される場所　rts.src 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

構文

#include <string.h>

size_t **strxfrm**(char *to, const char *from, size_t n);

C++ の構文

#include <cstring>

size_t **std::strxfrm**(char *to, const char *from, size_t n);

説明

strxfrm 関数は from が指す n 個の文字を to が指す n 個の文字に変換します。

tan タンジェント

構文

#include <math.h>

double **tan**(double x);

C++ の構文

#include <cmath>

double **std::tan**(double x);

定義される場所

rts.src の tan.c

説明

tan 関数は、浮動小数点数 x のタンジェントを戻します。x は、ラジアンで表した角度です。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);              /* return value = 1.0 */
```

tanh　ハイパボリック・タンジェント

構文

#include　<math.h>

double **tanh**(double x);

C++ の構文

#include <cmath>

double **std::tanh**(double x);

定義される場所　rts.src の tanh.c

説明　tanh 関数は、浮動小数点数 x のハイパボリック・タンジェントを戻します。

例

```
double x, y;

x = 0.0;
y = tanh(x);              /* return value = 0.0 */
```

time　時間

構文

#include　<time.h>
time_t **time**(time_t *timer);

C++ の構文

#include <ctime>
time_t **std::time**(time_t *timer);

定義される場所　rts.src の time.c

説明　time 関数は、秒数で表される現在のカレンダ時を判定します。カレンダ時を使用できないとき、この関数は -1 を戻します。timer がヌル・ポインタでない場合、この関数は timer が指すオブジェクトへの戻り値の代入も行います。

time.h ヘッダで宣言と定義が行われる関数と型の詳細は、7.3.17 項「時間関数（time.h/ctime）」（7-27 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固有

time 関数はターゲットシステムによって異なるため、ユーザ固有の time 関数を記述する必要があります。

tmpfile 一時ファイルの作成

構文

```
#include    <stdio.h>
FILE *tmpfile(void);
```

C++ の構文

```
#include <cstdio>
FILE *std::tmpfile(void);
```

定義される場所 rts.src の tmpfile.c

説明 tmpfile 関数は、一時ファイルを作成します。

tmpnam 有効なファイル名の生成

構文

```
#include    <stdio.h>
char *tmpnam(char *_s);
```

C++ の構文

```
#include <cstdio>
char *std::tmpnam(char *_s);
```

定義される場所 rts.src の tmpnam.c

説明 tmpnam 関数は、有効なファイル名である文字列を生成します。

toascii ASCII への変換

構文

```
#include    <ctype.h>
int toascii(int c);
```

C++ の構文

```
#include <cctype>
int toascii(int c);
```

定義される場所 rts.src 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

構文

```
#include   <ctype.h>

int tolower(int c);
int toupper(int c);
```

C++ の構文

```
#include <cctype>

int std::tolower(int c);
int std::toupper(int c);
```

定義される場所 rts.src の tolower.c 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 が既に小文字の場合、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 が既に大文字の場合、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。

ungetc ストリームへの文字の書き込み

構文

```
#include   <stdio.h>

int ungetc(int c, FILE *_fp);
```

C++ の構文

```
#include <cstdio>

int std::ungetc(int c, FILE *_fp);
```

定義される場所 rts.src の ungetc.c

説明 ungetc 関数は、_fp が指すストリームに文字 c を書き込みます。

va_arg/va_end/va_start　可変引数マクロ

構文

```
#include   <stdarg.h>

typedef  char *va_list;
         type va_arg(va_list, _type);
         void va_end(va_list);
         void va_start(va_list, parmN);
```

C++ の構文

```
#include <cstdarg>

typedef  char *std::va_list;
         type std::va_arg(va_list, _type);
         void std::va_end(va_list);
         void std::va_start(va_list, parmN);
```

定義される場所　stdarg.h/cstdarg

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。このような関数は可変引数関数といいますが、以下のマクロを使用して、その引数リストを実行時に 1 つずつ処理することができます。_ap パラメータは可変引数リスト内の引数を指します。

- va_start マクロは、可変引数関数の引数リスト内の先頭の引数を指すように _ap を初期化します。parmN パラメータは、宣言された固定リスト内の右端のパラメータを指します。
- va_arg マクロは、可変引数関数に対する呼び出しで次の引数の値を戻します。va_arg 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

構文

#include　<stdio.h>

int **vfprintf**(FILE *_fp, const char *_format, char *_ap);

C++ の構文

#include <cstdio>

int **std::vfprintf**(FILE *_fp, const char *_format, char *_ap);

定義される場所　rts.src の vfprintf.c

説明　vfprintf 関数 は、_fp が指すストリームへの書き込みを行います。format が指す文字列には、ストリームを書き込む方法が記述されています。引数リストは _ap で指定します。

vprintf　標準出力への書き込み

構文

#include　<stdio.h>

int **vprintf**(const char *_format, char *_ap);

C++ の構文

#include <cstdio>

int **std::vprintf**(const char *_format, char *_ap);

定義される場所　rts.src の vprintf.c

説明　vprintf 関数は、標準出力デバイスへの書き込みを行います。_format が指す文字列には、ストリームを書き込む方法が記述されています。引数リストは _ap で指定します。

vsnprintf　上限付きのストリームの書き込み

構文

#include　<stdio.h>

int **vsnprintf**(char *_string, size_t n, const char *_format, char *_ap);

C++ の構文

#include <cstdio>

int **std::vsnprintf**(char *string, size_t n, const char *_format, char *_ap);

定義される場所　rts.src の vsnprintf.c

説明

vsnprintf 関数は、最高で n 個の文字を _string が指す配列へ書き込みます。_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。文字列に上限が設定されていない場合に書き込まれたであろう文字数を返します。

vsprintf　ストリームの書き込み

構文

#include　<stdio.h>

int **vsprintf**(char *string, const char *_format, char *_ap);

C++ の構文

#include <cstdio>

int **std::vsprintf**(char *_string, const char *_format, char *_ap);

定義される場所　rts.src の vsprintf.c

説明

vsprintf 関数 は、string が指す配列への書き込みを行います。_format が指す文字列には、ストリームを書き込む方法が記述されています。引数リストは _ap で指定します。

第 8 章

ライブラリ作成ユーティリティ

TMS320C55x™ 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 mk55 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。mk55 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、TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

項目 **ページ**

8.1 ライブラリ作成ユーティリティの起動方法 8-2
8.2 ライブラリ作成ユーティリティのオプション 8-3
8.3 オプションのまとめ 8-4

8.1 ライブラリ作成ユーティリティの起動方法

ライブラリ作成ユーティリティを起動する構文は、以下のとおりです。

mk55 [*options*]*src_arch1* [**-l***obj.lib1*][*src_arch2* [**-l***obj.lib2*]]...

mk55 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、8.2 節および 8.3 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。mk55 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、mk55 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。

mk55 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 C55X_C_OPTION、C_OPTION、C55X_C_DIR、および C_DIR は無視されます。

8.2 ライブラリ作成ユーティリティのオプション

コマンド行のオプションのほとんどは、コンパイラ、アセンブラ、リンカ、およびシェルが使用する同じ名前のオプションに直接対応しています。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完了後は、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）。

--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）。

8.3 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 8–1 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 8–1. オプションとその機能のまとめ

(a) コンパイラ／シェルを制御するオプション

オプション	機能	ページ
-g または --symdebug:dwarf	DWARF2 デバッグ・フォーマットを使ったシンボリック・デバッグを有効にします。	2-26、3-13
--profile:breakpt	ブレイクポイントのプロファイルに影響する最適化を無効にします。	2-26
--profile:power	累乗プロファイルの実行を有効にします。	2-26
--symdebug:coff	代替 STABS デバッグ・フォーマットを使ったシンボリック・デバッグを有効にします。	2-34、3-13
-k	.asm ファイルを保持します。	2-21
-U*name*	name を未定義にします。	2-24
-v*device*[:*revision*]	コンパイラが、C55x に最適なコードおよび任意に指定された改訂番号を生成できるようにします。	2-21

(b) パーサを制御するオプション

オプション	機能	ページ
-pc	マルチバイト文字をサポートします。	--
-pe	組み込み C++ モードを有効にします。	5-38
-pi	定義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）。	2-50
-pk	コードを K&R 互換コードにします。	5-36
-pn	組み込み関数を無効にします。	--
-pr	緩和モードを可能にします。厳密な ANSI/ISO 違反を無視します。	5-36
-ps	厳密な ISO モード（C に対してであり、K&R C に対してではない）を可能にします。	5-36

表 8–1. オプションとその機能のまとめ（続き）

（c）診断を制御するオプション

オプション	機能	ページ
-pdr	注釈（軽い警告）を発行します。	2-43
-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。	2-40
-pdw	警告診断を抑止します（エラーは発行されます）。	2-40

（d）最適化レベルを制御するオプション

オプション	機能	ページ
-O0	レジスタ最適化によってコンパイルします。	3-2
-O1	-O0 最適化とローカル最適化によってコンパイルを実行します。	3-2
-O2（または -O）	-O1 最適化とグローバル最適化によってコンパイルを実行します。	3-2
-O3	-O2 による最適化を行い、さらにファイルに対して最適化を行います。mk55 によって -oI0 と -op0 が自動的に設定されることに注意してください。	3-2

（e）ターゲット固有のオプション

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。	3-10
-mb	すべてのデータ・メモリがオンチップにあるよう指定します。	2-18
-mc	通常 .const セクションに配置される定数を、読み取り専用の初期化された静的変数として処理できるようにします。	2-18
-mg	代数アセンブリ・ファイルを前提とします。	2-18
-ml	ラージ・メモリ・モデルを使用します。	6-3
-mn	-g が無効化した最適化オプションを有効にします。	3-13
-mo	ファイル内の関数ごとのコードを、.clink 疑似命令でマークされた独立したサブセクションに配置します。	2-19
-mr	コンパイラがハードウェアのブロックリピート、ローカルリピート、およびリピート命令を生成するのを防ぎます。-O2 または -O3 も指定されたときのみ有効です。	2-19
-ms	最小限のコード空間を最適化します。	2-19

表 8–1. オプションとその機能のまとめ（続き）

（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	シンボル・テーブルにラベルを格納します。	2-28
-ata	アセンブラに対し、このソース・ファイルの実行中に、ARMS ステータス・ビットが有効になることを前提としていると表明します。	2-28
-atb	アセンブラに、パラレル・バス・コンフリクト・エラーを警告として処理させます。	2-28
-atc	アセンブラに対し、このソース・ファイルの実行中に、CPL ステータス・ビットが有効になることを前提としていると表明します。	2-28
-ath	アセンブラに、サイズよりスピードの C54x 命令をエンコードさせます。	2-28
-atl	アセンブラに対し、このソース・ファイルの実行中に、C54CM ステータス・ビットが有効になることを前提としていると表明します。	2-28
-atn	アセンブラに C54x の遅延した分岐または呼び出し命令の遅延スロットに位置する NOP を削除させます。	2-28
-atp	アセンブリ命令プロファイル・ファイル（.prf）を生成します。	2-28
-ats	(ニーモニック・アセンブリのみ)。リテラル・シフト・カウント上の “#” を任意選択にします。	2-28
-att	アセンブラに対し、このソース・ファイルの実行中に、SST ステータス・ビットが無効になると通知します。	2-28
-atv	アセンブラに、特定の可変長命令の最大形式を使用させます。	2-28
-atw	(代数アセンブラのみ)。アセンブラ警告メッセージを抑止します。	2-29

（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[.]*newextension*	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。	2-25
-eo[.]*newextension*	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。	2-25

第 9 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**

9.1 C++ ネーム・デマングラの起動方法 9-2
9.2 C++ ネーム・デマングラのオプション 9-2
9.3 C++ ネーム・デマングラの使用例 9-3

9.1 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。

dem55 [*options*][*filenames*]

dem55 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、コマンド行またはリンカ・コマンド・ファイルの任意の場所に指定できます（オプションについては、9.2 節「C++ ネーム・デマングラのオプション」（9-2 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、dem55 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。

9.2 C++ ネーム・デマングラのオプション

C++ ネーム・デマングラを制御するオプションと、その機能についての説明を以下に示します。

-h C++ ネーム・デマングラ・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）。

9.3 C++ ネーム・デマングラの使用例

例 9–1 は、C++ プログラム例と、その結果 TMS320C55x™ コンパイラによって出力されるアセンブリ・コードを示しています。例 9–1（a）では、すべての関数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 9–1. 名前のマングリング

（a）C++ プログラム

```
class banana {
public:
    int calories(void);
    banana();
    ~banana();
};

int calories_in_a_banana(void)
{
    banana x;
    return x.calories();
}
```

（b）calories_in_a_banana へのアセンブリ出力

```
_calories_in_a_banana__Fv:
            AADD #-3, SP
            MOV SP, AR0
            AMAR *AR0+
            CALL #___ct__6bananaFv
            MOV SP, AR0
            AMAR *AR0+
            CALL #_calories__6bananaFv
            MOV SP, AR0
            MOV T0, *SP(#0)
            MOV #2,T0
            AMAR *AR0+
            CALL #___dt__6bananaFv
            MOV *SP(#0), T0
            AADD #3, SP
            RET
```

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。

```
% dem55 banana.asm
```

例 9–2 は、その結果を示しています。関数のリンク名がデマングルされていることに注目してください。

例 9–2. C++ ネーム・デマングラ実行後の結果

```
_calories_in_a_banana():
            AADD #-3, SP
            MOV SP, AR0
            AMAR *AR0+
            CALL #banana::banana()
            MOV SP, AR0
            AMAR *AR0+
            CALL #banana::_calories()
            MOV SP, AR0
            MOV T0, *SP(#0)
            MOV #2,T0
            AMAR *AR0+
            CALL #banana::~banana()
            MOV *SP(#0), T0
            AADD #3, SP
            RET
```

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。業界の自主的規格を設定する組織。

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 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ためのソフトウェア・プログラム。

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。

I

ISO： 国際標準化機構。国家規格の国際的組織で、業界の自主的規格を設定します。

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 ISO に対応しない C コンパイラ用に記述された K&R C のプログラムの大部分は、修正なしで正しくコンパイルされ、実行されます。

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。アーカイバを使うと、新しいメンバの追加だけでなく、アーカイブ・ライブラリの中のファイルを削除、抽出、置換することができます。

アーカイブ・ライブラリ： 単一のファイル内にグループ化されたファイルの集合。

アセンブラ： アセンブリ言語命令、疑似命令、およびマクロ疑似命令を含んだソース・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。

い

インターリスト： 元の C ソース文をアセンブラから出されるアセンブリ言語出力にコメントとして挿入する機能。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。

エイリアス指定： エイリアス指定は、単一のオブジェクトに複数の方法でアクセスできる場合、たとえば、2 つのポインタが単一のオブジェクトを指す場合などに実行されます。エイリアス指定を実行すると、間接参照によって別のオブジェクトが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。

エミュレータ： TMS320C55x™ ハードウェア上でソフトウェアを直接テストするために使われる開発システム。

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・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。

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。

間接呼び出し： ある関数が、呼び出された関数のアドレスを受け取って別の関数を呼び出す関数呼び出し。

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、最適化プログラムは周囲のコードとの文脈で関数の最適化を実行できます。

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）： セクションの考え方をサポートすることにより、モジュラー・プログラミングを拡張するバイナリ・オブジェクト・ファイル・フォーマット。すべての COFF セクションは、メモリ空間に個別に再配置可能です。すべてのセクションは、ターゲット・メモリに割り当てられたどのブロックにも配置できます。

く

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

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。

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。

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

シェル・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。シェルは、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。

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。

シミュレータ： C55x ハードウェアを使わずに、ワークステーション上でソフトウェアをテストするために使われる開発システム。

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することのできる、リンクされた実行可能オブジェクト・ファイル。

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・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 つのプログラに制限されている変数の一種。静的変数の値は、その関数またはプログラムが終了しても破棄されず、それらの関数またはプログラムが再び開始すると、前の値が再び使用されます。

セクション： コードまたはデータの再配置可能なブロックで、最終的にはメモリ・マップ内の連続した空間に入れられます。

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 コードまたはアセンブリ言語コードを含んだファイル。コードをコンパイルまたはアセンブルすることによって、オブジェクト・ファイルを作成します。

た

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： C プリプロセッサにはパーサが組み込まれているので、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。

は

バイト： 1 文字を含めることのできる記憶域の最小アドレス可能単位。

パーサ： ソース・ファイルを読み取り、前処理機能を実行し、構文をチックし、オプティマイザやコード・ジェネレータの入力として使用できる中間ファイルを生成するソフトウェア・ツール。

ひ

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してください。

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つ）。

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に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、単体として処理されるた一連の文。

へ

変数： 一連の値のうちのどれかを想定する、ある量を表すシンボル。

ま

マクロ： 命令として使用できるユーザ定義ルーチン。

マクロ呼び出し： マクロを起動すること。

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」を参照してください。

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・ライブラリ： ランタイムサポート関数のソースが納められている、ライブラリ・ファイル rts.src。

ランタイムサポート関数： C 言語には含まれない作業（メモリの割り当て、文字列の変換、文字列の検索など）を実行する ISO 標準関数。

ランタイム（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してください。

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、ユーザが -cr 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。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

>> 記号　2-41
-@ コンパイラ・オプション　2-17

数

2 次システム・スタック　6-6
2 次スタック・ポインタ　6-6
3 文字符号
　系列　2-35
　　定義　A-1

A

-a リンカ・オプション　4-5
-aa アセンブラ・オプション　2-27
abort 関数　7-39
abs 関数　7-39
-abs コンパイラ・オプション　2-17
-abs リンカ・オプション　4-5
-ac アセンブラ・オプション　2-27
acos 関数　7-40
-ad アセンブラ・オプション　2-27
-ahc アセンブラ・オプション　2-27
-ahc アセンブラ・オプションを使用したファイルのコピー　2-27
-al アセンブラ・オプション　2-27
ANSI　A-1
ANSI C
　言語　5-1, 7-84, 7-86
-apd アセンブラ・オプション　2-27
-api アセンブラ・オプション　2-27
-ar アセンブラ・オプション　2-27
-ar リンカ・オプション　4-5
--args リンカ・オプション　4-5
ARMS モード
　-ata アセンブラ・オプション　2-28
-as アセンブラ・オプション　2-28
ASCII 変換関数　7-43
asctime 関数　7-40, 7-48
asin 関数　7-41
.asm 拡張子　2-24
　変更　2-25
asm 文　6-27
　C 言語　5-16
　最適化されたコードの中で　3-9
assert 関数　7-41
assert.h ヘッダ　7-17
　関数のまとめ　7-30
-ata アセンブラ・オプション　2-28
atan 関数　7-42
atan2 関数　7-42
-atb アセンブラ・オプション　2-28
-atc アセンブラ・オプション　2-28
atexit 関数　7-43, 7-50
-ath アセンブラ・オプション　2-28
-atl アセンブラ・オプション　2-28
-atn アセンブラ・オプション　2-28
atof 関数　7-43
atoi 関数　7-43
atol 関数　7-43
-atp アセンブラ・オプション　2-28
-ats アセンブラ・オプション　2-28
-att アセンブラ・オプション　2-28
-atv アセンブラ・オプション　2-29
-au アセンブラ・オプション　2-29
-aw アセンブラ・オプション　2-29
-ax アセンブラ・オプション　2-29

B

-b オプション
　コンパイラ　2-17
　リンカ　4-5
boot.asm　6-45

boot.obj 4-8, 4-10
bsearch 関数 7-44
.bss セクション 4-12, 6-5
　定義 A-1

C

-c オプション
　コンパイラ 2-17
　リンカ 4-2, 4-5, 4-10
.C 拡張子 2-24
.c 拡張子 2-24
C 言語
　C プログラミング言語 vi, 5-1, 5-35
　interrupt キーワード 5-12
　ioport キーワード 5-9
　ISO C との互換性 5-35
　onchip キーワード 5-12
　restrict キーワード 5-13
　アセンブラ定数のアクセス方法 6-26
　アセンブラ文の配置方法 6-27
　アセンブラ変数のアクセス方法 6-25
　アセンブリ言語とインターフェイスする 6-22
　アセンブリへ差し込む方法 2-49
　キーワード 5-8
　データ型 5-6
　特性 5-2
　レジスタ変数 5-15
　割り込みルーチン 6-39
　　レジスタの保存方法 6-38
C 言語における変数の初期化方法
　グローバル 5-33
　静的 5-33
C とアセンブリ言語をインターフェイスする 6-22
C 入出力
　実装 7-6
　低レベル・ルーチン 7-6
　ライブラリ 7-4
--c ライブラリ作成ユーティリティ・オプション 8-3
C++ 言語
　iostream 5-5
　組み込み C++ モード 5-37
　特性 5-5
　ランタイム型情報 5-5
　例外処理 5-5
C++ ネーム・デマングラ
　オプション 9-2
　起動方法 9-2
　説明 1-4, 1-7, 9-1
　例 9-3
C/C++ コードのコンパイル 2-2
C54x 互換モード
　-atl アセンブラ・オプション 2-28
C55x バイト 5-6
C55X_C_OPTION 環境変数 2-31, 2-32
-call オプション 2-17
calloc 関数 7-45, 7-58, 7-71
　動的メモリ割り当て 6-7
C_DIR 環境変数 2-31, 2-34
ceil 関数 7-45
.cinit セクション 4-10, 4-11, 6-5, 6-45
_c_int00 4-10, 6-45
.cio セクション 4-12, 6-5, 7-4
ciso646 ヘッダ 7-22
cl55 -z 4-2
clearerr 関数 7-46
clock 関数 7-46
CLOCKS_PER_SEC マクロ 7-27, 7-46
clock_t データ型 7-27
CLOSE 入出力関数 7-12
Code Composer Studio
　およびコード生成ツール 1-8
CODE_SECTION プラグマ 5-18
const 型修飾子 5-34
const キーワード 5-8
.const セクション 4-11, 6-5, 6-49
　変数の初期化に使用 5-34
　リード・オンリーの定数として使用 2-18
--consultant コンパイラ・オプション 2-18
C_OPTION 環境変数 2-31
cos 関数 7-47
cosh 関数 7-47
CPL モード
　-atc アセンブラ・オプション 2-28
.cpp 拡張子 2-24
-cr リンカ・オプション 4-2, 4-5, 4-10
csetjmp ヘッダ 7-22
ctime 関数 7-48
ctype.h ヘッダ 7-17
　関数のまとめ 7-30
.cxx 拡張子 2-24

D

-d コンパイラ・オプション 2-18
.data セクション 6-6
DATA_ALIGN プラグマ 5-21
DATA_SECTION プラグマ 5-22
_ _DATE_ _ 2-33
dem55 9-2
difftime 関数 7-48
div 関数 7-49

div_t 型　7-26
DWARF デバッグ・フォーマット　2-23

E

-e リンカ・オプション　4-5
-ea コンパイラ・オプション　2-25
-ec コンパイラ・オプション　2-25
EDOM マクロ　7-18
-eo コンパイラ・オプション　2-25
EOF クリア関数　7-46
EOF マクロ　7-25
EPROM プログラマ　1-3
ERANGE マクロ　7-18
errno.h ヘッダ　7-18
-es コンパイラ・オプション　2-25
ETSI 関数
　組み込み関数　6-35
exit 関数　7-39, 7-43, 7-50
exp 関数　7-50
extaddr.h ヘッダ　7-18

F

-f リンカ・オプション　4-5
-fa コンパイラ・オプション　2-24
fabs 関数　7-51, 7-52
FAR プラグマ　5-23, 6-41
　データ・オブジェクトへのアクセス方法　6-42
far_near_memcpy 関数　7-52
FARPTR integer 型　6-40
-fb コンパイラ・オプション　2-26
-fc コンパイラ・オプション　2-24
fclose 関数　7-53
feof 関数　7-53
ferror 関数　7-53
-ff コンパイラ・オプション　2-26
fflush 関数　7-54
-fg コンパイラ・オプション　2-25
fgetc 関数　7-54
fgetpos 関数　7-54
fgets 関数　7-55
_ _FILE_ _　2-33
file.h ヘッダ　7-18
FILENAME_MAX マクロ　7-25
float.h ヘッダ　7-19
floor 関数　7-55
fmod 関数　7-55
-fo コンパイラ・オプション　2-24
fopen 関数　7-56
FOPEN_MAX マクロ　7-25
-fp コンパイラ・オプション　2-24
FP レジスタ　6-12
fpos_t データ型　7-25
fprintf 関数　7-56
fputc 関数　7-56
fputs 関数　7-57
-fr コンパイラ・オプション　2-26
fread 関数　7-57
free 関数　7-58
freopen 関数
　説明　7-58
frexp 関数　7-59
-fs コンパイラ・オプション　2-26
fscanf 関数　7-59
fseek 関数　7-59
fsetpos 関数　7-60
-ft コンパイラ・オプション　2-26
ftell 関数　7-60
FUNC_CANNOT_INLINE プラグマ　5-24
FUNC_EXT_CALLED プラグマ　5-24
　-pm オプションとともにに使う　3-7
FUNC_IS_PURE プラグマ　5-25
FUNC_IS_SYSTEM プラグマ　5-26
FUNC_NEVER_RETURNS プラグマ　5-26
FUNC_NO_GLOBAL_ASG プラグマ　5-27
FUNC_NO_IND_ASG プラグマ　5-27
fwrite 関数　7-60

G

-g オプション
　コンパイラ　2-23
　シェル　2-23
　リンカ　4-5
getc 関数　7-61
getchar 関数　7-61
getenv 関数　7-61
gets 関数　7-62
gmtime 関数　7-62
-gp コンパイラ・オプション　2-30
-gt コンパイラ・オプション　2-30
-gw コンパイラ・オプション　2-30

H

-h オプション
C++ デマングラ・ユーティリティ 9-2
リンカ 4-5
--h ライブラリ作成ユーティリティ・オプション 8-3
-heap オプション 7-67
-heap リンカ・オプション 4-5
Hex 変換ユーティリティ 1-3
HUGE_VAL 7-22

I

-i オプション
コンパイラ 2-18
シェル 2-34, 2-35
リンカ 4-6
#include ファイル 2-18, 2-33, 2-34
include ファイル
検索されるディレクトリの追加 2-18
代替ディレクトリ 2-35
_INLINE 2-33
プリプロセッサ 2-46
inline キーワード 2-46
interrupt キーワード 6-39
INTERRUPT プラグマ 5-28
int_fastN_t 整数型 7-24
int_leastN_t 整数型 7-24
intmax_t 整数型 7-24
INTN_C マクロ 7-24
intN_t 整数型 7-24
intprt_t 整数型 7-24
ioport キーワード 5-9
iostream サポート 5-5
isalnum 関数 7-63
isalpha 関数 7-63
isascii 関数 7-63
iscntrl 関数 7-63
isdigit 関数 7-63
isgraph 関数 7-63
islower 関数 7-63
ISO A-1
ISO C
緩和モードの有効化 5-37
規格の概要 1-5
組み込み C++ モードの有効化 5-37
厳密モードの有効化 5-37
iso646.h ヘッダ 7-22
isprint 関数 7-63
ispunct 関数 7-63
isspace 関数 7-63
isupper 関数 7-63
isxdigit 関数 7-63
isxxx 関数 7-17, 7-63

J

-j リンカ・オプション 4-6

K

-k オプション
コンパイラ 2-18
リンカ 4-6
--k ライブラリ作成ユーティリティ・オプション 8-3
K&R C vi, 5-35
互換性 5-1, 5-35
定義 A-1

L

-l オプション
ライブラリ作成ユーティリティ 8-2
リンカ 4-6, 4-8
labs 関数 7-39
_ _LARGE_MODEL_ _ 2-33
ldexp 関数 7-64
ldiv 関数 7-49
ldiv_t 型 7-26
limits.h ヘッダ 7-19
_ _LINE_ _ 2-33
localtime 関数 7-48, 7-65, 7-69
log 関数 7-65
log10 関数 7-66
long long データ型
printf とともに使う 5-7
説明 5-7
longjmp 関数 7-78
LSEEK 入出力関数 7-12
L_tmpnam マクロ 7-25
ltoa 関数 7-66

M

-m リンカ・オプション 4-6
main から戻る 4-9
malloc 関数 7-58, 7-67, 7-71
　動的メモリ割り当て 6-7
math.h ヘッダ 7-22
　関数のまとめ 7-31
-mb コンパイラ・オプション 2-18
-mc コンパイラ・オプション 2-18
memchr 関数 7-67
memcmp 関数 7-68
memcpy 関数 7-68
memmove 関数 7-69
memset 関数 7-69
-mg コンパイラ・オプション 2-18
minit 関数 7-71
mk55 8-2
mktime 関数 7-69
-mo コンパイラ・オプション 2-19
modf 関数 7-71
-ms コンパイラ・オプション 2-19
MUST_ITERATE プラグマ 5-28

N

-n オプション 2-19
NDEBUG マクロ 7-17, 7-41
.nfo 拡張子 3-4
NULL マクロ 7-23, 7-25

O

-o オプション
　C++ デマングラ・ユーティリティ・オプション 9-2
　コンパイラ 3-2
　リンカ 4-6
.obj 拡張子 2-24
　変更 2-25
offsetof マクロ 7-23
-oi コンパイラ・オプション 3-11
-ol コンパイラ・オプション 3-3
-on コンパイラ・オプション 3-4
-op コンパイラ・オプション 3-5
OPEN 入出力関数 7-13

P

-pdel コンパイラ・オプション 2-39
-pden コンパイラ・オプション 2-39
-pdf コンパイラ・オプション 2-39
-pdr コンパイラ・オプション 2-39
-pds コンパイラ・オプション 2-39
-pdse コンパイラ・オプション 2-39
-pdsr コンパイラ・オプション 2-39
-pdsw コンパイラ・オプション 2-39
-pdv コンパイラ・オプション 2-40
-pdw コンパイラ・オプション 2-40
-pe コンパイラ・オプション 5-37
perror 関数 7-72
-pg コンパイラ・オプション 2-35
-pi コンパイラ・オプション 2-46
.pinit セクション 4-11, 6-5
-pk コンパイラ・オプション 5-35, 5-37
-pm コンパイラ・オプション 3-5
pow 関数 7-72
.pp ファイル 2-35
-ppa コンパイラ・オプション 2-36
-ppc コンパイラ・オプション 2-36
-ppd コンパイラ・オプション 2-36
-ppi コンパイラ・オプション 2-36
-ppl コンパイラ・オプション 2-36
-ppo コンパイラ・オプション 2-35
-pr コンパイラ・オプション 5-37
#pragma 疑似命令 5-4
printf 関数 7-72
-priority リンカ・オプション 4-6
--profile: breakpt コンパイラ・オプション 2-23
--profile: power コンパイラ・オプション 2-23
-ps コンパイラ・オプション 5-37
ptrdiff_t 型 5-2, 7-23
--purecirc アセンブラ・オプション 2-29
putc 関数 7-73
putchar 関数 7-73
puts 関数 7-73

Q

-q オプション
　コンパイラ 2-19
　リンカ 4-6
--q ライブラリ作成ユーティリティ・オプション 8-3
-qq オプション 2-19

qsort 関数　7-74

R

-r リンカ・オプション　4-6
rand 関数　7-75
RAND_MAX マクロ　7-26
READ 入出力関数　7-14
realloc 関数　6-7, 7-58, 7-71, 7-75
register 記憶クラス　5-3
remove 関数　7-76
rename 関数　7-76
RENAME 入出力関数　7-14
rewind 関数　7-77
rts.lib　7-2
rts.src　7-2, 7-26

S

-s オプション
　コンパイラ　2-20
　シェル　2-49
　リンカ　4-6
.s 拡張子　2-24
scanf 関数　7-77
.sect 疑似命令　6-38
setbuf 関数　7-77
setjmp 関数　7-78
setjmp.h ヘッダ　7-22
　関数とマクロのまとめ　7-32
setvbuf 関数　7-79
sin 関数　7-80
sinh 関数　7-80
sizeof
　char への適用　5-6
size_t　5-2
　型　7-23
　データ型　7-25
snprintf 関数　7-81
sprintf 関数　7-81
sqrt 関数　7-81
srand 関数　7-75
-ss コンパイラ・オプション　2-20
sscanf 関数　7-82
SST モード
　-att アセンブラ・オプション　2-28
STABS デバッグ・フォーマット　2-30
-stack オプション　4-6
.stack セクション　4-12, 6-5
_ _STACK_SIZE 定数　6-7
stdarg.h ヘッダ　7-23
　マクロのまとめ　7-32
stddef.h ヘッダ　7-23
stdint.h ヘッダ　7-24
stdio.h ヘッダ　7-25
　関数のまとめ　7-33
stdlib.h ヘッダ　7-26
　関数のまとめ　7-35
strcat 関数　7-82
strchr 関数　7-83
strcmp 関数　7-84
strcoll 関数　7-84
strcpy 関数　7-85
strcspn 関数　7-85
strerror 関数　7-86
strftime 関数　7-86
string.h ヘッダ　7-27
　関数のまとめ　7-37
strlen 関数　7-87
strncat 関数　7-88
strncmp 関数　7-89
strncpy 関数　7-89
strpbrk 関数　7-90
strrchr 関数　7-91
strspn 関数　7-91
strstr 関数　7-92
strtod 関数　7-92
strtok 関数　7-93
strtol 関数　7-92
strtoul 関数　7-92
strxfrm 関数　7-94
STYP_CPY フラグ　4-11
.switch セクション　4-11, 6-5
--symdebug: coff コンパイラ・オプション　2-30
--symdebug: dwarf コンパイラ・オプション　2-23
--symdebug: none コンパイラ・オプション　2-23
--symdebug: profile_coff コンパイラ・オプション　2-30
--symdebug: skeletal コンパイラ・オプション　2-23
.sysmem セクション　6-5
sysmem セクション　4-12
_ _SYSMEM_SIZE　6-7
-sysstack オプション　4-6
.sysstack セクション　6-5
_ _SYSSTACK_SIZE 定数　6-7

T

tan 関数　7-94
tanh 関数　7-95
.text セクション　4-11, 6-5
_ _TI_COMPILER_VERSION_ _　2-33
_ _TIME_ _　2-33
time.h ヘッダ　7-27
　関数のまとめ　7-38
time_t データ型　7-27
tm 構造体　7-27
tmpfile 関数　7-96
TMP_MAX マクロ　7-25
tmpnam 関数　7-96
_ _TMS320C55X_ _　2-33
toascii 関数　7-96
tolower 関数　7-97
toupper 関数　7-97

U

-u オプション
　C++ デマングラ・ユーティリティ　9-2
　コンパイラ　2-20
　リンカ　4-6
--u ライブラリ作成ユーティリティ・オプション　8-3
uint_fastN_t 符号なし整数型　7-24
uint_leastN_t 符号なし整数型　7-24
uintmax_t 符号なし整数型　7-24
UINTN_C マクロ　7-24
uintN_t 符号なし整数型　7-24
uintprt_t 符号なし整数型　7-24
ungetc 関数　7-97
UNLINK 入出力関数　7-15
UNROLL プラグマ　5-30
unsigned
　定義　A-6

V

-v オプション
　C++ デマングラ・ユーティリティ　9-2
　コンパイラ　2-20, 2-21
--v ライブラリ作成ユーティリティ・オプション　8-3
va_arg 関数　7-98
va_end 関数　7-98
va_start 関数　7-98
-version コンパイラ・オプション　2-20
vfprintf 関数　7-99
vprintf 関数　7-99
vsnprintf 関数　7-100
vsprintf 関数　7-100

W

-w リンカ・オプション　4-6
WRITE 入出力関数　7-15

X

-x リンカ・オプション　4-7
--xml_link_info リンカ・オプション　4-7

Y

y/x の逆タンジェント　7-42

Z

-z コンパイラ・オプション　2-2, 2-4, 2-20, 4-2

あ

アーカイバ　1-3, A-2
アーカイブ・ライブラリ　4-8, A-2
アーク・コサイン　7-40
アーク・サイン　7-41
アーク・タンジェント　7-42
アセンブラ　1-3
　-ats アセンブラ・オプション　2-28
　オプション　2-27
　警告メッセージの抑止方法　2-29
　定義　A-2
　パイプライン・コンフリクト警告の有効化（-aw オプション）　2-29
アセンブリ言語
　C 言語とのインターフェイス方法　6-22
　C 言語の差し込み方法　2-49
　モジュール　6-22
アセンブリ言語における変数の定義方法　6-24

アセンブリ・コードのスピード
-ath アセンブラ・オプション 2-28
アセンブリ・ソースのデバッグ 2-23
アセンブリ・リスト・ファイル
作成方法 2-27

い

一時ファイル作成関数 7-96
インターリスト機能
オプティマイザとともに使用 3-12
起動方法 2-20
コンパイラを使った起動方法 2-49
定義 A-2
インターリスト・ユーティリティ
起動方法 2-20
インライン
アセンブリ言語 6-27
関数
定義 A-3
関数の宣言方法 2-46
定義制御の 2-46
展開 2-45
無効化 2-46
インライン展開
自動展開 3-11
制約事項 2-48
保護されない定義制御の 2-46

え

エイリアス指定 3-10
定義 A-2
エイリアスの明確化
説明 3-17
定義 A-2
エスケープ・シーケンス 5-2, 5-36
エピローグ
定義 A-2
エミュレータ
定義 A-2
エラー
標識関数 7-46
プリプロセッサからのメッセージ 2-33
マップ関数 7-72
メッセージ・マクロ 7-30
エラー報告 7-18
エラー・テスト関数 7-53
エラー・メッセージ
オプションの処理 2-40, 5-36
⇒診断メッセージも参照
エラー・メッセージ（続き）
マクロ
assert 7-41
エントリ・ポイント
C/C++ コードの 4-10
_c_int00 4-10
システム・リセット 6-38
定義 A-2
リセット・ベクトル 4-10

お

オーバーフロー
ランタイム・スタック 6-6
大文字と小文字の区別
ファイル名拡張子における 2-24
オブジェクト・ライブラリ 4-12
オブジェクトの格納関数 7-54
オブジェクト・ファイル
定義 A-2
オブジェクト・ライブラリ
定義 A-3
オプション
C++ ネーム・デマングラ 9-2
C55x 固有 2-12
アセンブラ 2-27
一般的な 2-17
規則 2-5
コンパイラ 2-6, 2-7, 2-8, 2-9, 2-11, 2-12, 2-13, 2-14, 2-15, 2-16
診断 2-11, 2-39
定義 A-3
ファイル指定子 2-25
プリプロセッサ 2-10
要約表 2-7, 2-8, 2-9, 2-11, 2-12, 2-13, 2-14, 2-15, 2-16
ライブラリ作成ユーティリティ 8-2
オプション 8-2
リンカ 4-5
オプティマイザ
オプションのまとめ 2-13
起動方法
コンパイラを使って 3-2
定義 A-3
オペランド
定義 A-3

か

カーネル
定義 A-3

外部シンボル
定義 A-3
外部宣言 5-36
外部変数 6-9
書き込み関数
fprintf 7-56
fputc 7-56
fputs 7-57
printf 7-72
putc 7-73
putchar 7-73
puts 7-73
snprintf 7-81
sprintf 7-81
ungetc 7-97
vfprintf 7-99
vprintf 7-99
vsnprintf 7-100
vsprintf 7-100
拡張アドレッシング 6-40
FARPTR の使用 6-40
コード例 6-43
の C 変数 6-41
の .cinit セクション 6-40
の .const セクション 6-43
の .switch セクション 6-40
拡張アドレッシング関数
far_near_memcpy 7-52
拡張子
C ソース・ファイル 2-24
C++ ソース・ファイル 2-24
nfo 3-4
アセンブリ言語ソース・ファイル 2-24
オブジェクト・ファイル 2-24
可変長命令
-atv アセンブラ・オプション 2-28
可変引数関数およびマクロ 7-23
va_arg 7-98
va_end 7-98
va_start 7-98
可変引数マクロ
のまとめ 7-32
仮定義 5-36
カレンダ時 7-27, 7-48, 7-69, 7-95
環境情報関数 7-61
環境変数
C55X_C_OPTION 2-31, 2-32
C_DIR 2-31
C_OPTION 2-31
定義 A-3
関数
アルファベット順の参照 7-39
インライン展開 2-45
定義 A-3
汎用ユーティリティ 7-35
関数（続き）
呼び出し
スタックの使用方法 6-6
ランタイムサポート A-7
関数のインライン展開 3-21
関数のプロトタイプ 5-35
関数の呼び出し
規則 6-16
間接呼び出し
定義 A-3
関連文献 vi

き

キーワード
C 言語キーワード 5-8
inline 2-46
interrupt 5-12
ioport 5-9
onchip 5-12
restrict 5-13
記憶クラス
定義 A-3
疑似命令
定義 A-3
疑似乱数 7-75
起動方法
C++ ネーム・デマングラ 9-2
インターリスト機能
シェルを使った 2-49
オプティマイザ
コンパイラ・オプションを使って 3-2
コンパイラ 2-4
ライブラリ作成ユーティリティ 8-2
リンカ
個別に 4-2
コンパイラを使って 4-2
共通オブジェクト・ファイル・フォーマット
定義 A-3
強度換算の最適化 3-22

く

組み込み C++ モード 5-37
組み込み演算子 2-45
組み込み関数
C/C++ コンパイラのための 6-30, 6-31, 6-32, 6-33
ETSI 関数 6-35
アセンブリ言語文を呼び出す 6-28

組み込み関数（続き）
新しい機能 6-30, 6-31, 6-32, 6-33
加法 6-30
極値 6-32
減算 6-30
算術演算 6-33
シフト 6-31
乗算 6-31
絶対値 6-30
端数処理 6-32
ビットカウント 6-32
符号反転 6-30
飽和 6-32
命令セットへのアクセス 6-30, 6-31, 6-32, 6-33
割り込み 6-39
グレゴリオ暦 7-27
グローバル
定義 A-4
グローバル変数 5-33, 6-9
確保された空間 6-4
グローバル変数の構造 4-10
クロスリファレンス・リスト
作成 2-29, 2-42
定義 A-3

け

警告メッセージ 2-37, 5-36
限界
コンパイラ 5-38
検索 7-44

こ

構造体
定義 A-4
構造体のパック 6-9
構造体のメンバ 5-3
コードを生成するプロセッサ・バージョン 2-20
コード・ジェネレータ
定義 A-4
互換性 5-35
コサイン 7-47
コストに基づいたレジスタ割り当ての最適化 3-17
コマンド・ファイル
コマンド行への追加 2-17
定義 A-4
コメント
定義 A-4
コンパイラ
C_OPTION 環境変数 2-31
アセンブラ・オプション 2-27
アセンブリ言語ファイルの保存 2-18
一般的なオプション 2-17
オプション
規則 2-5
シンボリック・デバッグ 2-7
非推奨 2-30
プロファイル 2-7
要約表 2-7, 2-8, 2-9, 2-11, 2-12, 2-13, 2-14, 2-15, 2-16
オプションのまとめ 2-5
オプティマイザ・オプション 2-13
概要 1-5, 2-2
起動方法 2-4
限界 5-38
コンパイラ・コンサルタント・アドバイス・ツール 2-18
コンパイルのみ 2-19
シリコン・リビジョンの指定方法 2-21
診断オプション 2-39
診断メッセージ 2-37
セクション 4-11
説明 2-1
定義 A-1
ディレクトリ指定子オプション 2-26
ハードウェアのブロック・リピート、ローカル・リピート、およびリピート命令を防ぐ 2-19
ファイル指定子オプション 2-24
補足ユーザ情報ファイルの生成 2-17
リンクの有効化 2-20
コンパイラにより生成されるリンク名 5-32
コンパイルのみ 2-19
コンパイル・メッセージの抑止 2-19
コンマを続ける
列挙型定数リスト 5-37

さ

サーキュラ・アドレッシング
--purecirc アセンブラ・オプション 2-29
最適化 3-2
TMS320C55x 固有の 3-23
インライン展開 3-21
エイリアスの明確化 3-17
強度換算 3-22
コストに基づいたレジスタ割り当て 3-17
式の簡略化 3-19
自動インクリメント・アドレッシング 3-22
情報ファイル・オプション 3-4
制御フローの簡略化 3-17
データ・フロー 3-19
のリスト 3-16

最適化（続き）
ファイルレベル 3-3, A-6
プログラムレベル
FUNC_EXT_CALLED プラグマ 5-24
説明 3-5
定義 A-7
分岐 3-17
誘導変数 3-22
ループの循環 3-22
ループ不変コードの移動 3-22
レベル 3-2
レベルの制御方法 3-5
最適化されたコード
デバッグ 3-14
最適化されたコードのプロファイル方法 3-15
再配置
定義 A-4
サイン 7-80
サブセクションに配置された関数 2-19
三角関数 7-22

し

シェル・プログラム
定義 A-4
時間関数 7-27
asctime 7-40
clock 7-46
ctime 7-48
difftime 7-48
gmtime 7-62
localtime 7-65
mktime 7-69
strftime 7-86
time 7-95
のまとめ 7-38
式 5-3
C 言語 5-3
簡略化 3-19
定義 A-4
識別子
C 言語 5-2
識別子を前に付ける方法
_ 6-23
指数関数 7-22, 7-50
システムの初期化 6-45
自動初期化 6-45
システムの制約
_ _STACK_SIZE 6-7
_SYSMEM_SIZE 6-7
_ _SYSSTACK_SIZE 6-7
システム・スタック 6-6
事前初期化された 5-33
自然対数 7-65
事前定義された名前 2-33
-ad アセンブラ・オプション 2-27
-au アセンブラ･オプションを使って未定義にする方法 2-29
_ _DATE_ _ 2-33
_ _FILE_ _ 2-33
_INLINE 2-33
_ _LARGE_MODEL_ _ 2-33
_ _LINE_ _ 2-33
_ _TI_COMPILER_VERSION_ _ 2-33
_ _TIME_ _ 2-33
_ _TMS320C55X_ _ 2-33
実行可能モジュール
定義 A-4
実装が定義する動作 5-2
自動インクリメント・アドレッシング 3-22
自動初期化 6-45, A-4
実行時に
定義 A-4
実行時の
説明 6-49
のタイプ 4-10
変数の 6-8
ロード時の
定義 A-8
シフト 5-3
ジャンプ関数 7-32
ジャンプ・マクロ 7-32
出力
ファイルの概要 1-6
出力セクション A-4
出力モジュール A-5
詳細時刻 7-27, 7-48, 7-69
小数部と指数部の関数 7-59
剰余 5-3
初期化
タイプ 4-10
変数の
ロード時の 6-8
ロード時の
説明 6-50
初期化されたセクション 4-11, 6-5
.cinit 6-5
.const 6-5
.pinit 6-5
.switch 6-5
.text 6-5
定義 A-5
初期化されないセクション 4-11, 6-5
定義 A-5
除算 5-3
シリコン・リビジョン
シェル内の指定方法 2-21

診断識別子
ロー・リスト・ファイルの 2-43
診断メッセージ 7-17
assert 7-41
エラー 2-37
警告 2-37
制御方法 2-39
生成方法 2-39
説明 2-37
その他のメッセージ 2-41
致命的エラー 2-37
注釈 2-37
フォーマット 2-37
抑止方法 2-39
診断メッセージの制御方法 2-39
シンボリック
デバッグ
疑似命令の生成方法 2-23
シンボリック・デバッグ
DWARF 疑似命令 2-23
STABS フォーマットの使用 2-30
クロスリファレンス 2-29
最小限（デフォルト） 2-23
定義 A-5
抑止 2-23
シンボル A-5
アセンブラ定義 2-27
アセンブラ定義シンボルを未定義にする方法 2-29
テーブル
定義 A-5

す

スタック 6-6
オーバーフロー
ランタイム・スタック 6-6
確保された空間 6-5
スタック管理 6-6
スタック・ポインタ 6-6
スタンドアロン・プリプロセッサ
定義 A-5
ステータス・レジスタ
コンパイラによる使用 6-14
スピードではなく、スペースを最適化 2-19

せ

制御フローの簡略化 3-17
制限値
コンパイラ 5-38
整数型 7-19
浮動小数点型 7-19
整数の除算 7-49
生成方法
シンボリック・デバッグ疑似命令 2-23
静的
定義 A-5
変数
確保された空間 6-5
静的変数 5-33, 6-9
セクション
.bss 6-5
.cinit 6-6, 6-45
.cio 6-5
.const 5-34
.data 6-6
.stack 6-5
.sysmem 6-5
.sysstack 6-5
.text 6-6
出力 A-4
初期化されない A-5
説明 4-11
定義 A-5
ヘッダ A-5
絶対コンパイラの限界 5-38
絶対値 7-39, 7-51, 7-52
絶対リスタ 1-4
絶対リスト
作成方法 2-27
絶対リスト・ファイルの作成 2-17
宣言
C 言語 5-3
専用のレジスタ 6-14

そ

ソース・ファイル
拡張子 2-24
定義 A-5
ソート 7-74
ソフトウェア開発ツール 1-2

た

ターゲット・システム A-5
代数表記アセンブリの使用 2-18
代入文 A-6

タンジェント　7-94

ち

地方時　7-27
致命的エラー　2-37
注釈　2-37
直接呼び出し
　定義　A-6

て

底が 10 の対数　7-66
定数
　C 言語　5-2
　.const セクション　5-34
　アセンブラ
　　C からのアクセス方法　6-26
　定義　A-6
　文字列　6-11
定数の未定義化　2-20
ディレクトリ
　include ファイルの　2-18
　アセンブリおよびクロスリファレンス・リスト・ファイル　2-26
　アセンブリ・ファイル　2-26
　一時ファイル　2-26
　オブジェクト・ファイル　2-26
　絶対リスト・ファイル　2-26
ディレクトリの指定　2-26
データ
　型
　　C 言語　5-2
　定義　A-1
　フローの最適化　3-19
データ型　5-6
データ・ブロックの書き込み関数　7-60
テキスト
　定義　A-2
デバッグ
　最適化されたコード　3-14
　シンボリックに
　　定義　A-5
デュアル MAC 最適化
　-mb コンパイラ・オプション　2-18

と

統合プリプロセッサ
　定義　A-6
動的メモリ割り当て
　説明　6-7
　定義　A-6
トークン　7-93
トークンを続ける
　プリプロセッサ疑似命令　5-37

な

夏時間　7-27

に

入出力
　関数のまとめ　7-33
入出力関数
　CLOSE　7-12
　LSEEK　7-12
　OPEN　7-13
　READ　7-14
　RENAME　7-14
　UNLINK　7-15
　WRITE　7-15
入出力バッファのフラッシュ関数　7-54

ね

ネーム・デマングラ
　説明　1-4, 1-7

は

パーサ
　定義　A-6
バージョン番号
　表示　2-20
バイト　A-6
ハイパボリック算術関数
　math.h ヘッダに定義される　7-22
　コサイン　7-47
　サイン　7-80

ハイパボリック算術関数（続き）
タンジェント 7-95
バッファ
指定関数 7-77
定義および関連付け関数 7-79
パラレル・バス・コンフリクトを警告として
-atb アセンブラ・オプションの使用方法 2-28
汎用ユーティリティ関数 7-26
abort 7-39
abs 7-39
atexit 7-43
atof 7-43
atoi 7-43
atol 7-43
bsearch 7-44
calloc 7-45
div 7-49
exit 7-50
free 7-58
labs 7-39
ldiv 7-49
ltoa 7-66
malloc 7-67
minit 7-71
qsort 7-74
rand 7-75
realloc 7-75
srand 7-75
strtod 7-92
strtol 7-92
strtoul 7-92

ひ

ヒープ
確保された空間 6-5
説明 6-7
引数ブロック
説明 6-16
非推奨コンパイラ・オプション 2-30
ビッグエンディアン A-6
ビット
アドレッシング 6-9
フィールド 5-3, 6-9
ビット・フィールド 5-37
非ローカル・ジャンプ 7-78
非ローカル・ジャンプ関数 7-32
非ローカル・ジャンプ関数とマクロ
のまとめ 7-32

ふ

ファイル
オプション 2-24
拡張子
変更方法 2-24
コピー 2-27
再命名関数 7-76
除去関数 7-76
名 2-24
ファイル位置の取得関数 7-60
ファイル位置の設定関数
fseek 関数 7-59
fsetpos 関数 7-60
ファイル位置標識設定関数 7-77
ファイル名
生成関数 7-96
ファイルレベルの最適化 3-3, A-6
ファイル・オープン関数 7-56, 7-58
ファイル・クローズ関数 7-53
フィールドの操作 6-9
浮動小数点 7-31
浮動小数算術関数 7-22
acos 7-40
asin 7-41
atan 7-42
atan2 7-42
ceil 7-45
cos 7-47
cosh 7-47
exp 7-50
fabs 7-51, 7-52
floor 7-55
fmod 7-55
ldexp 7-64
log 7-65
log10 7-66
modf 7-71
pow 7-72
sin 7-80
sinh 7-80
sqrt 7-81
tan 7-94
tanh 7-95
浮動小数点の剰余 7-55
プラグマ
定義 A-6
プラグマ疑似命令 5-17
C54X_CALL 5-19
C54X_FAR_CALL 5-19
CODE_SECTION 5-18
DATA_ALIGN 5-21
DATA_SECTION 5-22, 6-41
FAR 5-23, 6-41, 6-42

プラグマ疑似命令（続き）
FUNC_CANNOT_INLINE 5-24
FUNC_EXT_CALLED 5-24
FUNC_IS_PURE 5-25
FUNC_IS_SYSTEM 5-26
FUNC_NEVER_RETURNS 5-26
FUNC_NO_GLOBAL_ASG 5-27
FUNC_NO_IND_ASG 5-27
INTERRUPT 5-28
MUST_ITERATE 5-28
UNROLL 5-30
プリプロセッサ
_INLINE シンボル 2-46
name を事前定義する方法 2-18
エラー・メッセージ 2-33
シンボル 2-33
スタンドアロン
定義 A-5
制御方法 2-33
定義 A-7
リスト・ファイル 2-35
プリプロセッサ疑似命令 2-33
C 言語 5-4
トークンを続ける 5-37
プログラム終了関数
abort (exit) 7-39
atexit 7-43
exit 7-50
プログラムレベルの最適化
実施 3-5
制御 3-5
定義 A-7
ブロック
セクションの割り振り 4-11, 6-4
定義 A-7
分岐の最適化 3-17
文献
関連 vi

へ

平方根 7-81
ヘッダ・ファイル
assert.h 7-17
ciso646 7-22
csetjmp 7-22
ctype.h 7-17
errno.h 7-18
extaddr.h 6-40, 7-18
file.h 7-18
float.h 7-19
iso646.h 7-22
limits.h 7-19
ヘッダ・ファイル（続き）
math.h 7-22
setjmp.h 7-22, 7-78
stdarg.h 7-23
stddef.h 7-23
stdint.h 7-24
stdio.h 7-25
stdlib.h 7-26
string.h 7-27
time.h 7-27
変換 5-3, 7-17
C 言語 5-3
変数
アセンブラ
C からのアクセス方法 6-25
定義 A-7
変数コンストラクタ（C++） 4-10
変数の実行時の初期化 > 6-8
変数の割り当て 6-9

ほ

ポインタの組み合わせ 5-36
保護されない定義制御のインライン展開 2-46
補足ユーザ情報ファイル
生成方法 2-17

ま

前処理されたリスト・ファイル 2-35
アセンブリ依存行 2-27
アセンブリ・インクルード・ファイル 2-27
マクロ
アルファベット順の参照 7-39
定義 2-33, A-7
展開 2-33
マクロ展開
定義 A-7
マクロ呼び出し
定義 A-7
マップ・ファイル
定義 A-7
マルチバイト文字 5-2

む

無効化
シンボリック・デバッグ 2-23

め

メモリ管理関数
calloc 7-45
free 7-58
malloc 7-67
minit 7-71
realloc 7-75
メモリ・プール 7-67
確保された空間 6-5
メモリ・マップ
定義 A-7
メモリ・モデル
構造体のパック 6-9
スタック 6-6
スモール 6-2
セクション 6-4
動的メモリ割り当て 6-7
フィールドの操作 6-9
変数の初期化 6-8
変数の割り当て方法 6-9
ラージ 6-3

も

文字
変換関数 7-94
のまとめ 7-30
読み込み関数
単一文字 7-54
複数文字 7-55
文字セット 5-2
文字定数 5-36
文字判定関数
isalnum 7-63
isalpha 7-63
isascii 7-63
iscntrl 7-63
isdigit 7-63
isgraph 7-63
islower 7-63
isprint 7-63
ispunct 7-63
isspace 7-63
isupper 7-63
isxdigit 7-63
文字変換関数
toascii 7-96
tolower 7-97
toupper 7-97
モジュール
出力 A-5
文字列関数 7-27, 7-37
strcmp 7-84
文字列のコピー 7-89
文字列の比較 7-89
文字列の連結 7-82, 7-88

ゆ

ユーティリティ
概要 1-7

よ

抑止方法
診断メッセージ 2-39
シンボリック・デバッグ 2-23
呼び出し
マクロ
定義 A-7
呼び出し規則
コンパイラに、特定の呼び出し規則との互換性を強制 2-17
読み込み
ストリーム関数
標準入力から 7-77
文字列 7-59, 7-82
文字列から配列へ 7-57
文字関数
単一文字 7-54
次の文字関数 7-61
複数文字 7-55
読み込み関数 7-62

ら

ライブラリ
C 入出力 7-4
オブジェクト
定義 A-3
ランタイムサポート 7-2, A-7
ライブラリ作成ユーティリティ 1-3, 8-1 ～ 8-6
オプション 8-2, 8-3
ラベル
定義 A-7
保持する方法 2-28
ランタイム型情報 5-5
ランタイム環境 6-1 ～ 6-52
2 次システム・スタック 6-6

ランタイム環境（続き）
アセンブリ言語での変数の定義　6-24
アセンブリ言語と C とのインターフェイス
6-22 〜 6-37
インライン・アセンブリ言語　6-27
関数呼び出し規則　6-16 〜 6-21
システムの初期化　6-45 〜 6-52
スタック　6-6
定義　A-8
メモリ・モデル
構造体のパック　6-9
自動初期化時　6-7
セクション　6-4
動的メモリ割り当て　6-7
フィールド操作　6-9
変数の割り当て　6-9
レジスタ規則　6-12 〜 6-15
割り込み処理　6-38 〜 6-39
ランタイムサポート　7-1
関数
定義　A-7
まとめ　7-29
マクロ
まとめ　7-29
ライブラリ　7-2, 8-1
説明　1-3
定義　A-7
とリンク　4-8

り

リスト・ファイル
クロスリファレンスの作成方法　2-29
定義　A-8
リトルエンディアン　A-8
リンカ　4-1
コマンド・ファイル　4-12
説明　1-3
定義　A-8
抑止方法　2-17
リンク方法
C コード　4-1
C/C++ コード　4-1
コンパイラを使って　4-2

る

累乗　7-72
ループ最適化　3-22
ループの展開
定義　A-8
ループ不変の最適化　3-22

れ

レジスタ
使用規則　6-13
レジスタ規則　6-12
ステータス・レジスタ　6-14
専用のレジスタ　6-14
列挙型定数リスト
コンマを続ける　5-37

ろ

ローダ　5-33
定義　A-8
ロー・リスト・ファイル
-pl オプションを使用する生成方法　2-43
識別子　2-43

わ

ワイルドカード　2-24
割り当て　A-8
割り込み
組み込み関数　6-39
割り込み処理　6-38

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/

TMS320C55x
オプティマイジング（最適化）
C/C++ コンパイラ
ユーザーズ・マニュアル

第 1 版 2005 年 9 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
’05.09

日本テキサス・インスツルメンツ株式会社

SPRUE45