TEXAS INSTRUMENTS

TMS320C54x

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

2001 *Digital Signal Processing Products*

SPRU103 翻訳版

JAJU017 WAS SPRU367A

TMS320C54x
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

2001 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 TI といいます）は、通知をすることなくその製品を変更し、もしくは半導体集積回路製品またはサービスの製造または提供を中止することがありますので、お客様は、発注される前に、これから参照しようとする情報が最新のものであることを確実にするため、最新版の情報を取得するようお勧めします。

TI は、その半導体集積回路製品および関連するソフトウェアが、TI の標準保証条件に従い販売の際の現行の仕様書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に関する特定の検査は、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。

半導体集積回路製品を使用する或る種の用途の中には、死亡、傷害、または財産もしくは環境に深刻な被害をもたらす危険の可能性を包含するものがあります。（以下、これらを「重大用途」といいます。）

TI の半導体集積回路製品は、生命維持の用途、装置、システム、その他の重大用途に使用できるように設計も、意図も、承認も、また保証もされておりません。

TI の製品を当該重大用途に組込むことは、お客様独自のリスクでなされることと解釈されます。TI 製品を当該用途に使用される場合は、事前に TI の役員の書面による承諾を必要とします。危険な可能性を有する用途に関する質問は、TI の営業所を通じて、TI 迄お寄せ下さい。

お客様の用途に TI 製品を使用することに伴う危険を最小のものとするため、製品固有の危険性もしくは、組み合わせによって起こりうる危険性を最小にするための、適切な設計上および作動する上での安全対策は、お客様がとらなくてはなりません。

TI は製品の使用用途に関する支援、お客様の製品の設計、ソフトウェアの性能、または特許侵害もしくはサービスに対する責任を負うものではありません。また TI は、その半導体集積回路製品もしくはサービスが使用されうる、もしくは使用されている組み合せ、機械装置、もしくは方法をカバーしている、もしくはそれ等に関連している特許権、著作権、回路配置利用権、その他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示もしておりません。

以上

IN-0104

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

最初にお読みください

本書について

本書は、以下のコンパイラ・ツールの使用方法について説明したものです。

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

本コンパイラは、米国規格協会 (ANSI) 準拠の標準 C ソース・コードだけでなく、C++ ソース・コードも受け付け、TMS320C54x デバイス用のアセンブリ言語ソース・コードを作成します。

本書では、C/C++ コンパイラの特性について説明します。本書では、C プログラムの作成方法を理解していることを前提とし ます。ANSI C 規格に準拠する C 言語については、ブライアン W. カーニハンとデニス M. 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。本書では、ANSI C に対立する語として「K&R C」という表現を使うことがあります。これは、カーニハンとリッチーの *The C Programming Language* の第 1 版に定義されている C 言語を指しています。

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、以下の表記規則を使用します。

- TMS320C54x デバイスは、C54x と示してあります。
- 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）との区別のために、ボールド **(bold version)**で示してあります。

  C コードの例を以下に示します。

```
#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)**、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、その表記通り入力する箇所です。 構文でイタリック体の部分は、入力する情報の種類を示しています。コマンド行で入力する構文は、以下のように縁取りのある枠囲みの中心に示します。

  cl500 [*options*] [*filenames*] [*–z* [*link_options*] [*object files*]]

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

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

- 大括弧（[]）は、任意のパラメータを示します。 任意のパラメータを使用する場合、指定内容はこの括弧内に入力します。括弧そのものは入力不要です。以下に、任意のパラメータ付きのコマンドの例を示します。

  cl500 [*options*] [*filenames*] [*–z* [*link_options*] [*object files*]]

  cl500 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。以下に、実際の構文には含まれていないが、−c または −cr のいずれ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示します。

  lnk500 {**–c** | **–cr**} *filenames* [–o *name.out*] **–l** *libraryname*

当社発行の関連文献

以下の文献は、TMS320C54x および関連サポート・ツールの解説書であり、当社から刊行しています。当社の刊行書を入手するには、タイトルと文献番号 (裏表紙に書いてあります) をお知らせの上、プロダクト・インフォメーション・センター (PIC)、0120−81−0026 にお問い合わせください。

TMS320C54x DSP Reference Set は、5 巻で構成されており、文献番号 SPRU210 でまとめて発注できます。どれか 1 巻だけ発注する場合は、個々の文献の番号を使います。

- ***TMS320C54x リファレンス・セット、Volume 1: CPU 及びペリフェラル*** (文献番号 SPRU131) は、TMS320C54x™ 16 ビット固定小数点汎用ディジタル・シグナル・プロセッサについて解説しています。このプロセッサのアーキテクチャ、内部レジスタ構造、データおよびプログラムのアドレッシング、および命令パイプラインについて記載しています。また、開発サポート情報、パーツ・リスト、XDS510™ エミュレータを使用する際のデザイン上の考慮事項も含まれています。

- ***TMS320C54x DSP Reference Set, Volume 2: Mnemonic Instruction Set*** (文献番号 SPRU172) は、TMS320C54x™ ディジタル・シグナル・プロセッサのニーモニック命令を個別に解説しています。また、命令セットのクラスとサイクルの一覧も含まれています。

- ***TMS320C54x DSP Reference Set, Volume 3: Algebraic Instruction Set*** (文献番号 SPRU179) は、TMS320C54x™ ディジタル・シグナル・プロセッサの代数表記命令について個別に解説しています。また、命令セットのクラスとサイクルの一覧も含まれています。

- ***TMS320C54x DSP Reference Set, Volume 4: Applications Guide*** (文献番号 SPRU173) は、TMS320C54x™ ディジタル・シグナル・プロセッサのソフトウェアとハードウェアのアプリケーションについて解説しています。また、開発サポート情報、パーツ・リスト、および XDS510™ エミュレータを使用する際のデザイン上の考慮事項も含まれています。

- ***TMS320C54x リファレンス・セット、Volume 5: 高機能ペリフェラル*** (文献番号 SPRU396) は、TMS320C54x™ ディジタル・シグナル・プロセッサで使用可能な高機能ペリフェラルについて解説しています。マルチチャネル・バッファド・シリアル・ポート (McBSP)、ダイレクト・メモリ・アクセス (DMA) コントローラ、内部でのプロセッサ間通信、HPI−8 および HPI−16 ホスト・ポート・インターフェイスについても解説しています。

TMS320C54x アセンブリ言語ツール ユーザーズ・マニュアル (文献番号 SPRU366) は、TMS320C54x™ 世代のデバイスのアセンブリ言語ツール (アセンブラ、リンカ、その他のアセンブリ言語コードの開発用ツール)、アセンブラ疑似命令、マクロ、共通オブジェクト・ファイル・フォーマット、およびシンボリック・デバッグ用の疑似命令について解説しています。

Code Composer ユーザーズ・マニュアル (文献番号 SPRU328) は、Code Composer が提供する開発環境を利用してリアルタイムの組み込み型 DSP アプリケーションをビルドする方法、およびそのデバッグ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、次の文献を参考にすることができます。

American National Standard for Information Systems—Programming Language C X3.159–1989 − 米国規格協会刊行 (C 言語に関する ANSI 規格)

ISO/IEC 14882-1998 − C++ プログラム言語に関する国際規格、米国規格協会 (ANSI) 刊行

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

The C Programming Language (第 2 版) − Brian W.Kernighan と Dennis M. Ritchie 著 Prentice−Hall (Englewood Cliffs, New Jersey) 刊行 (1988 年)

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

The Annotated C++ Reference Manual − Margaret A. Ellis と Bjarne Stroustup 著、Addison−Wesley Publishing Company (Reading,Massachusetts) 刊行 (1990 年)

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

商標

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

目次

図

表

例

第 1 章

はじめに

TMS320C54x 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。

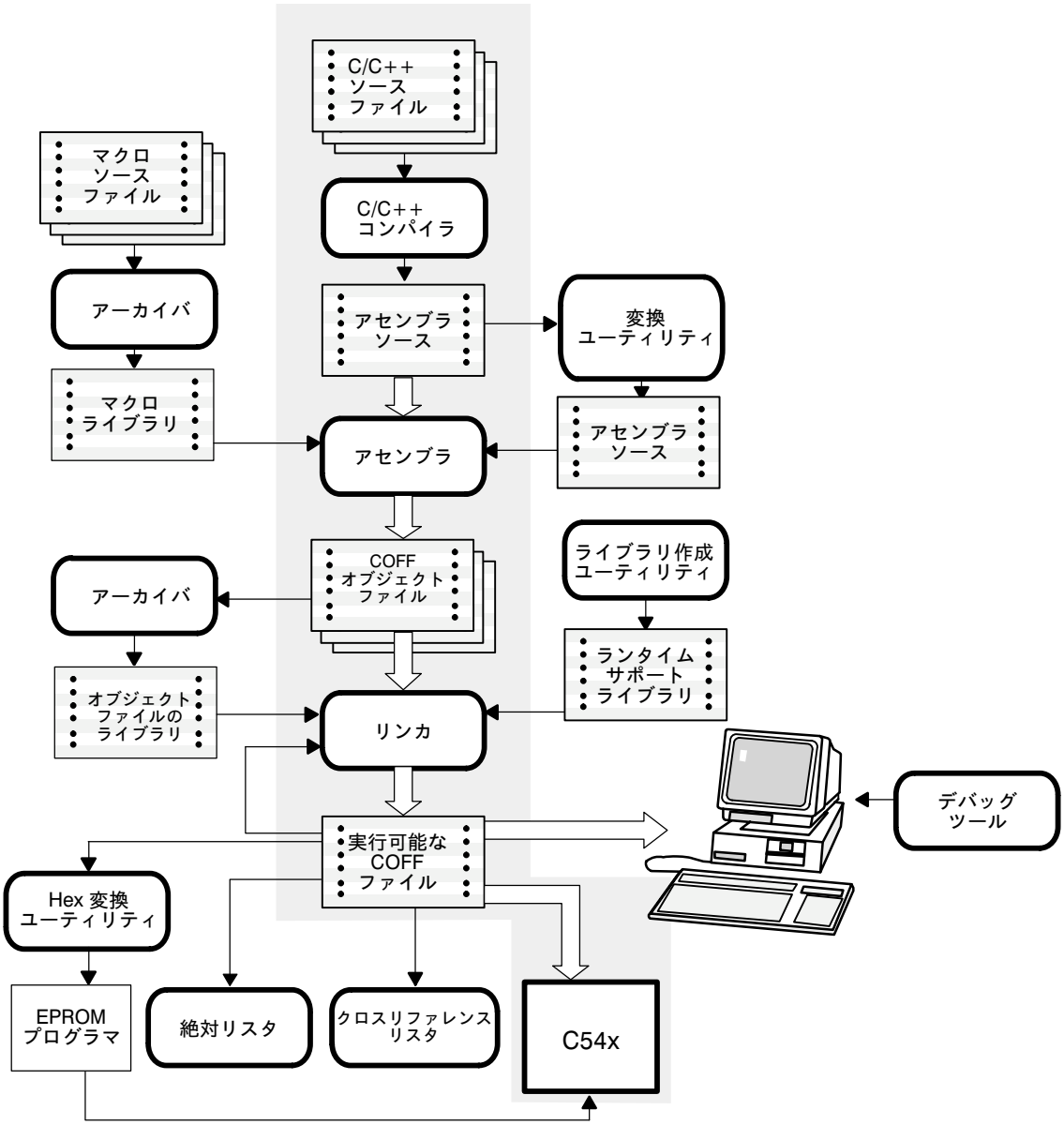
この章では、それらのツールの概要と最適化 C/C++ コンパイラの機能について説明します。アセンブラとリンカについては、TMS320C54xアセンブリ言語ツール ユーザーズ・マニュアル に詳しい説明があります。

項目 **ページ**

1.1 ソフトウェア開発ツールの概要

図 1−1 は、C54x ソフトウェア開発のフローを示したものです。図の中の陰影を付けた部分は、 C/C++ 言語プログラムのソフトウェア開発における最も一般的な経路 です。それ以外の部分は、開発プロセスを強化する周辺機能です。

図 1−1. TMS320C54x のソフトウェア開発フロー



以下で、図 1−1 で示した各ツールについて解説します。

- **C/C++ コンパイラ**は C/C++ のソース・コードを受け取り、C54x のアセンブリ言語ソース・コードを生成します。本コンパイラ・パッケージには、**シェル・プログラム、オプティマイザ**、および**インターリスト・ユーティリティ**が含まれています。
  - シェル・プログラムを使用すると、ソース・モジュールのコンパイル、アセンブル、およびリンクを 1 ステップで行うことができます。
  - 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。
  - インターリスト・ユーティリティは、C/C++ のソース文をアセンブリ言語の出力に差し込みます。

  シェル・プログラムを使用した C/C++ コンパイラ、オプティマイザ、およびインターリスト・ユーティリティの起動方法については、第 2 章「C/C++ コンパイラの使用」を参照してください。

- **アセンブラ**は、アセンブリ言語のソース・ファイルを機械語のオブジェクト・ファイルに変換します。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は、TMS320C54x アセンブリ言語ツール ユーザーズ･マニュアルで解説されています。

- **リンカ**は、複数のオブジェクト･ファイルを結合して 1 つの実行可能なオブジェクト･モジュールを作成します。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 コードのリンク」を参照してください。リンカの詳細は、TMS320C54x 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 1 つのアーカイブ･ファイルにまとめることができます。このアーカイブ･ファイルを「ライブラリ」といいます。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。アーカイバの使用が最も便利なのは、オブジェクト･モジュールのライブラリを作成する場合です。アーカイバの使用 方法については、TMS320C54x 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。

- **ニーモニック表記から代数表記への変換ユーティリティ**は、アセンブリ言語のソース・ファイルを変換します。このユーティリティは、ニーモニック命令で記述されたアセンブリ言語ソース・ファイルを受け入れます。その後、ニーモニック命令を代数表記命令に変換し、代数表記命令で記述されたアセンブリ言語ソース・ファイルを生成します。

- **ライブラリ作成ユーティリティ**を使用すれば、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)。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、 'C54xコンパイラによってサポートされているANSI 標準ランタイムサポート関数、コンパイラ・ユーティリティ関数、浮動小数点算術関数、および C I/O 関数が入っています。詳細は、第 7 章「ランタイムサポート関数」を参照してください。

- 'C54x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、ほとんどの 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、Tektronix のいずれかのオブジェクト・フォーマットに変換します。変換後のファイルは、EPROM プログラマにダウンロードできます。Hex 変換ユーティリティの使用方法については、TMS320C54x 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、リンク後のオブジェクト・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生成します。これらの .abs ファイルをアセンブルすると、相対アドレスでなく絶対アドレスが入ったリストを作成できます。絶対リスタを使用しないと、そのようなリストの作成は多くの手動操作が必要となるため、面倒な作業となります。絶対リスタの使用方法については、TMS320C54x 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、シンボル、シンボルの定義、およびリンク済みソース・ファイルでのシンボルの参照をリストにして示します。クロス・リファレンス・リスタの使用方法については、TMS320C54x 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。

- この開発プロセスの主な目的は、 TMS320C54x のデバイスで実行できるモジュールを生成することで、いずれかのデバッグ・ツールを使用すれば、生成したコードに改善や修正を加えることができます。使用できるデバッグ・ツールを次に示します。

  - 命令の精度を確認するためのソフトウェア・シミュレータ
  - 拡張開発システム (XDS510™) エミュレータ
  - 評価モジュール (EVM)

  これらのツールには、Code Composer Studio の内部からアクセスします。詳細は、Code Composer Studio ユーザーズ・マニュアル を参照してください。

1.2 C/C++ コンパイラの概要

C54x C/C++ コンパイラは、標準 ANSI C/C++ プログラムを C54x 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。以下に、本コンパイラの主要な特徴を紹介します。

1.2.1 ANSI 標準

次の特徴は、ANSI 規格に関するものです。

- **ANSI 標準 C**

  C54x コンパイラは、 ANSI の規格により定義され、カーニハンとリッチーの *The C Programming Language (K & R)* 第 2 版に記述された、ANSI C 規格に完全準拠しています。ANSI 標準規格 C には、C 言語の拡張機能が含まれています。これらの拡張機能により、最大限の移植性と機能の強化を実現しています。

- **C++**

  C54x C/C++ コンパイラは、エリスとストラウストラップの *The Annotated C++ Reference Manual* (ARM) に定義された C++ のほか、C++ に関する ISO/IEC 14882−1998 規格の多数の機能もサポートしています。

- **ANSI 標準ランタイムサポート**

  このコンパイラ・ツールには、完全なランタイム・ライブラリが添付されています。すべてのライブラリ関数は、ANSI 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含まれています。信号処理関数は、ターゲット・システムによって異なるので、含まれていません。

  C++ ライブラリには、ANSI C サブセット、および言語サポートに必要な各種のコンポーネントが含まれています。

  詳細は、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・ソースの出力**

  本コンパイラでは、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) を使用すると、リンク時にシステムのメモリ･マップを定義できます。これにより、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豊富にサポートしています。

- **ROM データを初期化するコード**

  スタンドアロンの組み込みアプリケーションの場合、本コンパイラを使用すれば、すべてのコードと初期設定データを ROM にリンクすることができます。これによって C/C++ コードを リセット時から実行できるようになります。

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・シェル・プログラム**

  本コンパイラ･ツールには、シェル･プログラムが含まれています。これを使用すると、プログラムのコンパイル、アセンブル、リンクを 1 ステップで実行できます。詳細は、2-2 ページの 2.1 節「シェル･プログラムについて 」を参照してください。

- **柔軟性のあるアセンブリ言語インターフェイス**

  本コンパイラは、単純化された呼び出し規則を採用しています。したがって、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 (実行時) 環境」を参照してください。

1.2.4 コンパイラ操作

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、2-24 ページの 2.5 節「プリプロセッサの制御」を参照してください。

- **最適化**

  本コンパイラは、高性能の最適化パスを使用します。この最適化パスでは、高度な技法により、効率性の高い、コンパクトなコードを C/C++ ソースから生成します。 一般的な最適化は、すべての C/C++ コードに適用できます。また、'C54x 固有の最適化では、'C54x アーキテクチャ固有の機能が利用されます。 C/C++ コンパイラの最適化技法の詳細は、第 3 章「コードの最適化」を参照してください。

1.2.5 ユーティリティ

次の特徴は、コンパイラ・ユーティリティに関するものです。

- **ソース・インターリスト・ユーティリティ**

  本コンパイラ･ツールには、オリジナルの C/C++ ソース文をコンパイラのアセンブリ言語出力に差し込むユーティリティが入っています。 このユーティリティにより、C/C++ ソース文ごとに生成されたアセンブリ･コードを調べることができます。詳細は、2-41 ページの 2.10 節「インターリスト・ユーティリティの使用方法」を参照してください。

- **ライブラリ作成ユーティリティ**

  ライブラリ作成ユーティリティを使用すると、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。

1.3 コンパイラと Code Composer Studio

Code Composer Studio は、コード生成ツールを使用する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 のプロジェクトは、ターゲット・プログラムまたはターゲット・ライブラリをビルドするために必要なすべての情報を常に記録しています。プロジェクトが記録するものは、次のとおりです。

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

Code Composer Studio で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、ユーザのプログラムに必要なコンパイル、アセンブル、リンクなどを実行します。

コンパイラ・オプション、アセンブラ・オプション、およびリンカ・オプションは、Code Composer Studio の「Build Options」ダイアログで指定できます。ほとんどすべてのコマンド行オプションは、このダイアログの中で表示されます。表示されないオプションは、ダイアログの上部に表示される編集可能なテキスト・ボックスに直接そのオプションを入力することで、指定できます。

本書では、コマンド行インターフェイスからコード生成ツールを使用する方法を解説します。Code Composer Studio の使用方法については、Code Composer Studio ユーザーズ・マニュアルを参照してください。Code Composer Studio の内部でコード生成ツールのさまざまなオプションを設定する方法については、「Code Generation Tools」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

ソース・プログラムを TMS320C54x で実行可能なコードに変換するプロセスは、複数のステップから構成されています。実行可能なオブジェクト・ファイルを作成するには、ソース･ファイルをコンパイル、アセンブル、およびリンクする必要があります。’C54x コンパイラ・ツールには、これらすべてのステップを 1 つのコマンドで実行できる特別なシェル・プログラム cl500 が組み込まれています。本章では、このシェル・プログラム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・ユーティリティについても説明します。

項目 **ページ**

2.1 シェル・プログラムについて

コンパイラ・シェル・プログラムを使用すると、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。シェルは、 1 つ以上のソース・モジュールに対して次の処理を実行します。

- ❑ **コンパイラ**には、パーサ、オプティマイザ、コード・ジェネレータが含まれています。C/C++ のソース・コードを取り込んで、C54x アセンブリ言語ソース･コードを生成します。

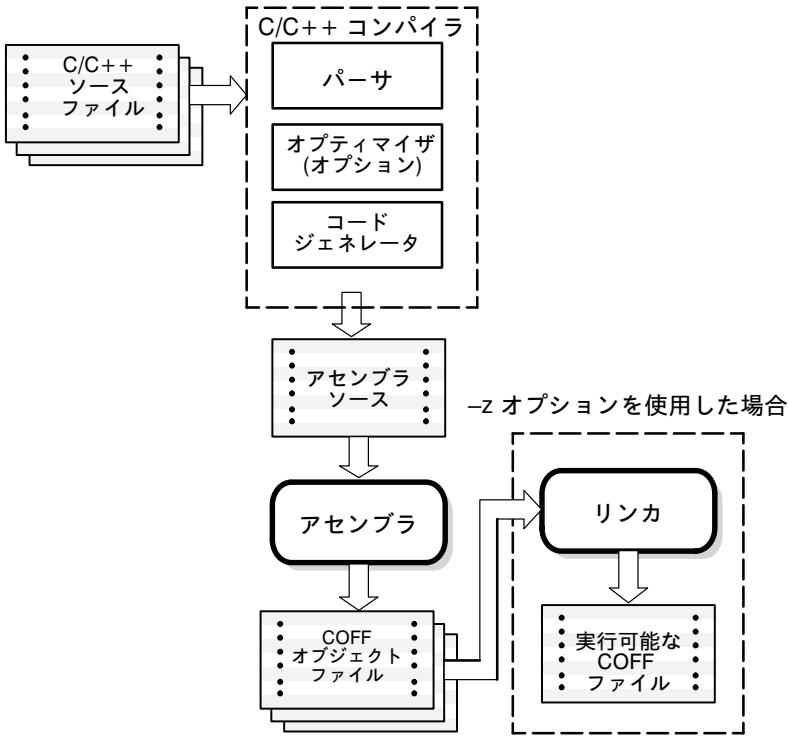
  C ファイルと C++ ファイルを 1 つのコマンドでコンパイルすることができます。コンパイラは、ファイル名拡張子の規則を使用して、C ファイルと C++ ファイルを区別します (詳細は、2.3.2 項「ファイル名の指定」を参照してください)。

- ❑ **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、ファイルをリンクして実行可能オブジェクト・ファイルを作成します。リンカの使用はこの時点では、必須ではありません。シェルでさまざまなファイルのコンパイルとアセンブルを行い、後からそれらのファイルをリンクすることもできます。独立したステップでファイルをリンクする方法については、第 4 章「C/C++ コードのリンク」を参照してください。

デフォルトでは、シェルはファイルのコンパイルとアセンブルを実行します。しかしながら、−z シェル・オプションを使用するとファイルのリンクも実行させることができます。図 2−1 は、シェルの経路 (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.　シェル・プログラムの概要



アセンブラとリンカの詳細は、TMS320C54x 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・シェルの起動

コンパイラ・シェルを起動するには、次のように入力します。

cl500 [*options*][*filenames*][**–z** [*link_options*][*object files*]]

cl500	コンパイラとアセンブラを実行するコマンドです。
options	シェルによる入力ファイルの処理方法に影響を与えるオプションです (各オプションについては、2-7 ページの表 2–1 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・ファイルのどれかを指定します。
–z	リンカの起動オプションです。リンカの起動の詳細は、第 4 章「C/C++ コードのリンク」を参照してください。
link_options	リンク・プロセスの制御オプションです。
object files	リンク・プロセスの追加オブジェクト・ファイル名です。

コマンド行では、–z オプションとその関連情報 (リンカ・オプションとオブジェクト・ファイル) は、すべてのファイル名とコンパイラ・オプションの後に指定してください。その他のオプション (リンカ・オプションを除く) とファイル名は、いずれもコマンド行で任意の順序で指定できます。たとえば、symtab.c と file.c という名前の 2 つのファイルをコンパイルし、seek.asm という名前の第 3 のファイルをアセンブルし、進捗状況のメッセージを抑止する (–q) 場合は、次のように入力します。

```
cl500 –q symtab.c file.c seek.asm
```

cl500 は検出したソース・ファイルを、C/C++ ファイルの場合には [C/C++ ファイル名]、アセンブリ言語の場合には<asmファイル名>のように括弧で囲んで出力します。上記の例では、–q オプションを指定することによって、cl500 が生成する追加の進捗情報の表示を抑止しています。上記のコマンドを入力すると、次のメッセージが生成されます。

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

一般の進捗情報は、各コンパイラ・パスの見出しと、処理したときの関数名で構成されます。次の例は、−q オプションなしで単一のファイル (symtab) をコンパイルしたときの進捗情報出力を示したものです。

```
% cl500 symtab
TMS320C54x ANSI C Compiler                Version x.xx
Copyright (c) 2001           Texas Instruments Incorporated
   ″symtab.c″   ==> main
TMS320C54x ANSI C Codegen                 Version x.xx
Copyright (c) 2001      Texas Instruments Incorporated
   ″symtab.c″:  ==> main
TMS320C54x COFF Assembler                 Version x.xx
Copyright (c) 2001      Texas Instruments Incorporated
 PASS 1
 PASS 2
```

2.3 オプションによるコンパイラの動作の変更

オプションの指定により、シェルとシェルが実行するプログラムの両方を制御できます。この節では、オプションの使用規則について説明した後、各オプションについての概要を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、詳細な説明も記載しています。

コンパイラ・オプションには、次の規則が適用されます。

- ❏ オプションは 1 文字だけか、または連続する文字で構成されます。
- ❏ オプションは、大文字小文字は区別されません。
- ❏ オプションの先頭にはハイフンを付けます。
- ❏ パラメータのない 1 文字のオプションは、続けて指定できます。たとえば、−sgq は −s −g −q と同じです。
- ❏ 2 文字のペアのオプションのうち、最初の文字が同じものは続けて指定できます。たとえば、−pi、−pk、および −pl は、−pikl のように指定できます。
- ❏ *–uname* や *–idirectory* のようなパラメータをもつオプションについては、続けて指定できません。個別に指定する必要があります。
- ❏ 必須パラメータをもつオプションの場合は、オプションとパラメータの間にスペースを入れても入れなくても構いません。たとえば、名前を定義解除するオプションを指定するには、–u *name* と入力しても *–uname* と入力しても同じです。ただし、必須の数値パラメータは、オプションの直後に (つまりオプションとパラメータの間にスペースを入れずに) 指定する必要があります。
- ❏ オプショナル・パラメータをもつオプションの場合は、パラメータの直後に続けてオプションを指定する必要があります (オプションとパラメータの間にスペースを入れてはいけません)。たとえば、最適化の最大量を指定するオプションは、−o 3 ではなく、−o3 と指定する必要があります。
- ❏ ファイルやオプションは、−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しなければなりません。

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 C54X_C_OPTION 環境変数を使用します。C_OPTION 環境変数の詳細は、2-22 ページの 2.4.2 項「デフォルト・シェル・オプションの設定 (C_OPTION および C54X_C_OPTION)」を参照してください。

表 2−1 は、全オプション (リンカ・オプションを含む) についてまとめたものです。表内には、各オプションについて、詳しい説明が載っているページを記載しています。必要に応じて参照してください。

コマンド行でパラメータを指定せずに cl500 と入力すると、オプションのまとめが表示されます。

表 2−1. シェル・オプションのまとめ

(a) コンパイラ・シェルを制御するオプション

オプション	機能	ページ
−*@filename*	コマンド行の延長として、ファイルの内容を解釈します。	2-14
−c	リンクを行いません (−z の無効化)。	2-14, 4-5
−d*name*[=*def*]	*name* を事前に定義します。	2-14
−g	シンボリック・デバッグを有効にします。	2-14
−gw	オブジェクト・ファイルで、DWARF デバッグ・フォーマットを使用してシンボリック・デバッグを有効にします。アプリケーションに C++ ソース・ファイルが含まれている場合は、このオプションでコンパイルしてください。	2-14
−i*directory*	#include 検索パスを定義します。	2-14, 2-25
−k	.asm ファイルを保存します。	2-14
−n	コンパイルのみを行います。	2-15
−q	進捗メッセージの出力を抑止します (静的実行)	2-15
−qq	すべてのメッセージの出力を抑止します (完全な静的実行)	2-15
−r *register*	グローバル・レジスタを予約します。	2-15
−s	オプティマイザのコメントをアセンブリ文に差し込みます。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文に差し込みます。	2-15
−ss	C/C++ のソースをアセンブリ文に差し込みます。	2-15, 3-13
−u*name*	*name* を未定義にします。	2-15
−v*value*	どのプロセッサの命令をビルトするか決定します。	2-16
−z	リンクの実行を有効にします。	2-16

表 2–1. シェル･オプションのまとめ (続き)

(b) 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[.]*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。	2-18
–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。	2-18
–ec[.]*newextension*	C ソース・ファイルのデフォルトの拡張子を設定します。	2-18
–ep[.]*newextension*	C++ ソース・ファイルのデフォルトの拡張子を設定します。	2-18
–es[.]*newextension*	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。	2-18

(c) ファイル名と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・ファイルとして取り扱います。	2-17
–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・ファイルとして取り扱います。	2-17
–fg*filename*	C *filename* を C++ ファイルとして処理します。	2-17
–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・ファイルとして取り扱います。	2-17
–fp*filename*	拡張子に関係なく、*filename* を C++ ファイルとして取り扱います。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++ ファイルとして取り扱います。	2-17

(d) 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。	2-19
–ff*directory*	アセンブリ・リスト・ファイルとクロスリファレンス・リスト・ファイルのディレクトリを指定します。	2-19
–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。	2-19
–fs*directory*	アセンブリ・ファイルのディレクトリを指定します。	2-19
–ft*directory*	一時ファイルのディレクトリを指定します。	2-19

表 2−1.　シェル・オプションのまとめ (続き)

(e) 構文解析を制御するオプション

オプション	機能	ページ
–pe	組み込みの C++ モードを有効にします。	5-29
–pi	定義制御されたインライン展開を抑止します (ただし、−o3 最適化の場合は自動インライン展開が行われます)。	2-37
–pk	K&R 互換性を許容します。	5-27
–pl	生リスト・ファイルを生成します。	2-34
–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。	3-6
–pr	リラックス・モードを有効にします。つまり、細かい ANSI 違反は無視します。	5-29
–ps	厳密な ANSI モードを有効にします (C/C++ の場合のみ。K&R C には適用されません)。	5-29
–px	クロスリファレンス・リスト・ファイルを生成します。	2-33
–rtti	ランタイム型情報 (RTTI) を有効にします。RTTI を使用すると、オブジェクトの型を実行時に決定できます。	5-5

(f) 前処理を制御するパーサ・オプション

オプション	機能	ページ
–ppa	前処理の後で、続けてコンパイルを実行します。	2-26
–ppc	前処理のみを実行します。前処理出力は、コメントとともに、入力ファイルと同じ名前で拡張子が .pp のファイルに書き込まれます。	2-26
–ppd	前処理のみを実行します。前処理出力は書き込まれず、標準の make ユーティリティへの入力に適した従続行のリストが書き込まれます。	2-27
–ppi	前処理のみを実行します。前処理出力は書き込まれず、#include 疑似命令によりインクルードされるファイルのリストが書き込まれます。	2-27
–ppl	前処理のみを実行します。前処理出力は、行制御情報 (#line 疑似命令) とともに、入力ファイルと同じ名前で拡張子が .pp のファイルに書き込まれます。	2-26
–ppo	前処理のみを実行します。前処理出力は、入力ファイルと同じ名前で拡張子が .pp のファイルに書き込まれます。	2-26

表 2−1. シェル・オプションのまとめ (続き)

(g) 診断を制御するパーサ・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)。	2-30
–pden	診断の識別子を、そのテキストとともに表示します。	2-30
–pdf	診断情報ファイルを生成します。	2-30
–pdr	注釈 (軽い警告) を発行します。	2-30
–pds*num*	*num* で識別される診断を抑止します。	2-30
–pdse*num*	*num* で識別される診断をエラーとして分類します。	2-30
–pdsr*num*	*num* で識別される診断を注釈として分類します。	2-30
–pdsw*num*	*num* で識別される診断を警告として分類します。	2-30
–pdv	行折り返し形式でオリジナル・ソースを表示する詳細な診断を提供します。	2-31
–pdw	診断の警告メッセージを抑止します (エラー・メッセージは発行されます)。	2-31

(h) C54x 固有のオプション

オプション	機能	ページ
–ma	特定のエイリアス指定技法を使用することを指示します。	3-11
–me	割り込みルーチン内の C 環境コードを抑止します。	2-15
–mf	コールはすべてファー・コールに、リターンはすべてファー・リターンにします。	2-15
–ml	遅延分岐の使用を抑止します。	2-15
–mn	−g によって無効化された最適化を有効にします。	3-15
–mo	バックエンド・オプティマイザを無効にします。	2-15
–mr	RPT 命令を無効にします。	2-15
–ms	コード空間が最小になるように最適化します。	2-15

表 2−1.　シェル・オプションのまとめ (続き)

(i)　最適化を制御するオプション

オプション	機能	ページ
–o0	レジスタの使用を最適化します。	3-2
–o1	−o0 による最適化を行い、更にローカルな最適化を行います。	3-2
–o2 or –o	−o1 による最適化を行い、更にグローバルな最適化を行います。	3-3
–o3	−o2 による最適化を行い、更にファイルに対して最適化を行います。	3-3
–oi*size*	自動インライン展開サイズを設定します (−o3 のみ)	3-12
–ol0 (–oL0)	ファイルが標準ライブラリ関数を変更することを、オプティマイザに通知します。	3-4
–ol1 (–oL1)	ファイルが標準ライブラリ関数を宣言することを、オプティマイザに通知します。	3-4
–ol2 (–oL2)	ファイルがライブラリ関数の宣言も変更も行わないことを、オプティマイザに通知します。 −ol0 および −ol1 オプションを取り消します。	3-4
–on0	最適化情報ファイルの生成を抑止します。	3-5
–on1	最適化情報ファイルを生成します。	3-5
–on2	詳細な最適化情報ファイルを生成します。	3-5
–op0	モジュールが、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含むことを指定します。	3-6
–op1	モジュールが、コンパイラに入力されるソース・コード外から変更される変数は含むが、ソース・コード外から呼び出される関数は使用しないことを指定します。	3-6
–op2	モジュールが、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、変更される関数や変数をどちらも含まないことを指定します (デフォルト)。	3-6
–op3	モジュールが、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は含むが、ソース・コード外から変更される変数は使用しないことを指定します。	3-6
–os	オプティマイザの注釈をアセンブリ文に差し込みます。	3-13

表 2−1.　シェル・オプションのまとめ (続き)

(j)　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	絶対リストの作成を有効にします。	2-20
–ac	アセンブリ・ソース・ファイルで大文字と小文字が区別されるようにします。	2-20
–ad*name*	*name* シンボルを設定します。	2-20
–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。	2-20
–ahi *filename*	アセンブリ・モジュールの先頭に指定されたファイルをインクルードします。	2-20
–al	アセンブリ・リスト・ファイルを生成します。	2-20
–amg	アセンブリ・ファイルに代数表記命令が含まれていることを指定します。	2-20
–ar*num*	*num* で識別されるアセンブラの要注意を抑止します。	2-20
–as	シンボル・テーブルにラベルを格納します。	2-20
–aw	パイプラインの衝突に対する警告を有効にします。	2-20
–au*name*	事前定義された定数 *name* の定義を解除します。	2-21
–ax	クロスリファレンス・ファイルを生成します。	2-21

表 2−1. オプションのまとめ (続き)

(k) 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。	4-6
–abs	絶対リスト・ファイルを生成します。このオプションの前に、−z オプションを指定しておく必要があります。	2-14
–ar	再配置可能な出力を生成します。	4-6
–b	シンボリック・デバッグ情報のマージを無効にします。	4-6
–c	実行時に変数を自動初期設定します。	4-6
–cr	リセット時に変数を自動初期設定します。	4-6
–e *global_symbol*	エントリ・ポイントを定義します。	4-6
–f *fill_value*	埋め込み値を定義します。	4-6
–g *global_symbol*	*global_symbol* のグローバルを維持します (−h の無効化)。	4-6
–h	グローバル・シンボルを静的にします。	4-6
–heap *size*	ヒープ・サイズ (ワード数) を設定します。	4-6
–i *directory*	ライブラリ検索パスを定義します。	4-6
–j	条件付きリンクを抑止します。	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。	4-7
–q	進捗メッセージの出力を抑止します (静的実行)。	4-7
–r	再配置可能な出力を生成します。	4-7
–s	シンボル・テーブルを除去します。	4-7
–stack *size*	スタック・サイズ (ワード数) を設定します。	4-7
–u *symbol*	シンボルを未定義にします。	4-7
–v*n*	出力 COFF フォーマットを指定します。デフォルトのフォーマットは COFF2 です。	4-7
–w	未定義の出力セクションが作成された場合にメッセージを表示します。	4-7
–x	ライブラリの再読み取りを強制します。	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の内容を付加します。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制限に対する回避策として使用できます。注釈を挿入する場合は、コマンド・ファイルの行頭に 「#」 か 「;」 を入力します。

コマンド・ファイル内部では、組み込みスペースまたはハイフンが含まれている filenames または option パラメータは、引用符で囲む必要があります。たとえば、“this-file.obj” のようにします。

–*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 C54X_C_OPTION 環境変数で –z を指定しているけれどもリンクをしたくないといった場合に便利です。詳細は、4-5 ページの 4.3 節「リンカを無効にする方法 (–c シェル・オプション)」を参照してください。

–d*name*[=*def*] プリプロセッサに対して定数 *name* を事前に定義します。各 C/C++ ソース･ファイルの先頭に #define name def を挿入するのと同じ結果が得られます。オプションの [= def] を省略すると、*name* は 1 に設定されます。

–g C/C++ ソース・レベル・デバッガが使用するシンボリック・デバッグ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。

–gw C/C++ ソース・レベルのデバッガが使用する DWARF シンボリック・デバッグ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・ソースのデバッグが可能になります。このオプションは、アプリケーションに C++ ソース・ファイルが含まれているときにデバッグ情報を生成する場合に便利です。DWARF デバッグ・フォーマットの詳細は、*DWARF Debugging Information Format Specification*, 1992–1993 (UNIX International, Inc. 刊行) を参照してください。

–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* に指定したディレクトリを追加します。ディレクトリを定義するこのオプションは、最高 32 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、2-25 ページの 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」を参照してください。

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

–me コンパイラが割り込み用に生成する特定のプロローグおよびエピローグ・コードを抑止します。デフォルトでは、コンパイラはこのコードを発行して、割り込みのための C/C++ 環境を明示的に設定します。このコードは、CPL ビットを設定し、OVM、SMUL、SST の各ビットをクリアします。このコードが不要な場合は、–me オプションを使用してください。

–mf コール命令をすべてファー・コールと解釈し、リターン命令をすべてファー・リターンと解釈します。ファー・コールは、16 ビット範囲外の関数を呼び出し、ファー・リターンは 16 ビット範囲外から値を戻します。コンパイラが C++ 用のファー・モードをサポートしないことに注意してください。

–ml 遅延分岐の使用を抑止します。

–mo バックエンド・オプティマイザを抑止します。

–mr 割り込みなしの RPT 命令の使用を抑止します。

–ms 速度ではなく、コード空間のサイズを最適化します。

–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。

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

–qq エラー･メッセージ以外のすべての出力を抑止します。

–r *register* *register* をグローバルに予約して、コード･ジェネレータとオプティマイザがレジスタを使用できないようにします。

–s インターリスト・ユーティリティを起動します。このユーティリティは、オプティマイザのコメントまたは C/C++ のソースをアセンブリ・ソースに差し込みます。オプティマイザを起動している場合は (–o*n* オプションを指定)、オプティマイザのコメントがコンパイラのアセンブリ言語出力に差し込まれます。オプティマイザを起動していない場合は、C/C++ ソース文がコンパイラのアセンブリ言語出力に差し込まれます。これは、個々の C/C++ 文に対して生成されたコードを検査するときに役立ちます。–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。

–ss インターリスト・ユーティリティを起動します。このユーティリティは、オリジナルの C/C++ ソースを、コンパイラが生成したアセンブリ言語に差し込みます。オプティマイザを起動 (–o*n* オプションを指定) するときにこの –ss オプションも指定すると、コードが大幅に再編成されることがあります。詳細は、2-41 ページの 2.10 節「インターリスト・ユーティリティの使用方法」を参照してください。

–u*name* 定義済みの定数 *name* の定義を解除します。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*value*　どのプロセッサに対する命令をビルトするか設定します。*value* には、次のいずれかを指定します。

541
542
543
545
545lp
546lp
548
549

–z　指定されたオブジェクト・ファイルに対してリンカを実行します。–z オプションとそのパラメータは、コマンド行で他のオプションの一番後に入力します。–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、4-2 ページの 4.1 節「リンカを 1 つの独立したプログラムとして起動する方法」を参照してください。

2.3.2　ファイル名の指定

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・ファイルを指定することができます。シェルは、ファイル名の拡張子によってファイルの種類を判別します。

拡張子	ファイル・タイプ
.c または拡張子なし (.c と見なされる)	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、使用するオペレーティング・システムによって決まります。大文字と小文字の区別がない場合、.C は C ファイルとして解釈されます。

すべてのソース・ファイルには拡張子が必要です。ファイル名拡張子の規則により、C/C++ ファイルのコンパイルとアセンブリ・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っています。

シェル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、2-17 ページの 2.3.3 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シェルの解釈の仕方、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、2-18 ページの 2.3.4 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、ディレクトリ内のすべての C ファイルをコンパイルする場合は、次のように入力します。

```
cl500 *.c
```

2.3.3 シェル・プログラム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)

シェルによるファイル名の解釈方法を、オプションによって変更することができます。シェルが認識しない拡張子を使用している場合、−fx オプションを使用することによってファイルの種類を指定できます。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種類に応じて、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・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って、シェルにファイル・タイプを正しく解釈させることができます。

```
cl500 –fc file.s –fa asmbly
```

−f オプションでは、ワイルドカードによる指定はできません。

−fg オプションを指定すると、コンパイラは C ファイルを C++ ファイルとして処理します。デフォルトでは、.c 拡張子の付いたファイルを C ファイルとして取り扱います。ファイル名拡張子の詳細は、2-16 ページの 2.3.2 項「ファイル名の指定」を参照してください。

2.3.4 シェル・プログラムによる拡張子の解釈、およびファイル作成時の拡張子の付け方の変更 (–e オプション)

シェル・プログラム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を、オプションによって変更することができます。コマンド行で、–e*x* オプションは、それを適用するすべてのファイル名の前に入力する必要があります。どのオプションについても、ワイルドカードにより指定することができます。

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

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

```
cl500 –ea .rrr –eo .o fit.rrr
```

拡張子のピリオド (.)、およびオプションと拡張子の間のスペースは入れても入れなくてもかまいません。上の例は次のように入力しても同じです。

```
cl500 –earrr –eoo fit.rrr
```

2.3.5 ディレクトリの指定

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

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

```
cl500 –fb d:\object ...
```

–ff*directory*
アセンブリ・リスト・ファイルとクロスリファレンス・リスト・ファイルを格納するディレクトリを指定します。デフォルトでは、オブジェクト・ファイルのディレクトリと同じディレクトリが使用されます。このオプションのみを指定してアセンブリ・リスト (–al) オプションもクロスリファレンス・リスト (–ax) オプションも指定しなかった場合、シェルは、–al オプションを指定したときと同じ機能になります。リスト・ファイル・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のディレクトリのパス名を入力します。

```
cl500 –ff d:\object ...
```

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

```
cl500 fr d:\object ...
```

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

```
cl500 –fs d:\assembly ...
```

–ft*directory*
一時中間ファイルを格納する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。

```
cl500 –ft d:\temp ...
```

2.3.6 アセンブラを制御するオプション

以下は、シェル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・ファイルにおいて、大文字と小文字が区別されないようにします。たとえば、−c はシンボルの ABC と abc を同じにします。このオプションを使用しない場合、大文字と小文字は区別されます (そ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・ファイルの先頭に *name* **.set** [*value*] を挿入するとの同じです。*value* を省略すると、シンボルは 1 に設定されます。

–ahc *filename* −hc オプションを指定して、アセンブラを起動します。アセンブラは、アセンブリ・モジュールに指定されたファイルをコピーします。ファイルは、ソース・ファイル文の前にコピーされます。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 オプションを指定して、アセンブラを起動します。アセンブラは、アセンブリ・モジュールに指定されたファイルをインクルードします。ファイルはソース・ファイル文の前にインクルードされます。インクルード・ファイルは、アセンブリ・リスト・ファイルには記録されません。

–al −l (小文字の L) アセンブラ・オプションを指定してアセンブラを起動します。アセンブリ・リスト・ファイルが生成されます。

–amg アセンブリ・ソース・ファイルに代数表記命令が含まれていることを指定します。

–ar*num* *num* で識別されるアセンブラの要注意を抑止します。要注意は、警告より重要度が低いアセンブラの情報メッセージです。*num* に値を指定しなかった場合は、すべての要注意が抑止されます。

–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。

–au*name* 事前定義された定数 *name* の定義を解除します。これにより、指定の定数に対する −ad オプションはすべて無効になります。

–aw アセンブリ・コードのパイプラインの衝突に対して警告を生成します。アセンブラはすべてのパイプラインの衝突を検出することができるわけではありません。パイプラインの衝突は、直線的コーディングでのみ検出されます。パイプラインの衝突を検出すると、アセンブラは警告を出力し、衝突を解決するために NOP などの命令によって埋める必要のあるレイテンシ・スロット (ワード数) を報告します。

–ax –x アセンブラ・オプションを指定してアセンブラを起動します。リスト・ファイルにシンボリック・クロスリファレンスが生成されます。

アセンブラ・オプションの詳細は、TMS320C54x 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。

2.4 環境変数の使用

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しておくことができます。環境変数は、ユーザが定義し、システム初期化ファイルの文字列に関連付ける特殊なシステム・シンボルです。コンパイラは、このシンボルを使用して、特定のタイプ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、上記のパラメータが自動的に指定されるので、コンパイラを毎回起動することが簡単になります。ツールを起動する場合、環境変数で設定されるデフォルトの多くは、コマンド行オプションを使用して無効にすることができます。

2.4.1 ディレクトリの指定 (C_DIR および C54X_C_DIR)

コンパイラは、C54X_C_DIR および C_DIR 環境変数を使用して、#include ファイルを格納する代替ディレクトリを指定します。シェルは、まず最初に C54X_C_DIR 環境変数を検索し、次にその環境変数を読み込んで処理します。この変数が見つからなかった場合、シェルは C_DIR 環境変数を読み込んで処理します。#include ファイル用のディレクトリを指定するには、次のいずれかのコマンドを使用して C_DIR を設定してください。

O.S.	入力
Windows™	`set C_DIR=`*directory1*`[;`*directory2* `...]`
UNIX	`setenv C_DIR "`*directory1* `[`*directory2* `...]"`

環境変数は、システムをリブートするか、その変数をリセットするまで有効になっています。

2.4.2 デフォルト・シェル・オプションの設定 (C_OPTION および C54X_C_OPTION)

コンパイラ、アセンブラ、およびリンカのシェルのデフォルト・オプションを設定するときに、C54X_C_OPTION または C_OPTION 環境変数を使用すると役立ちます。この方法で設定すると、シェルを実行するたびに、シェルはユーザが C_OPTION で指定したデフォルトのオプションまたはファイル名 (あるいはその両方) を使用します。

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、同じ一組のオプションまたは入力ファイル (あるいはその両方) を使用して、同じシェルを連続して実行したい場合に便利です。シェルは、コマンド行と入力ファイルの読み込みをすべて完了すると、まず C54X_C_OPTION 環境変数を検索し、次にそれを読み込んで処理します。C54X_C_OPTION が見つからなかった場合、シェルは C_OPTION 環境変数を読み込んで処理します。

次の表は、C_OPTION 環境変数の設定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。

O.S.	入力
UNIX with C shell	**setenv C_OPTION "***option*$_1$ [*option*$_2$ . . .]"
Windows™	**set C_OPTION=***option*$_1$[;*option*$_2$ . . .]

環境変数オプションは、コマンド行の場合と同じ方法で指定し、その意味も同じです。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 用のリンクを指定 (−z オプション) したい場合は、C_OPTION 環境変数を次のように設定します。

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

次の例では、コンパイラ・シェルを実行するたびに、リンカが実行されます。コマンド行または C_OPTION の −z の後に指定されているオプションは、すべてリンカに渡されます。したがって、C_OPTION 環境変数を使用してデフォルトのコンパイラ・オプションおよびリンカ・オプションを指定しておき、追加のコンパイラ・オプションやリンカ・オプションをシェルのコマンド行で指定するという方法が可能です。環境変数に −z を設定してあり、コンパイルだけを実行したい場合は、シェルの −c オプションを使用します。次の追加例では、C_OPTION を上記のように設定してあるものと想定しています。

```
cl500  *.c              ; compiles and links
cl500  -c *.c           ; only compiles
cl500  *.c -z lnk.cmd   ; compiles/links using .cmd file
cl500  -c *.c -z lnk.cmd  ;only compiles (-c overrides -z)
```

シェル・オプションの詳細は、2-6 ページの 2.3 節「オプションによるコンパイラの動作の変更」を参照してください。リンカ・オプションの詳細は、4-6 ページの 4.4 節「リンカ・オプション」を参照してください。

2.5 プリプロセッサの制御

この節では、パーサの一部分である C54x プリプロセッサを制御する固有な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。C54x C/C++ コンパイラには、標準の 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)

プリプロセッサは読めば分かるようなエラー・メッセージを出力します。エラーが発生した行番号とファイル名は、診断メッセージとともに表示されます。

2.5.1 事前定義マクロ名

コンパイラは、表 2−2 に示す事前定義マクロ名を保管し、認識します。

表 2−2. 事前定義マクロ名

マクロ名	説明
_TMS320C5XX	1 に展開します ('C54x プロセッサを識別します)。
__LINE__†	カレント行番号に展開します。
__FILE__†	カレント・ソース・ファイル名に展開します。
__DATE__†	*mm dd yyyy* 形式でコンパイル日に展開します。
__TIME__†	*hh:mm:ss* 形式でコンパイル時刻に展開します。
_INLINE	最適化が指定されている場合には、 1 に展開します。それ以外のオプションでは、定義しません。どの最適化であるかに関係なく、−pi が指定されると定義しません。
_C_MODE	コールおよび分岐は、すべて通常の 16 ビット・アドレス範囲内にあることを指定します (デフォルト処理)。
_FAR_MODE	コールおよび分岐は、すべて拡張アドレス空間に対するものであることを指定します ('C548 の拡張アドレッシングに対応)。

† ANSI 規格で定義

表 2−2 のマクロ名は、他の定義名と同じように使用できます。次に、例を示します。

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

この場合、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) C54X_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。
  4) −i オプションで指定されたディレクトリ
  5) C54X_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」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 (C_DIR および C54X_C_DIR)」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* ...]

−i オプション 1 つにつきディレクトリ (directory) を 1 つ指定します。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。

O.S.	入力
Windows	`cl500 -ic:\tools\files source.c`
UNIX	`cl500 -i/tools/files source.c`

2.5.3　前処理リスト・ファイルの生成 (−ppo オプション)

−ppo オプションを指定すると、ソース・ファイルの前処理したバージョンを作成することができます。このファイルには、.pp という拡張子が付きます。コンパイラの前処理では、ソース・ファイルに対して次の処理が行われます。

- ❏ バックスラッシュ (\) で終わっているソース行の次の行との連結
- ❏ 3 文字符号系列の展開
- ❏ 注釈の除去
- ❏ 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継続 (−ppa オプション)

ユーザが前処理を指定した場合、プリプロセッサでは前処理のみが実行されます。デフォルトでは、ソース・コードのコンパイルは行われません。この機能を変更して、ソース・コードの前処理の完了後に引き続きコンパイルも実行するには、他の前処理オプションとともに −ppa オプションも指定してください。たとえば、−ppa を −ppo とともに使用すると、前処理を実行し、前処理出力を .pp 拡張子付きのファイルに書き込み、続いてソース・コードのコンパイルを実行することができます。

2.5.5　コメント付きの前処理リスト・ファイルの生成 (−ppc オプション)

−ppc オプションでは、コメントを除去しないだけで、他のすべての前処理機能が実行され、ソース・ファイルと同じ名前で拡張子が .pp の前処理済みファイルが生成されます。したがって、コメントを残しておきたい場合は、−ppo オプションでなく −ppc オプションを使用してください。

2.5.6 行制御情報付きの前処理リスト・ファイルの生成 (−ppl オプション)

デフォルトでは、前処理出力ファイルにはプリプロセッサ疑似命令は含まれません。#line 疑似命令をインクルードしたい場合は、−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のみを行い、前処理出力を行制御情報 (#line 疑似命令) とともに、ソース・ファイルと同じ名前で拡張子が .pp のファイルに書き込みます。

2.5.7 メイク・ユーティリティ用の前処理出力の生成 (−ppd オプション)

−ppd オプションでは前処理のみが実行されます。ただし、前処理出力は書き込まれず、標準メイク・ユーティリティへの入力に適した従属行のリストが書き込まれます。このリストは、ソース・ファイルと同じ名前で拡張子が .pp のファイルに書き込まれます。

2.5.8 #include 疑似命令によりインクルードされるファイルのリストの生成 (−ppi オプション)

−ppi オプションでは前処理のみが実行されます。ただし、前処理出力は書き込まれず、#include 疑似命令でインクルードされるファイルのリストが書き込まれます。このリストは、ソース・ファイルと同じ名前で拡張子が .pp のファイルに書き込まれます。

2.6 診断メッセージについて

本コンパイラの主要機能の 1 つは、ソース・プログラムに関する診断情報を報告することです。コンパイラは、疑わしい状況を検出すると、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***"**	該当するファイルの名前
line *n***:**	診断対象の行番号
diagnostic severity	診断メッセージの重大度 (それぞれの重大度カテゴリ別の説明が後に続きます)
diagnostic message	問題を示すメッセージの本文

診断メッセージに関連した重大度は次のとおりです。

- **fatal error** (致命的エラー)。コンパイルの続行が不可能な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としては、コマンド行エラー、内部エラー、インクルード・ファイルの欠落などがあります。複数のソース・ファイルをコンパイルしている場合は、問題が発生した現在のファイルより後のファイルはコンパイルされません。
- **error** (エラー)。これは、C/C++ 言語の構文またはセマンティック規則に違反していることを示します。コンパイルは続行されますが、オブジェクト・コードは生成されません。
- **warning** (警告)。有効では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・コードも生成されます (エラーが検出されなかった場合)。
- **remark** (要注意)。これは警告より軽いものです。有効であって意図的に指定された可能性はあるが、チェックが必要なものがあることを示します。コンパイルは続行され、オブジェクト・コードも生成されます (エラーが検出されなかった場合)。デフォルトでは、要注意は発行されません。要注意が発行されるようにするには、−pdr シェル・オプションを使用します。

診断は、次の例のような形式で、標準エラー出力に書き込まれます。

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

デフォルトでは、ソース行は省略されます。ソース行とエラー位置の表示を有効にするには、−pdv シェル・オプションを使用します。上記の例では、このオプションを使用しています。

上の例のメッセージには、診断の対象になったファイルと行が示され、そのメッセージの後にソース行そのものが続きます (シンボル「^」で位置が示されます)。複数の診断が 1 つのソース行に適用された場合は、個々の診断がそれぞれ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、そのつど該当の位置が示されます。

長いメッセージは必要に応じて折り返され、複数行に表示されます。

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要求することができます。診断識別子を表示した場合、この識別子には、その診断の重大度がコマンド行で無効にできるものかどうかも示されます。無効にできる場合は、診断識別子にサフィックス −D (*discretionary* : 自由裁量) が含まれています。そうでない場合は、サフィックスは含まれていません。次に例を示します。

```
"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はないこともあります。警告と要注意の場合は、すべて自由裁量です。

一部のメッセージでは、エンティティ (関数、ローカル変数、ソース・ファイルなど) のリストが役立ちます。エンティティは、最初のエラー・メッセージに続いてリストされます。

```
"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);
    ^
```

追加のコンテキスト情報が表示される場合もあります。コンテキスト情報が特に役立つのは、テンプレート・インスタンス化の実行中、またはコンストラクタ、デストラクタ、代入演算子関数の生成中に、フロント・エンドが診断メッセージを発行した場合です。次に例を示します。

```
"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ければ、エラーが何を指しているかを判断することは困難です。

2.6.1 診断の制御

本コンパイラが提供する診断オプションを使用して、パーサでコードを解釈する方法を変更することができます。診断の制御に使用できるオプションには次のものがあります。

−pdel*num*
エラー数の上限を *num* に設定します。*num* には、任意の 10 進数を指定できます。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共に表示します。このオプションを使用して、診断抑止オプション (−pds、−pdse、−pdsr、−pdsw) に指定する必要のある引数を判別することができます。

このオプションには、診断が自由裁量かどうかも示されます。自由裁量の診断とは、重大度を無効にできる診断です。自由裁量診断には、サフィックス −D が含まれています。それ以外の場合は、サフィックスは表示されません。詳細は、2.6 節「診断メッセージ」を参照してください。

−pdf
診断情報ファイルを生成します。このファイルは、対応するソース・ファイルと同じ名前ですが、拡張子は *.err* です。

−pdr
要注意 (軽い警告) を発行します。これは、デフォルトでは抑止されます。

−pds*num*
num で識別される診断を抑止します。診断メッセージの数値識別子を判別するには、別のコンパイルで先に −pden を使用します。その後で、−pds *num* を使用して、診断を抑止します。抑止できるのは、自由裁量診断だけです。

−pdse*num*
num で識別される診断をエラーとして分類します。診断メッセージの数値識別子を判別するには、別のコンパイルで先に −pden を使用します。その後で、−pdse *num* を使用して、その診断をエラーとして再分類します。変更できるのは、自由裁量診断の重大度だけです。

−pdsr*num*
num で識別される診断を要注意として分類します。診断メッセージの数値識別子を判別するには、別のコンパイルで先に −pden を使用します。その後で、−pdsr *num* を使用して、その診断を要注意として再分類します。変更できるのは、自由裁量診断の重大度だけです。

−pdsw*num*
num で識別される診断を警告として分類します。診断メッセージの数値識別子を判別するには、別のコンパイルで先に −pden を使用します。その後で、−pdsw *num* を使用して、その診断を警告として再分類します。変更できるのは、自由裁量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	診断の警告メッセージを抑止します (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、コンパイラから発行される診断メッセージをどのように制御できるかを示しています。

次のコード・セグメントを見てください。

```
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回避するために各 case ブロックの終わりに break 文を組み込むのはプログラミングの常套手段なので、上記の 2 つの警告は無視できます。−pden オプションを使用することで、これらの警告の診断識別子を調べることができます。結果は次のようになります。

```
[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して、この警告を要注意として取り扱うことを指定します。これで、(要注意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。

この種の制御は便利ですが、非常に危険でもあります。コンパイラから発行されるメッセージの中には、明白とは言えないまでも問題のある状況を示すものも少なくありません。抑止オプションを使用する前に、発行されたすべての診断メッセージを慎重に解析してください。

2.6.3 その他のメッセージ

ソースに関係のないその他のエラー・メッセージ(コマンド行構文が正しくない、指定されたファイルが見つからないなど)は、一般に致命的エラーです。このようなエラーの場合は、メッセージの前に >> 記号が付いています。

次に例を示します。

```
cl500 –j
>> invalid option –j (ignored)
>> 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・オプションを使用すると、クロスリファレンス・リスト・ファイル (拡張子 .crl) を生成することができます。このファイルには、ソース・ファイル内の各識別子に対する参照情報が含まれています (−px オプションは、−ax とは別のものです。−ax は、シェル・オプションではなく、アセンブラ・オプションです)。クロスリファレンス・リスト・ファイルの中の情報は、次の形式で表示されます。

sym-id name X filename line number column number

sym-id 各識別子に一意的に割り当てられた整数

name 識別子名

X 次の値のいずれかです。

X の値	意味
D	定義
d	宣言 (定義ではありません)。
M	修正
A	取得アドレス
U	使用
C	変更 (1 つのオペレーションの中で使用され修正されました)
R	その他の参照
E	エラー (参照が不確定です)

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・ファイル内の桁番号

2.8 ロー・リスト・ファイルの生成 (−pl オプション)

−pl オプションを使用すると、ロー・リスト・ファイル (拡張子 .rl) を生成することができます。このファイルは、ソース・ファイルをコンパイラがどのように前処理するかを理解するために役立ちます。前処理リスト・ファイル (−ppo、−ppc、−ppl の各プリプロセッサ・オプションで生成されたファイル) にはソース・ファイルの前処理済みバージョンが示されるのに対して、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。

- ❏ 個々のオリジナル・ソース行
- ❏ インクルード・ファイルへの移動、およびインクルード・ファイルからの移動
- ❏ 診断結果
- ❏ 重要な処理が実行された場合、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・ファイルの各ソース行の先頭には、表 2−3 に示す識別子のいずれかが付いています。

表 2−3. 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続けて表示されます。
S	スキップされたソース行 (偽の #if 文節)
L	ソース位置の変更。これは次の形式で表示されます。 L *line number filename key* *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、変更の原因がインクルード・ファイルの入口または出口にある場合のみです。*key* の値は次のどちらかです。 1 = インクルード・ファイルの開始 2 = インクルード・ファイルの終了

−pl オプションには、表 2−4 に定義されている診断識別子も含まれています。

表 2−4. ロー・リスト・ファイルの診断識別子

診断識別子	定義
E	エラー
F	致命的エラー
R	要注意
W	警告

診断のロー・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 2–4 に示す識別子の 1 つで、診断の重大度を表します。
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・ファイル内の桁番号
diagnostic	診断結果を示すメッセージ・テキスト

ファイル終わりの後の診断は、列番号 0 を付け、ファイルの最終行として示されます。診断メッセージ・テキストに複数の行が必要な場合は、各後続行に、同じファイル、行、列の情報が含まれますが、診断識別子は小文字で表示されます。診断メッセージの詳細は、2.6 節「診断メッセージ」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。この機能は、次の 2 つ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。

インライン展開は、次のいずれかの方法で行われます。

- ❑ 組み込み演算子を指定してインライン展開を実行します。(組み込み関数は常にインライン展開します)。
- ❑ 自動でインライン展開を実行します。
- ❑ 保護されない inline キーワードを指定して、定義制御されたインライン展開を実行します。
- ❑ 保護された inline キーワードを指定して、定義制御されたインライン展開を実行します。

注: 関数をインライン展開すると、コード･サイズが大幅に増えます。

呼び出される回数が非常に多い関数をインライン展開すると、コード･サイズが増えます。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、呼び出される回数は多いものの、サイズが小さな関数に向いた処理です。

2.9.1 インライン組み込み演算子

C54x には、多数の組み込み演算子があります。コンパイラは、組み込み演算子を有効なコード (通常はひとつの命令) で置き換えます。この「インライン展開」は、オプティマイザを使用してもしなくても自動的に発生します。

組み込み演算子の詳細、および演算子のリストは、6-22 ページの 6.5.4 項「組み込み演算子を使用してアセンブリ言語文にアクセスする方法」を参照してください。

インラインを展開する関数は、次のとおりです。

- ❑ abs
- ❑ labs
- ❑ fabs
- ❑ _assert
- ❑ _nassert
- ❑ memcpy

2.9.2 自動インライン展開

−o3 オプションを指定して C/C++ ソース・コードをコンパイルすると、小さな関数に対してインライン関数展開が実施されます。詳細は、3-12 ページの 3.6 節「自動インライン展開 (−oi オプション) を参照してください。

2.9.3 無保護の定義制御されたインライン展開

inline キーワードは、標準の呼び出し手順を使用せずに、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 キーワードで宣言されている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、−o オプション (−o0、−o1、−o2、または −o3) のいずれかを指定してオプティマイザを起動する必要があります。−o3 を使用すると、自動インライン展開も同時にオンになります。

次に inline キーワードの使用例を示します。この例では、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 キーワードの使用例

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

−pi オプションを使用すると、定義制御されたインライン展開がオンになります。このオプションは、特定のレベルの最適化は必要だが、定義制御されたインライン展開は必要ではないという場合に便利です。

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静的インライン関数として宣言する場合は、−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実行されない場合には、コード・サイズが増加する危険性が潜在します。これを避けるためには、追加の手順が必要になります。

インライン展開をオフにしたときに、ヘッダ・ファイル内の静的インライン関数が原因でコード・サイズが増加するのを防止するには、以下で説明する手順を使用します。この手順を実行することで、インライン展開がオフにされたときの外部結合が可能になり、したがってオブジェクト・ファイル全体を通じて存在する関数定義が 1 つだけになります。

- ❑ 静的インライン関数バージョンのプロトタイプを作成します。次に代替プロトタイプとして、同じ関数の静的でない外部とリンク可能なバージョンを作成します。例 2−2 に示すように、_INLINE プリプロセッサ・シンボルを指定して、これらの 2 つのプロトタイプを条件付きで前処理します。
- ❑ 例 2−2 に示すように、同一バージョンの関数定義を入れた .c ファイルまたは .cpp ファイルを作成します。

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、プロトタイプが静的インラインとして宣言されるのは、_INLINE が真の場合のみです (オプティマイザが使用され、かつ −pi が指定されていなければ、_INLINE が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 バージョンが存在している状態が確保されます。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) 2001 Texas Instruments Incorporated                        */


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

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

; . . .
_ _INLINE size_t  strlen(const char *s);
; . . .

#ifdef _INLINE

/*************************************************************************/
/*  strlen                                                               */
/*************************************************************************/
static inline size_t strlen(const char *s)
{
   register const char *rstr = string;
   register size_t n = 0;
   while (*rstr++) ++n;
   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) 2001  Texas Instruments Incorporated                     */
#undef _INLINE
#include <string.h>

size_t strlen(const char *string)
{
   register const char *rstr = string;
   register size_t n = 0;

   while (*rstr++) ++n;
   return (n);
}
```

2.9.5 インライン展開の制限

自動インライン展開の場合も定義制御されたインライン展開の場合も、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限があります。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、静的インラインとして宣言されたものは展開されます。関数を静的インラインとして宣言した場合は、ローカルな静的変数が存在していても、展開が行われます。再帰関数やノンリーフ関数の場合は、インライン展開の深さにも制限があります。また、インライン展開を使用するのは、小さい関数や呼び出し箇所が少ない関数に限るようにしてください （ただしこれはコンパイラの必須条件というわけではありません)。

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 パラメータまたは union パラメータが指定されている。
- ❏ volatile パラメータが指定されている。
- ❏ 可変長の引数リストがある。
- ❏ struct 型、union 型、または enum 型が宣言されている。
- ❏ 静的変数が含まれている。
- ❏ 揮発性変数が含まれている。
- ❏ 再帰的である。
- ❏ プラグマが含まれている。
- ❏ 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･ユーティリティの使用方法

コンパイラ・ツールには、C/C++ ソース文をコンパイラのアセンブリ言語出力に差し込むインターリスト・ユーティリティが含まれています。このユーティリティを使用すると、各 C/C++ ソース文に対して生成されたアセンブリ・コードを検査することができます。インターリスト・ユーティリティは、オプティマイザの使用の有無、および指定するオプションによって、動作が異なります。

インターリスト・ユーティリティ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・ユーティリティを実行するには、次のように入力します。

```
cl500 -ss function
```

−ss オプションは、シェルに差し込み後のアセンブリ言語出力ファイルを削除しないよう指示します。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・ユーティリティを起動すると、インターリスト・ユーティリティはコード・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・ユーティリティは、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組み合わせ、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・ユーティリティを使用する方法の詳細は、3-13 ページの 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」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                                               *
;**********************************************************************
_main:
          FRAME  #-3
          NOP
;----------------------------------------------------------------------
;   5 | printf("Hello World\n");
;----------------------------------------------------------------------
          ST     #SL1,*SP(0)
          CALL   #_printf
          ; call occurs [#_printf]   ;
          FRAME  #3
          RET
; return occurs
```

第 3 章

コードの最適化

コンパイラ・ツールには、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の作業を実行することによって C/C++ プログラムの実行速度を速くし、サイズを小さくする最適化プログラムが組み込まれています。

本章では、オプティマイザの起動方法と、オプティマイザを使用したときに実行される最適化について説明します。また、インターリスト・ユーティリティをオプティマイザと組み合わせて使用する方法、最適化されたコードを認識したり、またはデバッグする方法についても説明します。

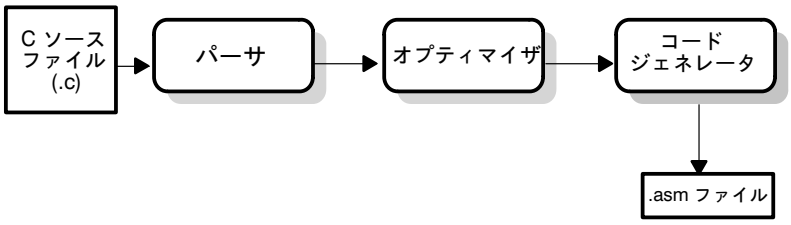
項目 **ページ**

3.1 オプティマイザの使用方法

C/C++ コンパイラは、さまざまな最適化処理を実行することができます。高レベルの最適化はオプティマイザで実行され、ターゲット固有の低レベルの最適化はコード・ジェネレータで実行されます。

高レベルのオプティマイザは、パーサとコード・ジェネレータの間の独立したパスとして実行されます。図 3−1 は、独立したオプティマイザによるコンパイラの実行の流れを示しています。

図 3−1. オプティマイザによる C プログラムのコンパイル



オプティマイザを起動するには、cl500 シェル・プログラムを使用するのが最も簡単です。その場合、cl500 の コマンド行で *–on* オプションを指定します。*n* は最適化のレベル (0、1、2、および 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、次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、次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かった場合は、−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、次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 を使用する場合は、3-4 ページの 3.2 項「−o3 オプションの使用方法」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って行われます。これ以外にも、コード・ジェネレータによって別の最適化が実行されます。特に、プロセッサ固有の最適化は、オプティマイザの起動とは無関係に実行されます。これらの最適化は常に実施可能であり、最適化の選択レベルの影響も受けません。

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に指示します。−o3 オプションを単独で使用して、一般的なファイル・レベルを実行することも、他のオプションと組み合わせて、さらに詳細な最適化を実行することもできます。表 3−1に示したオプションと −o3 を一緒に使用して、表中に記述する最適化を行います。

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ためのファイルがある	–ol*n*	3-4
最適化情報ファイルを作成する	–on*n*	3-5
複数のソース・ファイルをコンパイルする	–pm	3-6

3.2.1 ファイル・レベルの最適化の制御 (−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、一部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、または 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

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が宣言されている	–ol0
標準ライブラリ関数と同一のライブラリ関数の定義を含むが、標準ライブラリ内で宣言された関数を変更しない	–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場合。−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した場合、–on オプションを指定すれば、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、または 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

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場合。–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 オプションと −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、3-5 ページの 3.2.2 項「最適化情報ファイルの作成 (−on オプション)」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)

−op オプションを使用すると、−pm −o3 を指定して起動したプログラム・レベルの最適化を制御できます。特に −op オプションは、他のモジュール内の関数がモジュールの外部関数を呼び出せるか、または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、この情報と独自のファイル・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、静的と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

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を含む	–op0
他のモジュールから呼び出される関数を含んでいないが、他のモジュール内で変更されるグローバル変数を含む	–op1
他のモジュールから呼び出される関数も、他のモジュール内で変更されるグローバル変数も含んでいない	–op2
他のモジュールから呼び出される関数は含んでいるが、他のモジュール内で変更されるグローバル変数は含んでいない	–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禁止にする場合があります。表 3−5 は、コンパイラが別の −op レベルに戻す原因となる条件と −op レベルの組み合わせの一覧です。

表 3−5.　−op オプションを使用する場合の特別な注意事項

−op の指定...	条件	−opレベル
指定なし	−o3 の最適化レベルが指定された	デフォルトの −op2 になる
指定なし	コンパイラが −o3 最適化レベルにある外部関数の呼び出しを検出した	−op0 に戻る
指定なし	main が定義されていない	−op0 に戻る
−op1 または −op2	エントリ・ポイントとして定義される main を含む関数がない	−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た関数がない	−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 プラグマを使用しなければなりません。これらの状況については、3-8 ページの 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」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されません。コンパイラはアセンブリ・コードによる C/C++ 関数の呼び出しや C 変数の変更を認識しないため、−pm オプションを指定すると、このような C/C++ 関数は最適化されてしまいます。それらの関数を保持するには、それらの関数の宣言や参照の前に FUNC_EXT_CALLED プラグマ (5-19 ページの 5.8.4 項「FUNC_EXT_CALLED プラグマ」を参照) を配置します。

プログラムの中でアセンブリ関数が使用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 (3-6 ページの 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)」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次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++ 関数を呼び出しも 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、コンパイラの最適化レベルは、デフォルトの −op2 から −op0 に戻ってしまいます。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が、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ためです。

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、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って変更される可能性がある変数に volatile キーワードを付加し、−pm −o3 −op2 を指定してコンパイルします(詳細は、5-11 ページの 5.4.5 項「volatile キーワード」を参照)。

−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・コードとアセンブリ・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 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。それらの C 関数は main と同じに機能し、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 キーワードを付加します。その後、次のいずれ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、アセンブリ言語の割り込み関数から呼び出されるすべてのエントリ・ポイント関数に FUNC_EXT_CALLED プラグマを適用し、その後、−pm −o3 −op2 を指定してコンパイルします。必ず、すべてのエントリ・ポイント関数にこのプラグマを使用してください。そうしないと、コンパイラは先頭に FUNC_EXT_CALLED プラグマ が指定していないエントリ・ポイント関数を除去します。
- −pm −o3 −op3 を指定してコンパイルします。FUNC_EXT_CALLED プラグマ を指定しないので、−op3 オプションを使用しなければなりません。このオプションは −op2 ほど強力ではなく、最適化があまり効果的でない場合もあります。

追加オプションを指定せずに −pm −o3 を使用すると、アセンブリ関数から呼び出される C 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 プラグマ については 5-19 ページの 5.8.4 項を、−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もあります。コンパイラによる最適化で asm 文が対象外となることはありませんが(その文がまったく処理できない場合を除いて)、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。通常は、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++ 環境とインターフェイスしたり C/C++ 変数にアクセスしたりすると予期しない結果を生じるおそれがあります。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 最適化コード内のエイリアス付き変数にアクセスする場合の注意

複数の方法で 1 つのオブジェクトにアクセスすると(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)、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妨害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る可能性があるからです。オプティマイザは、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。そして、プログラムの正確さを保ちながら、できる限りプログラムを最適化します。オプティマイザには、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す可能性が少しでもあれば、それらのポインタが同じオブジェクトを指すと見なされます。

オプティマイザは、アドレスが引数として関数に渡される変数は、 呼び出し先の関数で設定されるエイリアスによってその後変更されることはないと想定します。次の例が含まれます。

- 関数からアドレスを戻す
- グローバル変数にアドレスを割り当てる

このようなエイリアスを使用する場合は、 −ma シェル・オプションを使用してコードを最適化する必要があります。たとえば、次のようなコードの場合には、−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*size* オプション)

−op3 オプションを指定してオプティマイザを起動すると、小さな関数は自動的にインライン展開されます。コマンド行オプションである−oi*size* には、インライン展開する関数のサイズのしきい値を指定します。*size* のしきい値より大きい関数は、自動的にインライン展開しません。−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が抑止されます。
- ❏ *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 *size* のしきい値を使用して、自動的にインライン展開する関数のサイズを制限します。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、結果が 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、インライン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かった場合でも、オプティマイザは非常に小さな関数しかインライン展開しません。

注： −o3 による最適化とインライン展開

自動インライン展開をオンにするには、−o3 オプションを使用する必要があります。−o3 オプションは、その他の最適化もオンにします。−o3 による最適化を実行したいが、自動インライン展開は行いたくないという場合は、−o3 オプションとともに −oi0 も指定してください。

注： インライン展開とコード・サイズ

関数をインライン展開すると、コード・サイズが増加します。特に、頻繁に呼び出される関数をインライン展開する場合はサイズがかなり大きくなります。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サイズが小さな関数に向いた処理です。インライン展開によるコード・サイズの増加を防ぐためには、−oi0 オプションと −pi オプションを使用します。これらのオプションを指定すると、コンパイラでインライン展開されるのは組み込み関数だけになります。

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 (−on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、オプティマイザ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 オプションを共に使用すると、オプティマイザのコメントとオリジナルの C/C++ ソースがアセンブリ・コードに差し込まれます。

−os オプションをオプティマイザと組み合わせて使用すると、インターリスト・ユーティリティは 1 つの独立したパスとしては実行されません。その代わりに、オプティマイザが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、差し込まれません。

インターリスト・ユーティリティは、C/C++ の文の境界にまたがって実施される一部の最適化を不可能にするため、最適化後のコードに影響を及ぼす場合があります。オプティマイザはプログラムの配置を大幅に変更するので、最適化を行うと、正常なソースの差し込みができなくなります。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、実行中のアプリケーションの正確な構文を反映していないことがあります。

例 3−1 は、オプティマイザ (−o2) と −os オプションを指定して 2-41 ページの例 2−3 の関数をコンパイルした結果を示しています。アセンブリ・ファイルの中に、オプティマイザのコメントがアセンブリ・コードで差し込まれていることに注意してください。

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");
;**     -------------------- return;
       FRAME     #-3
       NOP
       ST        #SL1,*SP(0)
       CALL      #_printf
       ;call occurs [#_printf]
       FRAME     #3
       RET
       ;return occurs
```

−ss オプションと −os オプションをオプティマイザと組み合わせて使用すると、オプティマイザはコメントを差し込み、コード・ジェネレータとアセンブラの間でインターリスト・ユーティリティが実行されることにより、元の C/C++ ソースとアセンブリ・ファイルがマージされます。

例 3−2 に、2-41 ページの例 2−3 の関数を、オプティマイザ(−o2) と −ss および −os オプションを指定してコンパイルした結果を示します。アセンブリ・ファイルの中で、アセンブリ・コードにオプティマイザのコメントと C ソースが差し込まれていることに注意してください。

例 3−2. 例 2−3 の関数を−o2、−os、および −ss オプションを指定してコンパイルした結果

```
_main:
;** 5  -------------------- printf((char*)”Hello world\n”);
;**    -------------------- return;
       FRAME     #-3
       NOP
;----------------------------------------------------------
; 5 | printf(”Hello World\n”);
;----------------------------------------------------------
       ST        #SL1,*SP(0)
       CALL      #_printf
       ;call occurs [#_printf]
       FRAME     #3
       RET
       ;return occurs
```

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、オプティマイザが大幅なコードの再配置を行うほか、多数の変数を多数のレジスタに割り振るため、ソース・コードとオブジェクト・コードを相関させることが難しくなるからです。この問題を軽減するためには、次の 2 つのうちどちらかを選択します。

- −g オプションと−o オプションを使用します。−g オプションを使用すると、C ソース・レベル・デバッガで使用するシンボリック・デバッグ疑似命令を生成できますが、コード・ジェネレータでの最適化の多くが適用できなくなります。−o オプション (オプティマイザを起動する) を −g 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。
- −g、−o、および −mn オプションを使用する。−mn オプションでは、−g によって無効にされた最適化を再び有効にすることができます。したがって、−g と −o のみを使用する場合より、最適化されたコードのデバッグがさらに容易になります。

どちらの代替方法を使用した場合も、デバッガの一部の機能については信頼性が保証されなくなります。

3.9 実行できる最適化の種類

TMS320C54x C/C++コンパイラは、さまざまな最適化の技法を使用して C/C++ プログラムの実行速度を向上させ、サイズを小さくします。コンパイラ全体を通じて、さまざまなレベルで最適化が実行されます。

ここで述べるほとんどの最適化は、独立した最適化パスによって実行されます。ユーザは、−o コンパイラ・オプション (3-2 ページの 3.1 項を参照) を指定してこのパスを有効にし、制御することができます。ただし、コード・ジェネレータによって行われる最適化を、選択して有効にしたり抑止したりすることはできません。

コンパイラによって行われる最適化は、次のとおりです。

3.9.1 コストに基づいたレジスタ割り当て

オプティマイザは、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、他の変数より優先するように加重されます。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振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動作を解析し、操作を直線的な順序 (基本ブロック) に再配置して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。いくつかの単純な制御フロー構造は条件付き命令に簡略化され、分岐の必要が完全になくなります。

例 3−3 では、この単純な有限状態機械の switch 文と state 変数が最適化された後、除外され、一連の簡潔な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 コンパイラの出力

```
; opt500 –o3 control.if control.opt

_fsm:
   PSHM   AR1
   LD     *AR1+,A
   BC     L2,ANEQ
   ;branch occurs
L1:
   LD     *AR1+,A
   BC     L2,AEQ
   ;branch occurs
   LD     *AR1+,A
   BC     L1,AEQ
   ;branch occurs
L2:
   LD     *AR1+,A
   BC     L2,AEQ
   ;branch occurs
   POPM   AR1
   RET
   ;return occurs
```

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。3-18 ページの例 3−3、および 3-20 ページの例 3−4 を参照してください。

- **共通部分式の除去**

  複数の式が同じ値を生成する場合、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(例 3−4 を参照)。

- **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);
   return b;
}
```

(b) C コンパイラの出力

```
; opt500 –o2 data.if data.opt

_simplify:
    STLM   A,T
    RETD
    MPY    #5,A
    ; return occurs
```

3.9.6 ランタイムサポート・ライブラリ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、コンパイラはインライン・ライブラリの中から C 関数 plus( ) に対応するコードを検出し、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 コード

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

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

(b) C コンパイラの出力

```
; opt500 -o3 inline.if inline.opt
   .sect ".text"
   .global _plus
;***************************************************
;* FUNCTION NAME: _plus                            *
;***************************************************
_plus:
   LD *SP(1), B
   ADD*(BL), A
   RET

   .sect ".text"
   .global _main
;***************************************************
;* FUNCTION NAME: _main                            *
;***************************************************
_main:
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> ENTERING plus()
; <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< LEAVING plus()
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> ENTERING plus()
; <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< LEAVING plus()
   RETD
   LD        #12,A
   NOP
   ;return occurs
```

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、ポインタのインクリメントによって配列を参照するコードに置換されます。

カウンタのインクリメントによって制御されるループは、ブロックのリピートとして書き込まれるか、効率的なデクリメント分岐命令を使用して書き込まれます。誘導変数の解析と強度換算を同時に行うと、多くの場合、ループ制御変数へのすべての参照は除去され、ループ制御変数全体を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計算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外されます。

3.9.10 ブロックのリピート

C54x は、RPTB (リピート・ブロック) 命令による 0 オーバーヘッドのループをサポートしています。オプティマイザを使用すると、コンパイラはカウンタによって制御されるループを検出でき、効率的なリピート形式を使用したループを生成できます。反復の回数は、定数と式のどちらでもかまいません。

誘導変数の除去とループ・テスト置換を使用すると、コンパイラは、ループを単純なカウントをするループとして認識でき、その後、リピート・ブロックを生成できます。強度変換は、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。

3.9.11 遅延、分岐、コール、およびリターン

C54x には、多くの遅延分岐命令、遅延コール命令、および遅延リターン命令があります。このうちの無条件分岐 (BD)、名前付き関数のコール (CALLD)、および単純リターン (RETD) の 3 つの命令は、コンパイラによって使用されます。これらの命令は、非遅延のものより 2 サイクル少ないサイクルで実行されます。これらの命令は、命令ストリームに入った後、2 つの命令ワードを実行します。シーケンスの正しい動作を保証するためには、遅延命令の後にNOP 命令を挿入しなければならない場合もあります。これにより、コードは非遅延シーケンスより 1 ワード長くなりますが、それでも 1 サイクル速く実行されます。遅延命令を実行する命令シーケンスには、コンパイラによってコメントが挿入されることに注意してください。3-24 ページの例 3−6 を参照してください。

例 3−6.　遅延分岐命令、遅延コール命令、および遅延リターン命令

(a) C ソース

```
main()
{
   int i0, i1;
   while(input(&i0) && input (&i1))
      process(i0, i1);
}
```

(b) コンパイラの出力

```
; opt500 –o2 delay.if delay.opt
_main:
         BD       L2
         PSHM     AR1
         FRAME    #–4
         ;branch occurs
L1:
         LD       *SP(3),A
         STL      A,*SP(0)
         LD       *SP(2),A
         CALL     #_process
         ;call occurs [#_process]
L2:
         LDM      SP,A
         CALLD    #_input
         ADD      #2,A
         ;call occurs [#_input]
         LD       *(AL),A
         BC       L3,AEQ
         ;branch occurs
         LDM      SP,A
         CALLD    #_input
         ADD      #3,A
         ;call occurs [#_input]
         STLM     A,AR1
         NOP
         NOP
         BANZ     L1,*AR1
         ;branch occurs
L3:
         FRAME    #4
         POPM     AR1
         RETD
         LD       #0,A
         NOP
         ;return occurs
```

3.9.12 代数の並べ換え／シンボルの簡略化／定数の畳み込み

計算を最適に行うために、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。たとえば、(a + b) − (c + d) という式の計算には 6 つの命令を必要としますが、4 つの命令で済む((a + b) − c) − d に最適化できます。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。3-20 ページの例 3−4 を参照してください。

第 4 章

C/C++ コードのリンク

TMS320C54x ™ 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。
- ❑ cl500 を使用して、コンパイルとリンクを 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。

この章では、それぞれの方法によるリンカの起動方法について説明します。また、C/C++ コードのリンクに関する特別な要件、たとえば、ランタイムサポート・ライブラリの取り込み方法、初期化モデルの指定方法、プログラムをメモリに割り当てる方法についても説明します。リンカの詳細は、TMS320C54x 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 リンカを 1 つの独立したプログラムとして起動する方法

この節の例は、プログラムをコンパイルまたはアセンブルしたあと、独立したステップとしてリンカを起動する方法を示しています。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。

lnk500 {**–c**|**–cr**} *filenames* [*options*] [**–o** *name*.out] –l *library* [*lnk.cmd*]

lnk500	リンカを起動するコマンドです。
–c | –cr	C/C++ 環境で定義した特別規則の使用をリンカに伝えるオプションです。lnk500 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、リセット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には、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、出力ファイル名はデフォルトの a.out になります。
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。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、4.4節「リンカ・オプション」で説明します)。
–o *name.out*	–o オプションは出力ファイルを指定します。
–l *libraryname*	(小文字の L)。C/C++ ランタイムサポート関数と浮動小数点算術関数が含まれている、適切なアーカイブ・ライブラリを指定します (–l オプションは、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指示します)。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・ライブラリを指定してある場合は、このパラメータを指定する必要はありません。
lnk.cmd	リンカ用のオプション、ファイル名、疑似命令、またはコマンドが含まれます。

ライブラリをリンカへの入力とし て指定した場合、リンカは未定 義の参照を解決するライブラリ・メンバのみを組み込み、リンクします。たとえば、prog1.obj、prog2.obj、および prog3.obj の各モジュールから構成されている C/C++ プログラムをリンクするには (出力ファイルの名前は prog.out)、次のように入力します。

```
lnk500 –c prog1 prog2 prog3 –o prog.out –l rts.lib
```

リンカは、デフォルトの割り当て アルゴリズムを使用して、プロ グラムをメモリに割り当てます。リンカ・コマンド・ファイルの中で MEMORY と SECTIONS の疑似命令を使用すると、割り当てプロセスをカスタマイズできます。詳細は、TMS320C54x アセンブリ言語ツール ユーザーズ・マニュアル を参照してください。

4.2 コンパイラ・シェルでリンカを起動する方法 (–z オプション)

この節で説明するオプションとパラメータは、両方のリンク方法に適用できます。ただし、シェルを使用してリンクする場合は、–z シェル・オプションの後ろにこのオプションを指定します (2-4 ページの 2.2 節「C コンパイラ・シェルの起動」を参照)。

デフォルトでは、コンパイラはリンカを実行しません。しかし、–z オプションを使用した場合は、プログラムのコンパイル、アセンブル、およびリンクが 1 ステップで実行されます。–z を使用してリンクを使用可能にする場合は、次の点に注意してください。

- ❑ –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、コマンド行のすべてのソース・ファイルとコンパイラ・オプションの後ろに指定するか、C_OPTION 環境変数で指定しなければなりません。

コマンド行の–z の後ろの引数はすべて、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。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し、リンクするには、次のように入力します。

```
cl500 –sq *.c –z c.cmd –o prog.out –l rts.lib
```

まず、カレント・ディレクトリ内にある拡張子 .c が付いているすべてのファイルが、–s オプション (C コードをアセンブリ・コードに差し込む) と –q オプション (静的モードで実行) を使用してコンパイルされます。次に、リンカが c.cmd コマンド・ファイルを使用してコンパイルされたオブジェクト・ファイルをリンクします。–o オプションは出力ファイルの名前を指定し、–l オプションはランタイムサポート・ライブラリの名前を指定します。

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

1) コマンド行から入力されたオブジェクト・ファイル名
2) コマンド行の –z オプションの後ろにある引数
3) C_OPTION または C54X_C_OPTION 環境変数の –z オプションの後ろにある引数

4.3 リンカを無効にする方法 (−c シェル・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 C54X_C_OPTION 環境変数の中で −z オプションを指定しているとき、コマンド行で −c オプションを使用してリンクを選 択的に無効にしたい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 C/C++ のリンク規則 (実行時の変数の自動初期化) を使用するよう指示します。リセット時に変数を自動初期化する場合は、−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。

4.4 リンカ・オプション

–z オプションの後ろに指定したすべてのコマンド行は、パラメータおよびオプションとしてリンカに渡されます。リンカを制御するオプションと、その効果についての詳しい説明を次に示します。

–a	絶対的な実行可能モジュールを生成します。これはデフォルトです。–a と –r をどちらも指定しなかった場合、リンカは –a が指定された場合と同じ動作を取ります。
–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。
–b	シンボリック・デバッグ情報のマージを抑止します。
–c	実行時に変数を自動初期化します。
–cr	リセット時に変数を自動初期化します。
–e *global_symbol*	出力モジュールの一次エントリ・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の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も、global_symbol だけは、グローバル・シンボルとして定義します。
–h	すべてのグローバル・シンボルを静的シンボルにします。
–heap *size*	ヒープ・サイズ (動的メモリ割り当て) を size で指定したワード数に設定し、ヒープ・サイズを指定するグローバル・シンボルを定義します。デフォルトは 1K ワードです。
–i*directory*	ライブラリ検索アルゴリズムを変更し、デフォルトの位置を検索する前に directory で指定したディレクトリを検索するようにします。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。
–j	条件付きリンクを抑止します。
–k	入力セクション内の位置合わせフラグを無視します。
–l *filename*	l (Lの小文字) は、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・ライブラリの名前であり、オペレーティング・システムの規則に従ったものでなければなりません。
–m *filename*	ホールを含む入出力セクションのマップまたはリストを生成し、filename で指定したファイルにそのリストを格納します。ファイル名はオペレーティング・システムの規則に従ったものでなければなりません。

–o *filename*	実行可能な出力モジュールの名前を指定します。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。
–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*	C システム・スタック・サイズを size で指定したワード数に設定し、スタック・サイズを指定するグローバル・シンボルを定義します。デフォルトは 1K ワードです。
–u *symbol*	symbol で指定した未解決の外部シンボルを出力モジュールのシンボル・テーブルに格納します。
–v*n*	出力 COFF フォーマットを指定します。n は 0、1、2 のいずれかです。デフォルトのフォーマットは COFF2 です。
–w	未定義の出力セクションが作成されると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。

リンカ・オプションの詳細は、TMS320C54x 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。

4.5 リンク・プロセスの制御

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、次のような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
- ❏ 初期化モデルを指定する
- ❏ プログラムをメモリに割り振る方法を決定する

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、TMS320C54x 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。

4.5.1 ランタイムサポート・ライブラリとのリンク

すべての C/C++ プログラムは、ランタイムサポート・ライブラリとリンクさ せる必要があります。このライブラリには、標準 C/C++ 関数のほか、本コンパイラが C/C++ 環境を管理するために使用する各種の関数も含まれています。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・オプションを使用します。 −l オプションはリンカに対して、−i オプションを調べ、次に C_DIR 環境変数も調べて、アーカイブ・パスまたはオブジェクト・ファイルを検出するように指示します。

−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。

lnk500 {**−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してください。なぜなら、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・ファイルがあると、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行われます。ライブラリをリンカへの入力として指定した場合、リンカは常に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。

4.5.2 ランタイム初期化

C/C++ プログラムはすべて、boot.obj というオブジェクト・モジュールとリンクする必要があります。C/C++ プログラムは、実行を開始するときは、最初に boot.obj を実行します。boot.obj ファイルには、ランタイム環境を初期化するためのコードとデータが入っています。リンク時に −c または −cr を指定して rts.lib を組み込むと、リンカは自動的に boot.obj を抽出してリンクします。

rts.lib および rts_ext.lib アーカイブ・ライブラリには、C/C++ ランタイムサポート関数が入っています。

boot.obj モジュールには、ランタイム環境を初期化するためのコードとデータが入っています。このモジュールは次の作業を実行します。

- ユーザ・モードに切り替えて、ユーザ・モード・スタックを設定します。
- 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)。
- main を呼び出します。
- main が戻ったときに exit を呼び出します。

第 7 章「ランタイムサポート関数」では、ライブラリに組み込まれているその他のランタイムサポート関数について説明しています。これらの関数には、ANSI C 標準ランタイムサポートが含まれています。

注: _c_int00 シンボル

ランタイムサポート・ライブラリに含まれている重要な関数の 1 つに _c_int00 があります。この _c_int00 シンボルは、boot.obj 内の開始点を示します。−c または −cr のリンカ・オプションを使用した場合、_c_int00 はプログラムのエントリ・ポイントとして自動的に定義されます。プログラムがリセットから開始される場合は、プロセッサが最初に boot.obj を実行するように、リセット・ベクトルに _c_int00 への分岐を設定してください。

4.5.3 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使用するグローバル C++ 変数の場合、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、起動時に呼び出すコンストラクタのテーブルを生成します。

このテーブルは、.pinit と呼ばれる名前付きセクションに入っています。コンストラクタは、テーブル内に配置されている順序で起動されます。

コンストラクタはすべて、グローバル変数の初期化の後、および main() が呼び出される前に呼び出されます。デストラクタは atexit() システム・コールによって登録されるので、exit() を呼び出すときに起動されます。

6-33 ページの 6.9.3 項「初期化テーブル」に、.pinit テーブルのフォーマットに関する解説があります。

4.5.4 初期化のタイプの指定

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。それらのテーブルの形式については、 6-33 ページの 6.9.3 項「初期化テーブル」で説明します。これらのテーブルは、.cinit という名前が付いたセクションに入っています。初期化テーブルは、次のどちらかの方法で使用されます。

- 実行時の変数の自動初期化。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 (6-36 ページの 6.9.4 項「実行時の変数の自動初期化」を参照)。
- ロード時の変数の自動初期化。グローバル変数はロード時に初期化されます。−cr リンカ・オプションを使用してください (6-37 ページの 6.9.5 項「ロード時の変数の自動初期化」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を指定することにより、 変数の自動初期化を実行時に行うかリセット時に行うかをリンカに指示します。プログラムをコンパイルしリンクする場合は、−c リンカ・オプションがデフォルトになります。−c リンカ・オプションを使用する場合は、 −z オプションの後ろに置く必要があります (4-4 ページの 4.2 節「コンパイラ・シェルでリンカを起動する方法 (−z オプション)」を参照)。次のリストは、−c または −cr で使用されるリンク規則をまとめたものです。

- _c_int00 シンボルはプログラムのエントリ・ポイントとして定義されています。このシンボルは、boot.obj 内の C/C++ ブート・ルーチンの先頭を示します。 −c または −cr を使用すると、 _c_int00 が自動的に参照されます。その結果、boot.obj がランタイムサポート・ライブラリから自動的にリンクされます。
- .cinit 出力セクションには、ローダ (リセット時の初期化) またはブート・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るよう、終了レコードが埋め込まれています。
- 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、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ことをローダに通知するための特別な属性です。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。

- ❑ 実行時の自動初期化 (−c オプション) の場合、リンカは .cinit セクションの開始アドレスとして cinit シンボルを定義します。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・ツールには含まれていないので注意してください。C54x シミュレータまたはエミュレータをソース・デバッガと組み合わせて、1 つのローダとして使用してください。

4.5.5 セクションをメモリ内のどこに割り当てるかを指定する方法

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。それらのブロックはセクションと呼ばれ、さまざまなシステム構成に対応するようにさまざまな方法でメモリに割り振ることができます。

コンパイラが作成するセクションには、基本的に、初期化されたセクションと初期化されないセクションという 2 種類があります。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	メモリの種類	ページ
.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	ROM または RAM	0
.const	明示的に初期化されるグローバル定数変数と静的定数変数および文字列リテラル	ROM または RAM	1
.text	実行可能なコードと定数	ROM または RAM	0
.switch	switch 文テーブル	ROM または RAM	0

(b) 初期化されないセクション

名前	内容	メモリの種類	ページ
.bss	グローバル変数および静的変数	RAM	1
.stack	スタック	RAM	1
.sysmem	malloc 関数用のメモリ	RAM	1

プログラムをリンクするときは、セクションをメモリ内のどこに割り振るかを指定する必要があります。一般に、初期化されたセクションは ROM または RAM 内にリンクされ、初期化されないセクションは RAM 内にリンクされます。コンパイラがこれらのセクションを使用する方法の詳細については、6-2 ページ 6.1.1 項「セクション」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 および SECTIONS 疑似命令を備えています。セクションをメモリ内に割り振る方法の詳細は、TMS320C54x 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。

4.5.6 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、名前が lnk.cmd で、いくつかのリンカ・オプションのリストを示しています。 このプログラムをリンクするには、次のように入力します。

最初に、例 4−1 のコマンド・ファイルでは、次のようなリンカ・オプションが指定してあります。

−c ROM モデルの自動初期化を使用するようリンカに指示します。

−m マップ・ファイルを作成するようリンカに指示します。この例では、マップ・ファイルの名前は example.map です。

−o 実行可能なオブジェクト・モジュールを作成するようリンカに指示します。この例では、そのモジュールの名前は example.out です。

次に、このコマンド・ファイルではリンクするすべてのオブジェクト・ファイルの名前が指定してあります。 この C プログラムは 2 つの C モジュール、main.c と sub.c から構成されています。これらのモジュールがコンパイルおよびアセンブルされて、main.obj と sub.obj という 2 つのオブジェクト・ファイルが作成されます。 また、この例では asm.obj というアセンブリ言語モジュールもリンクしています。

これらのファイルのいずれかが main シンボルを定義していなければなりません。なぜならば、boot.obj が main を C プログラムの開始点として呼び出すからです。これらの単一オブジェクト・ファイルが、すべてリンクされます。

最後に、このコマンド・ファイルではリンカが検索しなければならないすべてのオブジェクト・ライブラリの名前が指定してあります (ライブラリは −l リンカ・オプションで指定されます)。 これは C のプログラムなので、ランタイムサポート・ライブラリの rts.lib は必ず指定しなければなりません。未定義の参照を解決するライブラリ・メンバだけがリンクされます。

```
lnk500 lnk.cmd
```

使用するシステムで正しく機能させるためには、 MEMORY 疑似命令と、場合によっては SECTIONS 疑似命令に修正を加える必要があります。これらの疑似命令については、TMS320C54x 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

```
/************************************************/
/        Linker command file link.cmd
/************************************************/

-c               /* ROM autoinitialization model */
-m example.map    /* Create a map file           */
-o example.out    /* Output file name            */

main.obj          /* First C module              */
sub.obj           /* Second C module             */
asm.obj           /* Assembly language module    */
-l rts.lib        /* Runtime-support library     */
-l matrix.lib     /* Object library              */

MEMORY
{
   PAGE 0 : PROG: origin = 30h,    length = 0EFD0h
   PAGE 1 : DATA: origin = 800h    length = 0E800h
}
SECTIONS
{
    .text   > PROG  PAGE 0
    .cinit  > PROG  PAGE 0
    .switch > PROG  PAGE 0
    .bss    > DATA  PAGE 1
    .const  > DATA  PAGE 1
    .sysmem > DATA  PAGE 1
    .stack  > DATA  PAGE 1
}
```

第 5 章

TMS320C54x C/C++ 言語

TMS320C54x ™ C/C++ コンパイラは、C プログラミング言語を標準化する目的で米国規格協会 (ANSI) 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。

C54x がサポートする C++ 言語は、*The Annotated C++ Reference Manual* (ARM) で定義されているものです。さらに、ISO/IEC 14882−1998 C++ 規格に基づく多数の拡張機能が実装されています。

項目 **ページ**

5.1 TMS320C54x C の特性

ANSI C は、カーニハンとリッチーによる *The C Programming Language* (初版) で述べられた事実上の標準 C 規格に代わるものです。この ANSI 規格は、米国規格協会 (American National Standard for Information Systems) の *Programming Language C X3.159–1989* で説明されています。*The C Programming Language* の第 2 版は ANSI 規格を基礎としており、K&R と呼ばれます。ANSI C は、現行のさまざまな C コンパイラが備えている言語拡張機能の多くを取り込み、以前に仕様が定まっていなかった C 言語の多数の特性を正式な仕様としています。

ANSI 規格では、ターゲット・プロセッサ、ランタイム環境、またはホスト環境の特性によって影響を受ける C 言語のいくつかの機能について記載しています。これらの機能に関しては、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、C54x C/C++ コンパイラでこれらの機能がどのように実装されているかについて説明します。

以下に、これらの事項のすべてを取り上げ、それぞれに対する C54x C/C++ コンパイラの動作を説明します。それぞれの説明には、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 の正式な ANSI 規格、またはカーニハンとリッチー (K&R) の *The C Programming Language* (第 2 版) についてのものです。

5.1.1 識別子と定数

- ❑ 識別子に関しては、先頭の 100 文字が有効です。また、各識別子ごとに大文字、小文字の区別もあります。 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。 (ANSI 3.1.2, K&R A2.3)

- ❑ ソース (ホスト) 文字セットと実行 (ターゲット) 文字セットは、ASCII であることを前提とします。マルチバイト文字はありません。 (ANSI 2.2.1, K&R A12.1)

- ❑ 文字定数や文字列定数における 16 進や 8 進のエスケープ・シーケンスの値は、最大で 32 ビットまでの値をもちます。 (ANSI 3.1.3.4, K&R A2.5.2)

- ❑ 複数の文字をもつ文字定数は、シーケンスの最後の文字としてコード化されます。以下に例を示します。

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

5.1.2 データ型

- ❑ データ型の表記方法の詳細は、 5.3 節を参照してください。
  (ANSI 3.1.2.5, K&R A4.2)
- ❑ size_t 型 (*sizeof* 演算子の結果) は、unsigned int です。
  (ANSI 3.3.3.4, K&R A7.4.8)
- ❑ ptrdiff_t 型 (ポインタ減算の結果) は、int です。
  (ANSI 3.3.6, K&R A7.7)

5.1.3 変換

- ❑ float から int への変換では、 0 への切り捨てが行われます。
  (ANSI 3.2.1.3, K&R A6.3)
- ❑ ポインタと整数は自由に変換できます。ただし、結果の型は、オリジナル値を十分格納できる大きさでなければなりません。 (ANSI 3.3.4, K&R A6.6)

5.1.4 式

- ❑ 2つの符号付き整数を除算するとき、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 (%) は剰余を求めるために使用します。以下に例を示します。

  ```
  10 / -3 == -3,     -10 / 3 == -3
  10 % -3 == 1,      -10 % 3 == -1
  ```
  (ANSI 3.3.5, K&R A7.6)

  符号付きの剰余演算では、被除数 (第 1 オペランド) の符号をとります。
- ❑ 符号付き数値の右シフトは、算術シフトです。つまり符号は保持されます。
  (ANSI 3.3.7, 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。 (ANSI 3.5.1, K&R A2.1)
- ❑ 構造体のメンバは (ビット・フィールドを除いて) ワードにはパックされません。各メンバは 16 ビットのワード境界に配置されます。 (ANSI 3.5.2.1, K&R A8.3)
- ❑ 整数型のビット・フィールドは、符号付きです。ビット・フィールドは高位のビットから順にワードにパックされます。ワード境界をまたがることはありません。
  (ANSI 3.5.2.1, K&R A8.3)

- ❑ interrupt (割り込み) キーワードは、引数を持たない void 関数に対してのみ適用できます。interrupt キーワードの詳細は、5-9 ページの 5.4.3 項を参照してください。 (TI C extension)

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。 (ANSI 3.8.6, K&R A12.8)

次のプラグマがサポートされています。

- CODE_SECTION
- DATA_SECTION
- FUNC_CANNOT_INLINE
- FUNC_EXT_CALLED
- FUNC_IS_PURE
- FUNC_IS_SYSTEM
- FUNC_NEVER_RETURNS
- FUNC_NO_GLOBAL_ASG
- FUNC_NO_IND_ASG
- IDENT
- INTERRUPT
- NO_INTERRUPT

プラグマの詳細は、5-16 ページの 5.8 節を参照してください。

5.2 TMS320C54x C++の特性

C54x コンパイラは、エリスとストラウストラップの *The Annotated C++ Reference Manual* (ARM) で定義された C++ をサポートしています。さらに、ISO/IEC 14882−1998 C++ 規格に基づく機能の多くを受け入れています。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・サポートは組み込まれていません。特に、iostream ライブラリはサポートされていないので注意してください。C のサブセットおよび基本言語サポートは組み込まれています。
- ❑ 例外処理はサポートされていません。
- ❑ デフォルトでは、ランタイム型情報 (RTTI) は無効にされます。RTTI を使用すると、オブジェクトの型を実行時に決定できます。 RTTI は、−rtti シェル・オプションを使用して有効にすることができます。
- ❑ 組み込まれている C++ 標準ライブラリ・ヘッダ・ファイルは、<typeinfo> と <new> だけです。bad_cast または bad_type_id に対するサポートは、typeinfo ヘッダには組み込まれていません。

5.3 データ型

表 5−1 は、C54x 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 に格納してあり、標準マクロとして使用できます。詳細は、7-15 ページの 7.3.5 項「制限値 (float.h と limits.h)」を参照してください。

表 5−1. TMS320C54x C/C++ のデータ型

型	サイズ	表現	最小値	最大値
signed char	16 ビット	ASCII	−32 768	32 767
char, 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
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
ポインタ	16 ビット	2 進	0	0xFFFF

注: C54x のバイトは 16 ビットです。

ANSI 定義では、sizeof 演算子はオブジェクトの保存に必要なバイト数を示すことになっています。さらに、ANSI では、char 型に sizeof 演算子を使用すると結果は 1 になると規定しています。C54x の char は、16 ビットになっているので (個別にアドレス指定できるようにするため)、1 バイトも 16 ビットです。このため、予期しない結果になることがあります。たとえば sizeof (int) = = 1 (2 ではない) となります。'C54x のバイトとワードは等価 (16 ビット) です。

5.4 キーワード

C54x 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。また、C54x C/C++ コンパイラは、C/C++ 言語を拡張して interrupt、ioport、near、および far キーワードをサポートしています。

5.4.1 const キーワード

C54x C/C++ コンパイラは ANSI 標準のキーワード 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振ります。const のデータ記憶域割り振り 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 に割り振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 内に割り振ることができます。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

ioport キーワードを使用すると、C54x デバイスの I/O ポート空間にアクセスすることができます。このキーワードの形式は、次のとおりです。

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

ioport	ポート変数であることを示すキーワードです。
type	char、short、int、または unsigned でなければなりません。
port*hex_num*	ポート番号を参照します。*hex_num* 引数は 16 進数です。

ポート変数の宣言は、すべてファイル・レベルで行う必要があります。関数レベルでのポート変数の宣言はサポートされていません。関数プロトタイプの中では、ioport キーワードを使用しないでください。

たとえば、次に示すコードは、I/O ポートを符号なし (unsigned) のポート 10h として宣言し、ポート 10h に a を書き込み、ポート 10h を b に読み込みます。

```
ioport unsigned port10; /* variable to access I/O port 10h */

int func ()
{
    ...

    port10 = a;         /* write a to port 10h            */
    ...

    b = port10;         /* read port 10h into b           */
    ...
}
```

ポート変数は、代入にしか使用できないわけではありません。他の変数と同じように、式でポート変数を使用することもできます。次に例を示します。

```
a = port10 + b;  /* read port 10h, add b, assign to a      */
port10 += a;     /* read port 10h, add a, write to port 10h */
```

コールの中では、ポート変数の受け渡しは、参照でなく値によって行われます。

```
call(port10);    /* read port 10h, pass (by value) to call */
call(&port10);   /* invalid pass by reference! */
```

5.4.3 interrupt キーワード

C54x C/C++ コンパイラは、C/C++ 言語を拡張して interrupt キーワードを追加します。これは、関数を割り込み関数として扱うように指定します。

割り込みを処理する関数は、特別なレジスタ保存規則と特別な出口シーケンスを必要とします。C/C++ コードに割り込みを行うと、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しなければなりません。関数の定義に interrupt キーワードを使用すると、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のコードを生成し、割り込み用の特別な出口シーケンスを生成します。

void を戻すように定義した、パラメータを持たない関数に、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、スタックも自由に使用できます。次に例を示します。

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

    ...
}
```

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

厳密な ANSI モード (−ps シェル・オプション) 用のコードを書くには、代替キーワード __interrupt を使用してください。

5.4.4 near および far キーワード

C54x C/C++ コンパイラは、C/C++ 言語を拡張し、near および far キーワードを追加します。これらは、関数の呼び出し方法を指定します。

構文上は、near および far キーワードは、記憶クラス修飾子として扱われます。これらのキーワードは、記憶クラス指定子の前、後、または 2 つの記憶クラス指定子の間の、どこにでも指定することができます。ただし、 1 つの宣言の中に 2 つの記憶クラス修飾子を使用することはできません。正しい使用例を次に示します。

```
far int foo();
static far int foo();
near foo();
```

near キーワードを使用すると、コンパイラは CALL 命令を使用してコールを生成します。far キーワードを使用すると、コンパイラは FCALL 命令を使用してコールを生成します。

コンパイラの起動時に −mf シェル・オプションを指定すると、コンパイラはすべてのコールに対して、デフォルトでファー・コールを生成します。−mf オプションを指定しないと、すべてのコールに対してニア・コールを生成します。

near および far キーワードが影響を与えるのは、関数に対して使用されたコール命令だけであることに注意してください。関数へのポインタには影響を与えません。デフォルトでは、すべてのポインタは 16 ビットになります。−mf オプションを使用すると、ポインタは 24 ビットになり、拡張メモリをポイントできるようになります。

5.4.5 volatile キーワード

オプティマイザはデータ・フローを解析し、できる限りメモリ・アクセスを回避します。C/C++ コードに書か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って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 はループ不変式なので、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 を次のように宣言します。

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

5.5 レジスタ変数

本 C/C++ コンパイラは、1 つの関数の中で最大で 2 つのレジスタ変数を使用します。使用するレジスタ変数は、引数リスト、または関数の最初のブロックで宣言する必要があります。ネストされたブロックの中のレジスタ宣言は、通常の変数として扱われます。

本コンパイラは、レジスタ変数に AR1 と AR6 を使用します。

- ❑ AR1 は、最初のレジスタ変数に割り当てられます。
- ❑ AR6 は、2 番目の変数に割り当てられます。

変数のアドレスは、アクセスを容易にするために、割り振られたレジスタに入れられます。このため、16 ビットの型 (char、short、int、およびポインタ) をレジスタ変数として使用できます。

実行時にレジスタ変数を設定するためには、レジスタ変数ごとに約 4 つの命令が必要となります。この機能を効率的に使用するには、3 回以上アクセスされる場合にだけレジスタ変数を使用してください。

5.6 グローバル・レジスタ変数

'C54x コンパイラは、グローバル・レジスタ の割り振りができるように、レジスタ記憶クラス指定子に特別な規則を追加することによって、 C 言語を拡張しています。この特別なグローバル宣言の形式を次に示します。

register *type regid*

ここで、regid には AR1 または AR6 を指定できます。

AR1 と AR6 という 2 つのレジスタは、一般的にはエントリ時保存レジ スタです。type に float 型と long 型は指定できません。

```
register struct data_struct *AR6;
#define data_pointer (AR6)

data_pointer->element;

data_pointer++;
```

グローバル・レジスタ変数の使用が望ましいのは、次の 2 つの場合です。

- プログラム全体でグローバル変数を使用しており、その変数をレジスタに永久的に割り当てることによってコード・サイズと実行速度が大幅に減少する場合
- 頻繁に実行される割り込みサービス・ルーチンを使用しており、実行のたびに使用するレジスタを保存および復元する必要がなければ実行速度が大幅に上がる場合

グローバル・レジスタ変数の割り当てがもつ意味について、慎重に考える必要があります。レジスタはコンパイラにとって貴重な資源であり、この機能をむやみに使用するとコードの質が低下する可能性があります。

また、グローバルに宣言されたレ ジスタ変数をもつコードが、ライブラリ関数を含め、そのレジスタに加えられた制限を認 識しない他のコードとどのよう に相互作用するのかについても、慎重に考慮する必要があります。

グローバル・レジスタ変数に使用 できるレジスタは、通常はエントリ時保存レジスタであるため、通常の関数呼び出しと復 帰がレジスタ内の値に影響する ことはなく、通常の割り込みに影響することもありません。ただし、グローバルに宣言されたレジスタ変数をもつコードと、予約されたレジスタの ないコードを混在させると、レ ジスタ内の値が破壊される可能性があります。値の破壊を避けるには、次の規則に従う必要があります。

- グローバル・レジスタ変数を変更する関数を、グローバル変数を認識しない関数から呼び出すことはできません。グローバル・レジスタ宣言を認識しないコード内でレジスタを予約するには、−r シェル・オプションを使用します。関数へのポインタを引数として渡す場合、注意が必要です。渡された関数がグローバル・レジスタ変数を変更し、呼び出された関数がレジスタを保存すると、レジスタ内の値は破壊されます。

- すべてのコードを、すべてのライブラリも含めて再コンパイルし、レジスタを予約した場合を除き、割り込みサービス・ルーチンの中でグローバル・レジスタ変数にアクセスすることはできません。その理由は、割り込みルーチンはプログラム内のあらゆる場所から呼び出される可能性があるからです。
- longjump ( ) 関数はグローバル・レジスタ変数を、 setjump ( ) のロケーションで保存されていた値に復元します。 これによってコードに問題が生じる場合は、この関数のコードを変更し、 rts.src を再コンパイルしなければなりません。
- グローバル・レジスタは、それを使用するモジュールの入口で保存し、出口で復元する必要があります。

cl500 シェルの −r *register* オプションを使用すると、指定した *register* をコンパイラに使用させないようにすることができます。上記のような問題を回避するために、グローバル・レジスタの変数宣言をもたないモジュール (ランタイムサポート・ライブラリなど) をコンパイルする必要がある場合、 このオプションを使用すると指定したレジスタをそのモジュール内で予約できます。

コンパイラによる AR1 と AR6 の使用を完全に抑止することができます。こうすると、割り込み関数の中で AR1 か AR6、またはその両方を保存することなしに使用することが可能になります。コンパイラに AR1 と AR6 の使用を禁止する場合は、すべてのコードを −r オプション付きでコンパイルし、ランタイムサポート・ ライブラリを再作成する必要があります。たとえば、次のコマンドは AR1 と AR6 を使用しないよう、 rts.lib ライブラリを再作成します。

```
mk500 –rAR1 –rAR6 –o rts.src –l rts.lib
```

5.7 asm 文

TMS320C54x C/C++ コンパイラでは、C54x アセンブリ言語命令や疑似命令を 、コンパイラのアセンブリ言語出力に、直接埋め込むことができます。 この機能は、C/C++ 言語の拡張機能である *asm* 文によるものです。asm 文は、C/C++ では不可能なハードウェア機能へのアクセスを可能にします。 asm 文は、構文的には、1 つの文字列定数引数をもつ asm という関数の呼び出しに似ています。

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

コンパイラは、引数文字列を直接、出力ファイルにコピーします。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 疑似命令を挿入することができます。

```
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、この行の先頭にはラベル、空白、タブ、あるいはコメント (* または ;) がきます。コンパイラは、この文字列 のチェックは行いません。エラーがある場合は、アセンブラが検出します。詳細については TMS320C54x 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: ams 文で C/C++環境を混乱させないための注意

asm 文で C/C++ 環境を混乱させないためには、細心の注意が必要です。コンパイラは、挿入された命令をチェックしません。ジャンプやラベルを C/C++ コードに挿入すると、挿入したコード内やその周囲で処理する変数に不測の事態が発生することがあります。セクションを変更したり、他の形でアセンブリ環境に影響を与える疑似命令もトラブルの原因になります。

asm 文と共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。オプティマイザで asm 文が削除されることはありませんが、asm 文周辺のコードの並びが大きく変更され、望ましくない結果になることがあります。

5.8 プラグマ疑似命令

プラグマ疑似命令は、コンパイラのプリプロセッサに関数の処理方法を指示します。C54x C/C++ コンパイラは、次のプラグマをサポートしています。

- CODE_SECTION
- DATA_SECTION
- FUNC_CANNOT_INLINE
- FUNC_EXT_CALLED
- FUNC_IS_PURE
- FUNC_IS_SYSTEM
- FUNC_NEVER_RETURNS
- FUNC_NO_GLOBAL_ASG
- FUNC_NO_IND_ASG
- IDENT
- INTERRUPT
- NO_INTERRUPT

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

プラグマは関数またはシンボルに適用されるものですが、C と C++ ではプラグマの構文が異なります。C の場合は、プラグマを適用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する必要があります。C++ の場合は、この名前を省略します。プラグマは、それに続いて指定されるオブジェクトまたは関数の宣言に適用されます。

プラグマを使用して関数にマークを付けると、どのような場合でもその関数がプラグマの仕様を満たすことを、コンパイラに保証したことになります。関数が仕様を満たさない場合があると、コンパイラの動作は不安定になります。

5.8.1 CODE_SECTION プラグマ

CODE_SECTION プラグマは、*section name* という名前のセクションに *symbol* 用のスペースを割り振ります。

C では、このプラグマの構文は次のとおりです。

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

C++ では、このプラグマ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 5−1 は、CODE_SECTION プラグマの使用例です。

例 5−1. 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 DEF: _funcA                                *
;*******************************************************
_main:
        FRAME     #-2
        nop
        STL       A,*SP(0)
        STL       A,*SP(1)
        FRAME     #2
        RET
        ;return occurs
```

5.8.2 DATA_SECTION プラグマ

DATA_SECTION プラグマは、*section name* という名前のセクションに *symbol* 用のスペースを割り振ります。このプラグマは、データ・オブジェクトを .bss セクションとは別の領域内にリンクする場合に便利です。

C では、このプラグマの構文は次のとおりです。

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

C++ では、このプラグマの構文は次のとおりです。

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

例 5−2 は、DATA_SECTION プラグマの使用例です。

例 5−2. 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
```

5.8.3 FUNC_CANNOT_INLINE プラグマ

FUNC_CANNOT_INLINE プラグマは、指定した関数をインライン展開できないことをコンパイラに指示します。このプラグマで指定した関数に対しては、 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、保持したい関数の宣言や参照より前に置かなければなりません。

C では、このプラグマの構文は次のとおりです。

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

C++ では、このプラグマ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 C 関数の名前です。C++ では、プラグマはその直後に宣言されている関数に適用されます。詳細は、 2-36 ページの 2.9 節「インライン関数展開の使用方法」を参照してください。

5.8.4 FUNC_EXT_CALLED プラグマ

−pm オプションを使用すると、コンパイラはプログラム・レベルの最適化を実行します。この種の最適化を実行した場合、コンパイラは main から直接的にも間接的にも呼び出されない関数を除去します。 main ではなく手書きのアセンブリ関数から呼び出される C/C++ 関数が存在する場合もあります。

FUNC_EXT_CALLED プラグマは、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 ておくようオプティマイザに指 示します。それらの関数は、C/C++ へのエントリ・ポイントとして機能します。

このプラグマは必ず、保持したい関数の宣言や参照より前に置かなければなりません。

C では、このプラグマの構文は次のとおりです。

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

C++ では、このプラグマの構文は次のとおりです。

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

C では、引数 func は除去したくない関数の名前です。C++ では、プラグマはその直後に宣言されてい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、*func* 引数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・レベルの最適化を実行するときは、FUNC_EXT_CALLED プラグマと共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。3-8 ページの 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」を参照してください。

5.8.5 FUNC_IS_PURE プラグマ

FUNC_IS_PURE プラグマは、指定した関数には副次作用がないことをオプティマイザに対して指定します。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用した場合、オプティマイザはこれらの副次作用を削除することができます。

C では、このプラグマの構文は次のとおりです。

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

C++ では、このプラグマ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、プラグマはその直後に宣言されている関数に適用されます。

5.8.6 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、ANSI 規格が定義している同じ名前の関数の動作と同じであることをオプティマイザに対して指定します。

このプラグマは、ANSI 規格で記述された関数 (strcmp や memcpy など) に対してのみ使用できます。このプラグマを使用すると、コンパイラは、関数の ANSI 実装をユーザが変更しなかったものと想定して処理を行うことができます。そして、コンパイラは実装についての想定をすることができます。たとえば、その関数が使用するレジスタを想定することができます。

このプラグマは、変更した ANSI の関数と一緒に使用しないでください。

このプラグマは、必ず、関数の宣言や参照より前に置かなければなりません。

C では、このプラグマの構文は次のとおりです。

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

C++ では、このプラグマの構文は次のとおりです。

```
#pragma FUNC_IS_SYSTEM [;]
```

C では、引数 *func* は、ANSI 標準関数として扱う関数の名前です。 C++ では、プラグマはその直後に宣言されている関数に適用されます。

5.8.7 FUNC_NEVER_RETURNS プラグマ

FUNC_NEVER_RETURNS プラグマは、どのような状況でも関数が決して呼び出し元に戻らないことをオプティマイザに 対して指定します。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一時停 止する関数は、決してその呼び出し元に戻ることはありません。関数が このプラグマによりマークされると、コンパイラは、その関数については (たとえばスタックをアンワインドするための) 関数エピローグを生成しません。

このプラグマは、必ず、関数の宣言や参照より前に置かなければなりません。

C では、このプラグマの構文は次のとおりです。

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

C++ では、このプラグマ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、プラグマはその直後に宣言されている関数に適用されます。

5.8.8 FUNC_NO_GLOBAL_ASG プラグマ

FUNC_NO_GLOBAL_ASG プラグマは、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オプティマイザに対して指定します。

このプラグマは、必ず、関数の宣言や参照より前に置かなければなりません。

C では、このプラグマの構文は次のとおりです。

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

C++ では、このプラグマの構文は次のとおりです。

```
#pragma FUNC_NO_GLOBAL_ASG [;]
```

C では、引数 *func* は代入を行わない関数の名前です。C++ では、プラグマはその直後に宣言されている関数に適用されます。

5.8.9 FUNC_NO_IND_ASG プラグマ

FUNC_NO_IND_ASG プラグマは、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オプティマイザに対して指定します。

このプラグマは、必ず、関数の宣言や参照より前に置かなければなりません。

C では、このプラグマの構文は次のとおりです。

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

C++ では、このプラグマの構文は次のとおりです。

```
#pragma FUNC_NO_IND_ASG [;]
```

C では、引数 *func* は代入を行わない関数の名前です。C++ では、プラグマはその直後に宣言されている関数に適用されます。

5.8.10 IDENT プラグマ

IDENT プラグマを使用すると、オブジェクト・コードにコメントを挿入できます。引数 *string* は、コメントのテキスト文字列です。この文字列は、COFF ファイル内の .comment セクションに挿入されます。リンカにより COPY セクションと見なされる .comment セクションは、プロセッサへダウンロードされません。このプラグマは、オブジェクト・コードに識別用の文字列（改訂番号など）を挿入するのに便利です。

このプラグマの構文は次のとおりです。

```
#pragma IDENT (string) [;]
```

COPY セクションの詳細は、TMS320C54x 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。

5.8.11 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、このプラグマの構文は次のとおりです。

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

C++ では、このプラグマの構文は次のとおりです。

```
#pragma INTERRUPT [;]
```

C では、引数 *func* は関数の名前です。C++ では、プラグマはその直後に宣言されている関数に適用されます。

C プログラム用のシステム・リセット割り込みに予約されている _c_int00 という名前を除き、割り込みの名前 (*func* 引数) は命名規則に従っていなくても構いません。

5.8.12 NO_INTERRUPT プラグマ

NO_INTERRUPT プラグマ は、特定の割り込みサービス・ルーチンが割り込みを許可しないことをコンパイラに通知します。指定したルーチンは割り込みを許可しないため、そのルーチンに割り込むことはできません。

C では、このプラグマの構文は次のとおりです。

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

C++ では、このプラグマの構文は次のとおりです。

```
#pragma NO_INTERRUPT [;]
```

割り込みサービス・ルーチンは、呼び出しを行わない場合、デフォルトの RETE 命令でなく RETF 命令によって復帰します。RETF 命令は RETE より 2 サイクル高速です。

5.9 リンク名の生成

コンパイラは、リンク名を作成するときに、外部から見える識別子の名前を変換します。使用されるアルゴリズムは、識別子が宣言されている有効範囲に応じて異なります。オブジェクトおよびC関数の場合は、識別子名の前に下線(_)が付きます。C++関数の場合にも前に下線が付きますが、さらに関数名が変更されます。

マングリングは、関数のシグニチャ(パラメータの数と型)をその関数の名前に組み込むプロセスです。マングリングはC++コードでのみ行われます。使用されるマングリング・アルゴリズムは、*The Annotated Reference Manual*(ARM)の記述に忠実に従っています。マングリングにより、関数のオーバーロード、演算子のオーバーロード、および型に左右されないリンクが可能になります。

たとえば、funcという名前の関数のC++リンク名の一般形式は、次のようになります。

```
___func__F parmcodes
```

*parmcodes*は、funcのパラメータの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・ファイルの例を見てみましょう。

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

結果のアセンブリ・コードは、次のようになります。

```
___foo__Fi;
```

fooのリンク名は___foo__Fiです。これは、fooがint型の引数を1つだけ取る関数であることを示しています。検査とデバッグが行いやすいように、名前をオリジナルのC++ソースの中の名前にデマングリングするネーム・デマングリング・ユーティリティが提供されています。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。

5.10 静的変数とグローバル変数の初期化

ANSI C 規格では、明示的に初期化されていない静的変数とグローバル (外部) 変数は、プログラムが実行を始める前に 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。したがっ て、この要件はユーザ・アプリ ケーション側で満足させる必要があります。

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: {} = 0x00;
      ...
   }
```

リンカは 0 で初期化された .bss セクションの完全なロード・イメージを出力 COFF ファイルに書き込むので、この方法では、出力ファイル (プログラムではなく) のサイズが大幅に増えるという望ましくない影響が発生する可能性があります。

アプリケーションを ROM に焼き込む場合は、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。上に示した方法では、 .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リセットされません。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変数を定義してください。

リンカ・コマンド・ファイルと SECTIONS 疑似命令の詳細は、TMS320C54x 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。

5.10.1 const 型修飾子をもつ静的変数とグローバル変数の初期化

明示的に初期化されていない *const* 型の静的変数とグローバル変数は、事前に 0 に初期化されない可能性があるという点で、他の静的変数やグローバル変数と同じです (5.10 節「静的変数とグローバル変数の初期化」で説明されているのと同じ理由です)。次に例を示します。

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

しかしながら、const 型の静的変数とグローバル変数は、これらの変数が .const というセクションで宣言され、初期化されるという点で異っています。次に例を示します。

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

この指定は .const セクション内の 1 つ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ときに特に便利です。なぜなら、テーブルを初期化する必要がないので、システム始動時に時間とスペースが節約できるからです。さらに、リンカで .const セクションを ROM に配置することもできます。

5.11 ANSI C 言語モードの変更 (−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++ がどのようにソース・コードを解釈するかを指定することができます。ソース・コードのコンパイルに使用できるモードは、次のとおりです。

- 通常の ANSI モード
- K&R C モード
- 緩和 ANSI モード
- 厳密な ANSI モード

デフォルトは通常の ANSI モードです。通常の ANSI モードでは、ほとんどの ANSI 違反はエラーとなります。しかし、厳密な ANSI 違反 (ANSI の厳密な解釈では違反ではあるが、C/C++ コンパイラでは一般に受け入れられるイディオムや許容値など) には、警告が発行されます。言語拡張機能は、ANSI C に矛盾する場合でも有効にされます。

C++ コードの場合は、ANSI モードはサポートされている最新の作業文書を指定するものです。K&R C モードは C++ コードには適用されません。

5.11.1 K&R C との互換性 (−pk オプション)

ANSI C 言語は、カーニハンとリッチーの *The C Programming Language* で定義された、事実上の標準　C 規格のスーパーセットです。ANSI 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及ぼす可能性のある微妙な変更が行われています。*The C Programming Language* (第 2 版、本書で K&R と呼んでいます) の付録 C には、ANSI C と、初版の従来の C 規格 (本書で K&R C と呼んでいるもの) との違いがまとめられています。

既存の C プログラムを、C54x ANSI C/C++ コンパイラで簡単にコンパイルできるようにするために、本コンパイラには K&R C オプション (−pk) が付いています。このオプションにより、C 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 よりも厳しくなっている ANSI C の規則を緩和することです。−pk オプションを指定することにより、関数のプロトタイプ、列挙法、初期化、あるいはプリプロセッサ構成要素などの C 言語の新しい機能が使用できなくなることはありません。−pk は、どの機能も無効にせずに ANSI 規則を緩和するだけのオプションです。

ANSI C と K&R C とで特に異なる点を以下に示します。

- unsigned 型からワイドな signed 型に拡張する場合の整数拡張規則が変更されました。K&R C モードの結果の型はワイド型の unsinged バージョンで、ANSI モードの結果の型はワイド型の singed バージョンになります。これが関係するのは、符号付きオペランドに適用されるか符号なしオペランドに適用されるかで動作が異なる演算、つまり比較、除算 (および mod)、および右シフトなどの演算の場合です。

```
unsigned short u;
int i;
if (u < i) ... /* SIGNED comparison, unless -pk used */
```

- ANSI 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 C 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 ANSI では禁じていますが、K&R C では認められています。

```
a;              /* illegal unless -pk used */
```

- ANSI 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。以下に例を示します。

```
int a;
int a;          /* illegal if -pk used, OK if not */
```

ANSI 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。ほとんどの K&R コンパイラでは、int a が 2 回定義されるので、このシーケンスは不正となります。

- ANSI では禁止していますが、K&R では外部リンクのあるオブジェクトを静的として再宣言できます。

```
extern int a;
static int a;      /* illegal unless -pk used */
```

- 文字列定数と文字定数内の認識できなかったエスケープ・シーケンスは、ANSI では明らかに不正ですが、K&R では無視されます。

```
char c = '\q';     /* same as 'q' if -pk used, error
                      if not */
```

- ANSI 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。 次に例を示します。

```
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 */
```

5.11.2 厳密な ANSI モードと緩和 ANSI モードの有効化 (−ps および −pr オプション)

厳密な ANSI モードでコンパイルしたい場合は、−ps オプションを使用します。このモードでは、ANSI 以外の機能を使用するとエラー・メッセージが表示され、プログラムが厳密に規格に準拠することを阻害する可能性のある言語拡張は無効にされます。このような拡張の例として、inline および asm キーワードがあります。

コンパイラが厳密な ANSI 違反を無視して、警告 (通常の ANSI モードの場合) またはエラー・メッセージ (厳密な ANSI モードの場合) 出さないようにしたい場合は、−pr オプションを使用します。緩和 ANSI モードでは、コンパイラは、ANSI C に矛盾する場合でも ANSI C 規格に対する拡張を受け入れます。

5.11.3 組み込み C++ モードの有効化 (−pe オプション)

本コンパイラは、組み込み C++ のコンパイルをサポートしています。このモードでは、C++ の機能のうち、あまり重要でないか、または組み込みシステムでサポートするには高価すぎる一部の機能は除外されています。組み込み C++ には含まれていない C++ 機能は、次のとおりです。

- テンプレート
- 例外処理
- ランタイム型情報
- 新しい cast 構文
- キーワード /mutable/
- 多重継承
- 仮想継承

組み込み C++ の標準定義では、名前空間も宣言の使用もサポートされていません。しかし、C++ ランタイムサポート・ライブラリがこれらの機能を使用するので、C54x コンパイラでは、組み込み C++ モードでこれらの機能を使用できるようになっています。また、これらの機能を使用することで実行時の条件が不利になることはありません。

5.12 コンパイラの制限値

C54x 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ていること、およびそのようなシステムの一部には機能上の制約があることにより、サイズが大きすぎたり構成が複雑すぎるソース・ファイルのコンパイルが、本コンパイラではできない場合があります。これらの条件のほとんどは、パーサによって検出されます。パーサは、そのような条件を検出すると、障害の原因となった条件を示す I クラスの診断メッセージを発行します。通常、そのメッセージには、超過した制限値の最大値も示されます。コード・ジェネレータにもコンパイルの制限値はありますが、パーサほど多くありません。

一般に、コンパイラの制限値を超えると、コンパイルは続行できなくなります。コンパイラはエラー・メッセージの表示後にただちに異常終了します。コンパイラの制限値を超えないようにするには、プログラムを簡素化します。

多くのコンパイラ・テーブルには、絶対的な制限値はありません。ホスト・システムで利用可能なメモリ容量によってのみ制限されます。表 5−2 は絶対的な制限値を示したものです。絶対制限値は、すべて ANSI C 規格の要求値以上の値になっています。

2 つの値が示されている場合、最初の値は PC ホスト・システムの値で、2 番目の値はその他のホストの値です。すべての絶対制限値は、ANSI C 規格で必要とされる値に等しいか、それ以上です。

PC など小規模なホスト・システムでは、オプティマイザでメモリ不足が発生する場合があります。その場合、オプティマイザは終了し、シェルはコード・ジェネレータでファイルのコンパイルを継続します。　その結果、ファイルは最適化されずにコンパイルされます。オプティマイザでは一度に 1 つの関数をコンパイルすることから、考えられる原因として、ユーザのソース・モジュール内の関数が大きいか、非常に複雑であることが挙げられます。これを防ぐには、次のような方法があります。

- ❑ 疑わしいモジュールは最適化しない。
- ❑ 問題の原因となる関数を特定し、より小さな関数に分割する。
- ❑ モジュールから関数を抽出し、最適化しないでコンパイルできる別のモジュールに配置してから、残りの関数を最適化する。

表 5−2.　コンパイラの絶対制限値

解説	制限値
ファイル名の長さ	512 文字
ソース行の長さ。この制限値は、継続行を連結した後の値です。この制限値は、単独のマクロ定義やマクロ起動にも適用されます。	16K 文字
# や ##から作成した文字列の長さ。この制限値は、連結の前の文字数を表します。その他の文字列には制限はありません。	512 文字
インライン・アセンブリ文字列の長さ	132 文字
−d オプションを指定して事前定義されるマクロ	64
マクロ・パラメータ	32 個
マクロのネスト・レベル。この制限値は、引数の置換を含みます。	64 レベル
#include 検索パス。この制限値は、−i と C_DIR ディレクトリを含みます。	64 パス
#include ファイルのネスト	64 レベル
条件付き組み込み (#if) のネスト	64 レベル
構造体、共用体、あるいはプロトタイプ宣言のネスト	20 レベル
関数パラメータ	48 個
1 つの型に対する、配列、関数、ポインタの派生	12 個
集成体初期化のネスト	32 レベル
ローカル初期化指定子。この制限値は概数です。	150 レベル
if 文、switch 文、およびループのネスト	32 レベル
グローバル・シンボル。使用可能なシステム・メモリにより、さらに制限される可能性があります。	2000 PC 10000 その他すべて
どのポイントからでも可視なブロック・スコープのシンボルの数	500 PC 1000 その他すべて
固有の文字列定数の数	400 PC 1000 その他すべて
固有の浮動小数点定数の数	400 PC 1000 その他すべて

第 6 章

ランタイム(実行時) 環境

本章では、TMS320C54x™ C/C++ の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・モデル

’C54x は、メモリをプログラム・メモリとデータ・メモリの 2 つの連続したブロックとして取り扱います。

- **プログラム・メモリ**は、実行可能なコードを含みます。
- **データ・メモリ**は、外部変数、静的変数、およびシステム・スタックを含みます。

C プログラムが生成するコード・ブロックやデー タ・ブロックは、それぞれ適切なメモリ空間内の連続したブロックに配置されます。

注: リンカによるメモリ・マップの定義

メモリ・マップを定義し、コードとデータをターゲット・メモリに割り当てるのは、コンパイラではなくリンカです。本コンパイラは、コードやデータに利用できるメモリの種類、利用できないロケーション (ホール)、あるいは I/O や制御のために予約されたロケーションについては、何も想定しません。コンパイラは、リンカでコードとデータを適切なメモリ空間に割り当てることができるように再配置可能なコードを作成します。

たとえば、リンカにより、グローバル変数を高速な内部 RAM に割り当てたり、実行可能コードを外部 ROM に割り当てたりすることができます。各コード・ブロックやデータ・ブロックを、メモリに個別に割り当てることができますが、一般的な方法ではありません (例外としてメモリマップド I/O がありますが、物理的なメモリのロケーションは、C/C++ ポインタ型でアクセスできます)。

6.1.1 セクション

本コンパイラは、再配置可能なコードとデータのブロックを生成します。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、TMS320C54x 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・フォーマットの概要」の章を参照してください。

セクションには、次の 2 つの基本型があります。

- **初期化されたセクション** には、データや実行可能なコードを入れます。C/C++ コンパイラは、次の 5 つの初期化されたセクションを作成します。
  - **.cinit セクション** は、変数と定数の初期化のためのテーブルを含みます。
  - **.pinit セクション** は、グローバル・オブジェクト・コンストラクタを実行時に呼び出すテーブルを含みます。
  - **.const セクション** は、C/C++ の修飾子 const によって定義された文字列定数とデータを含みます (その定数が volatile としても定義されている場合は除く)。
  - **.switch セクション** は、switch 文のためのテーブルを含みます。

- **.text セクション**は、文字列リテラルとコンパイラが生成する定数のほか、すべての実行可能コードを含みます。

❑ **初期化されないセクション**は、メモリ (通常は RAM) に空間を確保します。プログラムは、この空間を実行時に変数の作成と格納に使用できます。 本コンパイラは、次の 3 つの初期化されないセクションを生成します。

- **.bss セクション** は、グローバル変数と静的変数用の空間を確保します。 ブート時またはロード時に、C のブート・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、そのデータを使用して .bss 内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振ります。このメモリは変数の受け渡しを行い、ローカル記憶域用に使用されます。
- **.sysmem セクション** は、動的なメモリ割り当て用の空間を確保します。確保された空間は、malloc、calloc、および realloc 関数によって使用されます。C/C++ プログラムでこれらの関数を使用しない場合、コンパイラはこの .sysment セクションを生成しません。

アセンブラによって .data という追加セクションが生成されることに注意してください。C/C++ コンパイラではこのセクションは使用されません。

リンカは、個々のモジュールからセクションを別々に取り出し、同じ名前のセクションをまとめます。この結果出力された 8 つのセクションと、各セクションに該当するメモリ内の配置は、表 6−1 のリストに示すとおりです。これらの出力セクションは、システム要件に合わせて、アドレス空間内の任意の場所に配置できます。

.text セクション、.cinit セクション、.switch セクションは、通常、ROM か RAM にリンクされ、プログラム・メモリ (ページ 0) 内になければなりません。.const セクションも ROM か RAM にリンクできますが、データ・メモリ (ページ 1) 内になければなりません。.bss セクション、.stack セクション、.sysmem セクションは RAM にリンクする必要があり、データ・メモリ内になければなりません。

表 6−1. セクションとメモリ配置の要約

セクション	メモリのタイプ	ページ	セクション	メモリのタイプ	ページ
.bss	RAM	1	.text	ROM または RAM	0
.cinit/.pinit	ROM または RAM	0	.stack	RAM	1
.const	ROM または RAM	1	.switch	ROM または RAM	0
.data	ROM または RAM	1	.sysmem	RAM	1

セクションをメモリ内に割り当てる方法の詳細は、TMS320C54x 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・フォーマットの概要」の章を参照してください。

6.1.2 C/C++ システム・スタック

C/C++ コンパイラは、スタックにより、次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。
- プロセッサのステータスを保存します。

ランタイム・スタックは、単一の連続するメモリ・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に向かってアドレスを消費します。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するかどうかのチェックは行いません。スタックのオーバーフローは、スタックが割り当てられたメモリ空間の限界を超えた場合に発生します。スタックには十分なメモリを割り当ててください。

スタック・サイズは、リンカで設定します。 リンカは、グローバル・シンボル __STACK_SIZE も作成し、スタック・サイズの値 (ワード単位) を割り当てます。デフォルトのスタック・サイズは、1K ワードです。スタックのサイズを変更するには、リンカ・コマンド行で –stack オプションを指定し、スタック・サイズをこのオプションの直後に定数で指定します。

6.1.3 .const のプログラム・メモリへの割り当て

使用しているシステム構成では、.const などの初期化されたセクションをデータ・メモリへ割り当てることができない場合、 .const セクションをロード用にプログラム・メモリに割り当てて、データ・メモリで実行する必要があります。ブート時に、 .const セクションをプログラム・メモリからデータ・メモリにコピーします。以下の手順は、この作業の方法を示したものです。

ブート・ルーチンの修正

1) ソース・ライブラリから boot.asm を取り出します。

```
ar500 –x rts.src boot.asm
```

2) boot.asm を編集して、CONST_COPY フラグを 1 に変更します。

```
CONST_COPY .set  1
```

3) boot.asm をアセンブルします。

```
asm500 boot.asm
```

4) ブート・ルーチンをオブジェクト・ライブラリにアーカイブします。

```
ar500 –r rts.lib boot.obj
```

次のエントリを含むリンカ・コマンド・ファイルでリンクします。

```
MEMORY
{
   PAGE 0 : PROG : ...
   PAGE 1 : DATA : ...
}

SECTIONS
{
   ...
   .const  : load = PROG PAGE 1, run = DATA PAGE 1
             {
                /*  GET RUN ADDRESS   */
                 _ _const_run = .;
                /*  MARK LOAD ADDRESS */
                 * (.c_mark)
                /*  ALLOCATE .const   */
                 * (.const)
                /*  COMPUTE LENGTH    */
                 _ _const_length = .- _ _const_run;
              }
   ...
}
```

使用しているリンカ・コマンド・ファイルでは、ページ 0 のメモリ領域の名前を PROG で、またページ 1 のメモリ領域の名前を DATA で代用して使用することができます。コマンド・ファイルの残りの部分では、上記と同じ名前を使用しなければなりません。CONST_COPY を 1 に変更したときに有効になる boot.asm 内のコードは、このようにこれらの名前を使用するリンカ・コマンド・ファイルによって異なります。名前を変更するには、boot.asm を編集して、同じ方法で名前を変更する必要があります。

6.1.4 動的なメモリ割り当て

コンパイラに添付されている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で提供されています。

メモリは、.sysmem セクションで定義されたグローバル・プール (ヒープ) から割り当てられます。.sysmem セクションのサイズは、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・シンボルである _ _SYSMEM_SIZE も作成し、このシンボルにヒープ・サイズをワード単位で表わした値を代入します。デフォルトのサイズは、2K バイト (1K ワード) です。−heap オプションの詳細については、4-6 ページの 4.4 項「リンカ・オプション」を参照してください。

動的に割り振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)。それに、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));
```

6.1.5 変数の初期化

C/C++ コンパイラは、ROM ベース・システムのファームウェアでの使用に適したコードを作成します。このようなシステムでは、.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・ルーチンは、これらの ROM 内のテーブルからデータをコピーして RAM 内の .bss にある変数を初期化します。

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、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、実行時に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、直接的に初期化を実施することができます。−cr リンカ・オプションを使用して、これをリンカに指示することができます。詳細は、6-32 ページの 6.9 節「システムの初期化」を参照してください。

6.1.6 静的変数とグローバル変数のメモリ割り当て

C/C++ プログラムで宣言された外部変数や静的変数のそれぞれに、固有の連続した空間が割り当てられます。空間のアドレスはリンカによって決定されます。コンパイラは、各変数がワード境界に位置合わせされるように、これらの変数空間がワードの倍数単位で割り当てられるようにします。

本 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 ものとみなします (.bss にグローバル変数用の空間を確保しています)。同じモジュールで宣言された変数は、単一の連続したメモリのブロックに割り当てられます。

6.1.7 フィールド／構造体の位置合わせ

本コンパイラは、構造体に空間を 割り当てる際には、その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) のメンバが含まれている場合、long 型メンバは 2 ワード (32 ビット) 境界に位置合わせされます。この場合、long 型メンバを正しく位置合わせし、構造体の sizeof 値が偶数値になるようにするために、構造体の前、内部、または終りで埋め込みが必要になることがあります。

フィールド型以外は、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、要求されただけのビットが割り当てられます。隣接するフィールドはワードの隣接するビットにパックされますが、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、そのフィールド全体が次のワードに入れられます。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充てんされます。

6.2 文字列定数

C では、文字列定数の使用方法には以下の 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。次に例を示します。

  ```
  char s[] = ″abc″;
  ```

  初期化指定子として使用されると、文字列は単純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、6-32 ページの 6.9 節「システムの初期化」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。次に例を示します。

  ```
  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 (終了を示す値 0 のバイトも含む) 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。以下の例では、終了バイトと同時に文字列 abc を定義します。ラベル SL5 は上記の例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 0 から順に 1 つ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。本コンパイラはこのラベルにより、式の中の文字列を参照します。

  ソース・モジュール内で同じ文字列が繰り返し使用される場合、コンパイラでは、文字列の定義をメモリ内に配置して、定義の回数を最小化しようとします。文字列の重複した利用を単一の定義範囲に収まるようにします。

  文字列は、.const セクション (ほとんどの場合 ROM 内) に格納され、潜在的に共用されるので、プログラムが文字列定数を修正することはお勧めできません。次のコードは、文字列の誤った使用例です。

  ```
  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++ プログラムと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、その規則に従ってください。

レジスタ規則は、コンパイラがレ ジスタをどのように使用するか、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。レジスタには、呼び出し時に保存されるものと、関数の入口で保存されるものと 2 種類があります。これら 2 種類のレジスタの違いは、関数呼び出しの際の保存方 法にあります。レジスタの値の 保存が必要な場合、関数の入口におけるレジスタ変数の保 存は呼び出される側の関数の役 割となり、関数呼び出し時のレジスタ変数の保存は関数を呼び出し元の関数の役割になります。

表 6−2 は、コンパイラによる 'C54x レジスタの使用方法と、関数呼び出しの保存の際に定義されるレジスタを示しています。

表 6−2. レジスタの用途と保存規則

レジスタ	用途	入口での保存	呼び出し時の保存
AR0	ポインタおよび式	No	Yes
AR1	ポインタおよび式	Yes	No
AR2 − AR5	ポインタおよび式	No	Yes
AR6	ポインタおよび式	Yes	No
AR7	ポインタ、式、フレーム・ポインタ (必要な場合)	Yes	No
A	式、第 1 引数を関数に渡す、関数からの結果を戻す	No	Yes
B	式	No	Yes
SP	スタック・ポインタ	†	†
T	乗算およびシフト式	No	Yes
ST0, ST1	ステータス・レジスタ	6-10 ページの 6.3.1 項を参照	
BRC	ブロック・リピート・カウンタ	No	Yes

† SP は「スタックにプッシュされたものは復帰の前にすべてポップされる」という規則によって保存されます。

6.3.1 ステータス・レジスタ

表 6−3 は、ステータス・レジスタのフィールドを示しています。

「推定値」欄に示す値は、次の条件に該当するものです。

- ❑ コンパイラが、関数の入口または関数からの戻りでこのフィールドに入っていることを期待している値
- ❑ アセンブリ関数が C/C++ コードから呼び出されたときに、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しなければならない値。この設定ができない場合、そのアセンブリ関数は C/C++ 関数とともに使用することはできません。

推定値欄にダッシュ (−) がある場合は、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」欄は、コンパイラが生成するコードによってそのフィールドが修正されるかどうかを示します。

その他のすべてのフィールドは使用されず、コンパイラが生成するコードにも影響を及ぼしません。

表 6−3. ステータス・レジスタ・フィールド

フィールド	名前	推定値	修正
ARP	補助レジスタ・ポインタ	0	Yes
ASM	アキュムレータ・シフト・モード	−	Yes
BRAF	ブロック・リピート・アクティブ・ビット	−	No
C	キャリー・ビット	−	Yes
C16	デュアル 16 ビット倍精度演算ビット	0	No
CMPT	互換性モード・ビット	0	No
CPL	コンパイラ・モード・ビット	1	No
FRCT	小数モード・ビット	0	No
OVA	アキュムレータ A のオーバーフロー・フラグ	−	Yes
OVB	アキュムレータ B のオーバーフロー・フラグ	−	Yes
OVM	オーバーフロー・モード	0	組み込み関数とともにのみ
SXM	符号拡張モード	−	Yes
SMUL	乗算器の飽和機能ビット	0	組み込み関数とともにのみ
SST	記憶域の飽和機能ビット	0	No
TC	テスト制御ビット	−	Yes

注: コンパイラは、組み込み関数が使用されている場合を除き、OVM ビットはクリアされていると見なします。

デフォルトでは、コンパイラは、常に (ハードウェア・リセットと同時にクリアされるステータス・レジスタ ST1 内の) OVM ビットが実際にクリアされていると見なします。アセンブリ・コードの中で OVM ビットをセットする場合は、C 環境に戻る前に必ずリセットしなければなりません。

結果を飽和 (オーバーフロー) させる組み込み関数が使用される場合、コンパイラは、飽和した組み込み関数を含むすべての関数で OVM ビットが確実にセットまたはリセットされるようにします。

6.3.2 レジスタ変数

コンパイラは、 register キーワードによって宣言された最大 2 つまでの変数にレジスタを割り当てます。必ず、最初の変数を AR1、2 番目の変数を AR6 として宣言しなければなりません。これらの変数は、グローバルに宣言しなければなりません。

これらの変数は、引数リストの中か、関数の最初のブロック内で宣言する必要があります。ネストしたブロック内でのレジスタ宣言は、通常の変数として扱われます。

本コンパイラは、これらのレジスタ変数に AR1 と AR6 を使用します。AR1 は最初の変数に割り振られ、AR6 は 2 番目の変数に割り振られます。

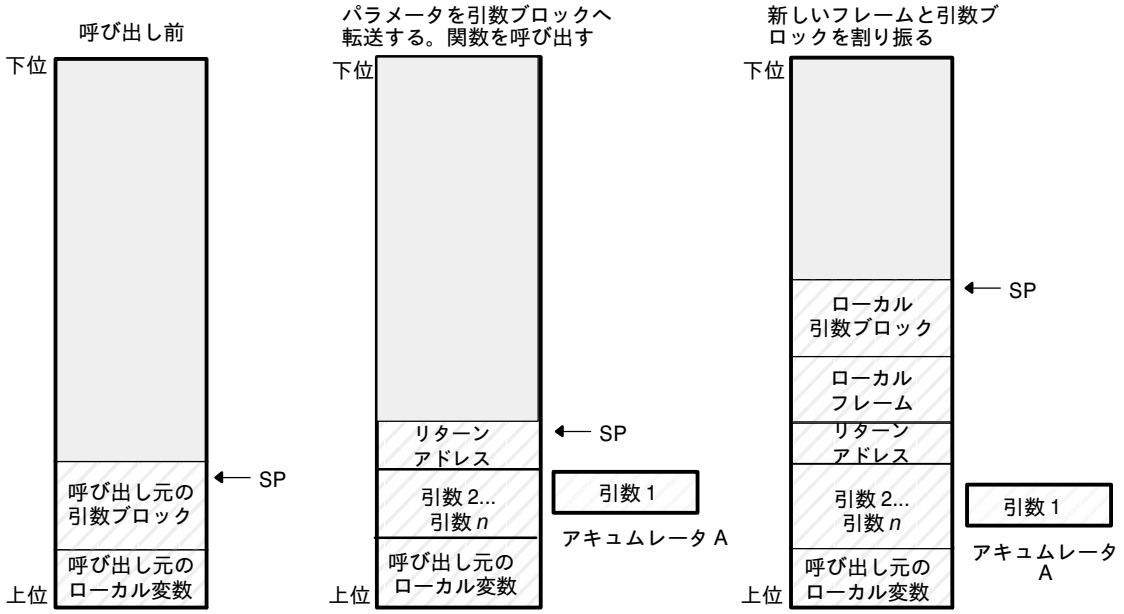
6.4 関数の構造と呼び出し規則

C/C++ コンパイラは、関数呼び出しについて、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 関数から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かった場合は、C/C++ 環境が破壊され、プログラムが異常終了する恐れがあります。

図 6−1 は、典型的な関数呼び出しの例です。この例では、パラメータが関数へ渡され、関数はローカル変数を使用して別の関数を呼び出します。第 1 パラメータがアキュムレータ A へ渡されることに注意してください。また、この例は、呼び出し先の関数に対するローカル・フレームと引数ブロックの割り当ても示しています。 ローカル変数をもたず、引数ブロックを必要としない関数は、ローカル・フレームを割り当てません。

「引数ブロック」とは、別の関数に引数を渡すために使用されるローカル・フレームの部分のことです。パラメータは、スタック上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へ渡されます。 それと同時に、ローカル・フレームと引数ブロックが割り振られます。

図 6−1. 関数呼び出しでのスタックの使用方法



6.4.1 関数の呼び出し方法

関数 (親の関数) は、別の関数を呼び出すとき、次の作業を実行します。

1) 呼び出し元は、最初 (左端) の引数をアキュムレータ A の中に入れます。呼び出し元は、残りの引数を引数ブロックへ逆の順序で転送し、最も左側の残りの引数が最下位アドレスに置かれます。これにより、その引数は関数が呼び出されたときにスタックの最上部にあります。

   省略記号を付けて関数を宣言すると、可変数の引数を指定してその関数を呼び出す意味になります。省略記号を付けて関数を宣言し、引数を 1 つだけ明示的に宣言すると、この規則により、呼び出し元は、その引数をアキュムレータ A でなく、スタック上に渡す必要があります。これにより、その引数のスタック・アドレスは、宣言されていない引数にアクセスするための参照として機能します。次に例を示します。

```
int vararg(int first, ...); /* 'first' is passed */
                            /*  on the stack     */
```

2) 関数が構造体を戻す場合、呼び出し元は、その構造体のための空間を割り振ってから、アキュムレータ A に戻し空間、その構造体の空間のアドレスをセットして呼び出し先関数に渡します。

3) 呼び出し元は、関数を呼び出します。

6.4.2 呼び出し先関数の対応方法

呼び出し先の関数 (子の関数) は、次の作業を実行します。

1) 呼び出し先の関数は、AR1、AR6、または AR7 を修正する場合、それらをスタックにプッシュします。

2) 呼び出し先の関数は、SP から定数を減算することにより、ローカル変数と引数ブロックにメモリを割り当てます。この定数は、次の式で計算されます。

   ローカル変数のサイズ + max + 埋め込み

   max 値は、1 回の呼び出しごとに引数ブロック内に置かれるパラメータのサイズです。埋め込み値は、1 ワードです。この値は、SP が確実に偶数の境界に位置合わせされるために必要な場合があります。

3) 呼び出し先関数は、関数のコードを実行します。

4) 関数が値を戻す場合、呼び出し先関数はその値をアキュムレータ A に入れます。

関数が構造体を戻す場合、呼び出し先関数はその構造体をアキュムレータ A が指すメモリ・ブロックにコピーします。呼び出し元が戻り値を使用しない場合、A は 0 に設定されます。この 0 は、呼び出し先関数に戻りの構造体をコピーしないよう指示します。

このように、呼び出し元は、構造体をどこに戻せばよいかを上手に呼び出し先関数に知らせることができます。たとえば、次のような文があるとします。

```
s = f( )
```

ここで、s は構造体であり、f は構造体を戻す関数です。呼び出し元は、単に s のアドレスを A に入れて f を呼び出します。すると、関数 f は戻りの構造体を s の中に直接コピーし、この代入は自動的に行われます。

構造体を戻す関数を宣言する場合、呼び出される位置で (呼び出し元が A を適切に設定できるように) と、定義される位置 (関数が結果をコピーすることを認知できるように) とで正しく宣言するように注意してください。

5) 呼び出し先の関数は、ステップ 2 で計算した定数を加算することにより、フレームと引数ブロックの割り当てを解除します。

6) 呼び出し先の関数は、保存したすべてのレジスタを復元します。

7) 呼び出し先の関数は戻りを実行します。

次に例を示します。

```
callee:              ; entry point to the function
   PSHM   AR6        ; save AR6
   PSHM   AR7        ; save AR7
   FRAME  #–15       ; allocate frame and
                     ; argument block

   ...               ; body of the function

   FRAME  #15        ; deallocate the frame and
                     ; argument block
   POPM   AR7        ; restore AR7
   POPM   AR6        ; restore AR6
   RET               ; return
```

6.4.3 引数とローカル変数へのアクセス方法

コンパイラは、引数とローカル変数にアクセスするために、コンパイラ・モード (ステータス・レジスタ ST1_55 の CPL ビットを 1 にセットしたときに選択される) を使用します。このビットがセットされていると、直接アドレッシング・モードでは、命令の dma フィールドに入っている定数を SP に加算することによって、データ・アドレスが計算されます。次に例を示します。

```
ADD         *SP(4), A     ; A += *(SP+4)
```

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 127 です。したがって、このモードでアクセスするにはオブジェクトが SP から離れすぎている場合、コンパイラは関数プロローグの中で SP を AR0 にコピーし、ロング・オフセット・アドレッシングを使用してデータにアクセスします。次に例を示します。

```
MVMM        SP, AR0       ; AR0 = SP (in prolog)
...
ADD         *AR0(129),A   ; A += *(AR0 + 129)
```

6.4.4 フレームの割り当てと 32 ビット・メモリ・リード命令の使用方法

'C54x のいくつかの命令は、一度に 32 ビットのメモリを読み書きします (DLD、DADD など)。これらの命令でメモリにアクセスする方法の詳細は、TMS320C54x DSP リファレンス・セット マニュアルを参照してください。その結果、コンパイラは、すべての 32 ビット・オブジェクトが確実に偶数ワード境界に存在するようにしなければなりません。このため、コンパイラは次のステップを実行します。

1) コンパイラは SP を偶数境界に初期化します。
2) CALL 命令は SP から 1 を減算するので、SP は関数の入口で奇数であると見なされます。FCALL では、SP から 2 が減算されることに注意してください。この場合、SP は偶数であると見なされます。
3) デフォルトでは、コンパイラは、PSHM 命令の数に FRAME 命令で割り振られたワード数を加えた合計が奇数になるようにし、SP が確実に偶数アドレスを指すようにします。この場合、スタックにプッシュされる戻りアドレスは 1 ワードで、これにより、SP は奇数になります。したがって、ある奇数を使用して SP が偶数アドレスになるよう調整する必要があります。

   しかし、ファー・モード (−mf を使用) では、FRAME と PSHM によって割り振られるワード数は、SP が偶数アドレスを指すよう、偶数でなければなりません。この場合、スタックにプッシュされる戻りアドレスは 2 ワードで、これにより、SP は偶数になります。したがって、ある偶数を使用して SP が偶数アドレスのまま保持されるよう調整する必要があります。
4) コンパイラは、32 ビット・オブジェクトが、SP 内の既知の偶数アドレスから数えて、確実に偶数アドレスに割り振られるようにします。
5) 割り込み関数は SP が奇数か偶数かを想定できないので、SP を偶数アドレスに位置合わせします。

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++ コードとともに使用する方法は、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)。最も応用範囲が広い方法です。
- ❑ 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 (6-18 ページの 6.5.2 項を参照)。
- ❑ 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 (6-20 ページの 6.5.3 項を参照)。
- ❑ C/C++ ソースの中に組み込み関数を使用し、直接、アセンブリ言語文を呼び出します (6-22 ページの 6.5.4 項を参照)。

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

6-9 ページの 6.3 項「レジスタ規則」で定義したレジスタ規則、および 6.4 項「関数の構造と呼び出し規則」で定義した呼び出し規則に従っている場合、アセンブリ言語関数と C/C++ をインターフェイスす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にアクセスしたり関数を呼び出したりできます。アセンブリ・コードからも、C/C++ の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レジスタは、保存しておく必要があります。専用レジスタは次のとおりです。

  - ■ AR1、AR6、AR7
  - ■ スタック・ポインタ (SP)

  通常の方法で SP を使用する場合は、SP を明示的に保存する必要はありません。言い換えると、スタックにプッシュしたものがアセンブリ関数からの復帰前にポップされて戻される (それによって SP が保存される) 限り、アセンブリ関数で自由にスタックを使用できます。

  専用レジスタ以外のレジスタは、最初に保存しなくても自由に使用できます。

- ❑ 割り込みルーチンは、使用するすべてのレジスタを保存しなければなりません。詳細は、6-27 ページの 6.6 節「割り込み処理」を参照してください
- ❑ アセンブリ言語から C 関数を呼び出す場合、最初 (左端) の引数をアキュムレータ A に入れる必要があります。残りの引数はスタックに逆の順序で入れます。つまり、右端の引数が最上位 (スタックの最も深い位置) のアドレスに置かれます。これを行うには、コンパイラが行うように引数をスタック上の引数ブロックへ直接転送するか、引数をプッシュします。

C 関数から渡された引数にアクセスするときにも、同じ規則が適用されます。

呼び出そうとする関数が、1 つの引数と不特定数の引数を受け入れる場合は、すべての引数をスタックに入れなければなりません。最初の引数は、アキュムレータ A に入りません。6-13 ページの 6.4.1 項「関数の呼び出し方法」を参照してください。

- C/C++ 関数を呼び出すときは、専用レジスタだけが保存されることに留意してください。C/C++ 関数は、それ以外のレジスタの内容を変更する場合があります。
- 構造体の配列内では、構造体に long 型メンバが含まれていない限り、各構造体はワード境界から始まります。long 型を含む構造体は、2 ワード境界に位置合わせされます。この場合、long 型を正しく位置合わせし、構造体の sizeof 値が偶数になるようにするために、構造体の前、内部、または終りにホールが必要になることがあります。
- long 型と float 型は、最上位のワードが下位アドレスになるようにメモリ内に格納されます。
- 関数は、戻り値を 6-13 ページの 6.4.2 項「呼び出し先関数の対応方法」の説明のように戻す必要があります。
- アセンブリ言語モジュールは .cinit セクションをグローバル変数の自動初期化以外の目的に使用してはなりません。boot.asm 内の 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。この名前の空間は、コンパイラが確保します。C/C++ からアクセスできる変数と関数の名前には、前に _ を付けてください。たとえば、x という C/C++ 変数は、アセンブリ言語では _x になります。

  アセンブリ言語モジュール (1 つまたは複数) の中でのみ使用される識別子の場合は、下線で始まりません。
- 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予定のオブジェクトや関数は、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。これによって、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。これによって、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、CP (コンパイラ・モード) ビットが 1 にセットされた状態で実行されるので、直接的にアドレス指定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用することです。次に例を示します。

```
LD *(global_var), A   ; works with CPL == 1
LD global_var, A      ; doesn't work with CPL == 1
```

アセンブリ言語関数の中で CPL ビットを 0 にセットした場合は、コンパイルされたコードに戻る前に、CPL ビットを 1 に戻す必要があります。

例 6−1.　アセンブリ言語関数を C から呼び出す

(a) C プログラム

```
/* declare external asm function */
extern int asmfunc(int, int *);
int gvar;           /* define global variable        */

main()
{
   int i;
   i = asmfunc(i, &gvar);   /* call function normally */
}
```

(b) アセンブリ言語プログラム

```
_asmfunc:

   ADD  *(_gvar),A    ; add gvar to A => i is in A
   STL  A, *(_gvar)   ; return result in A
   RETD               ; start return
```

例 6−1 のアセンブリ言語コードでは、アセンブリ・コード内で使用されているすべての C/C++ シンボル名に下線が付いていることに注意してください。

パラメータ i はアキュムレータ A の中で渡されます。また、gvar にアクセスするために間接絶対モードが使用されていることにも注意してください。CPL ビットは 1 にセットされているので、直接アドレッシング・モードでは SP に dma フィールドが加算されます。したがって、直接アドレッシング・モードを使用してグローバル変数にアクセスすることはできません。

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。これを実現するためには 3 つの方法があり、どの方法を使用するかは、項目が定義されている場所と方法によって異なります。つまり、 .bss セクション内で定義されている変数か、.bss セクション内で定義されていない変数か、あるいは定数かどうかです。

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 という名前の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ことは簡単です。つまり、

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 6−2 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 6−2. C から変数にアクセスする方法

(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           */
```

変数を必ず .bss セクションに入れるとは限りません。たとえば、RAM の中に入れたくないアセンブリ言語で定義したルックアップ・テーブルも、その一般的な例です。その場合は、オブジェクトへのポインタを定義し、そのオブジェクトに、C/C++ から間接的にアクセスしなければなりません。

最初のステップは、オブジェクトを定義することです。オブジェクトは独自の初期化されたセクションに入れると便利です（ただし、必須ではありません）。オブジェクトの先頭を指すグローバル・ラベルを宣言しておけば、その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++ でそのオブジェクトにアクセスするには、オブジェクトを extern として定義する必要があります。その時、前に下線を付けてはなりません。これで、オブジェクトに通常の方法でアクセスできます。

例 6−3 は、.bss の中で定義されていない変数にアクセスする例を示しています。

例 6−3. C から .bss で定義されていない変数にアクセスする方法

(a) C プログラム

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

(b) アセンブリ言語プログラム

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

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。あるいは、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、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なの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ためには、& 演算子 (アドレス演算子) を使用して値を取り出す必要があります。つまり、x がアセンブリ言語定数ならば、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 6−4 に示すように、キャストと #define を使用できます。

例 6−4.　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、シンボル・テーブルに 格納されたようなシンボルの値 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 6−4 では、int が使用されています。リンカで定義したシンボルも、これとほとんど同じ方法で参照できます。

6.5.3　インライン・アセンブリ言語の使用

C/C++ プログラムの中で asm 文を使用すると、コンパイラの作成する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。asm 文を連続して使用すると、他のコードを間に入れることなくア センブリ言語の連続した行をコ ンパイラ出力の中に挿入できます。詳細は、5-15 ページの 5.7 節「asm 文」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 (;) を付けます。

```
asm(";*** this is an assembly language comment");
```

注: asm 文の使用方法

asm 文を使用するときは、次の点に留意してください。

- C/C++ 環境を壊さないよう、十分な注意が必要です。コンパイラは、挿入された命令のチェックや解析を行いません。
- C/C++ コードの中にジャンプやラベルを挿入すると、コード・ジェネレータが使用しているレジスタ追跡アルゴリズムが混乱し、予期できない結果をもたらす恐れがあります。
- asm 文を使用するときは、C/C++ 変数の値を変更しないでください。
- asm 文を使用して、アセンブリ環境を変更する アセンブラ疑似命令を挿入しないでください。

6.5.4 組み込み関数 (Intrinsic) を使用してアセンブリ言語文にアクセスする方法

本コンパイラは、多くの組み込み関数 (Intrinsic) を認識します。組み込み関数は通常の関数のように使用され、C/C++ では他の方法によって表現できないアセンブリ言語文を生成します。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ってアクセスできます。次に例を示します。

```
int x1, x2, y;
y = _sadd(x1, x2);
```

表 6−4 のリストに示す組み込み関数が組み込まれています。これらの関数は、表に示されている C54x アセンブリ言語命令に対応しています。組み込み関数を ETSI (ヨーロッパ電気通信規格研究所) 関数にマップするには、6-26 ページの図 6−2 に示すヘッダ・ファイル intrindefs.h を使用します。ETSI 関数に関する追加サポート情報が、6-25 ページの 6.5.4.1 項に示されています。OVM および FRCT ステータス・レジスタ・ビットの詳細は、TMS320C54x DSP リファレンス・セット、Volume 1 : CPU 及びペリフェラルを参照してください。

表 6−4. TMS320C54x C/C++ コンパイラの組み込み関数

コンパイラの組み込み関数	アセンブリ命令	解説
short _abs(short src);	ABS	16 ビットの絶対値を作成します。
long _labs(long src);	ABS	32 ビットの絶対値を作成します。
short _abss(short src);	ABS	飽和を考慮した 16 ビットの絶対値を作成します _abss (0x8000) => 0x7FFF (OVM はセットされます)。
long _labss(long src);	ABS	飽和を考慮した 32 ビットの絶対値を作成します _labss (0x8000000) => 0x7FFFFFFF (OVM はセットされます)。
short _addc(short src1, short src2);	ADDC	src1、src2、およびキャリー・ビットを加算し、 16 ビットの結果を生成します。
long _laddc(long src1, short src2);	ADDC	src1、src2、およびキャリー・ビットを加算し、 32 ビットの結果を生成します。
short _norm(short src);	EXP	src を正規化するために必要な左シフトの数を生成します。
short _lnorm(long src);	EXP	src を正規化するために必要な左シフトの数を生成します。

表 6−4. TMS320C54x C/C++ コンパイラの組み込み関数 (続き)

コンパイラの組み込み関数	アセンブリ命令	解説
short _rnd(long src);	RND または ADD	2^{15} を加算することによって src を丸めます。16 ビットの飽和を考慮した結果を生成します (OVM はセットされます)。
short _sadd(short src1, short src2);	ADD	2 個の 16 ビット整数を加算し、飽和を考慮した 16 ビットの結果を生成します (OVM はセットされます)。
long _lsadd(long src1, long src2);	ADD	2 個の 32 ビット整数を加算し、飽和を考慮した 32 ビットの結果を生成します (OVM はセットされます)。
long _smac(long src, short op1, short op2);	MAC	op1 と op2 を乗算し、結果を 1 だけ左にシフトし、src に加算します。飽和を考慮した 32 ビットの結果を生成します (OVM と FRCT はセットされます)。
short _smacr(long src, short op1, short op2);	MACAR	op1 と op2 を乗算し、結果を 1 だけ左にシフトし、その結果を src に加算し、さらに 2^{15} を加算することによって結果を丸めます (OVM と FRCT はセットされます)。
long _smas(long src, short op1, short op2);	MAS	op1 と op2 を乗算し、結果を 1 だけ左にシフトし、src から減算します。32 ビットの結果を生成します (OVM と FRCT はセットされます)。
short _smasr(long src, short op1, short op2);	MASAR	op1 と op2 を乗算し、結果を 1 だけ左にシフトし、その結果を src から減算し、さらに 2^{15} を加算することによって結果を丸めます (OVM と FRCT はセットされます)。
short _smpy(short src1, short src2);	MPYA	src1 と src2 を乗算し、結果を 1 だけ左にシフトします。飽和を考慮した 16 ビットの結果を生成します (OVM と FRCT はセットされます)。
long _lsmpy(short src1, short src2);	MPY	src1 と src2 を乗算し、結果を 1 だけ左にシフトします。飽和を考慮した 32 ビットの結果を生成します (OVM と FRCT はセットされます)。
short _smpyr(short src1, short src2);	MPYR	src1 と src2 を乗算し、結果を 1 だけ左にシフトし、その結果に 2^{15} を加算することによって丸めます (OVM と FRCT はセットされます)。

表 6−4.　TMS320C54x C/C++ コンパイラの組み込み関数 (続き)

コンパイラの組み込み関数	アセンブリ命令	解説
short _sneg(short src);	NEG	飽和処理付きで 16 ビット値の否定演算を行います。 _sneg (0xffff8000) => 0x00007FFF
long _lsneg(long src);	NEG	飽和処理付きで 32 ビット値の否定演算を行います。 _lsneg (0x80000000) => 0x7FFFFFFF
short _sshl(short src1, short src2);	SFTA	src1 を src2 だけ左にシフトし、16 ビットの結果を生成します。src2 が 8 以下ならば、飽和を考慮した結果が生成されます (OVM はセットされます)。
long _lsshl(long src1, short src2);	SFTA	src1 を src2 だけ左にシフトし、32 ビットの結果を生成します。src2 が 8 以下ならば、飽和を考慮した結果が生成されます (OVM はセットされます)。
short _ssub(short src1, short src2);	SUB	OVM をセットした状態で src1 から src2 を減算し、飽和を考慮した 16 ビットの結果を生成します。
long _lssub(long src1, long src2);	DSUB	OVM をセットした状態で src1 から src2 を減算し、飽和を考慮した 32 ビットの結果を生成します。
short _subc(long src1, short src2);	SUBB	src2 と符号ビットの論理否定とを src1 から減算し、16 ビットの結果を生成します。
long _lsubc(long src1, short src2);	SUBB	src2 と符号ビットの論理否定とを src1 から減算し、32 ビットの結果を生成します。

6.5.4.1 組み込み関数と ETSI 関

表 6−5 の各関数は、組み込み関数に対して ETSI サポートを提供します。この表にある関数は、ランタイム関数です。

表 6−5. ETSI サポート関数

コンパイラの組み込み関数	解説
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。この関数は単一のアセンブリ命令へマップされず、インライン・マクロへマップされます。
long L_sub_c(long src1, long src2);	src1 から src2 と符号ビットの論理否定を減算します。この関数は単一のアセンブリ命令へマップされず、インライン・マクロへマップされます。
long L_sat(long src1);	オーバーフローがセットされている場合は、L_add_c または L_sub_c より後のすべての結果を飽和させます。
int clshft(int x, int y);	x を y だけ左にシフトし、結果の飽和を保証します。
int crshft(int x, int y);	x を y だけ右にシフトし、結果の飽和を保証します。
long l_clshft(long x, int y);	x を y (32 ビット) だけ左にシフトし、結果の飽和を保証します。
long l_crshft(long x, int y);	x を y (32 ビット) だけ右にシフトし、結果の飽和を保証します。
int crshft_r(int x, int y);	x を y だけ右にシフトし、飽和を考慮して結果を丸めます。
long L_crshft_r(long x, int y);	x を y だけ右にシフトし、飽和を考慮して結果を丸めます。
int divs(int x, int y);	飽和を考慮して x を y で除算します。

図 6−2.　組み込み関数用ヘッダ・ファイル、 intrindefs.h

```
#define MAX_16 0x7fff
#define MIN_16 –32768
#define MAX_32 0x7fffffff
#define MIN_32 0x80000000

#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) (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) ((b) < 0 ? L_crshft((a),(–b)) : \
                (b) < 9 ? _lsshl((a),(b)) : \
                L_clshft((a),(b)))
#define L_shr(a,b) (L_crshft((a),(b)))
#define L_shr_r(a,b) (L_crshft_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) (_rnd(a))
#define mac_r(a,b,c) (_smacr((a),(b),(c)))
#define msu_r(a,b,c) (_smasr((a),(b),(c)))
#define mult(a,b) (_smpy((a),(b)))
#define mult_r(a,b) (_smpyr((a),(b)))
#define norm_s(a) (_norm(a))
#define norm_l(a) (_lnorm(a))
#define negate(a) (_sneg(a))
#define shl(a,b) (clshft((a),(b)))
#define shr(a,b) (crshft((a),(b)))
#define shr_r(a,b) (crshft_r((a),(b)))
#define div_s(a,b) (divs(a,b))
```

6.6 割り込み処理

この節のガイドラインに従えば、C/C++ 環境を破壊せずに C/C++ コードに割り込み、そして 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、始動ルーチンによって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 (ハードウェアのリセットによってシステムを初期化すると、割り込みは無効になります)。システムで割り込みが使用される場合、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の責任となります。このような操作は、 C/C++ 環境に影響を与えずに、 簡単に asm 文で組み込むことができます。

6.6.1 割り込みの概要

割り込みルーチンでは、グローバル変数へのアクセス、ローカル変数の割り当て、および他の関数の呼び出しなど、他の関数で実行できるすべてのタスクを実行できます。

割り込みルーチンを作成するときは、以下の点に注意してください。

- ❑ IMR レジスタによる特別な割り込みのマスク処理は、ユーザの責任となります。インライン・アセンブリを使用して、C/C++ 環境や C/C++ ポインタを破壊せずに割り込みを許可したり抑止したり、 IMR レジスタを変更したりすることができます。
- ❑ 割り込み処理ルーチンに引数を使用することはできません。引数を宣言しても無視されます。
- ❑ 割り込み処理ルーチンは、通常の C/C++ コードから呼び出すことができません。
- ❑ 戻るには、C/C++ で書いた割り込みルーチンは RETE 命令を実行し、それによって 1 ワードをスタックからポップして PC に入れます。この C/C++ ルーチンが 'C54x 拡張メモリ・プロセッサ (C548 以上) 用にコンパイルされている場合は、戻り命令 FRETE スタックから PC に 1 ワードをポップし、別の 1 ワードをスタックから XPC にポップします。C/C++ コードからでなくアセンブリ・コードから C/C++ 割り込みルーチンに入った場合は、C/C++ ルーチンから戻ったときのスタックの正しい状態をユーザが確保する必要があります。
- ❑ コンパイラは、C/C++ 環境を明示的に設定するために、割り込みルーチンのプロローグ・セクションとエピローグ・セクシ ョンにコードを組み込みます。 CPL ビットがセットされ、OVM、SMUL、および SST ビットはクリアされます。この余分なコードが C/C++ 割り込みルーチンに追加されないようにするには、 −me シェル・オプションを使用します。
- ❑ 割り込み処理ルーチンでは、単一の割り込みでも複数の割り込みでも処理できます。コンパイラは、特定の割り込みに対して固有のコードを生成することはしません。ただし、c_int00 だけは例外で、これはシステム・リセット割り込みです。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振ることはできず、ランタイム・スタックに情報を保存することもできません。

- ❏ 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しなければなりません。アセンブラとリンカを使用してこの作業を行なうには、.sect アセンブラ疑似命令を使用して分岐命令の単純なテーブルを作成します。
- ❏ アセンブリ言語内では、シンボル名の前に下線を付けるのを忘れないでください。たとえば、c_int00 は _c_int00 となります。
- ❏ スタックは偶数 (long 型の位置合わせの) アドレスに位置合わせしてください。

6.6.2 C/C++ 割り込みルーチンの使用方法

interrupt キーワードを使用することによって、C/C++ 関数で直接、割り込みを処理できるようになります。次に例を示します。

```
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。コンパイラは、それらのルーチンの 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。この方法により、標準的な C/C++ シグナル・メカニズムより多くの機能が得られます。これにより、シグナル関数の実装を妨げずに、シグナル関数全体を C/C++ で書くことができるようになります。

6.6.3 割り込みエントリのコンテキストの保存方法

ステータス・レジスタも含めて、割り込みルーチンで使用するすべてのレジスタは、保存しておかなければなりません。割り込みルーチンで別の関数を呼び出す場合は、6-9 ページの 表 6−2 に示したすべてのレジスタを保存しなければなりません。

'C54x のいくつかの命令は、一度に 32 ビットのメモリにアクセスします (DLD、DADD など)。その結果、コンパイラはスタック・ポインタに常に偶数の値が入るように処理する必要があります。それらの方法については、6-15 ページの 6.4.4 項「フレームの割り当てと 32 ビット・メモリ読み取り命令の使用方法」に詳しい説明があります (これらの命令でメモリにアクセスする方法の詳細は、TMS320C54x リファレンス・セット マニュアルを参照してください)。

割り込みルーチンでは、スタック・ポインタが偶数か奇数かを認識しません。したがって、コンパイラは次の命令を発行してレジスタを保存し、スタック・ポインタの位置合わせを行います。

```
PSHM  ST0           ; first save off all other registers
. . .
PSHM  SP            ; push the SP
ANDM  #0FFFEH,*(SP) ; align to even boundary
```

コンパイラは、SP を偶数アドレスに位置合わせする前に、SP をスタックに保管するためのコードを生成します。そして、割り込みルーチンの終了時に、スタックから SP を復元します。

6.7 整数式の分析

この節では、整数式を評価する際に、特に注意が必要な考慮事項について説明します。

6.7.1 算術オーバーフローと算術アンダーフロー

'C54x では、データ・オペランドとして 16 ビット値または 32 ビット値が使用されているときでも、生成結果は 40 ビットになります。このため、 算術オーバーフローと算術アンダーフローを予測可能な方法で処 理することはできません。コードが特定のタイプのオーバーフロー／アンダーフロー処理 に依存している場合、そのコードが正しく実行される保証はありません。

6.7.2 RTS 呼び出しで評価される演算

'C54x は、一部の C/C++ 演算を直接的にはサポートし ません。これら の演算の評価は、ランタイムサポート・ルーチンの呼び出しによって行われます。 これらのルーチンは、アセンブリ言語でハード・コーディ ングされています。これらは、 本ツールセットに含まれるオブジェクトおよびソース RTS ライブラリ (rts.lib および rts.src) のメンバです。

これらのルーチンを呼び出すための規則は、標準 C/C++ 呼び出し規則をモデルとしています。バイナリ・ルーチン (divide など) の場合、左オペランドはアキュムレータ A に渡され、右オペランドはスタックに渡されます。その結果は、アキュムレータ A に戻されます。単項ルーチンの場合、引数と結果は、どちらもアキュムレータ A に渡されます。

演算の型	RTS 呼び出しで計算される演算
16 ビット int	除算
	剰余
32 ビット long	除算
	剰余
	乗算
	左シフト
	右シフト

6.7.3 16 ビット乗算の上位 16 ビットへの C コードによるアクセス

以下の方法により、C 言語で 16 ビット乗算の上位 16 ビットにアクセスできます。

- 符号付き結果の方法

```
int m1, m2;
int result;
result = ((long) m1 * (long) m2) >> 16;
```

- 符号なし結果の方法

```
unsigned m1, m2;
unsigned result;
result = ((unsigned long) m1 * (unsigned long) m2) >> 16;
```

どちらの result 文も、32 ビット乗算ルーチンの呼び出しを行わずに、コンパイラによって実装されます。

注: 式の構造が複雑な場合の危険

コンパイラが予期される結果を戻すには、式の構造を正しく認識する必要があります。次のような、複雑な式は避けてください。

```
((long)((unsigned)((a*b)+c)<5)*(long)(z*sin(w)>6))>>16
```

6.8 浮動小数点式の分析

C54x C/C++ コンパイラでは、浮動小数点値を IEEE 単精度数として表します。単精度も倍精度も、浮動小数点は 32 ビット値として表されます。この 2 つの形式に違いはありません。

C54x ランタイムサポート・ライブラリ rts.lib には、以下の機能をサポートする浮動小数点演算関数のカスタムコード・セットが格納されています。

- ❑ 加算、減算、乗算、除算
- ❑ 比較 (>, <, >=, <=, ==, !=)
- ❑ 符号付きと符号なしの両方の、int または long から浮動小数点への変換と、その逆の変換
- ❑ 標準的なエラー処理

これらのルーチンを呼び出すための規則は、整数演算ルーチンを呼び出すための規則と同じものです。変換は単項の演算です。

6.9 システムの初期化

C/C++ プログラムを実行するには、C/C++ のランタイム環境を構築しなければなりません。この作業は、C/C++ ブート・ルーチンが行います。このルーチンは、_c_int00 という関数です。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 というモジュールの中に納められています。

システムの実行を開始するには、ハードウェアをリセットすることによって _c_int00 関数を呼び出すことができなければなりません。_c_int00 関数をその他のオブジェクト・モジュールとリンクする必要があり ます。これを自動で行うには、 −c または −cr リンカ・オプションを使用し、rts.lib をリンカ入力ファイルに含めます。

C/C++ プログラムをリンクすると、リンカは実行可能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、以下の作業を実行して C/C++ 環境を初期化します。

1) .bss 内のスペースをランタイム・スタック用に予約し、スタック・ポインタ (SP) の初期値を設定します。

2) グローバル変数を初期化するために、.cinit 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)。詳細は、6.9.1 項「変数の自動初期化」を参照してください。

3) 関数 main を呼び出して、C/C++ プログラム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。

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る必要があります。これらの変数のデータを検索し、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、これらの初期化テーブルが含まれています。リンカは、これらのテーブルを 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

ANSI C では、明示的に初期化されないグローバル変数と静的変数は、プログラムの実行前に 0 に設定しておかなければなりません。C/C++ コンパイラが、初期化されない変数を事前に初期化することはありません。 初期値が 0 でなければならない変数は、明示的に初期化しなければなりません。

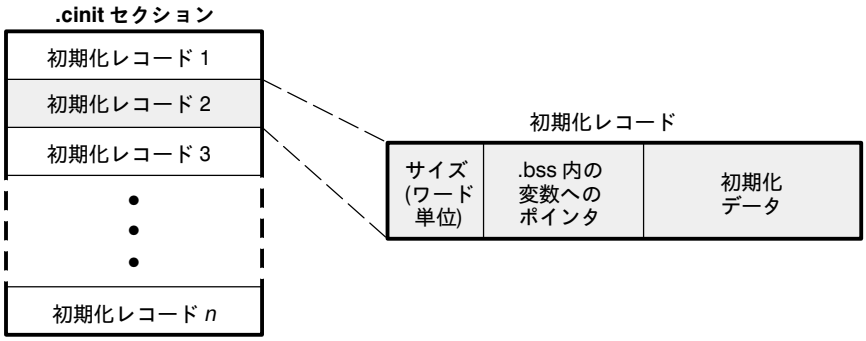
6.9.2 グローバル・コンストラクタ

コンストラクタをもつすべてのグローバル C++ 変数については、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 というセクション内で 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ビルドします。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 セクション内で 1 つのテーブルにします。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 6−3 は、.cinit セクションと初期化レコードの形式を示しています。

図 6−3.　.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、以下の情報が含まれています。

- 先頭のフィールド (ワード 0) は、変数の初期化データのサイズ (ワード単位) を示します。
- 2 番目のフィールド (ワード 1) は、初期化データをコピーする .bss セクション内の領域の先頭アドレスを示します。
- 3 番目のフィールド (ワード 2 ～ *n*) には、初期化時に変数にコピーされる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入っています。例 6−5 (a) は、C/C++ で定義された 2 つの初期化される変数を示しています。例 6−5 (b) は、これに対応する初期化テーブルを示しています。

例 6−5.　初期化変数と初期化テーブル

(a) C で定義された初期化される変数

```
int    i = 23;
int    a[5] = { 1, 2, 3, 4, 5 };
```

(b) (a) で定義された変数の初期化される情報

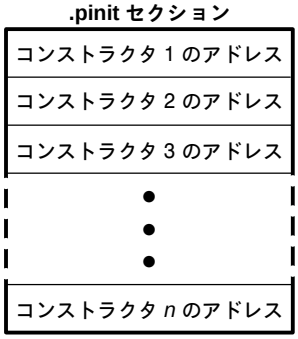
```
   .sect     ".cinit"     ; Initializationsection
* Initialization record for variable i
   .word    1             ; length of data (1 word)
   .word    _i            ; address in .bss
   .word    23            ; data to initialize i

* Initialization record for variable a
   .word    5             ; length of data (5 words)
   .word    _a            ; address in .bss
   .word    1,2,3,4,5     ; data to initialize a
```

.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ときは、.cinit セクションを他の目的に使用しないでください。

−c や −cr リンカ・オプションを使用すると、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 6−4. .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・オプションを使用すると、リンカは、すべての C/C++ モジュールの .pinit セクションをすべてまとめてリンクし、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、ヌルのコンストラクタ・アドレスを見つけると、グローバル・コンストラクタ・テーブルの最後に達したことを認識します。

const で修飾された変数の初期化方法は異なるので、注意してください。詳細は、5-26 ページの 5.10.1 項「const 型修飾子をもつ静的変数とグローバル変数の初期化」を参照してください。

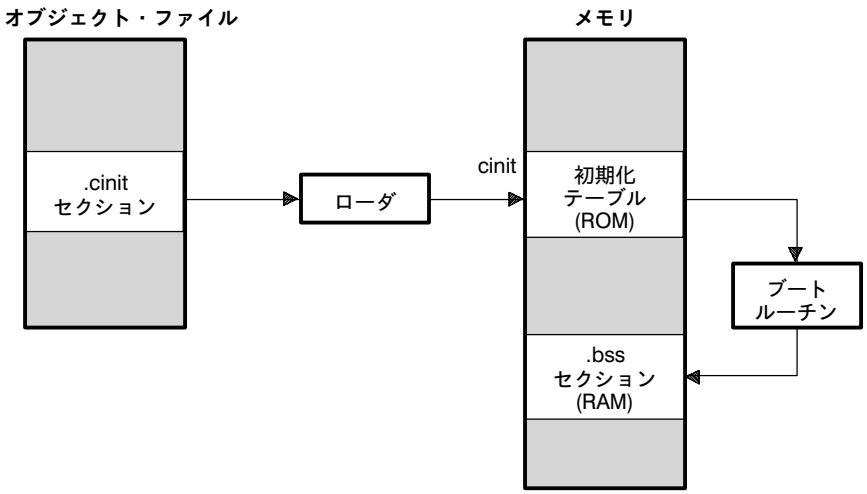
6.9.4 実行時の変数の自動初期化

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。

この方法を使用すると、.cinit セクションは、その他の初期化されたすべてのセクションとともにメモリ (場合によっては ROM) 内にロードされ、グローバル変数は実行時に初期化されます。リンカは、メモリ内の初期化テーブルの先頭を指す、cinit という特別なシンボルを定義します。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の中へコピーします。これにより、初期化 データを ROM に格納し、プログラムが起 動するたびに RAM にコピーすることができます。

図 6−5 は、実行時の自動初期化の例を示しています。ROM に組み込まれたコードからアプリケーションを実行するシステムでは、この方法を使用してください。

図 6−5. 実行時の自動初期化



6.9.5 ロード時の変数の自動初期化

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って使用されるメモリも節約できるため、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。

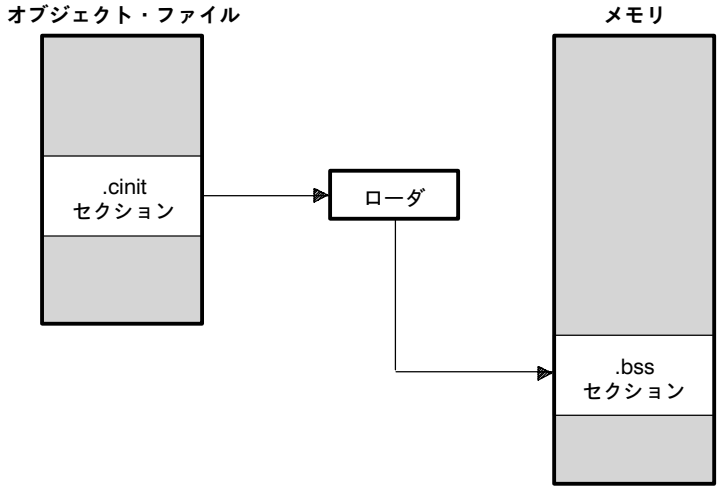
−cr リンカ・オプションを使用すると、リンカは、.cinit セクションのヘッダ内に STYP_COPY ビットを設定します。これにより、.cinit セクションをメモリ内にロードしないよう、ローダに指示します (.cinit セクションは、メモリ・マップ内のスペースを占有しません)。また、リンカは、cinit シンボルを −1 に設定します (通常では、cinit は初期化テーブルの先頭を指します)。これにより、初期化テーブルがメモリ内に存在しないことをブート・ルーチンに示します。したがって、ブート時に実行時の初期化は行われません。

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) は、以下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 が設定されていることを判別して、.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ようにする。
- ❑ 初期化テーブルのフォーマットを理解する。

図 6−6 は、自動初期化の RAM モデルの例を示しています。

図 6−6. ロード時の自動初期化



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 (たとえば、入出力、動的メモリ割り当て、文字列操作、文字列検索など) の中には、C/C++ 言語自体の一部ではないものがあります。C/C++ コンパイラに組み込まれているランタイムサポート関数は、これらの作業を実行する標準 ANSI 関数です。

ランタイムサポート・ライブラリである rts.src には、このような標準 ANSI 関数のソースとともに、その他の関数やルーチンのソースが含まれています。基礎のオペレーティング・システムを必要とする関数 (シグナル関数など) を除いて、すべての ANSI 関数が提供されています。

コード生成ツールに含まれている、ライブラリ作成ユーティリティを使用すると、カスタマイズしたランタイムサポート・ライブラリを作成できます。ライブラリ作成ユーティリティの使用方法については、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

TMS320C54x C/C++ コンパイラに含まれているライブラリは、次のとおりです。

- *rts.lib* には、ANSI ランタイムサポート・オブジェクト ・ライブラリが含まれています。
- *rts.src* には、ANSI ランタイムサポート・ルーチン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、本章で説明する標準 C/C++ ランタイムサポート関数、浮動小数点ルーチン、システム始動ルーチン、_c_int00 が含まれています。オブジェクト・ライブラリは、rts.src にある C/C++ ソースおよびアセンブリ ・ソースから作成されます。

プログラムをリンクするときには、入出力関数とランタイムサポート関数に対する参照を解決できるように、リンカ入力ファイルの 1 つとしてオブジェクト・ライブラリを指定する必要があります。

ライブラリはリンカ・コマンド行の最後に指定する必要があります。リンカは、コマンド行でライブラリを検出すると、未解決の参照のためにライブラリを検索するからです。−x リンカ・オプションを使用して、リンカが解決できる参照がなくなるまで、各ライブラリの検索を強制的に繰り返すこともできます。

ライブラリをリンクすると、リンカは、未定義の参照を解決するのに必要なライブラリ・メンバだけを組み込みます。リンクの詳細は、TMS320C54x アセンブリ言語ツール ユーザーズ・マニュアル のリンカに関する説明の章を参照してください。

7.1.1 rts.src の標準外ヘッダ・ファイル

rts.src ファイルには、ライブラリの作成に使用される以下の非 ANSI ヘッダ・ファイルが含まれています。

- *values.h* ファイルには、三角関数と超越算術関数の再コンパイルに必要な定義が含まれています。values.h の関数は、必要に応じてカスタマイズできます。
- *file.h* ファイルには、低レベル入出力関数に使用されるマクロと定義が含まれています。
- *format.h* ファイルには、printf と scanf で使用される構造体とマクロが含まれています。
- *trgcio.h* ファイルには、低レベルのターゲット固有の C 入出力マクロ定義が含まれています。trgcio.h は、必要に応じてカスタマイズできます。

7.1.2 ライブラリ関数の修正方法

アーカイバで 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することができます。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。

```
ar500 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。

```
cl500 –options atoi.c strcpy.c      ;recompile
ar500 –r rts.src atoi.c strcpy.c   ;rebuild library
```

この方法により、rts.lib をリビルドさせる代わりに、新しいライブラリをビルドすることもできます。アーカイバの詳細は、TMS320C54x 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。

7.1.3 さまざまなオプションによるライブラリの作成方法

ライブラリ作成ユーティリティ mk500 を使用することにより、rts.src から新しいライブラリを作成できます。たとえば、次のコマンドによって、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。

```
mk500 –-u –o2 rts.src –l rts.lib
```

– –u オプションを指定すると、mk500 ユーティリティは、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・ファイルを使用します。オプティマイザ (–o2) を使用しても、これらのオプションを使用しないでコンパイルされたコードとの互換性に影響はありません。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、C54x プログラムで実行される printf 文は、デバッガのコマンド・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、関数を参照するそれぞれのモジュールにヘッダ・ファイル stdio.h、または C++ コード用の cstdio を組み込んでください。

たとえば、main.c というファイルに次のプログラムが指定してあると想定します。

```
#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、コンパイルおよびリンクが行われ、ファイル main.out が作成されます。

```
cl500 main.c –z –heap 400 –l rts.lib –o main.out
```

SPARC ホスト上の C54x デバッガのもとで main.out を実行すると、次の操作が実行されます。

1) デバッガが起動されたディレクトリ内で、ファイル *myfile* がオープンされます。
2) そのファイルに文字列 *Hello, world* が出力されます。
3) ファイル *myfile* がクローズされます。
4) デバッガのコマンド・ウィンドウに、文字列 *Hello again, world* が表示されます。

また、正しく作成されたデバイス・ドライバがあれば、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 入出力バッファのヒープに十分なスペースがないと、ファイルに対するバッファ付き操作は失敗します。printf() の呼び出しが失敗した原因が不明の場合、スペース不足が原因になっていることがあります。ヒープのサイズを調べてください。ヒープ・サイズを設定するには、リンク時に −heap オプションを使用します。

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、高レベル、低レベル、デバイス・レベルの 3 層に分かれます。

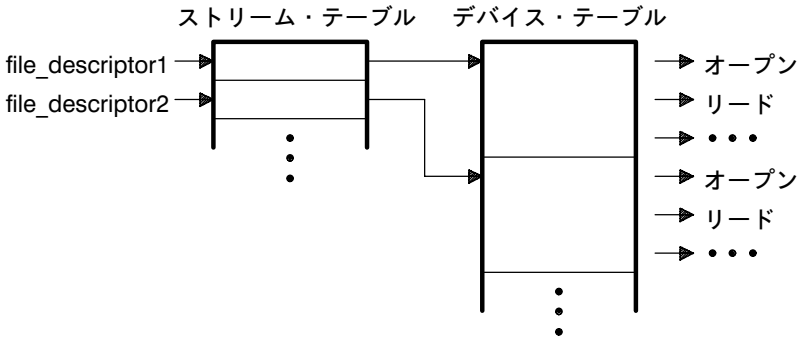
高レベル関数は、ストリーム入出力ルーチン (printf、scanf、fopen、getchar など) の標準 C ライブラリです。これらのルーチンは、入出力要求を低レベル・シェルによって処理される 1 つまたは複数の入出力コマンドにマップします。

低レベル関数は、基本的な入出力関数 (open、read、write、close、lseek、rename、および unlink) から構成されます。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・レベルの 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。

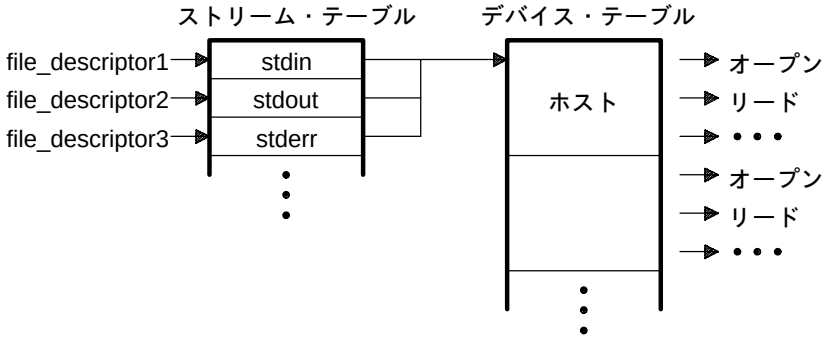
このようなデータ構造の相互作用を、図 7–1 に示します。

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、および 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・レベルのドライバです。このデバイス・レベルのドライバは、低レベルの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上で C 入出力を実行するために必要なデバイス・ドライバが組み込まれています。

低レベルのルーチンとインターフェイスするようにデバイス・レベルのルーチンを作成する際の仕様については、7-8 ～ 7-11 ページで説明しています。必要に応じて、それぞれに固有のデータ構造を設定し維持するように関数を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

低レベル関数には、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あります。この機能を使用するための手順を、以下に示します。

1)　7-5 ページの 7.2.1 項で説明したように、デバイス・レベル関数を定義します。

注:　一意的な関数名を使用してください。

open()、close()、read() などの関数名は、低レベル・ルーチンで使用されています。作成するデバイス・レベル関数には、別の名前を使用してください。

2)　低レベル関数 add_device() を使用して、device_table にユーザ作成のデバイスを追加します。デバイス・テーブルは、*n* 個のデバイスをサポートする静的に定義された配列です。この *n* は、stdio.h の中にあるマクロ _NDEVICE で定義されます。デバイスに対応する構造体も stdio.h 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します。
function pointers	以下のデバイス・レベル関数へのポインタ

- 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・デバイス用に事前定義されています。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 関数の詳細については、7-34 ページを参照してください。

3) デバイスを追加したあと、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用します。

次のプログラムでは、C 入出力用デバイスの追加と使用が示されています。

```
#include <stdio.h>
/*************************************************************************/
/* Declarations of the user-defined device drivers                       */
/*************************************************************************/
extern int my_open(char *path, unsigned flags, int fno);
extern int my_close(int fno);
extern int my_read(int fno, char *buffer, unsigned count);
extern int my_write(int fno, char *buffer, unsigned count);
extern int my_lseek(int fno, long offset, int origin);
extern int my_unlink(char *path);
extern int my_rename(char *old_name, char *new_name);

main()
{
   FILE *fid;
   add_device(”mydevice”, _MSA, my_open, my_close, my_read, my_write,
                          my_lseek, my_unlink, my_rename);
   fid = fopen(”mydevice:test”,”w”);
   fprintf(fid,”Hello, world\n”);

   fclose(fid);
}
```

close　入出力用ファイルまたはデバイスのクローズ

C の構文

#include <stdio.h>
#include <file.h>
int close(int *file_descriptor***);**

C++ の構文

#include <cstdio.h>
#include <file.h>
int std::close(int *file_descriptor***);**

解説

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、オープンしているデバイスまたはファイルに関連付けられている低レベルのルーチン群に割り当てられたストリーム番号です。

戻り値

戻り値は、次のいずれかです。

0　　成功した場合
−1　　失敗した場合

lseek　ファイル位置標識の設定

C の構文

#include <stdio.h>
#include <file.h>
long lseek(int *file_descriptor***, long** *offset***, int** *origin***);**

C++ の構文

#include <cstdio.h>
#include <file.h>
long std::lseek(int *file_descriptor***, long** *offset***, int** *origin***);**

解説

lseek 関数は、指定されたファイルのファイル位置標識を *origin* + *offset* に設定します。ファイル位置標識では、ファイルの先頭から文字単位で位置が測定されます。

- *file_descriptor* は、デバイス・レベルのドライバがオープンしているファイルまたはデバイスに関連付ける低レベルのルーチン群に割り当てられた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、*offset* が測定される基準位置を指示するために使用します。*origin* は、次のマクロのいずれかによって戻される値にする必要があります。

SEEK_SET	(0x0000) ファイルの先頭
SEEK_CUR	(0x0001) ファイル位置標識の現行値
SEEK_END	(0x0002) ファイルの終わり

戻り値

戻り値は、次のいずれかです。

#	成功した場合、ファイル位置標識の新しい値
EOF	失敗した場合

open　入出力用ファイルまたはデバイスのオープン

C の構文

```
#include <stdio.h>
#include <file.h>
int open(const char *path, unsigned flags, int mode);
```

C++ の構文

```
#include <cstdio.h>
#include <file.h>
int std::open(const char *path, unsigned flags, int mode);
```

解説

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、パス情報も含みます。
- *flags* は、デバイスまたはファイルの操作方法を指定する属性です。次のシンボルを使用して指定します。

```
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 */
```

  デバイスでのデータの解釈方法によっては、これらのパラメータは無視できることもあります。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。

- *mode* は、必須ですが、値は無視されます。

戻り値

この関数は、次の値のいずれかを戻します。

#	成功した場合は、デバイス・レベルのドライバがオープンしているファイルまたはデバイスに関連付ける低レベル・ルーチン群によって割り当てられるストリーム番号
< 0	失敗した場合

read　バッファからの文字の読み取り

C の構文

#include <stdio.h>
#include <file.h>
int read(int *file_descriptor*, **char** **buffer*, **unsigned** *count*);

C++ の構文

#include <cstdio.h>
#include <file.h>
int std::read(int *file_descriptor*, **char** **buffer*, **unsigned** *count*);

解説

read 関数は、*count* で指定した数の文字を、*file_descriptor* に関連しているデバイスまたはファイルから読み取って *buffer* に入れます。

- *file_descriptor* は、オープンされたファイルまたはデバイスに関連している低レベル・ルーチンによって割り当てられるストリーム番号です。
- *buffer* は、読み取られた文字が入るバッファのロケーションです。
- *count* は、デバイスまたはファイルから読み取る文字数です。

戻り値

この関数は、以下の値のいずれかを戻します。

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　ファイル名の変更

C の構文

#include <stdio.h>
#include <file.h>
int rename(const char ***_old_name_, const char ***_new_name_**);**

C++ の構文

#include <cstdio.h>
#include <file.h>
int std::rename(const char ***_old_name_, const char ***_new_name_**);**

解説

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、以下の値のいずれかを戻します。

0	名前の変更が成功した場合
0 以外の値	失敗した場合

unlink　ファイルの削除

C の構文

#include <stdio.h>
#include <file.h>
int unlink(const char *_path_);

C++ の構文

#include <cstdio.h>
#include <file.h>
int std::unlink(const char *_path_);

解説

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、パス情報も含みます。

戻り値

この関数は、以下の値のいずれかを戻します。

0　　成功した場合
−1　　失敗した場合

write　バッファへの文字の書き込み

C の構文

#include <stdio.h>
#include <file.h>
int write(int *file_descriptor*, **const char *** *buffer*, **unsigned** *count*);

C++ の構文

#include <cstdio.h>
#include <file.h>
int std::write(int *file_descriptor*, **const char *** *buffer*, **unsigned** *count*);

解説

write 関数は、*count* で指定した文字数を、*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、オープンしているファイルまたはデバイスに関連付けられている低レベルのルーチン群に割り当てられたストリーム番号です。
- ❏ *buffer* は、書き込む文字が格納されてい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、以下の値のいずれかを戻します。

#　　成功した場合、書き込まれた文字数
−1　　失敗した場合

7.3 ヘッダ・ファイル

各ランタイムサポート関数は、ヘッダファイル内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、以下のとおりです。

- ❏ 一連の関連する関数（またはマクロ）
- ❏ 関数を使用するために必要な型
- ❏ 関数を使用するために必要なマクロ

ANSI C のランタイムサポート関数を宣言するヘッダ・ファイルは、次のとおりです。

assert.h	float.h	stdarg.h	string.h
ctype.h	limits	stddef.h	time.h
errno.h	math.h	stdio.h	
file.h	setjmp.h	stdlib.h	

ANSI C ヘッダ・ファイルに加えて、次の C++ ヘッダ・ファイルが組み込まれます。

cassert	cmath	cstdlib	rtti.h
cctype	csetjmp	cstring	stdexcept
cerrno	cstdarg	ctime	typeinfo
cfloat	cstddef	exception	
climits	cstdio	new	

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・ファイルを組み込む必要があります。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む必要があります。

C/C++ コンパイラで組み込まれるヘッダ・ファイルについては、7.3.1 項から 7.3.16 項までで説明します。

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) の場合は、プログラムは実行を継続します。
- ❑ 式が偽の場合は、assert マクロは、その式、ソース・ファイル名、および式を含む文の行番号が入っているメッセージを出力 します。その後、プログラムは (abort 関数により) 終了します。

assert.h/cassert ヘッダは、NDEBUG マクロを参照します (assert.h/cassert は、NDEBUG を定義しません)。assert.h/cassert を組み込むときに NDEBUG マクロが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 が定義されていない場合、assert は有効になります。

assert.h/cassert ヘッダは、NASSERT という別のマクロを参照します (assert.h/cassert は、NASSERT を定義しません)。assert.h/cassert を組み込むときに NASSERT マクロが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、コードを何も生成せず、assert で宣言されている式が真であることをオプティマイザに通知します。これは、どのような最適化が有効かを判断するためのヒントをオプティマイザに与えます。NASSERT が定義されていない場合、 assert は通常どおり有効になります。

assert 関数は、7-24 ページの表 7−3 (a) に掲載されています。

7.3.2 文字の判別と変換(ctype.h/cctype)

ctype.h/cctype ヘッダは、文字をテスト (判別) し、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、表示可能な文字か、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。そのマクロの 1 つに渡される引数に副次作用がある場合は、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が付きます (たとえば、*_isdigit*)。

文字入力関数と変換関数は、7-24 ページの 表 7−3 (b) に掲載されています。

7.3.3 エラー・レポート (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を関数に渡した場合、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 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 (ファイル位置のエラー)

算術関数を呼び出す C/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 低レベル入出力関数 (file.h)

file.h ヘッダは、入出力操作を実現するのに使用する低レベルの入出力関数を宣言します。7.2 節「C 入出力関数」では、C54x に関する入出力の実現方法を説明しています。

7.3.5 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 に、これらのマクロと、それに対応する制限値の一覧を示します。

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/climits)

マクロ	値	解説
CHAR_BIT	16	char 型のビット数
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	–32 768	short int の最小値
SHRT_MAX	32 767	short int の最大値
USHRT_MAX	65 535	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	–2 147 483 648	long int の最小値
LONG_MAX	2 147 483 647	long int の最大値
ULONG_MAX	4 294 967 295	unsigned long int の最大値
MB_LEN_MAX	1	マルチバイト char 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/cfloat)

マクロ	値	解説
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、最大の整数
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1.19209290e–07 1.19209290e–07 1.19209290e–07	1.0 + x ≠ 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 の n 乗が正規化した float、double、または long double の範囲内におさまるような整数n の最小値
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	38 38 38	10 の n 乗が表現可能な有限の float、double、または long double の範囲内におさまるような整数n の最大値

型接頭辞の意味: 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。

7.3.6 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 関数の計算では、引数として倍 精度浮動小数点値を入力し、倍精度浮動小数点値を戻します。

math.h/cmath ヘッダでは、*HUGE_VAL* というマクロを定義します。算術関数はこのマクロにより、範囲外の値を表現します。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。

7.3.7 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。次の定義が含まれます。

- *jmp_buf* は、呼び出し環境の復元に必要な情報の保持に適した配列型です。
- *setjmp* は、あとで longjmp 関数で使用できるように、呼び出し環境 jmp_buf 引数に保管するマクロです。
- *longjmp* は、jmp_buf 引数を使用してプログラム環境を復元する関数です。非ローカル jmp マクロおよび関数は、7-26 ページの表 7−3 (d) に掲載されています。

7.3.8 可変引数 (stdarg.h/cstdarg)

関数の中には、型が変化する可変 数の引数をもつものがあります。このような関数を可変引数関数といいます。stdarg.h/cstdarg ヘッダでは、可変引数関数の使用に役立つ 3 つのマクロと 1 つの型を宣言しています。

マクロは、*va_start*、*va_arg*、*va_end* の 3 つです。これらのマクロは、引数の数と型が関数の呼び出しごとに異なるときに使用します。

型 *va_list* は、*va_start*、*va_end*、*va_arg* の情報を保持できるポインタ型です。

可変引数関数は、stdarg.h/cstdarg で宣言されたマクロを使用して、実行時にその引数リストを 1 つずつ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ようにするには、可変引数関数への呼び出しがその関数のプロトタイプに対する可視性を備えていることを、必ず確認する必要があります。可変引数関数は、7-26 ページの表 7−3 (e) に掲載されています。

7.3.9 標準定義 (stddef.h/cstddef)

stddef.h/cstddef ヘッダは、2 つの型と 2 つのマクロを定義します。型は次のとおりです。

- *ptrdiff_t* 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* 演算子のデータ型を示す unsigned int 型です。

マクロは次のとおりです。

- *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 (type) の先頭から構造体メンバ (identifier) までのオフセットの値 (バイト単位) です。

以上の型とマクロは、一部のランタイムサポート関数で使用します。

7.3.10 入出力関数 (stdio.h/cstdio)

stdio.h/cstdio ヘッダでは、7 つのマクロ、2 つの型、1 つの構造体、および多数の関数が定義されています。その型と構造体は、次のとおりです。

- *size_t* 型 − *sizeof* 演算子のデータ型を示す unsigned int 型です。元の宣言は、stddef.h/cstdio にあります。
- *fpos_t* 型 − ファイル内のすべての位置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。

マクロは次のとおりです。

- *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/cstddef にあります。すでに定義されている場合には、再定義されません。
- *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 − 最長のファイル名の長さ (文字数) に展開されます。
- *L_tmpnam* マクロ − tmpnam() が生成できる最長のファイル名の文字列に展開されます。
- *SEEK_CUR*、*SEEK_SET*、*SEEK_END* マクロ − ファイル内の位置 (それぞれ現在位置、ファイルの先頭位置、ファイルの終端位置) 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*、 *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。

入出力関数は、7-27 ページの表 7−3 (f) に掲載されています。

7.3.11 汎用ユーティリティ (stdlib.h/cstdlib)

stdlib.h/cstdlib ヘッダは、多くの関数、 1 つのマクロ、および 2 つの型を宣言します。型は、次のとおりです。

- *div_t* − div 関数が戻す値の構造体の型
- *ldiv_t* − ldiv 関数が戻す値の構造体の型

マクロ *RAND_MAX* は、rand 関数が戻す最大のランダム値です。

次に示す共通ライブラリ関数の多くは、 stdlib.h/cstdlib ヘッダで宣言されます。

- 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や異常終了させます。
- 整数演算 − C 言語の標準部分としては提供されていません。

汎用ユーティリティ関数は、7-29 ページの表 7−3 (g)に掲載されています。

7.3.12 文字列関数(string.h/cstring)

string.h/cstring ヘッダでは、文字配列 (文字列) に関して次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、これを使えば、オブジェクトの最後が 0 になっていない任意のバイト・シーケンス (データ・オブジェクト) に対して、これと同じ処理ができます。これらの関数の名前の形式は、**mem***xxx* です。

文字列の移動やコピーを行う関数を使用するときは、デスティネーションに結果を格納するだけの十分な大きさがあることを確認しておいてください。文字列関数は、7-31 ページの表 7−3 (h) に掲載されています。

7.3.13 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、次の 2 通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、時刻を年、月、日、時、分、秒を組み合わせて表すためのメンバが含まれています。このように表示された時刻を詳細時刻と呼びます。この構造体には、以下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      */
```

時刻は、time_t または struct_tm のいずれの型で表される場合でも、次のように異なる計測基準で表すことができます。

- ❏ カレンダ時は、グレゴリオ暦で現在の日付と時刻を表します。
- ❏ 地方時は、特定のタイム・ゾーンで使用されるカレンダ時を表します。

時計関数およびマクロは、7-32 ページの表 7−3 (i) に掲載されています。

地方時は、夏時間に合わせて調整できます。当然、地方時はタイム・ゾーンによって異なります。time.h/ctime ヘッダでは、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、米国の中部標準時です。

time.h/ctime 内のすべての関数の基本となるのは、clock と time の 2 つのシステム関数です。time は現在の時刻 (time_t 形式) を示し、clock はシステム時刻 (任意の単位) を示します。clock によって戻される値は、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。

7.3.14 例外処理 (exception および stdexcept)

例外処理はサポートされていません。exception および stdexcept インクルード・ファイル (C++ の場合のみ) は、空のファイルです。

7.3.15 動的メモリ管理 (new)

new ヘッダ (C++ の場合のみ) は、new、new[]、delete、delete[]、およびこれらの配置バージョンのため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。

7.3.16 ランタイム型情報 (typeinfo)

typeinfo ヘッダ (C++ の場合のみ) は、実行時に C++ 型情報を示すために使用される type_info 構造体を定義します。

7.4 ランタイムサポート関数とマクロのまとめ

表 7−3 に、TMS320C54x ANSI C/C++ コンパイラが提供するランタイムサポート・ヘッダ・ファイルを (アルファベット順に) まとめてあります。ここで解説する関数のほとんどは ANSI 規格に準拠したものであり、この規格に定義されているとおりに動作します。

表 7−3 に掲載した関数とマクロは、7-33 ページの 7.5 節「ランタイムサポート関数とマクロの解説」で詳しく説明します。個々の関数またはマクロの詳細は、表の右端の欄に示すページを参照してください。

次の説明では、肩付きの数字で指数を表しています。たとえば、x^y は、x の y 乗を表します。

表 7−3.　ランタイムサポート関数とマクロのまとめ

(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。	7-37

(b) 文字の入力および変換関数 (ctype.h/cctype)

関数	説明	ページ
int **isalnum**(int c);	c が英数字 ASCII 文字かどうかをテストします。	7-57
int **isalpha**(int c);	c が英字 ASCII 文字かどうかをテストします。	7-57
int **isascii**(int c);	c が ASCII 文字かどうかをテストします。	7-57
int **iscntrl**(int c);	c が制御文字かどうかをテストします。	7-57
int **isdigit**(int c);	c が数字かどうかをテストします。	7-57
int **isgraph**(int c);	c がスペース以外の印刷文字かどうかをテストします。	7-57
int **islower**(int c);	c が ASCII の小文字かどうかをテストします。	7-57
int **isprint**(int c);	c が印刷可能な ASCII 文字 (スペースも含む) かどうかをテストします。	7-57
int **ispunct**(int c);	c が ASCII 句読文字かどうかをテストします。	7-57
int **isspace**(int c);	c が ASCII のスペース・バー、タブ (水平または垂直)、復帰、書式送り、または改行であるかどうかをテストします。	7-57
int **isupper**(int c);	c が ASCII の大文字かどうかをテストします。	7-57
int **isxdigit**(int c);	c が 16 進数字かどうかをテストします。	7-57
char **toascii**(int c);	c をマスクして有効な ASCII 値にします。	7-92
char **tolower**(int char c);	c が大文字の場合は小文字に変換します。	7-92
char **toupper**(int char c);	c が小文字の場合は大文字に変換します。	7-92

(c) 浮動小数点算術関数 (math.h/cmath)

関数	説明	ページ
double **acos**(double x);	x のアークコサインを戻します。	7-34
double **asin**(double x);	x のアークサインを戻します。	7-37
double **atan**(double x);	x のアークタンジェントを戻します。	7-38

(c) 浮動小数点算術関数 (math.h/cmath)(続き)

関数	説明	ページ
double **atan2**(double y, double x);	y/x のアークタンジェントを戻します。	7-38
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はインライン展開します。	7-41
double **cos**(double x);	x のコサインを戻します。	7-43
double **cosh**(double x);	x のハイパボリック・コサインを戻します。	7-43
double **exp**(double x);	e^x を戻します。	7-46
double **fabs**(double x);	x の絶対値を戻します。	7-46
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はインライン展開します。	7-50
double **fmod**(double x, double y);	x/y の厳密な浮動小数点剰余を戻します。	7-50
double **frexp**(double value, int *exp);	$.5 \leq \lvert f \rvert < 1$ であり、value が $f \times 2^{exp}$ に等しいという条件を満たす、f と exp を戻します。	7-53
double **ldexp**(double x, int exp);	$x \times 2^{exp}$ を戻します。	7-58
double **log**(double x);	x の自然対数を戻します。	7-60
double **log10**(double x);	x の常用対数を戻します。	7-60
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。	7-66
double **pow**(double x, double y);	x^y を戻します。	7-67
double **sin**(double x);	x のサインを戻します。	7-74
double **sinh**(double x);	x のハイパボリック・サインを戻します。	7-75
double **sqrt**(double x);	x の負でない平方根を戻します。	7-75
double **tan**(double x);	x のタンジェントを戻します。	7-90
double **tanh**(double x);	x のハイパボリック・タンジェントを戻します。	7-90

(d) 非ローカル・ジャンプ・マクロと関数 (setjmp.h/csetjmp)

関数またはマクロ	説明	ページ
int **setjmp**(jmp_buf env);	longjmp で使用する呼び出し環境を保存します。これはマクロです。	7-72
void **longjmp**(jmp_buf env, int _val);	jmp_buf 引数を使用して、前に保存されていた環境を復元します。	7-72

(e) 可変引数マクロ (stdarg.h/cstdarg)

マクロ	説明	ページ
type **va_arg**(va_list, type);	環境変数引数リスト内の型 type の次の引数にアクセスします。	7-93
void **va_end**(va_list);	va_arg を使用したあとで、呼び出しメカニズムをリセットします。	7-93
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。	7-93

(f) C 入出力関数 (stdio.h/cstdio)

関数	説明	ページ
int **add_device**(char *name, unsigned flags, int (*dopen)(), int (*dclose)(), int (*dread)(), int (*dwrite)(), fpos_t (*dlseek)(), int (*dunlink)(), int (*drename)());	デバイス・レコードをデバイス・テーブルに追加します。	7-34
void **clearerr**(FILE *_fp);	_fp が指すストリームの EOF 標識とエラー標識をクリアします。	7-42
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。	7-47
int **feof**(FILE *_fp);	_fp が指すストリームの EOF 標識をテストします。	7-48
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。	7-48
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。	7-48
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。	7-49
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に格納します。	7-49
char ***fgets**(char *_ptr, register int _size, register FILE *_fp);	_fp が指すストリームから、次の _size−1 個の文字を配列 _ptr に読み込みます。	7-49
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 が指す文字列に、ファイルのオープン方法が記述されています。	7-51
int **fprintf**(FILE *_fp, const char *_format, ...);	_fp が指すストリームへの書き込みを行います。	7-51
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。	7-51
int **fputs**(const char *_ptr, register FILE *_fp);	_ptr が指す文字列を _fp が指すストリームに書き込みます。	7-52
size_t **fread**(void *_ptr, size_t _size, size_t _count, FILE *_fp);	_fp が指すストリームからの読み込みを行い、入力データを _ptr が指す配列に格納します。	7-52
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 が指す文字列に、ファイルのオープン方法が記述されています。	7-53

(f) C 入出力関数 (stdio.h/cstdio) (続き)

関数	説明	ページ
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を読み込みます。	7-54
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。	7-54
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 には、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。	7-54
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。	7-55
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、1 ブロック分のデータを書き込みます。	7-55
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。	7-55
int **getchar**(void);	fgetc() を呼び出し、stdin (標準入力) を引数として提供するマクロです。	7-56
char ***gets**(char *_ptr);	fgets() と同じ機能を実行しますが、入力ストリームとして stdin を使用します。	7-56
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。	7-66
int **printf**(const char *_format, ...);	fprintf と同じ機能を実行しますが、出力ストリームとして stdout (標準出力) を使用します。	7-67
int **putc**(int _x, FILE *_fp);	fputc() と同じ機能を実行するマクロです。	7-68
int **putchar**(int _x);	fputc() を呼び出し、出力ストリームとして stdout を使用するマクロです。	7-68
int **puts**(const char *_ptr);	_ptr が指す文字列を stdout に書き込みます。	7-68
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。	7-71
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。	7-71
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。	7-71
int **scanf**(const char *_fmt, ...);	fscanf() と同じ機能を実行しますが、stdin から入力を読み込みます。	7-72

(f) C 入出力関数 (stdio.h/cstdio) (続き)

関数	説明	ページ
void **setbuf**(register FILE *_fp, char *_buf);	値を何も戻しません。setbuf() は、setvbuf() の制限付きバージョンで、バッファを定義してストリームに関連付けます。	7-72
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。	7-74
int **sprintf**(char *_string, const char *_format, ...);	fprintf() と同じ機能を実行しますが、_string が指す配列に出力を書き込みます。	7-75
int **sscanf** (const char *_str, const char *_fmt, ...);	fscanf() と同じ機能を実行しますが、_str が指す文字列から入力を読み込みます。	7-76
FILE ***tmpfile**(void);	一時ファイルを作成します。	7-91
char ***tmpnam**(char *_s);	有効なファイル名である文字列 (つまり、すでに使用されているファイル名以外のファイル名) を生成します。	7-91
int **ungetc**(int _c, register FILE *_fp);	_fp が指す入力ストリームに、_c で指定された文字をプッシュします。	7-93
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 で置き換えます。	7-94
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 で置き換えます。	7-95
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 で置き換えます。	7-95

(g) 汎用関数 (stdlib.h/cstdlib)

関数	説明	ページ
void **abort**(void);	プログラムを異常終了させます。	7-33
int **abs**(int i);	val の絶対値を戻します。−x0 が使用されていない場合はインライン展開します。	7-33
int **atexit**(void (*fun)(void));	fun が指す関数を登録します。この関数は、プログラム終了時に引数なしで呼び出されます。	7-39
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はインライン展開します。	7-39
int **atoi**(register const char *st);	文字列を整数に変換します。	7-39
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はインライン展開します。	7-39

(g) 汎用関数 (stdlib.h/cstdlib)(続き)

関数	説明	ページ
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。	7-40
void ***calloc**(size_t num, size_t size);	num 個のオブジェクト用のメモリ (それぞれ size バイト) の割り当ておよびクリアを行います。	7-41
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。	7-45
void **exit**(int status);	プログラムを正常終了させます。	7-46
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。	7-52
char ***getenv**(const char *_string)	_string に関連した変数の環境情報を戻します。	7-56
long **labs**(long i);	i の絶対値を戻します。−x0 が使用されていない場合はインライン展開します。	7-33
ldiv_t **ldiv**(register long numer, register long denom);	numer を denom で除算します。	7-45
int **ltoa**(long val, char *buffer);	val を等価の文字列に変換します。	7-61
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。	7-61
void **minit**(void);	malloc、calloc、または realloc によって以前に割り振られたメモリをすべてリセットします。	7-64
void **qsort**(void *base, size_t nmemb, size_t size, int (*compar) ());	nmemb 個のメンバで構成される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ます。size は、各メンバのサイズを示します。	7-69
int **rand**(void);	0 ～ RAND_MAX の範囲内で、疑似乱数整数のシーケンスを戻します。	7-69
void ***realloc**(void *packet, size_t size);	割り振り済みのメモリ空間のサイズを変更します。	7-70
void **srand**(unsigned int seed);	乱数発生ルーチンをリセットします。	7-69
double **strtod**(const char *st, char **endptr);	文字列を浮動小数点値に変換します。	7-88
long **strtol**(const char *st, char **endptr, int base);	文字列を倍長整数に変換します。	7-88
unsigned long **strtoul**(const char *st, char **endptr**,** int base)**;**	文字列を符号なしの倍長整数に変換します。	7-88

(h) 文字列関数 (string.h/cstring)

関数	説明	ページ
void ***memchr**(const void *cs, int c, size_t n);	cs の先頭の n 文字の中で最初に現れる c 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。	7-62
int **memcmp**(const void *cs, const void *ct, size_t n);	cs の先頭の n 文字を ct と比較します。−x が使用されている場合はインライン展開します。	7-62
void ***memcpy**(void *s1, const void *s2, register size_t n);	s1 から s2 に n 文字をコピーします。	7-63
void ***memmove**(void *s1, const void *s2, size_t n);	s1 から s2 に n 文字を移動します。	7-63
void ***memset**(void *mem, register int ch, register size_t length);	mem の先頭の length 文字に ch の値をコピーします。−x が使用されている場合はインライン展開します。	7-63
char ***strcat**(char *string1, const char *string2);	string2 を string1 の末尾に追加します。	7-76
char ***strchr**(const char *string, int c);	string の中で最初に現れる文字 c 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。	7-77
int **strcmp**(register const char *string1, register const char *s2);	文字列を比較して、次のいずれかの値を戻します。<0 (string1 が string2 より小さい場合); 0 (string1 がstring2 に等しい場合); > 0 (string1 が string2 より大きい場合)。−x が使用されている場合はインライン展開します。	7-78
int **strcoll**(const char *string1, const char *string2);	文字列を比較して、次のいずれかの値を戻します。<0 (string1 が string2 より小さい場合); 0 (string1 が string2 に等しい場合); > 0 (string1 が string2 より大きい場合)。	7-78
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はインライン展開します。	7-78
size_t **strcspn**(register const char *string, const char *chs);	chs の中に含まれていない文字のみで構成される string の最初の部分の長さを戻します。	7-79
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。	7-80
size_t **strlen**(const char *string);	文字列の長さを戻します。	7-82
char ***strncat**(char *dest, const char *src, register size_t n);	src の最大 n 個の文字を dest に追加します。	7-82
int **strncmp**(const char *string1, const char *string2, size_t n);	2 つの文字列の中の最大 n 個の文字を比較します。−x が使用されている場合はインライン展開します。	7-84

(h) 文字列関数 (string.h/cstring)(続き)

関数	説明	ページ
char ***strncpy**(register char *dest, register const char *src, register size_t n);	最大 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はインライン展開します。	7-85
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。	7-86
char ***strrchr**(const char *string, int c);	string の中で文字 c が最後に現れる位置を検索します。−x が使用されている場合はインライン展開します。	7-86
size_t **strspn**(register const char *string, const char *chs);	chs にある文字だけで構成される string の最初の部分の長さを戻します。	7-87
char ***strstr**(register const char *string1, const char *string2);	string1 の中で string2 が最初に現れる位置を検出します。	7-87
char ***strtok**(char *str1, const char *str2);	str1 を一連のトークンに分割し、各トークンを str2 の文字で区切ります。	7-89
size_t **strxfrm**(register char *to, register const char *from, register size_t n);	n 文字を from から to に変換します。	7-89

(i) 時間サポート関数 (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。	7-36
clock_t **clock**(void);	使用したプロセッサ時間を判別します。	7-42
char ***ctime**(const time_t *timer);	カレンダ時を地方時に変換します。	7-44
double **difftime**(time_t time1, time_t time0);	2 つのカレンダ時の差を戻します。	7-44
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。	7-57
struct tm ***localtime**(const time_t *timer);	time_t の値を詳細時刻に変換します。	7-59
time_t **mktime**(register struct tm *tptr);	詳細時刻を time_t の値に変換します。	7-65
size_t **strftime**(char *out, size_t maxsize, const char *format, const struct tm *time);	時間を書式化して文字列に変換します。	7-80
time_t **time**(time_t *timer);	現在のカレンダ時を戻します。	7-90

7.5 ランタイムサポート関数とマクロの解説

この節では、ランタイムサポート関数とマクロについて解説します。それぞれの関数とマクロについては、C および C++ の両方の構文が掲載されています。これは、C ヘッダ・ファイルを元とする関数とマクロであっても、プログラム例は C コードでのみ表示されるからです。C++ コードでは、同じプログラムは、ヘッダ・ファイルで宣言された型や関数が std ネームスペースで紹介されているという点で異なります。

abort 異常終了

C の構文

#include <stdlib.h>

void **abort**(void);

C++ の構文

#include <cstdlib>

void **std::abort**(void);

定義箇所

rts.src の exit.c

解説

abort 関数は、プログラムを終了させます。

例

```
void abort(void)
{
   exit(EXIT_FAILURE);
}
```

7-46 ページの exit 関数を参照してください。

abs/labs 絶対値

C の構文

#include <stdlib.h>

int **abs**(int j);
long **labs**(long i);

C++ の構文

#include <cstdlib>

int **std::abs**(int j);
long **std::labs**(long i);

定義箇所

rts.src の abs.c

解説

C/C++ コンパイラは、次のように、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 k の絶対値を戻します。

acos　アークコサイン

C の構文

```
#include <math.h>

double acos(double x);
```

C++ の構文

```
#include <cmath>

double std::acos(double x);
```

定義箇所

rts.src の asin.c

解説

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 */
```

add_device　デバイス・テーブルへのデバイスの追加

C の構文

```
#include <stdio.h>

int add_device(char *name,
               unsigned flags,
               int (*dopen)(),
               int (*dclose)(),
               int (*dread)(),
               int (*dwrite)(),
               fpos_t (*dlseek)(),
               int (*dunlink)(),
               int (*drename)());
```

C++ の構文

```
#include <cstdio>

int std::add_device(char *name,
               unsigned flags,
               int (*dopen)(),
               int (*dclose)(),
               int (*dread)(),
               int (*dwrite)(),
               fpos_t (*dlseek)(),
               int (*dunlink)(),
               int (*drename)());
```

定義箇所

rts.src の lowlev.c

解説

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 から入出力用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に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、fopen() を使用してください。その最初の引数に書式 *devicename:filename* の文字列を指定します。

- *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 デバイスでは、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 デバイスでは、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、および 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って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、7-5 ページの 7.2.1 項「低レベル入出力実装の概要」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、次の値のどれかを戻します。

0 成功した場合
−1 失敗した場合

例

この例では、次の操作が実行されます。

- デバイス *mydevice* をデバイス・テーブルに追加する。
- そのデバイス上で *test* という名前のファイルをオープンし、ファイル **fid* に関連付ける。
- 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);
}
```

asctime　内部時間から文字列への変換

C の構文

#include <time.h>

char *asctime(const struct tm *timeptr**);**

C++ の構文

#include <ctime>

char ***std::asctime**(const struct tm *timeptr);

定義箇所

rts.src の asctime.c

解説

asctime 関数は、詳細時刻を次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、7-20 ページの 7.3.13 項「時間関数 (time.h)」を参照してください。

asin　アークサイン

C の構文

#include <math.h>

double **asin**(double x);

C++ の構文

#include <cmath>

double **std::asin**(double x);

定義箇所

rts.src の asin.c

解説

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　診断情報挿入マクロ

C の構文

#include <assert.h>

void **assert**(int expr);

C++ の構文

#include <cassert>

void **std::assert**(int expr);

定義箇所

assert.h/cassert（マクロとして定義）

解説

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、実行を継続するかします。このマクロは、デバッグのときに便利です。

- ❏ expr が偽の場合、assert マクロは、失敗した特定の呼び出しに関する情報を標準出力に書き込み、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、別のマクロである NDEBUG を参照します。assert.h ヘッダをソース・ファイルに組み込むときに、NDEBUG がマクロ名として定義してあると、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、プログラムの実行を打ち切ります。

```
int   i, j;
assert(j);
q = i/j;
```

atan　極アークタンジェント

C の構文

#include <math.h>

double **atan**(double x);

C++ の構文

#include <cmath>

double **std::atan**(double x);

定義箇所

rts.src の atan.c

解説

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・アークタンジェント

C の構文

#include <math.h>

double **atan2**(double y, double x);

C++ の構文

#include <cmath>

double **std::atan2**(double y, double x);

定義箇所

rts.src の atan2.c

解説

atan2 関数は、y/x の逆タンジェントを戻します。この関数は、両方の引数の符号を使用して、戻り値の座標象限を定めます。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] ラジアンの角度です。

例

```
double rvalu, rvalv;
double radians;
rvalu  = 0.0;
rvalv  = 1.0;
radians = atan2(rvalr, rvalu);   /* return value = 0 */
```

atexit　Exit () で呼び出される関数の登録

C の構文

```
#include <stdlib.h>

int atexit(void (*fun)(void));
```

C++ の構文

```
#include <cstdlib>

int std::atexit(void (*fun)(void));
```

定義箇所

rts.src の exit.c

解説

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。関数は 32 個まで登録できます。

exit 関数の呼び出しによってプログラムが終了すると、登録された関数が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

C の構文

```
#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);
```

定義箇所

rts.src の atof.c、atoi.c、および atol.c

解説

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、書式送り文字、あるいは改行文字で表します。*space* の後にオプションの符号を表す *sign* が続きます。さらに数値の整数部を表す *digits* が続きます。その後には数値の小数部が続き、オプションの *sign* をもつ指数が続きます。

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 配列検索

C の構文

```
#include <stdlib.h>

void *bsearch(register const void *key, register const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
```

C++ の構文

```
#include <cstdlib>

void *std::bsearch(register const void *key, register const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
```

定義箇所

rts.src の bsearch.c

解説

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、次の値のどれかを戻します。

< 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　メモリの割り当てとクリア

C の構文

#include <stdlib.h>

void ***calloc**(size_t num, size_t size);

C++ の構文

#include <cstdlib>

void ***std::calloc**(size_t num, size_t size);

定義箇所

rts.src の memory.c

解説

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 (ヒープ) 内にあります。定数 _ _SYSMEM_SIZE では、ヒープのサイズは 1K ワードに定義されています。この量は、リンク時に変更できます。そのためには、−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 (ワード) を指定して、リンカを起動します。6-6 ページの 6.1.4 項「動的なメモリ割り当て」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

C の構文

#include <math.h>

double **ceil**(double x);

C++ の構文

#include <cmath>

double **std::ceil**(double x);

定義箇所

rts.src の ceil.c

解説

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 標識およびエラー標識の消去

C の構文

```
#include <stdio.h>
void clearerr(FILE *_fp);
```

C++ の構文

```
#include <cstdio>
void std::clearerr(FILE *_fp);
```

定義箇所

rts.src の clearerr

解説

clearerr 関数は、_fp が指すストリームの EOF 標識とエラー標識を消去します。

clock　プロセッサ時間

C の構文

```
#include <time.h>
clock_t clock(void);
```

C++ の構文

```
#include <ctime>
void std::clock(void);
```

定義箇所

rts.src の clock.c

解説

clock 関数は、使用したプロセッサ時間の合計を定めます。プログラムが実行を開始してから使用した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。したがって、ユーザはそれぞれシステム固有の clock 関数を記述する必要があります。また、clock() で戻る値 —クロックの目盛数—を CLOCKS_PER_SEC で割って値を秒数で表すことができるようにするためには、クロックの単位に基づいて CLOCKS_PER_SEC マクロを定義する必要もあります。

cos　コサイン

C の構文

#include <math.h>

double **cos**(double x);

C++ の構文

#include <cmath>

void **std::cos**(double x);

定義箇所

rts.src の cos.c

解説

cos 関数は、浮動小数点 x のコサインを戻します。x はラジアンで表した角度です。ラジアンで表します。引数の値が大きすぎると、有意性のある結果がほとんど得られないか、まった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

C の構文

#include <math.h>

double **cosh**(double x);

C++ の構文

#include <cmath>

void **std::cosh**(double x);

定義箇所

rts.src の cosh.c

解説

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

C の構文

#include <time.h>

char ***ctime**(const time_t *timer);

C++ の構文

#include <ctime>

char ***std::ctime**(const time_t *timer);

定義箇所

rts.src の ctime.c

解説

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、7-20 ページの 7.3.13 項「時間関数 (time.h)」を参照してください。

difftime　時間差

C の構文

#include <time.h>

double **difftime**(time_t time1, time_t time0);

C++ の構文

#include <ctime>

double **std::difftime**(time_t time1, time_t time0);

定義箇所

rts.src の difftime.c

解説

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、 7-20 ページの 7.3.13 項「時間関数 (time.h)」を参照してください。

div/ldiv　除算

C の構文

#include <stdlib.h>

div_t **div**(int numer, int denom);
ldiv_t **ldiv**(long numer, long denom);

C++ の構文

#include <cstdlib>

div_t **std::div**(int numer, int denom);
ldiv_t **std::ldiv**(long numer, long denom);

定義箇所

rts.src の div.c

解説

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が得られます。

- 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、次のように定義されています。

```
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、次のように定義されています。

```
typedef struct
{
   long int  quot;    /*  quotient    */
   long int  rem;     /* remainder    */
} ldiv_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 正常終了

C の構文

```
#include <stdlib.h>

void exit(int status);
```

C++ の構文

```
#include <cstdlib>

void std::exit(int status);
```

定義箇所

rts.src の exit.c

解説

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 (7-33 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。修正されなければ、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、呼び出し元には戻れないので注意してください。

exp 指数

C の構文

```
#include <math.h>

double exp(double x);
```

C++ の構文

```
#include <cmath>

double std::exp(double x);
```

定義箇所

rts.src の exp.c

解説

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

C の構文

#include <math.h>

double **fabs**(double x);

C++ の構文

#include <cmath>

double **std::fabs**(double x);

定義箇所

rts.src の fabs.c

解説

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 */
```

fclose ファイルのクローズ

C の構文

#include <stdio.h>

int **fclose**(FILE *_fp);

C++ の構文

#include <cstdio>

int **std::fclose**(FILE *_fp);

定義箇所

rts.src の fclose.c

解説

fclose 関数は、_fp が指すストリームをフラッシュし、そのストリームに関連付けられたファイルをクローズします。

feof　EOF 標識のテスト

C の構文

```
#include <stdio.h>
int feof(FILE *_fp);
```

C++ の構文

```
#include <cstdio>
int std::feof(FILE *_fp);
```

定義箇所

rts.src の feof.c

解説

feof 関数は、_fp が指すストリームの EOF 標識をテストします。

ferror　エラー標識のテスト

C の構文

```
#include <stdio.h>
int ferror(FILE *_fp);
```

C++ の構文

```
#include <cstdio>
int std::ferror(FILE *_fp);
```

定義箇所

rts.src の ferror.c

解説

ferror 関数は、_fp が指すストリームのエラー標識をテストします。

fflush　入出力バッファのフラッシュ

C の構文

```
#include <stdio.h>
int fflush(register FILE *_fp);
```

C++ の構文

```
#include <cstdio>
int std::fflush(register FILE *_fp);
```

定義箇所

rts.src の fflush.c

解説

fflush 関数は、_fp が指すストリームの入出力バッファをフラッシュします。

fgetc　次の文字の読み込み

C の構文

#include <stdio.h>

int **fgetc**(register FILE *_fp);

C++ の構文

#include <cstdio>

int **std::fgetc**(register FILE *_fp);

定義箇所

rts.src の fgetc.c

解説

fgetc 関数は、_fp が指すストリーム内の次の文字を読み込みます。

fgetpos　オブジェクトの格納

C の構文

#include <stdio.h>

int **fgetpos**(FILE *_fp, fpos_t *pos);

C++ の構文

#include <cstdio>

int **std::fgetpos**(FILE *_fp, fpos_t *pos);

定義箇所

rts.src の fgetpos.c

解説

fgetpos 関数は、_fp が指すストリームのファイル位置標識の現行値を、pos が指すオブジェクトに格納します。

fgets　次の複数文字の読み込み

C の構文

#include <stdio.h>

char ***fgets**(char *_ptr, register int _size, register FILE *_fp);

C++ の構文

#include <cstdio>

char ***std::fgets**(char *_ptr, register int _size, register FILE *_fp);

定義箇所

rts.src の fgets.c

解説

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、_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

C の構文

#include <math.h>

double **floor**(double x);

C++ の構文

#include <cmath>

double **std::floor**(double x);

定義箇所

rts.src の floor.c

解説

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 浮動小数点剰余

C の構文

#include <math.h>

double **fmod**(double x, double y);

C++ の構文

#include <cmath>

double **std::fmod**(double x, double y);

定義箇所

rts.src の fmod.c

解説

fmod 関数は、x を y で割った剰余の浮動小数点数を戻します。y = = 0 のとき、この関数は 0 を戻します。

例

```
double x, y, r;
x = 11.0;
y = 5.0;
r = fmod(x, y);            /* fmod returns 1.0 */
```

fopen　ファイルのオープン

C の構文

```
#include <stdio.h>

FILE *fopen(const char *_fname, const char *_mode);
```

C++ の構文

```
#include <cstdio>

FILE *std::fopen(const char *_fname, const char *_mode);
```

定義箇所

rts.src の fopen.cr

解説

fopen 関数は、_fname が指すファイルをオープンします。_mode が指す文字列には、ファイルのオープン方法が記述されています。

fprintf　ストリームの書き込み

C の構文

```
#include <stdio.h>

int fprintf(FILE *_fp, const char *_format, ...);
```

C++ の構文

```
#include <cstdio>

int std::fprintf(FILE *_fp, const char *_format, ...);
```

定義箇所

rts.src の fprintf.c

解説

fprintf 関数は、_fp が指すストリームへの書き込みを行います。_format が指す文字列には、ストリームへの書き込み方法が記述されています。

fputc　文字の書き込み

C の構文

```
#include <stdio.h>

int fputc(int _c, register FILE *_fp);
```

C++ の構文

```
#include <cstdio>

int std::fputc(int _c, register FILE *_fp);
```

定義箇所

rts.src の fputc.c

解説

fputc 関数は、_fp が指すストリームに 1 文字を書き込みます。

fputs 文字列の書き込み

C の構文

#include <stdio.h>

int **fputs**(const char *_ptr, register FILE *_fp);

C++ の構文

#include <cstdio>

int **std::fputs**(const char *_ptr, register FILE *_fp);

定義箇所

rts.src の fputs.c

解説

fputs 関数は、_fp が指すストリームに、_ptr が指す文字列を書き込みます。

fread ストリームの読み込み

C の構文

#include <stdio.h>

size_t **fread**(void *_ptr, size_t size, size_t count, FILE *_fp);

C++ の構文

#include <cstdio>

size_t **std::fread**(void *_ptr, size_t size, size_t count, FILE *_fp);

定義箇所

rts.src の fread.c

解説

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。入力データは、_ptr が指す配列に格納されます。読み込まれるオブジェクトの数は _count です。オブジェクトのサイズは _size です。

free メモリの解放

C の構文

#include <stdlib.h>

void **free**(void *ptr);

C++ の構文

#include <cstdlib>

void **std::free**(void *ptr);

定義箇所

rts.src の memory.c

解説

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、6-6 ページの 6.1.4 項「動的なメモリ割り当て」を参照してください。

例

次の例では 10 バイトを割り当ててから、解放します。

```
char *x;
x = malloc(10);         /*  allocate 10 bytes   */
free(x);                /*  free 10 bytes       */
```

freopen　ファイルのオープン

C の構文

#include <stdio.h>

FILE ***freopen**(const char *_fname, const char *_mode, register FILE *_fp);

C++ の構文

#include <cstdio>

FILE ***std::freopen**(const char *_fname, const char *_mode, register FILE *_fp);

定義箇所

rts.src の freopen.c

解説

freopen 関数は、_fname が指すファイルをオープンし、_fp が指すストリームに関連付けます。_mode が指す文字列は、ファイルのオープン方法を記述します。

frexp　小数部と指数部

C の構文

#include <math.h>

double **frexp**(double value, int *exp);

C++ の構文

#include <cmath>

double **std::frexp**(double value, int *exp);

定義箇所

rts.src の frexp.c

解説

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。この関数は、value == x × 2^{exp} となるような [1/2, 1] の範囲か 0 の数値 x を戻します。frexp 関数は、exp が指す整数にベキを保存します。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

C の構文

```
#include <stdio.h>

int fscanf(FILE *_fp, const char *_fmt, ...);
```

C++ の構文

```
#include <cstdio>

int std::fscanf(FILE *_fp, const char *_fmt, ...);
```

定義箇所

rts.src の fscanf.c

解説

fscanf 関数は、_fp が指すストリームからの読み込みを行います。 _fmt が指す文字列には、ストリームの読み込み方法が記述されています。

fseek　ファイル位置標識の設定

C の構文

```
#include <stdio.h>

int fseek(register FILE *_fp, long _offset, int _ptrname);
```

C++ の構文

```
#include <cstdio>

int std::fseek(register FILE *_fp, long _offset, int _ptrname);
```

定義箇所

rts.src の fseek.c

解説

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。その位置は、_ptrname で指定します。バイナリ・ファイルの場合、_offset を使用して、_ptrname からの標識の位置 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

C の構文

```
#include <stdio.h>

int fsetpos(FILE *_fp, const fpos_t *_pos);
```

C++ の構文

```
#include <cstdio>

int std::fsetpos(FILE *_fp, const fpos_t *_pos);
```

定義箇所

rts.src の fsetpos.c

解説

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 現在のファイル位置標識の取得

C の構文

```
#include <stdio.h>
long ftell(FILE *_fp);
```

C++ の構文

```
#include <cstdio>
long std::ftell(FILE *_fp);
```

定義箇所

rts.src の ftell.c

解説

ftell 関数は、_fp が指すストリームのファイル位置標識の現行値を取得します。

fwrite データ・ブロックの書き込み

C の構文

```
#include <stdio.h>
size_t fwrite(const void *_ptr, size_t _size, size_t _count, register FILE *_fp);
```

C++ の構文

```
#include <cstdio>
size_t std::fwrite(const void *_ptr, size_t _size, size_t _count, register FILE *_fp);
```

定義箇所

rts.src の fwrite.c

解説

fwrite 関数は、_ptr が指すメモリからデータ・ブロックを取り出し、_fp が指すストリームに書き込みます。

getc 次の文字の読み込み

C の構文

```
#include <stdio.h>
int getc(FILE *_fp);
```

C++ の構文

```
#include <cstdio>
int std::getc(FILE *_fp);
```

定義箇所

rts.src の fgetc.c

解説

getc 関数は、_fp が指すファイル内の次の文字を読み込みます。

getchar 標準入力からの次の文字の読み込み

C の構文

```
#include <stdio.h>
int getchar(void);
```

C++ の構文

```
#include <cstdio>
int std::getchar(void);
```

定義箇所 rts.src の fgetc.c

解説 getchar 関数は、標準入力デバイスから次の文字を読み込みます。

getenv 環境情報の取得

C の構文

```
#include <stdlib.h>
char *getenv(const char *_string);
```

C++ の構文

```
#include <cstdlib>
char *std::getenv(const char *_string);
```

定義箇所 rts.src の trgdrv.c

解説 getenv 関数は、_string で指定された環境変数の情報を戻します。

gets 標準入力からの次行の読み込み

C の構文

```
#include <stdio.h>
char *gets(char *_ptr);
```

C++ の構文

```
#include <cstdio>
char *std::gets(char *_ptr);
```

定義箇所 rts.src の fgets.c

解説 gets 関数は、標準入力デバイスから入力行を読み込みます。読み込まれた文字は、_ptr で指定された配列に入ります。

gmtime　グリニッジ標準時

C の構文

#include <time.h>

struct tm ***gmtime**(const time_t *timer);

C++ の構文

#include <ctime>

struct tm ***std::gmtime**(const time_t *timer);

定義箇所

rts.src の gmtime.c

解説

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、 7-20 ページの 7.3.13 項「時間関数 (time.h)」を参照してください。

isxxx　文字の判別

C の構文

#include <ctype.h>

int **isalnum**(int c);
int **isalpha**(int c);
int **isascii**(int c);
int **iscntrl**(int c);
int **isdigit**(int c);
int **isgraph**(int c);
int **islower**(int c);
int **isprint**(int c);
int **ispunct**(int c);
int **isspace**(int c);
int **isupper**(int c);
int **isxdigit**(int c);

C++ の構文

#include <cctype>

int **std::isalnum**(int c);
int **std::isalpha**(int c);
int **std::isascii**(int c);
int **std::iscntrl**(int c);
int **std::isdigit**(int c);
int **std::isgraph**(int c);
int **std::islower**(int c);
int **std::isprint**(int c);
int **std::ispunct**(int c);
int **std::isspace**(int c);
int **std::isupper**(int c);
int **std::isxdigit**(int c);

定義箇所

rts.src の isxxx.c と ctype.c
ctype.h/cctype（マクロとしても定義）

解説

以上の関数は、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	英数字の 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	英字の 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	英小文字の 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、書式送り文字、および改行文字を認識します。
isupper	英大文字の ASCII 文字を認識します。
isxdigit	16 進数字 (0～9、a～f、A～F) を認識します。

C/C++ コンパイラは、これらの関数と同じ機能をもつ一連のマクロもサポートしています。これらのマクロの名前は、これらの関数と同じですが、先頭にアンダースコアがつ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

7-33 ページの abs/labs を参照

ldexp

2 の累乗による乗算

C の構文

```
#include <math.h>

double ldexp(double x, int exp);
```

C++ の構文

```
#include <cmath>

double std::ldexp(double x, int exp);
```

定義箇所

rts.src の ldexp.c

解説

ldexp 関数は、浮動小数点数 x に exp で指定した 2 の累乗を掛け、$x \times 2^{exp}$ を戻します。指数部 (exp) は、負でも正でも指定できます。結果の値が大きすぎると、範囲エラーになります。

例

```
double result;
result = ldexp(1.5, 5);          /* result is 48.0 */
result = ldexp(6.0, -3);         /* result is 0.75 */
```

ldiv

7-45 ページの div/ldiv を参照

localtime

地方時

C の構文

```
#include <time.h>

struct tm *localtime(const time_t *timer);
```

C++ の構文

```
#include <ctime>

struct tm *std::localtime(const time_t *timer);
```

定義箇所

rts.src の localtime.c

解説

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、7-20 ページの 7.3.13 項「時間関数 (Time.h)」を参照してください。

log　自然対数

C の構文

#include <math.h>

double **log**(double x);

C++ の構文

#include <cmath>

double **std::log**(double x);

定義箇所

rts.src の log.c

解説

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。

解説

```
float x, y;
x = 2.718282;
y = log(x);          /* Return value = 1.0 */
```

log10　常用対数

C の構文

#include <math.h>

double **log10**(double x);

C++ の構文

#include <cmath>

double **std::log10**(double x);

定義箇所

rts.src の log10.c

解説

log10 関数は、底が 10 の実数 x の対数（常用対数）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　7-72 ページの setjmp/longjmp を参照

ltoa　倍長整数から ASCII への変換

C の構文

プロトタイプはありません。

int **ltoa**(long val, char *buffer);

C++ の構文

プロトタイプはありません。

int **std::ltoa**(long val, char *buffer);

定義箇所

rts.src の ltoa.c

解説

ltoa 関数は、標準外 (非 ANSI) 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、先頭の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　メモリ割り当て

C の構文

#include <stdlib.h>

void ***malloc**(size_t size);

C++ の構文

#include <cstdlib>

void ***std::malloc**(size_t size);

定義箇所

rts.src の memory.c

解説

malloc 関数は、size バイトの空間をオブジェクトに割り当て、その空間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)、ヌル・ポインタ (0) が戻ります。この関数は、割り当てたメモリの内容を変更しません。

malloc が使用するメモリは、特別なメモリ・プール (ヒープ) 内にあります。定数 _ _SYSMEM_SIZE により、ヒープのサイズは 1K ワードに定義されています。このサイズは、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希望するヒープ・サイズ (ワード) を指定して、リンカを起動します。詳細は、6-6 ページの 6.1.4 項「動的なメモリ割り当て」を参照してください。

memchr　バイトの最初の出現の検出

C の構文

#include <string.h>

void ***memchr**(const void *cs, int c, size_t n);

C++ の構文

#include <cstring>

void ***std::memchr**(const void *cs, int c, size_t n);

定義箇所

rts.src の memchr.c

解説

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 を指定できる点が違います。

memcmp　メモリ比較

C の構文

#include <string.h>

int **memcmp**(const void *cs, const void *ct, size_t n);

C++ の構文

#include <cstring>

int **std::memcmp**(const void *cs, const void *ct, size_t n);

定義箇所

rts.src の memcmp.c

解説

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、次のいずれかの値を戻します。

< 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違います。

memcpy　メモリ･ブロックのコピー — オーバーラップ非対応

C の構文

```
#include <string.h>

void *memcpy(void *s1, const void *s2, size_t n);
```

C++ の構文

```
#include <cstring>

void *std::memcpy(void *s1, const void *s2, size_t n);
```

定義箇所

rts.src の memcpy.c

解説

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、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･ブロックのコピー — オーバーラップ対応

C の構文

```
#include <string.h>

void *memmove(void *s1, const void *s2, size_t n);
```

C++ の構文

```
#include <cstring>

void *std::memmove(void *s1, const void *s2, size_t n);
```

定義箇所

rts.src の memmove.c

解説

memmove 関数は、s2 で指定したオブジェクトから s1 で指定したオブジェクトに n 文字を移動します。この関数は s1 の値を戻します。memmove 関数では、オーバーラップしたオブジェクト間でも正しく文字をコピーできます。

memset　メモリ内での値のコピー

C の構文

```
#include <string.h>

void *memset(void *mem, register int ch, size_t length);
```

C++ の構文

```
#include <cstring>

void *std::memset(void *mem, register int ch, size_t length);
```

定義箇所

rts.src の memset.c

解説

memset 関数は、ch の値を、mem が指すオブジェクトの先頭の length 文字にコピーします。この関数は、 mem の値を戻します。

minit 動的メモリ・プールのリセット

C の構文

プロトタイプはありません。

void **minit**(void);

C++ の構文

プロトタイプはありません。

void **std::minit**(void);

定義箇所

rts.src の memory.c

解説

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 1K ワードに定義されています。このサイズは、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希望するヒープ・サイズ (ワード) を指定して、リンカを起動します。詳細は、6-6 ページの 6.1.4 項「動的なメモリ割り当て」を参照してください。

注: 以前に割り当てられたオブジェクトは、minit のあとでは使用できなくなります。

minit 関数を呼び出すと、ヒープ内のメモリ空間すべてを再び使用できるようになります。以前に割り当てられたオブジェクトはすべてなくなります。これらのオブジェクトにはアクセスしないでください。

mktime　カレンダ時への変換

C の構文

#include <time.h>

time_t　***mktime**(struct tm *timeptr);

C++ の構文

#include <ctime>

time_t　***std::mktime**(struct tm *timeptr);

定義箇所

rts.src の mktime.c

解説

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、time_t 型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、7-20 ページの 7.3.13 項「時間関数 (time.h)」を参照してください。

例

この例では、2001 年の 7 月 4 日が何曜日になるかを求めます。

```
#include <time.h>
static const char *const wday[] = {
             "Sunday", "Monday", "Tuesday", "Wednesday",
             "Thursday", "Friday", "Saturday" };

struct tm time_str;

time_str.tm_year  = 2001 - 1900;
time_str.tm_mon   = 7;
time_str.tm_mday  = 4;
time_str.tm_hour  = 0;
time_str.tm_min   = 0;
time_str.tm_sec   = 1;
time_str.tm_isdst = 1;

mktime(&time_str);

/* After calling this function, time_str.tm_wday    */
/*    contains the day of the week for July 4, 2001 */
```

modf 符号付き整数と小数

C の構文

#include <math.h>

double **modf**(double value, double *iptr);

C++ の構文

#include <cmath>

double **std::modf**(double value, double *iptr);

定義箇所

rts.src の modf.c

解説

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

C の構文

#include <stdio.h>

void **perror**(const char *_s);

C++ の構文

#include <cstdio>

void **std::perror**(const char *_s);

定義箇所

rts.src の perror.c

解説

perror 関数は、エラー番号をマップして得られた文字列と s の文字列を結合し、エラー・メッセージを出力します。

pow　累乗

C の構文

#include <math.h>

double **pow**(double x, double y);

C++ の構文

#include <cmath>

double **std::pow**(double x, double y);

定義箇所

rts.src の pow.c

解説

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

C の構文

#include <stdio.h>

int **printf**(const char *_format, ...);

C++ の構文

#include <cstdio>

int **std::printf**(const char *_format, ...);

定義箇所

rts.src の printf.c

解説

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、ストリームへの書き込み方法を記述します。

putc 文字の書き込み

C の構文

```
#include <stdio.h>
int putc(int _x, FILE *_fp);
```

C++ の構文

```
#include <cstdio>
int std::putc(int _x, FILE *_fp);
```

定義箇所 rts.src の putc.c

解説 putc 関数は、_fp が指すストリームに 1 文字 _x を書き込みます。

putchar 標準出力への文字の書き込み

C の構文

```
#include <stdio.h>
int putchar(int _x);
```

C++ の構文

```
#include <cstdio>
int std::putchar(int _x);
```

定義箇所 rts.src の putchar.c

解説 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

C の構文

```
#include <stdio.h>
int puts(const char *_ptr);
```

C++ の構文

```
#include <cstdio>
int std::puts(const char *_ptr);
```

定義箇所 rts.src の puts.c

解説 puts 関数は、_ptr が指す文字列を標準出力デバイスに書き込みます。

qsort　配列のソート

C の構文

#include <stdlib.h>

void **qsort**(void *base, size_t nmemb, size_t size, int (*compar) ());

C++ の構文

#include <cstdlib>

void **std::qsort**(void *base, size_t nmemb, size_t size, int (*compar) ());

定義箇所

rts.src の qsort.c

解説

qsort 関数は、nmemb 個のメンバから成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、配列要素のキーを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

C の構文

#include <stdlib.h>

int **rand**(void);
void **srand**(unsigned int seed);

C++ の構文

#include <cstdlib>

int **std::rand**(void);
void **std::srand**(unsigned int seed);

定義箇所

rts.src の rand.c

解説

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

C の構文

```
#include <stdlib.h>

void *realloc(void *packet, size_t size);
```

C++ の構文

```
#include <cstdlib>

void *std::realloc(void *packet, size_t size);
```

定義箇所

rts.src の memory.c

解説

realloc 関数は、packet が指す割り当て済みのメモリのサイズを、size によってワード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ほうのサイズまで) 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、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 で 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。この操作によって解放されるメモリは、割り当てを解除されます。エラーが発生した場合、この関数は、ヌル・ポインタ (0) を戻します。

realloc が使用するメモリは、特別なメモリ・プールまたはヒープの中のメモリです。定数 _ _SYSMEM_SIZE により、ヒープのサイズは 1K ワードに定義されています。この量は、リンク時に変更できます。そのためには、−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 (ワード) を指定して、リンカを起動します。詳細は、6-6 ページの 6.1.4 項「動的なメモリ割り当て」を参照してください。

remove　ファイルの除去

C の構文

```
#include <stdio.h>
int remove(const char *_file);
```

C++ の構文

```
#include <cstdio>
int std::remove(const char *_file);
```

定義箇所

rts.src の remove.c

解説

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　ファイルの名前変更

C の構文

```
#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);
```

定義箇所

rts.src の rename.c

解説

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

C の構文

```
#include <stdio.h>
void rewind(register FILE *_fp);
```

C++ の構文

```
#include <cstdio>
void std::rewind(register FILE *_fp);
```

定義箇所

rts.src の rewind.c

解説

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

C の構文

#include <stdio.h>

int **scanf**(const char *_fmt, ...);

C++ の構文

#include <cstdio>

int **std::scanf**(const char *_fmt, ...);

定義箇所

rts.src の fscanf.c

解説

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を記述します。

setbuf　ストリーム用のバッファの指定

C の構文

#include <stdio.h>

void **setbuf**(register FILE *_fp, char *_buf);

C++ の構文

#include <cstdio>

void **std::setbuf**(register FILE *_fp, char *_buf);

定義箇所

rts.src の setbuf.c

解説

setbuf 関数は、_fp が指すストリームで使用されるバッファを指定します。_buf をヌルに設定すると、バッファリングはオフになります。値は戻りません。

setjmp/longjmp　非ローカル・ジャンプ

C の構文

#include <setjmp.h>

int **setjmp**(jmp_buf env)
void **longjmp**(jmp_buf env, int _val)

C++ の構文

#include <csetjmp>

int **std::setjmp**(jmp_buf env)
void **std::longjmp**(jmp_buf env, int _val)

定義箇所

rts.src の setjmp

解説

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、関数を 1 つずつ定義します。

- **jmp_buf** 型は、呼び出し環境の復元に必要な情報の保存に適した配列型です。
- **setjmp** マクロは、あとで longjmp 関数で使用できるように、呼び出し環境を jmp_buf 引数に保存します。

  直接の呼び出しからの戻りの場合、setjmp マクロは 0 を戻します。longjmp 関数の呼び出し結果として戻る場合、setjmp マクロは 0 以外の値を戻します。
- **longjmp** 関数は、setjmp マクロの一番最後の呼び出しで 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、 longjmp の動作は予測できません。

  longjmp が完了すると、対応する setjmp の呼び出しで _val に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。longjmp 関数は、_val が 0 でも setjmp に 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われます。

```
#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　バッファの定義とストリームとの関連付け

C の構文

#include <stdio.h>

int **setvbuf**(register FILE *_fp, register char *_buf, register int _type,
register size_t _size);

C++ の構文

#include <cstdio>

int **std::setvbuf**(register FILE *_fp, register char *_buf, register int _type,
register size_t _size);

定義箇所

rts.src の setvbuf.c

解説

setvbuf 関数は、_fp が指すストリームで使用されるバッファの定義と関連付けを行います。_buf をヌルに設定すると、バッファが割り当てられます。_buf でバッファを指定すると、そのバッファがストリーム用に使用されます。_size には、バッファのサイズを指定します。 _type には、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を行う
_IOLBF	行バッファリングを行う
_IONBF	バッファリングは行わない

sin　サイン

C の構文

#include <math.h>

double **sin**(double x);

C++ の構文

#include <cmath>

double **std::sin**(double x);

定義箇所

rts.src の sin.c

解説

sin 関数は、浮動小数点数 x のサインを戻します。x はラジアンで表した角度です。引数が大きすぎると、有意性のある結果がほとんど得られないか、まった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

C の構文

#include <math.h>

double **sinh**(double x);

C++ の構文

#include <cmath>

double **std::sinh**(double x);

定義箇所

rts.src の sinh.c

解説

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 */
```

sprintf　ストリームの書き込み

C の構文

#include <stdio.h>

int **sprintf**(char _string, const char *_format, ...);

C++ の構文

#include <cstdio>

int **std::sprintf**(char _string, const char *_format, ...);

定義箇所

rts.src の sprintf.c

解説

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、ストリームへの書き込み方法を記述します。

sqrt　平方根

C の構文

#include <math.h>

double **sqrt**(double x);

C++ の構文

#include <cmath>

double **std::sqrt**(double x);

定義箇所

rts.src の sqrt.c

解説

sqrt 関数は、実数 x の負でない平方根を戻します。引数が負の場合、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

7-69 ページの rand/srand を参照

sscanf

ストリームの読み込み

C の構文

#include <stdio.h>

int **sscanf**(const char *str, const char *format, ...);

C++ の構文

#include <cstdio>

int **std::sscanf**(const char *str, const char *format, ...);

定義箇所

rts.src の sscanf.c

解説

sscanf 関数は、str が指す文字列からの読み込みを行います。_format が指す文字列には、ストリームの読み込み方法を記述します。

strcat

文字列の連結

C の構文

#include <string.h>

char ***strcat**(char *string1, char *string2);

C++ の構文

#include <cstring>

char ***std::strcat**(char *string1, char *string2);

定義箇所

rts.src の strcat.c

解説

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、元は string1 の終了文字であったヌル文字を上書きします。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 文字の最初の出現の検出

C の構文

#include <string.h>

char ***strchr**(const char *string, int c);

C++ の構文

#include <cstring>

char ***std::strchr**(const char *string, int c);

定義箇所

rts.src の strchr.c

解説

strchr 関数は、string において最初に現れる c を検出します。strchr で目的の文字が検出されると、その文字へのポインタが戻ります。その文字が存在しない場合は、ヌル･ポインタ (0) が戻ります。

例

```
char *a = "When zz comes home, the search is on for zs.";
char *b;
char the_z = 'z';
b = strchr(a,the_z);
```

この例では、*b は zz の最初の z を指します。

strcmp/strcoll　文字列の比較

C の構文

#include <string.h>

int **strcmp**(const char *string1, 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);

定義箇所

rts.src の strcmp.c

解説

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 ANSI C との互換性に対応するための関数です。

この関数は、次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

C の構文

#include <string.h>

char ***strcpy**(char *dest, const char *src);

C++ の構文

#include <cstring>

char ***std::strcpy**(char *dest, const char *src);

定義箇所

rts.src の strcpy.c

解説

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

C の構文

#include <string.h>

size_t **strcspn**(const char *string, const char *chs);

C++ の構文

#include <cstring>

size_t **std::strcspn**(const char *string, const char *chs);

定義箇所

rts.src の strcspn.c

解説

strcspn 関数は、全体が chs 内にない文字から構成される string の最初の部分の長さを戻します。string の中の先頭の文字が chs にある場合、この関数は 0 を戻します。

例

```
char *stra = ″who is there?″;
char *strb = ″abcdefghijklmnopqrstuvwxyz″;
char *strc = ″abcdefg″;
size_t length;

length = strcspn(stra,strb);   /* length = 0 */
length = strcspn(stra,strc);   /* length = 9 */
```

strerror 文字列エラー

C の構文

```
#include <string.h>

char *strerror(int errno);
```

C++ の構文

```
#include <cstring>

char *std::strerror(int errno);
```

定義箇所

rts.src の strerror.c

解説

strerror 関数は、文字列「string error」を戻します。この関数は、ANSI との互換性に対応するための関数です。

strftime 時間の書式化

C の構文

```
#include <time.h>

size_t *strftime(char *s, size_t maxsize, const char *format,
                 const struct tm *timeptr);
```

C++ の構文

```
#include <ctime>

size_t *std::strftime(char *s, size_t maxsize, const char *format,
                 const struct tm *timeptr);
```

定義箇所

rts.src の strftime.c

解説

strftime 関数は、format 文字列に基づいて、(timeptr が指す) 時間を書式化し、その結果を文字列 s で戻します。s には、最高で、maxsize 文字数まで書き込めます。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	曜日名の省略形(Mon, Tue, . . .)
%A	省略しない曜日名
%b	月名 の省略形 (Jan, Feb, . . .)
%B	各地域ごとの省略しない月名
%c	日付と時刻の表現
%d	10 進数 (0～31) で表した日
%H	10 進数(00～23)で表した時刻
%I	10 進数(01～12)で表した時刻
%j	10 進数(001～366)で表した日
%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	日付の表現
%X	時刻の表現
%y	10 進数で表した世紀を表す最初の 2 桁を除いた年 (00～99)
%Y	10 進数で表した世紀を表す最初の 2 桁を付けた年
%Z	時間帯の名称(ない場合は空白)

time.h ヘッダで宣言と定義が行われる関数と型の詳細は、7-20 ページの 7.3.13 項「時間関数 (time.h)」を参照してください。

strlen 文字列の長さの検出

C の構文

#include <string.h>

size_t **strlen**(const char *string);

C++ の構文

#include <cstring>

size_t **std::strlen**(const char *string);

定義箇所

rts.src の strlen.c

解説

strlen 関数は、string の長さを戻します。C では、文字列は値が 0 の最初の文字（ヌル文字）で終了します。戻った結果にはヌル文字は含まれません。

例

```
char *stra = "who is there?";
char *strb = "abcdefghijklmnopqrstuvwxyz";
char *strc = "abcdefg";
size_t length;
length = strlen(stra);     /* length = 13 */
length = strlen(strb);     /* length = 26 */
length = strlen(strc);     /* length = 7  */
```

strncat 文字列の連結

C の構文

#include <string.h>

char ***strncat**(char *dest, const char *src, size_t n);

C++ の構文

#include <cstring>

char ***std::strncat**(char *dest, const char *src, size_t n);

定義箇所

rts.src の strncat.c

解説

strncat 関数は、s2（終了ヌル文字を含む）の最大 n 個の文字を dest に追加します。元の dest の終了文字のヌル文字は、src の先頭の文字上に上書きされます。strncat 関数は結果にヌル文字を付けます。この関数は、 dest の値を戻します。

例

次の例では、*a、*b、*c が指す文字列には、コメントに示されている値が割り当てられています。コメントの中の \0 の表記は、ヌル文字を表します。

```
char *a, *b, *c;
size_t size = 13;
.
.
.
/* a--> ″I do not like them,\0″             */;
/* b--> ″ Sam I am, \0″                     */;
/* c--> ″I do not like green eggs and ham\0″ */;

strncat (a,b,size);

/* a--> ″I do not like them, Sam I am, \0″   */;
/* b--> ″ Sam I am, \0″                     */;
/* c--> ″I do not like green eggs and ham\0″ */;

strncat (a,c,size);

/* a--> ″I do not like them, Sam I am, I do not like\0″  */;
/* b--> ″ Sam I am, \0″                                  */;
/* c--> ″I do not like green eggs and ham\0″             */;
```

strncmp　文字列の比較

C の構文

#include <string.h>

int **strncmp**(const char *string1, const char *string2, size_t n);

C++ の構文

#include <cstring>

int **std::strncmp**(const char *string1, const char *string2, size_t n);

定義箇所

rts.src の strncmp.c

解説

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、次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 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

C の構文

#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);

定義箇所

rts.src の strncpy.c

解説

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 文字未満のとき、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、「am」、および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 一致文字の検索

C の構文

#include <string.h>

char ***strpbrk**(const char *string, const char *chs);

C++ の構文

#include <cstring>

char ***std::strpbrk**(const char *string, const char *chs);

定義箇所

rts.src の strpbrk.c

解説

strpbrk 関数は、文字列の中で、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);
```

この例のあとでは、*a は wasn't の中の w を指します。

strrchr 文字の最後の出現の検出

C の構文

#include <string.h>

char ***strrchr**(const char *string, int c);

C++ の構文

#include <cstring>

char ***std::strrchr**(const char *string, int c);

定義箇所

rts.src の strrchr.c

解説

strrchr 関数は、string の中で文字 c が最後に現れる位置を検索します。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 一致文字数の検索

C の構文

#include <string.h>

size_t ***strspn***(const char *string, const char *chs);

C++ の構文

#include <cstring>

size_t ***std::strspn***(const char *string, const char *chs);

定義箇所

rts.src の strspn.c

解説

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 = strcspn(stra,strb);       /* length = 3 */
length = strcspn(stra,strc);       /* length = 0 */
```

strstr 一致文字列の検索

C の構文

#include <string.h>

char ***strstr***(const char *string1, const char *string2);

C++ の構文

#include <cstring>

char ***std::strstr***(const char *string1, const char *string2);

定義箇所

rts.src の strstr.c

解説

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は、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は、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、最初の文字列の what の中の w を指します。

strtod/strtol/ strtoul 文字列から数値への変換

C の構文

```
#include <stdlib.h>

double strtod(const char *st, char **endptr);
long strtol(const char *st, char **endptr, int base);
unsigned long strtoul(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);
```

定義箇所

rts.src の strtod.c、strtol.c、および strtoul.c

解説

これら 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、変換された文字列に続く最初の文字を指すように、このポインタを設定します。整数への変換を行う関数には、もう 1 つの引数 base もあります。この引数は、どの底で文字列を変換するかを関数に指示します。

- strtod 関数は、文字列を浮動小数点値に変換します。 文字列の形式は次のとおりです。

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

  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。変換後の値がオーバーフローになると、この関数は、± HUGE_VAL を戻します。変換後の値がアンダーフローになると、この関数は 0 を戻します。変換後の値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。文字列は、次の形式で指定します。

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

space は水平タブか垂直タブ、スペース・バー、復帰、書式送り、改行を組み 合わせて示します。space の後は、オプションの符号 sign、さらに数値の整数部を示す digits が続きます。そのあとは、数値の小数部が続き、オプションの符号 sign をもつ指数部が続きます。

認識できない文字が初めて出現した位置で、文字列は終わります。endptr が指すポインタは、この文字を指すように設定されます。

strtok 文字列のトークンへの分割

C の構文

#include <string.h>

char ***strtok**(char *str1, const char *str2);

C++ の構文

#include <cstdio>

char ***std::strtok**(char *str1, const char *str2);

定義箇所

rts.src の strtok.c

解説

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あと、ポインタ stra は文字列 excuse\0 を指します。これは、最初の空白のあった位置に strtok がヌル文字を挿入したからです。コメントの \0 はヌル文字を表します。

```
char *stra = ″excuse me while I kiss the sky″;
char *ptr;

ptr = strtok (stra,″ ″);    /* ptr –> ″excuse\0″  */
ptr = strtok (0,″ ″);       /* ptr –> ″me\0″      */
ptr = strtok (0,″ ″);       /* ptr –> ″while\0″   */
```

strxfrm 文字の変換

C の構文

#include <string.h>

size_t **strxfrm**(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);

解説

strxfrm 関数は、from が指す n 文字を、to が指す n 文字に変換します。

tan　タンジェント

C の構文

#include <math.h>

double **tan**(double x);

C++ の構文

#include <cmath>

double **std::tan**(double x);

定義箇所

rts.src の tan.c

解説

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。引数の値が大きすぎると、有意性のある結果がほとんど得られないか、まった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

C の構文

#include <math.h>

double **tanh**(double x);

C++ の構文

#include <cmath>

double **std::tanh**(double x);

定義箇所

rts.src の tanh.c

解説

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　時刻

C の構文

#include <time.h>

time_t **time**(time_t *timer);

C++ の構文

#include <ctime>

time_t **std::time**(time_t *timer);

定義箇所 rts.src の time.c

解説 time 関数は、秒で表される現在のカレンダ時を戻します。カレンダ時を使用できないときは、この関数は −1 を戻します。timer がヌル・ポインタでなければ、この関数は timer が指すオブジェクトに戻り値も代入します。

time.h ヘッダが宣言し定義する関数と型の詳細は、7-20 ページの 7.3.13 項「時間関数 (time.h)」を参照してください。

> **注: time 関数はターゲット・システム固有の関数です**
>
> time 関数はターゲット・システム固有の関数です。したがって、各ユーザはそれぞれの time 関数を記述する必要があります。

tmpfile 一時ファイルの作成

C の構文

```
#include <stdio.h>
FILE *tmpfile(void);
```

C++ の構文

```
#include <cstdio>
FILE *std::tmpfile(void);
```

定義箇所 rts.src の tmpfile.c

解説 tmpfile 関数は、一時ファイルを作成します。

tmpnam 有効なファイル名の生成

C の構文

```
#include <stdio.h>
char *tmpnam(char *_s);
```

C++ の構文

```
#include <cstdio>
char *std::tmpnam(char *_s);
```

定義箇所 rts.src の tmpnam.c

解説 tmpnam 関数は、有効なファイル名を示す文字列を生成します。

toascii　ASCII への変換

C の構文

```
#include <ctype.h>

int toascii(int c);
```

C++ の構文

```
#include <cctype>

int std::toascii(int c);
```

定義箇所

rts.src の toascii.c

解説

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、小文字の変換

C の構文

```
#include <ctype.h>

int tolower(int c);
int toupper(int c);
```

C++ の構文

```
#include <cctype>

int std::tolower(int c);
int std::toupper(int c);
```

定義箇所

rts.src の tolower.c
rts.src の toupper.c

解説

これら 2 つの関数は、単独の英字 c を大文字または小文字に変換します。

- tolower 関数は、大文字の引数を小文字に変換します。c が大文字でない場合、tolower はそのまま戻します。
- toupper 関数は、小文字の引数を大文字に変換します。c が小文字でない場合、toupper はそのまま戻します。

この関数には、同機能のマクロ _tolower と _toupper があります。

ungetc　ストリームへの文字の書き込み

C の構文

#include <stdio.h>

int **ungetc**(int c, FILE *_fp);

C++ の構文

#include <cstdio>

int **std::ungetc**(int c, FILE *_fp);

定義箇所

rts.src の ungetc.c

解説

ungetc 関数は、_fp が指すストリームに文字 c を戻します。

va_arg/va_end/va_start　可変引数マクロ

C の構文

#include <stdarg.h>

typedef char ***va_list;**
type **va_arg**(va_list, _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);

定義箇所

stdarg.h/cstdarg

解説

関数の中には、型が変化する可変数の引数で呼び出されるものがあります。このような関数 (可変引数関数) では、以下のようなマクロを使用して、実行時に引数リストを調べることができます。_ap パラメータは、可変引数リスト内の引数を指します。

- va_start マクロは、可変引数関数の引数リスト内の先頭の引数を指すように _ap を初期化します。parmN パラメータは、固定され宣言されたリストの中の右端のパラメータを指します。
- va_arg マクロは、可変引数関数に対する呼び出しで、次の引数の値を戻します。va_arg を呼び出すたびに _ap が変更されるので、可変引数関数の一連の引数は、va_arg に対する連続呼び出しにより戻されます (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 を初期化する必要があることに注意ください。

例

```
int   printf (char *fmt...)
      va_list ap;
      va_start(ap, fmt);
      .
      .
      .
      i = va_arg(ap, int);    /* Get next arg, an integer */
      s = va_arg(ap, char *); /* Get next arg, a string    */
      l = va_arg(ap, long);   /* Get next arg, a long      */
      .
      .
      .
      va_end(ap);             /* Reset                     */
}
```

vfprintf　ストリームへの書き込み

C の構文

#include <stdio.h>

int **vfprintf**(FILE *_fp, const char *_format, char *_ap);

C++ の構文

#include <cstdio>

int **std::vfprintf**(FILE *_fp, const char *_format, char *_ap);

定義箇所

rts.src の vfprintf.c

解説

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を記述します。引数リストは、_ap で指定します。

vprintf　標準出力への書き込み

C の構文

```
#include <stdio.h>

int vprintf(const char *_format, char *_ap);
```

C++ の構文

```
#include <cstdio>

int std::vprintf(const char *_format, char *_ap);
```

定義箇所

rts.src の vprintf.c

解説

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を記述します。引数リストは、_ap で指定します。

vsprintf　ストリームの書き込み

C の構文

```
#include <stdio.h>

int vsprintf(char *string, const char *_format, char *_ap);
```

C++ の構文

```
#include <cstdio>

int std::vsprintf(char *string, const char *_format, char *_ap);
```

定義箇所

rts.src の vsprintf.c

解説

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を記述します。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

TMS320C54x™ C/C++ コンパイラを使用すると、互いに互換性を維持する必要のない多くの構成や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・ライブラリに可能なすべての組み合わせを組み込む作業はかなり煩雑であることから、このパッケージには、ソース・アーカイブである rts.src が含まれています。rts.src には、ランタイムサポート関数がすべて入っています。

アーカイバと mk500 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。mk500 ユーティリティについては、本章で説明します。アーカイバについては、TMS320C54x 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

ライブラリ作成ユーティリティを起動する構文は、次のとおりです。

mk500 [*options*] *src_arch1* [**–l***obj.lib1*] [*src_arch2* [**–l***obj.lib2*]] ...

mk500	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。オプションは、コマンド行またはリンカ・コマンド・ファイルの任意の位置に指定できます（オプションについては、8.2 と 8.3 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。mk500 は、コマンド行オプションで指定されたランタイム・モデルに基いて、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、mk500 は、ソース・アーカイブの名前に拡張子 *.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。

mk500 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・ファイルに対してシェル・プログラムを実行します。次に、すべてのオブジェクト・ファイルが収集されて、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 C54X_C_OPTION、C_OPTION、C54X_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。次のオ プションは、ライブラリ作成ユーティリティにだけ適用します。

– –c ソース·アーカイブに含まれる C ソース·ファイルをライブラリから抽出します。ユーティリティの実行後は、これらをカレント・ディレクトリに残します。

– –h ソース・アーカイブに含まれるヘッダ・ファイルを使用します。このヘッダ・ファイルは、ユーティリティの実行完了後にカレント・ディレクトリに残されます。ツールに付属している rts.src アーカイブからランタイムサポート・ヘッダ・ファイルをインストールするときに、このオプションを使用します。

– –k ファイルを上書きします。デフォルトでは、このユーティリティが作成するオブジェクト・ファイルと同じ名前をもつオブジェクト・ファイルが、すでにカレント・ディレクトリ内に存在する場合、このユーティリティは終了します。この場合、ユーザが指定したオブジェクト・ファイル名であるか、またはユーティリティが生成したファイル名であるかどうかは関係ありません。

– –q ヘッダ情報を抑止します (静的)。

– –u2 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	シンボリック・デバッグを有効にします。	2-14

(b) パーサを制御するオプション

オプション	機能	ページ
−pi	定義制御されたインライン展開を無効にします (ただし、−o3 最適化は依然として自動インライン展開を実行します)。	2-37
−pk	コードを K&R 互換コードにします。	5-27
−pr	緩和モードを有効にします。また、厳格な ANSI 違反を無視します。	5-27
−ps	厳格な ANSI モードを有効にします (C に対して適用し、K&R C に対しては不適用)。	5-27

(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 (深刻ではない警告) を発行します。	2-30
−pdv	行折り返し付きのオリジナル・ソースを表示する詳細な診断を提供します。	2-31
−pdw	警告診断を抑止します (エラーは発行し続けます)。	2-31

(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します。	3-3
−o3	−o2 最適化とファイル最適化によってコンパイルします。mk500 によって−ol0 と −op0 が自動的に設定されるので注意してください。	3-3

(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。	3-11
−mf	呼び出しがすべてファー・コールになります。	2-15
−mn	−g によって無効になっていたオプティマイザ・オプションを有効にします。	3-15

(f) アセンブラを制御するオプション

オプション	機能	ページ
−as	ラベルをシンボルとして保持します。	2-20

(g) デフォルトのファイル拡張子を変更するオプション

オプション	機能	ページ
−ea[.]extension	アセンブリ・ファイルのデフォルト拡張子を設定します。	2-18
−eo[.]extension	オブジェクト・ファイルのデフォルト拡張子を設定します。	2-18

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、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

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。

dem500 [*options*][*filenames*]

dem500	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、dem500 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効果について、次に説明します。

−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	冗長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 TMS320C54x™ コンパイラによって出力されるアセンブリを示しています。例 9−1 (b) では、foo( ) とcompute( ) 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

```
int compute(int val, int *err);

int foo(short val, int *err)
{
   static int last_err = 0;
   int        result   = 0;

   if (last_err == 0)
       result = compute((int)val, &last_err);

   *err = last_err;
   return result;
}
```

(b) foo() の結果のアセンブリの一部

```
;*******************************************************
;* FUNCTION DEF: ___foo__FiPi                          *
;*******************************************************
'
___foo__FiPi:
: * A       assigned to _val
..... .bss  _last_err$1,1,0,0
..... PSHM ..... AR1
..... FRAME ..... #-4
..... NOP
;-------------------------------------------------------------
;  5  |    static int last_err = 0;
;-------------------------------------------------------------
..... STL ..... A,*SP(2)
;-------------------------------------------------------------
;  6  |    int result = 0;
;-------------------------------------------------------------
      ..... ST       #0, *SP(3)
;-------------------------------------------------------------
;  8  |    if (last_err == 0)
;-------------------------------------------------------------
..... LD ..... *(_last_err$1),A
..... BC ..... L1,ANEQ
...             ; branch occurs
;-------------------------------------------------------------
;  9  |    result = compute((int)val, &last_err);
;-------------------------------------------------------------
..... ST ..... #_last_err$1,*SP(0)
..... LD ..... *SP(2),A
..... CALL ..... #___compute__FiPi   ; call occurs
..... STL ..... A,*SP(3)
L1:
;-------------------------------------------------------------
;  11  |   *err = last_err;
;-------------------------------------------------------------
..... MVDK ..... *SP(6),*(AR1)
..... MVDK ..... (_last_err$1),*AR1
;-------------------------------------------------------------
;  13  |   return result;
;-------------------------------------------------------------
..... LD ..... *SP(3),A
..... FRAME ..... #4
..... POPM ..... AR1
..... RET
```

C++ ネーム・デマングラ・ユーティリティを実行すると、マングル対象と見なされるすべての名前をデマングルします。次のように入力すると仮定します。

```
% dem500 foo.asm
```

例 9−2 は、その結果を示しています。foo( ) と compute( ) 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・ユーティリティの実行後の結果

```
;***************************************************
;* FUNCTION DEF: foo(int, int *)                   *
;***************************************************
foo(int, int *):
: * A       assigned to _val
..... .bss  _last_err$1,1,0,0
..... PSHM ..... AR1
..... FRAME ..... #-4
..... NOP
;----------------------------------------------------------------
;  5  |    static int last_err = 0;
;----------------------------------------------------------------
..... STL ..... A,*SP(2)
;----------------------------------------------------------------
;  6  |    int result = 0;
;----------------------------------------------------------------
      ..... ST       #0, *SP(3)
;----------------------------------------------------------------
;  8  |    if (last_err == 0)
;----------------------------------------------------------------
..... LD ..... *(_last_err$1),A
..... BC ..... L1,ANEQ
...             ; branch occurs
;----------------------------------------------------------------
;  9  |    result = compute((int)val, &last_err);
;----------------------------------------------------------------
..... ST ..... #_last_err$1,*SP(0)
..... LD ..... *SP(2),A
..... CALL .....  #compute(int, int *)   ; call occurs
..... STL ..... A,*SP(3)
L1:
;----------------------------------------------------------------
;  11  |   *err = last_err;
;----------------------------------------------------------------
..... MVDK ..... *SP(6),*(AR1)
..... MVDK ..... (last_err$1),*AR1
;----------------------------------------------------------------
;  13  |   return result;
;----------------------------------------------------------------
..... LD ..... *SP(3),A
..... FRAME ..... #4
..... POPM ..... AR1
..... RET
```

付録 A

用語集

A

ANSI: 米国規格協会。業界の自主的規格を策定する組織

B

.bss セクション: デフォルトの COFF セクションの 1 つ。.bss 疑似命令を使って、メモリ・マップに一定量の空間を予約して、あとでそこに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セクションで、初期化されたデータを含んでいます。.data 疑似命令を使用すると、コードを .data セクションにアセンブルできます。

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、事実上の標準規格です。以前の ANSI に対応しない C コンパイラ用に記述された K&R C のプログラムの大部分は、修正なしで正しくコンパイルされ、実行されます。

T

.text セクション: デフォルトの COFF セクションの 1 つ。実行可能なコードが納められている初期化されたセクション。.text 疑似命令を使って、コードを .text セクションにアセンブルすることができます。

U

unsigned 値: 実際の符号に関係なく、正の数として取り扱われる値の一種

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 1 つ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･ライブラリの中のメンバを削除、抽出、置換することができます。

アーカイブ･ライブラリ: アーカイバによって 1 つのファイルにグループ化されたファイルの集合

アセンブラ: アセンブリ言語命令、疑似命令、およびマクロ疑似命令を含むソース･ファイルから機械語のプログラムを作成するソフトウェア･プログラム。アセンブラは、シンボリックな命令コードを絶対的な命令コードに換え、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･ユーティリティ: 元の C ソース文を、アセンブラから出力されるアセンブリ言語にコメントとして挿入するユーティリティ。C の文は、それに相当するアセンブリ命令の後ろに挿入されます。

え

エイリアス指定: 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。これを使用すると、コンパイラは、それらの式を自由に最適化できるようになります。

エピローグ: スタックの復元と戻りを行う関数のコード部分

エミュレータ: TMS320C54x™ のハードウェア上で、ソフトウェアを直接テストするために使用される開発システム

エントリ･ポイント: ターゲット･メモリの実行が開始される点

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･ファイルが入ったアーカイブ･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、または特定の機能を実行させるために使うコマンド･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。これによって、命令または疑似命令で実行される操作に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・エピローグの間にあるソフトウェア・パイプライン・ループの本体

外部シンボル: 現行プログラム･モジュールで使用されるが、別のプログラム･モジュールで定義または宣言されているシンボル

環境変数: ユーザが定義して文字列に割り当てるシステム・シンボル。多くの場合、環境変数はバッチ・ファイル (.cshrc など) に組み込まれます。

関数のインライン展開: 関数のコードを呼び出し点に挿入するプロセス。これによって関数呼び出しのオーバーヘッドが軽減され、オプティマイザは周囲のコードのコンテキストで関数の最適化を実行できます。

間接呼び出し: 1 つの関数から別の関数を呼び出す関数呼び出し。呼び出し先の関数のアドレスを渡すことによって行われます。

き

記憶クラス: シンボルへのアクセス方法を示す、シンボル･テーブルのエントリ

疑似命令: 特別な目的をもったコマンドで、ソフトウェア･ツールの動作、機能を制御するためのもの

共通オブジェクト・ファイル・フォーマット (COFF): セクションの概念を採用することにより、モジュラ・プログラミングを促進するバイナリ・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、メモリ空間内に別々に再配置できます。任意のセクションは、ターゲット・メモリに割り当てられた任意のブロックに配置できます。

く

グローバル・シンボル: 現行モジュール内で定義されて、別モジュール内でアクセスされる、または現行モジュール内でアクセスされるが、別モジュール内で定義されているシンボル

クロスリファレンス･リスト: アセンブラが作成する出力ファイルで、定義されたシンボル、シンボルを定義した行、シンボルを参照している行、およびその最終値がリストされています。

こ

構造体: グループ化されて 1 つの名前が付けられた、1 つまたは複数の変数の集まり

コード・ジェネレータ: パーサまたはオプティマイザによって生成されたファイルを受け取り、アセンブリ言語ソース・ファイルを生成するコンパイラ・ツール

コマンド・ファイル: リンカまたは Hex 変換ユーティリティのオプションが入っており、リンカまたは Hex 変換ユーティリティ用の入力ファイルを指定するファイル

コメント: ソース･ファイルを文書化したり、読みやすくするためのソース文 (または、その一部)。コメントは、コンパイル、アセンブル、およびリンクされません。つまり、オブジェクト･ファイルには何の効果も及ぼしません。

さ

再配置: シンボルのアドレスが変更されたときに、リンカがそのシンボルのすべての参照を調整するプロセス

し

シェル・プログラム: コンパイルとアセンブルを (場合によってはリンクも) 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、およびコード・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: 算術演算子で区切られた定数、シンボル、または一連の定数とシンボル

実行可能モジュール: ターゲット・システムで実行することができる、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、実行時に変数が初期化されます。

自動初期化: プログラムの実行を開始する前に、C のグローバル変数 (.cinit セクションに入っている) を初期化するプロセス

シミュレータ: C54x のハードウェアのないワークステーション上で、ソフトウェアを直接テストするために使用される開発システム

出力セクション: リンク後の実行可能モジュール内の最終的な割り当て済みセクション

出力モジュール: ターゲット・システム上にダウンロードして実行することのできる、リンクされた実行可能オブジェクト・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初期化されたデータを含む COFF セクション。初期化されたセクションは、.data 疑似命令、.text 疑似命令、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をもたない COFF セクション。このセクションは、.bss と .usect 疑似命令から構成されます。

シンボリック・デバッグ: シミュレータやエミュレータなどのデバッグ・ツールがシンボルを使用できるようにシンボルの情報を保持する、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: COFF オブジェクト・ファイルの中の、ファイルが定義して使用するシンボルについての情報を含んだ部分

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 つのプログラムに制限されている変数の一種。静的変数の値は、その関数またはプログラムが終了しても破棄されず、それらの関数またはプログラムが再び開始すると、前の値が再び使用されます。

セクション: 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・ファイル内の、そのファイルの中のセクションに関する情報を含んだ部分。各セクションには専用のヘッダがあり、そこに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でいるファイル。これをコンパイルまたはアセンブルして、オブジェクト・ファイルが作成されます。

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: 関数の名前を使用して、ある関数が別の関数を呼び出す関数呼び出し

て

定数: 値を変更できない数値

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。

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、これらの関数を使って、ヒープからメモリを割り当てることによって行います。

は

パーサ: ソース･ファイルを読み取り、前処理機能を実行し、構文をチェックし、オプティマイザやコード･ジェネレータの入力として使用できる中間ファイルを生成するソフトウェア･ツール

バイト: 1 文字を格納できる記憶域としてアドレス可能な単位。C54x では、1 バイトは 16 ビットです。

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指定方式。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)。

プラグマ: コンパイラに対し、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: マクロ定義とマクロ展開、ヘッダ・ファイルの解釈、条件付きコンパイル、およびプリプロセッサ疑似命令を処理するソフトウェア･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: 中かっこ ({}) でまとめられ、エンティティとして使われる一連の文

へ

変数: 一連の値のうちのいずれかをとる、ある量を表すシンボル

ま

マクロ: ユーザが定義した、命令として使うことができるルーチン

マクロ定義: マクロを構成する名前とコードを定義するソース文のブロック

マクロ展開: マクロ呼び出しに代わって、コードにソース文を挿入するプロセス

マクロ呼び出し: マクロを起動するプロセス

マップ･ファイル: リンカによって作成される出力ファイルで、メモリ構成、セクション構成、セクションの割り当て、シンボル定義、さらに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で、複数の機能ブロックに区画分けされています。実行時のパラメータで、プログラムはその範囲内で機能しなければなりません。

ら

ラベル: アセンブリ･ソース文の 1 カラム目で始まるシンボルで、その文のアドレスに対応します。ラベルは、1 カラム目から始まることのできる唯一のアセンブラ文です。

ランタイム（実行時）環境: プログラムを実行する必要のあるランタイム・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って定義されます。

ランタイムサポート関数: C 言語には含まれない作業（メモリの割り当て、文字列の変換、文字列の検索など）を実行する ANSI 標準関数

ランタイムサポート･ライブラリ: ランタイムサポート関数のソースが納められている、ライブラリ･ファイル rts.src

り

リスト･ファイル: アセンブラによって作成される出力ファイルで、ソース文、その行番号、セクション・プログラム・カウンタ (SPC) への効果をリストします。

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で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付け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: 複数のオブジェクト・ファイルを結合して 1 つのオブジェクト・モジュールを作成するソフトウェア・プログラム。このモジュールはシステム・メモリ内に割り当てられ、デバイスによって実行されます。

る

ループの展開: 小さいループを展開してループの各反復がコードに表示できるようにする最適化。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、−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

数字

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。

索引

記号

数字

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

あ

い

え

お

か

き

く

け

こ

さ

す

せ

ひ

ふ

へ

ほ

ま

み

む

め

り

る

れ

ろ

わ

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)
横浜オフィス　〒222-0033　横浜市港北区新横浜2丁目3番12号　新横浜スクエアビル3階　045(476)7870(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/

TMS320C54x

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 2 版　2001年 9月
第 1 版　1999年10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
’01.09

日本テキサス・インスツルメンツ株式会社

SPRU367A