TEXAS INSTRUMENTS

TMS320C2x/C2xx/C5x

オプティマイジング (最適化) C コンパイラ

ユーザーズ・マニュアル

2000 *Microprocessor Development Systems*

SPRU024E 翻訳版

JAJU001 WAS SCJ2222A

TMS320C2x/C2xx/C5x
オプティマイジング (最適化)
C コンパイラ
ユーザーズ・マニュアル

2000 年 6 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資料が最新のものであることを確実にするため、最新版の資料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ー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、また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。

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、露結しないこと。)
   - 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

TEXAS INSTRUMENTS

最初にお読みください

このマニュアルについて

本書では、以下のコンパイラ・ツールの使用方法について説明しています。

- ❏ コンパイラ
- ❏ ソース・インターリスト・ユーティリティ
- ❏ オプティマイザ
- ❏ パーサ
- ❏ ライブラリ作成ユーティリティ

本コンパイラは、米国規格協会 (ANSI) 準拠の標準 C ソース・コードを受け付け、TMS320C2x/C2xx/C5x デバイス用のアセンブリ言語ソース・コードを生成します。本書では、C コンパイラの特性について説明します。本書では、読者は 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、カーニハンとリッチーの *The C Programming Language* (第 2 版) に解説しています。必要に応じて、本書の参考文献としてお読みください。

本書の C コンパイラに関する情報を使用する前に、*TMS320C2xx Code Generation Tools Installation Instructions* を参照して、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。

本書のご利用方法

本書の目的は、TMS320C2x/C2xx/C5x デバイス用に特別に作成された、当社の C コンパイラ・ツールの使用方法を習得するために役立つ情報を提供することです。本書の構成は次のとおりです。

- 第 1 章**「はじめに」**では、TMS320C2x/C2xx/C5x 開発ツールの概要を説明します。
- 第 2 章、第 5 章、第 6 章、第 7 章、および第 8 章の**「コンパイラについて」**では、C コンパイラとシェル・プログラムの操作方法、および ANSI C 仕様に関連する C コンパイラ固有の特性について説明します。TMS320C2x/ C2xx/C5x アーキテクチャに関する技術情報、およびアセンブリ言語を C プログラムにインターフェイスするときに必要な情報も含まれています。マクロ、関数、およびそこで宣言される型のほかに、ライブラリとヘッダ・ファイルについても説明します。最後に、ライブラリ作成ユーティリティについて説明します。
- 付録 A には、**「参考資料」**として用語集が収められています。

表記規則

本書では、以下の表記規則を使用しています。

- 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）で示しています。例は、強調のため、ボールド (**bold version**) で示しています。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) とを区別するために、ボールド (**bold version**) で示しています。

  以下に、C コードの例を示します。

```
# ifdef  NDEBUG
# define assert
```

- 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 (*italics*) で示しています。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。以下に、疑似命令の構文例を示します。

  #include “*filename*”

  #include プリプロセッサ疑似命令には、*filename* という必須パラメータが 1 つあります。ファイル名は、二重引用符 (“ ”) または不等号括弧 (< >) で囲みます。

❑ 大括弧 ([ ]) は、任意のパラメータを示しています。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。

clist *asmfile* [*outfile*] [*–options*]

- **clist** コマンドには、3 つのパラメータがあります。
- 最初のパラメータ *asmfile* は必須です。
- 2 番目と 3 番目のパラメータ *outfile* と*–options* は任意です。
- outfile を省略すると、アセンブリ・ファイルに拡張子 *.cl* を付けた名前になります。
- オプションの先頭にはハイフンが付きます。

当社発行の関連文献

以下の文献は、TMS320C2x/C2xx/C5x および関連サポート・ツールの解説書です。当社の刊行物を入手するには、タイトルと文献番号をご確認の上、プロダクト・インフォメーション・センター (PIC)、0120−81−0026 にお問い合わせください。

TMS320C2x User's Guide (文献番号 SPRU014) は、'C2x 固定小数点ディジタル・シグナル・プロセッサのハードウェアについて解説しています。ピンの割り当て、アーキテクチャ、命令セット、および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も説明しています。また、すべての 'C2x デバイスの電気的仕様、およびパッケージ・メカニカル・データの情報が含まれています。本書には、'C1x から 'C2x への DSP システムの移行に関する節も含まれています。

TMS320C2xx User's Guide (文献番号 SPRU127) は、'C2xx 固定小数点ディジタル・シグナル・プロセッサのハードウェアについて解説しています。ピンの割り当て、アーキテクチャ、命令セット、および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も説明しています。また、すべての 'C2xx デバイスの電気的仕様、およびパッケージ・メカニカル・データの情報が含まれています。本書には、'C2x から 'C2xx への命令を比較した節も含まれています。

TMS320C5x User's Guide (文献番号 SPRU056) は、TMS320C5x 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、命令セット、パイプライン、仕様、DMA、および入出力ポートについて記載しています。ソフトウェアのアプリケーションについては、独立した章で解説しています。

TMS320C1x/C2x/C2xx/C5x アセンブリ言語ツール ユーザーズ・マニュアル (文献番号 SPRU018) は、'C1x、'C2x、'C2xx、および 'C5x 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。

TMS320C2x Software Development System Technical Reference (文献番号 SPRU072) は、TMS320C2x ソフトウェア開発システム (SWDS) ボードを使用するアプリケーションと設計についての情報を提供しています。

TMS320C5x Software Development System Technical Reference (文献番号 SPRU066) は、TMS320C5x ソフトウェア開発システム (SWDS) ボードを使用するアプリケーションと設計についての情報を提供しています。

TMS320C2x C Source Debugger User's Guide (文献番号 SPRU070) は、'C2x エミュレータ、ソフトウェア開発システム (SWDS)、およびシミュレータで 'C2x C ソース・デバッガを使用する方法を解説しています。

TMS320C5x C Source Debugger User's Guide (文献番号 SPRU055) は、'C5x エミュレータ、EVM、および C ソース・デバッガ・インターフェイスのシミュレータ・バージョンの起動方法について解説しています。本書では、デバッガ・インターフェイスを、ウィンドウ管理、コマンド入力、コード実行、データ管理、およびブレークポイントなどのさまざまな観点から解説しています。また、デバッガの基本機能を学習するためのチュートリアルも含まれています。

関連文献

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

以下の文献も参考にしてください。

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)

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

商標

サポートが必要な場合・・・

❑ World-Wide Web Sites

TI Online	http://www.ti.com
日本 TI	http://www.tij.co.jp
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/pic/home.htm
DSP Solutions	http://www.ti.com/dsps
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.htm

❑ North America, South America, Central America

Product Information Center (PIC)	(972) 644-5580		
TI Literature Response Center U.S.A.	(800) 477-8924		
Software Registration/Upgrades	(214) 638-0333	Fax: (214) 638-7742	
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285		
U.S. Technical Training Organization	(972) 644-5580		
DSP Hotline	(281) 274-2320	Fax: (281) 274-2324	Email: dsph@ti.com
DSP Modem BBS	(281) 274-2323		

DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/pub/tms320bbs

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:			
Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32	Email: epic@ti.com
Deutsch	+33 1 30 70 11 68		
English	+33 1 30 70 11 65		
Francais	+33 1 30 70 11 64		
Italiano	+33 1 30 70 11 67		
EPIC Modem BBS	+33 1 30 70 11 99		
European Factory Repair	+33 4 93 22 25 40		
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10	

❑ Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200	
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002	
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828	
Korea DSP Modem BBS	+82 2 551 2914		
Singapore DSP Hotline		Fax: +65 390 7179	
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718	
Taiwan DSP Modem BBS	+886 2 376 2592		

Taiwan DSP Internet BBS via anonymous ftp to ftp://dsp.ee.tit.edu.tw/pub/TI/

❑ 日本

TI 製品についてのお問い合わせ	プロダクト・インフォメーション・センター (PIC) へご連絡ください。		
および資料の請求	0120-81-0026	Fax: 0120-81-0036	Email: pic-japan@ti.com

❑ Documentation

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251-1443

Email: comments@books.sc.ti.com

Note: When calling a Literature Response Center to order documentation, please specify the literature number of the book.

目次

図

表

例

注

第 1 章

はじめに

TMS320C2x、TMS320C2xx、および TMS320C5x のデバイスは、TMS320 ファミリーを構成する高性能な CMOS マイクロプロセッサで、ディジタル・シグナル処理アプリケーション用に最適化されています。

TMS320C2x/C2xx/C5x の DSP は、ソフトウェア開発ツールのセットによりサポートされています。このセットには、オプティマイジング (最適化) C コンパイラ、アセンブラ、リンカ、アーカイバ、ソフトウェア・シミュレータ、フルスピード・エミュレータ、およびソフトウェア開発ボードが含まれています。

本章では、これらのツールの概要と最適化 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、*TMS320C1x/C2x/C2xx/C5x* 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。

注: バージョン情報

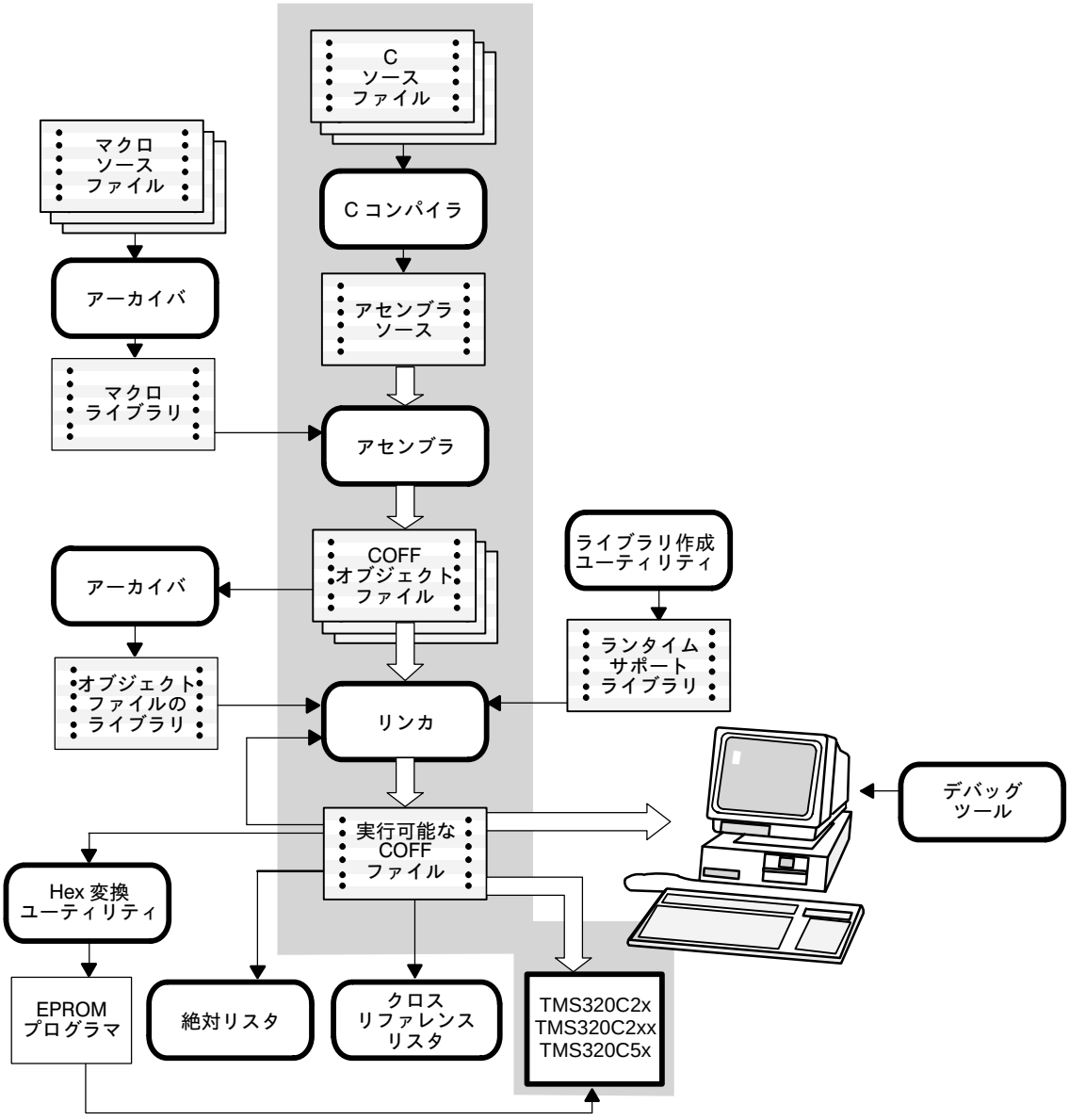
TMS320C2x/C2xx/C5x C コンパイラを使用するには、TMS320C1x/C2x/C2xx/C5x アセンブラおよびリンカのバージョン 5.0 (以上) も必要になります。

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、TMS320C2x/C2xx/C5x ソフトウェア開発のフローを示して います。図の中の陰影を付けた部分は、C 言語プログラムのソフト ウェア開発における最も一般的なパスです。それ以外の部分は、開発プロセスを補強する周辺機能です。

図 1−1. TMS320C2x/C2xx/C5x ソフトウェア開発のフロー



以下のリストは、図 1−1 に示したツールについて説明しています。

- **C コンパイラ**は、C ソース・コードを取り込んで、TMS320C2x、TMS320C2xx、または TMS320C5x 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、およびリンクを自動的に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 プログラムの効率を向上させます。
  - **インターリスト・ユーティリティ**は、C ソース文をアセンブリ言語の出力に差し込みます。

コンパイラ、シェル、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と操作方法については、第 2 章を参照してください。

- **アセンブラ**は、アセンブリ言語のソース・ファイルを機械語のオブジェクト・ファイルに変換します。この機械語は、COFF (共通オブジェクト・ファイル・フォーマット) に基づいています。アセンブラの使用方法については、*TMS320C1x/C2x/C2xx/C5x* アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。
- **リンカ**は、複数のオブジェクト・ファイルを結合して 1 つの実行可能なオブジェクト・モジュールを作成します。またリンカは、実行可能モジュールを作成する際に再配置を行い、外部参照の解決をします。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。
- **アーカイバ**を使用すると、複数のファイルをライブラリと呼ばれる 1 つのアーカイブ・ファイルにまとめることができます。さらにアーカイバでは、メンバの削除、置換、抽出、または追加により、ライブラリの内容を変更できます。アーカイバは、オブジェクト・モジュールのライブラリを作成するときに大変便利なツールです。アーカイバの使用方法については、*TMS320C1x/C2x/C5x* アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。C コンパイラには、以下の 3 つのオブジェクト・ライブラリが同梱されています。
  - **rts25.lib** には、TMS320C2x 用の ANSI 標準ランタイムサポート関数とコンパイラ・ユーティリティが含まれています。
  - **rts50.lib** には、TMS320C5x 用の ANSI 標準ランタイムサポート関数とコンパイラ・ユーティリティが含まれています。
  - **rts2xx.lib** には、TMS320C2xx 用の ANSI 標準ランタイムサポート関数とコンパイラ・ユーティリティが含まれています。
- **ライブラリ作成ユーティリティ**を使用すると、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。標準ランタイムサポート・ライブラリ関数は、ソース・コードとして rts.src に収められています。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。

- ❑ **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。変換されたファイルは、EPROM プログラマにダウンロードできます。

- ❑ **絶対リスタ**は、デバッグ・ツールです。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。.abs ファイルをアセンブルすると、相対アドレスではなく絶対アドレスを含んだリストを生成できます。絶対リスタの使用方法については、*TMS320C1x/C2x/C2xx/C5x* アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

- ❑ **クロスリファレンス・リスタ**は、オブジェクト・ファイルからクロスリファレンス・リストを生成します。生成されたリストには、シンボル、シンボルの定義、およびリンク先ソース・ファイル内のシンボル参照リストなどが表示されています。クロスリファレンス・リスタの使用方法については、*TMS320C1x/C2x/C2xx/C5x* 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、**TMS320C2x/C2xx/C5x デバイス**で実行できるモジュールを生成することです。いずれかのデバッグ・ツールを使用すれば、生成したコードに改善や修正を加えることができます。使用できるデバッグ・ツールを次に示します。

  - ■ 命令の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 (XDS-510) エミュレータ
  - ■ 評価モジュール (EVM)

1.2 C コンパイラの概要

TMS320C2x/C2xx/C5x C コンパイラは、標準 ANSI C プログラムを TMS320C2x/C2xx/C5x 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。

- **ANSI 標準 C**

  TMS320C2x/ C2xx/C5x 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* (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。

- **ANSI 標準ランタイム・サポート**

  本コンパイラ・ツールには、完全なランタイム・ライブラリが標準装備されています。すべてのライブラリ関数は、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、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、および双曲線関数が含まれています。入出力関数と信号処理関数は、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。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。

- **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 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、リンク時に使用するシステムのメモリ・マップを定義できます。これにより、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されています。

- **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、アセンブリ最適化、アセンブル、およびリンクを 1 ステップで実行できます。詳細は、2-2 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。

❑ **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したりできます。詳細は、2-22 ページの 2.5 節「プリプロセッサの制御方法」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 ソースから生成します。一般的な最適化は、すべての C コードに適用できます。また、TMS320C2x/C2xx/C5x 固有の最適化では、TMS320C2x/C2xx/C5x アーキテクチャが利用されます。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。

❑ **データを初期化するコードを ROM に格納する機能**

スタンドアロン型の組み込みアプリケーションの場合には、本コンパイラを使用すると、すべてのコードと初期設定データを ROM にリンクできます。これにより、C コードをリセット時から実行できます。

❑ **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 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を使用すると、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、2-33 ページの 2.7 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。

第 2 章

C コンパイラについて

ソース・プログラムを TMS320C2x/C2xx/C5x 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・ファイルのコンパイル、アセンブル、およびリンクを実行する必要があります。TMS320C2x/C2xx/C5x パッケージには、これらすべてのステップを 1 つのコマンドで実行する特別なシェル・プログラムが含まれています。本章では、この dspcl シェル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、およびリンク方法について詳細に説明します。

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

項目 **ページ**

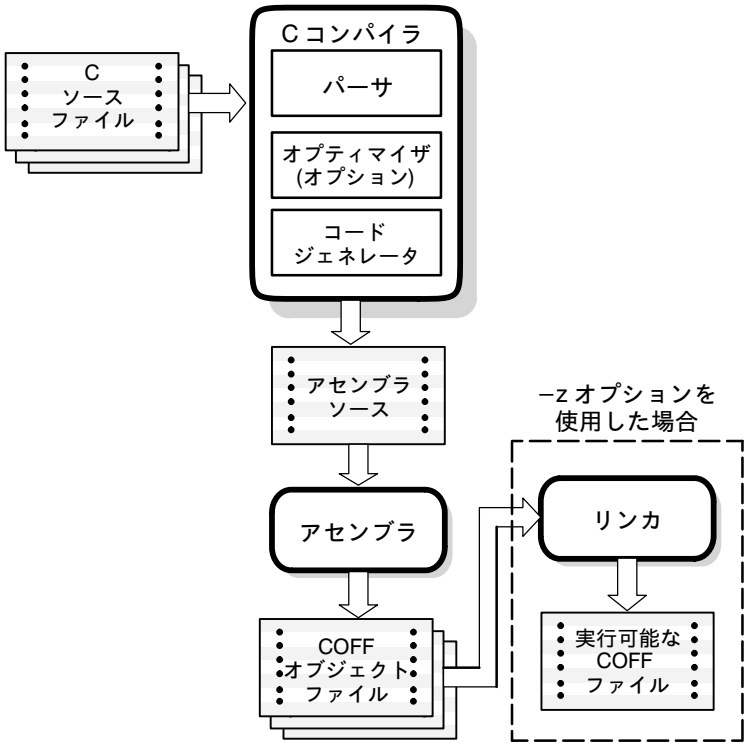
2.1 シェル・プログラムについて

コンパイラ・シェル・プログラム (dspcl) 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、以下の処理を 1 つ以上のソース・モジュールに対して実行します。

- ❑ **コンパイラ**には、パーサ、オプティマイザ、およびコード・ジェネレータが含まれます。C ソース・コードを取り込んで、'C2x、'C2xx、'C5x アセンブリ言語ソース・コードを生成します。
- ❑ **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なオブジェクト・ファイルを作成します。この時点では、リンカの使用はオプションです。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 コードのリンク方法」を参照してください。

デフォルトでは、シェルはファイルのコンパイルとアセンブルだけを実行します。–z オプションを使用すると、dspcl はファイルのコンパイル、アセンブル、およびリンクを行います。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



アセンブラおよびリンカの詳細は、*TMS320C1x/C2x/C2xx/C5x* 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。コンパイラ・ツールを個別に起動する方法については、2-38 ページの 2.9 節「ツールの個別起動方法」を参照してください。

2.2 コンパイラ・シェルの起動方法

コンパイラ・シェルを起動するには、以下のように入力します。

dspcl [*options*] *filenames* [**–z** [*linker options*] [*object files*]]

dspcl	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。
filenames	1 つまたは複数の 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・ファイルのいずれかを指定します。
–z	リンカを実行するオプションです。リンカの起動方法については、第 4 章「C コードのリンク方法」を参照してください。
linker options	リンク・プロセスを制御するオプションです。
object files	コンパイラが作成するオブジェクト・ファイルの名前を指定します。

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・オプションの後ろ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、以下のように入力します。

```
dspcl –q symtab file seek.asm
```

dspcl は各ソース・ファイルを検出すると、C ファイル名に対しては大括弧（[ ]）で、アセンブリ言語ファイル名に対しては不等号括弧（< >）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、dspcl 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、次のメッセージが表示されます。

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

一般の進捗情報は、各コンパイラ・パスの見出し、および定義したときの関数名で構成されます。以下の例は、 **–q** オプションなしで単一のモジュールをコンパイルしたときの出力を示したものです。

```
$ dspcl symtab
[symtab]
TMS320C2x/2xx/5x ANSI C Compiler         Version X.XX
Copyright (c) 1987–1995, Texas Instruments Incorporated
   ″symtab.c″:  ==> main
   ″symtab.c″:  ==> lookup
TMS320C2x/2xx/5x ANSI C Codegen      Version X.XX
Copyright (c) 1987–1995, Texas Instruments Incorporated
   ″symtab.c″:  ==> main
   ″symtab.c″:  ==> lookup
DSP Fixed Point COFF Assembler       Version X.XX
Copyright (c) 1987–1995, Texas Instruments Incorporated
   PASS 1
   PASS 2

No Errors, No Warnings
```

2.3 オプションによるコンパイラ動作の変更方法

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、オプションの規則について説明した後で、各オプションについてまとめた表を示します。データ型のチェックやアセンブル用に指定するオプションなど、使用頻度の高いオプションについては、詳細に説明しています。

コンパイラ・オプションには、以下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 2 文字のペアで構成されます。
- ❑ オプションでは、大文字と小文字の区別はありません。
- ❑ 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、−pe、−pf、および −pk は、−pefk と指定できます。
- ❑ *–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。
- ❑ パラメータ付きの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ることができます。また、続けて指定してもかまいません。
- ❑ ファイルとオプションは、任意の順序で指定できます (−z オプションを除く)。−z オプションは、他のすべてのコンパイラ・オプションの後、かつリンカ・オプションの前に指定する必要があります。

シェルに対してデフォルトのオプションを定義するには、C_OPTION 環境変数を使用します。C_OPTION 環境変数の詳細は、2-20 ページの 2.4.1 節「デフォルトのシェル・オプションの設定方法 (C_OPTION)」を参照してください。

表 2−1 は、すべてのシェル・オプションとリンカ・オプションについてまとめたものです。表内には、各オプションの詳細が記述され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。

すべてのシェル・オプションとリンカ・オプションをオンラインで参照するには、コマンド行でパラメータを指定せずに **dspcl** と入力してください。

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*	コマンド行への追加要素として、ファイルの内容を解釈します。	2-13
–c	リンクを使用不可にします (−z の無効化)。	2-13, 4-5
–d *name*[=*def*]	*name* を事前に定義します。	2-13
–g	シンボリック・デバッグを有効にします。	2-13
–i *directory*	#include 検索パスを定義します。	2-13
–k	アセンブリ言語 (.asm) ファイルを保存します。	2-13
–n	コンパイルのみを行います。	2-13
–q	進捗メッセージの出力を抑止します (静的実行)。	2-13
–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)。	2-13
–r <*register*>	グローバル・レジスタを予約します。	2-13
–s	オプティマイザのコメントをアセンブリ・ソース文に差し込みます。オプティマイザのコメントがない場合は、C ソース文をアセンブリ・ソース文に差し込みます。	2-14; 2-33
–ss	オプティマイザのコメントを C ソースと一緒にアセンブリ文に差し込みます。	2-14
–u*name*	*name* を未定義にします。	2-14
–v *xx*	プロセッサを決定します。*xx* = 25、2xx または 50	2-14
–z	リンクの実行を有効にします。	2-14

(b) 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*	アセンブリ言語ファイルを識別します (デフォルトで .asm または .s*)。	2-15
–fc*filename*	C ソース・ファイルを識別します (デフォルトで .c または拡張子なし)。	2-15
–fo*filename*	オブジェクト・ファイルを識別します (デフォルトで .o*)。	2-15

表 2−1. シェル・オプションのまとめ (続き)

(c) デフォルトのファイル拡張子を変更するオプション

オプション	機能	ページ
–ea*extension*	アセンブリ・ファイルのデフォルトの拡張子を設定します。	2-16
–eo*extension*	オブジェクト・ファイルのデフォルトの拡張子を設定します。	2-16

(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
–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。	2-16
–fs*directory*	アセンブリ・ファイルのディレクトリを指定します。	2-16
–ft*directory*	TMP 環境変数を無効にします。	2-16

(e) ANSI C 型チェックを無視するオプション

オプション	機能	ページ
–tf	プロトタイプ・チェック条件を緩和します。	2-17
–tp	ポインタの組み合わせ条件を緩和します。	2-17

(f) C ランタイム・モデルを変更するオプション

オプション	機能	ページ
–ma	変数がエイリアスをもつことを前提とします。	2-18
–mb	RPTK 命令を抑止します。	2-18
–ml	LDPK 最適化を行いません。	2-18
–mn	−g によって抑止された最適化を有効にします。	2-18
–mp	プロローグまたはエピローグのインラインを生成します。	
–mr	レジスタ使用情報をリスト表示します。	2-18
–ms	コード・スペースの最適化を行います。	2-18
–mx	'C5x シリコン・バグを回避します。	2-18

表 2−1.　シェル・オプションのまとめ (続き)

(g) パーサを制御するオプション

オプション	機能	ページ
–p?	3 文字符号の展開を有効にします。	2-26
–pe	E クラス・エラーを警告として扱います。	2-36
–pf	関数のプロトタイプのリスト・ファイルを生成します。	2-26
–pk	K&R 互換性を許容します。	
–pl	前処理リスト (.pp ファイル) を生成します。	2-25
–pm	プログラム・レベルの最適化を実施するためにソース・ファイルを結合します。	3-6
–pn	.pp ファイルの #line 疑似命令を抑止します。	2-25
–po	前処理のみを実行します。	2-25
–pr	エラー・リストを生成します。	2-36
–pw0	警告メッセージの出力をすべて抑止します。	2-36
–pw1	重大な警告メッセージの出力を有効にします(デフォルト)。	2-36
–pw2	警告メッセージの出力をすべて有効にします。	2-36
–px*filename*	−pm オプションを使用した場合に作成される出力ファイルの名前を指定します。	3-9

(h) 定義制御のインライン関数展開を制御するオプション

オプション	機能	ページ
–x0	組み込み演算子のインライン展開を抑止します。	2-28
–x1	組み込み演算子のインライン展開を有効にします(デフォルト)。	2-28
–x2 または –x	シンボル _INLINE を定義し、−o2 を指定してオプティマイザを起動します。	2-28

表 2−1. シェル・オプションのまとめ (続き)

(i) アセンブラを制御するオプション

オプション	機能	ページ
–aa	絶対リストを生成します。	2-19
–ad*name*	コマンド行から変数を定義します。	2-19
–ahc*filename*	入力ファイルからソース文がアセンブルされる前に、*filename* をコピーします。	2-19
–ahi*filename*	入力ファイルからソース文がアセンブルされる前に、*filename* をインクルードします。	2-19
–al	アセンブリ・リスト・ファイルを生成します。	2-19
–ap	'C2x と 'C2xx または 'C5x 間のポート切り換えを可能にします。	2-19
–app	'C2x から 'C2xx へのポート切り換えを可能にし、.TMS32025 および .TMS3202xx を定義します。	2-19
–as	シンボル・テーブルにラベルを格納します。	2-19
–au*name*	コマンド行から変数の定義を解除します。	2-19
–ax	クロスリファレンス・ファイルを生成します。	2-19

表 2−1.　シェル・オプションのまとめ (続き)

(j)　最適化を制御するオプション

オプション	機能	ページ
–o0	レジスタの使用を最適化します。	3-2
–o1	−o0 による最適化を行い、さらにローカルな最適化を行います。	3-2
–o2 or –o	−o1 による最適化を行い、さらにグローバルな最適化を行います。	3-2
–o3	−o2 による最適化を行い、さらにファイルに対して最適化を行います。	3-3
–oe	モジュール内の関数が割り込みルーチンから呼び出されないこと、およびモジュール内のルーチンが再帰的に呼び出されないものと想定します。	3-11
–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 (デフォルト)。	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	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、修正される関数と変数を、モジュールが含むことを指定します。	3-6
–op1	モジュールはコンパイラに入力されるソース・コード外から修正される変数は含むが、ソース・コード外から呼び出される関数は使用しないことを指定します。	3-6
–op2	モジュールはコンパイラに入力されるソース・コード外から呼び出されたり、修正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	モジュールはコンパイラに入力されるソース・コード外から呼び出される関数を含むが、ソース・コード外から修正される変数は使用しないことを指定します。	3-6
–os	オプティマイザのコメント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)

(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	実行可能な絶対出力を生成します。	4-6
–ar	再配置可能で実行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抑止します。	4-6
–c	実行時に変数を自動初期化します。	4-6
–cr	リセット時に変数を初期化します。	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
–l*filename*	ライブラリまたはコマンド・ファイル名を指定します。	4-6
–m*filename*	マップ・ファイルを指定します。	4-6
–n	MEMORY 疑似命令のすべての埋め込み指定を無視します。	4-6
–o*filename*	出力ファイルを指定します。	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	再配置可能で実行不能な出力を生成します。	4-7
–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。	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*	symbol を未定義にします。	4-7
–v0	バージョン 0 の COFF フォーマットを生成します。	4-7
–v1	バージョン 1 の COFF フォーマットを生成します。	4-7
–v2	バージョン 2 の COFF フォーマットを生成します。	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

–@ *filename* コマンド行にファイル内容を付加します。このオプションは、ホスト・オペレーティング・システムによるコマンド行の長さ制限に対する回避策として使用できます。コメントを挿入する場合は、コマンド・ファイルの行頭に # か ; を入力します。

–c リンカを抑止し、リンクの実行を指定する –z オプションを無効にします。このオプションは、C_OPTION 環境変数で –z を指定しているがリンク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 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を得ることができます。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 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。

–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 回まで使用できます。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、2-24 ページの 2.5.3 節「–i オプションを使用した #include ファイル検索パスの変更方法」を参照してください。

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

–n コンパイルのみを実行します。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、および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、通常のエントリ時保存レジスタとして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では、オプティマイザのコメントまたはC のソースをアセンブリ・ソースに差し込むことができます。オプティマイザを起動している場合は (–o*n* オプションを指定)、オプティマイザのコメントがコンパイラのアセンブリ言語出力に差し込まれます。オプティマイザを起動していない場合は、C ソース文がコンパイラのアセンブリ言語出力に指し込まれ、これにより各 C の文に対して生成されたコードを検査することができます。 –s オプションを指定すると、–k オプションも暗黙指定されます。

–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 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している (–o*n* オプション) 場合、このオプションによりコードが大幅に再編成される場合があります。詳細は、2-33 ページの 2.7 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*xxx* ターゲット・プロセッサを指定します。*xxx* には、’C2x プロセッサの場合は 25、’C2xx の場合は 2xx、’C5x の場合は 50 を選択できます。

–z 指定されたオブジェクト・ファイルに対してリンカを実行します。–z オプションとそのパラメータは、コマンド行で他のオプションとパラメータの一番後ろに入力します。–z の後ろにある引数は、すべてリンカに渡されます。詳細は、4-2 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 ファイル名の指定方法

コマンド行では、入力ファイルとして、C ソース・ファイル、アセンブリ・ソース・ファイル、またはオブジェクト・ファイルを指定できます。シェルは、ファイル名の拡張子によってファイルのタイプを判別します。

拡張子	ファイル・タイプ
.c または拡張子なし (.c と見なされる)	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

拡張子なしのファイルは、C ソース・ファイルと見なされます。ファイル名拡張子の規則により、C ファイルのコンパイル、およびアセンブリ・ファイルの最適化とアセンブルを 1 つのコマンドで行うことができます。

シェルが個々のファイル名を解釈する方法を変更する方法については、2-15 ページの 2.3.3 節を参照してください。シェルがアセンブリ・ソース・ファイルとオブジェクト・ファイルの拡張子を解釈する方法、およびファイルを作成するときの拡張子の付け方を変更する方法については、2-16 ページの 2.3.5 節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できます。ワイルドカードの使用方法は、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。

```
dspcl *.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、および –fo オプション)

シェルがファイル名を解釈する方法は、オプションによって変更できます。シェルが認識できない拡張子を使用している場合は、–fa、–fc、または –fo オプションを使用することによって、ファイルのタイプを指定できます。オプションとファイル名の間には、任意でスペースを入れることができます。指定するファイルのタイプに応じて、以下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

たとえば file.s という C ソース・ファイルと assy というアセンブリ言語ソース・ファイルがある場合は、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。

```
dspcl –fc file.s –fa assy
```

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

2.3.4 シェル・プログラムによる拡張子の解釈方法と拡張子の付け方の変更方法 (−ea および −eo オプション)

シェルによるファイル名の拡張子の解釈方法、およびファイル作成時の拡張子の付け方は、オプションによって変更できます。コマンド行で、−ea および −eo オプションは処理対象のファイル名の前に入力し なければなりません。どちらの オプションについても、ワイルドカードによる指定が可能です。拡張子の長さは 9 文字まで有効です。指定する拡張子のタイプに応じて、次のいずれか適切なオプションを使用してください。

−ea[.] *new extension*　　アセンブリ言語ファイルの場合

−eo[.] *new extension*　　オブジェクト・ファイルの場合

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。

```
dspcl -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、次のように入力することもできます。

```
dspcl -earrr -eoo fit.rrr
```

2.3.5 ディレクトリの指定方法

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、および一時ファイルをカレント・ディレクトリに格納します。これらのファイルを別のディレクトリに格納する場合は、以下のオプションを指定します。

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

```
dspcl -fr d:\object
```

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

```
dspcl -fs d:\assembly
```

–ft*directory*　　一時中間ファイルのディレクトリを指定します。−ft オプションは、TMP 環境変数を無効にします (詳細は、2-21 ページの 2.4.2 節「一時ファイル・ディレクトリの指定方法 (TMP)」を参照してください)。一時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。

```
dspcl -ft c:\temp
```

2.3.6 ANSI C 型チェックを無視するオプション

以下のオプションを使用すると、コードに対する厳格な ANSI C 型チェックの一部を無視できます。

–tf プロトタイプ関数の再宣言に対する型チェックを無視します。ANSI C では、(次の例に示すように) 古い形式で宣言された関数を後からプロトタイプで宣言すると、エラーになります。これは、プロトタイプのパラメータの型がデフォルトの引数プロモーション (float を double に、char を int に変換する) と異なるからです。

```
int func( )                    /* old format */
int func(float a, char b)      /* new format */
```

–tp ポインタの組み合わせに対する型チェックを無視します。このオプションには、次の 2 つの機能があります。

- 演算時に、符号付きの型へのポインタを、対応する符号なし型へのポインタと組み合わせることができます。

```
int *pi;
unsigned *pu;
pi = pu; /* Illegal unless -tp used */
```

- 修飾方法が異なる型のポインタ同士を組み合わせることができます。

```
char *p;
const char *pc;
p = pc;   /* Illegal unless -tp used */
```

–tp オプションは、ポインタをプロトタイプ関数に渡すときに特に便利です。これは渡されるポインタの型が、通常、プロトタイプで宣言されているパラメータの型と異なるからです。

2.3.7 ランタイムモデル・オプション

– ma 変数にエイリアスが付いているものと見なします。コンパイラは、ポインタによる代入が行われるとき、ポインタが名前付き変数にエイリアスを付けている(ポイントしている)と見なし、レジスタ最適化を中止します。

– mb 構造体を移動することに対して、割り込み不能な RPTK 命令の使用を禁止します。

– ml LDPK 命令の使用を最小限に抑えるためにコード・ジェネレータが実行する最適化を抑止します。この最適化により、プログラムの.bss セクション内に小さいホールが生じる可能性があります。– ml オプションを使用すると、これらのホールは完全になくなりますが、コード内の LDPK 命令の追加が必要になります。システムが使用するメモリの形式が、データ・メモリ・スペース用よりもプログラム・メモリ・スペース用の方がコストが低い場合には、これをお勧めします。

– mn **– g** で抑止した最適化を再度有効にします。– g オプションを使用してシンボリック・デバッグ情報を生成すると、コード・ジェネレータの最適化の多くが無効化されますが、これは最適化によってデバッガの動作が影響を受けるからです。

– mr レジスタ使用情報をリスト表示します。コード・ジェネレータが各C 文をコンパイルした後、–mr は、アセンブリ言語ファイル内にコメントとしてコンテント・テーブルを登録します。–mr オプションは、レジスタ追跡最適化のために追跡するのがむずかしいコードの検査に便利です。

– ms 速度ではなく、コード・スペースを最適化します。

– mx 'C5x シリコン・バグを回避します。このスイッチを使用しなければならないのは、割り込みの実現または最適化を使用したコンパイルを行うプログラムを、2.0 より前の 'C5x デバイス・バージョンで使用する準備をする場合です。

OVLY と RAM ステータス・ビットがオンのときにコンパイラが実行されると、コンパイルされた特定のコード・シーケンスが正しく実行しないのは、コードとデータの両方が 'C51 のオンチップ RAM の 1K、または 'C50 のオンチップ RAM の 9K の内の同じ 2K ブロックにある場合です。リンカ・コマンド・ファイルを使用して、この競合が発生しないようにプログラムとデータ・スペースを設定してください。

2.3.8 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。
−ad*name*[=*def*]	アセンブラ用の *name* 定数を事前に定義します。これは、各ソース・ファイルの先頭に #define *name def* を挿入するのと同じです。オプションの [=*def*] を省略すると、*name* は 1 に設定されます。
−ahc*filename*	−hc アセンブラ・オプションを指定してアセンブラを起動し、アセンブリ・モジュールに指定されたファイルをコピーするようにアセンブラに指示します。ファイルは、ソース・ファイル文の前に挿入されます。コピーされたファイルは、アセンブリ・リスト・ファイルに記録されます。
−ahi*filename*	−hi アセンブラ・オプションを指定してアセンブラを起動し、アセンブリ・モジュールに指定されたファイルをインクルードするようにアセンブラに指示します。ファイルは、ソース・ファイル文の前にインクルードされます。インクルード・ファイルは、アセンブリ・リスト・ファイルには記録されません。
−al	−l (小文字の L) アセンブラ・オプションを指定してアセンブラを起動し、アセンブリ・リスト・ファイルを生成します。
−ap	'C2x から 'C2xx または 'C5x へのポートの切り換えを有効にします。−ap は、対応する −v2xx または −v50 オプションと一緒に使用してください。
−app	'C2x から 'C2xx へのポート切り換えを有効にし、.TMS32025 および .TMS3202xx アセンブラ・シンボルを定義します。−app は、−v2xx オプションと一緒に使用してください。
−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*	−u アセンブラ・オプションを指定してアセンブラを起動し、事前定義された定数 *name* の定義を解除します。−au オプションは、指定された定数の −ad オプションを無効にします。
−ax	−x アセンブラ・オプションを指定してアセンブラを起動し、リスト・ファイルにシンボリック・クロスリファレンスを生成します。

アセンブラ・オプションの詳細は、*TMS320C1x/C2x/C2xx/C5x*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 環境変数によるコンパイラの動作の変更方法

通常使用するソフトウェア・ツール・パラメータを設定する環境変数を定義できます。環境変数は、システム初期化ファイル内において、ユーザが定義し、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、特定タイプの情報を検出または取得します。

環境変数を使用すると、デフォルト値が設定され、コンパイラの個別の起動が簡単になります。これは、これらのパラメータが自動的に指定されるからです。ツールを起動する場合に、コマンド行オプションを使用すると、環境変数で設定されるデフォルトの多くを無効にできます。

2.4.1 デフォルト・シェル・オプションの設定方法 (C_OPTION)

C_OPTION 環境変数を使用して、コンパイラ、アセンブラ、およびリンカのデフォルトのシェル・オプションを設定すると便利です。これを行う場合、シェルは、ユーザがシェルを実行するたびに C_OPTION に設定したデフォルトのオプションまたは入力ファイル名を使用します。

C_OPTION 環境変数を使用したデフォルト・オプションの設定は、同じオプションや入力ファイル (またはその両方) を使用して連続してシェルを実行する場合に便利です。コマンド行と入力ファイル名を読み込んだ後、シェルは C_OPTION 環境変数を探してから、それを読み込んで処理します。

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

オペレーティング・システム	入力
DOS または OS/2	**set C_OPTION=***option*$_1$[;*option*$_2$ . . .]
UNIX (C シェル)	**setenv C_OPTION "***option*$_1$ [*option*$_2$ . . .]"
UNIX (Bourne または Korn シェル)	**C_OPTION="***option*$_1$ [*option*$_2$ . . .]" **export C_OPTION**

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

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

次の例では、コンパイラ・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 内の −z の後に指定したオプションは、すべてリンカに渡されます。これにより、C_OPTION 環境変数を使用してデフォルトのコンパイラとリンカのオプションを指定した後、追加のコンパイラとリンカのオプションをシェル・コマンド行で指定できます。環境変数に −z を設定したときに、コンパイルだけを行いたい場合 は、シェルの −c オプションを使用します。以下のコマンド例では、上記で説明したように C_OPTION が設定されていることを前提としています。

```
dspcl *c                ; compiles and links
dspcl –c *.c            ; only compiles
dspcl *.c –z lnk.cmd    ; compiles and links using a
                        ; command file
dspcl –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方法」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。

2.4.2 一時ファイル・ディレクトリの指定方法 (TMP)

コンパイラ・シェル・プログラムは、プログラムの処理中に中間ファイルを作成します。デフォルトでは、シェルは中間ファイルをカレント・ディレクトリに入れます。しかし TMP 環境変数を使用すると、一時ファイル用の特定のディレクトリを指定できます。

TMP 環境変数を使用すると、RAM ディスクまたはその他のファイル・システムを使用できるようになります。また、ソース・ファイルが置かれているディレクトリにファイルを書き込むことなく、リモート・ディレクトリからソース・ファイルをコンパイルすることもできます。これは、ディレクトリが保護されている場合に便利です。

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

オペレーティング・システム	入力
DOS または OS/2	**set TMP=***pathname*
UNIX (C シェル)	**setenv TMP "***pathname***"**
UNIX (Bourne または Korn シェル)	**TMP="***pathname***"** **export TMP**

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

たとえば、Windows 用ハード・ディスクで中間ファイル用に、 temp という名前のディレクトリを設定する場合には、次のように入力します。

```
set TMP=c:\temp↵
```

2.5 プリプロセッサの制御方法

コンパイル時、コードはパーサの一部であるプリプロセッサを介して実行されます。シェル・プログラムを使用すると、マクロとその他の各種疑似命令を使用してプリプロセッサを制御できます。

この節では、TMS320C2x/C2xx/C5x プリプロセッサを制御する特別な機能を説明します。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。TMS320C2x/C2xx/C5x C コンパイラには、標準 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、以下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、読めば分かるエラー・メッセージを生成します。エラーが発生した行番号とファイル名は、診断メッセージとともに表示されます。

2.5.1 事前定義マクロ名

コンパイラは、表 2−2 に示す事前定義マクロ名を保管し、認識を行います。

表 2−2. 事前定義マクロ名

マクロ名	説明
__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。
_dsp	1 に展開します (TMS320C2x/C2xx/C5x コンパイラを識別します)。
_TMS320C25	−v25 オプションでは 1 に展開します。
_TMS320C2XX	−v2xx オプションでは 1 に展開します。
_TMS320C50	−v50 オプションでは 1 に展開します。
_INLINE	−x または −x2 オプションでは 1 に展開します。それ以外のオプションでは、未定義です。

† 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。たとえば、次のとおりです。

```
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” , ”Jan 14 1988”, ”13:58:17”);
```

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込むようにコンパイラに指示します。ファイルを指定する際は、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 コンパイラで指定されたディレクトリ
  3) C_DIR 環境変数で設定されたディレクトリ
- ファイル名を不等号括弧 (< >) で囲んだ場合、コンパイラは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) −i オプションで指定されたディレクトリ
  2) C_DIR 環境変数で設定されたディレクトリ

−i オプションの使用方法については、2-24 ページの 2.5.3 節「−i オプションによる include ファイル検索パスの変更方法」を参照してください。C_DIR 環境変数の使用方法については、*TMS320C2xx Code Generation Tools Installation Instructions* を参照してください。

2.5.3 –i オプションによる #include ファイル検索パスの変更方法

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

–i *directory1* [**–i** *directory2* ...]

コンパイラの 1 回の起動で、最大 32 回まで –i オプションを指定できます (–i オプション 1 つにつき *directory* を 1 つ指定)。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できます。たとえば、カレント・ディレクトリに source.c という名前のファイルがあるとします。この source.c ファイルには、次のような疑似命令文が入っています。

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

alt.h の完全なパス名を、次のように仮定します。

UNIX /6xtools/files/alt.h

DOS または OS/2 c:\6xtools\files\alt.h

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

オペレーティング・システム	入力
DOS または OS/2	`dspcl –ic:\dsp\files source.c`
UNIX	`dspcl –i/dsp/files source.c`

2.5.4 前処理リスト・ファイルの生成方法 (– pl オプション)

– pl オプションを指定すると、ソース・ファイルの前処理したバージョンを生成できます。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・ファイルに対して以下の処理が行われます。

- ❏ バックスラッシュ (\) で終わっている各ソース行と、次の行との連結
- ❏ 3 文字符号系列の展開 (–p? オプションが指定されている場合)
- ❏ コメントの除去
- ❏ ファイルへの #include ファイルのコピー
- ❏ マクロ定義の処理
- ❏ 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実行

上記の処理は、K&R の A12 節に指定されている変換フェーズ 1 ～ 3 に対応します。

前処理後の出力ファイルには、#line 以外のプリプロセッサ疑似命令は入っていません。コンパイラは、出力ファイルに同期化 された行およびファイル情報を示す #line 疑似命令を挿入し、オリジナルのソース・ファイルの入力位置が判別できるようにします。– pn オプションを使用すると、#line 疑似命令は挿入されません。詳細は、2-25 ページの 2.5.4.2 節「前処理リスト・ファイルからの #line 疑似命令の除去方法 (–pn オプション)」を参照してください。

2.5.4.1 前処理リスト・ファイルの生成方法 – コード生成なし (–po オプション)

–po オプションは、前処理だけを実行し、前処理リスト・ファイルを書き出します。–po オプションは、 –pl オプションの代わりに使用 されます。構文チェックやコード生成は行われません。–po オプションは、マクロ定義をデバッグするときに便利です。生成されるリスト・ファイルは、コンパイラを通じて再実行可能な、有効な C ソース・ファイルです。

2.5.4.2 前処理リスト・ファイルからの #line 疑似命令の除去方法 (–pn オプション)

–pn オプションは、前処理リスト・ファイル内の行およびファイル情報を抑止します。–pn オプションは、 –po または –pl オプションで生成される .pp ファイルで #line 疑似命令を抑止します。

#line 疑似命令の例は、次のとおりです。

```
#line 123 file.c
```

–pn オプションは、マシンで生成されたソースをコンパイルする際に便利です。

2.5.5 #error および #warn 疑似命令によるカスタム・エラー・メッセージの作成方法

標準の #error プリプロセッサ疑似命令は、コンパイラに強制的に診断メッセージを発行させ、コンパイルを停止させます。コンパイラでは、#error 疑似命令の拡張として #warn 疑似命令も使用できます。#warn 疑似命令は診断メッセージを発行させますが、コンパイルを停止させません。#warn の構文は #error の構文と同じです。

```
#error token-sequence
```

```
#warn token-sequence
```

2.5.6 3 文字符号系列の展開の有効化 (−p? オプション)

3 文字符号とは、特定の意味をもつ 3 つの文字のことです (それぞれの意味は ISO646〜1983 不変コード・セットにより定義されています)。文字セットが限定されているシステムでは、これらの文字は表現できません。たとえば、3 文字符号の ??' は ^ に相当します。ANSI C 規格では、3 文字系列の順番を定義しています。

デフォルトでは、コンパイラは 3 文字符号を認識しません。3 文字符号系列の展開を有効にするには、−pg オプションを使用します。それによりコンパイル時 に、3 文字符号系列は対応する単一文字に展開されます。3 文字符号系列の詳細は、ANSI 仕様の § 2.2.1.1 を参照してください。

2.5.7 関数プロトタイプ・リスト・ファイルの作成方法 (−pf オプション)

−pf オプションを使用すると、プリプロセッサは、各々の C ファイルに対して、ファイル内のすべての関数のプロトタイプを含んだファイルを作成します。各関数プロトタイプ・ファイルには、対応する C ファイルの名前に拡張子 .pro を付けた名前が付けられます。

2.6 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、関数のインライン展 開と呼ばれます。インライン関 数展開は、以下の理由から短い関数には便利です。

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

インライン関数展開は、以下のいずれかの方法で行われます。

- ❑ 組み込み演算子はデフォルトで展開されます。
- ❑ オプティマイザは、−o3 オプションを指定して呼び出された小さい関数に対して、自動的にインライン展開を実行し ます。詳細は、3-12 ページの 3.5 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。
- ❑ 最適化 (−x オプション) を指定してコンパイラを起動した場合、コンパイラはコード内に inline キーワードを検出すると、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

関数をインライン展開すると、コード・サイズが増えます。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・サイズが大幅に増えます。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サイズが小さい関数に適しています。コード・サイズが大きすぎるように見える場合は、−oi0 オプションでコンパイルして、コード・サイズの違いを確認してください。

2.6.1 組み込み演算子のインライン展開

コンパイラは、デフォルトで ターゲット・システムの組み込み演算子 (abs など) を自動的にインライン展開します。この展 開は、オプティマイザの使用や、コマンド行でのコンパイラおよびオプティマイザ・オプションの指定に関わらず実行されます (この自動インライン展開を無効にするには、−x0 オプションを指定してコンパイラを起動します)。組み込み演算子関数には、以下のものがあります。

- ❑ abs
- ❑ labs
- ❑ fabs

2.6.2 インライン関数展開の制御方法 (−x オプション)

−x オプションは、_INLINE プリプロセッサ・シンボルの定義と一部のタイプのインライン関数展開を制御します。展開には、次の 3 つのレベルがあります。

−x0	定義制御のインライン展開を行いません。このオプションを指定すると、組み込み演算子関数に対するデフォルトの展開は無効になりますが、インライン関数展開 (3-12 ページの 3.5 節「自動インライン展開 (−o*size* オプション)」を参照) は無効になりません。
−x1	デフォルトの動作にリセットします。組み込み演算子 (abs、labs、および fabs) は、呼び出しのたびにインライン展開されます。環境変数やコマンド・ファイルで別の −x オプションを使用している場合は、コマンド行からこのオプションを指定してデフォルトの動作にリセットできます。
−x2 または −x	_INLINE プリプロセッサ・シンボルを 1 に定義します。独立したコマンド行オプションでオプティマイザが起動されない場合でも、デフォルト・レベル (−o2) でオプティマイザを起動します。

2.6.3 inline キーワードの使用方法

最適化を指定してコンパイラを起動した場合、コンパイラはコード内に inline キーワードを検出すると、定義制御されたインライン展開を実行します。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。静的インラインとして宣言された関数の場合、ローカルな静的変数が存在しても展開が行われます。また、再帰関数やノンリーフ関数に対しては、インライン展開の深さが制限されます。インライン展開は、小さい関数や数回しか呼び出されない関数に対して使用します (ただし、コンパイラがこれを強制することはありません)。inline キーワードを使用すると、このタイプの関数インライン展開を制御できます。

inline キーワードは、関数が標準の呼び出し手順を使用して呼び出される代りに、呼び出し位置でインライン展開されることを示します。コンパイラは、inline キーワードを指定して宣言された関数のインライン展開を実行し、小さい関数を自動的にインライン展開できます。

次の場合は、関数のインライン展開を有効にできます。

- inline キーワードを使用して関数を宣言するか、または
- −o3 スイッチを使用してオプティマイザを起動します。さらに、以下のことが必要です。
  - 関数が非常に小さい（−oi スイッチによって制御される）、かつ
  - 関数が、呼び出される前に宣言される。

関数のインライン展開は、次の場合に無効になります。

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

2.6.3.1 モジュール内で関数をインラインとして宣言する方法

モジュール内で (inline キーワードを使用して) 関数をインライン関数として定義すると、そのモジュール内では、その関数はインライン展開されることになります。その関数のグローバル・シンボルが作成され (コードが生成され) ますが、その関数は、インライン関数として定義したモジュール内でしかインライン展開されません。他のモジュールにその関数の静的インライン宣言が含まれていない場合、そのモジュールは、グローバル・シンボルを呼び出すことができます。

インラインとして宣言された関数は、オプティマイザが起動されるときに展開されます。−x2 オプションを使用すると、オプティマイザが自動的にデフォルト・レベル (−o2) で起動されます。

モジュール内で関数をインラインとして定義するには、次の構文を使用してください。

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

2.6.3.2 関数を静的インラインとして宣言する方法

ヘッダ・ファイル内である関数を静的インラインとして宣言されている場合は、その関数は該当のヘッダを含むすべてのモジュールでインライン展開するように指定されます。この宣言では、対象の関数の名前を指定し、その関数をインライン展開することを指定しますが、関数宣言そのものに対するコードは生成されません。このように宣言された関数は、ヘッダ・ファイルに入れ、プログラムのすべてのソース・モジュールに組み込むことができます。

関数を静的インラインとして宣言するには、次の構文を使用してください。

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

2.6.4 _INLINE プリプロセッサ・シンボル

_INLINE プリプロセッサ・シンボルは、−x2 (または −x) オプションを指定してパーサ (またはコンパイラのシェル・ユーティリティ) を起動するときに定義されます (値は 1 に設定されます)。このシンボルを使用すると、オプティマイザが使用されるかどうかに関係なく動作するようにコードを書くことができます。このシンボルは、コンパイラの標準ヘッダ・ファイルで使用され、標準 C ランタイム関数の宣言を制御します。

2-32 ページの例 2−1 は、ランタイムサポート・ライブラリで _INLINE プリプロセッサ・シンボルをどのように使用しているかを示しています。

_INLINE プリプロセッサ・シンボルは、インライン展開が使用されるかどうかに関係なく、関数を正しく宣言するために string.h ヘッダ・ファイルで使用されます。_INLINE プリプロセッサ・シンボルは、INLINE プリプロセッサ・シンボルが定義されている場合に限って strlen を静的インラインとして宣言するように、条件付きで _ _INLINE を定義します。

モジュールの他の部分がインライン展開を有効にしてコンパイルされ、さらに string.h ヘッダが含まれている場合、 strlen に対する参照はすべてインライン展開されるので、リンカはランタイムサポート・ライブラリの strlen を使用して参照を解決する必要はありません。それ以外の場合には、ランタイムサポート・ライブラリ・コードは strlen に対する参照を解決し、関数呼び出しが生成されます。

関数ライブラリの場合と同様に、 ヘッダ・ファイルで _INLINE プリプロセッサ・シンボルを使用すれば、プログラム内のモジュールの一部または全部に対するインライン展開の指定に関わりなく動作するプログラムを書くことができます。

インラインとして宣言された関数は、オプティマイザをどのレベルで起動しても必ず展開されます。ランタイム・ライブラリ関数のように、インラインとして宣言され、_INLINE プリプロセッサ・シンボルによって制御される関数は、オプティマイザを起動し、かつ _INLINE プリプロセッサ・シンボルが 1 のときには必ず展開されることになります。ライブラリでインライン関数を宣言する場合は、_INLINE プリプロセッサ・シンボルを使用して宣言を制御することをお勧めします。_INLINE を使用して展開を制御できない場合は、その後オプティマイザを使用しないでコンパイルを実行すると、関数呼び出しは解決されません。

例 2−1 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、静的インラインとしてプロトタイプが宣言されるのは、_INLINE が真の場合、つまりこのヘッダを含むモジュールが −x オプションを指定してコンパイルされる場合だけです。

2 番目の定義はライブラリ用で、インライン展開が無効の場合に、呼び出し可能な strlen が存在するように定義されたものです。この strlen はインライン関数ではないので、string.h を組み込む前に _INLINE プリプロセッサ・シンボルは未定義に設定されます (#undef)。その結果、インライン関数ではない strlen のプロトタイプが生成されます。

例 2−1. 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 HEADER FILE                                                      */
/*****************************************************************************/
typdef unsigned size_t

#if _INLINE
#define __INLINE static inline    /* Declaration when inlining        */
#else
#define __INLINE                 /*No declaration when not inlining   */
#endif

__INLINE  void *memcpy (void *_s1, const void *_s2, size_t _n);

#if _INLINE                      /* Declare the inlined function     */

static inline void *memcpy (void *to, const void *from, size_t n)
{
   register char *rto  = (char *) to;
   register char *rfrom= (char *) from;
   register size_t rn;

   for (rn = 0; rn < n; rn++) *rto++ =rfrom++;
   return (to);
}

#endif   /* _INLINE   */

#undef __INLINE
```

(b) strlen.c

```
/*****************************************************************************/
/* MEMCPY.C   (rts2xx.lib)                                                   */
/*****************************************************************************/
#undef _INLINE      /* Turn off so code will be generated                    */

#include <string.h>

void *memcpy (void *to, const void *from, size_t n)
{
   register char *rto  = (char *) to;
   register char *rfrom= (char *) from;
   register size_t rn;

   for (rn = 0; rn < n; rn++) *rto++ =rfrom++;
   return (to);
}
```

2.7 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 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 ソース文に対して生成されたアセンブリ・コードを検査できます。インターリスト・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、–s 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。

```
dspcl –s function
```

–s オプションは、差し込み後のアセンブリ言語ファイルを削除しないようにシェルに指示します。出力されるアセンブリ・ファイル 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きは、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 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 ソース文をコメントとして書き込みます。

例 2–2 は、差し込み後のアセンブリ・ファイルの一般的な例を示したものです。

例 2−2.　差し込み後のアセンブリ言語ファイル

```
;>>>>     main()
;>>>>        int i, j;
*****************************************
* FUNCTION DEF : _main
*****************************************
_main:
      SAR       AR0,*+
      SAR       AR1,*
      LARK      AR0,3
      LAR       AR0,*0+,AR2
;>>>>        i += j;
      LARK      AR2,1
      MAR       *0+
      LAC       *−
      ADD       *
      SACL      *+
;>>>>        j = i + 123;
      ADDK      123
      SACL      *,AR1
EPI0_1:
      SBRK      4
      LAR       AR0,*
      RET
      .end
```

dspcl 以外の、インターリスト・ユーティリティの起動方法については、2-45 ページの 2.9.5 節「インターリスト・ユーティリティの起動方法」を参照してください。

注:　オプティマイザでの −s オプションの使用方法

最適化を実行すると、ソースの通常の差し込みは、実際上不可能になります。これは、オプティマイザが広範囲にわたってプログラムの再配置を行うからです。−s 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 の文を書き込みます。コメントには、割り当てられたレジスタ変数のリストも含まれています。複数の変数または等価の変数を同じレジスタに複写伝播または代入することもあるので、オプティマイザの差し込みコメントがまぎらわしくなる場合があります。

2.8 コンパイラ・エラーの理解と処理方法

本コンパイラの主要な機能の 1 つに、ソース・プログラムのエラーの検出と報告があります。本コンパイラはユーザ・プログラム内でエラーを検出すると、以下の形式のメッセージを表示します。

"*file.c***", line** *n*: **[***ECODE***]** *error message*

"*file.c***"**	ファイル名を示します。
line *n*	エラーが発生した行番号を示します。
ECODE	4 文字のエラー・コードです。単一の英大文字はエラー・クラスを示します。その後に続く 3 桁の数字は、各エラーを示す固有の番号です。
error message	メッセージの本文です。

C コードのエラーは、重大度に応じて、*W*、*E*、*F*、および *I* (大文字の *i*) で示される 4 つのクラスに分かれています。コンパイラは、C コードに関係していなくても、コンパイルを妨げる他のエラーも報告します。エラー・メッセージの各レベルの例については、表 2–3 を参照してください。

- **W クラス・エラー**は警告です。言語規則の上では技術的に定義できない状態、あるいは予期しない結果を招く恐れのある状態が原因で発生します。このエラーが発生しても、コンパイラは処理を続けます。
- **E クラス・エラー**は、回復可能なエラーです。言語の構文に関する規則に反する状態が原因で発生します。通常、このエラーは致命的なエラーですが、 –pe オプションを使用すると、コンパイラがエラーから回復して出力ファイルを生成します。詳細は、2-36 ページの 2.8.2 節「警告としての E クラス・エラーの対応 (–pe オプション)」を参照してください。
- **F クラス・エラー**は致命的エラーです。言語の統語または構文に関する規則に反する状態が原因で発生します。コンパイラはエラーを回復できないので、F クラス・エラーの場合は出力は生成されません。
- **I クラス・エラー**は実装エラーです。コンパイラ内部の制限値のいずれかを超えると発生します。通常、このエラーは明示的なエラーではなく、ソース・コードでの過剰動作が原因で発生します。ほとんどの場合、I クラス・エラーが発生すると、コンパイラはただちに異常終了します。I クラスのメッセージの大部分には、原因となった制限値の最大値が示されています (絶対的な制限値は、5-16 ページの 5.9 節「コンパイラの制限値」にも記載されています)。
- **その他のエラー・メッセージ** (コマンド行構文が正しくない、指定されたファイルが見つからないなど) は、通常、致命的エラーを示します。メッセージの前には、 >> 記号が付きます。

表 2−3.　エラー・メッセージの例

エラー・レベル	エラー・メッセージの例
W クラス	”file.c”, line 42:[W029] extra text after preprocessor directive ignored
E クラス	”file.c”, line 66: [E055] illegal storage class for function ’f’
F クラス	”file.c”, line 71: [F0108] structure member ’a’ undefined
I クラス	”file.c”, line 99: [I011] block nesting too deep (max=20)
その他	>> Cannot open source file ’mystery.c’

2.8.1　エラー・リストの生成方法 (−pr オプション)

エラー・リストを生成するには、−pr オプションを使用します。エラー・リストには、*source*.**err** (*source* は C ソース・ファイルの名前) という名前が付きます。

2.8.2　警告としての E クラス・エラーの対応(− pe オプション)

致命的エラーとは、コンパイラによる出力ファイルの生成が妨げられるエラーです。通常、E、F、および I クラス・エラーは致命的ですが、W クラス・エラーは致命的ではありません。−pe オプションを指定すると、コンパイラは E クラス・エラーを警告として処理するので、エラーが発生してもファイルに対するコードを生成します。

−pe を使用すると、言語規則を曲げることができるので、注意が必要です。どの場合も、コンパイラから予測どおりの出力が得られるとは限りません。

F または I クラス・エラーの回復を指定する 方法はありません。これらのエラーは常に致命的です。

−pe オプションの例については、2.8.4 節「エラー・オプションの使用例」を参照してください。

2.8.3　警告メッセージのレベル変更方法 (− pw オプション)

−pw オプションを指定して警告メッセージを設定すると、表示される警告メッセージのレベルを判別できます。−pw に続く数値は、レベル (0、1、または 2) を表します。表 2−4 を使用して、適切なレベルを選択してください。−pw オプションの例については、2.8.4 節「エラー・オプションの使用例」を参照してください。

表 2−4.　−pw オプションのレベルの選択方法

設定内容	使用オプション
すべての警告メッセージを表示しない。このレベルは、警告を発生させる原因が判明していて、それでも支障がないと判断できる場合に便利です。	−pw0
重大な警告メッセージを表示する。これはデフォルトです。	−pw1
すべての警告メッセージを表示する。	−pw2

2.8.4　エラー・オプションの使用例

以下の例は、−pe オプションによるエラー出力の抑止、および −pw オプションによるエラー・メッセージのレベル変更の 方法を示しています。この例では、次のコードを使用します。

```
   int *pi; char *pc;

#if STDC
   pi = (int *) pc;
#else
   pi = pc;
#endif
```

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

  ```
  [err]
  ”err.c”, line3: [E104] operands of ’=’ point to different types
  ```

  この場合、E クラス・エラーは致命的なので、コンパイラはコードを生成しません。

- −pe オプションを指定してコンパイラを起動すると、以下のような結果になります。

  ```
  [err]
  ”err.c”, line3: [E104] operands of ’=’ point to different types
  ```

  この場合、同じメッセージが生成されますが、−pe を使用しているのでコンパイラはエラーを無視し、出力ファイルを生成します。

- −pew2 オプション（−pe と −pw2 を結合したもの）を指定してコンパイラを起動すると、以下のような結果になります。

  ```
  [err.c]
  ”err.c”, line5: [W038] undefined preprocessor symbol ’STDC’
  ”err.c”, line8: [E122] operands of ’=’ point to different types
  ```

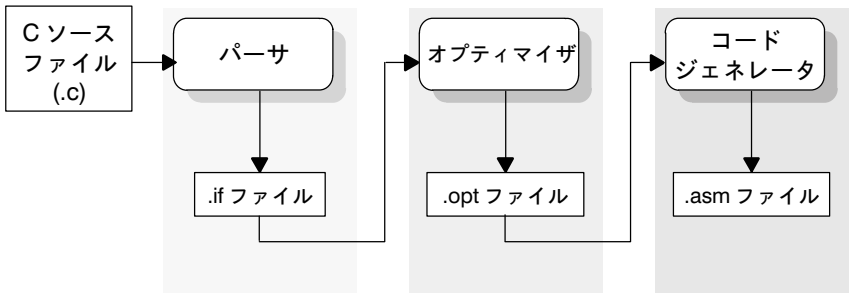
  上記の例のように、−pe を指定すると、コンパイラはエラーを無視し、コードを生成します。−pw2 オプションが指定されているので、すべての警告メッセージが生成されます。

2.9 ツールの個別起動方法

TMS320C2x/C2xx/C5x C コンパイラでは、dspcl を使用することによりすべてのツールを同時に起動したり、各ツールを個別に起動したりできます。各種のアプリケーションに対応するように、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）、アセンブラ、およびリンカを個々のプログラムとして起動できます。この節では、dspcl 以外のインターリスト・ユーティリティの起動方法についても説明します。

- **コンパイラ**は、3 つの独立したプログラム、つまりパーサ、オプティマイザ、およびコード・ジェネレータで構成されています。

図 2−2. コンパイラの概要



パーサの入力は、C ソース・ファイルです。パーサは、ソース・ファイルを読み込み、構文エラーと意味上のエラーがないかどうかをチェックし、中間ファイルと呼ばれるプログラムの内部表現に書き出します。パーサの起動方法については、2-39 ページの 2.9.1 節「パーサの起動方法」を参照してください。さらにパーサは、2 つのパスで動作できます。第 1 のパスはコードの前処理を行い、第 2 のパスはコードの構文解析を行います。

オプティマイザは、オプションのパスであり、パーサとコード・ジェネレータとの間で実行されます。入力は、パーサが生成する中間ファイル (.if) です。オプティマイザを実行するときに、ユーザは最適化のレベルを選択します。オプティマイザは、中間ファイル上で最適化を行い、同じ中間ファイル形式で効率性の高いファイルを作成します。オプティマイザについては、第 3 章「コードの最適化」を参照してください。

コード・ジェネレータの入力は、パーサが作成した中間ファイル (.if) か、オプティマイザが作成した中間ファイル (.opt) のどちらかです。コード・ジェネレータは、アセンブリ言語ソース・ファイルを作成します。コード・ジェネレータの実行方法については、2-43 ページの 2.9.4 節「コード・ジェネレータの起動方法」を参照してください。

- **アセンブラ**の入力は、コード・ジェネレータが作成したアセンブリ言語ファイルです。アセンブラは COFF オブジェクト・ファイルを作成します。アセンブラの 詳細は、*TMS320C1x/C2x/C2xx/C5x* 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。

- **インターリスト・ユーティリティ**の入力は、コンパイラが作成したアセンブリ・ファイルと C ソース・ファイルです。このユーティリティは、C ファイルから得た文をアセンブリ言語のコメントとして含んだ、展開アセンブリ・ソース・ファイルを作成します。インターリスト・ユーティリティの使用方法は、2-33 ページの 2.7 節「インターリスト・ユーティリティの使用方法」、および 2-45 ページの 2.9.5 節「インターリスト・ユーティリティの起動方法」を参照してください。
- **リンカ**の入力は、アセンブラが作成した COFF オブジェクト・ファイルです。リンカは、実行可能なオブジェクト・ファイルを作成します。リンカの実行方法については、第 4 章「C コードのリンク方法」を参照してください。リンカの詳細は、*TMS320C1x/C2x/C2xx/C5x* 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。

2.9.1 パーサの起動方法

TMS320C2x/C2xx/C5x C プログラムのコンパイルの第 1 ステップは、C パーサを起動することです。パーサは、ソース・ファイルを読み取り、関数の前処理を実行し、構文をチェックし、コード・ジェネレータの入力として使用できる中間 ファイルを作成します。パーサを起動する場合には、次のように入力してください。

dspac *input file* [*output file*] [*options*]

dspac パーサを起動するコマンドです。

input file パーサが入力として使用する C ソース・ファイル名を指定します。拡張子を指定しないと、パーサはファイルの拡張子を *.c* と見なします。入力ファイル名を指定しない場合、パーサは、指定するように求めるプロンプトを出します。

output file パーサが作成する中間ファイル名を指定します。出力ファイルのファイル名を指定しない場合、パーサは入力ファイル名に拡張子 *.if* を付けたものを使用します。

options パーサの動作に影響を与えます。スタンドアロン・パーサ用の各オプションには、同じ機能を実行する、対応した dspcl オプションがあります。2-40 ページの表 2−5 は、パーサ・オプション、dspcl シェル・オプション、および対応する機能をまとめたものです。

表 2−5.　パーサ・オプションと dspcl オプション

dspac オプション	dspcl オプション	機能
−d*name* [def]	−d*name* [def]	マクロの *name* を事前定義します。
−e	−pe	E クラス・エラーを警告として扱います。
−f	−pf	関数のプロトタイプ・リスト・ファイルを生成します。
−i *dir*	−i *dir*	#include 検索パスを定義します。
−k	−pk	K&R 互換性を許容します。
−l (小文字の L)	−pl	.pp ファイルを生成します。
−n	−pn	#line 疑似命令を抑止します。
−o	−po	前処理のみを実行します。
−q	−q	進捗メッセージを出力しません (静的な実行)。
−tf	−tf	プロトタイプ・チェック条件を緩和します。
−tp	−tp	ポインタの組み合わせ条件を緩和します。
−u*name*	−u*name*	マクロの *name* の定義を解除します。
−v25	−v25	TMS320C2x 命令の使用を可能にします。
−v2xx	−v2xx	TMS320C2xx 命令の使用を可能にします。
−v50	−v50	TMS320C5x 命令の使用を可能にします。
−w	−pw	警告メッセージを抑止します。
−x	−x2	ユーザ関数のインライン展開を有効にします (−o2 を暗黙指定)。
−x0	−x0	関数のインライン展開を抑止します。
−?	−p?	3 文字符号の展開を有効にします。

dspac を単独で実行し、−l を使用して前処理リスト・ファイルを生成する場合には、前処理リスト・ファイルの名前をコマンド行で 3 番目のファイル名として指定できます。このファイル名は、ソース・ファイル名と中間ファイル名の後であれば、コマンド行のどこにあってもかまいません。

2.9.2 2 つのパスによる構文解析方法

PC などの小型ホスト・システムで非常に大きいソース・プログラムをコンパイルすると、コンパイラがメモリを使い果たし、障害を起こす可能性があります。ホスト・メモリの制限事項を回避するには、パーサを 2 つの独立したパスとして実行します。最初のパスはファイルの前処理を行い、2 番目のパスはファイルの構文解析を行います。

パーサを 1 つのパスとして実行する場合、パーサはホスト・メモリを使用して、マクロ定義とシンボル定義の両方を同時に保存します。パーサを 2 つのパスとして実行すると、これらの機能を分離できます。最初のパスは前処理だけを行うので、メモリが必要なのはマクロ定義だけです。2 番目のパスでは、マクロ定義がないので、メモリが必要なのはシンボル・テーブルだけです。

以下の例は、パーサを 2 つのパスとして実行する方法を示しています。

1) −po オプションで前処理だけを指定して、パーサを実行します。

```
dspcl –po file.c
```

−d、−u、または −i オプションを使用する場合は、この最初のパスで指定します。このパスは、前処理済みの出力ファイル file.pp を作成します。プリプロセッサの詳細は、2-22 ページの 2.5 節「プリプロセッサの制御方法」を参照してください。

2) 前処理済みのファイルでコンパイラ全体を再実行して、コンパイルを終了します。

```
dspcl file.pp
```

この最終パスでは、他のどのオプションも使用できます。

2.9.3 オプティマイザの起動方法

TMS320C2x/C2xx/C5x C プログラムのコンパイルの第 2 段階である最適化を実行するかどうかは、任意選択です。C ソース・ファイルの構文解析後、オプティマイザを使用した中間ファイルの処理を選択できます。オプティマイザにより、実行速度を上げ、C プログラムのサイズを縮小できます。オプティマイザは中間ファイルを読み込み、ユーザが選択するレベルに応じてそれを最適化し、別の中間ファイルを作成します。この最適化された中間ファイルの形式は、元の中間ファイルと同じですが、この中間ファイルを使用すると、コード・ジェネレータによって、より効率の良いコードを作成できます。

オプティマイザを起動するには、次のように入力します。

dspopt [*input file* [*output file*]] [*options*]

dspopt オプティマイザを起動するコマンドです。

input file パーサが作成する中間ファイル名を指定します。オプティマイザでは、拡張子を *.if* とします。入力ファイルを指定しない場合、オプティマイザは、指定するように求めるプロンプトを出します。

output file オプティマイザが作成する中間ファイル名を指定します。出力ファイルのファイル名を指定しない場合、オプティマイザは、入力ファイル名に拡張子 *.opt* を付けたものを使用します。

options オプティマイザによる入力ファイルの処理方法を制御します。スタンドアロン・オプティマイザで使用するオプションは、dspcl に使用するオプションと同じです。表 2−6 は、オプティマイザ・オプション、dspcl シェル・オプション、および対応する機能を示しています。各レベルで実行される最適化の詳細リストは、3-2 ページの 3.1 節「C コンパイラ・オプティマイザの使用方法」に記載されています。

表 2−6. オプティマイザ・オプションと dspcl オプション

dspopt オプション	dspcl オプション	機能
–a	–ma	変数がエイリアスをもつことを前提とします。
–b	–mb	構造体を移動することに対して、割り込み不能な RPTK 命令の使用を禁止します。
–g*register*	–r*register*	グローバル・レジスタを予約します。†
–h*n*	–ol*n*	ライブラリ関数呼び出しについての前提事項を制御します。
–i*size*	–oi*size*	自動インライン展開サイズのしきい値を設定します (–o3 のみ)。
–j	–oe	関数が割り込み関数を呼び出さないか、割り込み関数によって呼び出されないものと想定します。
–k	–pk	K&R 互換性を許容します。
–n*n*	–on*n*	最適化情報ファイルを生成します (–o3 のみ)。
–o0	–o0	レベル 0 での最適化。レジスタ最適化
–o1	–o1	レベル 0 での最適化 + ローカル最適化

† **–g** オプションを指定すると、コード・ジェネレータは、指定されたレジスタがグローバル用途用に予約されていることを認識します。詳細は、5-10 ページの 5.6 節「グローバル・レジスタ変数の作成方法」を参照してください。

表 2−6.　オプティマイザ・オプションと dspcl オプション (続き)

dspopt オプション	dspcl オプション	機能
−o2 または −o	−o2	レベル 1 での最適化 + グローバル最適化
−o3	−o3	レベル 2 での最適化 + ファイル最適化
−q	−q	進捗メッセージを出力しません (静的な実行)。
−s	−s	C ソースを差し込みます。
−v25	−v25	TMS320C2x 命令の使用を可能にします。
−v2xx	−v2xx	TMS320C2xx 命令の使用を可能にします。
−v50	−v50	TMS320C5x 命令の使用を可能にします。

2.9.4　コード・ジェネレータの起動方法

TMS320C2x/C2xx/C5x C プログラムのコンパイルの第 3 ステップは、C コード・ジェネレータを起動することです。コード・ジェネレータは、パーサが作成した中間ファイルをアセンブリ言語ソース・ファイルに変換します。この出力ファイルを変更したり、アセンブラの入力として使用したりできます。コード・ジェネレータは、アセンブルとリンクの後で ROM に保存できる再入可能で再配置可能なコードを作成します。

コード・ジェネレータを独立プログラムとして起動する場合には、次のように入力します。

dspcg [*input file* [*output file* [*tempfile*]]] [*options*]

dspcg　コード・ジェネレータを起動するコマンドです。

input file　コード・ジェネレータが入力として使用する中間ファイル名を指定します。拡張子を指定しないと、コード・ジェネレータは、ファイルの拡張子を *.if* と見なします。入力ファイルを指定しない場合、コード・ジェネレータは、指定するように求めるプロンプトを出します。

output file　コード・ジェネレータが作成するアセンブリ言語ファイルの名前を指定します。出力ファイルのファイル名を指定しないと、コード・ジェネレータは、入力ファイル名に拡張子 *.asm* を付けたものを使用します。

tempfile コード・ジェネレータが作成し、使用する一時ファイルの名前を指定します。一時ファイルのファイル名を指定しないと、コード・ジェネレータは、入力ファイル名に拡張子 *.tmp* を付けたものを使用します。コード・ジェネレータは、このファイルを使用後に削除します。

options コード・ジェネレータが入力ファイルを処理する方法を制御します。スタンドアロン・コード・ジェネレータで使用できる各オプションには、同じ機能を実行する、対応した dspcl シェル・オプションがあります。

表 2–7 は、コード・ジェネレータ・オプション、dspcl シェル・オプション、およびそれに対応する機能を示しています。

表 2–7. コード・ジェネレータ・オプションと dspcl オプション

dspcg オプション	dspcl オプション	機能
–a	–ma	変数がエイリアスをもつことを前提とします。
–b	–mb	構造体を移動することに対して、割り込み不能な RPTK 命令の使用を禁止します。
–g*register*	–r*register*	グローバル・レジスタを予約します。†
–l	–ml	LDPK 命令を減らすための最適化を抑止します。
–n	–mn	シンボリック・デバッグによって抑止された最適化を再度有効にします。
–o	–g	C ソース・レベルのデバッグを有効にします。
–q	–q	進捗メッセージを出力しません (静的な実行)。
–r	–mr	レジスタ使用情報をリスト表示します。
–s	–ms	速度ではなく、スペースを最適化します。
–v25	–v25	TMS320C2x 命令の使用を可能にします。
–v2xx	–v2xx	TMS320C2xx 命令の使用を可能にします。
–v50	–v50	TMS320C5x 命令の使用を可能にします。
–x	–mx	'C5x シリコン・バグを回避します。
–z		入力ファイルを保持します。‡

† **–g** オプションを指定すると、コード・ジェネレータは、指定されたレジスタがグローバル用途用に予約されていることを認識します。詳細は、5-10 ページの 5.6 節「グローバル・レジスタ変数の作成方法」を参照してください。

‡ **–z** オプションを指定すると、コード・ジェネレータは、入力ファイル (パーサによって作成された中間ファイル) を保持します。–z オプションを指定しない場合、中間ファイルは削除されます。

2.9.5 インターリスト・ユーティリティの起動方法

TMS320C2x/C2xx/C5x C プログラムのコンパイルの第 4 ステップを実行するかどうかは、任意選択です。プログラムのコンパイルが終わったら、インターリスト・ユーティリティをスタンドアロン・プログラムとして実行できます。インターリスト・ユーティリティをコマンド行から実行する場合の構文は、次のとおりです。

clist *asmfile* [*outfile*] [*options*]

clist インターリスト・ユーティリティを起動するコマンドです。

asmfile コンパイラからのアセンブリ言語出力です。

outfile 差し込み後の出力ファイル名を指定します。出力ファイルのファイル名を指定しない場合、インターリスト・ユーティリティは、アセンブリ言語ファイル名に拡張子 *.cl* を付けたものを使用します。

options 以下のようにユーティリティの動作を制御します。

–b ブランクと不要な行 (コメントが入っている行と {あるいは} だけが入っている行) を削除します。

–q 見出しと状況情報を削除します。

–r シンボリック・デバッグ疑似命令を削除します。

インターリスト・ユーティリティは、コード・ジェネレータが作成した .line 疑似命令を使用して、アセンブリ言語コードを C ソースと関連付けます。このため、C ソースを差し込む場合は、プログラムをコンパイルするときに、–g dspcl オプションを使用してシンボリック・デバッグを指定する必要があります。出力ファイル内にデバッグ疑似命令が不要であれば、–r インターリスト・オプションを使用して、差し込み後のファイルから削除してください。

以下の例は、function.c のコンパイルと差し込み方法を示しています。コンパイルの場合は、次のように入力します。

```
dspcl –gk –mn function
```

この例では、コンパイルを行うとともに、シンボリック・デバッグ疑似命令を生成し、アセンブリ言語ファイルを保存します。差し込みファイルを作成するには、次のように入力します。

```
clist –r function
```

この例では、差し込みファイルを作成し、シンボリック・デバッグ疑似命令を削除します。この例から得られる出力は function.cl となります。

第 3 章

コードの最適化

本コンパイラ・ツールには、ループの簡略化、ソフトウェア・パイプライン、文と式の再配置、およびレジスタへの変数の割り当てなどのタスクを実行し、C プログラムの実行速度を向上させたり、プログラム・サイズを縮小させたりする最適化プログラムが組み込まれています。

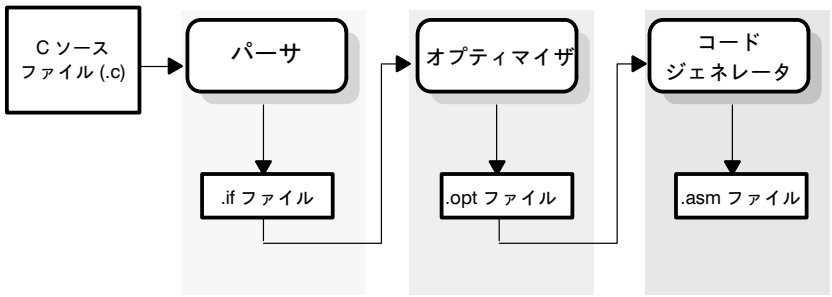
本章では、オプティマイザの起動方法、およびオプティマイザを使用したときに実行される最適化について説明します。また、インターリスト・ユーティリティをオプティマイザと組み合わせて使用する方法、および最適化されたコードのプロファイルまたはデバッグを行う方法についても説明します。

項目 **ページ**

3.1 C コンパイラ・オプティマイザの使用方法

本オプティマイザは、パーサとコード・ジェネレータ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フローを示しています。

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方法



本オプティマイザを起動する最も簡単な方法は、dspcl シェル・プログラムを使用します。この場合、dspcl 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呼び出しを展開します。

❑ **−o1**

−o0 のすべての最適化のほかに、以下の最適化を実行します。

- ローカルな複写／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ほかに、以下の最適化を実行します。

- ループの最適化を行います。
- グローバルな共通部分式を除去します。
- 不要でグローバルな代入を除去します。
- ループ内の配列参照を、増分されるポインタ形式に変換します。
- ループの展開を行います。

−o に最適化レベルを指定しない場合は、−o2 がデフォルトとして使用されます。

❑ **−o3**

−o2 のすべての最適化のほかに、以下の最適化を実行します。

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

−o3 を使用する場合、詳細は、3-4 ページの 3.2 節「−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複数の別の最適化が実行されます。特に、TMS320C2x/C2xx/C5x 固有の最適化は、オプティマイザの起動に関係なく実行されます。これらの最適化は、常に実行可能であり、最適化の選択レベルの影響を受けません。

また、dspcl 以外の方法でオプティマイザを起動することもできます。オプティマイザを独立したステップとして起動する方法については、2-41 ページの 2.9.3 節「オプティマイザの起動方法」を参照してください。

3.2 −o3 オプションの使用方法

−o3 オプションは、コンパイラに対して、ファイル・レベルの最適化を行うように指示します。−o3 オプションは、単独で使用して一般的なファイル・レベルの最適化を実行することも、他のオプションと組み合わせて、より具体的な最適化を実行することもできます。表 3−1 は、表中に記述する最適化を実行するために −o3 と組み合わせて使用するオプションを示しています。

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

3.2.1 ファイル・レベルの最適化の制御方法 (−ol*n* オプション)

−o3 オプションを指定してオプティマイザを起動すると、いくつかの最適化に標準ライブラリ関数の既知のプロパティが使用されます。こ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、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方法

ソース・ファイルの状況	使用するオプション
標準ライブラリ関数と同じ名前の関数を宣言している場合	−ol0
標準ライブラリで宣言された関数を含んでいるが、関数を変更していない場合	−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方法 (−on*n* 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、−on オプションを指定すると、読み取り可能な最適化情報ファイルを作成できます。−on に続く番号は、レベル (0、1、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方法

状況	使用するオプション
情報ファイルを必要としないが、コマンド・ファイルや環境変数で −on1 オプションまたは −on2 オプションを使用している場合。−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場合	−on1
詳細な最適化情報ファイルを作成する場合	−on2

3.3 プログラム・レベルの最適化の実行方法 (−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未適用のままになっている次のような最適化を実行します。

- ❑ 関数内の特定の引数が常に同じ値をとる場合、コンパイラはその引数を値に置き換え、引数の代わりに値を渡します。
- ❑ 関数の戻り値が一度も使用されない場合、コンパイラはその関数内の戻りコードを削除します。
- ❑ 直接か間接かにかかわらず、関数が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・レベルの最適化を表示するには、−on2 オプションを使用して情報ファイルを作成します。詳細は、3-5 ページの 3.2.2 節「最適化情報ファイルの作成方法 (–on*n* 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方法 (−op*n* オプション)

−pm −o3 を指定して起動するプログラム・レベルの最適化は、−op オプションを使用することにより制御できます。具体的には、−op オプションは、他のモジュール内の関数が、あるモジュールの外部関数を呼び出せるか、またはあるモジュールの外部変数を変更できるかを指定します。−op に続く番号は、呼び出しや変更を許可しようとしているモジュールに設定するレベルを指定します。−o3 オプションは、この情報を独自のファイル・レベル解析と組み合わせて、このモジュールの外部関数と変数の宣言を、静的に宣言された場合と同じ扱いにするかどうかを決定します。表 3−4 を使用して、−op オプションに指定する適切なレベルを選択してください。

表 3−4.　−op オプションのレベルの選択方法

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

特定の環境では、コンパイラは、指定された −op レベルとは異なるレベルに戻したり、プログラム・レベルの最適化をすべて使用不可にしたりする場合があります。表 3−5 は、コンパイラが異なる −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	関数は FUNC_EXT_CALLED プラグマによって特定されている	−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。この状況については、3-8 ページの 3.3.2 節「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 のソース・コードだけを認識し、アセンブリ・コードが指定されていても認識できません。コンパイラでは、アセンブリ・コードによる C 関数の呼び出しや C 変数の変更は認識されないので、−pm オプションを指定すると、このような C 関数は最適化の対象外になります。これらの C 関数に最適化を実行するには、関数の宣言や参照の前に FUNC_EXT_CALLED プラグマ (5-8 ページの 5.4.3 節「FUNC_EXT_CALLED プラグマ」を参照) を配置します。

プログラムの中にアセンブリ関数が指定されている場合に使用できるもう 1 つの方法は、−opn 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 節「プログラム・レベルの最適化の制御方法」を参照してください)。

一般的に、FUNC_EXT_CALLED プラグマを −pm −o3 および −op1 または −op2 と組み合わせて適切に使用することにより、最良の結果を得ることができます。

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

状況 アプリケーションは、アセンブリ関数を呼び出す C ソース・コードで構成されています。これらのアセンブリ関数は、C 関数を呼び出すことも、C 変数を変更することもしません。

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 関数の呼び出しも 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。

−pm −o3 オプションだけを指定してコンパイルすると、コンパイラは最適化レベルがデフォルトの −op2 から −op0 に戻ります。コンパイラで −op0 を使用する理由は、C で定義されているアセンブリ言語関数の呼び出しによって、他の C 関数を呼び出したり C 変数を変更したりする可能性があることを想定しているからです。

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

解決策 次の両方の解決策を試して、ご使用のコードで順調に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。

–op*n* オプションについては、3-6 ページの 3.3.1 節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。こ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。これらの C 関数は main と同じように機能し、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ために、アセンブリ言語割り込みから呼び出されるすべてのエントリ・ポイント関数に FUNC_EXT_CALLED プラグマを適用し、その後 −pm −o3 −op2 を指定してコンパイルします。必ず、すべてのエントリ・ポイント関数にこのプラグマを使用してください。このプラグマを使用しないと、コンパイラは、先頭に FUNC_EXT_CALL プラグマが指定していないエントリ・ポイント関数を除去してしまいます。
- −pm −o3 −op3 を指定してコンパイルします。FUNC_EXT_CALL プラグマを指定しないので、−op3 オプションを使用しなければなりません。このオプションは −op2 ほど強力ではなく、最適化があまり効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。この C 関数を残しておくには、FUNC_EXT_CALLED プラグマを指定します。

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

3.3.3　プログラム・コンパイル出力ファイルの名前の指定方法 (−px オプション)

−pm オプションでプログラム全体のコンパイルを指定すると、−px *filename* オプションを使用して出力ファイルの名前を指定できます。アセンブリを指定しない場合 (−n シェル・オプション)、出力ファイルのデフォルトのファイル拡張子は .asm です。アセンブリを許可する場合 (デフォルトのシェル動作)、出力ファイルのファイル拡張子は .obj です。リンクを指定する場合、−z オプションの後ろに −o オプションを指定して出力ファイルの名前を指定しなければなりません。指定しないと、出力ファイルの名前はデフォルトの a.out になります。

3.4 オプティマイザを使用する場合の特別な注意事項

本オプティマイザは、ANSI 準拠の C プログラムの正確さを維持しながら、このプログラムを改善するように設計されています。しかし、オプティマイザ用にコードを作成する場合は、プログラムが正常に稼働するように、次の特別な事項に注意してください。

3.4.1 最適化コード内で asm 文を使用する場合の注意

最適化コード内で asm (インライン・アセンブリ) 文を使用するときは、特に注意が必要です。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 ソース・コードと大きく異なってしまう場合があります。通常、asm 文を使用して割り込みマスクなどのハードウェア制御を操作することは安全ですが、asm 文を使用して C 環境とインターフェイスしたり 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恐れがあります。コンパイル後にアセンブリ出力をチェックし、asm 文が誤っていないかどうか、またプログラムの整合性が維持されているかどうかを確認してください。

3.4.2 volatile キーワードを使用する場合の注意

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 コードに書き込まれているとおりのメモリ・アクセスに依存しているコードを使用する場合は、必ず volatile キーワードを使用してそれらのアクセスを特定しなければなりません。volatile キーワードを使用して修飾された変数は、初期化されないセクションに割り当てられます。コンパイラは、volatile 変数への参照を最適化によって除去することはありません。

次の例で、このループは位置が 0xFF として読み込まれるのを待ちます。

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

この例では、`*ctrl` はループ不変式なので、このループは単一のメモリ読み取りに最適化されます。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;
```

3.4.3 エイリアス付き変数にアクセスする場合の注意

単一のオブジェクトに何通りかの方法でアクセスする場合は (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中断する場合があります。これは、間接参照によって他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。本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ある場合、オプティマイザは、これらのポインタが同じオブジェクトを指すものと見なします。

本コンパイラは、ローカル変数のアドレスが関数に渡されると、その関数がポインタによる書き込みによってローカル変数を変更する可能性があるものと見なします。このため、戻った後のローカル変数のアドレスはどこにも使用できません。たとえば、呼び出し先の関数がローカル変数のアドレスをグローバル変数に割り当てたり、ローカル変数のアドレスを戻したりすることはできません。

3.4.4 割り込み関数でないことの想定

−oe オプションでは、モジュールの関数がどれも割り込み関数でなく、割り込みによって呼び出されることもなく、または他の方法で非同期で実行されることもないものと見なされます。これにより、オプティマイザは特定の変数割り当てを最適化できます。−oe オプションは、オプティマイザを自動的にレベル 2 で起動します。

また、−oe オプションは、モジュールがいずれも再帰的 (直接的か間接的かにかかわらず) に呼び出されないものと見なします。再帰的関数が含まれているモジュールと −oe とを組み合わせて使用しないように注意してください。

3.5 自動インライン展開 (−oi オプション)

−o3 オプションを使用して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 –o*size* は、インライン展開する関数のサイズを指定します。−oi を使用する場合は、インライン展開する最大の関数に対して *size* の限度を指定します。–o*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、サイズで制御するすべてのインライン展開は抑止されます。
- *size* パラメータをゼロ以外の整数に設定した場合、コンパイラは *size* に基づいて関数をインライン展開します。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。その結果、算出された値が size パラメータより小さい場合にだけ、関数をインライン展開します。コンパイラは関数のサイズを任意の単位で測定しますが、最適化情報ファイル (−on1 または −on2 オプションで作成する) では、各関数のサイズが −oi オプションと同じ単位で報告されます。

–o*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。–o*size* オプションを使用しない場合でも、オプティマイザは非常に小さい関数をインライン展開します。−x オプションは、インラインとして宣言された関数のインライン展開を制御します (2-30 ページの 2.6.3.1 節を参照)。

3.6 インターリスト・ユーティリティとオプティマイザを組み合わせて使用する方法

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

- ❑ −os オプションを使用すると、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 オプションを一緒に使用すると、オプティマイザのコメントと元の C ソースがアセンブリ・コードに差し込まれます。

オプティマイザで −os オプションを使用した場合、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、オプティマイザはコードの中にコメントを挿入し、そのコメントには、オプティマイザがどのようにコードの再配置と最適化を実行したかが示されます。これらのコメントは、アセンブリ言語ファイルの中に ;** で始まるコメントとして出力されます。C 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を除き、差し込まれません。

インターリスト・ユーティリティは、C の文の境界にまたがった一部の最適化を不可能にするので、最適化後のコードに影響を及ぼす場合があります。最適化は、通常のソースの差し込みを不可能にします。これは、オプティマイザが広範囲にわたってプログラムの再配置を行うからです。したがって、−os オプションを使用した場合、オプティマイザは再構築後の C の文を書き込みます。

3.7 最適化されたコードのデバッグ方法

最適化しないでプログラムをデバッグし、それを最適化した後で正確さを再検証するのが、理想的な方法です。最適化されたコードでデバッガを使用できますが、オプティマイザが大幅なコードの再配置を行うほか、複数の変数を 1 つのレジスタに割り当てるので、ソース・コードとオブジェクト・コードを関連付けるのが難しくなります。

注: シンボリック・デバッグと最適化コード

−g オプションを使用して、シンボリック・デバッグ情報を生成すると、コード・ジェネレータの最適化の多くが無効にされます。これは、最適化によってデバッガの動作が影響を受けるからです。シンボリック・デバッグを使用して、同時に完全に最適化されたコードを生成する場合は、−mn オプションを使用してください。−mn は、−g によって抑止された最適化を再び有効にします。

3.8 実行できる最適化の種類

TMS320C2x/C2xx/C5x C コンパイラは、さまざまな最適化の技法を使用して C プログラムの実行速度を向上させ、サイズを小さくします。最適化は、コンパイラ全体を通じて各種のレベルで実行されます。

ここで説明するほとんどの最適化は、独立したオプティマイザ・パスによって実行されます。ユーザは、−o コンパイラ・オプションを指定してこのパスを有効にし、制御できます (3-2 ページの 3.1 節「コンパイラ・オプティマイザの使用方法」を参照)。しかし、コード・ジェネレータが実行する一部の最適化は、選択して有効にしたり、抑止したりできません。

コンパイラによって行われる最適化は、次のとおりです。これらの最適化は、すべての C コードを改善します。

最適化	ページ
コストに基づいたレジスタ割り当て	3-15
自動インクリメント・アドレッシング	3-15
ブロックのリピート	3-15
遅延、分岐、コール、およびリターン	3-16
代数表記の並べ換え、シンボルの簡略化、および定数の畳み込み	3-18
エイリアスの明確化	3-18
データ・フローの最適化 ❑ 複写伝播 ❑ 共通部分式の除去 ❑ 冗長な代入の除去	3-18
分岐の最適化と制御フローの簡略化	3-20
ループ誘導変数の最適化と強度換算	3-21
ループの循環	3-21
ループ不変コードの移動	3-21
ランタイムサポート・ライブラリ関数のインライン展開	3-21

3.8.1 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、他の変数より高い優先順位が割り当てられます。他の変数と重複して使用されない変数は、同じレジスタに割り当てられる場合があります。

3.8.2 自動インクリメント・アドレッシング

*p++ という形式のポインタ式の場合、コンパイラは、効率的な TMS320C2x/C2xx/C5x 自動インクリメント・アドレッシング・モードを使用します。多くの場合、ループ内で配列を順に参照するコードの場合 (たとえば、for (i = 0; i < n; ++i) a[i]...)、配列参照は、ループの最適化によって、自動的にインクリメントされるレジスタ変数ポインタによる間接参照に変換されます。例 3−1 を参照してください。

3.8.3 ブロックのリピート

TMS320C2x/C2xx/C5x 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し、効率的なリピート形式を使用したループを生成します。反復回数は、定数と式のどちらでもかまいません。リピート・ブロック命令がない TMS320C2x の場合、コンパイラは AR をループ・カウンタとして割り当て、BANZ 命令を指定してループを設定します。例 3−1、および例 3−5 を参照してください。

例 3−1.　リピート・ブロック、自動インクリメント・アドレッシング、並列命令、強度換算、誘導変数除去、レジスタ変数、およびループ・テスト置換

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

TMS320C2x/C2xx/C5x C コンパイラの出力

```
_scale:
   . . .
   LRLK   AR6,_a           ; AR6 = &a[0]
   LRLK   AR5,_b           ; AR5 = &b[0]
   LACK   9
   SAMM   BRCR             ; BRCR = 9
   LARK   AR2,-3+LF1       ; AR2 = &k
   MAR    *0+,AR5
   RPTB   L4-1             ; repeat block 10 times
   LT     *+,AR2           ; t = *AR5++
   MPY    * ,AR6           ; p = t * *AR2
   SPL    *+,AR5           ; *AR6++ = p
L4:
   . . .
```

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。強度換算は、配列参照を効率的なポインタ自動インクリメントにします。

3.8.4　遅延、分岐、コール、およびリターン

TMS320C5x には、複数の遅延分岐命令、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、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、非遅延命令より 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も 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。例 3−2 を参照してください。

例 3−2.　遅延分岐命令、遅延コール命令、および遅延リターン命令

```
main()
{
    int i0, i1;

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

TMS320C2x/C2xx/C5x C コンパイラの出力

```
_main:
      SAR    AR0,*+           ; function prolog
      POPD   *+               ; save AR0 and return address
      SAR    AR1,*            ; begin to set up local frame
      BD     L2               ; begin branch to loop control
      LARK   AR0,3            ; finish setting up local frame
      LAR    AR0,*0+

***   B      L2 OCCURS        ; branch to loop control
L1:                           ; loop body
      LARK   AR2,2            ; AR2 = &i1
      MAR    *0+
      LAC    *-,AR1           ; ACC = *AR2, AR2 = &i0
      SACL   *+,AR2           ; stack ACC
      CALLD  _process         ; begin call
      LAC    * ,AR1           ; ACC = *AR2
      SACL   *+               ; stack ACC
***   CALL   _process OCCURS  ; call occurs
      SBRK   2                ; pop stack
L2:                           ; loop control
      MAR    * ,AR5           ; AR5 = &i0
      LARK   AR5,1
      CALLD  _input           ; begin call
      MAR    *0+,AR1
      SAR    AR5,*+           ; stack AR5
***   CALL   _input OCCURS    ; call occurs
      MAR    *-               ; clear stack
      BZ     EPI0_1           ; quit if _input returns 0
      MAR    * ,AR4           ; AR4 = &i1
      LARK   AR4,2
      CALLD  _input           ; begin call
      MAR    *0+,AR1
      SAR    AR4,*+           ; stack AR4
***   CALL   _input OCCURS    ; call occurs
      MAR    *-,AR2           ; clear stack
      BNZ    L1               ; continue if _input returns !0
EPI0_1:
      MAR    * ,AR1           ; function epilog
      SBRK   4                ; clear local frame
      PSHD   *-               ; push return address on hardware stack
      RETD                    ; begin return
      LAR    AR0,*            ; restore AR0
      NOP                     ; necessary, no PSHD in delay slot
***   RET OCCURS              ; return occurs
      . . .
```

3.8.5 代数表記の並べ換え／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、式を簡略化して、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。例 3−3 を参照してください。

3.8.6 エイリアスの明確化

一般に、C 言語で作成されたプログラムでは、多数のポインタ変数が使用されます。多くの場合、コンパイラは、2 つ以上の l (小文字の L) 値 (シンボル、ポインタ参照、または構造体参照) が同じメモリ位置を参照しているかどうかを判断できません。このメモリ位置のエイリアス指定により、コンパイラがレジスタ内に値を保持できなくなる場合が少なくありません。その理由は、時間が経過すると、レジスタとメモリが同じ値を保持し続けているかどうかを保証できないからです。エイリアスの明確化は、2 つのポインタ式が同じ位置を指す可能性がなくなる時期を判別する技法です。この技法を使用すると、コンパイラが自由にそれらの式を最適化できるようになります。

3.8.7 データ・フローの最適化

以下の 3 種類のデータ・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。例 3−3、および例 3−4 を参照してください。

- **複写伝播**

  変数への代入が終わると、コンパイラは、その変数の参照を代入された値に置換します。この値には、別の変数、定数、または共通部分式を指定できます。その結果、定数の畳み込みや共通部分式の除去、または変数全体の除去にも十分に活用できます。例 3−3、および例 3−4 を参照してください。

- **共通部分式の除去**

  複数の式から同じ値が得られる場合、コンパイラは値を一度だけ計算し、それを保存して再利用します。例 3−3 を参照してください。

- **冗長な代入の除去**

  多くの場合、複写伝播と共通部分式の除去による最適化の結果、変数への不要な代入 (それ以降、別の代入があるまで、または関数が終了するまで次の参照がない変数) が生じます。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 3−3 を参照してください。

例 3−3.　データ・フローの最適化

```
   simp(int j)
   {
        int a = 3;
        int b = (j*a) + (j*2);
        int c = (j < a);
        int d = (j >> 3) + (j < b);

        call(a,b,c,d);
        ...
   }
```

TMS320C2x/C2xx/C5x C コンパイラの出力

```
_simp:
   . . .
*****************************************************
* b = j * 5;
*****************************************************
     LARK   AR2,-3+LF1       ; AR2 = &j
     MAR    *0+
     LT     *                ; t = *AR2
     MPYK   5                ; p = t * 5
     ADRK   4-LF1            ; AR2 = &b
     SPL    *                ; *AR2 = p
*****************************************************
* call(3, b, j < 3, (j >> 3) + (j < b));
*****************************************************
     LT     *                ; t = *AR2 (b)
     SBRK   4-LF1            ; AR2 = &j
     LACT   * ,AR1           ; ACC = j < b
     SACL   * ,AR2           ; save off ACC on TOS (top of stack)
     SSXM                    ; need sign extension for right shift
     LAC    * ,12,AR1        ; high ACC = j >> 3
     ADD    * ,15            ; add TOS to high ACC
     SACH   *+,1,AR2         ; stack high ACC
     LAC    * ,3,AR1         ; ACC = j < 3
     SACL   *+,AR2           ; stack ACC
     ADRK   4-LF1            ; AR2 = &b
     LAC    * ,AR1           ; ACC = b
     SACL   *+               ; stack ACC
     CALLD  _call            ; call begins
     LACK   3                ; ACC = 3
     SACL   *+               ; stack ACC
***  CALL   _call OCCURS     ; call occurs
     . . .
```

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ます。そして a は不要な変数となり、除去されます。j に 3 (a) 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、これは共通部分式として認識されます。c と d への代入は不要であり、式に置き換えられます。このような最適化は、ジャンプにまたがって実行されます。

3.8.8 分岐の最適化と制御フローの簡略化

本コンパイラは、プログラムの分岐動作を解析し、オペレーションを直線的なシーケンス (基本ブロック) に再配置することにより、分岐や冗長な条件を除去します。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ることもあります。単純な制御フロー構造は条件付き命令に簡略化され、必要な分岐の数が全体的に少なくなります。例 3−4 を参照してください。

例 3−4. 複写伝播と制御フローの簡略化

```
     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;
         }
     }
```

TMS320C2x/C2xx/C5x C コンパイラの出力

```
_fsm:
      . . .
*
* AR5 assigned to variable 'input'
*
      LAC    *+      ; initial state == ALPHA
      BNZ    L5      ; if (input != 0) go to state GAMMA
L2:
      LAC    *+      ; state == BETA
      BZ     L4      ; if (input == 0) go to state GAMMA
      LAC    *+      ; state == ALPHA
      BZ     L2      ; if (input == 0) go to state BETA
      B      L5      ; else            go to state GAMMA
L4:
      LAC    *+      ; state == GAMMA
      BNZ    EPI0_1  ; if (input != 0) go to state OMEGA
L5:
      LARP   AR5
L6:
      LAC    *+      ; state = GAMMA
      BZ     L6      ; if (input == 0) go to state GAMMA
EPI0_1:               ; state == OMEGA
      ...
```

この単純な有限状態機械の例では、switch 文および state 変数に対して完全に最適化が行われ、一連の再編成された条件付き分岐が残ります。

3.8.9 ループ誘導変数の最適化と強度換算

ループ誘導変数とは、あるループの値がそのループ内の実行回数に直接関係する変数のことです。ループでの配列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、さらに効率的な式に置換するプロセスのことです。たとえば、インデックスを使用して一連の配列要素を参照するコードは、ポインタを使用してその配列をインクリメントするコードに置換されます。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、TMS320C2x/C2xx/C5x リピート・ブロックとして、あるいは効率的なデクリメント分岐命令を使用して実現されます。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ます。この場合には、そのループの制御変数は完全に除去できます。例 3−1、および例 3−5 を参照してください。

3.8.10 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、ループ外への無駄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され、除去されます。

3.8.11 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。この計算は、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。例 3−5 を参照してください。

3.8.12 ランタイムサポート・ライブラリ関数のインライン展開

本コンパイラは、小さなランタイムサポート関数の呼び出しをインライン・コードに置換することにより、関数呼び出しに関連したオーバ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 を参照してください。

例 3−5.　関数のインライン展開

```
#include <string.h>
struct s { int a,b,c[10]; };
struct t { int x,y,z[10]; };

proc_str(struct s *ps, struct t *pt)
{
   . . .
    memcpy(ps,pt,sizeof(*ps));
   . . .
}
_proc_str:
   . . .
```

TMS320C2x/C2xx/C5x C コンパイラの出力

```
*
* AR5 assigned to variable 'memcpy_1_rfrom'
* AR6 assigned to variable 'memcpy_1_rto'
* BRCR       assigned to temp var 'L$1'
*
     . . .

     LARK   AR2,-3+LF1 ; AR2 = &ps
     MAR    *0+
     LAR    AR6,*-     ; AR6 = ps, AR2 = &pt
     LAR    AR5,* ,AR5 ; AR5 = pt
     LACK   11
     SAMM   BRCR       ; repeat 12 times
     RPTB   L4-1
     LAC    *+,AR6     ; *ps++ = *pt++
     SACL   *+,AR5
     NOP               ; must have 3 words in repeat block
L4:
     . . .
```

本コンパイラは、インライン・ライブラリの中から C 関数 memcpy() の中間ファイル・コードを検出し、呼び出しの代わりに複写します。memcpy のパラメータに対応して、変数 memcpy_1_from と memcpy_1_to が作成されることに注目してください (多くの場合、複写伝播では、引数が呼び出し後に参照されない場合は、インライン展開された関数のパラメータへのこのような代入を除去できます)。

第 4 章

C コードのリンク方法

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。
- ❑ dspcl 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。

本章では、それぞれの方法によるリンカの起動方法について説明します。また、C コードのリンクに関する特別な要件、たとえばランタイムサポート・ライブラリの取り込み方法、初期化モデルの指定方法、プログラムをメモリに割り振る方法についても説明します。リンカの詳細は、*TMS320C1x/C2x/C2xx/C5x* 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後で、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 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。

dsplnk {**–c**|**–cr**} *filenames* [*-options*] [**–o** *name.out*] **–l** *libraryname* [*lnk.cmd*]

dsplnk	リンカを起動するコマンドです。
–c | **–cr**	C 環境で定義している特別規則を使用することを、リンカに伝えるオプションです。dsplnk を使用する場合は、−c または −cr を必ず使用しなければなりません。−c オプションを指定すると、実行時に変数の自動初期化を行います。−cr オプションを指定すると、ロード時に変数の自動初期化を行います。
filenames	オブジェクト・ファイル、リンカ・コマンド・ファイル、またはアーカイブ・ライブラリの名前を指定します。すべての入力ファイルのデフォルト拡張子は、*.obj* です。これ以外の拡張子を使用する場合は、明示的に指定しなければなりません。リンカは、入力ファイルがオブジェクトであるか、あるいはリンカ・コマンドが含まれた ASCII ファイルであるかを判別できます。−o オプションを使用して出力ファイル名を指定した場合を除き、デフォルトの出力ファイル名は *a.out* 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 節を参照してください)。
–o *name.out*	−o オプションは、出力ファイルの名前を指定します。
–l *libraryname*	l (小文字の L) は、C ランタイムサポート関数、および浮動小数点算術関数を含んだ適切なアーカイブ・ライブラリを識別します (–l オプションは、そのファイルがアーカイブ・ライブラリであることをリンカに伝えます)。C コードをリンクしている場合は、ランタイムサポート・ライブラリを使用しなければなりません。コンパイラに添付されているライブラリを使用しても、または独自のランタイムサポート・ライブラリを作成して使用しても構いません。リンカ・コマンド・ファイル内でランタイムサポート・ライブラリを指定する場合、このパラメータは不要です。
lnk.cmd	リンカのオプション、ファイル名、疑似命令、またはコマンドを含みます。

表 4−1. ランタイムサポート・ライブラリ

ライブラリ名	ライブラリの内容
rts25.lib	TMS320C2x ランタイムサポート
rts2xx.lib	TMS320C2xx ランタイムサポート
rts50.lib	TMS320C5x ランタイムサポート

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・メンバのみを取り込み、リンクします。たとえば、prog1、prog2、および prog3 の各モジュールから構成されている 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。

```
dsplnk –c prog1 prog2 prog3 –o prog.out –l rts25.lib
```

リンカは、デフォルトの割り振りアルゴリズムを使用して、プログラムをメモリに割り振ります。リンカ・コマンド・ファイルの中で MEMORY および SECTIONS の疑似命令を使用すると、割り振り処理をカスタマイズできます。詳細は、*TMS320C2x/C2x/C2xx/C5x* 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、コンパイルの一環としてリンクを実行する場合、リンカ・オプションは –z オプションの後ろに指定しなければなりません (2-4 ページの 2.2 節「C コンパイラ・シェルの起動方法」を参照)。

デフォルトでは、コンパイラ・シェルはリンカを起動しません。しかし –z オプションを指定すると、プログラムのコンパイル、アセンブル、およびリンクが 1 ステップで実行されます。–z を使用してリンクを実行する場合は、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 の後ろにあるオプション) 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 の後ろに指定した引数は、すべてリンカへ渡されます。これらの引数には、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。

```
dspcl –sq *.c –z c.cmd –o prog.out –l rts25.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、コンパイルされたオブジェクト・ファイルを、リンカが c.cmd コマンド・ファイルを使用してリンクします。–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 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 シェル・オプションを使用すると、−z オプションを無効にできます。−c オプションは、C_OPTION 環境変数の中で −z オプションを指定しているとき、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 シェル・オプションとは異なる独自の機能を備えています。デフォルトでは、−z オプションを使用した場合、コンパイラは −c リンカ・オプションを使用します。このオプションはリンカに対して、C のリンク規則 (実行時の変数の自動初期化) を使用するように指示します。ロード時に変数を自動初期化する場合には、−z オプションの後ろに −cr リンカ・オプションを指定します。

4.4 リンカ・オプション

−z シェル・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以下に示します。

−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 のどちらも指定しなかった場合、リンカは、−a が指定された場合と同じ動作をします。
−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。詳細は、6-34 ページの 6.8.4 節を参照してください。
−cr	ロード時に変数を初期化します。詳細は、6-35 ページの 6.8.5 節を参照してください。
−e *global_symbol*	出力モジュールのプライマリ・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* をグローバル・シンボルとして定義します。
−h	すべてのグローバル・シンボルを静的シンボルにします。
−heap *size*	ヒープ・サイズ (動的メモリ割り当て) を *size* で指定したワード数に設定し、ヒープ・サイズを指定するグローバル・シンボルを定義します。デフォルトは 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。
−l *libraryname*	l (小文字の L) は、アーカイブ・ライブラリ・ファイルまたはリンカ・コマンド・ファイルの名前を、リンカへの入力として指定します。*libraryname* はアーカイブ・ライブラリの名前であり、オペレーティング・システムの規則に従ったものでなければなりません。
−m *filename*	ホールを含む入出力セクションのマップまたはリストを作成し、*filename* で指定したファイルにそのリストを格納します。ファイル名は、オペレーティング・システムの規則に従ったものでなければなりません。
−n	memory 疑似命令内のすべての fill (埋め込み) 指定を無視します。プロジェクトの開発段階では、memory 疑似命令の fill 指定によって大きな .out ファイルが生成されることを回避するために、このオプションを使用してください。

–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 (見出しと進捗情報の抑止) 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。
–v0	バージョン 0 の COFF フォーマットを生成します。
–v1	バージョン 1 の COFF フォーマットを生成します。
–v2	バージョン 2 の COFF フォーマットを生成します。
–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、*TMS320C1x/C2x/C2xx/C5x* 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・プロセスの制御方法

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

- ❏ コンパイラのランタイムサポート・ライブラリを組み込む
- ❏ 初期化モデルを指定する
- ❏ 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、*TMS320C1x/ C2x/C2xx/C5x* 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。

4.5.1 ランタイムサポート・ライブラリとのリンク方法

C プログラムは、すべてランタイムサポート・ライブラリとリンクしなければなりません。このライブラリには、標準 C 関数および本コンパイラが C 環境を管理するために使用する関数も組み込まれています。−l リンカ・オプションを使用して、どの TMS320C2x/C2xx/C5x ランタイムサポート・ライブラリを使用するかを指定しなければなりません。また、−l オプションはリンカに対して、アーカイブ・パスやオブジェクト・ファイルを検索する際に −i オプションに注目し、次に C_DIR 環境変数に注目するように伝えます。−l オプションを使用するには、次のようにコマンド行に入力します。

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

コンパイラ・パッケージには、標準ランタイムサポート・ライブラリの 3 つのバージョンが含まれています。つまり、TMS320C2x プログラム用の rts25.lib、TMS320C2xx プログラム用の rts2xx.lib、および TMS320C5x プログラム用の rts50.lib です。

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しなければなりません。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。−c または −cr を使用し、rts25.lib、rts2xx.lib または rts50.lib をリンクするようにした場合、リンカは自動的に boot.obj を抽出してリンクします。

注: _c_int0 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_int0 があります。この _c_int0 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_int0 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_int0 への分岐を設定してください。

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、以下の作業を実行します。

1) スタックを設定します。

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

3) main を呼び出します。

4) main が戻ったときに exit を呼び出します。

ライブラリに含まれているその他のランタイムサポート関数については、第 7 章を参照してください。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。

4.5.2 初期化のタイプの指定方法

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。これらのテーブルのフォーマットについては、6-33 ページの 6.8.3 節「初期化テーブル」を参照してください。これらのテーブルは、*.cinit* という名前付き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。

- 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 (6-34 ページの 6.8.4 節「実行時の変数の自動初期化」を参照)。
- グローバル変数はロード時に初期化されます。−cr リンカ・オプションを使用してください (6-35 ページの 6.8.5 節「ロード時の変数の初期化」を参照)。

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しなければなりません。これらのオプションは、リンカに対して、初期化を実行時に行うかロード時に行うかを選択するように伝えます。プログラムをコンパイルし、リンクする場合は、−c リンカ・オプションがデフォルトになります。−c リンカ・オプションを使用する場合は、−z オプションの後ろに置かなければなりません (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」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_int0 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 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_int0 が自動的に参照されます。その結果、boot.obj がランタイムサポート・ライブラリから自動的にリンクされます。
- *.cinit* 出力セクションには終了レコードが埋め込まれているので、ローダ (ロード時の初期化) やブート・ルーチン (実行時の初期化) が、初期化テーブルの読み取りを停止する時期を認識できます。
- ロード時に自動初期化を行うと (−cr リンカ・オプション)、次のことが実行されます。
  - 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 (010h) が .cinit セクション・ヘッダ内に設定されます。STYP_COPY は、ローダに対して自動初期化を直接実行し、.cinit セクションをメモリにロードしないように通知する特別な属性です。リンカは、メモリ内に .cinit セクション用のスペースを割り当てません。
- 実行時に自動初期化を行うと (−c リンカ・オプション)、リンカは、.cinit セクションの開始アドレスとしてシンボル cinit を定義します。ブート・ルーチンは、このシンボルを自動初期化用の開始点として使用します。

4.5.3 セクションをメモリ内の指定場所に割り振る方法

コンパイラは、コードとデータが入った再配置可能ブロックを作成します。これらのブロックはセクションと呼ばれ、各種の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、初期化されたセクションと初期化されないセクションという基本的な 2 種類のセクションがあります。表 4−2 は、それらのセクションをまとめたものです。

表 4−2. コンパイラが作成するセクション

(a) 初期化されたセクション

名前	内容
.cinit	明示的に初期化されるグローバル変数と静的変数のテーブル
.const	明示的に初期化される文字列リテラル、グローバル const 変数、静的 const 変数
.switch	大きな switch 文用のジャンプ・テーブル
.text	実行可能なコードと浮動小数点定数

(b) 初期化されないセクション

名前	内容
.bss	グローバル変数および静的変数
.stack	ソフトウェア・スタック
.sysmem	malloc 関数 (heap) 用の動的メモリ領域

プログラムをリンクするときには、セクションをメモリ内のどこに割り振るかを指定しなければなりません。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。.const セクションは書き込まれることはないので、ROM に置くことができますが、アクセス方法のためデータ・メモリとして構成しなければなりません (詳細は、6-5 ページの 6.1.3 節「プログラム・メモリへの .const の割り当て方法」を参照)。コンパイラがこれらのセクションをどのように使用するかについては、6-3 ページの 6.1.1 節「セクション」を参照してください。リンカは、セクションを割り振るための MEMORY 疑似命令と SECTIONS 疑似命令を備えています。

次の表は、各セクション・タイプに必要なメモリの種類とページ指定を示しています。

セクション	メモリの種類	ページ
.text	ROM または RAM	0
.cinit	ROM または RAM	0
.const	ROM または RAM	1
.switch	ROM または RAM	0
.bss	RAM	1
.stack	RAM	1
.sysmem	RAM	1

セクションをメモリ内に割り振る方法の詳細は、*TMS320C1x/C2x/C2xx/C5x* アセンブリ言語ツール ユーザーズ・マニュアルの「リンカ」の章を参照してください。

4.5.4 リンカ・コマンド・ファイルの例

例 4−1 は、C プログラムのリンクに使用できる典型的なリンカ・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、link.cmd という名前で、次のリンカ・オプションが指定されています。

−c リンカに対して、実行時に自動初期化を使用するように指示します。

−m リンカに対して、マップ・ファイルを作成するように指示します。この例のマップ・ファイルの名前は example.map です。

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

例 4−1. リンカ・コマンド・ファイル例

```
/************************************************/
/        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 rts25.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
}
```

続いて、リンクするすべてのオブジェクト・ファイルが指定されています。この C プログラムは、2 つの C モジュール main.c および sub.c で構成されています。これらのモジュールがコンパイルされ、アセンブルされて、main.obj と sub.obj の 2 つのオブジェクト・ファイルが作成されました。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* シンボルを定義しなければなりません。これは、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・ファイルは、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・ライブラリが指定されています (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 プログラムなので、ランタイムサポート・ライブラリ (rts25.lib、rts2xx.lib、または rts50.lib) を指定しなければなりません。このプログラムは、matrix.lib と呼ばれるアーカイブ・ライブラリに含まれるいくつかのルーチンを使用するので、このライブラリはリンカへの入力としても名前が指定されています。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ることに注意してください。

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。

```
dsplnk link.cmd
```

使用するシステムで機能させるには、MEMORY 疑似命令と、多分 SECTIONS 疑似命令に修正を加えなければなりません。これらの疑似命令については、*TMS320C1x/C2x/C2xx/C5x アセンブリ言語ツール ユーザーズ・マニュアル*の「リンカ」の章を参照してください。

第 5 章

TMS320C2x/C2xx/C5x C 言語

TMS320C2x/C2xx/C5x C コンパイラは、C プログラム言語を標準化する目的で ANSI (米国規格協会) の委員会で制定された C 言語規格をサポートしています。

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 規格に基づくものです。ANSI C は現在のさまざまな C コンパイラが備えている言語拡張機能の多くを取り込み、以前に仕様が定まっていなかった言語の多数の特性を正式な仕様としています。

項目 **ページ**

5.1 TMS320C2x/C2xx/C5x C 言語の特性

ANSI 規格では、ターゲット・プロセッサ、ランタイム環境、あるいはホスト環境の特性によって影響を受ける C 言語のいくつかの事項について記載しています。これらの事項は、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、TMS320C2x/C2xx/C5x C コンパイラで、これらの特性がどのように実装されているかを説明します。

以下に、これらの事項のすべてを取り上げ、それぞれに対する TMS320C2x/C2xx/C5x 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、正式な ANSI 規格、またはカーニハンとリッチー (K&R) による *The C Programming Language* の第 2 版に対するものです。

5.1.1 識別子と定数

- ❑ 識別子に関しては、先頭の 100 文字が有効です。また、各識別子ごとに大文字と小文字が区別されています。これらの特性は、内部および外部を問わずすべての識別子に適用されます。
  (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。たとえば、次のとおりです。

  ```
  '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については、5-5 ページの 5.2 節「データ型」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 integer 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。
  (ANSI 3.3.4, K&R A6.6)

5.1.4 式

- 2 つの符号付き整数で除算をしたとき、どちらかが負であれば、商は負になり、剰余の符号は被除数の符号と同じになります。スラッシュ記号 (/) は商を求めるために使用し、パーセント記号 (%) は剰余を求めるために使用します。たとえば、次のとおりです。

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

(ANSI 3.3.5, K&R A7.6)

- 符号付きの値の右シフトは、算術シフトです。つまり、符号は保存されます。
(ANSI 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 A8.1)

- 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位置合わせされます。
(ANSI 3.5.2.1, K&R A8.3)

- 整数として定義されたビット・フィールドは、符号付きです。ビット・フィールドは、下位のビットから順にワードにパックされます。ワード環境をまたがることはありません。
(ANSI 3.5.2.1, K&R A8.3)

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)

以下のプラグマがサポートされています。

- CODE_SECTION
- DATA_SECTION
- FUNC_EXT_CALLED

プラグマの詳細は、5-7 ページの 5.4 節を参照してください。

5.2 データ型

- 整数型 (char、short、int、およびそれぞれの符号なし型) は、すべて型が等価であり、16 ビット 2 進数値で表されます。
- long 型および符号なしの long 型は、32 ビット 2 進数値で表されます。
- 符号付きの型は 2 の補数表記で表されます。
- char 型は符号付きの型であり、 int と等価です。
- enum 型のオブジェクトは 16 ビット値で表されます。式の中では、enum 型は int と等価です。
- 浮動小数点型 (float、double、および long double) はすべて等価であり、TMS320C2x/C2xx/C5x の 32 ビット浮動小数点形式で表されます。
- long と float は、下位アドレスに最小重みワードを入れてメモリに格納されます。

表 5–1 は、各スカラ・データ型のサイズ、表現、および範囲を掲載しています。

表 5−1. TMS320C2x/C2xx/C5x C のデータ型

型	サイズ	表現	範囲 最小	範囲 最大
char, signed char	16 ビット	ASCII	−32768	32767
unsigned char	16 ビット	ASCII	0	65535
short	16 ビット	2 の補数	−32768	32767
unsigned short	16 ビット	2 進	0	65535
int, signed int	16 ビット	2 の補数	−32768	32767
unsigned int	16 ビット	2 進	0	65535
long, signed long	32 ビット	2 の補数	−2147483648	2147483647
unsigned long	32 ビット	2 進	0	4294967295
enum	16 ビット	2 の補数	−32768	32767
float	32 ビット	TMS320C2x/C2xx/C5x	1.19209290e−38	3.4028235e+38
double	32 ビット	TMS320C2x/C2xx/C5x	1.19209290e−38	3.4028235e+38
long double	32 ビット	TMS320C2x/C2xx/C5x	1.19209290e−38	3.4028235e+38
ポインタ	16 ビット	2 進	0	0xFFFF

範囲値の多くは、コンパイラに付属の ヘッダ・ファイル limits.h 内で標準マクロとして使用できます。詳細は、7-6 ページの 7.2.4 節「制限値 (float.h と limits.h)」を参照してください。

注: TMS320C2x/C2xx/C5x のバイトは 16 ビットです

ANSI C 定義では、sizeof 演算子によってオブジェクトを格納するために必要なバイト数が生成されます。さらに ANSI では、sizeof が char に適用される場合、結果は 1 であると規定されています。TMS320C2x/C2xx/C5x では char は 16 ビット (個別にアドレス指定できるように) なので、1 バイトも 16 ビットになります。これにより、予期しない結果が発生します。たとえば、sizeof (int) = = 1 (2ではない) などです。
TMS320C2x/C2xx/C5x では、バイトとワードは等しく (ともに 16 ビット) なります。

5.3 レジスタ変数

C コンパイラは、1 つの関数の中で最高 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内のレジスタ宣言は、通常の変数として扱われます。

コンパイラは、レジスタ変数に AR6 と AR7 を使用します。

- ❑ AR6 は、最初のレジスタ変数に割り当てられます。
- ❑ AR7 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、割り当てられたレジスタに入れられます。これにより、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。この機能を効率良く使用するには、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。

オプティマイザもレジスタ変数を作成しますが、使用方法が異なっています。

5.4 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。TMS320C2x/C2xx/C5x C コンパイラは、以下のプラグマをサポートしています。

- ❑ CODE_SECTION
- ❑ DATA_SECTION
- ❑ FUNC_EXT_CALLED

上記のプラグマのうちの 2 つが、引数 *func* と *symbol* を使用します。これらの引数には、ファイル・スコープが必要です。 つまり、これらの引数を関数の 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 する必要があり、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。

5.4.1 CODE_SECTION プラグマ

CODE_SECTION プラグマは、*section name* セクションに *symbol* 用のスペースを割り当てます。このプラグマの構文は、次のとおりです。

#pragma CODE_SECTION (*symbol*, *"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(fn, "my_sect")

int fn(int x)
{
   return c;
}
```

(b) アセンブリ・ソース・ファイル

```
   .file   "CODEN.c"
   .sect   "my_sect"
   .global _fn
   .sym    _fn,_fn,36,2,0
   .func   3
```

5.4.2 DATA_SECTION プラグマ

DATA_SECTION プラグマは、*section name* セクションに *symbol* 用のスペースを割り当てます。このプラグマの構文は、次のとおりです。

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

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

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) アセンブリ・ソース・ファイル

```
            .global _bufferA
            .bss     _bufferA,512,4
            .global _bufferB
_bufferB:   .usect  "my_sect",512,4
```

5.4.3 FUNC_EXT_CALLEDプラグマ

−pm オプションを指定した場合、コンパイラは、プログラム・レベルの最適化を実施します。この種の最適化を実施した場合、コンパイラは、main から直接または間接に呼び出されない関数を除去します。main の代わりに、手書きのアセンブリ・コードによって呼び出される C 関数が存在する場合もあります。

FUNC_EXT_CALLED プラグマはオプティマイザに対して、これらの C 関数、またはこれらの C 関数に呼び出される他の関数を保持しておくように指示します。これらの関数は、C へのエントリ・ポイントとして機能します。

このプラグマは、必ず、保持する関数の宣言や参照より前に置かなければなりません。このプラグマの構文は、次のとおりです。

#pragma FUNC_EXT_CALLED (*func***);**

引数 *func* は、除去したくない C 関数の名前です。

プログラム・レベルの最適化を使用するときは、FUNC_EXT_CALLED プラグマと一緒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 節「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。

5.5 asm 文

TMS320C2x/C2xx/C5x C コンパイラでは、TMS320C2x/C2xx/C5x 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 言語の拡張機能である *asm* 文によるものです。asm 文は、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と同様に、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しません。エラーがある場合は、アセンブラが検出します。アセンブリ言語文の詳細は、*TMS320C1x/C2x/C2xx/C5x* 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 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。この機能は、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です。

注: asm 文で C 環境を混乱させないための注意

asm 文で 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 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 文と一緒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。

5.6 グローバル・レジスタ変数の作成方法

TMS320C2x/C2xx/C5x コンパイラは C 言語を拡張し、グローバル・レジスタの割り当てができるように、register キーワードに特別な規則を追加します。この特別な場合には、register キーワードは記憶クラス修飾子として扱われます。宣言は、関数定義の前に指定しなければなりません。この特別な宣言の形式は、次のとおりです、

register *type* **AR6**
または
register *type* **AR7**

2 つのレジスタ R6 と R7 は、通常はエントリ時保存レジスタです。*type* には、float も long も指定できません。ファイル・レベルで割り当て宣言を行うと、レジスタは永久に予約され、そのファイルに対するオプティマイザとコード・ジェネレータによって使用されることはありません。レジスタに初期値を割り当てることはできません。#define 文を使用すると、レジスタに意味のある変数名を割り当て、その変数を通常通りに使用できます。たとえば、次のとおりです。

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

data_pointer->element;
data_pointer++;
```

5.6.1 グローバル・レジスタ変数を使用する場合

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いて、その変数をレジスタに永続的に割り当てると、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いて、そのルーチンが、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・レジスタ変数を割り当てる意味については、慎重に検討しなければなりません。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、コードの効率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て、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かについても、慎重に考慮しなければなりません。

5.6.2 レジスタ値の破壊の回避方法

グローバル・レジスタ変数に使用できる 2 つのレジスタは、通常はエントリ時保存レジスタなので、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。レジスタ値の破壊を避けるには、以下の規則に従う必要があります。

- ❏ すべてのライブラリを含めて、すべてのコードを再コンパイルしてレジスタを予約する場合を除いて、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。
- ❏ グローバル・レジスタ変数を変更する関数を、グローバル・レジスタを認識しない関数から呼び出すことはできません。関数へのポインタを引数として渡す場合は、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。
- ❏ グローバル・レジスタは、それを使用するモジュールの入り口で保存し、出口で復元する必要があります。
- ❏ longjmp( ) 関数はグローバル・レジスタ変数を、setjmp( ) 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、longjmp のソースを rts.src から取り出し、変更しなければなりません。

5.6.3 コンパイラによる AR6 と AR7 使用の抑止方法

–r*register* シェル・オプション、およびオプティマイザとコード・ジェネレータ用の対応する –g*register* オプション (ツールを個別に起動する場合) を使用すると、コンパイラは、指定された *register* を使用することができません。上記の問題を避けるために、グローバル・レジスタ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は、本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。

コンパイラが AR6 と AR7 を使用するのを完全に抑止できるので、割り込み関数の中で AR6 または AR7 (あるいはその両方) を保存せずに使用できます。コンパイラが AR6 と AR7 を使用できないようにする場合は、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しなければなりません。たとえば、次のコマンドは、AR6 と AR7 を使用しないように、rts25.lib ライブラリを再作成します。

```
dspmk –rAR6 –rAR7 –o –v25 rts.src –l rts25.lib
```

5.7 静的変数とグローバル変数の初期化方法

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、プログラムが実行し始める前に 0 に初期化されなければならないと定めています。この作業は、通常はプログラムのロード時に行われます。ロード処理はターゲット・アプリケーション・システム固有の環境によって大きく異なるので、コンパイラ自体は、変数を事前に初期化しません。この要件を満たすのは、アプリケーションです。

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、出力ファイルのサイズが大幅に増えるという望ましくない効果が発生する可能性があります。

5.7.1 const 型修飾子をもつ静的変数とグローバル変数の初期化

型修飾子 *const* をもつ静的変数とグローバル変数では、他の型の静的変数やグローバル変数とは異なる方法で処理されます。

明示的に初期化されていない *const* 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似ています (上記で説明されたのと同じ理由で)。たとえば、次のとおりです。

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

しかし、const 型のグローバル変数と静的変数の初期化は、.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なっています。たとえば、次のとおりです。

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

この指定は、.const セクション内のエントリと対応しています。

```
      .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。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を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。

5.7.2 入出力ポート・スペースへのアクセス方法

ioport キーワードを使用すると、TM320C2x/C2xx/C5x デバイスの入出力ポート・スペースにアクセスできます。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、または符号なし変数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* は 16 進数です。

ポート変数の宣言は、すべてファイル・レベルで行わなければなりません。関数レベルで宣言されたポート変数は、サポートされません。

たとえば、以下のコードは、入出力ポートを符号なしポート 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           */
    ...
}
```

ポート変数の使用は、代入に限定されていません。他の変数のように、式でポート変数を使用できます。たとえば、次のとおりです。

```
call(port10);    /* read port 10h and pass to call         */
a = port10 + b;  /* read port 10h, add b, assign to a      */
port10 += a;     /* read port 10h, add a, write to port 10h */
```

5.8 K&R C との互換性

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 C 標準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作成されたプログラムの大半は、修正なしで正しくコンパイルされ、実行されます。

ただし、C 言語には、既存のコードに影響を与える微妙な変更が行われています。*The C Programming Language* の第 2 版 (本書で K&R と呼ばれているもの) の付録 C には、ANSI C と初版の C 規格 (本書で K&R C と呼ばれているもの) との違いがまとめられています。

既存の C プログラムを、TMS320C2x/C2xx/C5x ANSI 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、コンパイラには K&R オプション (−pk) が付属しています。このオプションにより、言語の意味上の規則が一部変更され、古いコードとの互換性に対応しています。一般には、−pk オプションの目的は、K&R C よりも厳しくなっている ANSI 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 で、特に異なる点を以下に示します。

- 符号なし型をより広範囲な符号付き型に拡張することについて、整数拡張規則が変更になりました。K&R C では、結果の型は広い型の符号なしバージョンであり、ANSI では、結果の型は広い型の符号付きバージョンでした。これが、符号付きオペランドまたは符号なしオペランドに適用されるときに動作が異なる演算 (つまり、比較、除算 (および mod)、および右シフト) に影響を与えます。

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

- 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 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 このような状況の診断は行われますが、さらに緩和された条件の下で行われます。

  ```
  int *p;
  char *q = p; /* error without –pk, warning with –pk */
  ```

  −pk を指定しなくても、この規則違反は E クラス (回復可能) エラーとなります。−pk の代わりに、E クラス・エラーを警告に変える −pe を使用できます。

- 型や記憶クラスなし (識別子のみ) で外部宣言を行うことを ANSI では禁じていますが、K&R では認めています。

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

- ANSI 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、各定義が個別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ます (通常はエラーとなります)。たとえば、次のとおりです。

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

  ANSI では、この 2 つの宣言は、オブジェクト a の 1 つの定義になります。a が 2 回定義されるので、ほとんどの K&R コンパイラでは、このシーケンスは不正となります。

- ANSI 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, 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定義できます。たとえば、次のとおりです。

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

- K&R 構文では、列挙型定数リスト内にコンマを続けることができます。

```
enum { a, b, c, };  /* illegal unless -pk used */
```

- K&R 構文では、プリプロセッサ疑似命令の後にトークンを続けることができます。

```
#endif NAME         /* illegal unless -pk used */
```

5.9 コンパイラの制限値

TMS320C2x/C2xx/C5x 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制限事項があることにより、コンパイラは、サイズが大きすぎたり、構成が複雑すぎるソース・ファイルをコンパイルできない可能性があります。これらの条件の大部分は、パーサによって検出されます。パーサは、こ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 クラス診断メッセージを発行します。通常は、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がありますが、パーサほど多くはありません。

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るので、エラー・メッセージを出力直後にコンパイラは異常終了します。プログラムを単純化して、コンパイラの制限値を超えないようにしてください。

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・システムで使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ています。絶対制限値は、すべて ANSI C 規格の要求値と等しいか、それ以上です。

PC などのような小型のホスト・システムでは、オプティマイザがメモリを使い果たす可能性があります。この状態が発生すると、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続行します。この結果、ファイルは最適化されずにコンパイルされます。オプティマイザは一度に 1 つの関数をコンパイルするので、最も可能性が高い原因は、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です。これを防ぐには、以下のような方法があります。

- ❑ 疑わしいモジュールを最適化しない。
- ❑ 問題の原因となった関数を特定し、より小さい関数に分割する。
- ❑ 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	16K 文字	(注 1 を参照)
# または ## から作成された文字列の長さ	512 文字	(注 2 を参照)
−d を指定して事前定義されたマクロ	64	
マクロ・パラメータ	32 個	
マクロのネスト	32 レベル	(注 3 を参照)
#include 検索パス	64 パス	(注 4 を参照)
#include ファイルのネスト	64 レベル	
条件付き組み込み (#if) 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	
集合初期化のネスト	32 レベル	
静的初期化指定子	1 つの初期化当たり 1500 (概数)	
ローカル初期化指定子	150 レベル (概数)	
if 文、switch 文、およびループのネスト	1 つの初期化当たり 1500 (概数)	
グローバル・シンボル	2000 --- PC 10000 -- その他すべて (注 5 を参照)	
どのポイントからでも可視のブロック・スコープ・シンボル	500 .... PC 1000 ... その他すべて	

注: 1) この制限値は、\ で行を連結した後の文字数を表します。この制限値は、単一マクロ定義または起動にも適用されます。

2) この制限値は、連結前の文字数を表します。その他のすべての文字列には、制限はありません。

3) この制限値には、引数置換が含まれます。

4) この制限値には、−i と C_DIR ディレクトリが含まれます。

5) 使用可能なシステム・メモリによって、さらに制限される場合があります。

表 5−2.　コンパイラの絶対制限値 (続き)

解説	制限値
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

注: 1) この制限値は、\ で行を連結した後の文字数を表します。この制限値は、単一マクロ定義または起動にも適用されます。

2) この制限値は、連結前の文字数を表します。その他のすべての文字列には、制限はありません。

3) この制限値には、引数置換が含まれます。

4) この制限値には、−i と C_DIR ディレクトリが含まれます。

5) 使用可能なシステム・メモリによって、さらに制限される場合があります。

第 6 章

ランタイム (実行時) 環境

本章では、TMS320C2x/C2xx/C5x C ランタイム環境について説明します。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 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

TMS320C2x/C2xx/C5x C コンパイラは、プログラム・メモリとデータ・メモリの連続した 2 つのブロックとしてメモリを扱います。

- **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またはデータの各ブロックは、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・メモリに割り振るのは、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)、あるいは入出力 (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、再配置可能なコードを作成し、リンカでコードとデータを適切なメモリ空間に割り振ることができるようにします。

たとえば、リンカにより、グローバル変数を高速の内部 RAM に割り振ったり、実行可能コードを内部 ROM に割り振っ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振ることはできますが、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 ポインタ型でアクセスできます)。

6.1.1 セクション

コンパイラは、セクションと呼ばれる、コードとデータが入った再配置可能ブロックを生成します。これらのセクションは、さまざまなシステム構成に整合するように、さまざまな方法でメモリ内に割り振ることができます。COFF セクションの詳細は、*TMS320C1x/C2x/C2xx/C5x* アセンブリ言語ツール ユーザーズ・マニュアルの「COFF の概要」を参照してください。

TMS320C2x/C2xx/C5x コンパイラは、以下のセクションを作成します。

- **初期化されたセクション**には、データ・テーブルや実行可能コードを入れます。C コンパイラは、以下の 4 つの初期化されたセクションを作成します。
  - **.text セクション**は、すべての実行可能コードとともに、浮動小数点定数を含みます。
  - **.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。
  - **.const セクション**は、文字列リテラル、および明示的に初期化されるグローバル変数と静的変数 (*const* によって修飾される) の宣言と初期化を含みます。
  - **.switch セクション**は、switch 文用のテーブルを含みます。
- **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保存に使用できます。本コンパイラは、次の 4 つの初期化されないセクションを生成します。
  - **.bss セクション**は、グローバル変数と静的変数用の空間を確保します。−c リンカ・オプションを指定する場合は、プログラムの始動時に、C ブート・ルーチンが .cinit セクション (ROM に入っている場合がある) からデータをコピーし、それを .bss セクションに保存します。
  - **.stack セクション**は、C システム・スタック用のメモリを割り当てます。このメモリは、関数に引数を渡し、ローカル変数用の空間を割り当てます。
  - **.sysmem セクション**は、動的メモリ割り当て用の空間を確保します。この確保された空間は calloc、malloc、および realloc 関数によって使用されます。C プログラムでこれらの関数を使用しない場合、コンパイラは .sysmem セクションを作成しません。

アセンブラは、デフォルト・セクション .text、.bss、および .data を作成します。しかし、C コンパイラは .data セクションを使用しません。追加のセクションを作成するようにコンパイラに指示するには、CODE_SECTION および DATA_SECTION プラグマを使用してください (5-7 ページの 5.4.1 節「CODE_SECTION プラグマ」、および 5-8 ページの 5.4.2 節「DATA_SECTION プラグマ」を参照)。

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結合して出力セクションを作成します。完全なプログラムは、アセンブラの .data セクションを含むこれらの出力セクションで構成されます。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・スペース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、および .switch セクションは、通常 ROM または RAM にリンクされ、プログラム・メモリ (ページ 0) 内に存在し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) 内に存在しなければなりません。.bss セクション、.stack セクション、および .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 (ページ 1) 内に存在しなければなりません。次の表は、各セクション・タイプに必要なメモリの種類とページ指定を示しています。

セクション	メモリの種類	ページ
.text	ROM または RAM	0
.cinit	ROM または RAM	0
.switch	ROM または RAM	0
.const	ROM または RAM	1
.bss	RAM	1
.stack	RAM	1
.sysmem	RAM	1

セクションをメモリ内に割り振る方法の詳細は、*TMS320C1x/C2x/C2xx/C5x* 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。

6.1.2 C システム・スタック

C コンパイラは、ソフトウェア・スタックを使用して以下の作業を実行します。

- ローカル変数を割り当てます。
- 関数に引数を渡します。
- プロセッサのステータスを保管します。
- 関数の戻りアドレスを保管します。
- 一時的な結果を保存します。
- レジスタを保管します。

ランタイム・スタックは、上位のアドレスから下位のアドレスへと増大します。コンパイラは、次の 2 つの補助レジスタを使用してこのスタックを管理します。

AR1 **スタック・ポインタ** (SP)。スタックの現在の先頭または スタックの現在の先頭に続くワードを指します。

AR0 **フレーム・ポインタ** (FP)。現在のフレームの先頭を指します。各関数呼び出しで、スタックの先頭に新しいフレームが作成されます。ローカル変数と一時変数は、このフレームから割り当てられます。

C 環境では、これらのレジスタは自動的に操作されます。ランタイム・スタックを使用するアセンブリ言語ルーチンを書く場合は、必ずこれらのレジスタを正しく使用してください。これらのレジスタの使用方法の詳細は、6-9 ページの 6.2 節「レジスタ規則」を参照してください。スタックの詳細は、6-14 ページの 6.3 節「関数の呼び出し規則」を参照してください。

リンカは、スタック・サイズを設定し、グローバル・シンボル__STACK_SIZE を作成し、ワード単位のスタック・サイズをそれに割り当てます。デフォルトのスタック・サイズは、1K ワードです。リンカで −stack オプションを使用すると、リンク時にスタック・サイズを変更できます。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。

システムの初期化時に、SP は、スタックの最下部に指定されたアドレスに設定されます。このアドレスは、.stack セクションの直後の位置です。スタックの位置は、.stack セクションが割り当てられる位置に応じて異なるので、スタックの実際のアドレスはリンク時に決定されます。スタックをメモリ内の最後のセクション (最上位アドレス) として割り振る場合、スタックが拡大する空間には制限がありません (システム・メモリの制限値内)。

注: スタックのオーバーフロー

コンパイラは、コンパイル時または実行時に、スタックのオーバーフローをチェックする手段を備えていません。スタックのオーバーフローがあると、ランタイム環境が混乱し、プログラムに障害が生じます。必ず、スタックが拡大できる十分な空間を確保してください。

6.1.3 プログラム・メモリへの .const の割り当て

使用しているシステム構成で、.const などの初期化されたセクションをデータ・メモリに割り振ることができない場合は、プログラム・メモリでロードし、データ・メモリで実行するように .const セクションを割り振らなければなりません。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ています。

まず、以下のステップを使用してブート・ルーチンを変更します。

1) ソース・ライブラリから boot.asm を抽出します。

```
dspar –x rts.src boot.asm
```

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

```
CONST_COPY .set  1
```

3) boot.asm をアセンブルします。

```
dspa –v<target> 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。

```
dspar –r rts<target>.lib boot.obj
```

次に、以下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;
              }
   ...
}
```

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、これらの名前をこのように使用するリンカ・コマンド・ファイルによって異なります。名前を変更する場合は、同様に boot.asm を編集し、名前を変更しなければなりません。

6.1.4 動的メモリ割り当て

動的メモリ割り当ては、C 言語の標準な部分にはありませんが、TMS320C2x/C2xx/C5x コンパイラに付属のランタイムサポート・ライブラリには、実行時に変数にメモリを動的に割り当てることができるいくつかの関数 (malloc、calloc、realloc など) が含まれています。

メモリは、.sysmem セクションで定義されたグローバル・プール (ヒープ) から割り当てられます。.sysmem セクションのサイズは、−heap *size* オプションとリンカ・コマンドを使用して設定できます。リンカは、グローバル・シンボル __SYSMEM_SIZE も作成し、このシンボルにワード単位で表したヒープ・サイズを割り当てます。デフォルトのサイズは 0x400 ワードです。−heap 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。

動的に割り当てられるオブジェクトは、直接的にはアドレス指定されません (常にポインタを使用してアクセスされます)。また、メモリ・プールは別のセクション (.sysmem) に入っています。したがって、動的メモリ・プールのサイズを制限するものは、システム内の使用可能メモリの量だけです。.bss セクション内の空間を節約するには、グローバルまたは静的な配列を定義するのでなく、ヒープから大きな配列を割り当てます。たとえば、

```
struct big table[100];
```

という定義の代わりにポインタを使用し、次のような malloc 関数を呼び出すことができます。

```
struct big *table
table = (struct big *)malloc(100*sizeof (struct big));
```

6.1.5 変数の初期化

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 (グローバル変数と静的変数の初期化に使用される) は ROM に格納されます。システムの初期化時に、C ブート・ルーチンによって、これらのテーブルのデータ (ROM 内の) は .bss (RAM) 内の初期化された変数にコピーされます。

プログラムをオブジェクト・ファイルからメモリに直接ロードして実行するような場合は、メモリ内の空間が .cinit セクションで占有されるのを防ぐことができます。ローダは、実行時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、直接初期化を実施できます。–cr リンカ・オプションを使用して、これをリンカに指定できます。システムの初期化の詳細は、6-31 ページの 6.8 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 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、各変数の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。

C コンパイラは、.bss 内にグローバル変数用の空間を確保して、その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見なします。同じモジュールで宣言された変数は、連続した 1 つのメモリのブロックに割り当てられます。

6.1.7 フィールドと構造体の位置合わせ

コンパイラは、構造体に空間を割り当てる際に、すべての構造体のメンバを保持するのに必要な数のワードを割り当てます。構造体の配列では、各構造体はワード境界から始まります。

フィールド以外の型は、すべてワード境界に位置合わせされます。フィールドには、要求される数のビットが割り当てられます。隣接するフィールドは、1 つのワードの隣接するビットにパックされますが、複数のワードにまたがることはありません。フィールドが次のワードにまたがってしまう場合は、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、検出される順にパックされ、構造体ワードの最下位ビットが最初に埋め込まれます。

6.1.8 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。たとえば、次のとおりです。

  ```
  char s[] = "abc";
  ```

  初期化指定子として使用されると、文字列は単に初期化された配列として扱われます。個々の文字は、独立した初期化指定子となります。初期化の詳細は、6-31 ページの 6.8 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。たとえば、次のとおりです。

  ```
  strcpy (s, "abc");
  ```

  文字列を式の中で使用すると、文字列自体は、その文字列 (終了を示す 0 バイトも含む) を指す一意のラベルとともに、.const セクションに .string アセンブラ疑似命令で定義されます。たとえば、以下の行は文字列 abc、および終了バイトを定義します (ラベル SL5 は、その文字列を指します)。

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

  文字列ラベルの形式は **SL***n* です。*n* はコンパイラが割り当てる番号であり、これによりラベルは一意になります。この番号は、0 から順に 1 ずつ増分して各文字列を定義します。ソース・モジュールで使用する文字列の定義は、すべてコンパイラ出力のアセンブリ言語モジュールの最後に配置されます。

  ラベル SL*n* は、文字列定数のアドレスを表します。コンパイラは、このラベルにより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であっても、この文字列はメモリ内で重複しません。同一の文字列定数を使用する場合は、すべてその文字列の単一の定義を共用します。

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

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

6.2 レジスタ規則

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 プログラムにインターフェイスする場合は、これらのレジスタ規則を理解し、従ってください。

レジスタ規則は、コンパイラがレジスタをどのように使用するか、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レジスタとエントリ時に保存されるレジスタの 2 種類があります。これら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。エントリ時にレジスタ変数を保存するのは、呼び出し先関数の役割であり、呼び出し時にレジスタ変数を保存するのは、呼び出し元関数の役割です。

オプティマイザを使用するかどうか (−o オプション) によって、コンパイラがレジスタを使用する方法は異なります。オプティマイザは、レジスタ変数 (メモリではなく、レジスタに置かれるように定義される変数) 用に追加のレジスタを使用します。しかし、関数呼び出しの際のレジスタ保存規則は、オプティマイザを使用するかどうかにかかわらず、同一です。

次の表は、関数呼び出しの際に、コンパイラが TMS320C2x/C2xx/C5x のレジスタをどのように使用するか、およびどのレジスタを保存するかを示しています。

表 6−1. レジスタの用途と保存規則

(a) TMS320C2x、TMS320C2xx、および TMS320C5x の規則

レジスタ	用途	呼び出しによる保存
AR0	フレーム・ポインタ	Yes
AR1	スタック・ポインタ	Yes
AR2	ローカル変数ポインタ	No
AR2～AR5	式の解析	No
AR6～AR7	レジスタ変数	Yes
アキュムレータ	式の解析／戻り値	No
P	式の解析	No
T	式の解析	No

(b) TMS320C5x だけの規則

レジスタ	用途	呼び出しによる保存
INDX	AR0 のシャドウ	Yes
ACCB	式の解析	No
TREG1	式の解析	No
BRCR	ループ・カウンタ	No
PASR/PAER	ブロック・リピート・レジスタ	No

6.2.1 ステータス・レジスタ・フィールド

表 6−2 は、コンパイラが使用するすべてのステータス・フィールドを示しています。推定値とは、コンパイラが関数へのエントリまたは戻り時に、そのフィールド内に予想する値です。推定値の欄にあるダッシュは、コンパイラが特定の値を予想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、このフィールドが修正されるかどうかを示します。

表 6−2. ステータス・レジスタ・フィールド

(a) TMS320C2x、TMS320C2xx、および TMS320C5x 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	1	Yes
C	キャリー	–	Yes
DP	データ・ページ	–	Yes
OV	オーバーフロー	–	Yes
OVM	オーバーフロー・モード	0	No
PM	プロダクト・シフト・モード	0	No
SXM	符号拡張モード	–	Yes
TC	テスト制御ビット	–	Yes

(b) TMS320C5x だけのフィールド

フィールド	名前	推定値	修正
BRAF	ブロック・リピート・アクティブ・フラグ	–	Yes
NDX	インデックス・レジスタ・イネーブル・ビット	0	No
TRM	複数 TREG の有効化	0	No

6.2.2 スタック・ポインタ、フレーム・ポインタ、およびローカル変数ポインタ

コンパイラは、独自のソフトウェア・スタックを作成し、それを使用して、関数の戻りアドレスを保存したり、ローカル (自動) 変数を割り当てたり、関数に引数を渡したりします。呼び出しの深さ (同時にスタックに積まれる関数呼び出しの数) が 8 レベル以下であることがコンパイラにとって確実である場合を除き、関数の戻りアドレスの保存に内部ハードウェア・スタックが使用されることはありません。関数にローカル記憶域が必要になると、関数は、独自の作業空間 (ローカル・フレーム) をスタックから作成します。ローカル・フレームは、関数の入口シーケンスで割り当てられ、出口シーケンスで解放されます。

スタック・ポインタ (SP)、フレーム・ポインタ (FP)、およびローカル変数ポインタ (LVP) の 3 つのレジスタは、スタック・フレームとローカル・フレームを管理します。

AR1 レジスタは、スタック・ポインタ (SP) 専用です。コンパイラは、従来の方法で SP を使用します。つまり、スタックは上位アドレスに向かって増大し、SP はスタック上の次に使用可能なワードを指します。

AR0 レジスタは、フレーム・ポインタ (FP) 専用です。FP は、現在の関数のローカル・フレームの先頭を指します。ローカル・フレームの最初のワードは、FP が直接指すワードであり、レジスタ間の転送を可能にするための一時メモリ・ロケーションとして使用され、再入可能な C 関数を作成するには必須です。

AR2 レジスタは、ローカル変数ポインタ (LVP) 専用です。ローカル・フレームに保存されたすべてのオブジェクトは、引数を含めて、LVP を通して間接的に参照されます。フレーム上のオブジェクトにアクセスするための LVP の使用方法については、6-18 ページの 6.3.4 節「引数とローカル変数へのアクセス方法」を参照してください。

6.2.3 TMS320C5x の INDX レジスタ

TMS320C5x では、*0+ アドレッシング・モードにより、ARP (ステータス・レジスタ ST0 の補助レジスタ・ポインタ・フィールド) によって示される AR レジスタに、AR0 ではなく、INDX レジスタが加算されます。コンパイラは、PMST ステータス・レジスタの NDX ビットが 0 であるものと想定します。これは、AR0 レジスタの変更のシャドウが INDX に作られることを意味します。またこの場合には、INDX レジスタを、AR0 と同じように呼び出しを通して保存する必要があります。NDX = 0 でコードを実行する場合は、AR0 を保存すると INDX も保存されます。しかし、NDX ビットを 1 に変更するアセンブリ・ルーチンを書く場合は、AR0 と INDX の両方を明示的に保存しなければなりません。

6.2.4 レジスタ変数

レジスタ変数は、メモリではなく、レジスタに置かれるように定義されたローカル変数またはコンパイラ一時変数です。コンパイラがレジスタ変数用にレジスタを使用する方法は、オプティマイザを使用するかどうかによって異なります。

6.2.4.1 オプティマイザを使用しない場合のレジスタ変数

オプティマイザを使用しない場合、コンパイラは、register キーワードを指定して宣言された最大 2 つの変数にレジスタを割り当てます。変数は、引数リスト、または関数の最初のブロックで宣言しなければなりません。ネストされたブロック内のレジスタ宣言は、通常の変数として扱われます。

コンパイラは、これらのレジスタ変数に AR6 と AR7 を使用します。AR6 は最初の変数に割り当てられ、AR7 は 2 番目のレジスタ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、割り当てられたレジスタに入れられます。16 ビットの型 (char、short、int、およびポインタ) だけを、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には、レジスタ変数ごとに約 4 つの命令が必要になります。この機能を効率良く使用するには、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。

6.2.4.2 オプティマイザを使用する場合のレジスタ変数

オプティマイザを使用する場合、ユーザのレジスタ宣言はすべて無視されます。どの変数またはコンパイラ一時変数がレジスタに割り当てられるかについての決定は、オプティマイザが行います。オプティマイザは、アドレスではなく、変数を直接レジスタに割り当てます。オプティマイザは、AR5、AR6、および AR7 レジスタをレジスタ変数に割り当てることができます。AR5 は複数の関数呼び出しを通して保存されないので、複数の呼び出しで重複する変数に使用されます。

変数に使用するレジスタの用途は、オプティマイザを使用するかどうかによって異なるので、特定の変数に割り当てられる特定のレジスタに依存するコードを作成しないようにしてください。

注: グローバル・レジスタ変数としての AR6 と AR7の使用方法

−r オプションを指定してコンパイラが AR6 と AR7 を使用できないようにした場合、AR6 と AR7 は、レジスタ変数として使用できません。詳細は、5-11 ページの 5.6.3 節「コンパイラによる AR6 と AR7 使用の抑止方法」を参照してください。

6.2.5 式レジスタ

コンパイラは、式を評価し、一時的な結果を保存する際に、レジスタ変数に使用されていないレジスタを使用します。式レジスタの内容は、関数呼び出しの際に保存されません。呼び出しが発生すると一次記憶域に使用されるレジスタは、すべて関数が呼び出される前にローカル・フレームに保存されます。これにより、呼び出し先関数は式レジスタを保存し、復元する必要はありません。

6.2.6 戻り値

関数からスカラ型 (整数、ポインタ、または浮動小数点) の値が戻される場合、その値は、アキュムレータに入れられます。

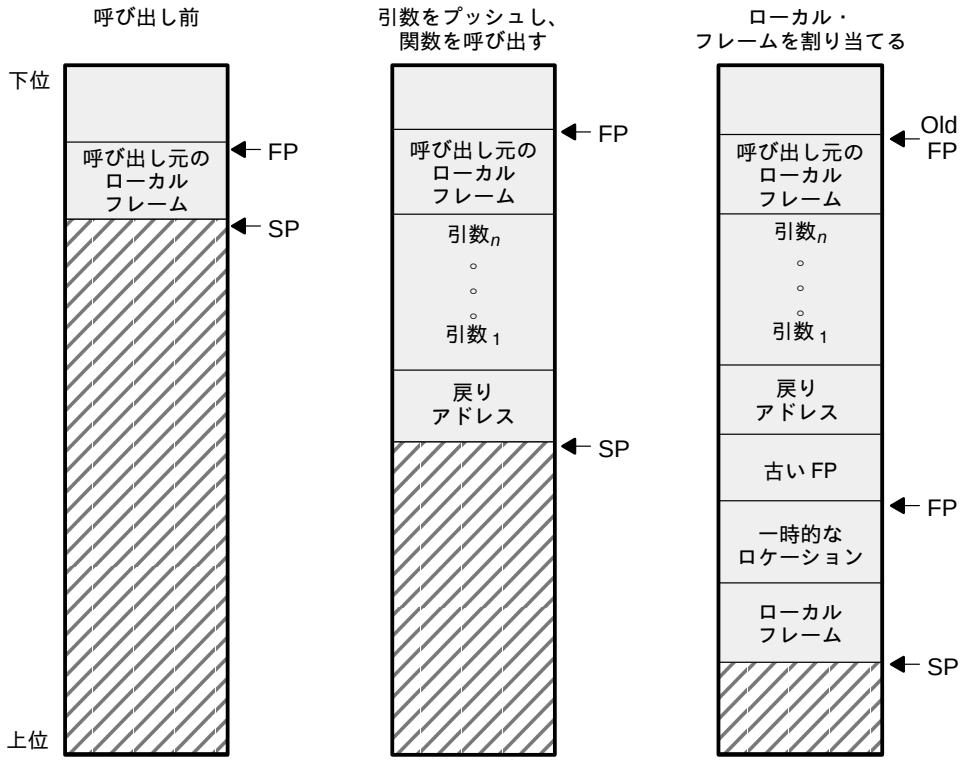
16 ビットの型 (char、short、int、またはポインタ) は、正しく符号拡張されてアキュムレータにロードされます。

6.3 関数の構造と呼び出し規則

C コンパイラでは、関数呼び出しに対して一連の厳格な規則を適用します。特別なランタイムサポート関数を除き、C 関数を呼び出すか C 関数によって呼び出されるすべての関数は、以下の規則に従わなければなりません。これらの規則に従わなかった場合は、C 環境が損なわれ、プログラムが異常終了する恐れがあります。

図 6−1 は、典型的な関数呼び出しを示しています。この例では、パラメータが関数に渡され、関数はローカル変数を使用します。またこの例は、呼び出し先関数に対するローカル・フレームの割り当ても示しています。ローカル変数をもたず、スタック上で渡される引数もない関数は、ローカル・フレームを割り当てません。

図 6−1. 関数呼び出し時のスタックの使用方法



6.3.1 関数の呼び出し方法

関数 (親関数) は、別の関数 (子関数) を呼び出すときに、以下の作業を実行します。必ず、ARP を AR1 に設定してください。

1) 呼び出し元は、引数を逆の順序でスタックに入れます (右端で宣言された引数が最初に入れられ、左端が最後に入れられます)。これにより、関数が呼び出されるときに、左端の引数がスタックの最上部に置かれます。

2) 呼び出し元 (親) が関数 (子) を呼び出します。

3) 呼び出し元は、関数からの戻り時に、ARP が AR1 に設定されるものと想定します。

4) 呼び出し先関数が完了すると、呼び出し元は、以下の命令で引数をスタックから取り去ります。

```
SBRK  n  (n は、入れられた引数ワード数です)
```

6.3.2 呼び出し先関数の対応方法

呼び出し先関数 (子関数) は、次の作業を実行します。関数のエントリでは、ARP は SP (AR1) に設定されるものと想定されます。

1) ハードウェア・スタックから戻りアドレスをポップし、それをソフトウェア・スタックにプッシュします。

2) FP をソフトウェア・スタックにプッシュします。

3) ローカル・フレームを割り当てます。

4) AR6 または AR7 が関数によって修正される場合は、スタックにプッシュします。その他のレジスタは、保存せずに修正できます。

5) 関数のコードを実行します。

6) 関数がスカラ値を戻す場合は、アキュムレータに入れます。16 ビットの整数およびポインタの戻り値を、正しい符号拡張でアキュムレータにロードします。

7) ARP を AR1 に設定します。

8) AR6 または AR7 (あるいは、その両方) が保存されていた場合は、それらを復元します。

9) ローカル・フレームの割り当てを解除します。

10) FP を復元します。

11) ソフトウェア・スタックから戻りアドレスをコピーし、それをハードウェア・スタックにプッシュします。

12) 戻ります。

例 6−1 は、4.3.2 節に掲載されている作業を実行する TMS320C2x コードの例です。

例 6−1.　呼び出し先関数としての TMS320C2x のコード

```
                    ; presume ARP = AR1 (SP)
POPD   *+           ; pop return address, push on software stack
SAR    AR0,*+       ; push AR0 (FP)
SAR    AR1,*        ; *SP = SP
LARK   AR0,SIZE     ; FP  = size of frame
LAR    AR0,*0+      ; FP  = SP, SP += size ==> allocate frame
SAR    AR6,*+       ; push AR6
SAR    AR7,*+       ; push AR7

...                 ; code for the function

MAR    *,AR1        ; set ARP = SP
MAR    *-           ; point to saved AR7
LAR    AR7,*-       ; pop AR7
LAR    AR6,*-       ; pop AR6
SBRK   SIZE+1       ; deallocate frame, point to saved FP
LAR    AR0,*-       ; pop FP
PSHD   *            ; push return address on hardware stack
RET                 ; return
```

6.3.3　呼び出し先関数の特殊な事例

呼び出し先関数には、次の 4 つの特殊な事例があります。

- 構造体を戻す場合
- 戻りアドレスをソフトウェア・スタックに移動しない場合
- フレームを割り当てない場合
- TMS320C5x の RETD 命令を使用する場合

これらの事例を、以下の各節で説明します。

6.3.3.1 構造体を戻す場合

関数が構造体を戻す場合、呼び出し元 (親関数) はその構造体用のスペースを割り当ててから、スタック上の追加の最終引数として戻りスペースのアドレスを呼び出し先関数 (子関数) に渡します。構造体を戻すには、呼び出し先関数は、その構造体をこの引数が指すメモリ・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、引数の値は 0 です。この 0 は、戻りの構造体をコピーしないように呼び出し先関数に指示します。

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正確に呼び出し先関数に知らせることができます。たとえば、s = f() 文で、s が構造体であり、f が構造体を戻す関数である場合、呼び出し元は実際に s のアドレスを最終引数として渡し、f を呼び出すことができます。すると、関数 f は戻りの構造体を s に直接コピーし、この代入を自動的に実行します。

構造体を正しく戻す関数を宣言するには、その関数が呼び出されるとき (呼び出し元が戻りスペースのアドレスを最終引数として渡すように) と、その関数を定義するとき (関数が結果をコピーすることを認識するように) のどちらにおいても注意が必要です。

6.3.3.2 戻りアドレスをソフトウェア・スタックに移動しない場合

この関数が他の関数を呼び出さない場合、または呼び出される関数が、呼び出しの深さが 8 以下であることをコンパイラが認識しているランタイムサポート関数のリストにあるものだけである場合、戻りアドレスをハードウェア・スタックからポップして、ソフトウェア・スタックにプッシュする必要はありません。6-15 ページの 6.3.2 節「呼び出し先関数の対応方法」のステップ 2 と 12 は省略されます。

6.3.3.3 ローカル・フレームを割り当てない場合

ローカル変数も引数もなく、AR0 が指す一時的なロケーションも使用せず、コードがデバッガのもとで実行するようにコンパイルされず、関数が構造体を戻さない場合は、ローカル・フレームを割り当てる必要はありません。6-15 ページの 6.3.2 節「呼び出し先関数の対応方法」のステップ 3、4、10、および 11 は省略されます。戻りアドレスがソフトウェア・スタックに保管されるときに、レジスタがスタックに保管されない場合、ステップ 10 は、保管された戻りアドレスを SP が指すように MAR *– で置き換えられます。

6.3.3.4 TMS320C5x の RETD 命令を使用する場合

デバッガは、コンパイラが上記のようにフレームを生成するものと見なします。しかし、デバッガのもとで実行されない 'C5x のコードを生成する場合、コンパイラは 6-15 ページの 6.3.2 節「呼び出し先関数の対応方法」のステップ 2 と 3、およびステップ 11 と 12 を切り替えます。これが生じるのは、PUSHD 命令は RETD の遅延スロット (遅延命令後の 2 ワード) に入ることができず、LAR AR0,* は入ることができるからです。上記の場合、最後の 3 つの命令は次のように変更されます。

```
PSHD   *-       ; push return address on hardware stack
RETD            ; return delayed
LAR    AR0,*    ; restore FP
NOP             ; fill delay slots of RETD
```

戻りアドレスがスタックに保存されない場合、つまり PSHD が生成されない場合、RETD は、関数の終わりから 2 ワード分移動されます。

6.3.4 引数とローカル変数へのアクセス方法

一般に、コンパイラが FP に対する変数のオフセットでロードすることにより LVP (AR2) を初期化して最初のローカル・アクセスを実行してから、MAR *0+ 命令が FP に追加されます。後続のアクセスは、LVP が指している現在のローカル変数のアドレスと次のアドレスとの差を、加算または減算することにより実行されます。LVP は呼び出しによって保存されないので、呼び出し後に初期化し直さなければなりません。

引数は常に FP からの負のオフセットであり、ローカル変数は常に FP からの正のオフセットです。

6.4 アセンブリ言語と C 言語間のインターフェイス

アセンブリ言語を 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 モジュールとリンクします (6.4.1 節「C コードでのアセンブリ言語モジュールの使用方法」を参照)。これは、最も応用範囲が広い方法です。
- ❑ インライン・アセンブリ言語を、直接 C ソース内に埋め込んで使用します (6-22 ページの 6.4.2 節「インライン・アセンブリ言語の使用方法」を参照)。
- ❑ アセンブリ言語の変数を、C ソースの中で使用します (6-23 ページの 6.4.3 節「アセンブリ言語変数に C からアクセスする方法」を参照)。
- ❑ コンパイラが生成したアセンブリ言語コードに修正を加えます (6-24 ページの 6.4.4 節「コンパイラ出力の修正方法」を参照)。

6.4.1 C コードでのアセンブリ言語モジュールの使用方法

6-14 ページの 6.3 節「関数の構造と呼び出し規則」で定義した呼び出し規則、および 6-9 ページの 6.2 節「レジスタ規則」で定義したレジスタ規則に従っている場合、アセンブリ言語関数と C 間のインターフェイスを取ることは難しくありません。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でき、関数を呼び出すことができます。アセンブリ・コードからも C の変数にアクセスしたり、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、以下の指針に従ってください。

- ❑ 関数が C で作成されているか、アセンブリ言語で作成されているかにかかわらず、すべての関数は、6-9 ページの 6.2 節「レジスタ規則」で概要が記載されているレジスタ規則に従わなければなりません。
- ❑ 関数によって変更される専用レジスタを保存しなければなりません。専用レジスタには、次のものがあります。

  AR0 (FP)
  AR1 (SP)
  AR6
  AR7
  INDX (TMS320C5x のみ)

  スタックを通常どおりに使用する場合は、SP を明示的に保存する必要はありません。つまり、スタックに入れたものを、関数が終了する前にすべてポップする限り、関数内でスタックを自由に使用できます。

  その他のすべてのレジスタは、内容を保存せずに自由に使用できます。

'C5x だけを使用するときに、アセンブリ・ルーチンでステータス・レジスタ PMST の NDX ビットを 0 から 1 に変更しない場合は、INDX レジスタは AR0 のシャドウとなり、AR0 を保存すると、INDX も保存されます。ルーチンで NDX ビットを変更する場合には、AR0 と INDX の両方を明示的に保存しなければなりません。

- 6-11 ページの表 6−2 に推定値が示されているステータス・レジスタ・フィールドのいずれかを変更する場合は、必ず値を復元する必要があります。特に、ARP が AR1 であることを確認してください。
- 割り込みルーチンは、使用するすべてのレジスタを保存しなければなりません (詳細は、6-25 ページの 6.5 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は、引数を逆の順序でスタックに入れてください。関数を呼び出した後で、引数を取り出します。
- C 関数を呼び出すときは、上記にリストされている専用レジスタだけが保存されることに注意してください。C 関数が、それ以外のレジスタの内容を変更する可能性があります。
- long と float は、最下位ワードが下位アドレスのメモリに格納されます。
- 関数は、アキュムレータに値を戻す必要があります。16 ビット整数値とポインタは、正しい符号拡張でアキュムレータにロードする必要があります。
- アセンブリ・モジュールは、グローバル変数の自動初期化以外の目的に .cinit セクションを使用するべきではありません。boot.asm 内の 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 (つまり、ラベル) の先頭にアンダースコア (_) を付けます。アセンブリ言語モジュールでは、C からアクセス可能なすべてのオブジェクトにアンダースコアの接頭部を付けなければなりません。たとえば、x という名前の C オブジェクトは、アセンブリ言語では _x となります。アセンブリ言語モジュールの中でのみ使用される識別子の場合は、アンダースコアで始まらない名前であれば、C の識別子と競合せずに使用できます。
- アセンブリ言語内で宣言され、C からアクセスまたは呼び出しが行われるオブジェクトや関数はすべて、アセンブラの中で .global 疑似命令を使用して宣言しなければなりません。これによって、シンボルが外部シンボルとして宣言され、リンカはそのシンボルへの参照を解決できます。

  同様に、アセンブリ言語から C 関数またはオブジェクトにアクセスする場合は、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宣言の外部参照が作成されます。

例 6−2 は、asmfunc アセンブリ言語関数を呼び出す、C の main 関数を示しています。asmfunc 関数は引数を 1 つ必要とし、それを C の gvar グローバル変数に加算し、その結果を戻します。

例 6−2. アセンブリ言語関数

(a) C プログラム

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

main()
{
   int i;

   i = asmfunc(i);    /* call function normally         */
}
```

(b) アセンブリ言語プログラム

```
_asmfunc:
   POPD      *+        ; Move return address to C stack
   SAR  AR0, *+        ; Save FP
   SAR  AR1, *         ; Save SP
   LARK AR0, 1         ; Size of frame
   LAR  AR0, *0+, AR2 ; Set up FP and SP

   LDPK _gvar          ; Point to gvar
   SSXM                ; Set sign extension
   LAC  _gvar          ; Load gvar
   LARK AR2, -3        ; Offset of argument
   MAR  *0+            ; Point to argument
   ADD  *, AR0         ; Add arg to gvar
   SACL _gvar          ; Save in gvar

   LARP AR1            ; Pop off frame
   SBRK 2
   LAR  AR0, *         ; Restore frame pointer
   PSHD *              ; Move return addr to C2x stack
   RET
```

例 6−2 の C プログラムでは、戻りの型が int であるため、asmfunc の外部宣言は省略してもかまいません。C 関数のように、非整数値を戻すか、非整数パラメータを渡す場合のみ、アセンブリ関数を宣言しなければなりません。

さらに、例 6−2 では、asmfunc は呼び出しを行わないので、戻りアドレスをハードウェア・スタックからソフトウェア・スタックに移動する必要はありません。このコードは、その方法を示すために例に含まれています。

6.4.2 インライン・アセンブリ言語の使用方法

C プログラムの中で *asm* 文を使用すると、コンパイラで作成されたアセンブリ言語ファイルの中に、アセンブリ言語の 1 行を挿入できます。asm 文を連続して使用すると、コンパイラ出力の中に他のコードを途中に入れることなく、アセンブリ言語の連続した行を挿入できます。

注: asm 文の使用方法

asm 文を使用すると、他の方法では C からアクセスできないハードウェアの機能にアクセスできます。asm 文を使用する際には、C 環境を損なわないように注意してください。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 コードの中にジャンプやラベルを挿入しないでください。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。

C 変数の値を変更しないでください。ただし、変数の現在の値は問題なく読み取ることができます。

asm 文を使用して、アセンブリ環境を変更するアセンブラ疑似命令を挿入しないでください。

また、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も便利です。次に示すように、単にアセンブリ・コードの文字列をアスタリスク (*) で始めるだけです。

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

6.4.3 アセンブリ言語変数に 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 プログラムからアクセスできると、便利な場合があります。.bss セクションまたは指定されたセクションに入っている、初期化されない変数にアクセスするのは簡単です。

1) .bss または .usect 疑似命令を使用して、変数を定義します。
2) .global 疑似命令を使用して、その定義を外部定義にします。
3) アセンブリ言語の中で、名前の前に下線を付けます。
4) C 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 6−3 は、.bss の中で定義された変数にアクセスする方法を示しています。

例 6−3. .bss で定義した変数に C からアクセスする方法

(a) C プログラム

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

(b) アセンブリ言語プログラム

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

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

変数を、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・テーブルの場合です。この場合、オブジェクトを指すポインタを定義し、C から間接的にそのオブジェクトにアクセスする必要があります。

最初にオブジェクトを定義します。オブジェクトを独自の初期化されたセクションに入れると便利ですが、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 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* として宣言する必要があります。この場合、前にアンダースコアを付けないでください。これで、オブジェクトに正常にアクセスできます。

例 6−4 は、.bss で定義されていない変数にアクセスする場合の例を示しています。

例 6−4.　.bss で定義されていない変数に C からアクセスする方法

(a) C プログラム

```
extern float sine[];   /* This is the object         */
f = sine[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
```

6.4.4　コンパイラ出力の修正方法

ソースをコンパイルしてから、アセンブルを実行する前に出力ファイルを編集すると、コンパイラが作成するアセンブリ言語出力を調べて変更できます。C インターリスト・ユーティリティは、コンパイラ出力を調べる際に役立ちます。インターリスト・ユーティリティについては、2-33 ページの 2.7 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。6.4.2 節に記載されている、C 環境の破壊についての警告は、コンパイラ出力の修正にも適用されます。

6.5 割り込み処理

この節のガイドラインに従うと、C 環境を損なわずに C コードに割り込み、C コードに戻ることができます。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が初期化されると、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は、必要となる割り込みの有効化やマスキングの処理は、ユーザが行わなければなりません。このような操作は、C 環境に影響を与えずに、簡単に asm 文で組み込んだり、アセンブリ言語関数を呼び出したりすることで実現できます。

6.5.1 割り込みの概要

- ❑ 割り込みルーチンは、グローバル変数へのアクセス、ローカル変数の割り当て、その他の関数の呼び出しを含めて、他の関数で実行されるすべての作業を実行します。
- ❑ 割り込みルーチンに入ると、ランタイムサポート関数 I$$SAVE が呼び出され、割り込まれた関数のコンテキスト全体が保存されます。すべてのレジスタが保存されます。割り込みルーチンから戻ると、ランタイムサポート関数 I$$REST が呼び出され、環境が復元され、割り込まれた関数に戻ります。
- ❑ 名前 c_int0 は、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元がないので、c_int0 はレジスタを保存しません。
- ❑ 割り込みルーチンと割り込みを関連付けるには、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。割り込みベクトル・テーブルの位置については、対象とするデバイスのユーザーズ・ガイドを参照してください。

6.5.2 C 割り込みルーチンの使用方法

割り込みは、以下の 2 つの規則のどちらかを使用すると、C 関数で直接処理できます。

- 名前 c_int*d* をもつ関数 (*d* は 0 ～ 9 の数字) は、すべて割り込みルーチンと見なされます。名前 c_int0 は、システム・リセットの割り込み用に予約されています。他の関数に、この名前を使用しないでください。たとえば、次のとおりです。

```
void c_int1()
{
  ...
}
```

- または、interrupt キーワードを使用できます。たとえば、次のとおりです。

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

上記の規則のどちらかを使用して、割り込みルーチンを定義します。コンパイラは、これらのルーチンの 1 つを検出すると、割り込みトラップから関数を起動できるコードを生成します。この方法では、標準 C シグナル・メカニズムよりも機能性が向上します。これにより、シグナル関数の実現が妨げられることはなく、これらの関数を完全に C で書くことができるようになります。

C 関数で割り込みを処理する際は、次の点に注意してください。

- 割り込みルーチンの型は void であり、引数なしで宣言する必要があります。
- コンパイラは、すべてのデバイス・レジスタを保存するわけではありません。コンパイラが保存するのは、表 6−1 に掲載されているレジスタだけです。
- IMR レジスタを介して、割り込みの特殊なマスキングを処理しなければなりません。インライン・アセンブリ言語を使用すると、C 環境を破壊せずに割り込みを有効または無効にし、IMR レジスタを修正できます。
- 割り込みルーチンは、通常の C コードで呼び出せますが、すべてのレジスタが保存されるので、効率が悪くなります。
- 1 つの割り込みルーチンで、1 つの割り込みを処理することも、複数の割り込みを処理することもできます。コンパイラは、特定の割り込みに固有のコードを生成しません。ただし、システム・リセット割り込みである c_int0 を除きます。
- 割り込みルーチンも、割り込みルーチンが呼び出す関数も、−oe シェル・オプション (−j オプティマイザ・オプション) を使用してコンパイルすることはできません。−oe オプションでは、モジュールの関数がどれも割り込みではなく、割り込みによって呼び出すことも、他の方法で非同期で実行することもできないものと想定されます。したがって、このオプションを指定して割り込みルーチンが入っているプログラムをコンパイルすると、使用できなくなります。

6.5.3 アセンブリ言語割り込みルーチンの使用方法

コンパイラと同じレジスタ規則に従うと、アセンブリ言語コードで割り込みを処理できます。次の点に注意してください。

- ❑ SP (AR1) が指すワードは、コンパイラで使用されている可能性があるので、保存する必要があります。
- ❑ 6-11 ページの表 6−1 のレジスタと表 6−2 のステータス・ビットを修正する場合、割り込みルーチンは、これらを保存しなければなりません。
- ❑ 割り込みルーチンは、C 関数を呼び出す場合、呼び出しによって保存されない、6-10 ページの表 6−1 内に掲載されているすべてのレジスタを保存しなければなりません。他のレジスタは、すべて C ルーチンによって修正できます。
- ❑ シンボル名の前には、必ずアンダースコアを付けてください。たとえば、c_int0 は _c_int0 にします。

6.5.4 TMS320C5x のシャドウ・レジスタ機能

TMS320C5x デバイスは、割り込みトラップがあると、1 組の内部シャドウ・レジスタに特定のレジスタを自動的に保存します。詳細は、*TMS320C5x* ユーザーズ・マニュアルを参照してください。割り込みがネストされない (つまり、この割り込みルーチン自身が割り込み可能なので、割り込みの再有効化を行わない) 場合、これらのレジスタを保存する最良の方法は、シャドウ・レジスタ機能を使用することです。

C で作成された割り込みがどれもネストされない場合、次のように、コンパイラがレジスタの保存に使用する I$$SAVE/I$$RESTORE ルーチンのソース内にあるアセンブル時フラグを修正すると、シャドウ・レジスタ機能を利用できます。

1) ソース・ライブラリからソースを取り出します。

```
dspar -x rts.src saverest.asm
```

2) ソース内の NEST フラグを 0 に変更します。

```
NEST   .set   0
```

3) 再アセンブルします。

```
dspa -v50 saverest.asm
```

4) 新しいオブジェクト・ファイルを、オブジェクト・ライブラリにアーカイブします。

```
dspar -r rts50.lib saverest.obj
```

6.6 整数式の解析

この節では、整数式を評価する際に注意しなければならない特別な考慮事項について説明します。

6.6.1 算術オーバーフローと算術アンダーフロー

TMS320C2x/C2xx/C5x では、データ・オペランドとして 16 ビット値が使用されている場合でも、32 ビットの結果が生成されます。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できません。コードが特定タイプのオーバーフローまたはアンダーフロー処理を使用している場合、そのコードが正しく実行される保証はありません。このようなコードは移植不能なので、一般的には使用しないことをお勧めします。

6.6.2 整数の除算と剰余

TMS320C2x/C2xx/C5x は整数の除算を直接サポートしていないので、除算と剰余演算は、すべてランタイムサポート・ルーチンの呼び出しにより実行されます。これらの関数は、演算の右側部分 (除数) をスタックに入れ、左側部分 (被除数) をアキュムレータの 16 LSB に入れます。この関数は、結果をアキュムレータに入れます。

6.6.3 long (32 ビット) 式の解析

C での long 式の解析演算は、標準 C 呼び出し規則に従っていない関数呼び出しで実行されます。これらの関数は、コンパイラと連携して速度を最大にし、コード空間を最小限に抑えます。これらの演算は、次のとおりです。

- ❑ 変数による左シフト
- ❑ 変数による右シフト
- ❑ 除算
- ❑ 剰余
- ❑ 乗算

6.6.4 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。たとえば、次のとおりです。

- 符号付きの結果

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

- 符号なしの結果

```
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関数呼び出しを行わずに、コンパイラによって実装されます。

6.7 浮動小数点式の解析

TMS320C2x/C2xx/C5x C コンパイラは、浮動小数点値を IEEE 単精度数として表します。単精度と倍精度の浮動小数点数は、どちらも 32 ビット値として表されます。この 2 つの形式には、違いはありません。

TMS320C2x/C2xx/C5x ランタイムサポート・ライブラリ rts.src には、以下の機能をサポートする浮動小数点算術関数のカスタム・コード・セットが入っています。

- ❑ 加算、減算、乗算、および除算
- ❑ 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
- ❑ 標準エラー処理

これらの関数は、標準 C 呼び出し規則に従っていません。その代わりに、コンパイラはランタイム・スタックに引数をプッシュし、浮動小数点関数の呼び出しを生成します。関数は、引数をポップし、演算を実行し、結果をスタックにプッシュします。

一部の浮動小数点関数は、int または long 引数を想定するか、int または long 値を戻します。浮動小数点関数の場合、int はすべてアキュムレータの 16 LSB で渡され、戻されます。long はすべてアキュムレータの全 32 ビットで渡され、戻されます。

6.8 システムの初期化

C プログラムを実行する前に、C ランタイム環境を構築する必要があります。この作業は、C ブート・ルーチンが、c_int0 と呼ばれる関数を使用して行います。このルーチンのソースは、ランタイムサポート・ソース・ライブラリ rts.src の boot.asm モジュール内に入っています。

システムの実行を開始するためには、c_int0 関数へ分岐するか、または呼び出します。しかし、この関数は通常、ハードウェアのリセットによって指定されます。c_int0 関数を、別のオブジェクト・モジュールにリンクしなければなりません。これを自動で行うには、−c または − cr リンカ・オプションを使用し、rts25.lib、rts2xx.lib、または rts50.lib をリンカ入力ファイルの 1 つとして組み込みます。

C プログラムをリンクした場合、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_int0 に設定します。しかしリンカは、リセット時に c_int00 に自動的に向けるようにハードウェアを設定しません。

c_int0 関数は、以下の作業を実行して環境を初期化します。

1) .stack と呼ばれるセクションをシステム・スタック用に定義し、初期のスタック・ポインタをセットアップします。

2) グローバル変数を初期化するため、.cinit セクションの初期化テーブルから、.bss セクション内の変数に割り当てられた記憶域にデータをコピーします。変数をロード時に初期化する場合 (−cr オプション) は、プログラムが実行される前に、ローダがこのステップを実行します (これはブート・ルーチンでは実行されません)。詳細は、6-32 ページの 6.8.2 節「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 プログラムを実行します。

ユーザのシステム要件に合わせて、ブート・ルーチンの置換や変更ができます。ただし、ブート・ルーチンでは C 環境を正しく初期化するため、上記の操作を必ず実行しなければなりません。

6.8.1 ランタイム・スタック

ランタイム・スタックは、メモリの連続した 1 つのブロック内で割り当てられ、下位のアドレスから上位のアドレスへと増大します。AR1 レジスタは通常、スタック内の次に使用可能なワード (スタックの最上部に +1 ワード) を指します。コンパイラは、このワードを一時的なメモリ・ロケーションとして使用できるので、割り込みルーチンは、このワードを保存しなければなりません。

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をチェックしません。スタック・オーバーフローが発生するのは、スタックが、割り当てられたメモリ空間の限界を超えて伸長した場合です。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・サイズをリンク時に変更するには、リンカ・コマンド行で −stack リンカ・オプションを使用し、このオプションの直後にスタック・サイズを定数として指定します。

6.8.2 変数の自動初期化

一部のグローバル変数には、C プログラムが実行を開始する前に、必ず初期値を割り当てなければなりません。これらの変数のデータを取り出し、そのデータを使用して変数を初期化するプロセスを、自動初期化と呼びます。

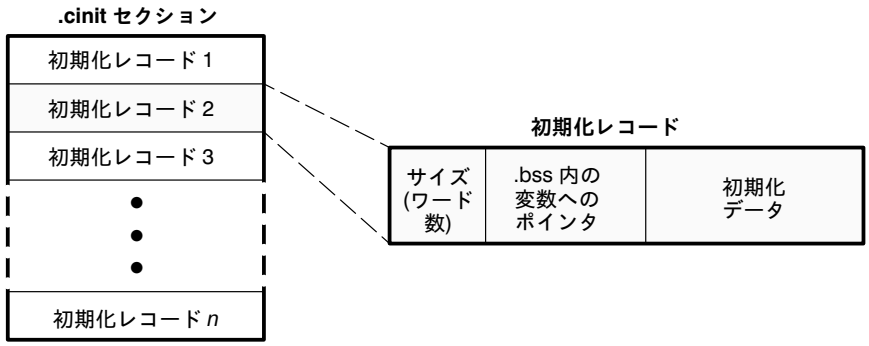
コンパイラは、.cinit と呼ばれる特殊なセクションに、グローバル変数と静的変数を初期化するためのデータが入ったテーブルを作成します。コンパイルされた各モジュールには、それらの初期化テーブルが含まれています。リンカは、それらのテーブルを 1 つのテーブル (1 つの .cinit セクション) にまとめます。ブート・ルーチンまたはローダは、このテーブルを使用して、すべてのシステム変数を初期化します。

グローバル変数は、実行時またはロード時のどちらかに自動初期化されます (6-34 ページの 6.8.4 節「実行時の変数の自動初期化」、および 6-35 ページの 6.8.5 節「ロード時の変数の初期化」を参照)。

6.8.3 初期化テーブル

.cinit セクションのテーブルは、可変サイズの初期化レコードで構成されます。自動初期化が必要な変数は、.cinit セクションにレコードをもっています。図 6−2 は、.cinit セクションの形式と初期化レコードを示しています。

図 6−2. .cinit セクション内の初期化レコードの形式



初期化レコードのフィールドには、次の情報が含まれています。

1) 初期化レコードの最初のフィールド (ワード 0) は、変数の初期化データのサイズ (ワード数) です。

2) 2 番目のフィールド (ワード 1) は、.bss セクション内の領域の先頭アドレスを示し、ここに初期化データをコピーします。

3) 3 番目のフィールド (ワード 2 ～ *n*) には、変数を初期化するために .bss セクションにコピーされるデータが入っています。

自動初期化が必要な各変数には、初期化レコードがあります。たとえば、2 つの初期化される変数が C で次のように定義されるものとします。

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

初期化テーブルは、次のように表示されます。

```
      .sect     ".cinit"    ; Initialization section
* 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 プログラムにインターフェイスする場合は、.cinit セクションを他の目的に使用しないでください。

−c や −cr リンカ・オプションを使用すると、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。

const で修飾された変数の初期化方法は異なります。5-12 ページの 5.7.1 節「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。

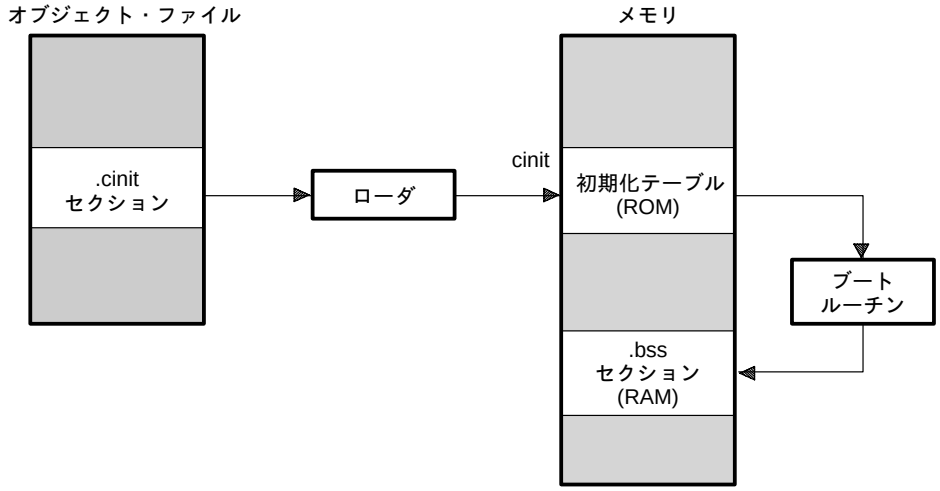
6.8.4 実行時の変数の自動初期化

実行時での変数の自動初期化は、自動初期化のデフォルト方式です。この方式を使用するには、−c オプションを指定してリンカを起動します。

この方式を使用すると、.cinit セクションは、その他の初期化されたすべてのセクションとともにメモリ内にロードされます。リンカは、cinit という特別なシンボルを定義し、このシンボルがメモリ内の初期化テーブルの先頭を指すようにします。プログラムの実行が開始されると、C ブート・ルーチンは、(.cinit が指す) テーブルのデータを .bss セクション内の指定された変数にコピーします。これにより、初期化データを ROM に格納し、プログラムが起動するたびに RAM にコピーできます。

図 6−3 は、実行時の自動初期化を示しています。ROM に組み込まれたコードからアプリケーションを実行するシステムでは、この方法を使用してください。

図 6−3. 実行時の自動初期化



6.8.5 ロード時の変数の初期化

ロード時の変数の初期化を使用すると、ブート時間が短くなり、初期化テーブルによって使用されるメモリも節約できるので、パフォーマンスが向上します。この方法を使用するには、−cr オプションを指定してリンカを起動します。

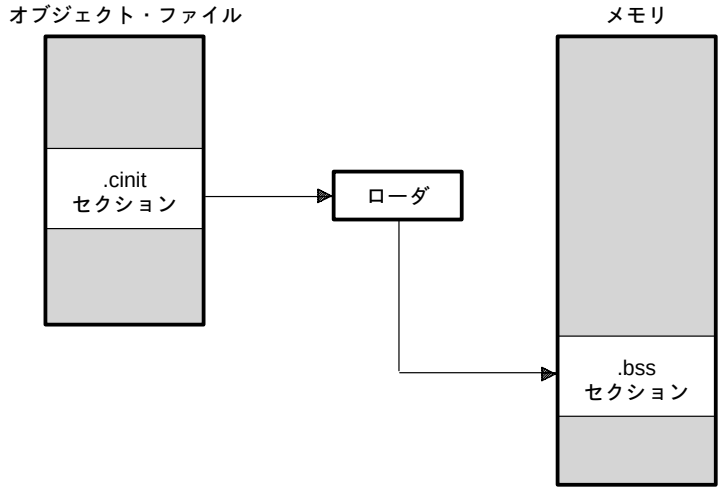
−cr リンカ・オプションを使用すると、リンカは .cinit セクションのヘッダ内に STYP_COPY ビットを設定します。これにより、.cinit セクションをメモリ内にロードしないことをローダに指示します (.cinit セクションは、メモリ・マップ内の空間を占有しません)。また、リンカは cinit シンボルを −1 に設定します (通常では、cinit は初期化テーブルの先頭を指します)。これにより、初期化テーブルがメモリ内に存在しないことをブート・ルーチンに示します。したがって、ブート時に実行時の初期化は行われません。

ロード時の初期化を使用するためには、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 セクションをメモリ内にコピーしないことを認識するように、.cinit セクション・ヘッダ内に STYP_COPY が設定されているかどうかを判別する。
- ❑ 初期化テーブルのフォーマットを理解する。

図 6−4 は、ロード時の変数の初期化を示しています。

図 6−4. ロード時の初期化



第 7 章

ランタイムサポート関数

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、三角関数など) の中には、C 言語自体の一部ではないものがあります。しかし ANSI C 標準は、これらの作業を実行する一連のランタイムサポート関数を定義します。TMS320C2x/C2xx/C5x C コンパイラは、特殊条件や地域別の項目 (居住地の言語、国名、または文化圏に応じて決まるプロパティ) を処理する機能を除き、ANSI 標準ライブラリに完全に準拠しています。ANSI 標準ライブラリを使用すると、関数は、C プログラムと整合がとれて移植性も高まります。

ANSI 指定の関数に加えて、TMS320C2x/C2xx/C5x ランタイムサポート・ライブラリには、プロセッサ固有のコマンドと直接 C 言語入出力要求を行うルーチンが組み込まれています。

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

TMS320C2x/C2xx/C5x 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。

- ❑ rts25.lib、rts2xx.lib および rts50.lib — ランタイムサポート・オブジェクト・ライブラリ。このライブラリには、シグナルと地域の項目を含む関数は入っていません。次のものが入っています。
  - ■ ANSI C 標準ライブラリ
  - ■ システム始動ルーチン、_c_int00
  - ■ C からの特定の命令へのアクセスを可能にする関数とマクロ
- ❑ rts.src — ランタイムサポート・ソース・ライブラリ。ランタイムサポート・オブジェクト・ライブラリは、rts.src ライブラリに入っている C ソースとアセンブリ・ソースから作成されます。

7.1.1 コードとオブジェクト・ライブラリとのリンク方法

プログラムをリンクするとき、リンカ入力ファイルの 1 つとしてオブジェクト・ライブラリを必ず指定することにより、入出力関数とランタイムサポート関数に対する参照が解決できるようになります。

ライブラリは、リンカ・コマンド行の最後に指定するべきです。リンカは、コマンド行でライブラリを検出すると、未解決参照を探すためにライブラリを検索するからです。−x リンカ・オプションを使用して、リンカが解決できる参照がなくなるまで、各ライブラリの検索を強制的に繰り返させることもできます。

ライブラリをリンクする際、リンカは、未定義の参照を解決するのに必要なライブラリ・メンバだけを組み込みます。リンクの詳細は、*TMS320C1x/C2x/C2xx/C5x* 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。

rts.src には、1 つのヘッダ・ファイル values.h があります。これは標準ヘッダではありませんが、これを使用すると、関数のカスタマイズが可能になります。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入っています。

7.1.2 ライブラリ関数の修正方法

アーカイバを使用してソース・ライブラリから適切なソース・ファイルを抽出することにより、ライブラリ関数を検査したり修正したりできます。たとえば、以下のコマンドでは、2 つのソース・ファイルを抽出できます。

```
dspar –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。コードに必要に応じて変更を加え、再コンパイルしてから、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。

```
dspar –r rts25.lib atoi.obj strcpy.obj
```

rts25.lib に作成し直すのではなく、この方法で新しいライブラリを作成することもできます。アーカイバの詳細は、*TMS320C1x/C2x/C2xx/C5x* アセンブリ言語ツール ユーザーズ・マニュアルを参照してください。

7.1.3 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 dspmk を使用すると、rts.src から新しいライブラリを作成できます。たとえば、以下のコマンドによって、最適化済みのコード・サイズの小さいランタイムサポート・ライブラリを作成できます。

```
dspmk --u –o2 –x –ms rts.src –l rtsf.lib
```

−−u オプションを指定すると、dspmk ユーティリティに対して、ヘッダ・ファイルをソース・アーカイブから抽出する代わりに、カレント・ディレクトリにあるヘッダ・ファイルを使用するように指示します。オプティマイザ (−o2) オプションとインライン関数展開 (−x) オプションを使用すると、これらのオプションを指定してコンパイルされたコードとの両立性に影響を与えません。−ms オプションは、コンパイラに対して、コード速度を無視し、可能な限り最短のコードを生成するように指示します。

ライブラリ作成ユーティリティの詳細は、第 8 章を参照してください。

7.2 ヘッダ・ファイル

各ランタイムサポート関数は、ヘッダ・ファイル内で宣言されます。各ヘッダ・ファイルで宣言する内容は、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

以下は、ランタイムサポート関数を宣言するヘッダ・ファイルを示しています。

```
assert.h     limits.h     stddef.h
ctype.h      math.h       stdlib.h
errno.h      setjmp.h     string.h
float.h      stdarg.h     time.h
ioports.h
```

ランタイムサポート関数を使用するには、まず #include プリプロセッサ疑似命令を使用して、その関数を宣言するヘッダ・ファイルを組み込まなければなりません。たとえば、isdigit 関数は、ctype.h ヘッダで宣言されています。isdigit 関数を使用するには、次のように、まず ctype.h を組み込む必要があります。

```
#include  <ctype.h>
   .
   .
   .
   val = isdigit(num);
```

ヘッダの組み込み順序に指定はありません。ただし、そのヘッダで宣言する関数やオブジェクトを参照する前にヘッダを組み込まなければなりません。

C コンパイラに組み込まれているヘッダ・ファイルについては、7-5 ページの 7.2.1 節「診断メッセージ (assert.h)」から、7-11 ページの 7.2.12 節「時間関数 (time.h)」までを参照してください。7-13 ページの 7.3 節「ランタイムサポート関数とマクロのまとめ」では、これらのヘッダが宣言する関数を掲載しています。

7.2.1 診断メッセージ (assert.h)

assert.h ヘッダは、assert マクロを定義します。これは、実行時に障害診断メッセージをプログラムに挿入するマクロです。assert マクロは、実行時に式をテストします。

- ❑ 式が真 (0 以外の値) の場合、プログラムは実行を継続します。
- ❑ 式が偽の場合、assert マクロは、その式、ソース・ファイル名、および式を含む文の行番号が入っているメッセージを出力します。その後、プログラムは (abort 関数により) 終了します。

assert.h ヘッダは、NDEBUG マクロを参照します (assert.h は NDEBUG の定義は行いません)。assert.h を組み込むときに NDEBUG をマクロ名としてすでに定義してある場合、assert は無効になり、何も実行されません。NDEBUG が定義されていない場合、assert は有効になります。

assert 関数は、7-14 ページの表 7−3 (a) に掲載されています。

7.2.2 文字の判別と変換(ctype.h)

ctype.h 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、文字をテストして、その文字が英字か、印字文字か、16 進数字かなどを判定します。これらの関数は、真 (0 以外の値) か偽 (0) を戻します。文字判別関数の名前の形式は、is*xxx* (*isdigit* など) です。

文字変換関数は、文字を小文字、大文字、あるいは ASCII に変換し、変換後の文字を戻します。文字変換関数の名前の形式は、to*xxx* (*toupper* など) です。

ctype.h 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。マクロは、同じ機能をもつ関数より処理速度が高速です。判別マクロは、フラグの配列 (この配列は ctype.c 内に定義されています) の中でルックアップ操作に展開されます。マクロの名前は、対応する関数と同じですが、各マクロの先頭には下線 (たとえば、*_isdigit*) が付きます。副次作用がある引数が渡される場合は、対応する関数を使用してください。

文字の判別と変換関数は、7-14 ページの表 7−3 (b) に掲載されています。

7.2.3 エラー報告(errno.h)

無効なパラメータ値が関数に渡された場合、または定義されている範囲外の結果を関数が戻した場合は、算術関数でエラーが起きる可能性があります。このような場合は、*errno* 変数が、以下のマクロのいずれかの値に設定されます。

- ❏ EDOM − 領域エラーの場合 (パラメータが不正)
- ❏ ERANGE − 範囲エラーの場合 (結果が不正)

算術関数を呼び出す C コードでは、errno の値を読み取ってエラーの状態を調べることができます。errno 変数は errno.h 内で宣言され、errno.c 内で定義されています。

7.2.4 制限値 (float.h と limits.h)

float.h と limits.h ヘッダは、TMS320C2x/C2xx/C5x の数値表現の有効な制限値やパラメータに展開するマクロを定義しています。表 7−1 と表 7−2 は、これらのマクロとその制限値を掲載しています。

表 7−1. 整数型の範囲に関する制限値を指定するマクロ (limits.h)

マクロ	値	解説
CHAR_BIT	16	ビット・フィールドでない最小オブジェクトの最大ビット数
SCHAR_MIN	−32768	signed char の最小値
SCHAR_MAX	32767	signed char の最大値
UCHAR_MAX	65535	unsigned char の最大値
CHAR_MIN	SCHAR_MIN	char の最小値
CHAR_MAX	SCHAR_MAX	char の最大値
SHRT_MIN	−32768	short int の最小値
SHRT_MAX	32767	short int の最大値
USHRT_MAX	65535	unsigned short int の最大値
INT_MIN	−32768	int の最小値
INT_MAX	32767	int の最大値
UINT_MAX	65535	unsigned int の最大値
LONG_MIN	−2147483648	long int の最小値
LONG_MAX	2147483647	long int の最大値
ULONG_MAX	4294967295	unsigned long int の最大値

表 7−2.　浮動小数点の範囲に関する制限値を指定するマクロ (float.h)

マクロ	値	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 (整数に丸め)
FLT_DIG DBL_DIG LDBL_DIG	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	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	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	FLT_RADIX の n 乗が表現可能な有限の float、double、または long double になるような最大の整数
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1.19209290E-07F	$1.0 + x \neq 1.0$ となる float、double、または long double の正の最小値 x
FLT_MIN DBL_MIN LDBL_MIN	1.17549435E-38F	float、double、または long double の正の最小値
FLT_MAX DBL_MAX LDBL_MAX	3.40282347E+38F	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	10 を整数乗にした値が正規化した float、double、または long double の範囲内に収まるような整数で、最小の負の整数
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	38	10 を整数乗にした値が有限の float、double、または long double の範囲内に収まるような整数で、最大の正の整数

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。

7.2.5 入出力 (I/O) ポート・マクロ (ioports.h)

ioports.h ヘッダは、TMS320C2x/C2xx/C5x の入出力 (I/O) ポートへのアクセスに使用される 2 つのマクロと、これに関連した関数を定義します。このマクロは、入出力 (I/O) ポートにアクセスする最も簡単な方法であり、次のものが含まれます。

- *inport* マクロ 。指定されたポートから値を読み取り、ポインタ *ret* を介してその値を戻します。
- *outport* マクロ 。指定されたポートに値を書き込みます。戻り値はありません。

関数には、次のものが含まれます。

- *_inport x* 関数。同じ番号のポートにアクセスし、その値を int として戻します。この関数の構文は、以下のとおりです。

  _inport *x* **( )**　　ここで、0 ≤ *x* ≤ 15 です。

- *_outport<x>* 関数。同じ番号のポートにアクセスします。戻り値はありません。この関数の構文は、以下のとおりです。

  _outport *x* **(***int value***)**　　ここで、0 ≤ *x* ≤ 15 です。

- *_in_port* 関数。指定されたポートから int を読み取り、値を戻します。この関数の構文は、以下のとおりです。

  _in_port (int port)

- *_out_port* 関数 。指定されたポートに出力を書き込みます。戻り値はありません。この関数の構文は、以下のとおりです。

  _out_port (int port, int value)

_PSWITCH の設定値により、inport マクロと outport マクロの実行にどの方法を使用するかが決まります。_PSWITCH が 0 (デフォルト) に設定される場合、正しい _inportx または _outportx 関数を選択する switch 文が使用されます。設定値 0 が効果があるのは、ポート番号が定数の場合です。これは、コンパイラが switch 文を、必要な関数への単純な呼び出しに最適化するからです。しかしポート番号が変数の場合は、switch 全体がユーザが作成した文のままです。したがって、ポート番号が変数の場合には、_PSWITCH を 1 に設定して、正しい _in_port または _out_port 関数を呼び出してください。_PSWITCH の値にかかわらず、このマクロは常に機能します。

inport および outport マクロは、7-14 ページの表 7−3(c) に掲載されています。

7.2.6 浮動小数点算術 (math.h)

math.h ヘッダは、三角関数、指数関数、ハイパボリック（双曲線）関数という算術関数を定義します。これらの関数は、7-15 ページの表 7−3 (d) に掲載されています。これらの算術関数では、倍精度浮動小数点引数を要求し、倍精度浮動小数点値を戻します。指定されている場合を除き、すべての三角関数は、ラジアン表記の角度を使用します。

math.h ヘッダは、HUGE_VAL という名前の 1 つのマクロも定義します。算術関数は、このマクロを使用して範囲外の値を表します。関数が浮動小数点値を戻すときに、その値が大きすぎて表現できない場合は、代わりに HUGE_VAL を戻します。

すべての math.h 関数の場合、領域と範囲のエラーは、必要に応じて errno を EDOM または ERANGE に設定して処理されます。関数の入出力値は、最も近い有効値に丸められます。

7.2.7 非ローカル・ジャンプ (setjmp.h)

setjmp.h ヘッダは、通常の関数の呼び出しと復帰に関する規律をバイパスするための型とマクロを定義し、関数を宣言します。次のものが含まれます。

- *jmpbuf* − 呼び出し環境の復元に必要な情報の保存に適した配列型
- *setjmp* − 後で longjmp 関数で使用するために、呼び出し環境を jmp_buf 引数に保管するマクロ
- *longjmp* − jmp_buf 引数を使用してプログラム環境を復元する関数

非ローカル・ジャンプのマクロと関数は、7-16 ページの表 7−3 (e) に掲載されています。

7.2.8 可変引数 (stdarg.h)

関数の中には、型が異なる可変数の引数をもつものがあります。このような関数を、可変引数関数と呼びます。stdarg.h ヘッダでは、可変引数関数の使用に役立つマクロと 1 つの型を宣言しています。

- マクロは、*va_start*、 *va_arg*、および *va_end* です。これらのマクロは、引数の数と型が関数の呼び出しごとに異なるときに使用します。
- 型 *va_list* は、*va_start*、*va_end*、および *va_arg* の情報を保持できるポインタ型です。

可変引数関数は、stdarg.h で宣言されたマクロを使用して、その引数リストを実行時に 1 つずつ処理できます。この時、関数は、実際に渡された引数の数と型を認識します。引数が正しく処理されるために、可変引数関数に対する呼び出しに、関数のプロトタイプに対する可視性があることを確認する必要があります。可変引数関数は、7-16 ページの表 7−3 (f) に掲載されています。

7.2.9 標準定義 (stddef.h)

stddef.h ヘッダは、型とマクロを定義します。型は、以下のとおりです。

- *ptrdiff_t* 型 − 2 つのポインタの減算結果のデータ型を示す signed int 型です。
- *size_t* 型 − *sizeof* 演算子のデータ型である unsigned int 型です。

マクロは、以下のとおりです。

- *NULL* マクロ − ヌル・ポインタ定数 (0) に展開されます。
- *offsetof (type, identifier)* マクロ − size_t 型の整数に展開されます。結果は、構造体 (*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。

7.2.10 汎用ユーティリティ (stdlib.h)

stdlib.h ヘッダは、1 つのマクロと 2 つの型を定義し、多くの関数を宣言します。マクロは *RAND_MAX* と呼ばれ、rand() 関数によって戻される最大の値を戻します。型は、以下のとおりです。

- *div_t* 型 − div 関数が戻す値の型である構造体型です。
- *ldiv_t* 型 − ldiv 関数が戻す値の型である構造体型です。

関数は、以下のとおりです。

- メモリ管理関数 − メモリのパケットの割り当てと割り当て解除を可能にします。これらの関数は、デフォルトで 1K ワードのメモリを使用できます。−heap オプションを指定してリンカを起動すると、リンク時にこの量を変更できます。
- 文字列変換関数 − 文字列を数値表現に変換します。

- ❑ 検索およびソート関数 – 配列の検索とソートを行います。
- ❑ シーケンス生成関数 – 疑似乱数シーケンスを生成し、シーケンスの開始点を選択できます。
- ❑ プログラム出口関数 – プログラムを正常終了または異常終了させます。
- ❑ 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、7-16 ページの表 7–3 (g) に掲載されています。

7.2.11 文字列関数 (string.h)

string.h ヘッダは、文字配列 (文字列) に関して、以下の作業を実行する標準関数を宣言します。

- ❑ 文字列の一部あるいは全体の移動やコピー
- ❑ 文字列の連結
- ❑ 文字列の比較
- ❑ 文字や他の文字列における文字列の検索
- ❑ 文字列の長さの検出

C では、すべての文字列の最後は 0 (ヌル) 文字です。str*xxx* という名の文字列関数は、すべてこの規則に従って処理を行います。string.h には別の関数も宣言されていて、これを使用すれば、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、7-17 ページの表 7–3 (h) に掲載されています。

7.2.12 時間関数 (time.h)

time.h ヘッダは、1 つのマクロと複数の型を定義し、日時を操作する関数を宣言します。この関数は、複数の型の時刻を扱います。

- ❑ 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です。
- ❑ 夏時間は、地方時の 1 つのバリエーションです。

time.h ヘッダは、1 つのマクロ *CLK_TCK* を宣言します。このマクロは、clock 関数が戻す値の 1 秒当たりの数値です。

型は、以下のとおりです。

- *clock_t* − 時刻を表す算術型
- *time_t* − 時刻を表す算術型
- *tm* − 詳細時刻と呼ばれるカレンダ時の構成要素を保持する構造体。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 (0–99)           */
int  tm_wday;    /* days since Saturday (0–6)         */
int  tm_yday;    /* days since January 1 (0–365)      */
int  tm_isdst;   /* Daylight Saving Time flag         */
```

  tm_isdst は、以下の 3 つの値のいずれかにすることができます。

  - 夏時間が有効である場合は、正の値
  - 夏時間が有効でない場合は、ゼロ
  - 情報が入手できない場合は、負の値

時間関数とマクロは、7-19 ページの表 7−3 (i) に掲載されています。

注: 時間関数のカスタマイズ

すべての時間関数は、clock 関数と time 関数によって決まります。これらの関数を、ご使用のシステムに合わせてカスタマイズしてください。

7.3 ランタイムサポート関数とマクロのまとめ

表 7−3 では、TMS320C2x/C2xx/C5x ANSI C コンパイラに付属のランタイムサポート・ヘッダ・ファイルを、アルファベット順にまとめています。説明されている関数の大部分は、ANSI 規格に従い、その規格内に定められているとおりに動作します。

表 7−3 に掲載されている関数とマクロの詳細は、7-20 ページの 7.4 節「ランタイムサポート関数とマクロの解説」を参照してください。関数またはマクロの詳細は、指示されているページを参照してください。

肩付きの数字は、指数を示すために以下の説明で使用されています。たとえば、x^y は、x の y 乗に相当します。

表 7−3.　ランタイムサポート関数とマクロのまとめ

(a) エラー・メッセージ・マクロ (assert.h)

マクロ	解説	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。	7-22

(b) 文字の判別と変換関数 (ctype.h)

関数	解説	ページ
int **isalnum**(int c);	ASCII の英数字かどうか、c をテストします。	7-34
int **isalpha**(int c);	ASCII の英字かどうか、c をテストします。	7-34
int **isascii**(int c);	ASCII 文字かどうか、c をテストします。	7-34
int **iscntrl**(int c);	制御文字かどうか、c をテストします。	7-34
int **isdigit**(int c);	数字かどうか、c をテストします。	7-34
int **isgraph**(int c);	スペース以外の印字文字かどうか、c をテストします。	7-34
int **islower**(int c);	ASCII の英小文字かどうか、c をテストします。	7-34
int **isprint**(int c);	印刷可能な ASCII 文字 (スペースを含む) かどうか、c をテストします。	7-34
int **ispunct**(int c);	ASCII の句読点文字かどうか、c をテストします。	7-34
int **isspace**(int c);	ASCII のスペース、タブ (水平または垂直)、復帰、改ページ、または改行文字かどうか、c をテストします。	7-34
int **isupper**(int c);	ASCII の英大文字かどうか、c をテストします。	7-34
int **isxdigit**(int c);	16 進数字かどうか、c をテストします。	7-34
char **toascii**(int c);	c を有効な ASCII 値にマスクします。	7-58
char **tolower**(int char c);	c が大文字の場合は、小文字に変換します。	7-59
char **toupper**(int char c);	c が小文字の場合は、大文字に変換します。	7-59

注:　−x オプションが使用されている場合、ctype.h 内の関数はインライン展開されます。

(c) I/O ポート・マクロ (ioports.h)

マクロ	解説	ページ
int **inport**(int port, int *ret);	ポインタ *ret* を介して、指定されたポートから値を戻します。	7-33
void **outport**(int port, int value);	指定されたポートに値を書き込みます。戻り値はありません。	7-33

(d) 浮動小数点算術関数 (math.h)

関数	解説	ページ
double **acos**(double x);	x のアーク・コサインを戻します。	7-21
double **asin**(double x);	x のアーク・サインを戻します。	7-21
double **atan**(double x);	x のアーク・タンジェントを戻します。	7-23
double **atan2**(double y, double x);	y/x のアーク・タンジェントを戻します。	7-23
double **ceil**(double x);	x 以上の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。	7-26
double **cos**(double x);	x のコサインを戻します。	7-27
double **cosh**(double x);	x のハイパボリック・コサインを戻します。	7-28
double **exp**(double x);	e^x を戻します。	7-30
double **fabs**(double x);	x の絶対値を戻します。	7-30
double **floor**(double x);	x 以下の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。	7-31
double **fmod**(double x, double y);	x/y の正確な浮動小数点数の剰余を戻します。	7-31
double **frexp**(double value, int *exp);	$.5 \leq \|f\| < 1$ であり、かつ値が $f \times 2^{exp}$ に等しくなるように f と exp を戻します。	7-32
double **ldexp**(double x, int exp);	$x \times 2^{exp}$ を戻します。	7-35
double **log**(double x);	x の自然対数を戻します。	7-36
double **log10**(double x);	x の底が 10 の対数を戻します。	7-36
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。	7-41
double **pow**(double x, double y);	x^y を戻します。	7-41
double **sin**(double x);	x のサインを戻します。	7-45
double **sinh**(double x);	x のハイパボリック・サインを戻します。	7-45
double **sqrt**(double x);	x の負でない平方根を戻します。	7-46
double **tan**(double x);	x のタンジェントを戻します。	7-56
double **tanh**(double x);	x のハイパボリック・タンジェントを戻します。	7-57

(e) 非ローカル・ジャンプ・マクロと関数(setjmp.h)

関数またはマクロ	解説	ページ
int **setjmp**(jmp_buf env);	longjmp が使用するために、呼び出し環境を保管します。これはマクロです。	7-44
void **longjmp**(jmp_buf env, int _val);	jmp_buf 引数を使用して、以前に保管された環境を復元します。	7-44

(f) 可変引数マクロ (stdarg.h)

マクロ	解説	ページ
type **va_arg**(va_list, type);	可変引数リスト内の型 type の次の引数にアクセスします。	7-59
void **va_end**(va_list);	va_arg を使用した後で、呼び出しメカニズムをリセットします。	7-59
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に ap を初期化します。	7-59

(g) 汎用関数 (stdlib.h)

関数	解説	ページ
void **abort**(void);	プログラムを異常終了させます。	7-20
int **abs**(int i);	値の絶対値を戻します。−x0 が使用される場合を除いて、インライン展開します。	7-20
int **atexit**(void (*fun)(void));	プログラム終了時に引数を指定せずに呼び出される、fun が指す関数を登録します。	7-23
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。	7-24
int **atoi**(register const char *st);	文字列を整数に変換します。	7-24
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。	7-24
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。	7-25
void ***calloc**(size_t num, size_t size);	size バイトごとに、num 個のオブジェクトのメモリの割り当てとクリアを行います。	7-26
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。	7-29
void **exit**(int status);	プログラムを正常終了させます。	7-30
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。	7-31

(g) 汎用関数 (stdlib.h)(続き)

関数	解説	ページ
long **labs**(long i);	i の絶対値を戻します。−x0 が使用されている場合を除いて、インライン展開します。	7-20
ldiv_t **ldiv**(register long numer, register long denom);	numer を denom で除算します。	7-29
int **ltoa**(long val, char *buffer);	val を等価の文字列に変換します。	7-36
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。	7-37
void **minit**(void);	malloc、calloc、または realloc によって以前に割り当てられたメモリすべてをリセットします。	7-39
void **qsort**(void *base, size_t nmemb, size_t size, int (*compar) ());	nmemb 個のメンバの配列をソートします。base は、ソートされていない配列の最初のメンバを指し、size は各メンバのサイズを指定します。	7-42
int **rand**(void);	0 ～ RAND_MAX の範囲の整数の疑似ランダム列を戻します。	7-43
void ***realloc**(void *packet, size_t size);	割り当てられたメモリ・スペースのサイズを変更します。	7-43
void **srand**(unsigned int seed);	乱数発生ルーチンをリセットします。	7-43
double **strtod**(const char *st, char **endptr);	文字列を浮動小数点値に変換します。	7-55
long **strtol**(const char *st, char **endptr, int base);	文字列を倍長整数に変換します。	7-55
unsigned long **strtoul**(const char *st, char **endptr**,** int base)**;**	文字列を符号なし倍長整数に変換します。	7-55

(h) 文字列関数 (string.h)

関数	解説	ページ
void ***memchr**(const void *cs, int c, size_t n);	cs の先頭の n 個の文字の中で、c の最初の出現を検出します。−x が使用されている場合は、インライン展開します。	7-37
int **memcmp**(const void *cs, const void *ct, size_t n);	cs の先頭の n 個の文字を ct と比較します。−x が使用されている場合は、インライン展開します。	7-38
void ***memcpy**(void *s1, const void *s2, register size_t n);	n 個の文字を s2 から s1 にコピーします。	7-38
void ***memmove**(void *s1, const void *s2, size_t n);	n 個の文字を s2 から s1 に移動します。	7-38
void ***memset**(void *mem, register int ch, register size_t length);	mem の先頭の length 個の文字に ch の値をコピーします。−x が使用されている場合は、インライン展開します。	7-39
char ***strcat**(char *string1, const char *string2);	string2 を string1 の末尾に付加します。	7-46
char ***strchr**(const char *string, int c);	s の中で文字 c の最初の出現を検出します。−x が使用されている場合は、インライン展開します。	7-47

(h) 文字列関数 (string.h)(続き)

関数	解説	ページ
int **strcmp**(register const char *string1, register const char *s2);	文字列を比較し、次の値のどれかを戻します。string1 が string2 より小さい場合は 0、string1 が string2 と等しい場合は 0、string1 が string2 より大きい場合は >0 です。−x が使用されている場合は、インライン展開します。	7-47
int **strcoll**(const char *string1, const char *string2);	文字列を比較し、次の値のどれかを戻します。string1 が string2 より小さい場合は <0、string1 が string2 と等しい場合は 0、string1 が string2 より大きい場合は >0 です。	7-47
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。	7-48
size_t **strcspn**(register const char *string, const char *chs);	全体が chs 内にない文字から構成される string の先頭部分の長さを戻します。	7-48
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。	7-49
size_t **strlen**(const char *string);	文字列の長さを戻します。	7-50
char ***strncat**(char *dest, const char *src, register size_t n);	最高 n 個の文字を src から dest に付加します。	7-50
int **strncmp**(const char *string1, const char *string2, size_t n);	2 つの文字列の最高 n 個の文字を比較します。−x が使用されている場合は、インライン展開します。	7-51
char ***strncpy**(register char *dest, register const char *src, register size_t n);	最高で n 個の文字を src から dest にコピーします。−x が使用されている場合は、インライン展開します。	7-52
char ***strpbrk**(const char *string, const char *chs);	string の中で、chs のいずれかの文字の最初の出現を検出します。	7-53
char ***strrchr**(const char *string, int c);	string の中で文字 c の最後の出現を検出します。−x が使用されている場合は、インライン展開します。	7-53
size_t **strspn**(register const char *string, const char *chs);	chs の文字だけで構成された string の先頭部分の長さを戻します。	7-54
char ***strstr**(register const char *string1, const char *string2);	string1 を検索して、string2 の最初の出現を検出します。	7-54
char ***strtok**(char *str1, const char *str2);	str1 を、str2 の文字で区切られる一連のトークンに分割します。	7-56

(i) 時間サポート関数(time.h)

関数	解説	ページ
char ***asctime**(const struct tm *timeptr);	時間を文字列に変換します。	7-21
clock_t **clock**(void);	使用されたプロセッサ時間を判定します。	7-27
char ***ctime**(const time_t *timer);	カレンダ時を地方時に変換します。	7-28
double **difftime**(time_t time1, time_t time0);	2 つのカレンダ時の差を戻します。	7-28
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。	7-32
struct tm ***localtime**(const time_t *timer);	time_t の値を詳細時刻に変換します。	7-35
time_t **mktime**(register struct tm *tptr);	詳細時刻を time_t の値に変換します。	7-40
size_t **strftime**(char *out, size_t maxsize, const char *format, const struct tm *time);	時間を文字列に形式設定します。	7-49
time_t **time**(time_t *timer);	現在のカレンダ時を戻します。	7-57

7.4 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説明します。肩付きの数字は、指数を示すために以下の説明で使用されています。たとえば、x^y は、x の y 乗に相当します。

abort 異常終了

構文

#include <stdlib.h>

void **abort**(void);

定義箇所

rts.src の exit.c

解説

abort 関数は、通常、エラー・コードをもってプログラムを終了させます。TMS320C2x/C2xx/C5x で abort 関数を実行すると、値 0 で exit 関数を呼び出し、次のように定義されます。

```
void abort ()
{
   exit(0);
}
```

この場合、abort 関数は exit 関数と等しくなります。

abs/labs 絶対値

構文

#include <stdlib.h>

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

定義箇所

rts.src の abs.c

解説

C コンパイラは、整数の絶対値を戻す以下の 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。

int と long int は、TMS320C2x/C2xx/C5x C では機能上は等しいので、abs 関数と labs 関数も機能上は同等です。abs 関数と labs 関数は、−x0 オプションが使用されていない場合、インライン展開されます。詳細は、2-27 ページの 2.6 節「インライン関数展開の使用方法」を参照してください。

例

```
int x = –5;
int y = abs (x);      /* abs returns 5 */
```

acos　アーク・コサイン

構文

#include <math.h>

double **acos**(double x);

定義箇所

rts.src の acos.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 = 0.0;
radians = acos(realval); /* acos return π/2 */
return (radians);
```

asctime　内部時間から文字列への変換

構文

#include <time.h>

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

定義箇所

rts.src の asctime.c

解説

asctime 関数は、詳細時刻を以下の形式の文字列に変換します。

```
Mon Jan 11 11:18:36 1988 \n\0
```

この関数は、変換された文字列へのポインタを戻します。

time.h ヘッダで宣言と定義が行われる関数と型の詳細は、7-11 ページの 7.2.12 節「時間関数」を参照してください。

asin　アーク・サイン

構文

#include <math.h>

double **asin**(double x);

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

構文

#include <assert.h>

void **assert**(int expr);

定義箇所

assert.h（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、assert はメッセージを発行して実行を打ち切るか、実行を継続します。このマクロは、デバッグのときに便利です。

- ❏ expr が偽の場合、assert マクロは、失敗した特定の呼び出しに関する情報を標準出力に書き込み、実行を打ち切ります。
- ❏ expr が真の場合、assert マクロは何も実行しません。

assert マクロを定義するヘッダ・ファイルは、別の NDEBUG マクロを参照します。assert.h ヘッダがソース・ファイルに組み込まれるときに NDEBUG をマクロ名として定義した場合、assert マクロは、無効であるように定義されます。

assert.h が組み込まれるときに NDEBUG が定義されない場合、assert マクロは、式をテストし、その結果が偽の場合はソース・ファイル名、行番号、式のテストを含む診断メッセージを書き込むように定義されます。

assert マクロは、printf 関数で定義されますが、ライブラリには含まれません。assert を使用するには、次のいずれかを実行しなければなりません。

- ❏ ユーザ独自のバージョンの printf を提供する
- ❏ 他の手段でメッセージを出力するように assert を修正する

例

この例では、整数 i を別の整数 j で割ります。0 による除算は不正な演算なので、この例では除算の前に assert マクロで j をテストしています。このコードを実行したときに j = = 0 の場合、assert はメッセージを発行し、プログラムを中止します。

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

atan 極アーク・タンジェント

構文

#include <math.h>

double **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 = 1.0;
radians = atan(realval);     /* return value = 0 */
```

atan2 デカルト・アーク・タンジェント

構文

#include <math.h>

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

定義箇所

rts.src の atan.c

解説

atan2 関数は、y/x のアーク・タンジェントを戻します。この関数は、これらの引数の符号を使用して、戻り値の座標象限を判定します。どちらの引数も 0 にすることはできません。戻り値は、範囲 [−π,π] ラジアンの角度です。

例

```
atan2 (1.0, 1.0)      /* returns π/4 */
atan2 (1.0, -1.0)     /* returns 3π/4 */
atan2 (-1.0, 1.0)     /* returns -π/4 */
atan2 (-1.0, -1.0)    /* returns -3π/4 */
```

atexit Exit () で呼び出される関数の登録

構文

#include <stdlib.h>

void **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。最高 32 個までの関数を登録できます。

exit 関数の呼び出し、abort の呼び出し、または main 関数からの戻りによってプログラムが終了すると、登録された関数は、登録順とは逆の順序で、引数なしで呼び出されます。

atof/atoi/atol　文字列から数値への変換

構文

#include <stdlib.h>

double **atof**(const char *st);
int **atoi**(const char *st);
long int **atol**(const char *st);

定義箇所

rts.src の atof.c と atoi.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、以下のとおりです。

  [*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、以下のとおりです。

  [*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、以下のとおりです。

  [*space*] [*sign*] *digits*

space は、スペース (文字)、水平タブか垂直タブ、復帰、書式送り、あるいは改行文字で表します。space の後ろ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。その後は、数値の小数部が続き、オプションの sign をもつ指数部が続きます。

文字列は、数値以外の文字が現れた時点で終わります。

int と long は、TMS320C2x/C2xx/C5x C では機能が等しいので、atoi 関数と atol 関数も機能上は同等です。

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。

例

```
int i;
double d;
i = atoi (“–3291”);   /* i = –3291 */
d = atof (“1.23e–2);  /* d = .0123 */
```

bsearch 配列検索

構文

#include <stdlib.h>

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

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なければなりません。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、以下のように宣言します。

```
int  cmp(const void *ptr1, const void *ptr2);
```

cmp 関数は、prt1 と ptr2 が指すオブジェクトを比較し、以下の値のいずれかを戻します。

< 0 : *ptr1 が *ptr2 より小さいとき
0 : *ptr1 が *ptr2 と等しいとき
> 0 : *ptr1 が *ptr2 より大きいとき

例

```
#include <stdlib.h>
#include <stdio.h>

int list [] = {1, 3, 4, 6, 8, 9};
int diff (const void *, const void *0;

main( )
{
    int key = 8;
    int p = bsearch (&key, list, 6, 1, idiff);
        /* p points to list[4] */
}
int idiff (const void *il, const void *i2)
{
    return *(int *) i1 – *(int *) i2;
}
```

calloc　メモリの割り当てとクリア

構文

#include <stdlib.h>

void ***calloc**(size_t num, size_t size);

定義箇所

rts.src の memory.c

解説

calloc 関数は、nmemb 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、memory.c にある .sysmem という初期化されない名前付きセクションで定義された特別なメモリ・プール (ヒープ) 内にあります。定数 __SYSTEM_SIZE は、ヒープを 1K ワードとして定義します。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 節「動的メモリ割り当て」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 (20,2);   /*Allocate and clear 20 bytes */
```

ceil　切り上げ

構文

#include <math.h>

double **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。−x2 オプションが使用される場合、ceil 関数はインライン展開されます。

例

```
double answer;
answer = ceil(3.1415);   /* answer = 4.0 */
answer = ceil(-3.5);     /* answer = -3.0 */
```

clock　プロセッサ時間

構文

#include <time.h>

clock_t **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判定します。プログラムの実行開始時以降の使用プロセッサ時間の概数値を戻します。戻り値をマクロ CLOCKS_PER_SEC の値で割ると、秒数に変換できます。

プロセッサ時間が利用不能、または表現できない場合は、clock 関数は、−1 の値を戻します。

> **注: ユーザ固有の clock 関数の記述方法**
>
> clock 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 clock 関数を記述する必要があります。また、clock( ) で戻る値 (クロックの目盛り数) 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。

time.h ヘッダで宣言と定義が行われる関数と型の詳細は、7-11 ページの 7.2.12 節「時間関数 (time.h)」を参照してください。

cos　コサイン

構文

#include <math.h>

double **cos**(double x);

定義箇所

rts.src の cos.c

解説

cos 関数は、浮動小数点数 x のコサインを戻します。角度 x は、ラジアンで表します。引数の値が大きすぎると、有意性のある結果がほとんど得られないか、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

構文

#include <math.h>

double **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 (errno は EDOM の値に設定されます)。

例

```
double x, y;

x = 0.0;
y = cosh(x); /* return value = 1.0 */
```

ctime　カレンダ時

構文

#include <time.h>

char ***ctime**(const time_t *timer);

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、以下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、7-11 ページの 7.2.12 節「時間関数 (time.h)」を参照してください。

difftime　時間差

構文

#include <time.h>

double **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。戻り値は秒数で表されます。

time.h ヘッダで宣言と定義が行われる関数と型の詳細は、7-11 ページの 7.2.12 節「時間関数 (time.h)」を参照してください。

div/ldiv 除算

構文

```
#include <stdlib.h>

div_t div(int numer, denom);
ldiv_t ldiv(long numer, 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を得ることができます。

- ❑ div 関数は、整数の除算を行います。入力する引数は整数です。この関数は、商と剰余を型 div_t の構造体で戻します。構造体は、以下のように定義します。

```
typedef struct
{
   int  quot;          /*  quotient    */
   int  rem;           /*  remainder   */
} div_t;
```

- ❑ ldiv 関数は、倍長整数の除算を行います。入力する引数は倍長整数です。この関数は、商と剰余を型 ldiv_t の構造体で戻します。構造体は、以下のように定義します。

```
typedef struct
{
   long int  quot;     /*  quotient    */
   long int  rem;      /*  remainder   */
} ldiv_t;
```

除算に剰余がある場合、その商の符号は代数計算の商と同じになります。商の最大値は、代数計算の商の値よりも小さい整数になります。剰余の符号は、numer (分子) の符号と同じになります。

int と long 型は TMS320C2x/C2xx/C5x C では等価なので、これらの関数も等価です。

例

```
int i = -10
int j = 3;
div_t result = div (i, j);      /* result.quot == -3 */
          /* result.rem  == -1 */
```

exit 正常終了

構文

#include <stdlib.h>

void **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 関数を修正して、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、呼び出し側には戻れないので注意してください。

TMS320C2x/C2xx/C5x では、abort 関数を実行すると、exit 関数と等しくなります。

exp 指数

構文

#include <math.h>

double **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 = 7.38905, which is e**2 */
```

fabs 絶対値

構文

#include <math.h>

double **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。−x0 オプションが使用されない場合、fabs 関数はインライン展開されます。

例

```
double x, y;

x = -57.5;
y = fabs(x);       /* return value = +57.5 */
```

floor 切り捨て

構文

#include <math.h>

double **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。−x オプションが使用される場合、floor 関数はインライン展開されます。

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

構文

#include <math.h>

double **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 */
```

free メモリの解放

構文

#include <stdlib.h>

void **free**(void *packet);

定義箇所

rts.src の memory.c

解説

free 関数は、malloc、calloc、または realloc の呼び出しにより割り当てられた (packet が指す) メモリ空間を解放します。これにより、メモリ空間を再び利用できます。割り当てられていない空間を解放しようとすると、関数は動作せずに戻ります。詳細は、6-6 ページの 6.1.4 節「動的メモリ割り当て」を参照してください。

例

この例では、10 バイトを割り当ててから解放します。

```
char *x;
x = malloc(10);       /*  allocate 10 bytes   */
free(x);              /*  free 10 bytes       */
```

frexp　小数部と指数部

構文

#include <math.h>

double **frexp**(double value, int *exp);

定義箇所

rts.src の frexp30.asm

解説

frexp 関数は、浮動小数点数を、正規化した小数部 (f) と 2 の累乗に分けます。この関数は、0.5−|f|<1.0 であり、value ==f × 2^{exp} となるような f と exp を戻します。frexp 関数は、exp が指す整数に累乗を保存します。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 */
```

gmtime　グリニッジ標準時

構文

#include <time.h>

struct tm ***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) を協定世界時 (詳細時刻として表示) に変換します。gmtime という名前には、グリニッジ標準時 (Greenwich Mean Time) を表す歴史的な意味があります。

time.h ヘッダで宣言と定義が行われる関数と型の詳細は、7-11 ページの 7.2.12 節「時間関数 (time.h)」を参照してください。

inport/outport　TMS320C2x/C2xx/C5x 入出力 (I/O) ポートとの間でのデータの送受信

構文

```
#include <ioports.h>

inport (int port, int *ret);
outport (int port, int value);
```

定義箇所

rts.src の ioports.asm

解説

TMS320C2x/C2xx/C5x 入出力 (I/O) ポートへのアクセスには、次のマクロが使用されます。

- inport マクロは、指定されたポートから値を読み取り、ポインタ ret を介してその値を戻します。
- outport マクロは、指定されたポートに値を書き込みます。戻り値はありません。

これらのルーチンは、関数ではなく、マクロとして実装されます。式の中では使用できません。次の例は、このマクロの誤った使用例です。

```
call (inport (1, &i));      /* Incorrect use of macro */
```

このマクロは、次のように使用しなければなりません。

```
inport (1, &i);
call (i);                   /* Correct use            */
```

ポート番号は 0 ～ 15 (0 と 15 を含む) でなければなりません。ポート番号にこれ以外の値を使用すると、未定義動作が生じます。

これらのマクロを、通常は定数ポート番号で使用する場合は、ioports.h に定義される定数 _PSWITCH を 0 (デフォルト) に設定してください。これらのマクロを、通常は変数ポート番号で使用する場合は、_PSWITCH を 1 に設定してください。_PSWITCH の値にかかわらず、このマクロは常に機能します。

入出力 (I/O) ポートの詳細は、*TMS320C2x User's Guide*、*TMS320C2xx User's Guide*、または *TMS320C5x* ユーザーズ・マニュアルを参照してください。

isxxx 文字の判別

構文

#include <ctype.h>

int **isalnum**(int c);
int **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);

定義箇所

rts.src の isxxx.c と ctype.c
ctype.h 内でもマクロとして定義

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。すべての文字判別関数は、−x オプションが使用されると、インライン展開されます。文字判別関数には、以下のような関数があります。

isalnum	英数字 ASCII 文字を認識します (isalpha や isdigit が真となるすべての文字をテストします)。
isalpha	英字 ASCII 文字を認識します (islower や isupper が真となるすべての文字をテストします)。
isascii	ASCII 文字 (0〜127 の範囲の文字) を認識します。
iscntrl	制御文字 (0〜31 の範囲と 127 の ASCII 文字) を認識します。
isdigit	数字 (0〜9) を認識します。
isgraph	空白以外の文字を認識します。
islower	英小文字の 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 のスペース、タブ (水平または垂直)、復帰、改ページ、または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 コンパイラは、これらの関数と同じ機能をもつ一連のマクロもサポートしています。これらのマクロの名前は、これらの関数と同じですが、先頭に下線が付いている点が異なります。たとえば、_isascii は、isascii 関数と同じ機能をもつマクロです。一般に、マクロは関数よりも処理速度が高速です。

labs

7-20 ページの abs/labs を参照。

ldexp

2 の累乗による乗算

構文

#include <math.h>

double **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 2^{exp} を掛け、$x \times 2^{exp}$ を戻します。指数部 (exp) は、負でも正でも指定できます。結果の値が大きすぎると、範囲エラーになる可能性があります。

例

```
double result;

result = ldexp(1.5, 5);       /* result is 48.0 */
result = ldexp(6.0, -3);      /* result is 0.75 */
```

ldiv

7-29 ページの div/ldiv を参照。

localtime

地方時

構文

#include <time.h>

struct tm ***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、7-11 ページの 7.2.12 節「時間関数 (time.h)」を参照してください。

log 自然対数

構文

#include <math.h>

double **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。

例

```
float x, y;

x = 2.718282;
y = log(x);       /* Return value = 1.0 */
```

log10 常用対数

構文

#include <math.h>

double **log10**(double x);

定義箇所

rts.src の log10.c

解説

log10 関数は、実数 x の底 10 の対数を戻します。x が負の場合は、領域エラーになります。x が 0 の場合は、範囲エラーになります。

例

```
float x, y;

x = 10.0;
y = log(x);       /* Return value = 1.0 */
```

longjump 7-44 ページの setjmp/longjmp を参照。

ltoa 倍長整数から ASCII への変換

構文

#include <stdlib.h>

int **ltoa**(long val, char *buffer);

定義箇所

rts.src の ltoa.c

解説

ltoa 関数は、倍長整数 val を等価な ASCII 文字列に変換して、buffer に書き込みます。入力した数値 val が負の場合、先頭に負符号が出力されます。ltoa 関数は、buffer に書き込まれた文字数を戻します。

例

```
int i;

char s[10];
i = ltoa (-92993L, s);    /* i = 6, s = "-92993"*/
```

malloc　メモリ割り当て

構文

#include <stdlib.h>

void ***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。この関数では、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 __SYSTEM_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 節「動的メモリ割り当て」を参照してください。

例

次の例では、構造体に空き空間を割り当てます。

```
struct xyz *p;
p = malloc (sizeof (struct xyz));
```

memchr　バイトの最初の出現の検出

構文

#include <string.h>

void ***memchr**(const void *es, int c, size_t n);

定義箇所

rts.src の memchr.c

解説

memchr 関数は、es 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。memchr 関数は、−x オプションが指定されると、インライン展開されます。

memcmp　メモリ比較

構文

#include <string.h>

int **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、以下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。memcmp 関数は、−x オプションが指定されると、インライン展開されます。

memcpy　メモリ・ブロックのコピー — オーバーラップ非対応

構文

#include <string.h>

void ***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 個の文字をコピーします。オーバーラップしたオブジェクトの文字をコピーする場合の、この関数の動作は予測できません。この関数は、 s1 の値を戻します。

memcpy 関数は strncpy に似ていますが、memcpy がコピーするオブジェクトに 0 を含めることができる点が異なります。memcpy 関数は、−x オプションが指定されると、インライン展開されます。

memmove　メモリ・ブロックのコピー — オーバーラップ対応

構文

#include <string.h>

void ***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

構文

#include <string.h>

void ***memset**(void *mem, int ch, size_t length);

定義箇所

rts.src の memset.c

解説

memset 関数は、mem が指すオブジェクトの先頭の length 文字に ch の値をコピーします。この関数は、mem の値を戻します。memset 関数は、−x オプションが指定されると、インライン展開されます。

minit　動的メモリ・プールのリセット

構文

#include <stdlib.h>

void **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 __SYSTEM_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 節「動的メモリ割り当て」を参照してください。

注:　以前に割り当てたオブジェクトは、minit の後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。以前に割り当てたオブジェクトは、すべてなくなります。そ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

構文

#include <time.h>

time_t ***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。timptr 引数は、詳細時刻を保持する構造体を指します。

この関数では、tm_wday と tm_yday の元の値は無視されます。また、構造体内に設定する値の範囲を制限しません。時間の変換が正常に完了すると、tm_wday と tm_yday は適切に設定され、構造体の他の構成要素には、制限範囲内の値が設定されます。tm_mday の最終値は、tm_mon と tm_year が決定されるまで設定されません。

戻り値は、型 time_t の値としてエンコードされます。カレンダ時を表現できない場合、この関数は、値 −1 を戻します。

time.h ヘッダで宣言と定義が行われる関数と型の詳細は、7-11 ページの 7.2.12 節「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 */

printf (”result is %s\n”, wday[time_str.tm_wday]);
```

modf 符号付き整数と小数

構文

#include <math.h>

double **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.           */
```

pow 累乗

構文

#include <math.h>

double **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;
z = pow(x, y);  /* return value = 8.0 */
```

qsort　配列のソート

構文

#include <stdlib.h>

void **qsort**(void *base, size_t nmemb, size_t size, int (*compar)
(const void *, const void *));

定義箇所

rts.src の qsort.c

解説

qsort 関数は、nmemb 個のメンバから構成される配列をソートします。引数 base は、ソートされていない配列の最初のメンバを指します。引数 size は、各メンバのサイズを示します。

この関数は、配列を昇順にソートします。

引数 compar は、キーを配列要素と比較する関数を指します。比較関数は、以下のように宣言します。

```
int  cmp(ptr1, *ptr2)
   void *ptr1, *ptr2;
```

cmp 関数は、ptr1 と ptr2 が指すオブジェクトを比較し、以下の値のいずれかを戻します。

< 0 : * ptr1 が *ptr2 より小さいとき
0 : * ptr1 が *ptr2 と等しいとき
> 0 : * ptr1 が *ptr2 より大きいとき

例

次の例では、qsort を指定して、整数の短いリストがソートされます。

```
#include  <stdlib.h>

int list[ ] = {3, 1, 4, 1, 5, 9, 2, 6};
int idiff (const void *, const void *);

main( )
{
   qsort (list, 8, 1, idiff);
   /* after sorting, list[ ]=={ 1, 1, 2, 3, 4, 5, 6, 9} */
}

int idiff (const void *i1, const void *i2)
{
   return *(int *)i1 – *(int *)i2;
```

rand/srand 乱整数

構文

```
#include <stdlib.h>
int rand(void);
void 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。TMS320C2x/C2xx/C5x C コンパイラの場合、RAND_MAX の値は 2147483646 ($2^{31}-2$) です。
- ❏ rand 関数に対する後続の呼び出しで新しい疑似乱数シーケンスが生成できるように、srand 関数はシード (種) の値を設定します。srand 関数は値を戻しません。

srand を呼び出す前に rand を呼び出すと、シードの値が 1 で srand を呼び出したときに生成される場合と同じシーケンスが rand で生成されます。同じシード値で srand を呼び出すと、rand は同じシーケンスの乱数を生成します。

realloc ヒープ・サイズの変更

構文

```
#include <stdlib.h>
void *realloc(void *packet, size_t size);
```

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 によってバイト単位で指定したサイズに変更します。メモリ空間の内容 (旧サイズと新規サイズのうちの小さいほうのサイズまで) 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、関数は処理をしないで戻ります。
- ❏ 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 のときに packet がヌルでない場合、realloc は、packet が指す空間を解放します。

より多くの空間を割り当てるために、オブジェクト全体を移動する必要がある場合、realloc は、新しい空間を指すポインタを戻します。この操作によ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 (ヒープ) 内にあります。定数 __SYSTEM_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 節「動的メモリ割り当て」を参照してください。

setjmp/longjmp 非ローカル・ジャンプ

構文

#include <setjmp.h>

int **setjmp**(jmp_buf env);
void **longjmp**(jmp_buf env, int returnval);

定義箇所

rts.src の setjmp.asm

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とマクロを定義し、関数を宣言します。

- ❏ jmp_buf 型は、呼び出し環境の復元に必要な情報の保存に適した配列型です。
- ❏ setjmp マクロは、後で longjmp 関数で使用できるように、呼び出し環境を jmp_buf 引数に保存します。

  直接の呼び出しからの戻りの場合、setjmp マクロは 0 を戻します。longjmp 関数の呼び出し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 が完了した後、対応する setjmp の呼び出しにより returnval が戻った場合と同様に、プログラムは引き続き実行されます。たとえ returnval が 0 であっても、longjmp 関数は、setjmp に値 0 を戻すことはありません。returnval が 0 の場合、setjmp マクロは値 1 を戻します。

例

これらの関数は一般的には、ネストの深い関数呼び出しから直ちに戻れるようにするために使用されます。

```
#include <setjmp.h>

jmp_buf env;

main()
{
   int errcode;

   if ((errcode = setjmp(env)) == 0)
      nest1();
   else
      switch (errcode)
   . . .
}
   . . .
nest42()
{
   if (input() == ERRCODE42)
   /* return to setjmp call in main */
      longjmp (env, ERRCODE42);
   . . .
}
```

sin　サイン

構文

#include <math.h>

double **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 radian, sval;           /* sval is re-
turned by sin */

radian = 3.1415927;
sval = sin(radian);   /* sin returns –1.0        */
```

sinh　ハイパボリック･サイン

構文

#include <math.h>

double **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

TMS320C2x/C2xx/C5x C コンパイラで提供されるランタイムサポート関数には、sprintf などの入出力 (I/O) 関数は含まれていません。しかし、時間関数で sprintf が使用されるので、time() で必要とされる書式化だけを実行する sprintf() の最小バージョンが提供されています。詳細は、7-58 ページの ti_sprintf の解説を参照してください。

sqrt 平方根

構文

#include <math.h>

double **sqrt**(double x);

定義箇所

rts.src の sqrt.c

解説

sqrt 関数は、実数 x の非負数の平方根を戻します。この引数が負の場合は、領域エラーになります。

例

```
double x, y;

x = 100.0;
y = sqrt(x);      /* return value = 10.0 */
```

srand

7-43 ページの rand/srand を参照。

strcat 文字列の連結

構文

#include <string.h>

char ***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、もとは string1 の終了文字であったヌル文字に上書きされます。この関数は、string1 の値を戻します。strcat 関数は、−x オプションが指定されると、インライン展開されます。string1 は、文字列全体を入れることができる大きさでなければなりません。

例

次の例では、*a、*b、*c が指す文字列は、コメントに示されている文字列を指すように割り当てられています。コメントの中の「\0」の表記は、ヌル文字を表します。

```
char *a, *b, *c;
.
.
.
/* a --> "The quick black fox\0"                              */
/* b --> "jumps over \0"                                      */
/* c --> "the lazy dog.\0"                                    */

strcat (a,b);
/* a --> "The quick black fox jumps over \0"                  */
strcat (a,c);
/* a --> "The quick black fox jumps over the lazy dog.\0"     */
```

strchr 文字の最初の出現の検出

構文

#include <string.h>

char ***strchr**(const char *string, char 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。strchr 関数は、−x オプションが指定されると、インライン展開されます。

例

```
char *a = ″When zz comes home, the search is on for z’s.″;
char *b;
char the_z = ’z’;
b = strchr(a,the_z);
```

この例では、b は zz の最初の z を指します。

strcmp/strcoll 文字列の比較

構文

#include <string.h>

int **strcoll**(const char *string1, const char *string2);
int **strcmp**(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。−x オプションを指定すると、strcmp 関数はインライン展開されます。

この関数は、以下の値のいずれかを戻します。

< 0 : *string1 が *string2 より小さいとき
0 : *string1 が *string2 と等しいとき
> 0 : *string1 が *string2 より大きいとき

例

```
char *stra = ″why ask why″;
char *strb = ″just do it″;
char *strc = ″why ask why″;
if (strcmp(stra, strb) > 0)
   {
      /* statements here will be executed      */
   }
if (strcoll(stra, strc) == 0)
   {
      /* statements here will be executed also */
   }
```

strcpy 文字列のコピー

構文

#include <string.h>

char ***strcpy**(char *string1, const char *string2);

定義箇所

rts.src の strcpy.c

解説

strcpy 関数は、string2 (終了ヌル文字を含む) を string1 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、string1 へのポインタを戻します。strcpy 関数は、−x オプションが指定されると、インライン展開されます。

例

次の例で、*a および *b が指す文字列は、2 つの独立した別々のメモリの位置です。コメントの中の「\0」の表記は、ヌル文字を表します。

```
char *a = ″The quick black fox″;
char *b = ″ jumps over ″;

/* a --> ″The quick black fox\0″        */
/* b --> ″ jumps over \0″               */

strcpy(a,b);

/* a --> ″ jumps over \0″               */
/* b --> ″ jumps over \0″               */
```

strcspn 不一致文字数の検出

構文

#include <string.h>

size_t **strcspn**(const char *string, const char *chs);

定義箇所

rts.src の strcspn.c

解説

strcspn 関数は、全体が string2 内にない文字から構成される string1 の最初の部分の長さを戻します。string1 の中の先頭の文字が string2 にある場合、この関数は 0 を戻します。

例

```
char *stra = ″who is there?″;
char *strb = ″abcdefghijklmnopqrstuvwxyz″;
char *strc = ″abcdefg″;
size_t length;

length = strcspn(stra,strb);   /* length = 0 */
length = strcspn(stra,strc);   /* length = 9 */
```

strerror 文字列エラー

構文

#include <string.h>

char ***strerror**(int errno);

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

構文

#include <time.h>

size_t ***strftime**(char *out, size_t maxsize, const char *format,
const struct tm *time);

定義箇所

rts.src の strftime.c

解説

strftime 関数は、format 文字列に基づいて、(time が指す) 時間を書式化し、書式化された結果を文字列 out に戻します。out には、最高 maxsize 個の文字を書き込むことができます。format パラメータは、strftime 関数に時間の書式化方法を指示するための文字列です。以下のリストは、有効な文字と、それぞれの展開内容を示したものです。

%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 (月のはじめからの累積)
%H 10 進数 (00～23) で表した時刻 (24 時間制)
%I 10 進数 (01～12) で表した時刻 (12 時間制)
%j 10 進数 (001～366) で表した日 (元日からの累積)
%m 10 進数 (01～12) で表した月
%M 10 進数 (00～59) で表した分 (時刻)
%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 (時刻)

%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、7-11 ページの 7.2.12 節を参照してください。

strlen 文字列の長さの検出

構文

#include <string.h>

size_t **strlen**(const char *string);

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。strlen 関数は、−x オプションが指定される場合、インライン展開されます。

例

```
char *stra = "who is there?";
char *strb = "abcdefghijklmnopqrstuvwxyz";
char *strc = "abcdefg";
size t length;

length = strlen(stra);      /* length = 13 */
length = strlen(strb);      /* length = 26 */
length = strlen(strc);      /* length = 7  */
```

strncat 文字列の連結

構文

#include <string.h>

char ***strncat**(char *dest, const char *src, size_t n);

定義箇所

rts.src の strncat.c

解説

strncat 関数は、src の最大 n 個の文字 (終了ヌル文字を含む) を dest に付加します。元の dest の終了文字のヌル文字は、src の先頭の文字上に上書きされます。 strncat 関数は結果にヌル文字を付けます。この関数は、dest の値を戻します。

例

次の例では、*a、*b、*c が指す文字列には、コメントに示されている値が割り当てられています。コメントの中の「\0」の表記は、ヌル文字を表します。

```
char *a, *b, *c;
size_t size = 13;
.
.
.
/* a--> ”I do not like them,\0”                              */;
/* b--> ” Sam I am, \0”                                      */;
/* c--> ”I do not like green eggs and ham\0”                 */;

strncat (a,b,size);

/* a--> ”I do not like them, Sam I am, \0”                   */;
/* b--> ” Sam I am, \0”                                      */;
/* c--> ”I do not like green eggs and ham\0”                 */;

strncat (a,c,size);

/* a--> ”I do not like them, Sam I am, I do not like\0”      */;
/* b--> ” Sam I am, \0”                                      */;
/* c--> ”I do not like green eggs and ham\0”                 */;
```

strncmp 文字列の比較

構文

#include <string.h>

int **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、string2 の最大 n 個の文字を string1 と比較します。この関数は、以下のいずれかの値を戻します。

< 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 (strncmp(stra, strb, size) > 0)
   {
      /* statements here will get executed      */
   }
if (strncmp(stra, strc, size) == 0)
   {
      /* statements here will get executed also */
   }
```

strncpy 文字列のコピー

構文

#include <string.h>

char ***strncpy**(const 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着目してください。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”;
size_t 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

構文

#include <string.h>

char ***strpbrk**(const char *string, const char *chs);

定義箇所

rts.src の strpbrk.c

解説

strpbrk 関数は、string の中で、chs のいずれかの文字が最初に現れる位置を検索します。一致する文字を検出すると、strpbrk はその文字を指すポインタを戻します。その文字が存在しない場合は、ヌル・ポインタ (0) を戻します。

例

```
char *stra = "it wasn't me";
char *strb = "wave";
char *a;

a = strpbrk (stra,strb);
```

この例の後では、a は wasn't の中の w を指します。

strrchr 文字の最後の出現の検出

構文

#include <string.h>

char ***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。strrchr 関数は、−x オプションが指定されると、インライン展開されます。

例

```
char *a = "When zz comes home, the search is on for z's";
char *b;
char the_z = 'z';
```

この例の後では、*b は、文字列の終わりに近い z を指します。

strspn 一致文字数の検索

構文

#include <string.h>

size_t ***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

構文

#include <string.h>

char ***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 wantfor nothing?″;
char *strb = ″what″;
char *ptr;

ptr = strstr(stra,strb);
```

ポインタ *ptr は、現在、最初の文字列の what の中の w を指します。

strtod/strtol/strtoul 文字列から数値への変換

構文

#include <stdlib.h>

double **strtod**(const char *test, char **endptr);
long int **strtol**(const char *test, char **endptr, int base);
unsigned long int **strtoul**(const char *test, char **endptr, int base);

定義箇所

rts.src の strtod.c、rts.src の strtol.c、および rts.src の 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 test は、元の文字列を指します。引数 endptr は、ポインタを指します。これらの関数は、変換された文字列のあとにある最初の文字を指すようにこのポインタを設定します。整数への変換を行う関数には、もう 1 つの引数、base もあります。この引数は、どの底で文字列を変換するかを関数に指示します。

- ❏ strtod 関数は、文字列を浮動小数点値に変換します。文字列の形式は以下のとおりです。

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

  この関数は、変換後の文字列を戻します。元の文字列が空のときや、その形式が正しくないときは、この関数は 0 を戻します。変換後の文字列がオーバーフローになると、この関数は、±HUGE_VAL を戻します。変換後の文字列がアンダーフローになると、この関数は 0 を戻します。変換後の文字列がオーバーフローやアンダーフローになると、errno が ERANGE の値に設定されます。

- ❏ strtol 関数は、文字列を倍長整数に変換します。文字列の形式は、以下のとおりです。

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

- ❏ strtoul 関数は、文字列を符号なし倍長整数に変換します。文字列の形式は、以下のとおりです。

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

space はスペース、水平タブか垂直タブ、復帰、書式送り、改行を組み合わせて示します。space の後は、オプションの sign、さらに数値の整数部を示す digits が続きます。その後は、数値の小数部が続き、オプションの sign をもつ指数部が続きます。

認識できない文字が初めて出現した位置で、文字列は終わります。endptr が指すポインタは、この文字を指すように設定されます。

strtok 文字列のトークンへの分割

構文

#include <string.h>

char ***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。strtok の最初の呼び出しでは、文字列 str1 を使用します。後続の呼び出しでは、最初の引数としてヌル・ポインタを使用します。str2 の値は、起動ごとに変更できます。str1 が strtok 関数ですでに変更されていることに注意してください。

例

以下の例の strtok の最初の呼び出しの後、ポインタ stra は文字列 excuse\0 を指します。これは、最初の空白のあった位置に strtok がヌル文字を挿入したからです。コメントの中の \0 の表記は、ヌル文字を表します。

```
char *stra = ”excuse me while I kiss the sky”;
char *ptr;

ptr = strtok (stra,” ”); /* ptr –> ”excuse\0” */
ptr = strtok (0,” ”);    /* ptr –> ”me\0”     */
ptr = strtok (0,” ”);    /* ptr –> ”while\0”  */
```

tan タンジェント

構文

#include <math.h>

double **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

構文

#include <math.h>

double **tanh**(double x);

定義箇所

rts.src の tanh.c

解説

tanh 関数は、浮動小数点引数 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

構文

#include <time.h>

time_t **time**(time_t *timer);

定義箇所

rts.src の time.c

解説

time 関数は、秒数で表される、1900 年 1 月 1 日午前 12 時以降の現在のカレンダ時を判定します。カレンダ時を使用できないときは、この関数は −1 を戻します。timer がヌル・ポインタでないならば、この関数は、timer が指すオブジェクトへの戻り値の代入も行います。

time.h ヘッダで宣言と定義が行われる関数と型の詳細は、7-11 ページの 7.2.12 節「時間関数 (time.h)」を参照してください。

注: ユーザ固有の time 関数の作成方法

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。

ti_sprintf　sprintf の特別バージョン

構文

#include <stdlib.h>

int ti_**sprintf** (char *s, const char *format, ...);

定義箇所

rts.src の tsprintf.c

解説

ti_sprintf 関数は、time() で必要とされる関数だけをサポートする sprintf() の最小バージョンです。具体的には、ti_sprintf は以下の変換だけをサポートします。

% [**0**] [*digits*] **(s** | **d)**

0　これが指定されている場合、フィールドがブランクではなく、0 で埋め込まれることを示します。

digits　これが指定される場合は、フィールドの最小幅を文字数単位で示します。変換に対応する引数がこの幅より小さい場合、引数はフィールド内で右寄せされ、フィールドには 0 またはブランク (上記のように 0 が使用されるかどうかに応じて) が埋め込まれます。

s | **d**　引数の型を指定します。**s** は、引数が char * 型であることを示し、**d** は、引数が int 型であることを示します。

ti_sprintf 関数を変更する場合には、rts.src からこの関数を抽出し、コードに変更を加え、ランタイムサポート・ライブラリを再コンパイルします。ti_sprintf 関数には、変更を容易にするために詳しいコメントが付いています。rts.src のライブラリから ti_sprintf.c を抽出するには、コマンド行に次のコマンドを入力します。

```
dspar –x rts.src tsprintf.c
```

toascii　ASCII への変換

構文

#include <ctype.h>

int **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 文字にすることできます。この関数には、同機能のマクロ呼び出し _toascii があります。

tolower/toupper 大文字と小文字の変換

構文

```
#include <ctype.h>

int tolower(int c);
int toupper(int c);
```

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。

例

```
tolower ('A')          /* returns 'a' */
tolower ('+')          /* returns '+' */
```

va_arg/va_end/va_start 可変引数マクロ／関数

構文

```
#include <stdarg.h>
typedef    char *va_list;
           va_arg(ap, type);
           void va_end(ap);
           void va_start(ap, parmN);
           va_list *ap
```

定義箇所

stdarg.h (マクロとして定義)

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、宣言された固定リスト内の右端のパラメータを指します。
- va_arg マクロは、可変引数関数に対する呼び出しで、次の引数の値を戻します。va_arg に対する連続呼び出しで可変引数関数の一連の引数を戻すことができるようにするため、va_arg を呼び出すたびに ap が変更されます (va_arg は、リスト内の次の引数を指すように ap を変更します)。type パラメータは、型の名前を示します。このパラメータは、リスト内の現在の引数の型を示します。

- ❑ va_end マクロは、va_start と va_arg の使用後に、スタック環境をリセットします。

va_arg や va_end を呼び出す前に、va_start を呼び出して ap を初期化しなければなりません。

例

```
int printf (char *fmt, ...)

{
   va_list ap;
   va_start(ap, fmt);
      .
      .
      .
/* Get next arg, an integer      */
   i = va_arg(ap, int);
/* Get next arg, a string        */
   s = va_arg(ap, char *);
/* Get next arg, a long          */
   l = va_arg(ap, long);
      .
      .
      .
   va_end(ap)       /* Reset     */
}
```

第 8 章

ライブラリ作成ユーティリティ

C コンパイラを使用すると、多くの構成や互いに互換性を維持する必要のないオプションでコードをコンパイルできます。個々のランタイムサポート・ライブラリにすべての可能な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・ファイル 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。

本章で説明している dspmk ユーティリティ、および *TMS320C1x/C2x/C2xx/C5x* アセンブリ言語ツール ユーザーズ・マニュアルで説明されているアーカイバを使用して、独自のランタイムサポート・ライブラリを作成できます。

項目 **ページ**

8.1 ライブラリ作成ユーティリティの起動方法

dspmk 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にシェル・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 に入れなければなりません。このユーティリティでは、環境変数 TMP、C_OPTION、C_DIR は無視され、無効になります。

ライブラリ作成ユーティリティを起動する構文は、以下のとおりです。

dspmk [*options*] *src_arch1* [**–l***obj.lib1*] [*src_arch2* [**–l***obj.lib2*]] ...

dspmk ユーティリティを起動するコマンドです。

options 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を制御します。このオプションは、コマンド行またはリンカ・コマンド・ファイルの任意の場所に指定できます (オプションについては、8.2 節と 8.3 節を参照してください)。

src_arch ソース・アーカイブ・ファイルの名前です。dspmk は、コマンド行オプションで指定されたランタイム・モデルに従って指定されたソース・アーカイブのオブジェクト・ライブラリを作成します。

–l*obj.lib* オプションのオブジェクト・ライブラリ名です。ライブラリ名が指定されていない場合、dspmk 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。

8.2 ライブラリ作成ユーティリティのオプション

コマンド行のオプションのほとんどは、コンパイラ、アセンブラ、リンカ、およびシェルが使用する同じ名前のオプションに直接対応しています。以下のオプションは、ライブラリ作成ユーティリティのみに適用します。

--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)。

--u オブジェクト・ライブラリの作成時に、ソース・アーカイブのヘッダ・ファイルを使用しません。必要なヘッダがすでにカレント・ディレクトリ内にある場合は、これらのヘッダ・ファイルを再インストールする必要はありません。このオプションを使用すると、各自のアプリケーションに合わせてランタイムサポート関数を自由に変更できます。

--v ユーティリティの実行時に、進捗情報を画面に表示します。通常は、ユーティリティの実行時にこ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、コンパイラで使用されるオプションに直接対応しています。表 8−1 は、これらのオプションのリストを示しています。

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
−g	シンボリック・デバッグを有効にします。
−r*register*	グローバル・レジスタを予約します。
−v*xx*	ターゲット・プロセッサ TMS320C*xx* (25, 50, 2xx) を指定します。

(b) パーサを制御するオプション

オプション	機能
−pk	コードを K&R 互換コードにします。
−pw	警告メッセージを抑止します。
−p?	3 文字符号の展開を有効にします。

(c) オプティマイザを制御するオプション

オプション	機能
−o0	最適化してコンパイルします (レジスタ最適化)。
−o1	最適化してコンパイルします (+ ローカル最適化)。
−o2 (または −o)	最適化してコンパイルします (+ グローバル最適化)。
−o3	最適化してコンパイルします (+ ファイル最適化)。dspmk が自動的に −ol0 と −op0 を設定することに注意してください。
−oe	割り込みによる呼び出しを想定しません。
−ox (−x2 と同じ)	_INLINE を定義し、上記のことを実行し、オプティマイザを (特に指定がなければ −o2 で) 起動します。

(d) 定義優先の制御によるインライン関数展開を制御するオプション

オプション	機能
−x1	組み込み関数のインライン展開を有効にします。
−x2 (または −x)	_INLINE を定義し、上記のことを実行し、オプティマイザを (特に指定がなければ −o2 で) 起動します。

(e) ランタイム・モデルを制御するオプション

オプション	機能
–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。
–mb	構造体の移動に対して RPTK の使用を回避します。
–ml	LDPK 最適化を抑止します。
–mn	–g で無効にされた最適化を有効にします。
–ms	速度ではなく、空間を最適化します。
–mx	'C5x シリコン・バグを回避します。

(f) 型チェックを無視するオプション

オプション	機能
–tf	プロトタイプ・チェック条件を緩和します。
–tp	ポインタ組み合わせチェック条件を緩和します。

(g) アセンブラを制御するオプション

オプション	機能
–ap	'C2x と 'C2xx または 'C5x へのポート切り換えを行います。
–app	'C2x と 'C2xx 間のポート切り換えを行い、.TMS32025 および .TMS3202XX を定義します。
–as	ラベルをシンボルとして保持します。

(h) 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
–ea[.]	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。
–eo[.]	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。

付録 A

用語集

数字

3 文字符号系列: 意味をもつ 3 つの文字系列 (ISO 646−1983-1983 不変コード・セットにより定義されている)。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 ^ に拡張されます。

A

ANSI: 米国規格協会を参照

B

.bss セクション: デフォルトの COFF セクションの 1 つ。.bss 疑似命令を使用すると、メモリ・マップに一定量の空間を確保でき、後でその空間にデータを格納できます。.bss セクションは初期化されません。

C

C オプティマイザ: 「オプティマイザ」を参照

C コンパイラ: C のソース文をアセンブリ言語のソース文に変換するプログラム

COFF: 「共通オブジェクト・ファイル・フォーマット」を参照

D

.data セクション: デフォルトの COFF セクションの 1 つ。.data セクションは、初期化されたデータを含む初期化されたセクションです。.data 疑似命令を使用すると、コードを .data セクションにアセンブルできます。

H

Hex 変換ユーティリティ: COFF オブジェクト・ファイルを、EPROM プログラマに適切にロードできる標準 ASCII 16 進数フォーマットの 1 つに変換するプログラム

K

K&R C: カーニハンとリッチーの C。*The C Programming Language* (K&R) の初版で定義された、事実上の標準規格。以前の ANSI に対応しない C コンパイラ用に記述された K&R C のプログラムの大部分は、修正なしで正しくコンパイルされ、実行されます。

T

.text セクション: デフォルトの COFF セクションの 1 つ。.text セクションは、実行可能なコードを含んだ初期化されたセクションです。.text 疑似命令を使用すると、コードを .text セクションにアセンブルできます。

あ

アーカイバ: 複数のファイルをアーカイブ・ライブラリと呼ばれる単一のファイルにグループ化するためのソフトウェア・プログラム。アーカイバを使用すると、アーカイブ・ライブラリのメンバを追加、削除、抽出、または置換することができます。

アーカイブ・ライブラリ: アーカイバによって単一のファイル内にグループ化されたファイルの集合

アセンブラ: アセンブリ言語命令、疑似命令、およびマクロ疑似命令を含むソース・ファイルから、機械語のプログラムを作成するソフトウェア・プログラム。アセンブラは、シンボリックな命令コードを絶対的な命令コードに置換し、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置換します。

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されるアセンブリ言語出力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: 単一のオブジェクトに複数の方法でアクセスできる場合、たとえば 2 つのポインタが 1 つの名前付きオブジェクトを指す場合などに行われます。エイリアス指定を実行すると、間接参照によって別のオブジェクトが参照される可能性があるので、最適化が正しく行われない場合があります。

エミュレータ: TMS320C2x、TMS320C2xx、または TMS320C5x の操作をエミュレートするハードウェア開発システム

エントリ・ポイント: ターゲット・メモリでの実行開始位置

お

オブジェクト・ファイル: 機械語オブジェクト・コードを含む、アセンブルまたはリンクされたファイル

オブジェクト・ライブラリ: 複数のオブジェクト・ファイルで構成されたアーカイブ・ライブラリ

オプション: ソフトウェア・ツールの起動時に、追加の機能や特定の機能を実行させるために使用するコマンド・パラメータ

オプティマイザ: C プログラムの実行速度を向上させ、サイズを縮小するソフトウェア・ツール

オペランド: アセンブリ言語命令、アセンブラ疑似命令、またはマクロ疑似命令の引数。これにより、命令または疑似命令で実行される操作に対して情報が提供されます。

か

外部シンボル: 現行のプログラム・モジュールで使用されるが、その定義が異なるプログラム・モジュールで行われるシンボル

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位置に挿入するプロセス。これにより、関数呼び出しのオーバーヘッドが軽減され、オプティマイザは周囲のコードとのコンテキストで関数の最適化を実行できます。

間接呼び出し: 1 つの関数が別の関数を呼び出す関数呼び出し。呼び出し先の関数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: 特別な目的をもつ複数のコマンド。ソフトウェア・ツールの動作および機能を制御します (デバイスの動作を制御する、アセンブリ言語命令とは反対の意味をもちます)。

共通オブジェクト・ファイル・フォーマット (COFF): AT&T の開発した標準に従って構成されるオブジェクト・ファイル・システム。これらのファイルは、メモリ空間内で再配置可能です。

く

クロスリファレンス・リスト: アセンブラにより作成される出力ファイル。定義されたシンボル、シンボルを定義した行、シンボルを参照する行、およびその最終値が表示されます。

グローバル・シンボル: 次のいずれかの条件を満たすシンボルの一種です。1) 現行のモジュールで定義されていて、他のモジュールでアクセスされている。または、2) 現行のモジュールでアクセスされているが、他のモジュールで定義されている。

こ

構造体: グループ化されて 1 つの名前を付けられた、単数または複数の変数の集まり

高レベル言語デバッグ: シンボル情報、および高レベル言語情報 (型や関数の定義など) をデバッグ・ツールが使用できるように保持するコンパイラの機能

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・オプションが入っており、リンカ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、またはリンクされません。つまり、オブジェクト・ファイルには何の影響も与えません。

さ

再配置: シンボルのアドレスが変更されるときに、リンカがそのシンボルに対するすべての参照を調整すること

し

シェル・プログラム: コンパイル、アセンブル、および任意のリンクを 1 ステップで実行できるユーティリティ。シェルは、コンパイラ (パーサ、オプティマイザ、コード・ジェネレータを含む)、アセンブラ、およびリンカによって 1 つまたは複数のソース・モジュールを実行します。

式: 定数、シンボル、または算術演算子によって区切られた一連の定数とシンボル

実行可能モジュール: ターゲット・システムで実行できる、リンクされたオブジェクト・ファイル

実行時の自動初期化: C コードをリンクする場合に、リンカが使用する自動初期化の方法。リンカは、−c オプションを指定して起動された場合に、この方法を使用します。リンカにより、.cinit セクションのデータ・テーブルがメモリにロードされ、変数が実行時に初期化されます。

自動初期化: プログラムの実行の前に、C のグローバル変数 (.cinit セクションに保持されている) を初期化すること

出力セクション: リンクされた実行可能モジュールの中の、最終的な割り当て済みセクション

出力モジュール: ターゲット・システム上にダウンロードして実行できる、リンクされた実行可能オブジェクト・ファイル

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、実際の内容はもたない COFF セクション。このセクションは .bss 疑似命令または .usect 疑似命令で構成されます。

シンボリック・デバッグ: シンボル情報を保持して、シミュレータやエミュレータなどのデバッグ・ツールがこの情報を使用できるようにするソフトウェア・ツールの機能

シンボル: アドレスまたは値を表す英数字の文字列

シンボル・テーブル: ファイルで定義して使用されるシンボルについての情報を含んだ COFF オブジェクト・ファイルの部分

す

スタンドアロン・プリプロセッサ: マクロ、#include ファイル、および条件付きコンパイルを独立したプログラムとして実行するソフトウェア・ツール。命令の解析を含む統合的な前処理も実行します。

せ

静的変数: スコープが 1 つの関数または 1 つのプログラムに制限されている変数。静的変数の値は、その関数またはプログラムが終了しても破棄されず、それらの関数またはプログラムが再び開始すると、前の値が再び使用されます。

セクション: コードまたはデータの再配置可能なブロック。最終的にはメモリ・マップ内に連続した空間を占めます。

セクション・ヘッダ: COFF オブジェクト・ファイルの一部分。そのファイルのセクションに関する情報が含まれます。各セクションには専用のヘッダがあり、ヘッダは、そのセクションの開始アドレスを指し、サイズなどの情報を含んでいます。

絶対アドレス: メモリ・ロケーションに永久的に割り当てられるアドレス

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むファイル。コードをコンパイルやアセンブルすることによって、オブジェクト・ファイルを作成します。

た

代入文: 値を指定して変数を初期化する文

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

ターゲット・メモリ: TMS320C2x、TMS320C2xx、TMS320C5x ベースのシステムの、実行可能なオブジェクト・コードがロードされる物理メモリ

ち

直接呼び出し: ある関数が、関数の名前を使用して別の関数を呼び出す関数呼び出し

て

定数: 値を変更できない型

と

統合プリプロセッサ: C プリプロセッサにはパーサが組み込まれており、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したりできます。

動的メモリ割り当て: いくつかの関数 (malloc、calloc、realloc など) によって使用される技法。このメモリ割り当てでは、実行時に変数のメモリを動的に割り当てることができます。これは、大きなメモリ・プール (ヒープ) を宣言し、これらの関数を使用してヒープからメモリを割り当てることによって行います。

な

生データ: 出力セクション内の実行可能データまたは初期化データ

は

バイト: 一般的には、1 つの単位として処理される、隣接する 8 ビットのシーケンス。ただし、TMS320C2x/C2xx/C5x のバイトは 16 ビットです。

注: TMS320C2x/C2xx/C5x のバイトは 16 ビットです

ANSI C 定義では、sizeof 演算子によってオブジェクトを格納するために必要なバイト数が生成されます。さらに ANSI では、sizeof が char に適用される場合、結果は 1 であると規定されています。TMS320C2x/C2xx/C5x では char は 16 ビット (個別にアドレス指定できるように) であるため、1 バイトも 16 ビットになります。これにより、予期しない結果が発生します。たとえば、sizeof (int) = = 1 (2 ではない) などです。TMS320C2x/C2xx/C5x では、バイトとワードは等しく (ともに 16 ビット) なります。

パーサ: ソース・ファイルの読み取り、前処理機能の実行、構文のチェック、およびオプティマイザやコード・ジェネレータの入力として使用される中間ファイルを生成するソフトウェア・ツール

ふ

ファイル・レベルの最適化: 最適化のレベルの 1 つ。コンパイラは、ファイル全体に関する情報を使用してコードを最適化します (コンパイラがプログラム全体に関する情報を使用してコードを最適化するプログラム・レベルの最適化とは反対の意味をもちます)。

フィールド: TMS320C2x、TMS320C2xx、および TMS320C5x では、ソフトウェアで構成可能なデータ型であり、長さ 1 ～ 16 ビットの範囲で自由にプログラミングできます。

符号拡張: 値の未使用の MSB を、その値の符号ビットで埋めること

符号なし値: 実際の符号にかかわらず、正数として取り扱われる値の一種

部分リンク: 再びリンクされるファイルのリンク

プラグマ: コンパイラに対して、特定の文の処理方法について指示を与えるプリプロセッサ疑似命令

プリプロセッサ: マクロ定義の解釈、マクロの展開、ヘッダ・ファイルの解釈、条件付きコンパイルの解釈、およびプリプロセッサ疑似命令の処理を行うソフトウェア・ツール

ブロック: 中括弧 { } でまとめられた一連の宣言または文

へ

米国規格協会 (ANSI): 業界の自主的規格を設定する組織

変数: 一連の値のうちのどれかを想定する、ある量を表すシンボル

ま

マクロ: 命令として使用できるユーザ定義ルーチン

マクロ定義: マクロを構成する名前とコードを定義する、ソース文のブロック

マクロ展開: マクロ呼び出しに代わってソース文をコードに挿入するプロセス

マクロ呼び出し: マクロを起動すること

マップ・ファイル: リンカが作成する出力ファイル。メモリ構成、セクション構成、セクションの割り当て、およびシンボルとシンボルが定義されているアドレスを示します。

め

メモリ・マップ: ターゲット・システムのメモリ空間のマップ。複数の機能ブロックに区画分けされています。

ら

ラベル: アセンブラ・ソース文の 1 カラム目から始まるシンボルで、その文のアドレスに対応します。ラベルは、1 カラム目から始めることのできる唯一のアセンブラ文です。

ランタイム (実行時) 環境: ユーザのプログラムで機能しなければならないランタイム・パラメータ。これらのパラメータは、メモリおよびレジスタの規則、スタックの編成、関数呼び出し規則、およびシステムの初期化により定義されます。

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、およびその他の関数とルーチンのソースが格納されているライブラリ・ファイル rts.src

ランタイムサポート関数: 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

り

リスト・ファイル: アセンブラが作成する出力ファイル。ソース文、その行番号、および SPC への効果が記述されています。

リンカ: オブジェクト・ファイルを結合して、オブジェクト・モジュールを生成するソフトウェア・ツール。生成されたモジュールは、システム・メモリに割り当てられ、デバイスにより実行できます。

ろ

ローダ: 実行可能モジュールをシステム・メモリにロードするデバイス

ロード時の初期化: C コードをリンクするときに、リンカが使用する自動初期化の方法。リンカは、ユーザが −cr オプションを指定して起動された場合に、この方法を使用します。この方法では、実行時でなくロード時に変数が初期化されます。

わ

割り当て: リンカが出力セクションの最終的なメモリ・アドレスを計算するプロセス

索引

数字

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

あ

い

え

お

か

き

く

け

こ

さ

し

ね

は

ひ

ふ

も

よ

ら

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

大阪営業所　〒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)
名古屋営業所　〒460-0003　名古屋市中区錦2丁目4番3号　錦パークビル10階　☎052(232)5601(代表)
松本営業所　〒390-0815　長野県松本市深志1丁目2番11号　松本昭和ビル6階　☎0263(33)1060(代表)
立川営業所　〒190-0012　東京都立川市曙町2丁目36番2号　ﾌｧｰﾚ立川ｾﾝﾀｰｽｸｴｱ10階　☎0425(27)6760(代表)
京都営業所　〒600-8216　京都市下京区塩小路通西洞院東入東塩小路町843−2　日本生命京都ヤサカビル5階　☎075(341)7713(代表)
大宮営業所　〒330-0802　埼玉県大宮市宮町1丁目38番1号　野村不動産大宮共同ビル4階　☎048(647)2622(代表)
米国駐在事務所　TEXAS INSTRUMENTS Inc., 8505 Forest Lane Dallas, 75243, U.S.A (TI Japan Ltd., U.S.A Sales Office)
☎1−972−480−7464

工　場　埼玉県・鳩ヶ谷市　大分県・日出町　茨城県・美浦村　静岡県・小山町（制御機器事業部）
株式会社テキサス･インスツルメンツ筑波開発センター　茨城県・つくば市

■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) ―――― ☎ 0120-81-0026　FAX 0120-81-0036
E−mail:pic−japan@ti.com

TMS320C2x/C2xx/C5x

オプティマイジング(最適化)Cコンパイラ
ユーザーズ・マニュアル

第3版　2000年 6月
第2版　1999年 4月
第1版　1996年11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)

Printed in Japan
’00.06

日本テキサス・インスツルメンツ株式会社

SCJ2222A