TEXAS INSTRUMENTS

TMS320C55x
アセンブリ言語ツール

ユーザーズ・マニュアル

2005 *Digital Signal Processing Solutions*

SPRU280 翻訳版 *JAJU028 WAS SPRUE46*

TMS320C55x
アセンブリ言語ツール
ユーザーズ・マニュアル

対応英文マニュアル：SPRU280H
2004 年 7 月

2005年 9 月

TEXAS INSTRUMENTS

ご注意

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

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

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

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

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

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

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

弊社半導体製品の取り扱い・保管について

半導体製品は、取り扱い、保管・輸送環境、基板実装条件によっては、お客様での実装前後に破壊/劣化、または故障を起こすことがあります。
弊社半導体製品のお取り扱い、ご使用にあたっては下記の点を遵守して下さい。

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

以上

2001.11

まえがき

最初にお読み下さい

本書について

TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアルは、以下のアセンブラ言語ツールの使い方を説明します。

- ❑ アセンブラ
- ❑ アーカイバ
- ❑ リンカ
- ❑ 絶対リスタ
- ❑ クロスリファレンス・リスタ
- ❑ Hex 変換ユーティリティ
- ❑ 逆アセンブラ
- ❑ 名前ユーティリティ

本書の使い方

本書の目的は、TMS320C55x DSP 専用に設計された Texas Instruments のアセンブリ言語ツールの使い方について、理解の手助けとなることです。本書は 4 つの部分に分かれています。

- ❑ **はじめに**では、アセンブリ言語開発ツールの概要を述べ、共通オブジェクト・ファイル・フォーマット（COFF）についても説明します。COFF を使うと、TMS320C55x ツールをより効果的に使用できます。アセンブラおよびリンカを使用する前に、第 2 章「共通オブジェクト・ファイル・フォーマットの概要」を読んで下さい。
- ❑ **アセンブラの説明**では、ニーモニックおよび代数表記アセンブラの使用に関する詳細を説明します。具体的には、アセンブラの起動方法、ソース文のフォーマット、有効な定数と式、アセンブラ出力、およびアセンブラ疑似命令について説明します。マクロ要素についても説明します。
- ❑ **追加アセンブリ言語ツール**では、アセンブラとともに提供され、アセンブリ言語ソース・ファイルの作成を支援する各ツールについて、詳細を説明します。たとえば、第 8 章では、リンカの起動方法、リンカの機能、およびリンカ疑似命令の使用方法を説明します。第 14 章では、Hex 変換ユーティリティの使用方法について説明します。

- **参考資料**では、補足情報を提供します。この節では、COFF オブジェクト・ファイルの内部フォーマットおよび構造に関する技術解説を記述しています。C/C++ コンパイラが使用するシンボリック・デバッグ疑似命令について説明します。最後に、Hex 変換ユーティリティ例、アセンブラおよびリンカのエラー・メッセージ、用語集が含まれます。

表記上の規則

本書では以下の規則を使用します。

- プログラム・リスト、プログラム例、および対話式表示は、特殊な書体で表示されます。例は、強調するために特殊な書体の**太字**を使い、対話式表示は特殊な書体の**太字**を使って、プロンプト、コマンド出力、エラー・メッセージなどシステムが表示する項目とユーザーの入力するコマンドを区別します。

  プログラム・リストの例を次に示します。

```
2 0001   2f          x       .byte  47
3 0002   32          z       .byte  50
4 0003                       .text
```

- 構文の記述では、命令、コマンド、または疑似命令は**太字**のフォント、パラメータはイタリックです。**太字**の構文の部分は、以下のように入力する必要があります。イタリックの構文の部分は、入力する必要のある情報をタイプを示しています。コマンド行の構文の例を次に示します。

  abs55 *filename*

  abs55 はコマンドです。コマンドは、絶対リスタを起動し、*filename* の示す 1 つのパラメータを持っています。絶対リスタの起動時、絶対リスタが入力として使用するファイルの名前を指定します。

- 大括弧（**[** および **]**）は、任意のパラメータを識別します。任意のパラメータを使う場合、カッコ内の情報を指定します。括弧そのものは入力しません。これは、任意のパラメータを持つコマンドの例です。

  hex55 [*-options*] *filename*

  hex55 コマンドには 2 つのパラメータがあります。最初のパラメータ *-options* は任意です。*options* は複数のため、複数のオプションを選択できます。2 番目のパラメータ *filename* は必須です。

❑ アセンブラ構文では、1 カラム目はラベルまたはシンボルの先頭の文字のため予約されています。ラベルまたはシンボルが**任意**の場合、通常は表示されません。**必須**パラメータの場合、以下の例のように、陰影を付けたボックスに対して左詰めで開始するように表示されます。シンボルまたはラベル以外の命令、コマンド、疑似命令、またはパラメータが 1 カラム目から始めてはなりません。

```
symbol .usect   "section name", size in words [, blocking flag]
                [, alignment flag]
```

symbol は、.usect 疑似命令では必須のため、1 カラム目から始めなければなりません。*section name* は二重引用符で囲み、セクション *size in words* はカンマでセクション名と区切る必要があります。*blocking flag* および *alignment flag* は任意です。使用する場合は、カンマで区切る必要があります。

❑ 一部の疑似命令は、異なる数のパラメータを持つ可能性があります。例えば、.byte 疑似命令は最大 100 のパラメータを持つ場合があります。この疑似命令の構文は次のとおりです。

.byte は 1 カラム目から始まらないことに注意してください。

```
.byte   value1 [, ... , valuen]
```

この構文では、.byte が最低 1 つの value パラメータを持たなければならないこと、しかしカンマで区切って追加の value パラメータを入れることもできることを示しています。

❑ 以下は、本書で使われる他のシンボルおよび略語です。

シンボル	説明	シンボル	説明
AR0-AR7	0 から 7 までの補足レジスタ	PC	プログラム・カウンタ・レジスタ
B,b	接尾部 — 2 進整数	Q,q	接尾部 — 8 進整数
H,h	接尾部 — 16 進整数	SP	スタック・ポインタ・レジスタ
LSB	最下位ビット	ST	ステータス・レジスタ
MSB	最上位ビット		

当社発行の関連文献

以下の文書は TMS320C55x デバイスおよび関連サポート・ツールについて説明しています。

TMS320C55x オプティマイジング（最適化）C/C++ コンパイラ ユーザーズ・マニュアル（文献番号 SPRU281）は、TMS320C55x™ C/C++ コンパイラについて解説しています。この C/C++ コンパイラは、ISO 標準の C/C++ ソース・コードを受け入れて、TMS320C55x デバイス用の アセンブリ言語ソース・コードを作成します。

TMS320C55x DSP CPU リファレンス・ガイド（文献番号 SPRU371）は、TMS320C55x™ デジタル・シグナル・プロセッサ（DSP）用の CPU のアーキテクチャ、レジスタ、および機能について説明しています。

TMS320C55x DSP ニーモニック命令セット・リファレンス・ガイド（文献番号 SPRU374）は、TMS320C55x™ DSP ニーモニック命令を個別に解説しています。また、命令セットの一覧、命令コードのリスト、および代数表記命令セットへのクロスリファレンスも含まれています。

TMS320C55x DSP 代数表記命令セット・リファレンス・ガイド（文献番号 SPRU375）は、TMS320C55x™ DSP 代数表記命令を個別に解説しています。また、命令セットの要約、命令オペコードの一覧、およびニーモニック命令セットへのクロスリファレンスも含まれています。

TMS320C55x プログラマ・ガイド（文献番号 SPRU376）は、TMS320C55x™ DSP 用に C/C++ とアセンブリコードを最適化する方法について解説し、DSP の特殊な機能と命令を使うコードの書き方を説明しています。

Code Composer ユーザーズ・マニュアル（文献番号 SPRU328）は、Code Composer が提供する開発環境を使用してリアルタイムの組み込み型リアルタイム DSP アプリケーションをビルドする方法、およびそのデバッグ方法について解説しています。

商標

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

目次

図

表

例

注

第 1 章

はじめに

TMS320C55x™ DSP は、以下のアセンブリ言語ツールによってサポートされています。

- ❑ アセンブラ
- ❑ アーカイバ
- ❑ リンカ
- ❑ 絶対リスタ
- ❑ クロスリファレンス・ユーティリティ
- ❑ オブジェクト・ファイル表示ユーティリティ
- ❑ 名前ユーティリティ
- ❑ ストリップ・ユーティリティ
- ❑ Hex 変換ユーティリティ
- ❑ 逆アセンブラ

本章では、上記の各ツールを一般的なソフトウェア開発手順にどのように当てはめるかについて、および個々のツールについて簡単に説明します。また、読者の理解を助けるために、C コンパイラとデバッグ・ツールについても要点を説明します。コンパイラとデバッガ、および TMS320C55x デバイスの詳細は、「まえがき」の中の「当社発行の関連文献」にリストアップした書籍を参照してください。

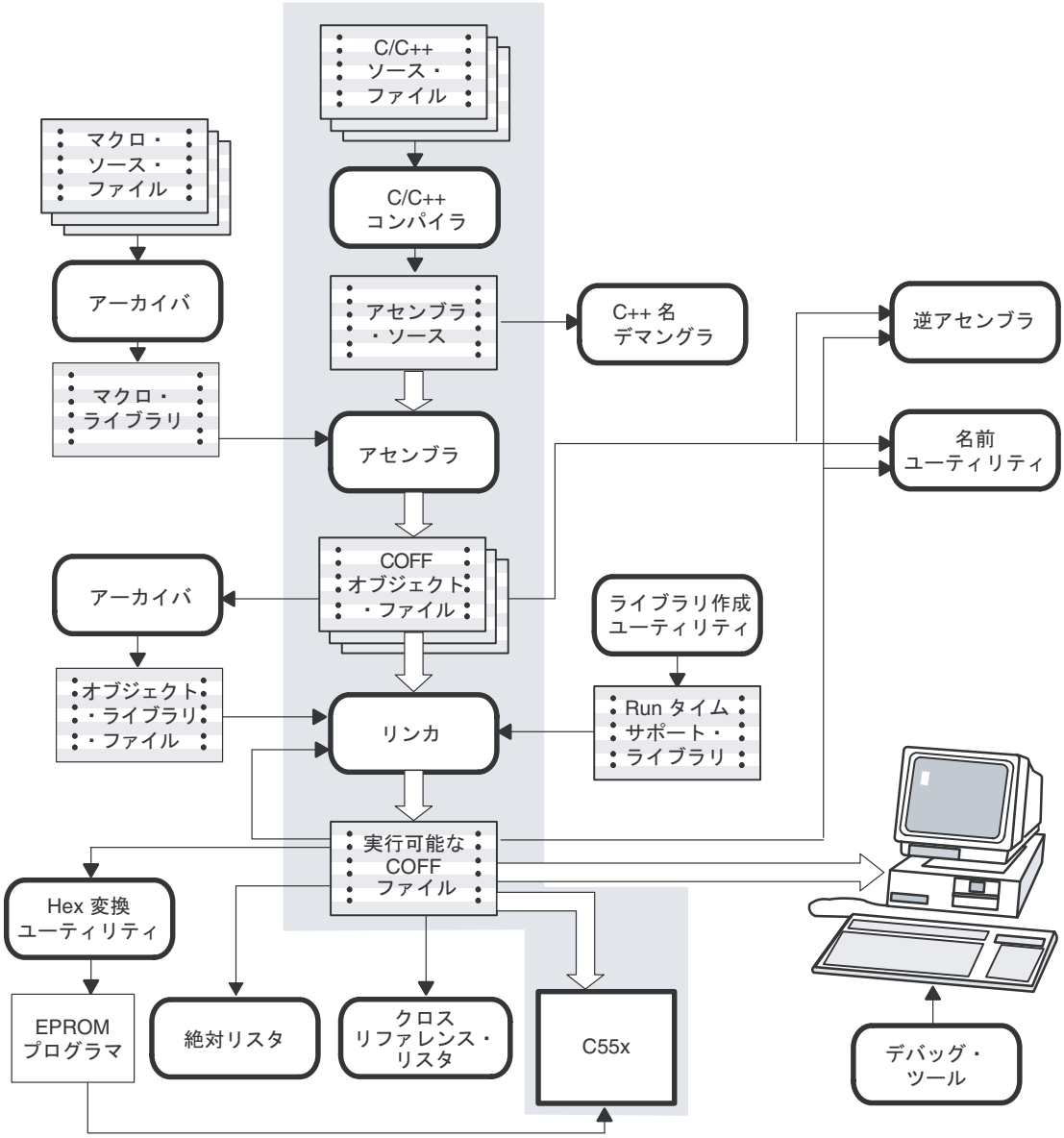
アセンブリ言語ツールでは、モジュラ・プログラミングに便利なように、共通オブジェクト・ファイル・フォーマット（COFF）を使ったオブジェクト・ファイルを生成し、使用しています。オブジェクト・ファイルには、それぞれが独立したコードおよびデータのブロック（セクションと呼ばれる）が含まれており、これを C55x™ メモリ空間にロードすることができます。COFF に関する基礎的な知識があると、C55x のプログラミングの能率が上がります。第 2 章「共通オブジェクト・ファイル・フォーマットの概要」で、このオブジェクト・フォーマットについて詳しく説明します。

項目 **ページ**

1.1 ソフトウェア開発ツールの概要

図 1-1 に C55x ソフトウェア開発のフローを示しています。図の中の陰影を付けた部分は、C 言語プログラムのソフトウェア開発における最も一般的な経路です。それ以外の部分は、開発プロセスを強化する周辺機能です。

図 1-1. TMS320C55x ソフトウェア開発のフロー



1.2 各ツールの説明

次に図 1-1 に示されているツールについて説明します。

- **C/C++ コンパイラ**は C/C++ ソース・コードを C55x アセンブリ言語ソース・コードに変換します。コンパイラ・パッケージには、**ライブラリ作成ユーティリティ**が含まれています。このユーティリティを使って、各自のランタイム・ライブラリを作成できます。

- **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 COFF オブジェクト・ファイルに変換します。TMS320C55x ツールには 2 つのアセンブラが含まれます。ニーモニック・アセンブラは C54x および C55x ニーモニック・アセンブリ・ソース・ファイルを受け入れます。代数表記アセンブラは C55x 代数表記アセンブリ・ソース・ファイルを受け入れます。ソース・ファイルには、命令、アセンブラ疑似命令、マクロ疑似命令を含めることができます。アセンブラ疑似命令を使用すると、ソース・リスト・フォーマット、データ位置合わせ、セクションの内容などアセンブリ・プロセスのさまざまな面を制御できます。

- **リンカ**は、アセンブラにより作成された複数の再配置可能な COFF オブジェクト・ファイルを結合して、1 つの実行可能な COFF オブジェクト・モジュールを作成します。リンカは、実行可能モジュールの作成時にシンボルをメモリ位置にバインドして、シンボルに対するすべての参照を解決します。オブジェクト・ファイルと同様、リンカ・ソース・ファイルはアーカイバ・ライブラリ・メンバ、リンカ・コマンド・ファイル、および前回リンカを実行したときに作成された出力モジュールにできます。リンカ疑似命令を使用すると、オブジェクト・ファイルのセクションを結合し、セクションやシンボルをアドレスやメモリ範囲内にバインドし、グローバル・シンボルを定義または再定義できます。

- **アーカイバ**を使用すると、複数のファイルを集めて 1 つのアーカイブ・ファイルを作成できます。たとえば、いくつかのマクロを集めて 1 つのマクロ・ライブラリにできます。アセンブラはこのライブラリを検索し、ソース・ファイルによりマクロとして呼び出されたメンバを使用します。アーカイバを使用して、いくつかのオブジェクト・ファイルを集めて 1 つのオブジェクト・ライブラリにすることもできます。リンカは、外部シンボルへの参照を解決する必要があるため、いくつかのオブジェクト・ライブラリ・メンバを 1 つのリンク済み出力ファイルに合体します。

- **ライブラリ作成ユーティリティ**は、カスタマイズされた独自の C/C++ ランタイムサポート・ライブラリを作成します。標準ランタイムサポート・ライブラリの関数は、rts.src のソース・コードおよび rts55.lib、ラージ・モデルの rts55x.lib、およびフェーズ 2 の rts55z.lib のオブジェクト・コードとして提供されます。

- TMS320C55x Code Composer Studio デバッガは、COFF ファイルを入力として受け入れますが、ほとんどの EPROM プログラマは COFF ファイルを入力として受け入れません。**Hex 変換ユーティリティ**は、COFF オブジェクト・ファイルを、TI-tagged、Intel、Motorola、Tektronix のいずれかのオブジェクト形式に変換します。変換後のファイルは、EPROM プログラマへダウンロードできます。

- **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。これらの .abs ファイルをアセンブルすると、相対アドレスでなく絶対アドレスが入ったリストを作成できます。また、リンカ -abs オプションを使って絶対リストも作成できます。絶対リスタがないと、そのようなリストの作成は多くの手動操作が必要となるため、面倒な作業となります。

- **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・リストを作成し、シンボル、シンボルの定義、およびリンク済みソース・ファイルでのシンボルの参照をリストにして示します。

この開発プロセスの主な目的は、C55x ターゲット・システムで実行できるモジュールを生成することです。次のいずれかのデバッグ・ツールを使用して、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・ツールには、次の製品があります。

- 命令の正確度を確認するためのソフトウェアであるシミュレータ
- XDS エミュレータ

これらのデバッグ・ツールは、Code Composer Studio の中でアクセスできます。詳細は、Code Composer Studio 入門マニュアルを参照してください。

第 2 章

共通オブジェクト・ファイル・フォーマットの概要

アセンブラとリンカを使って、TMS320C55x™ デバイスで実行可能なオブジェクト・ファイルを作成することができます。これらのオブジェクト・ファイルのフォーマットを共通オブジェクト・ファイル・フォーマット（COFF）と呼びます。

COFF を使用するとアセンブリ言語プログラムを書くときに、コードまたはデータのブロック単位で考えることができるので、モジュラ・プログラミングをさらに容易に行えるようになります。このようなブロックをセクションと呼びます。アセンブラおよびリンカでは、セクションを作成し、操作するための疑似命令が用意されています。

本章では COFF のセクションの概要を示します。詳細は、付録 A「共通オブジェクト・ファイル・フォーマット」を参照してください。COFF 構造についての説明があります。

項目 **ページ**

2.1 セクション

オブジェクト・ファイルの最小単位をセクションと呼びます。セクションはコードまたはデータのブロックで、最終的にはメモリ・マップの中の連続した空間に格納されます。オブジェクト・ファイルの個々のセクションは分かれており、お互いに独立しています。COFF オブジェクト・ファイルには、必ず次の 3 つのデフォルトのセクションが含まれています。

.text セクション	通常は実行可能なコードを含みます。
.data セクション	通常は初期化されたデータを含みます。
.bss セクション	通常は、初期化されない変数のための空間を確保します。

さらに、アセンブラとリンカを使うと、.data、.text、および .bss の各セクションと同様の使い方をする名前付きセクションを作成し、名前を付け、リンクすることができます。

セクションには 2 つの基本的な型があります。

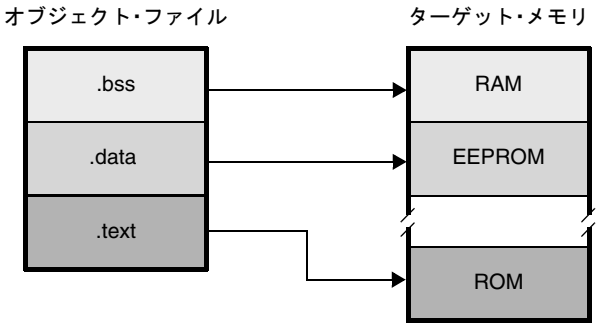
初期化されたセクション	データまたはコードを含みます。.text セクションと .data セクションは初期化されます。.sect アセンブラ疑似命令により作成された名前付きセクションも初期化されます。
初期化されないセクション	初期化されないデータのための空間を確保します。.bss セクションは初期化されません。.usect アセンブラ疑似命令により作成された名前付きセクションも初期化されません。

いくつかのアセンブラ疑似命令を使用すれば、コードおよびデータのさまざまな部分を適切なセクションに関連付けることができます。アセンブラは、アセンブリ処理中にこれらのセクションを作成し、図 2-1 のような構造のオブジェクト・ファイルを作成します。

リンカの機能の 1 つに、セクションのターゲット・システムのメモリ・マップへの配置があります。これを割り当てと呼びます。ほとんどのシステムではいくつかの異なった型のメモリが使用されているので、セクションを使用するとターゲット・メモリをさらに効率的に利用できるようになります。すべてのセクションは独立していて、再配置が可能です。それぞれのセクションは、ターゲット・メモリ内の割り当てられたブロックに入れることができます。たとえば初期化ルーチンを持つセクションを定義して、それをメモリ・マップの ROM を含む部分に割り当てることができます。

図 2-1 は、オブジェクト・ファイルにあるセクションと仮想のターゲット・メモリとの関係を示したものです。

図 2-1. メモリの論理ブロックへの区画分け



2.2 アセンブラによるセクションの処理

アセンブラは、アセンブリ言語プログラムの複数の部分が 1 つのセクションに属することを識別します。アセンブラには、この機能をサポートするいくつかの疑似命令があります。

- ❑ **.bss**
- ❑ **.usect**
- ❑ **.text**
- ❑ **.data**
- ❑ **.sect**アセンブラ疑似命令**:**セクションの定義**:.data[**あせんぶらぎじめいれい**:**せくしょんのていぎ **:.data]**

.bss 疑似命令および .usect 疑似命令は、初期化されないセクションを作成します。その他の疑似命令は初期化されたセクションを作成します。

任意のセクションのサブセクションを作成して、メモリ・マップをより厳密に制御できます。.sect 疑似命令と .usect 疑似命令を使用して、サブセクションを作成します。サブセクションは、コロンで区切られたベース・セクション名とサブセクション名で識別できます。詳細は、2.2.4 項「サブセクション」（2-8 ページ）を参照してください。

注： デフォルトの SECTIONS 疑似命令

SECTIONS 疑似命令を 1 つも使用しない場合、アセンブラは .text セクション内にすべてをアセンブルします。

2.2.1 初期化されないセクション

初期化されないセクションは、プロセッサ・メモリに空間を確保します。この空間は、通常は RAM に割り当てられます。このようなセクションは、オブジェクト・ファイルに実際の内容をもたず、メモリを確保するだけです。プログラムは、実行時にこの空間を使用して変数の作成と格納を行うことができます。

初期化されないデータ領域は、.bss および .usect アセンブラ疑似命令を使用して作成できます。

- ❑ .bss 疑似命令は、空間を .bss セクションに確保します。
- ❑ .usect 疑似命令は、空間を特定の初期化されない名前付きセクションに確保します。

.bss 疑似命令を起動するたびに、アセンブラはさらに多くの空間を適切なセクションに確保します。.usect 疑似命令を起動するたびに、アセンブラはさらに多くの空間を指定された名前付きセクションに確保します。

これらの疑似命令の構文は次のとおりです。

```
        .bss symbol, size in words [, [blocking flag] [, alignment flag]]
symbol  .usect "section name ", size in words [, [blocking flag] [, alignment flag]]
```

symbol
.bss 疑似命令または .usect 疑似命令の起動により確保される空間の最初のワードを指します。*symbol* は、空間を確保する変数の名前に対応します。symbol は、他のどのセクションからも参照でき、また（.global アセンブラ疑似命令を使用して）グローバル・シンボルとして宣言することもできます。

size in words
必須のパラメータです。
- ❏ .bss 疑似命令は、.bss セクションに *size* ワードを確保します。
- ❏ .usect 疑似命令は、*section name* に *size* ワードを確保します。

blocking flag
任意のパラメータです。このパラメータに 0 より大きい値を指定すると、アセンブラは *size* で指定されたワード数を連続したものとして扱います。*size* が 1 ページを越えていない限り、割り当てられた空間がページ境界にまたがることはありません。1 ページを越える場合は、オブジェクトはページ境界から開始されます。

alignment flag
任意のパラメータです。

section name
アセンブラに対して、どの名前付きセクションに空間を確保すべきかを指定します。名前付きセクションに関する詳細は、2.2.3 項「名前付きセクション」（2-7 ページ）を参照してください。

.text、.data、および .sect の各疑似命令は、アセンブラに対して現行のセクションへのアセンブリ作業を中止し、指定されたセクションへのアセンブルを開始するように指示します。ただし .bss 疑似命令と .usect 疑似命令の場合は、現行のセクションを終了せずに新しいセクションを開始します。現行のセクションから一時的にエスケープするだけです。.bss 疑似命令と .usect 疑似命令は初期化されたセクションのどこででも使用でき、セクションの内容には何の影響も及ぼしません。

初期化されないサブセクションを作成するには、.usect 疑似命令を使用します。アセンブラは、初期化されないサブセクションを初期化されないセクションと同様に扱います。サブセクション作成についての詳細は、2.2.4 項「サブセクション」（2-8 ページ）を参照してください。

2.2.2 初期化されたセクション

初期化されたセクションには、実行可能なコードまたは初期化されたデータが入ります。これらのセクションの内容は、オブジェクト・ファイルに格納され、プログラムのロード時にプロセッサ・メモリに入れられます。個々の初期化されたセクションは別々に再配置することが可能で、他のセクションで定義されているシンボルを参照できます。リンカは自動的にこれらのセクションの相対的な参照を解決します。

3 種類の疑似命令で、アセンブラに対してコードまたはデータを 1 つのセクションに入れるように指示します。各疑似命令の構文は次のとおりです。

.text [*value*]
.data [*value*]
.sect "*section name***" [,** *value*]

アセンブラがこれらの疑似命令を見つけると、現行のセクションへのアセンブルを中止します（暗黙に「現行のセクションを中止せよ」というコマンドを受けたように動作します）。そしてアセンブラは、次にまた .text、.data、または .sect のいずれかの疑似命令を見つけるまでその後のコードを指定のセクション内にアセンブルします。*value*（指定されている場合）には、セクション・プログラム・カウンタ（SPC）の開始値を指定します。セクション・プログラム・カウンタの開始値の指定は 1 度だけ行われます。セクション・プログラム・カウンタの開始値は、アセンブラがそのセクションに対する疑似命令を最初に処理するときに実施されなければなりません。デフォルトでは、SPC は 0 からスタートします。

セクションは反復工程により構築されます。たとえば、アセンブラが最初に .data 疑似命令を見つけたときには、その .data セクションは空です。この最初の .data 疑似命令に続く文は（アセンブラが次に .text 疑似命令か .sect 疑似命令のいずれかを見つけるまで）.data セクション内にアセンブルされます。アセンブラは次の .data 疑似命令を見つけると、その .data 疑似命令に続く文をすでに .data セクションに入っている文に追加します。このようにして、メモリに連続的に配置できる 1 つの .data セクションを作成します。

初期化されたサブセクションは .sect 疑似命令で作成します。アセンブラは、初期化されたサブセクションを初期化されたセクションと同様に扱います。サブセクション作成についての詳細は、2.2.4 項「サブセクション」（2-8 ページ）を参照してください。

2.2.3 名前付きセクション

名前付きセクションは、ユーザーが作成します。名前付きセクションはデフォルトの .text、.data、および .bss セクションと同様に使用できますが、別々にアセンブルされます。

たとえば、.text 疑似命令を繰り返して使用することにより、オブジェクト・ファイルに 1 つの .text セクションを構築できます。この .text セクションは、リンク時に独立したユニットとしてメモリに割り当てることができます。ここに .text に割り当てたくない実行可能コードの部分（初期化ルーチンなど）があるとします。このコード・セグメントを名前付きセクション内にアセンブルすると、.text とは別にアセンブルされます。これを .text とは別のメモリ位置に割り当てることができます。.data セクションとは別に初期化されたデータをアセンブルすることも、.bss セクションとは別に初期化されない変数の空間を確保することも可能です。

次の疑似命令を使用して名前付きセクションを作成できます。

- **.usect** 疑似命令を使うと、.bss セクションと同様に使えるセクションを作成できます。これらのセクションは、変数を入れるための空間を RAM に確保します。
- **.sect** 疑似命令を使うと、デフォルトの .text セクションや .data セクションと同様にコードやデータを含むことのできるセクションを作成できます。.sect 疑似命令を使用すると、再配置可能なアドレスをもった名前付きセクションを作成できます。

各疑似命令の構文は次のとおりです。

symbol **.usect** "*section name*", *size in words* [, [*blocking flag*] [, *alignment*]]
.sect "*section name*"

section name パラメータはセクションの名前です。最大 32767 個の独立した名前付きセクションを作成することができます。セクション名は 200 文字まで有効です。.sect 疑似命令と .usect 疑似命令では、セクション名がサブセクション名を指す場合があります（詳細は、2.2.4 項「サブセクション」を参照してください）。

これらの疑似命令を新しい名前を付けて起動するたびに、新しい名前付きセクションが作成されます。これらの疑似命令をすでに使われた名前を付けて起動するたびに、アセンブラはコードまたはデータをその名前の付いたセクションにアセンブルします（または空間を確保します）。複数の疑似命令で同じ名前を使うことはできません。つまり、.usect 疑似命令を使用してセクションを作成し、同じ名前のセクションを .sect で使用することはできません。

2.2.4 サブセクション

サブセクションは、セクション内に存在する小さなセクションです。セクション同様、サブセクションについてもリンカで操作できます。サブセクションを設定すると、メモリ・マップをより詳細に制御できます。サブセクションを作成するには、.sect または .usect 疑似命令を使用します。サブセクション名の構文は次のとおりです。

section name:*subsection name*

サブセクションは、ベース・セクション名と固有のサブセクション名をコロンでつないだ名前で識別されます。サブセクションは個別に割り当てることも、同じベース・セクション名をもつ他のセクションとグループ化して割り当てることもできます。たとえば .text セクション内に _func という名前のサブセクションを作成するには、次のように入力します。

```
.sect ".text:_func"
```

_func サブセクションは、個別に割り当てることも、.text の全セクションの中に割り当てることもできます。

次の 2 つのタイプのサブセクションを作成できます。

- ❑ 初期化されたサブセクションは .sect 疑似命令によって作成します。2.2.2 項「初期化されたセクション」（2-6 ページ）を参照してください。
- ❑ 初期化されないサブセクションは .usect 疑似命令によって作成します。2.2.1 項「初期化されないセクション」（2-4 ページ）を参照してください。

サブセクションの割り当て方法は、セクションの場合と同じです。詳細は、8.9 節「SECTIONS 疑似命令」（8-32 ページ）を参照してください。

2.2.5 セクション・プログラム・カウンタ

アセンブラはセクションごとに別々のプログラム・カウンタを使用します。このようなプログラム・カウンタは、セクション・プログラム・カウンタ （SPC）と呼ばれます。

SPC は、コードまたはデータのセクション内の現行のアドレスを表します。始めに、アセンブラは各 SPC を 0 に設定します。アセンブラはセクションにコードやデータを入れるたびに、該当するセクションの SPC をインクリメントします。あるセクションへのアセンブルを再開する場合、アセンブラは該当する SPC の以前の値を取り出し、SPC の元の値からインクリメントし続けます。

アセンブラは、各セクションがアドレス 0 で始まるかのように処理します。リンカは、セクションの最終ロケーションに従って各セクションをメモリ・マップ内に再配置します。詳細は、2.4 節「再配置」（2-15 ページ）を参照してください。

2.2.6 SECTIONS 疑似命令の使用例

例 2–1 は、SECTIONS 疑似命令を使用して異なるセクション間を前後にスワップしながら、インクリメンタルに COFF セクションを構築する方法を示したものです。SECTIONS 疑似命令を使用することにより、セクションへのアセンブルを開始することや、すでにコードを組み込んでいるセクションへのアセンブルを継続することができます。継続の場合には、アセンブラの作業は、すでにそのセクション内にあるコードに新しいコードを追加することだけです。

例 2–1 のフォーマットは、リスト・ファイルです。例 2–1 は、アセンブリ中に SPC がどのように修正されるかを示しています。リスト・ファイル中の行には 4 つのフィールドがあります。

フィールド 1　ソース・コードの行カウンタが入ります。

フィールド 2　セクション・プログラム・カウンタが入ります。

フィールド 3　オブジェクト・コードが入ります。

フィールド 4　オリジナルのソース文が入ります。

例 2–1. SECTIONS 疑似命令の使用法

```
      2                 ************************************************
      3                 **  Assemble an initialized table into .data. **
      4                 ************************************************
      5  000000                       .data
      6  000000 0011    coeff        .word     011h,022h,033h
         000001 0022
         000002 0033
      7                 ************************************************
      8                 **   Reserve space in .bss for a variable.    **
      9                 ************************************************
     10  000000                       .bss      buffer,10
     11                 ************************************************
     12                 **              Still in .data.               **
     13                 ************************************************
     14  000003 0123    ptr          .word     0123h
     15                 ************************************************
     16                 **    Assemble code into the .text section.   **
     17                 ************************************************
     18  000000                       .text
     19  000000 A01E    add:         MOV       0Fh,AC0
     20  000002 4210    aloop:       SUB       #1,AC0
     21  000004 0450                 BCC       aloop,AC0>=#0
         000006 FB
     22                 ************************************************
     23                 **    Another initialized table into .data.   **
     24                 ************************************************
     25  000004                       .data
     26  000004 00AA    ivals        .word     0AAh, 0BBh, 0CCh
         000005 00BB
         000006 00CC
     27                 ************************************************
     28                 ** Define another section for more variables. **
     29                 ************************************************
     30  000000         var2         .usect    "newvars", 1
     31  000001         inbuf        .usect    "newvars", 7
     32                 ************************************************
     33                 **       Assemble more code into .text.       **
     34                 ************************************************
     35  000007                       .text
     36  000007 A114    mpy:         MOV       0Ah,AC1
     37  000009 2272    mloop:       MOV       T3,HI(AC2)
     38  00000b 1E0A                 MPYK      #10,AC2,AC1
         00000d 90
     39  00000e 0471                 BCC       mloop,!overflow(AC1)
         000010 F8
     40                 ************************************************
     41                 **  Define a named section for int. vectors.  **
     42                 ************************************************
     43  000000                       .sect     "vectors"
     44  000000 0011                 .word     011h, 033h
     45  000001 0033
```

フィールド 1　フィールド 2　フィールド 3　　フィールド 4

図 2-2 に示されているように、例 2–1 のファイルでは 5 つのセクションが作成されます。

.text	17 バイトのオブジェクト・コードが入ります。
.data	7 ワードのオブジェクト・コードが入ります。
vectors	.sect 疑似命令により作成される名前付きセクションです。2 ワードの初期化されたデータが入ります。
.bss	10 ワードをメモリに確保します。
newvars	.usect 疑似命令を使用して作られた名前付きセクションです。8 ワードをメモリに確保します。

2 列目にこれらのセクション内にアセンブルされたオブジェクト・コードが示されています。最初の列には、オブジェクト・コードを生成したソース文の行番号が示されています。

図 2-2. 例 2–1 のファイルで生成されたオブジェクト・コード

行番号	オブジェクト・コード	セクション
19	A01E	.text
20	4210	
21	0450	
21	FB	
36	A114	
37	5272	
38	1E0A	
38	90	
39	0471	
39	F8	
6	0011	.data
6	0022	
6	0033	
14	0123	
26	00aa	
26	00bb	
26	00cc	
44	0011	vectors
45	0033	
10	データなし 10 ワード 確保済み	.bss
30	データなし 8 ワード 確保済み	newvars
31		

2.3 リンカによるセクションの処理

リンカには、セクションに関して 2 つの主要な機能があります。第 1 に、リンカは COFF オブジェクト・ファイルにあるセクションを構成ブロックとして使用します。リンカは、（複数のファイルがリンクされている場合には）入力セクションを結合して実行可能な COFF 出力モジュールに出力セクションを作成します。第 2 に、リンカは出力セクションのためのメモリ・アドレスを選択します。

リンカには、上記の機能をサポートするために 2 つの疑似命令が用意されています。

- **MEMORY 疑似命令**を使用すると、ターゲット・システムのメモリ・マップを定義できます。メモリの各部分に名前を付け、その開始アドレスと長さを指定します。
- **SECTIONS 疑似命令**は、入力セクションを結合して出力セクションにする方法、および出力セクションをメモリ内に配置する場所をリンカに伝えます。

サブセクションを使用すると、より正確にセクションを操作できます。リンカの SECTIONS 疑似命令を使用して、サブセクションを指定できます。サブセクションを明示的に指定しない場合、そのサブセクションは同じベース・セクション名をもつ他のセクションと結合されます。

リンカ疑似命令を必ず使用する必要はありません。リンカ疑似命令を使用しない場合、リンカは 8.13 節「デフォルトの割り当てアルゴリズム」（8-64 ページ）に説明されているターゲット・プロセッサのデフォルトのメモリ配置アルゴリズムを使用します。リンカ疑似命令を使用する場合には、それをリンカ・コマンド・ファイル内で指定しなければなりません。

リンカ・コマンド・ファイルとリンカ疑似命令の詳細は、以下の各節を参照してください。

セクション番号	見出し	ページ
8.6	リンカ・コマンド・ファイル	8-22
8.8	MEMORY 疑似命令	8-28
8.9	SECTIONS 疑似命令	8-32
8.13	デフォルトの割り当てアルゴリズム	8-64

2.3.1 デフォルトのメモリ割り当て

図 2-3 は、2 つのファイルのリンク処理を示しています。

図 2-3. 入力セクションの結合による実行可能オブジェクト・モジュールの作成

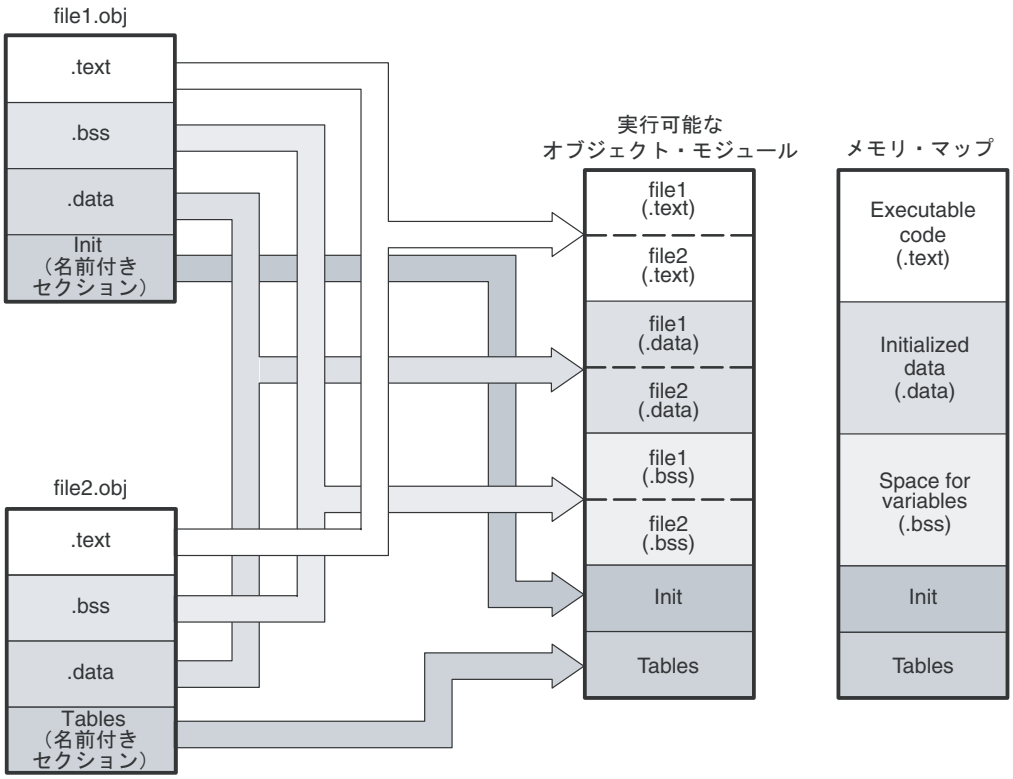


図 2-3 では、file1.obj および file2.obj がアセンブルされていて、リンカ入力として使用されます。各ファイルには、.text、.data、および .bss というデフォルトのセクションと名前付きセクションがあります。実行可能出力モジュールは、これらのセクションの結合を示しています。リンカは、file1.obj の .text と file2.obj の .text を結合して 1 つの .text セクションにします。続いて 2 つの .data セクション、および 2 つの .bss セクションを結合し、最後に名前付きセクションをモジュールの最後に配置します。メモリ・マップは、各セクションのメモリ内での配置を示しています。デフォルトでは、リンカはアドレス 080 から開始して、図で示すようにセクションを順に配置していきます。

2.3.2 セクションのメモリ・マップへの配置

図 2-3 は、リンカのデフォルトでのセクションの結合方法を示しています。デフォルトの設定を使用しない場合もあります。たとえば、.text セクションの全部を結合して 1 つの .text 出力セクションにする必要がない場合があります。また、名前付きセクションを、通常は .data セクションが割り当てられる位置に配置する場合もあります。ほとんどのメモリ・マップは、サイズの異なるさまざまな種類のメモリ（DRAM、ROM、EPROM、など）が含まれています。あるセクションを特定のタイプのメモリに配置したい場合もあります。

メモリ・マップ内のセクションの配置についての詳細は、8.8 節「MEMORY 疑似命令」（8-28 ページ）および 8.9 節「SECTIONS 疑似命令」（8-32 ページ）を参照してください。

2.4 再配置

アセンブラは各セクションがアドレス 0 から始まるのと同じように扱います。すべての再配置可能なシンボル（ラベル）は、そのセクション内でアドレス 0 と相対的な関係にあります。もちろん、すべてのセクションがメモリのアドレス 0 から始まることはありえないので、リンカは次の方法でセクションの**再配置**を行います。

- ❑ セクションをメモリ・マップに割り当てて、適切なアドレスで始まるようにします。
- ❑ シンボルの値を新しいセクション・アドレスに対応するように調整します。
- ❑ 再配置されたシンボルに対する参照を調整後のシンボルの値に合わせて調整します。

リンカは、再配置エントリを使用してシンボルの値に対する参照を調整します。アセンブラは、再配置可能なシンボルが参照されるたびに再配置エントリを作成します。リンカはシンボルの再配置が行われた後、このエントリを使って参照をパッチします。例 2–2 には、C55x の場合に再配置エントリを生成するコード・セグメントが含まれています。

例 2–2. 再配置エントリを生成するコード

```
1                   .ref   X
2                   .ref   Z
3  000000           .text
4  000000 4A04      B   Y
5  000002 6A00      B   Z   ;Generates relocation entry
   000004 0000!
6  000006 7600      MOV #X,AC0;Generates relocation entry
   000008 0008!
7  00000a 9400   Y:  reset
```

例 2–2 では、シンボル X は別のモジュールで定義されているため、再配置が可能です。シンボル Y は PC に対して相対的で、再配置は必要ありません。シンボル Z は、PC に相対的ですが、異なるファイルにあるため、再配置が必要です。コードのアセンブルを行うと、X および Z は値 0 を取ります（アセンブラは、未定義の外部シンボルはすべて値 0 を取るとみなします)。アセンブラは X および Z の再配置エントリを生成します。X および Z に対する参照は外部参照です （リスト内の**！** 文字によって示されます)。

COFF オブジェクト・ファイルの各セクションには、再配置エントリのテーブルがあります。このテーブルには、セクション内の各再配置可能な参照に対して 1 つの再配置エントリがあります。リンカは通常、再配置エントリを使用すると、それを除去します。こうすることにより、(再リンクされたり、ロードされたりした場合に）出力ファイルが再度再配置されることを防ぎます。再配置エントリが含まれていないファイルは絶対ファイルと呼ばれます（すべてのアドレスが絶対アドレスです)。リンカに再配置エントリを引き続き保持させる場合は、リンカを起動する時に -r オプションを使用します。

2.4.1 再配置に対するメッセージの発行

リンカは、特定の再配置に対して警告やエラー・メッセージを発行する可能性があります。

アセンブリのプログラムでは、PC に相対するフィールドの命令が、シンボル、ラベル、またはアドレスへの参照を含む場合には、相対変位がその命令フィールドで当てはまることが要求されます。変位がそのフィールドに当てはまらない（参照アイテムの位置が遠過ぎるために）場合、リンカはエラーを発行します。たとえば、8 ビット、符号なし、PC に相対するフィールドを持つ命令が、その命令から 256 またはそれ以上のバイト数に位置するシンボルを参照するとき、リンカはエラー・メッセージを発行します。

同様に、絶対アドレス・フィールドをもつ命令がシンボル、ラベル、またはアドレスへの参照を含む場合、その参照アイテムは、その命令フィールド内に当てはまるアドレスに置かれることが要求されます。たとえば、ある関数が 0x10000 にリンクされる場合、そのアドレスは 16 ビットの命令フィールドにエンコードすることができません。

いずれのケースでも、リンカは値の上位ビット部を切り捨てます。

これらのメッセージに対処するには、リンク・マップおよびリンカ・コマンド・ファイルを調べます。出力セクションを再アレンジして、参照されるシンボルを参照する命令のより近い位置に置くこともできます。

反対に、幅の広いフィールドをもつ異なるアセンブリ命令を使用することを考えてみましょう。また、下位ビットのみを必要とするシンボルの場合は、マスクして上位ビットを 0 とします。

2.5 実行時の再配置

ときにはコードをメモリのある領域にロードし、それを別の領域で実行する必要が生ずることがあります。たとえば、ROM ベースのシステムにパフォーマンスを左右するコードがあるとします。そのコードは ROM にロードしなければなりませんが、実行は RAM を使用した方がはるかに速くなります。

リンカを使用すると、この指定を簡単に行うことができます。SECTIONS 疑似命令では、オプションでリンカに同じセクションを 2 回割り当てるように指示することが可能です。つまり、最初はロード・アドレスを、2 回目は実行アドレスを設定するように指示できます。ロード・アドレスには *load* キーワードを使用し、実行アドレスには *run* キーワードを使用します。

ロード・アドレスにより、ローダはセクションの生データを配置する場所を決定します。そのセクションに対するすべての参照（ラベルの参照など）は実行アドレスを参照します。アプリケーションは、そのセクションをロード・アドレスから実行アドレスへコピーする必要があります。この作業が自動的に行われないのは、別々の実行アドレスを指定するからです。実行時にコードのブロックがどのように移動するかを示す例については、例 8–7（8-51 ページ）を参照してください。

1 つのセクションに対して 1 回の割り当て（load または run）しか行わない場合は、そのセクションは 1 回だけ割り当てられ、ロードも実行も同じアドレスで行われます。2 つの割り当てを指定した場合、そのセクションは同じサイズの 2 つの異なったセクションであるかのように割り当てられます。

初期化されないセクション（.bss など）はロードされないので、意味のあるアドレスは実行アドレスのみです。リンカは、初期化されないセクションを 1 回だけ割り当てます。ユーザーが実行アドレスとロード・アドレスの両方を指定した場合、リンカは警告を発行し、ロード・アドレスは無視されます。

実行時の再配置の詳細は、8.10 節「セクションのロード時および実行時アドレスの指定方法」（8-45 ページ）を参照してください。

2.6 プログラムのロード

リンカは、実行可能な COFF オブジェクト・モジュールを作成します。実行可能なオブジェクト・ファイルには、リンカ入力に使用されるオブジェクト・ファイルと同じ COFF フォーマットが使用されます。ただし、実行可能なオブジェクト・ファイル内のセクションはターゲット・メモリに適合するように結合され再配置されます。

プログラムをロードする方法は、実行環境に応じていくつかあります。よく使用される 2 つの方法を次に示します。

- ソフトウェア・シミュレータおよびソフトウェア開発システムなどの TMS320C55x デバッグ・ツールには、ローダが組み込まれています。これらのツールには、ローダを起動するための LOAD コマンドが用意されています。ローダは実行可能ファイルを読み取り、プログラムをターゲット・メモリにコピーします。

- Hex 変換ユーティリティ（アセンブリ言語パッケージの一部として出荷されている hex55）を使用して、実行可能 COFF オブジェクト・モジュールを数種類のオブジェクト・ファイル・フォーマットの 1 つに変換することができます。この後、EPROM プログラマを使用して変換済みファイルを EPROM に焼き込むことができます。

2.7 COFF ファイルで使われるシンボル

COFF ファイルには、プログラムで使用されるシンボルに関する情報を格納したシンボル・テーブルがあります。リンカは、このテーブルを使用して再配置を行います。デバッグ・ツールも、シンボリック・デバッグを行うときにこのシンボル・テーブルを使用します。

2.7.1 外部シンボル

あるモジュールで定義されて、別のモジュールで参照されるシンボルを外部シンボルと呼びます。シンボルを外部シンボルとして識別するためには、**.def**、**.ref**、または **.global** 疑似命令を使います。

.def	現行のモジュールで定義され、別のモジュールで使用される。
.ref	別のモジュールで定義されているが、現行のモジュールで参照される。
.global	上記のどちらかに該当する。

次のコード・セグメントは、これらの定義を示しています。

```
    .def        x             ; DEF of x
    .ref        y             ; REF of y
x:   ADD        #86,AC0,AC0   ; Define x
      B             y         ; Reference y
```

x の .def 定義には、x がこのモジュールで定義されている外部シンボルで、他のモジュールは x を参照することができると宣言されています。y の .ref 定義には、y は未定義のシンボルで、他のモジュールで定義されていると宣言されています。

アセンブラは、x と y の両方をオブジェクト・ファイルのシンボル・テーブルに入れます。このファイルが他のオブジェクト・ファイルにリンクされると、x のエントリは、他のファイルからの x に対する未解決の参照を定義します。y のエントリに対して、リンカは他のファイルのシンボル・テーブルにある y の定義を検索します。

リンカは、すべての参照を対応する定義と一致させます。リンカがシンボルの定義を見つけられなかった場合は、その未解決の参照についてのエラー・メッセージを出力します。このようなエラーが発行されると、リンカは実行可能なオブジェクト・モジュールを作成できません。

2.7.2 シンボル・テーブル

アセンブラは、外部シンボル（定義と参照の両方）を見つけると、必ずシンボル・テーブルにエントリを生成します。またアセンブラは、個々のセクションの始まりを指し示す特別なシンボルを作成します。リンカはこのシンボルを使用して、セクション内で定義される参照シンボル、および参照シンボルのアドレスを解決します。

アセンブラは通常は、これ以外の種類のシンボルについてはシンボル・テーブル・エントリを作成しません。リンカがシンボル・テーブル・エントリを必要としないからです。たとえば、ラベルは .global で宣言されていない限り、シンボル・テーブルには組み込まれません。シンボリック・デバッグのためには、プログラム内の個々のシンボルに対するエントリをシンボル・テーブルに含めた方が便利な場合があります。その場合は、アセンブラを起動するときに -as オプションを使用します。

第 3 章

アセンブラの説明

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。このファイルは共通オブジェクト・ファイル・フォーマット（COFF）で作成されます。このフォーマットについては、第 2 章「共通オブジェクト・ファイル・フォーマットの概要」および付録 A「共通オブジェクト・ファイル・フォーマット」に詳細があります。ソース・ファイルには、次のアセンブリ言語要素を使用できます。

アセンブラ疑似命令	第 4 章で説明
マクロ疑似命令	第 5 章で説明
アセンブリ言語命令	TMS320C55x™ Instruction Set リファレンス・ガイドで説明

項目 **ページ**

3.1 アセンブラの概要

TMS320C55x™ には 2 つのアセンブラが含まれます。

- ニーモニック・アセンブラは C54x™ および C55x™ ニーモニック・アセンブリ・ソースを受け入れます。
- 代数表記アセンブラは C55x 代数表記アセンブリ・ソースを受け入れます。

各アセンブラには次の機能があります。

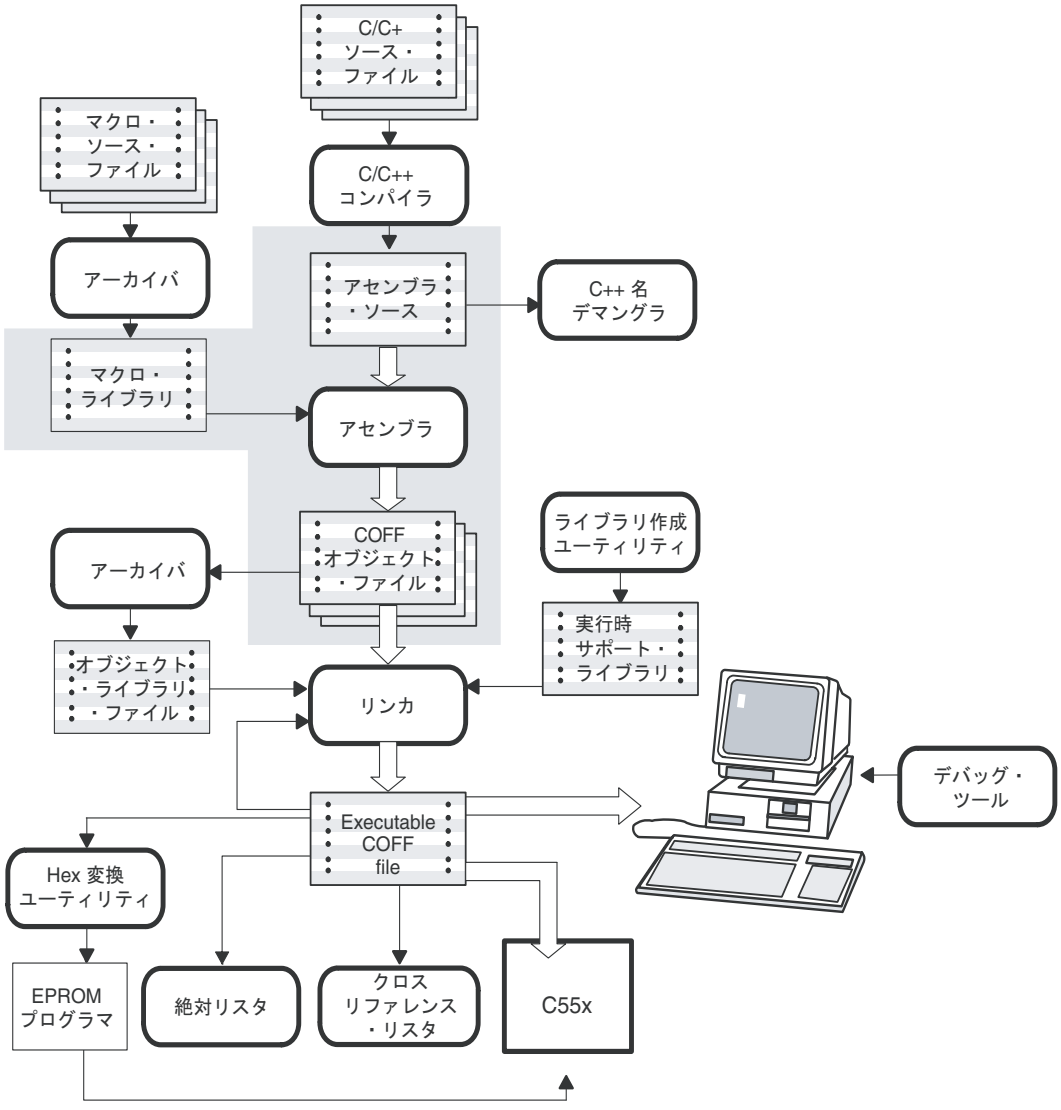
- テキスト・ファイルにあるソース文を処理して再配置可能な C55x オブジェクト・ファイルを作成する。
- ソース・リストを必要に応じて作成し、ユーザーにリストの制御手段を与える。
- コードを複数のセクションに分割し、オブジェクト・コードの各セクションに対応する SPC （セクション・プログラム・カウンタ）を保守する。
- グローバル・シンボルの定義と参照を行い、（必要に応じて）ソース・リストにクロスリファレンス・リストを追加する。
- 条件ブロックをアセンブルする。
- マクロをサポートすることによって、インラインかまたはライブラリで、マクロのユーザー定義ができるようにする

ニーモニック・アセンブラは、サポートされていない C54x 命令にエラー・メッセージおよび警告メッセージを生成します。いくつかの C54x 命令は、1 つの C55x 命令に直接マッピングしません。ニーモニック・アセンブラは、これらの命令を適切な C55x 命令のシリーズに変換します。アセンブラにより生成されたリスト・ファイル（-l オプション付き）に、実行された変換を示します。C55x における C54x コードの実行についての詳細は、第 6 章を参照してください。

3.2 アセンブラ開発のフロー

図 3-1 は、アセンブリ言語開発フローにおけるアセンブラの役割を示したものです。アセンブラは、アセンブリ言語自体により作成されたものでも、C/C++ コンパイラにより作成されたものでも、アセンブリ言語のソース・ファイルを入力として受け入れます。

図 3-1. アセンブラ開発のフロー



3.3 アセンブラの起動方法

コンパイラを使ってアセンブラを起動するには、次のように入力します。

cl55 *input file*[*-options*]

cl55 コンパイラを使ってアセンブラを起動するためのコマンドです。コンパイラは拡張子 .asm を持つファイルをすべてアセンブリ・ファイルと認識し、アセンブラと呼びます。

input file アセンブリ言語ソース・ファイルの名前を指定します。ソース・ファイルには、ニーモニックまたは代数表記命令のどちらかを含む必要があります。両方を含むことはできません。デフォルトの命令セットは、ニーモニックです。代数表記命令セットを指定するには、-mg オプションを使用します。

options 使用するアセンブラ・オプションを指定します。オプションでは大文字と小文字は区別されません。オプションは、コマンドの後ならそのコマンド行のどこにでも指定できます。各オプションの前にはハイフンを付けます。オプションは個別に指定する必要があります。

有効なアセンブラ・オプションは次のとおりです。

-@ **-@=** *filename* は、コマンド行にファイルの内容を添付します。このオプションは、ホスト・オペレーティング・システムにより課せられたコマンド行の長さの制限を無効にするために使用できます。コメントを挿入するには、アスタリスクまたはセミコロン（* または ;）をコマンド・ファイルの行頭に使います。他のカラムからコメントを始める場合は、最初の文字を必ずセミコロン（;）にしなければなりません。

コマンド・ファイル内では、組み込みのスペースまたはハイフンを含んでいるファイル名やオプション・パラメータは、引用符で囲まれる必要があります。次に例を示します。“this-file.obj”

-aa 絶対リストを作成します。-aa を使用すると、アセンブラはオブジェクト・ファイルを作成しません。-aa オプションは絶対リスタと組み合わせて使用します。

-ac アセンブリ言語ファイル内で大文字と小文字を区別しないようにします。たとえば、-ac を指定すると ABC と abc は等価になります。このオプションを指定しない場合、大文字と小文字は区別されます（デフォルト）。大文字と小文字の区別は主にシンボル名で行われ、ニーモニック名やレジスタ名では行われません。

-ad **-ad=***name* [**=***value*] は *name* シンボルを設定します。これは、*name* **.set** [*value*] をアセンブリ・ファイルの先頭に挿入するのと同じです。*value* を省略する場合、シンボルは 1 に設定されます。詳細は、3.10.3 項「シンボル定数の定義（-ad オプション）」（3-31 ページ）を参照してください。

-ahc **-ahc=*filename*** は、アセンブラに対し、アセンブリ・モジュールに指定された *filename* をコピーするように指示します。このファイルはソース・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・ファイルに表示されます。

-ahi **-ahi=*filename*** は、アセンブラに対し、アセンブリ・モジュールに指定された *filename* をインクルードするように指示します。このファイルは、ソース・ファイル文の前にインクルードされます。インクルードされたファイルは、アセンブリ・リスト・ファイルには表示されません。

-al （L の小文字）asm リスト・ファイルを作成します。

-apd アセンブリ・ファイルの前処理を実行しますが、標準作成ユーティリティには前処理された出力を書き込まず、入力に適した従属行のリストを書き込みます。リストは、ソース・ファイルと同じ名前のファイルに、.ppa の拡張子を付けて書き込まれます。

-api アセンブリ・ファイルの前処理を実行しますが、前処理された出力を書き込まず、# インクルード疑似命令に含まれているファイルのリストを書き込みます。リストは、ソース・ファイルと同じ名前のファイルに、.ppa の拡張子を付けて書き込まれます。

-ar **-ar**[#] は、# により特定されるアセンブラ注釈を抑止します。注釈はアセンブラの情報メッセージであり、警告ほど深刻ではありません。# の値を指定しない場合は、すべての注釈は抑止されます。アセンブラ注釈の説明については、7.7 節（7-35 ページ）を参照してください。

-as 定義されたすべてのローカルのシンボルを、オブジェクト・ファイルのシンボル・テーブルに入れます。通常、アセンブラはグローバル・シンボルだけをシンボル・テーブルに入れます。-as を使用すると、ラベルまたはアセンブル時定数として定義されているシンボルもテーブルに入ります。

-ata （ARMS モード）アセンブラに対し、このソース・ファイルの実行中は、最初に ARMS ステータス・ビットが設定されることを通知します。デフォルトでは、アセンブラはこのビットを無効とします。

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

-atc （CPL モード）アセンブラに対し、このソース・ファイルの実行中は、最初に CPL ステータス・ビットが設定されることを通知します。これにより、アセンブラは SP 相対アドレッシング構文の使用を実現します。デフォルトでは、アセンブラはこのビットを無効とします。

-ath アセンブラは、ユーザーの C54x ファイルの移植時に、少ないコード数ではなく速いコードを生成します。デフォルトでは、アセンブラは少ないコード・サイズにエンコードしようとします。

-atl （C54x 互換モード）アセンブラに対し、このソース・ファイルの実行中は、最初に C54x ステータス・ビットが設定されることを通知します。デフォルトでは、アセンブラはこのビットを無効とします。

-atn アセンブラは C54x の遅延あり分岐 / 呼び出し命令の遅延スロットに位置する NOP を除去します。詳細は、7.2.4 項（7-9 ページ）を参照してください。

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

-ats （ニーモニック・アセンブリのみ）即値データ・シフト・カウント・オペランドは # 文字で始めるという要求仕様を緩めます。これにより、ニーモニック・アセンブラの初期バージョンと互換性が持てます。このオプションが使われ、# が省略されると、新しい構文に変更するよう警告が発行されます。

-att アセンブラに対し、このソース・ファイルの実行中は、SST ステータス・ビットがゼロであることを通知します。デフォルトでは、アセンブラはこのビットを有効とします。

-atv アセンブラに対し、すべての goto/call が 24 ビット・オフセットとしてエンコードされることを通知します。デフォルトでは、アセンブラは、すべての可変長命令を、最小のサイズに解決しようとします。

-atw （代数表記アセンブリのみ）アセンブラの警告メッセージをすべて抑制します。

-au **-au=***name* は、事前定義された定数 *name* を未定義にします。また、-ad*name* オプションによる定数の定義も無効にします。

-aw パイプライン・コンフリクト警告を有効にします。

-ax クロスリファレンス・ファイルを作成し、リスト・ファイルの最後に追加します。また、クロスリファレンス・ユーティリティが使えるように、クロスリファレンス情報をオブジェクト・ファイルに追加します。リスト・ファイルを要求しなかった場合でも、アセンブラによりリスト・ファイルが自動的に作成されます。

-g C ソース・デバッガにおけるアセンブラ・ソースのデバッグを可能にします。アセンブリ言語のソース・ファイルの各行ごとに、行情報が COFF ファイルに出力されます。.line 疑似命令を含むアセンブリ・コードには、-g オプションは使用できません。詳細は、3.14 節「アセンブリ・ソースのデバッグ方法」（3-45 ページ）を参照してください。

-ɪ .copy、.include または .mlib 疑似命令が命名するファイルをアセンブラが検索するディレクトリを指定します。-ɪ オプションのフォーマットは -ɪ*pathname* です。詳細は、3.6.1 節「-ɪ アセンブラ・オプションの使用」（3-19 ページ）を参照してください。

-mg アセンブラは代数表記アセンブリ・ファイルを受け入れます。代数表記アセンブリ入力ファイルをアセンブルするには、-mg オプションを使わなければなりません。代数表記のソース・コードとニーモニック・ソース・コードを、同じソース・ファイル内に混在させることはできません。

--purecirc （ニーモニック・アセンブリのみ）アセンブラに対し、C54x ファイルが C54x サーキュラ・アドレッシングを使うことを表明します（C55x リニア／サーキュラ・モード・ビットは使いません）。詳細は、7.2.3 項（7-7 ページ）を参照してください。

-v **-v** デバイスは、命令を作成するプロセッサを指定します。有効な値についての詳細は、TMS320C55x オプティマイジング（最適化）C/C++ コンパイラ ユーザーズ・マニュアルを参照してください。

3.4 アセンブラを直接起動する方法

注： asm55 および masm55

コード作成ツールを今後拡張するために、代数表記（asm55）アセンブラおよびニーモニック（masm55）アセンブラの直接起動は推奨しません。しかし、必要な場合はアセンブラを直接起動できます。

代数表記アセンブラおよびニーモニック・アセンブラを直接起動するには、次のように入力します。

masm55 [*input file*[*object file*[*listing file*]]][*-options*]
asm55 [*input file*[*object file*[*listing file*]]][*-options*]

masm55
asm55
アセンブラを起動するためのコマンドです。**masm55** はニーモニック・アセンブラを起動し、**asm55** は代数表記アセンブラを起動します。

input file
アセンブリ言語ソース・ファイルの名前を指定します。masm55 は、*input file* に有効なニーモニック・アセンブリ・ソース・ファイル（代数表記命令ではない）を設定し、そして asm55 は、*input file* に有効な代数表記アセンブリ・ソース・ファイル（ニーモニック命令ではない）を設定する。拡張子を付けないと、-f アセンブラが使用されていない場合、アセンブラはデフォルトの拡張子である *.asm* を使用します。入力ファイル名を指定しないと、アセンブラはプロンプトを出して指定を促します。

object file
アセンブラが作成する C55x オブジェクト・ファイルの名前を指定します。拡張子を付けない場合、アセンブラはデフォルトの *.obj* を付けます。オブジェクト・ファイルを指定しないと、アセンブラは入力ファイル名に拡張子 *.obj* を付けたファイルを作成します。

listing file
アセンブラが作成するオプションのリスト・ファイルの名前を指定します。

- *listing file* を指定しない場合、-l（小文字の L）オプションまたは -x オプションを使用しない限り、アセンブラはリスト・ファイルを作成しません。この場合、アセンブラは .lst 拡張子の付いた入力ファイル名を使用し、リスト・ファイルをその入力ファイルに直接置きます。
- *listing file* を指定しても拡張子を付けない場合、アセンブラはデフォルトの拡張子として *.lst* を使用します。

options
使用するアセンブラ・オプションを指定します。オプションでは大文字と小文字は区別されません。オプションは、アセンブラ名の後ならそのコマンド行のどこにでも指定できます。各オプションの前にはハイフンを付けます。オプションは個別に指定する必要があります。

有効なアセンブラ・オプションは次のとおりです。

-a 絶対リストを作成します。-a を使用すると、アセンブラはオブジェクト・ファイルを作成しません。-a オプションは絶対リスタと組み合わせて使用します。

-c アセンブリ言語ファイル内で大文字と小文字を区別しないようにします。たとえば、-c を指定すると ABC と abc は等価になります。このオプションを指定しない場合、大文字と小文字は区別されます（デフォルト）。大文字と小文字の区別は主にシンボル名で行われ、ニーモニック名やレジスタ名では行われません。

-d **-d***name* [=*value*] は *name* シンボルを設定します。これは、*name* **.set** [*value*] をアセンブリ・ファイルの先頭に挿入するのと同じです。*value* を省略する場合、シンボルは 1 に設定されます。詳細は、3.10.3 項「シンボル定数の定義（-ad オプション）」（3-31 ページ）を参照してください。

-f 拡張子を含まないソース・ファイル名に .asm 拡張子を追加するという、アセンブラのデフォルト動作を抑止します。

-g ソース・デバッガにおけるアセンブラ・ソースのデバッグを可能にします。アセンブリ言語のソース・ファイルの各行ごとに、行情報が COFF ファイルに出力されます。すでに .line 疑似命令を含むアセンブリ・コードには -g オプションを使えません（つまり、C/C++ コンパイラによって生成されたコードは、-g または --symdebug:dwarf とともに実行されます）。

-h
-help
-? これらのオプションは、いずれも使用可能なアセンブラ・オプションのリストを表示します。

-hc **-hc***filename* は、アセンブラに対し、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指示します。このファイルはソース・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・ファイルに表示されます。

-hi **-hi***filename* は、アセンブラに対し、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指示します。このファイルは、ソース・ファイル文の前にインクルードされます。インクルードされたファイルは、アセンブリ・リスト・ファイルには表示されません。

-ɪ .copy、.include または .mlib 疑似命令が命名するファイルをアセンブラが検索するディレクトリを指定します。-ɪ オプションのフォーマットは -ɪ*pathname* です。詳細は、3.6.1 項「-ɪ アセンブラ・オプションの使用」（3-19 ページ）を参照してください。

-l （L の小文字）リスト・ファイルを作成します。

-ma （ARMS モード）アセンブラに対し、このソース・ファイルの実行中は、ARMS ステータス・ビットが有効になることを通知します。デフォルトでは、アセンブラはこのビットを無効とします。

-mc （CPL モード）アセンブラに対し、このソース・ファイルの実行中は、CPL ステータス・ビットが有効になることを通知します。これにより、アセンブラは SP 相対アドレッシング構文の使用を実現します。デフォルトでは、アセンブラはこのビットを無効とします。

-mh アセンブラは、ユーザーの C54x ファイルの移植時に、少ないコード数ではなく速いコードを生成します。デフォルトでは、アセンブラは少ないコードを生成しようとします。詳細は、7.2.2 項（7-6 ページ）を参照してください。（cl55 のみにサポート）

-ml C55x ラージ・メモリ・モデルを指定します。このオプションは _ _large_model シンボルを 1 に設定します。このオプションを使う場合、アセンブラはオブジェクト・ファイルをラージ・モデル・ファイルとしてマークを付けます。この操作により、オブジェクト・モジュールのスモール・モデルとラージ・モデルの無効な組み合わせを検出するための情報をリンカに提供します。

-mn （ニーモニック・アセンブリのみ）アセンブラは C54x の遅延あり分岐 / 呼び出し命令の遅延スロットに位置する NOP を除去します。詳細は、7.2.4 項（7-9 ページ）を参照してください。

-ms （ニーモニック・アセンブリのみ）即値データ・シフト・カウント・オペランドは # 文字で始めるという要求仕様を緩めます。これにより、ニーモニック・アセンブラの初期バージョンと互換性が持てます。このオプションが使われ、# が省略されると、新しい構文に変更するよう警告が発行されます。

-mt （ニーモニック・アセンブリのみ）アセンブラに対し、SST ステータス・ビットがこの移植された C54x ソース・ファイルの実行中に無効になることを通知します。デフォルトでは、アセンブラはこのビットを有効とします。詳細は、7.2.1 項（7-5 ページ）を参照してください。

-mv アセンブラに対し、特定の可変長命令に最大の（P24）書式を使用するようにします。デフォルトでは、アセンブラは、すべての可変長命令を、最小のサイズに解決しようとします。

-mw （代数表記アセンブリのみ）アセンブラの警告メッセージを抑制します。

--purecirc （ニーモニック・アセンブリのみ）アセンブラに対し、C54x ファイルが C54x サーキュラ・アドレッシングを使うことを表明します（C55x リニア／サーキュラ・モード・ビットは使いません）。詳細は、7.2.3 項（7-7 ページ）を参照してください。

-q （静的な実行）見出しとすべての進捗情報を出力しません。

-r **-r**[*num*] は、*num* により特定されるアセンブラ注釈を抑止します。注釈はアセンブラの情報メッセージであり、警告ほど深刻ではありません。*num* の値を指定しない場合は、すべての注釈は抑止されます。アセンブラ注釈の説明については、7.7 節（7-35 ページ）を参照してください。

-s 定義されたすべてのシンボルを、オブジェクト・ファイルのシンボル・テーブルに入れます。通常、アセンブラはグローバル・シンボルだけをシンボル・テーブルに入れます。-s を使用すると、ラベルまたはアセンブル時定数として定義されているシンボルもテーブルに入ります。

-u **-u***name* は、事前定義された定数 *name* を未定義にします。また、-d オプションによる定数の定義も無効にします。

-x クロスリファレンス・テーブルを作成し、リスト・ファイルの最後に追加します。また、クロスリファレンス・ユーティリティが使えるように、クロスリファレンス情報をオブジェクト・ファイルに追加します。リスト・ファイルを要求しなかった場合でも、アセンブラによりリスト・ファイルが自動的に作成されます。

3.5 C55x アセンブラの機能

この項では、以下の C55x アセンブラ特有の機能に関する重要情報を記載しています。

- ❑ バイト / ワード・アドレッシング（3.5.1 項）
- ❑ パラレル命令規則（3.5.2 項）
- ❑ 可変長命令（3.5.3 項）
- ❑ メモリ・モード（3.5.4 項）
- ❑ MMR アドレス使用についての警告（3.5.5 項）

3.5.1 バイト / ワード・アドレッシング

C55x メモリは、コードの場合はバイト・アドレス可能な 8 ビットで、データの場合はワード・アドレス可能な 16 ビットです。アセンブラおよびリンカは、該当セクションに適切なアドレスの記録、相対オフセット、およびユニットのビット・サイズを保管します。データ・セクションの場合はワード、コード・セクションの場合はバイトです。

注： .struct および .union 構造体に含まれるオフセット

.struct または .union 構造体に定義されたフィールドのオフセットは、現行のセクションに関わらず、ワードでカウントされます。アセンブラは、.struct または .union が、常にデータ・コンテキストの中で使用されると仮定しています。

3.5.1.1 コード・セクションの定義

アセンブラは、以下のいずれかが正しい場合は、セクションをコード・セクションとして識別します。

- ❑ セクションが .text 疑似命令により導入されているとき。
- ❑ セクションに、少なくとも 1 つの命令がアセンブルされているとき。

セクションが .text、.data.、または .sect 疑似命令により導入されたものでない場合、アセンブラはこのセクションが .text（コード）セクションであると仮定します。セクション・タイプはアセンブラのオフセットとサイズの計算を決定するため、オブジェクトをセクションにアセンブルする前に、現行の作業セクションをコードまたはデータとして明確に定義することが大切です。

3.5.1.2 アセンブリ・プログラムとネイティブ・ユニット

アセンブラとリンカは、使用するコードがデータ・セグメントのコンテキストではワード・アドレスとオフセットを使用して、コード・セグメントのコンテキストではバイト・アドレスとオフセットを使用して作成されていると仮定しています。

- プログラム・アドレス・バスを経由してアドレスが送信される場合（呼び出しまたは分岐のターゲットとして使われる場合など）、プロセッサは、フル 24 ビット・アドレスを想定しています。このコンテキストで使用される定数は、バイトで表現する必要があります。

- データ・アドレス・バスを使用してアドレスを送信する場合（アドレスにより、読み書きの対象となるメモリの位置が指示される場合など）、プロセッサは 23 ビットのワード・アドレスを想定しています。このコンテキストで使用される定数は、ワードで表現する必要があります。

- アセンブリ・リスト・ファイルの PC 値カラムは、リスト中のセクションに適した単位でカウントされます。コード・セクションの場合、PC はバイトでカウントされ、データ・セクションの場合、PC はワードでカウントされます。

  次に例を示します。

```
1 000000             .text  ; PC is counted in BYTES
2 000000 2298        MOV AR1,AR0
3 000002 4010        ADD #1,AC0
4
5 000000             .data  ; PC is counted in WORDS
6 000000 0004        .word 4,5,6,7
  000001 0005                ; PC is 1 word
  000002 0006                ; PC is 2 words ...
  000003 0007
7 000004 0001  foo   .word 1
```

- 文字を処理するデータ定義疑似命令（.byte、.ubyte、.char、.uchar および .string）は、コード・セクションの中ではバイトごとに 1 文字を割り振り、データ・セクションの中ではワードごとに 1 文字を割り振ります。しかし、Texas Instruments では、データ定義疑似命令は、データ・セクションの中でのみ使用されるよう、強くお勧めします（完全なリストについては表 4–1（b）（4-3 ページ）を参照してください）。

- アドレス可能な単位で表現されたサイズ・パラメータを持つ疑似命令は、このパラメータがコード・セクションの場合はバイトで、データ・セクションの場合はワードで表現されることを想定しています。

  たとえば、次のようになります。

```
.align 2
```

  PC を、コード・セクションでは 2 バイト（16 ビット）の境界に合わせ、データ・セクションでは 2 ワード（32 ビット）の境界に合わせます。

下記のサンプル・コードには、C55x に関するデータとコードが示されています。

例 3–1. C55x データ例

```
.def Struct1, Struct2
.bss Struct1, 8  ; allocate 8 WORDS for Struct1
.bss Struct2, 6  ; allocate 6 WORDS for Struct2

.text
MOV *(#(Struct1 + 2)),T0  ; load 3rd WORD of Struct1
MOV *(#1000h),T1      ; 0x1000 is an absolute WORD
                          ; address (i.e., byte 0x2000)
```

例 3–2. C55x コード例

```
.text
.ref Func
CALL #(Func + 3) ;jump to address "Func plus 3 BYTES"
CALL #0x1000    ;0x1000 is an absolute BYTE address
```

3.5.1.3 コードをデータとして使う方法とデータをコードとして使う方法

アセンブラは、コード・アドレスをデータ・アドレスとして処理する方法（プログラム・スペースへのデータの読み書きなど）はサポートしていません。ただし、コードが別々のロード・メモリ配置と実行メモリ配置を持っている場合は除きます。そのような場合、コードはワード・アドレスに位置合わせしなければなりません。詳細は、8.17 節「リンカが生成するコピー・テーブル」（8-77 ページ）を参照してください。

同様に、アセンブラは、データ・アドレスをコード・アドレスとして処理する方法（データ・ラベルへの分岐の実行など）もサポートしていません。アドレス可能単位のサイズが異なるため、この機能はサポートできません。コード・ラベル・アドレスは 24 ビットのバイト・アドレスですが、データ・ラベル・アドレスは 23 ビットのワード・アドレスです。

その結果、次のようなことがいえます。

- ❑ コードとデータを 1 つのセクションの中で混同してはなりません。データはすべて（定数データも）、コードとは分けてセクションに置く必要があります。
- ❑ アプリケーションでプログラム・セクションにビット・データを読み書きすることは危険であり、動作が保証されない可能性があります。

3.5.2 並列命令規則

アセンブラは、並列になった対の命令を、TMS320C55x 命令設定リファレンス・ガイドに指定された規則に従って意味をチェックします。

アセンブラは、並列性を有効にするために、2 つの命令をスワップすることがあります。たとえば、下記の命令セットは両方とも有効で、同一のオブジェクト・ビットにコード化されます。

```
AC0 = AC1 || T0 = T1 ^ #0x3333
T0 = T1 ^ #0x3333 || AC0 = AC1
```

3.5.3 可変長命令のサイズ決定

デフォルトでは、アセンブラは、スタンドアロンの可変長命令を、可能な限り最小のサイズまで決定しようとします。たとえばアセンブラは、使用可能な 3 つの無条件のアドレス分岐命令の中から、選択可能な最小のものを選択しようとします。

```
goto L7
goto L16
goto P24
```

可変長命令で使われるアドレスがアセンブル時に認識されないと（たとえば、別のファイルで定義されたシンボルである場合など）、アセンブラはその命令の使用可能な最大の書式を選択します。上記 3 つの使用可能な分岐命令のうち、goto P24 を取り上げてみます。

以下の命令グループでサイズ決定が行われます。

```
goto L7, L16, P24
if (cond) goto l4, L8, L16, P24
call L16, P24
if (cond) call L16, P24
```

場合によっては、ある命令の最大の書式（P24）を使用したいことがあります。ある命令の P24 バージョンは、同じ命令のより小さなバージョンよりも少ないサイクルで実行します。たとえば、“goto P24” は 4 バイト、3 サイクルを使用しますが、“goto L7” は 2 バイトで 4 サイクルを使用します。

-mv アセンブラ・オプションまたは .vli_off 疑似命令を使用して、次の命令を最大の書式に保ちます。

```
goto P24
call P24
```

-mv アセンブラ・オプションは、ファイル全体の中で、上記の命令のサイズ決定を抑制します。.vli_off 疑似命令と .vli_on 疑似命令を使用して、アセンブリ・ファイルの領域に対するこの動作を切り替えることができます。コマンド行オプションと疑似命令でコンフリクトが起こった場合、疑似命令が優先します。

-mv オプションを使用しているか、.vli_off 疑似命令を使用しているかに関わらず、アセンブラは、他の可変長命令を可能な最小のサイズに決定し続けます。

.vli_off および .vli_on 疑似命令の有効範囲は静的であり、アセンブリ・プログラムの制御フローには影響されません。

3.5.4 メモリ・モード

アセンブラは、3 つのメモリ・モード・ビット（または 8 つのメモリ・モード）をサポートしています。C54x 互換性、CPL および ARMS です。アセンブラは、指定されたモードに基づいて、入力を受け入れたり、拒否したりします。また、モードに基づいて、同じ入力の別のエンコーディングも作成することがあります。

メモリ・モードは、C54CM、CPL、および ARMS のステータス・ビットの値に相当します。アセンブラは、ステータス・ビットの値は保管できません。これらのビットの値をアセンブラに通知するには、アセンブラ疑似命令とコマンド行オプション、あるいはそのいずれかを使用する必要があります。C54CM、CPL、または ARMS ステータス・ビットの値を変更する命令の直後には、該当するアセンブラ疑似命令が必要です。アセンブラは、これらのビット値の変更を通知されると、これらのモードの構文および意味の違反について有効なエラー・メッセージおよび警告メッセージを発行できます。

3.5.4.1 C54x 互換モード

C54X 互換モードは、ソース・ファイルが C54x コードから変換された場合に必要です。変換後のソース・コードを C55x ネイティブ・コードにするよう変更するまでは、コードの領域について C54x 互換モードを指定するには .c54cm_on および .54cm_off 疑似命令を使用します。.c54cm_on および .c54cm_off 疑似命令は、引数を取りません。コマンド行オプションと疑似命令でコンフリクトが起こった場合、疑似命令が優先します。

.c54cm_on および .c54cm_off 疑似命令の有効範囲は静的であり、アセンブリ・プログラムの制御フローには影響されません。.c54cm_on 疑似命令と .c54cm_off 疑似命令との間にあるすべてのアセンブリ・コードは、C54x 互換モードでアセンブルされます。

C54x 互換モードでは、メモリ・オペランドで T0（C55x インデックス・レジスタ）ではなく AR0 が使われます。たとえば C54x 互換モードでは、*（AR5 + T0）は無効であり、*（AR5 + AR0）を使わなければなりません。

3.5.4.2 CPL モード

CPL モードは、直接アドレッシングに影響します。アセンブラは、CPL ステータス・ビットの値は保管できません。したがって、CPL 値をモデル化するには、.cpl_on および .cpl_off 疑似命令を使う必要があります。これらの疑似命令の 1 つを発行した直後に、CPL ビットの値を変更する命令を続けます。.cpl_on 疑似命令は、CPL ステータス・ビットを 1 にセットするのと同様です。これは、-mc コマンド行オプションを使うのと同等です。.cpl_off 疑似命令は、CPL ステータス・ビットを 0 に設定します。.cpl_on および .cpl_off 疑似命令は、引数を取りません。コマンド行オプションと疑似命令でコンフリクトが起こった場合、疑似命令が優先します。

.cpl_on および .cpl_off 疑似命令の有効範囲は静的であり、アセンブリ・プログラムの制御フローには影響されません。.cpl_on と .cpl_off 疑似命令の間にあるすべてのアセンブリ・コードは、CPL モードでアセンブルされます。

CPL モード（.cpl_on）では、直接メモリ・アドレッシングはスタック・ポインタ（SP）に関連しています。dma 構文は *SP（ *dma* ）で、*dma* には、定数または再配置可能なシンボル式が可能です。アセンブラは、*dma* の値を出力ビットにコード化します。

デフォルトでは（.cpl_off）、直接メモリ・アドレッシング（dma）はデータ・ページ・レジスタ（DP）に関連しています。dma 構文は @*dma* で、*dma* には、定数または再配置可能なシンボル式が可能です。アセンブラは、*dma* と DP レジスタの値の差を計算し、この差を出力ビットにコード化します。

DP は 1 つのファイルで参照できますが、そのファイルでは定義できません（外部で設定されます）。したがって、DP 値を使用前にアセンブラに通知するには、.dp 疑似命令を使う必要があります。この疑似命令を発効した直後に、DP レジスタの値を変更する命令を続けます。疑似命令の構文は、次のとおりです。

.dp *dp_value*

dp_value には、定数または再配置可能なシンボル式を指定できます。

疑似命令がファイルで使われない場合、アセンブラは DP の値が 0 であると仮定します。.dp 疑似命令の有効範囲は静的であり、プログラムの制御フローには影響されません。疑似命令に設定された値は、次の .dp 疑似命令が検出されるまで、またはソース・ファイルの最後に到達するまで使われます。

MMR ページおよび I/O ページへの dma アクセスは、CPL モードが指定されているかどうかに関わらず、アセンブラによって同様に処理されます。MMR ページへのアクセスは、構文中 mmap() 修飾子によって示されます。I/O ページへのアクセスは、readport() および writeport() 修飾子によって示されます。dma アクセスは、0 の原点に相対的なものとして、アセンブラにより必ずコード化されます。

3.5.4.3 ARMS モード

ARMS モードは、間接アドレッシングに影響し、コントローラ・コードのコンテキストにおいて便利です。アセンブラは、ARMS ステータス・ビットの値は保管できません。したがって、ARMS 値をアセンブラにモデル化するには、.arms_on および .arms_off 疑似命令を使う必要があります。これらの疑似命令の 1 つを発行した直後に、ARMS ビットの値を変更する命令を続けます。.arms_on 疑似命令は、1 に設定された ARMS ステータス・ビットをモデル化します。これは、-ma オプションを使用することと同等です。.arms_off 疑似命令は、0 にセットされた ARMS ステータス・ビットをモデル化します。.arms_on および .arms_off 疑似命令は、引数を取りません。

-ma オプションと疑似命令の間にコンフリクトが起こった場合、疑似命令が優先します。

.arms_on および .arms_off 疑似命令の有効範囲は静的であり、アセンブリ・プログラムの制御フローには影響されません。.arms_on 行と .arms_off 行の間にあるすべてのアセンブリ・コードは、ARMS モードでアセンブルされます。

デフォルト（.arms_off）により、アセンブリ・コードを指定する間接メモリ・アクセス修飾子が選択されます。

ARMS モード（.arms_on）では、間接メモリ・アクセス用のショート・オフセット修飾子が使われます。これらの修飾子は、コード・サイズの最適化により効果的です。

3.5.5 MMR アドレス使用についてのアセンブラ警告

シングル・メモリ・アクセス・オペランド（Smem）が要求されるコンテキストでメモリ・マップ・レジスタ（MMR）が使われる場合、ニーモニック・アセンブラ（cl55）が「MMR アドレスの使い方」の警告を発行します。この警告は、アセンブラが MMR の使用を DP 相対直接アドレス・オペランドとして解釈していることを示しています。命令を記載の通りに動作させるには、DP は 0 でなければなりません。次に例を示します。

```
ADD   SP, T0
```

ここで “MMR アドレスの使用 ” 警告を受信します。

```
“file.asm”, WARNING! at line 1:[W9999] Using MMR address
```

この命令の働きについて、アセンブラは以下のように警告します。

```
ADD   value at address(DP + MMR address of SP), T0
```

DP が 0 の場合に限り、SP の値にアクセスできます。

1 バイト長くなりますが、この命令を書く最良の方法は以下のとおりです。

```
ADD   mmap(SP), T0
```

DP が 0 であるとわかっていて、このような参照が意図的である場合、接頭部 @ を使うことにより警告を避けられます。

```
ADD   @SP, T0
```

この警告は、C54x から継承した C55x 命令には生成されません。

3.6 アセンブラ入力のための代替ファイルと代替ディレクトリの命名方法

.copy、.include、および .mlib 疑似命令は、アセンブラに対して外部ファイルからのコードを使用するように指示します。.copy および .include 疑似命令は、アセンブラに対して外部ファイルからのソース文を読み取るように指示し、.mlib 疑似命令は、マクロ関数を含んだライブラリ名を指定します。第 4 章「アセンブラ疑似命令」には、.copy、.include および .mlib 疑似命令の例があります。これらの疑似命令の構文は次のとおりです。

```
.copy "filename"
.include "filename"
.mlib "filename"
```

filename には、アセンブラが文を読み取るコピー / インクルード・ファイルの名前、またはマクロ定義の入ったマクロ・ライブラリの名前が入ります。*filename* が数字で始まる場合、二重引用符が必要です。filename は、完全パス名、相対パス名、パス情報の指定のないファイル名のどれでも構いません。アセンブラは、次の順序でファイルを検索します。

1) 現行のソース・ファイルが入っているディレクトリ。現行のソース・ファイルとは、.copy、.include、または .mlib のいずれかの疑似命令が見つかったときにアセンブル中だったファイルを指します。
2) アセンブラ・オプション -ɪ を使って指定したすべてのディレクトリ
3) 環境変数 C55X_A_DIR および A_DIR を使って設定したすべてのディレクトリ
4) 環境変数 C55X_C_DIR および C_DIR を使って設定したすべてのディレクトリ

アセンブラのディレクトリ検索アルゴリズムは、-ɪ アセンブラ・オプションや C55X_ADIR および A_DIR 環境変数を使うことにより、補強されています。

3.6.1 -ɪ アセンブラ・オプションの使用

-ɪ アセンブラ・オプションを使って、コピー / インクルード・ファイルやマクロ・ライブラリを含んだ代替ディレクトリを命名します。-ɪ オプションのフォーマットは次のとおりです。

```
cl55 -ɪpathname   source filename
```

各 -ɪ オプションは 1 つの pathname を指定します。指定できるパス数には制限がありません。アセンブリ・ソースでは、パス情報を指定せずに .copy、.include、または .mlib 疑似命令を使用できます。アセンブラは、現行のソース・ファイルがあるディレクトリでファイルが見つからない場合、-ɪ オプションで指定されたパスを検索します。

たとえば、source.asm という名前のファイルが現行のディレクトリにあり、source.asm には次の疑似命令文が含まれているとします。

```
.copy "copy.asm"
```

copy.asm ファイルのパスを以下のように仮定します。

Windows™　c:\tools\files\copy.asm

UNIX　/tools/files/copy.asm

O.S.	入力するコマンド
Windows	`cl55 -Ic:\tools\files source.asm`
UNIX	`cl55 -I/tools/files source.asm`

source.asm が現行ディレクトリにあるため、アセンブラは、まず現行ディレクトリで copy.asm を検索します。次にアセンブラは、-I オプションを使って指定されたディレクトリで検索します。

3.6.2　環境変数 C55X_A_DIR および A_DIR の使い方

環境変数は、ユーザー定義により文字列を割り当てるシステム・シンボルです。アセンブラは、環境変数を使ってコピー / インクルード・ファイル、またはマクロ・ライブラリを含む代替ディレクトリの名前を指定します。

アセンブラは C55X_A_DIR 環境変数を検索し、次にその環境変数を読み込んで処理します。この変数が見つからない場合、アセンブラは A_DIR 環境変数を読み込んで処理します。両方の変数が設定されている場合は、プロセッサ固有の変数の設定が使用されます。プロセッサ固有の変数は、異なるプロセッサのために弊社のツールを同時に使用している場合に便利です。

アセンブラは、C55X_A_DIR および A_DIR が見つからない場合には、C55X_C_DIR および C_DIR を検索します。

環境変数を割り当てるためのコマンド構文には、次のようなものがあります。

O.S.	入力するコマンド
Windows	**set A_DIR=** *pathname*$_1$**;***pathname*$_2$**;** . . .
UNIX (Bourne shell)	**set A_DIR** "*pathname*$_1$**;***pathname*$_2$**;** .. ."**; export A_DIR**

pathnames にはコピー / インクルード・ファイル、またはマクロ・ライブラリを含んだディレクトリが入ります。このパス名は、セミコロンまたは空白文字で区切ることができます。アセンブリ・ソースでは、パス情報を指定せずに .copy、.include、または .mlib 疑似命令を使用できます。アセンブラは、現行のソース・ファイルがあるディレクトリまたは -I オプションを使って指定したディレクトリでファイルが見つからない場合には、環境変数を使って指定したパスを検索します。

たとえば、source.asm というファイルに次のような文があるとします。

```
.copy "copy1.asm"
.copy "copy2.asm"
```

ファイルは次のディレクトリに保管されていると仮定します。

Windows　c:\tools\files\copy1.asm
　　c:\dsys\copy2.asm

UNIX　/tools/files/copy1.asm
　　/dsys/copy2.asm

次の表に示すコマンドを使用して、検索パスを設定します。

O.S.	入力するコマンド
Windows	`set A_DIR=c:\dsys` `cl55 -Ic:\tools\files source.asm`
UNIX (Bourne shell)	`A_DIR="/dsys";export A_DIR` `cl55 -I/tools/files source.asm`

source.asm が現行ディレクトリにあるため、アセンブラは、まず現行ディレクトリで copy1.asm と copy2.asm を検索します。次にアセンブラは、-I オプションを使って指定されたディレクトリで copy1.asm を検索します。最後にアセンブラは、A_DIR を使って指定されたディレクトリを検索して copy2.asm を検出します。

環境変数は、次にシステムをリブートするか次のコマンドを入力して変数をリセットするまでその設定は変わりません。

O.S.	入力するコマンド
Windows	`set A_DIR=`
UNIX	`unsetenv A_DIR`

3.7 ソース文のフォーマット

TMS320C55x アセンブリ言語ソース・プログラムは、アセンブラ疑似命令、アセンブリ言語命令、マクロ疑似命令、およびコメントを含むことのできるソース文で構成されます。ソース文の 1 行は、ソース・ファイル・フォーマットで使える範囲であればいくら長くても構いません。

ソース文の例を次に示します。

(a) ニーモニック命令

```
SYM1        .set      2              ; Symbol SYM1 = 2.
Begin:      MOV       #SYM1, AR1     ; Load AR1 with 2.
            .data
            .byte     016h           ; Initialize word (016h)
```

(b) 代数表記命令

```
SYM1        .set      2              ; Symbol SYM1 = 2.
Begin:      AR1 = #SYM1              ; Load AR1 with 2.
            .data
            .byte     016h           ; Initialize word (016h)
```

3.7.1 ソース文の構文

ソース文には、一定の順序で 4 つのフィールドを含めることができます。ソース文の一般的な構文は次のとおりです。

ニーモニックの構文 :
[*label*] [:] *mnemonic* [*operand list*] [;*comment*]

代数表記の構文 :
[*label*] [:] *instruction* [;*comment*]

次の規則に従って下さい。

- ❏ すべての文は必ず、ラベル、空白、アスタリスク、またはセミコロンのいずれかで始めなければなりません。
- ❏ アセンブラ疑似命令を含む文は、1 つの行で全体を指定する必要があります。
- ❏ ラベルの使用は任意です。使う場合には 1 カラム目から始めなければなりません。
- ❏ 各フィールドは、1 つ以上の空白で区切る必要があります。タブ文字は空白と同じ働きをします。
- ❏ コメントの使用は任意です。1 カラム目から始まるコメントはアスタリスクまたはセミコロン（* または ;）で始められますが、それ以外のカラムから始まるコメントは必ずセミコロンで始めなければなりません。
- ❏ ソース行は、最初の行をバックスラッシュ（\）文字で終了することにより、次の行へ続けることができます。

3.7.2 ラベル・フィールド

ラベルの使用は、すべてのアセンブリ言語命令、およびほとんどの（すべてではない）アセンブラ疑似命令で任意です。使用する場合は、ラベルは必ずソース文の 1 カラム目から始まらなければなりません。ラベルには、32 文字までの英数字（A-Z、a-z、0-9、_、および $）を使用できます。ラベルでは大文字と小文字が区別され、先頭に数字を使用することはできません。ラベルの後ろにコロン（:）を付けることはできますが、コロンはラベル名の一部としては処理されません。ラベルを使用しない場合は、最初の文字は空白か、セミコロンか、またはアスタリスクでなければなりません。

ラベルを使用する場合は、その値はセクション・プログラム・カウンタの現行の値となります（ラベルは、それが関連付けられている文を指します）。たとえば .word 疑似命令を使用していくつかのワードを初期化する場合は、疑似命令と同じ行のラベルはその最初のワードを指します。次の例では、ラベル Start の値は 40h となります。

```
5  000000               .data
6  000000  00         ; Assume other code was assembled.
7                            ...
8                            ...
9  000040  000A   Start:.word 0Ah,3,7
   000041  0003
   000042  0007
```

1 つの行がラベルだけでできていても、それは 1 つの有効な文です。セクション・プログラム・カウンタの現行の値がラベルに割り当てられます。

1 つの行にラベルだけがある場合は、そのラベルは次の行の命令のアドレスに割り当てられます（SPC はインクリメントされません）。

```
3    000043             Here:
4    000043 0003         .word 3
```

3.7.3 ニーモニック命令フィールド

ニーモニック・アセンブリでは、ラベル・フィールドの次にはニーモニック・フィールドとオペランド・リスト・フィールドが続きます。これらのフィールドについては、次の 2 つの節で説明します。

3.7.3.1 ニーモニック・フィールド

ラベル・フィールドの後ろにはニーモニック・フィールドが続きます。ニーモニック・フィールドは 1 カラム目から始めることはできません。1 カラム目から始める場合は、ラベルとして解釈されます。ニーモニック・フィールドには、次の命令コードのうち 1 つを含めることができます。

- ❏ 機械命令ニーモニック（たとえば、ABS、MPYU、STH）
- ❏ アセンブラ疑似命令（たとえば、.data、.list、.set）
- ❏ マクロ疑似命令（たとえば、.macro、.var、.mexit）
- ❏ マクロ・コール

3.7.3.2 オペランド・リスト・フィールド

オペランド・リスト・フィールドは、ニーモニック・フィールドに続くオペランドのリストです。オペランドとして使えるのは、定数（3.8 節「定数」（3-26 ページ）を参照）、シンボル（3.10 節「シンボル」（3-30 ページ）を参照）または式の中の定数とシンボルの組み合わせ（3.11 節「式」（3-36 ページ）を参照）です。オペランドとオペランドは、コンマで区切らなければなりません。

❑ 命令のオペランド接頭部

アセンブラでは、定数、シンボル、または式を、アドレス、即値、または間接値として使用することを指定できます。命令のオペランドには、次の規則があります。

- **# 接頭部 ― オペランドが即値であることを表します。**# 記号を接頭部として使用すると、アセンブラはオペランドを即値として処理します。これは、オペランドがレジスタまたはアドレスの場合にも当てはまります。この場合、アセンブラはアドレスの内容を使用するのではなく、アドレスを値として処理します。次の例は、オペランドを # 接頭部とともに使用する命令の例です。

  ```
  Label: ADD #123, AC0
  ```

  オペランド #123 は即値です。命令は、指定されたアキュムレータの内容に 123（10 進）を加算します。

  シフト・カウントが組み込まれた命令には、シフト・カウント・オペランドの # 接頭部が必要です。命令にシフトを実行させたい場合、シフト・カウントに # を使う必要があります。

- *** 接頭部 ― オペランドが間接アドレスであることを表します。*** 記号を接頭部として使用した場合、アセンブラは、オペランドを間接アドレスとして処理します。つまり、アセンブラはオペランドの内容をアドレスとして使用します。次の例は、オペランドを * 接頭部とともに使用する命令の場合です。

  ```
  Label: MOV *AR4,AC0
  ```

  オペランド *AR4 は間接アドレスを指定します。アセンブラは、レジスタ AR4 の内容により指定されたアドレスに行き、その位置の内容を指定されたアキュムレータに移動させます。

3.7.4 代数表記命令フィールド

代数表記アセンブリで、命令は代数数式に似たフォーマットで書き込まれます。命令の意味は、式の演算子で表されます。式の項は、実行されるオペランドを指定します。

次の項目は、代数表記構文において命令フィールドを使用する方法を示しています。

- ❑ 通常、オペランドはコンマで区切られていません。ただし、一部の代数表記命令はニーモニックとオペランドから構成されます。このような代数文では、オペランドを区切るためにコンマが使われます。たとえば、lms（Xmem、Ymem）のようになります。
- ❑ 単一のオペランドとして使用される複数の用語をもつ代数式は、小括弧で囲む必要があります。この規則は、関数呼び出しフォーマットを使用する文には適用されません。このような文は、すでに括弧で区切られているからです。たとえば、AC0 = AC1 & #(1 << sym) << 5 という式があるとします。1 << sym という式は 1 つのオペランドとして使われるので、括弧により区切られる必要があります。
- ❑ すべてのレジスタ名は、予約済みです。

3.7.5 コメント・フィールド

コメントはどのカラムからも始めることができ、ソース行の最後まで続きます。コメントには ASCII 文字と空白を使用できます。コメントはアセンブリ・ソース・リストには出力されますが、アセンブリ・オブジェクト・ファイルの内容には影響を与えません。

コメントのみを含むソース文も有効です。コメントを 1 カラム目から始める場合は、その先頭の文字にセミコロン（;）、またはアスタリスク（*）を使用できます。行の先頭以外の位置からコメントを始める場合は、最初の文字を必ずセミコロン（;）にしなければなりません。アスタリスク（*）は、1 カラム目で使われている場合にのみコメントとして識別されます。

3.8 定数

アセンブラは 7 つの型の定数をサポートします。

- ❏ 2 進整数
- ❏ 8 進整数
- ❏ 10 進整数
- ❏ 16 進整数
- ❏ 文字定数
- ❏ アセンブル時定数
- ❏ 浮動小数点定数

アセンブラは、各定数を 32 ビット数量として内部に保持しています。この定数値では、符号拡張は行われません。たとえば、定数 0FFH は 00FF（底 16）または 255（底 10）と同じです。-1 と同じではありません。

C55x 代数アセンブリ・ソース・コードでは、大半の定数は先頭が '#' でなければなりません。

3.8.1 2 進整数

2 進整数定数とは、32 桁までの 2 進数（0 と 1）に接尾部 B（または b）が付いた文字列です。32 桁未満の数値を指定した場合、アセンブラはその数値を右揃えとし、未指定の桁をゼロで埋めます。有効な 2 進定数の例を次に示します。

00000000B	0_{10} または 0_{16} に等しい定数
0100000b	32_{10} または 20_{16} に等しい定数
01b	1_{10} または 1_{16} に等しい定数
11111000B	248_{10} または $0F8_{16}$ に等しい定数

3.8.2 8 進整数

8 進整数定数は、0（ゼロ）という接頭部あるいは Q または q という接尾部が付いた最大 11 桁の 8 進数（0 ～ 7）の文字列です。有効な 8 進定数の例を次に示します。

10Q	8_{10} または 8_{16} に等しい定数
100000Q	32768_{10} または 8000_{16} に等しい定数
226q	150_{10} または 96_{16} に等しい定数

あるいは、8 進定数についての C 表記法を使用することもできます。

010	8_{10} または 8_{16} に等しい定数
0100000	32768_{10} または 8000_{16} に等しい定数
0226	150_{10} または 96_{16} に等しい定数

3.8.3 10 進整数

10 進整数定数とは 10 進数字で、-4294967296 から 4294967295 までの範囲です。これらは有効な 10 進定数の例です。

1000	1000_{10} または $3E8_{16}$ に等しい定数
-32768	-32768_{10} または 8000_{16} に等しい定数
25	25_{10} または 19_{16} に等しい定数

3.8.4 16 進整数

16 進整数定数とは、8 桁までの 16 進数に接尾部 H（または h）が付いた文字列です。16 進数には、0 ～ 9 の 10 進数と、A ～ F または a ～ f のアルファベットが使われます。16 進の定数は 10 進数（0 ～ 9）で始める必要があります。8 桁以下の 16 進数字が指定された場合は、アセンブラは、ビットの右揃えを行います。有効な 16 進定数の例を次に示します。

78h	120_{10} または 0078_{16} に等しい定数
0FH	15_{10} または $000F_{16}$ に等しい定数
37ACh	14252_{10} または $37AC_{16}$ に等しい定数

あるいは、16 進定数についての C 表記法を使用することもできます。

0x78	120_{10} または 0078_{16} に等しい定数
0x0F	15_{10} または $000F_{16}$ に等しい定数
0x37AC	14252_{10} または $37AC_{16}$ に等しい定数

3.8.5 文字定数

文字定数とは、単一引用符で囲まれた 4 文字までの文字列です。文字は、内部的に 8 ビットの ASCII 文字で表されます。文字定数の一部として使用する個々の単一引用符を示すには、単一引用符を 2 つ続けて使用します。2 つの単一引用符だけの文字定数も有効で、値 0 が割り当てられます。4 文字以下を指定した場合は、アセンブラは、ビットの右揃えを行います。有効な文字定数の例を次に示します。

'a'	内部的には 61_{16} として表示されます。
'C'	内部的には 43_{16} として表示されます。
"'D'	内部的には $2\ 744_{16}$ として表示されます。

文字定数と文字列は同じではないことに注意してください（文字列については、3.9 節「文字列」(3-29 ページ）で説明します)。文字定数が 1 つの整数値を表すのに対して、文字列は文字のリストです。

3.8.6 浮動小数点定数

浮動小数点定数は、10 進数の文字列で、後にオプションの小数点、小数部、および指数部が続きます。浮動小数点数の構文は、次のとおりです。

```
[ +|- ] [ nnn ] .[ nnn [ E|e [ +|- ] nnn ] ]
```

nnn を 10 進数の文字列に置き換えます。*nnn* の前に + または - を 1 つ付けることができます。小数点は必ず指定します。たとえば 3.e5 は有効ですが、3e5 は無効です。指数は 10 の累乗を示します。有効な定数の例を次に示します。

```
3.0
3.14
.3
-0.314e13
+314.59e-2
```

.double 疑似命令は、浮動小数点定数を IEEE 倍精度 64 ビット形式の浮動小数点値に変換します。.float 疑似命令は、浮動小数点定数を IEEE 単精度 32 ビット形式の浮動小数点値に変換します。.double 疑似命令および .float 疑似命令の詳細については、それぞれ 4-48 ページおよび 4-56 ページを参照してください。

3.9 文字列

文字列とは、二重引用符で囲まれた文字列です。文字列の中に二重引用符を使用する場合は、二重引用符を 2 個続けます。文字列の最大長は一定ではなく、文字列を必要とする疑似命令ごとに定義されます。文字は、内部的には 8 ビットの ASCII 文字で表されます。

有効な文字列の例を次に示します。

"sample program"	14 文字の文字列 *sample program* を定義しています。
"PLAN ""C"""	8 文字の文字列 *PLAN "C"* を定義しています。

文字列は次のように使用されます。

- .copy "filename" のようなファイル名
- .sect "section name" のようなセクション名
- .byte "charstring" のようなデータ定義疑似命令
- .string 疑似命令のオペランド

3.10 シンボル

シンボルは、ラベル、定数、および置換シンボルに使われます。シンボル名は、英数字（A-Z、a-z、0-9、$、および _）の文字列です。シンボルの最初の文字として数字を使用することはできません。また、シンボルに空白を埋め込むことはできません。定義するシンボルでは、大文字と小文字が区別されます。たとえばアセンブラは、ABC、Abc、abc を 3 つの異なったシンボルとして認識します。ただし、-c アセンブラ・オプションを使用すると大文字と小文字は区別されなくなります。この種類のシンボルは、.global 疑似命令を使用して外部シンボルとして宣言する場合以外は、それが定義されたアセンブル時にのみ有効です。

3.10.1 ラベル

シンボルをラベルとして使用すると、プログラム内の位置に関連付けられたシンボル・アドレスとなります。ファイルの中でローカルに使用されるラベルは、固有なものでなければなりません。

ラベルを .global、.ref、.def、または .bss 疑似命令のオペランドとして使用することもできます。次に例を示します。

```
            .global  label1

label2      nop
            ADD @label1,AC1,AC1
            B label2
```

予約語は、有効なラベル名ではありません。

3.10.2 シンボル定数

シンボルは定数値に設定できます。定数を使用すると、意味のある名前を定数値と同様に使用できます。.set および .struct/.tag/.endstruct 疑似命令を使用すると、定数をシンボル名に設定できます。シンボル定数を再定義することはできません。これらの疑似命令の使用方法を次に示します。

```
K       .set  1024              ;constant definitions
maxbuf  .set  2*K
value   .set  0
delta   .set  1

item    .struct                 ;item structure definition
        .int  value
        .int  delta
i_len   .endstruct              ;i_len=length of .struct (2)

array   .tag  item              ;array declaration
        .bss  array, i_len*K
```

アセンブラには事前定義されたシンボル定数もいくつかあります。このようなシンボル定数については、次の項で説明します。

3.10.3 シンボル定数の定義（-ad オプション）

-ad オプションは、定数値とシンボルを等価にできます。このオプションで定義したシンボルは、アセンブリ・ソースの中で値の代わりに使用できます。

-ad オプションのフォーマットは次のとおりです。

cl55 -ad*name*=[*value*]

name は定義するシンボルの名前です。*value* はシンボルに割り当てる値です。*value* を省略すると、シンボルは 1 に設定されます。

アセンブラ・ソース内では、次の疑似命令を使用してシンボルをテストできます。

テストのタイプ	疑似命令の使用法
存在する	.if $isdefed("*name*")
存在しない	.if $isdefed("*name*") = 0
値が等しい	.if *name* == *value*
値が等しくない	.if *name* != *value*

$isdefed ビルトイン関数への引数は、引用符で囲まなければならないことに注意してください。引用符で囲むことにより、引数は置換シンボルとしてでなく、そのままの文字列として解釈されます。

3.10.4 事前定義されたシンボル定数

アセンブラには、次のものを含めて事前定義されたシンボルがいくつかあります。

- セクション・プログラム・カウンタ（**SPC**）の現行の値を表示する **$**（ドル記号文字）
- **_ _large_model** は、使用するメモリ・モデルを指定します。デフォルトでは、値 は 0 です（小容量モデル）。-mk オプションを使用すると、このシンボルを 1 に設定します。このシンボルを使って、以下のようなメモリモデル独立コードを書くことができます。

```
.if  _ _large_model
AMOV #addr, XAR2 ; load 23-bit address
.else
AMOV #addr, AR2  ; load 16-bit address
.endif
```

大容量メモリ・モデルについての詳細は、TMS320C55x C コンパイラの最適化入門マニュアルを参照してください。

- **.TOOLS_v*n*** は、使用するアセンブラのバージョンを指定します。*n* の値は、アセンブラの見出しに表示されるバージョン番号を示します。たとえば、バージョンは 1.70 は .TOOLS_v170 のように指定されます。このシンボルを使い、アセンブラのバージョンに応じて条件付でアセンブルされるコードを書くことができます。

```
.if    $isdefed(".TOOLS_v170")
.word  0x110
.endif
.if    $isdefed(".TOOLS_v160")
.word  0x120
.endif
```

- アセンブラは、ユーザーが**メモリマップド・レジスタ**をすべて参照できるように、定義済みシンボルを設定します。

3.10.5 置換シンボル

シンボルは、文字列値（変数）を割り当てることができます。シンボル名を文字列と同様に扱うことにより、文字列にエイリアスを付けられるようになります。文字列を表すシンボルを置換シンボルと呼びます。アセンブラは、置換シンボルを見つけると、そのシンボルを文字列値に変えます。シンボル定数の場合と異なり、置換シンボルは再定義が可能です。

プログラム内のどこでも、文字列を置換シンボルに割り当てることができます。次に例を示します。

```
.asg  "errct",   AR2     ;register 2
.asg  "*+",      INC     ;indirect auto-increment
.asg  "*-",      DEC     ;indirect auto-decrement
```

マクロを使用するときに、置換シンボルは重要な役割を果たします。マクロ・パラメータとは、実際にはマクロ引数に割り当てられた置換シンボルだからです。次のコードは、置換シンボルがマクロでどのように使用されるかを示しています。

```
add2   .macro    ADDRA,ADDRB    ;add2 macro definition

       MOV ADDRA,AC0
       ADD ADDRB,AC0,AC0
       MOV AC0,ADDRB
       .endm

; add2 invocation
       add2 LOC1, LOC2

; the macro will be expanded as follows:
       MOV LOC1,AC0
       ADD LOC2,AC0,AC0
       MOV AC0,LOC2
```

マクロの詳細は、第 5 章「マクロ言語」を参照してください。

3.10.6 ローカル・ラベル

ローカル・ラベルは、一時的な有効範囲と効力をもつ特殊ラベルです。ローカル・ラベルの形式は、次のように 2 つあります。

- ❑ $*n*。*n* は、0 ～ 9 の範囲の 10 進数です。たとえば、$4 と $1 は有効なローカル・ラベルです。
- ❑ *name*?。*name* は、上述した任意の有効なシンボル名です。アセンブラは、疑問符を、ピリオドとそれに続く固有な番号に置き換えます。ソース・コードを展開しても、リスト・ファイル上でこの固有番号を参照することはできません。ラベルは、マクロ定義で行われたように疑問符が付いて表示されます。このラベルをグローバルとして宣言することはできません。

通常のラベルは、固有でなければならず（1 回しか宣言できません）、オペランド・フィールドで定数として使用できます。しかしローカル・ラベルの場合は、未定義にして再び定義したり、自動生成したりできます。ローカル・ラベルは、疑似命令を使用して定義することはできません。

ローカル・ラベルは、次のいずれかの方法で未定義に、またはリセットできます。

- ❑ .newblock 疑似命令を使います。
- ❑ セクションを変更します（.sect、.text または .data 疑似命令を使用）。
- ❑ インクルード・ファイル（.include または .copy 疑似命令で指定された）を入力します。
- ❑ インクルード・ファイルを去る（インクルード・ファイルの終わりに達する）。

例 3–3 は、ローカル・ラベルの $*n* 形式の使用例です。この例では、シンボル ADDRA、ADDRB、ADDRC がすでに定義されていると仮定しています。

例 3–3. $n ローカル・ラベル

(a) ローカル・ラベルを正しく使用したコード

```
Label1:     MOV ADDRA,AC0      ; Load Address A to AC0.
            SUB ADDRB,AC0,AC0 ; Subtract Address B.
            BCC $1,AC0 < #0    ; If < 0, branch to $1
            MOV ADDRB,AC0 ; otherwise, load ADDRB to AC0
            B $2           ; and branch to $2.
$1          MOV ADDRA,AC0      ; $1:load ADDRA to AC0.
$2          ADD ADDRC,AC0,AC0  ; $2:add ADDRC.
            .newblock     ; Undefine $1 so it can be used
                          ; again.
            BCC $1,AC0 < #0      ; If less than zero,
                                 ; branch to $1.
            MOV AC0,ADDRC     ; Store AC0 low in ADDRC.
$1          NOP
```

例 3–3. $n ローカル・ラベル（続き）

（b）ローカル・ラベルを不正に使用したコード

```
Label1:     MOV ADDRA,AC0
            SUB ADDRB,AC0,AC0
            BCC $1,AC0 < #0
            MOV ADDRB,AC0
            B $2
$1          MOV ADDRA,AC0
$2          ADD ADDRC,AC0,AC0
            BCC $1,AC0 < #0
            MOV AC0,ADDRC
$1          NOP           ; Wrong:$1 is multiply defined.
```

ローカル・ラベルは、マクロで使用すると特に便利です。通常のラベルが含まれているマクロが 2 回以上呼び出されると、アセンブラは複数定義エラーを発行します。しかし、マクロ内部にローカル・ラベルと .newblock を指定してあれば、マクロを展開するたびにそのローカル・ラベルが使用され、リセットされます。

一時点で有効にできる $*n* 形式のラベルの数は最大 10 個です。*name*? の形式のローカル・ラベルに制限はありません。ローカル・ラベルの定義を取り消した後に、同じラベルを再び定義して使用できます。ローカル・ラベルは、オブジェクト・コードのシンボル・テーブルには表示されません。

例 3–4 は、ローカル・ラベルの *name*? 形式の使用例です。

例 3–4. name? ローカル・ラベル

```
; First definition of local label 'mylab'
          nop
mylab?    nop
          B mylab?

; Include file has second definition of 'mylab'
          .copy "a.inc"

; Third definition of 'mylab',reset upon exit from include

mylab?    nop
          B mylab?

;  Fourth definition of 'mylab' in macro, macros use
;  different namespace to avoid conflicts

mymac     .macro
mylab?    nop
          B mylab?
          .endm

;  Macro invocation

          mymac

;  Reference to third definition of 'mylab', note that
;  definition is not reset by macro invocation nor
;  conflicts with same name defined in macro

          B mylab?

;  Changing section, allowing fifth definition of 'mylab'
          .sect "Secto_One"
          nop
          .data
mylab?    .int 0
          .text
          nop
          nop
          B mylab?
;.newblock directive, allowing sixth definition of 'mylab'
          .newblock
          .data
mylab?    .int 0
          .text
          nop
          nop
          B mylab?
```

3.11 式

式とは、定数、シンボル、または算術演算子により区切られた定数とシンボルです。オペランドは、アセンブリ時定数またはリンク時に再配置可能なシンボルです。有効な式の値の範囲は、-4294967296 から 4294967295 です。式の計算順序に影響を与える主要な 3 つの要素は、次のとおりです。

括弧

括弧で囲まれた式は、必ず最初に計算します。

8 /（4 / 2）= 4、ただし 8 / 4 / 2 = 1 です。

この丸括弧の代わりに中括弧（{ }）や大括弧（ [ ] ）を使うことはできません。

優先順位グループ

C55x アセンブラは、C 言語と同じ優先順位を使用します（表 3-1 で概要を説明）。これは、他の TMS320 アセンブラの優先順位とは異なります。括弧により式の計算順位が決められていない場合、優先順位の最も高い演算子が最初に計算されます。

8 + 4 / 2 = 10 （4 / 2 が最初に計算されます。）

左から右へ

括弧および優先順位グループにより式の計算順位が決められていない場合、式は C 言語で処理されるのと同様に左から右への順で計算されます。

8 / 4*2 = 4、ただし 8 /（4*2）= 1 です。

3.11.1 演算子

表 3–1 では、式で使える演算子について説明しています。

関係演算子を未定義のシンボルに適用すると、シンボルの参照に、ブール式のために値 0 を割り当てます。

表 3–1. 式で使用できる演算子（優先順位）

シンボル	演算子	計算
+ - ~ !	単項プラス、単項マイナス、1 の補数	右から左へ
* / %	乗算、除算、剰余	左から右へ
+ -	加算、減算	左から右へ
<< >>	左シフト、右シフト	左から右へ
< <= > >=	小さい、小さいか等しい、大きい、大きいか等しい	左から右へ
!=, =[=]	等しくない、等しい	左から右へ
&	ビット単位 AND	左から右へ
^	ビット単位 XOR	左から右へ
|	ビット単位 OR	左から右へ

注： 単項の +、-、および * は、2 項演算形式よりも優先順位が高くなります。

3.11.2 式のオーバーフローとアンダーフロー

アセンブラは、アセンブル時に算術演算が行われると、オーバーフローとアンダーフローの条件をチェックします。アセンブラは、オーバーフローやアンダーフローが発生すると、必ず Value Truncated の警告を発行します。アセンブラは、乗算の場合のオーバーフローとアンダーフローはチェックしません。

3.11.3 整合定義式

アセンブラ疑似命令の中には、整合定義式をオペランドとして求めるものがあります。整合定義式には、その式を検出する前にすでに定義されているシンボルまたはアセンブル時定義のみが含まれます。整合定義式の計算は、絶対でなければなりません。

例 3–5. 整合定義式

```
        .data
label1  .word  0
        .word  1
        .word  2
label2  .word  3

X       .set   50h

goodsym1.set   100h + X         :Because value of X is defined before
                                ; referenced, this is a valid well-defined
                                ; expression

goodsym2.set   $                ; All references to previously defined local
goodsym3.set   label1           :labels, including the current SPC ($), are
                                ; considered to be well-defined.

goodsym4.set   label2 - label1  ; Although label1 and label2 are not
                                ; absolute symbols, because they are local
                                ; labels defined in the same section, their
                                ; difference can be computed by the assembler.
                                ; The difference is absolute, so the
                                ; expression is well-defined.
```

3.11.4 条件式

アセンブラでは、再配置可能なリンク時オペランドを除き、すべての式で使える関係演算子がサポートされています。関係演算子は、条件付きアセンブリを行うときに特に便利です。関係演算子は、未定義シンボルにも再配置可能なシンボル適用できます。条件式のコンテキストにおいて、未定義シンボルは値 0 に置き換えられます。関係演算子には次のようなものがあります。

=	等しい	==	等しい
!=	等しくない		
<	小さい	<=	小さいか等しい
>	大きい	>=	大きいか等しい

条件式の計算は、真の場合は 1、偽の場合は 0 になります。条件式が使用できるのは、オペランドのタイプが等価の場合だけです。たとえば、絶対値を絶対値と比較することはできますが、絶対値を再配置可能値と比較することはできません。

3.12 ビルトイン関数

アセンブラは、変換およびさまざまな数学計算を行うためのビルトイン関数をサポートしています。表 3–2 に、ビルトイン関数を示します。*expr* は定数値でなければならないことに注意してください。アセンブラの非数学ビルトイン関数については、表 5–1 を参照してください。

表 3–2. アセンブラ・ビルトイン数学関数

関数	説明
$acos(*expr*)	*expr* のアーク・コサインを浮動小数点値として返します。
$asin(*expr*)	*expr* のアーク・サインを浮動小数点値として返します。
$atan(*expr*)	*expr* のアーク・タンジェントを浮動小数点値として返します。
$atan2(*expr*)	*expr* のアーク・タンジェントを、浮動小数点値（-π ～ π）として返します。
$ceil(*expr*)	式よりも小さくない最小の整数を返します。
$cosh(*expr*)	*expr* のハイパボリック・コサインを浮動小数点値として返します。
$cos(*expr*)	*expr* のコサインを浮動小数点値として返します。
$cvf(*expr*)	*expr* を浮動小数点値に変換します。
$cvi(*expr*)	*expr* を整数値に変換します。
$exp(*expr*)	e^{expr} を返します。
$fabs(*expr*)	*expr* の絶対値を浮動小数点値として返します。
$floor(*expr*)	式よりも大きくない最大の整数を返します。
$fmod(*expr1*, *expr2*)	*expr1* および *expr2* を割った剰余を返します。
$int(*expr*)	*expr* の結果が整数である場合に 1 を返します。
$ldexp(*expr1, expr2*)	*expr1* x 2 *expr2* を返します。
$log10(*expr*)	*expr* の基数を 10 とした対数を返します。
$log(*expr*)	*expr* の自然対数を返します。
$max(*expr1, expr2*)	2 つの式の最大を返します。
$min(*expr1, expr2*)	2 つの式の最小を返します。

表 3–2. アセンブラ・ビルトイン数学関数（続き）

関数	説明
$pow(*expr1, expr2*)	*expr1*^{expr2} を返します。
$round(*expr*)	最も近い整数に丸められた *expr* の結果を返します。
$sgn(*expr*)	*expr* の符号を返します。
$sin(*expr*)	*expr* のサインを浮動小数点値として返します。
$sinh(*expr*)	*expr* のハイパボリック・サインを浮動小数点値として返します。
$sqrt(*expr*)	*expr* の平方根を浮動小数点値として返します。
$tan(*expr*)	*expr* のタンジェントを浮動小数点値として返します。
$tanh(*expr*)	*expr* のハイパボリック・タンジェントを浮動小数点値として返します。
$trunc(*expr*)	ゼロの方向に丸められた *expr* の結果を返します。

3.13 ソース・リスト

ソース・リストは、ソース文とソース文が作成するオブジェクト・コードを示します。リスト・ファイルを取得するには、オプション -l（小文字の L）を付けてアセンブラを起動します。

各ソース・リスト・ページの上部には、2 つの見出し行、空白行、およびタイトル行があります。.title 疑似命令で指定されるすべてのタイトルは、このタイトル行に出力されます。ページ番号はタイトルの右側に出力されます。.title 疑似命令を使用しない場合は、ソース・ファイル名が出力されます。アセンブラは、タイトル行の下に空白行を挿入します。

ソース・ファイルの各行に対して、リスト・ファイルに、ソース文番号、SPC 値、アセンブルされたオブジェクト・コード、ソース文を示す行が生成されます。1 つのソース文が 2 ワード以上のオブジェクト・コードを生成することもあります。アセンブラは、1 ワードを追加するたびに SPC 値とオブジェクト・コードを含む別の行を生成します。追加される各行は、ソース文の行のすぐ下に置かれます。

フィールド 1: ソース文番号

行番号

ソース文番号は 10 進数です。アセンブラは、ソース・ファイルでソース行を検出するたびに番号を付けます。文の中には、行カウンタを増やしてもリストには入れられないものもあります（たとえば、.title 文や .nolist に続く文は出力されません）。2 つの連続するソース行番号の差は、ソース・ファイル内にリストされない文の数を表します。

インクルード・ファイル文字

アセンブラは行の前に文字を置くことがあります。この文字は、その行がインクルード・ファイルからアセンブルされることを示します。

ネスト・レベル数

アセンブラは、行の前に数字を置くことがあります。この数字は、マクロ展開またはループ・ブロックのネスト・レベルを示します。

フィールド 2: セクション・プログラム・カウンタ

このフィールドには、セクション・プログラム・カウンタ（SPC）の値（16 進数）が入ります。各セクション（.text、.data、.bss、名前付きセクション）には、すべて別々の SPC があります。疑似命令の中には SPC に影響を与えないものもあります。その場合は、このフィールドは空のままです。

フィールド 3: オブジェクト・コード

このフィールドには、16 進数で表したオブジェクト・コードが入ります。すべての機械命令と疑似命令は、このフィールドにオブジェクト・コードを出力します。また、フィールドの最後に次のいずれかの文字を追加して、再配置の種類を示します。

!	未定義の外部参照
'	.text 再配置可能
"	.data 再配置可能
+	.sect 再配置可能
-	.bss、.usect 再配置可能
%	複合再配置式

フィールド 4: ソース文フィールド

このフィールドには、アセンブラがスキャンしたソース文の文字が入ります。このフィールドの幅は、ソース文の幅指定により決まります。

例 3–6 は、4 つのフィールドのそれぞれが判別できるアセンブラ・リストを示したものです。

例 3–6. アセンブラ・リスト

(a) ニーモニックの例

```
       1                            .global RSET, INT0, INT1, INT2
       2                            .global TINT, RINT, XINT, USER
       3                            .global ISR0, ISR1, ISR2
       4                            .global time, rcv, xmt, proc
       5
       6                    initmac  .macro
       7                    ;* initialize macro
       8                            BSET #9,ST1_55  ;disable overflow
       9                            MOV #0,DP       ;set dp
      10                            MOV #55,AC0     ;set AC0
      11                            BCLR #11,ST1_55 ;enable ints
      12                            .endm
      13                    ****************************************
      14                    *       Reset and interrupt vectors       *
      15                    ****************************************
      16   000000                   .sect "rset"
      17   000000  6A00     RSET:    B init
           000002  0010+
      18   000004  6A00     INT0:    B ISR0
           000006  0000!
      19   000008  6A00     INT1:    B ISR1
           00000a  0000!
      20   00000c  6A00     INT2:    B ISR2
           00000e  0000!
      21
      22                    *
      23   000000                   .sect "ints"
      24   000000  6A00     TINT     B time
           000002  0000!
      25   000004  6A00     RINT     B rcv
           000006  0000!
      26   000008  6A00     XINT     B xmt
           00000a  0000!
      27   00000c  6A00     USER     B proc
           00000e  0000!
      28                    ****************************************
      29                    *         Initialize processor.         *
      30                    ****************************************
      31   000010           init:    initmac
1                           * initialize macro
1          000010  4693              BSET #9,ST1_55
           000012  7800              MOV #0,DP
           000014  0000
1          000016  7600              MOV #55,AC0
           000018  3708
1          00001a  46B2              BCLR #11,ST1_55
```

フィールド 1 フィールド 2 フィールド 3 フィールド 4

例 3–6. アセンブラ・リスト（続き）

（b） 代数表記の例 :

```
       1                            .global RSET, INT0, INT1, INT2
       2                            .global TINT, RINT, XINT, USER
       3                            .global ISR0, ISR1, ISR2
       4                            .global time, rcv, xmt, proc
       5
       6                    initmac  .macro
       7                    ;* initialize macro
       8                            bit(ST1, #ST1_SATD) = #1 ;disable oflow
       9                            DP = #((01FFH & 0) << 7) ;set dp
      10                            AC0 = #55                ;set AC0
      11                            bit(ST1, #ST1_INTM) = #0 ;enable ints
      12                            .endm
      13                    ****************************************
      14                    *       Reset and interrupt vectors      *
      15                    ****************************************
      16   000000                   .sect "rset"
      17   000000 6A00      RSET:    goto  #(init)
           000002 0010+
      18   000004 6A00      INT0:    goto  #(ISR0)
           000006 0000!
      19   000008 6A00      INT1:    goto  #(ISR1)
           00000a 0000!
      20   00000c 6A00      INT2:    goto  #(ISR2)
           00000e 0000!
      21
      22                    *
      23   000000                   .sect "ints"
      24   000000 6A00      TINT     goto  #(time)
           000002 0000!
      25   000004 6A00      RINT     goto  #(rcv)
           000006 0000!
      26   000008 6A00      XINT     goto  #(xmt)
           00000a 0000!
      27   00000c 6A00      USER     goto  #(proc)
           00000e 0000!
      28                    ****************************************
      29                    *         Initialize processor.        *
      30                    ****************************************
      31   000010           init:    initmac
1                           * initialize macro
1          000010 4693              bit(ST1, #ST1_SATD) = #1
1          000012 7800              DP = #((01FFH & 0) << 7)
           000014 0000
1          000016 7600              AC0 = #55
           000018 3708
1          00001a 46B2              bit(ST1, #ST1_INTM) = #0
```

フィールド 1 フィールド 2 フィールド 3 フィールド 4

3.14 アセンブリ・ソースのデバッグ方法

アセンブリ・ファイルのコンパイル中に -g を使って CL55 を起動すると、アセンブラはシンボリック・デバッグ情報を提供します。これにより、ユーザーは Code Composer Studio の Disassembly ウィンドウを使わずに、デバッガのアセンブリ・コードを使って操作することができます。これにより、デバッグ中にソース・コメントおよびその他のソース・コード注釈を表示できます。

.asmfunc および .endasmfunc 疑似命令により、アセンブリ・コード内で C の特性を使うことができます。この特性により、アセンブリ・ファイルのデバッグ方法は、C/C++ ソース・ファイルのデバッグに類似します。

.asmfunc および .endasmfunc 疑似命令（4-32 ページを参照）により、ユーザーのコードの特定の領域を指定でき、これらの領域を C 関数としてデバッガに表示できます。.asmfunc および .endasmfunc 疑似命令により囲まれていないアセンブリ・コードの連続したセクションは、アセンブラ定義関数に自動的に配置されます。この関数は次の構文によって指定されます。

$*filename***:***starting source line***:***ending source line***$**

ユーザーの変数を C コードのユーザー定義タイプとして表示したい場合、C ファイルでそのタイプを宣言し、変数を定義する必要があります。C ファイルは、.ref 疑似命令を使ってアセンブリ・コードで参照できます（4-57 ページを参照）。

例 3–7 は、構造体 X として変数 svar を定義する cvar.c の C プログラムを示しています。svar 変数は、addfive.asm アセンブリ・プログラムで参照され、svar の 2 番目のデータ・メンバに 5 が加算されます。

例 3–7. C タイプとしてのアセンブリ変数の表示

（a）C プログラム cvar.c

```
typedef struct
{
   int m1;
   int m2;
} X;

X svar = { 1, 2 };
```

例 3–7. C タイプとしてのアセンブリ変数の表示（続き）

（b）アセンブリ・プログラム addfive.asm

```
;-------------------------------------------------------------------------------
; Tell the assembler we're referencing variable "_svar", which is defined in
; another file (cvars.c).
;-------------------------------------------------------------------------------
          .ref _svar

;-------------------------------------------------------------------------------
; addfive() - Add five to the second data member of _svar
;-------------------------------------------------------------------------------
          .text
          .align 4
          .global addfive
addfive:  .asmfunc
           ADD #5, *abs16(#(_svar+1)) ; add 5 to svar.m2
           RET                        ; return from function
          .endasmfunc
```

-g オプションを使って両方のソース・ファイルをコンパイルし、以下のようにリンクします。

```
cl55 -g cvars.c addfive.asm -z -l=lnk.cmd -l=rts55.lib -o=addfive.out
```

このプログラムを 1 つのシンボリック・デバッガにロードすると、addfive が C 関数として表示されます。通常の C 変数の場合と同様、主な操作の実行中に svar の値をモニタできます。

3.15 クロスリファレンス・リスト

クロスリファレンス・リストには、シンボルとその定義が示されます。クロスリファレンス・リストを取得するには、アセンブラを起動するときに -ax オプションまたは .option 疑似命令を使用します。アセンブラは、クロスリファレンスをソース・リストの最後に付け加えます。

アセンブラが .include 疑似命令を含むアセンブリ・ファイルにクロスリファレンス・リストを生成するとき、シンボルが定義／参照されているインクルード・ファイルと行番号の記録が取られます。これは、インクルード・ファイルごとに、文字参照（A、B、C など）を割り当てることにより行われます。この文字は、.include 疑似命令がアセンブリ・ソース・ファイルに出現する順に割り当てられます。

例 3–8. クロスリファレンス・リストのサンプル

```
LABEL                           VALUE     DEFN      REF

INT0                            000004+   25        5
INT1                            000008+   27        5
INT2                            00000c+   29        5
ISR0                            REF               9     25
ISR1                            REF               9     27
ISR2                            REF               9     29
RINT                            000004+   37        7
RSET                            000000+   23        5
TINT                            000000+   35        7
XINT                            000008+   39        7
init                            000010+   45        23
```

LABEL このカラムには、アセンブル時に定義または参照された各シンボルが入ります。

VALUE このカラムには、シンボルに割り当てられた値またはシンボルの属性を表す名前を表す 4 桁の 16 進数が入ります。値の後には、シンボルの属性を示す文字が続くこともあります。表 3–3 に、このような文字と名前を掲載しています。

Definition （DEFN）このカラムには、シンボルを定義する文の番号が入ります。未定義のシンボルの場合には、このカラムは空欄となります。

Reference （REF）このカラムには、シンボルを参照している文の番号がリストされます。このカラムが空欄であれば、そのシンボルがまだ使われていないことを示します。

表 3–3.　シンボルの属性

文字または名前	意味
REF	外部参照（.global シンボル）
UNDF	未定義
'	.text セクションで定義されているシンボル
"	.data セクションで定義されているシンボル
+	.sect セクションで定義されているシンボル
-	.bss または .usect セクションで定義されているシンボル

第 4 章

アセンブラ疑似命令

アセンブラ疑似命令は、プログラム・データを提供し、アセンブリ・プロセスを制御します。アセンブラ疑似命令を使用すると、次の操作を行うことができます。

- ❑ コードおよびデータを、指定したセクションにアセンブルする。
- ❑ メモリに初期化されない変数のための空間を確保する。
- ❑ リスト上の表示を制御する。
- ❑ メモリの初期化を行う。
- ❑ 条件付きブロックをアセンブルする。
- ❑ グローバル変数を宣言する。
- ❑ アセンブラがマクロを取得するためのライブラリを指定する。
- ❑ シンボリック・デバッグ情報を提供する。

本章は 2 つの部分に分かれています。第 1 部（4.1 節から 4.11 節）では疑似命令を機能別に説明し、第 2 部（4.12 節以降）では疑似命令のリファレンスをアルファベット順に掲載しています。

項目 **ページ**

4.1 疑似命令のまとめ

この節では、アセンブラ疑似命令をまとめています。

アセンブラ疑似命令およびそのパラメータは、1 つの行で全体を指定する必要があります。

ここで説明しているアセンブラ疑似命令の他に、TMS320C55x™ ソフトウェアのツールは以下のような疑似命令もサポートしています。

- ❑ アセンブラは、いくつかのマクロ用の疑似命令を使用します。本章にマクロ疑似命令のリストがありますが、詳細は第 5 章「マクロ言語」で説明します。
- ❑ 絶対リスタも疑似命令を使用します。絶対リスト疑似命令（.system および .setsect）は、ユーザー入力による命令ではなく、絶対リスタによりソース・プログラムに挿入されます。これらの疑似命令については、本章ではなく第 10 章「絶対リスタの説明」で説明します。
- ❑ C/C++ コンパイラは、シンボリック・デバッグ用の疑似命令を使用します。他の疑似命令とは違い、ほとんどのアセンブリ言語プログラムではシンボリック・デバッグ疑似命令を使用しません。これらの疑似命令については、本章ではなく付録 B「シンボリック・デバッグ疑似命令」で説明します。DWARF デバッグ疑似命令は、.dwattr、.dwcfa、.dwcie、.dwendentry、.dwendtag、.dwfde、.dwpsn、および .dwtag です。

注： 構文内のラベルとコメント

ほとんどの場合、疑似命令を含むソース文にはラベルとコメントも含める場合があります。ラベルは最初のカラムから始まります（コメントを除けば、ラベルは最初のカラムに表示される唯一の要素です）。コメントの前にはセミコロンを付けます。また、コメントのみの行の場合は、先頭にアスタリスクを付ける必要があります。読みやすさを考慮して、ラベルとコメントは疑似命令の構文には示していません。ただし、一部の疑似命令の場合はラベルが必要であり、構文に示されます。

表 4–1. 疑似命令のまとめ

(a) セクションに関する疑似命令

ニーモニックと構文	説明	ページ
.bss *symbol*, *size in words* [*, blocking*] [**,** *alignment*]	*size* に指定されたワード数を、.bss（初期化されていないデータ）セクション内に確保します。	4-34
.clink ["*section name*"]	現行のセクションまたは指定のセクション用に対する条件付きリンクを可能にします。	4-39
.data	.data（初期化されたデータ）セクションにアセンブルします。	4-47
.sect "*section name*"	名前付き（初期化された）セクションにアセンブルします。	4-82
.text	.text（実行可能コード）セクションにアセンブルします。	4-93
symbol **.usect** "*section name*", *size in words* [**,** *blocking*] [,*alignment*]	*size* に指定されたワード数を名前付き（初期化されていない）セクション内に確保します。	4-97

(b) データを定義する疑似命令

ニーモニックと構文	説明	ページ
.byte $value_1$ [**,** ..., $value_n$]	現行のセクション内の 1 つ以上の連続したバイトまたはワードを初期化します。	4-37
.char $value_1$ [**,** ..., $value_n$]	現行のセクション内の 1 つ以上の連続したバイトまたはワードを初期化します。	4-37
.double $value_1$ [**,** ..., $value_n$]	1 つ以上の 64 ビット、IEEE 倍精度、浮動小数点定数を初期化します。	4-48
.field *value* [**,** *size in bits*]	可変長フィールドを初期化します。	4-54
.float *value* [**,** ..., $value_n$]	1 つ以上の 32 ビット、IEEE 単精度、浮動小数点定数を初期化します。	4-56
.half $value_1$ [**,** ..., $value_n$]	1 つ以上の 16 ビット整数を初期化します。	4-60
.int $value_1$ [**,** ..., $value_n$]	1 つ以上の 16 ビット整数を初期化します。	4-63
label: **.ivec** [*address* [, *stack mode*]]	割り込みベクター・テーブルの入力を初期化します。	4-64
.ldouble $value_1$ [**,** ..., $value_n$]	1 つ以上の 64 ビット、IEEE 倍精度、浮動小数点定数を初期化します。	4-48
.long $value_1$ [**,** ..., $value_n$]	1 つ以上の 32 ビット整数を初期化します。	4-71
.pstring "$string_1$" [, ...,"$string_n$"]	1 つ以上のパックされたテキスト文字列を初期化します。	4-88
.short $value_1$ [**,** ..., $value_n$]	1 つ以上の 16 ビット整数を初期化します。	4-60

表 4–1. 疑似命令のまとめ（続き）

(b) データを定義する疑似命令（続き）

ニーモニックと構文	説明	ページ
.space *size in bits;*	*size in bits* に指定されたビット数を現行のセクション内に確保します。ラベルは、確保された空間の始まりを指します。	4-84
.string "*string*$_1$**" [,** ..., "*string*$_n$**"]**	1 つ以上のテキスト文字列を初期化します。	4-88
.ubyte *value*$_1$ **[,** ...**,** *value*$_n$**]**	現行のセクション内の 1 つ以上の符号なしの連続したバイトまたはワードを初期化します。	4-37
.uchar *value*$_1$ **[,** ...**,** *value*$_n$**]**	現行のセクション内の 1 つ以上の符号なしの連続したバイトまたはワードを初期化します。	4-37
.uhalf *value*$_1$ **[,** ...**,** *value*$_n$**]**	1 つ以上の符号なしの 16 ビット整数を初期化します。	4-60
.uint *value*$_1$ **[,** ...**,** *value*$_n$**]**	1 つ以上の符号なしの 16 ビット整数を初期化します。	4-63
.ulong *value*$_1$ **[,** ...**,** *value*$_n$**]**	1 つ以上の符号なしの 32 ビット整数を初期化します。	4-71
.ushort *value*$_1$ **[,** ...**,** *value*$_n$**]**	1 つ以上の符号なしの 16 ビット整数を初期化します。	4-60
.uword *value*$_1$ [, ..., *value*$_n$**]**	1 つ以上の符号なしの 16 ビット整数を初期化します。	4-63
.word *value*$_1$ [, ..., *value*$_n$**]**	1 つ以上の 16 ビット整数を初期化します。	4-63
.xfloat *value*$_1$ [, ...,*value*$_n$**]**	1 つ以上の 32 ビット、IEEE 単精度、浮動小数点定数を初期化しますが、倍長ワード境界での位置合わせを行いません。	4-56
.xlong *value*$_1$ [, ...,*value*$_n$**]**	1 つ以上の 32 ビット整数を初期化しますが、倍長ワード境界での位置合わせを行いません。	4-71

(c) 位置合わせに影響する疑似命令

ニーモニックと構文	説明	ページ
.align [*size*]	SPC を、パラメータの指定するバイト境界またはワード境界に位置合わせします。パラメータは 2 の累乗でなくてはならず、デフォルトは 128 バイト境界または 128 ワード境界です。	4-28
.even	.align 2 と等価です。	4-28
.localalign	最大の localrepeat ループ・サイズを許容するローカル・リピート・ブロックの開始位置合わせ	4-69
.sblock ["]*section name*["] [, ... , "*section name*"]	ブロック化するセクションを指定します。	4-81

表 4–1. 疑似命令のまとめ（続き）

(d) 出力リストを制御する疑似命令

ニーモニックと構文	説明	ページ
.drlist	すべての疑似命令行のリスト出力を可能にします(デフォルト)。	4-49
.drnolist	特定の疑似命令行のリスト出力を抑止します。	4-49
.fclist	偽の条件付きコード・ブロックのリスト出力を許可します（デフォルト)。	4-53
.fcnolist	偽の条件付きコード・ブロックのリスト出力を抑止します。	4-53
.length *page length*	ソース・リストのページの長さを設定します。	4-67
.list	ソース・リストの出力を再開します。	4-68
.mlist	マクロ・リストとループ・ブロックの出力を許可します（デフォルト)。	4-76
.mnolist	マクロ・リストとループ・ブロックの出力を抑止します。	4-76
.nolist	ソース・リストの出力を停止します。	4-68
.option {**B** | **L** | **M** | **R** | **T** | **W** | **X**}	出力リスト・オプションを選択します。	4-79
.page	ソース・リストのページ替えを行います。	4-80
.sslist	置換シンボルの展開リスト出力を許可します。	4-85
.ssnolist	置換シンボルの展開リスト出力を抑止します（デフォルト)。	4-85
.tab *size*	タブ・サイズを設定します。	4-92
.title "*string***"**	リストのページ見出しに指定のタイトルを出力します。	4-94
.width *page width*	ソース・リストのページの幅を設定します。	4-67

(e) 他のファイルを参照する疑似命令

ニーモニックと構文	説明	ページ
.copy ["]*filename*["]	他のファイルからソース文をインクルードします。コピーされたファイルはリストに示されます。	4-40
.include ["]*filename*["]	他のファイルからソース文をインクルードします。インクルードされたファイルはリストに示されません。	4-40

表 4–1. 疑似命令のまとめ（続き）

(f) シンボルに関連する疑似命令

ニーモニックと構文	説明	ページ
.def *symbol*$_1$ [**,** ..., *symbol*$_n$]	現行のモジュール内に定義されていて、他のモジュールで使用される 1 つ以上のシンボルを特定します。	4-57
.global *symbol*$_1$ [**,** ..., *symbol*$_n$]	1 つ以上のグローバル(外部)シンボルを特定します。	4-57
.ref *symbol*$_1$ [**,** ..., *symbol*$_n$]	現行のモジュールで使用されているが、他のモジュールで定義される 1 つ以上のシンボルを特定します。	4-57

(g) 条件付きアセンブリを制御する疑似命令

ニーモニックと構文	説明	ページ
.break [*conditional or boolean expression*]	条件が真の場合、.loop アセンブリを終了します。.break の使用はオプションです。	4-72
.else	.if 条件が偽の場合、コード・ブロックをアセンブルします。.else の使用はオプションです。この疑似命令は、デフォルトの場合として条件付きブロックで使用できます。	4-61
.elseif *conditional or boolean expression*	.if 条件が偽で、なおかつ .elseif 条件が真の場合、コード・ブロックをアセンブルします。.elseif の使用はオプションです。	4-61
.endif	.if コード・ブロックを終了します。	4-61
.endloop	.loop コード・ブロックを終了します。	4-72
.if *boolean expression*	条件が真の場合、コード・ブロックをアセンブルします。	4-61
.loop [*well-defined expression*]	コード・ブロックの反復アセンブリを開始します。*well-defined expression* はループ・カウントです。	4-72

表 4–1. 疑似命令のまとめ（続き）

(h) マクロを定義する疑似命令

ニーモニックと構文	説明	ページ
symbol **.macro**[*macro parameters*]	ソース文がマクロ定義の第 1 行目であることを特定します。マクロは *symbol.* を使って起動できます。	4-73
.mlib ["]*filename*["]	指定のマクロ・ライブラリを現行のソース・ファイルで使用できるようにします。	4-74
.mexit	.endm に移動します。この疑似命令は、エラーのテストでマクロ展開の失敗が確認された場合に使用すると便利です。	5-3
.endm	.macro 定義を終了します。	4-52
.var	ローカルなマクロ置換シンボルを定義します。	4-100

† マクロ疑似命令の詳細は、第 5 章「マクロ言語」を参照してください。

(i) アセンブル時にシンボルを定義する疑似命令

ニーモニックと構文	説明	ページ
.asg ["]*character string*["], *substitution symbol*	置換シンボルに文字列を割り当てます。	4-30
.cstruct	C 構造体の定義を開始します。	4-44
.cunion	C 共用体の定義を開始します。	4-45
.endstruct	構造体の定義を終了します。	4-44, 4-89
.endunion	共用体の定義を終了します。	4-44, 4-95
.equ	値をシンボルと等価にします。	4-83
.eval *well-defined expression*, *substitution symbol*	数値置換シンボルの算術計算を実行します。	4-30
.label *symbol*	セクションのロード時の位置を参照する再配置可能なシンボルを定義します。	4-66
.set	値をシンボルと等価にします。	4-83
.struct	構造体の定義を開始します。	4-89
.tag	構造体属性をラベルに割り当てます。	4-89
.union	共用体の定義を開始します。	4-95

表 4–1. 疑似命令のまとめ（続き）

(j) 実行時環境の詳細をアセンブラに通知する疑似命令

ニーモニックと構文	説明	ページ
.dp *DP_value*	DP レジスタの値を指定します。	4-49
.lock_off	lock() 修飾子を無効とし、デフォルトの動作を再開します。	4-71
.lock_on	リード・モディファイ・ライト命令を含むコード・ブロックの始まりを特定します。	4-71
.vli_off	アセンブラが特定の可変長命令の最大書式を使用するコード・ブロックの始まりを特定します。	4-101
.vli_on	可変長命令を最小書式に分解するときのデフォルト動作を再開します。	4-101

(k) C55x アドレッシング・モードに関連する疑似命令

ニーモニックと構文	説明	ページ
.arms_off	間接メモリ・アクセス修飾子を使ってアセンブラのデフォルト動作を再開します。	4-29
.arms_on	ARMS モードでアセンブルするコード・ブロックの始まり特定します。	4-29
.c54cm_off	C55x コードのデフォルト動作を再開します。	4-38
.c54cm_on	C54x 互換モード・コード（C54x コードから変換されたコード）のブロックの開始を特定します。	4-38
.cpl_off	DP 相対 dma のデフォルト動作を再開します。	4-42
.cpl_on	CPL モードでアセンブルするコード・ブロックの始まりを特定します（SP 相対 dma）。	4-42

表 4–1. 疑似命令のまとめ（続き）

(l) C54x ニーモニック・アセンブリの移植に影響する疑似命令

ニーモニックと構文	説明	ページ
.port_for_size	C54x コードをより小さいサイズに最適化するデフォルト動作を再開します。	4-81
.port_for_speed	アセンブラが移植された C54x コードのスピードを最適化するコード・ブロックの始まりを特定します。	4-81
.sst_off	アセンブラが SST ビットの無効化を仮定するコード・ブロックの始まりを特定します。	4-87
.sst_on	SST ビットの有効化を仮定するデフォルト動作を再開します。	4-87

(m) その他の疑似命令

ニーモニックと構文	説明	ページ
.asmfunc	関数を含むコード・ブロックの始まりを特定します。	4-32
.emsg *string*	ユーザー定義のエラー・メッセージを stdout に送ります。	4-50
.end	プログラムを終了します。	4-52
.endasmfunc	関数を含むコード・ブロックの終わりを特定します。	4-32
.mmsg *string*	ユーザー定義のメッセージを stdout に送ります。	4-50
.newblock	ローカル・ラベルの定義を解除します。	4-77
.noremark [*num*]	アセンブラの *num* 注釈が抑制されるコード・ブロックの始まりを特定します。	4-78
.remark [*num*]	.noremark により事前に抑止された注釈を生成するというデフォルト動作を再開します。	4-78
.warn_off	アセンブラの警告メッセージが抑制されるコード・ブロックの始まりを特定します。	4-102
.warn_on	アセンブラ警告メッセージをレポートするデフォルト動作を再開します。	4-102
.wmsg *string*	ユーザー定義の警告メッセージを stdout に送ります。	4-50

4.2 セクションを定義する疑似命令

次の疑似命令は、アセンブリ言語プログラムのさまざまな部分を適切なセクションと関連付け、または特定のセクションを示すフラグを有効化します。

- **.bss** は、初期化されていない変数用の .bss セクションに空間を確保します。データ・セクションなので、指定したサイズ・パラメータはワード単位でなくてはなりません。
- **.clink** は、名前付きセクションのタイプ・フィールドに STYP_CLINK フラグを設定します。.clink 疑似命令は、初期化されたセクションまたは初期化されていないセクションのいずれにも適用できます。STYP_CLINK フラグは、リンカに対して、そのセクション内のシンボルへの参照が 1 つも検出されない場合は、そのセクションをリンカの最終 COFF 出力から除去するように指示し、条件リンクを有効化します。
- **.data** は、.text セクションのコード部分を特定します。.data セクションは、通常は初期化されたデータを含んでいます。C55x では、データ・セクションはワード・アドレス可能です。
- **.sect** は、初期化された名前付きセクションを定義し、それに続くコードまたはデータをそのセクションに関連付けます。.sect で定義されたセクションは、実行可能なコードまたはデータを含みます。
- **.text** は、.text セクションのコード部分を特定します。.text セクションは、通常は実行可能コードを含んでいます。C55x では、コード・セクションはバイト・アドレス可能です。
- **.usect** は、初期化されていない名前付きセクションに空間を確保します。.usect 疑似命令の機能は .bss 疑似命令と似ています。ただし、.usect を使用すると .bss セクションとは別に空間を確保できます。データ・セクションなので、指定したサイズ・パラメータはワード単位でなくてはなりません。

第 2 章「共通オブジェクト・ファイル・フォーマットの概要」で、COFF セクションについて詳しく説明します。

例 4–1 は、セクション疑似命令を使用してコードとデータを正しいセクションに関連付ける方法を示しています。この例は出力リストです。カラム 1 は行番号を示し、カラム 2 は SPC 値を示しています（各セクションには専用のプログラム・カウンタ、つまり SPC があります）。最初にコードがセクションに配置されるときの SPC は 0 です。他のコードをアセンブルした後で、セクションへのアセンブルを再開すると、そのセクションの SPC は、介在コードがなかったものとして、カウントを再開します。

例 4–1 の疑似命令は、次の作業を実行します。

.text	基本的な追加およびロードの命令が入っています。
.data	値 9、10、11、12、13、14、15、および 16 をもつワードを初期化します。
var_defs	値 17 および 18 をもつワードを初期化します。

.bss 19 ワードを確保します。

.usect 20 ワードを確保します。

.bss 疑似命令と .usect 疑似命令は、現行のセクションの終了も新規セクションの開始も実行しません。この 2 つの疑似命令は指定量の空間を確保し、その後、アセンブラは現行のセクションへのコードまたはデータのアセンブルを再開します。

例 4–1. セクション疑似命令

```
 1                ************************************************
 2                *     Start assembling into the .text section  *
 3                ************************************************
 4  000000                .text
 5  000000 3CA0           MOV #10,AC0
 6  000002 2201           MOV AC0,AC1
 7
 8                ************************************************
 9                *     Start assembling into the .data section  *
10                ************************************************
11  000000                .data
12  000000 0009           .word   9, 10
    000001 000A
13  000002 000B           .word   11, 12
    000003 000C
14
15                ************************************************
16                *       Start assembling into a named,         *
17                *       initialized section, var_defs          *
18                ************************************************
19  000000                .sect   "var_defs"
20  000000 0011           .word   17, 18
    000001 0012
21
22                ************************************************
23                *    Resume assembling into the .data section    *
24                ************************************************
25  000004                .data
26  000004 000D           .word   13, 14
    000005 000E
27  000000                .bss    sym, 19   ; Reserve space in .bss
28  000006 000F           .word   15, 16    ; Still in .data
    000007 0010
29
30                ************************************************
31                *    Resume assembling into the .text section    *
32                ************************************************
33  000004                .text
34  000004 2412           ADD AC1,AC2
35  000000      usym      .usect  "xy", 20   ; Reserve space in xy
36  000006 2220           MOV AC2,AC0      ; Still in .text
```

4.3 データ定義疑似命令

この節では、値を現行のセクションにアセンブルする疑似命令について説明します。

注： これらの疑似命令はデータ・セクションで使用してください。
コード・セクションとデータ・セクションのアドレス指定のしかたは異なるので、C55x 疑似命令が組み込まれているセクション内でこれらの疑似命令を使用すると、実行時にデータへの無効なアクセスが生成される可能性があります。ですから、これらの疑似命令は、データ・セクション内でのみ発行されるようにすることをお勧めします。

- **.space** 疑似命令は、指定のビット数を現行のセクションに確保します。アセンブラは、これらの確保されたビットに 0 を埋め込みます。

  ビット数を 16 で割ることにより、指定のワード数を確保できます。

  .space でラベルを使用すると、そのラベルは確保済みのビットを含む最初のバイト（コード・セクション内）またはワード（データ・セクション内）を指します。

  この例では、次のコードをアセンブルしています。

```
1
2                       ** .space directive
3  000000                       .data
4  000000  0100                 .word           100h, 200h
   000001  0200
5  000002               Res_1:  .space          17
6  000004 000F                  .word 15
7                       ** reserve 3 words
8  000005               Res_3:  .space          3*16
9  000008  000A                 .word           10
```

  Res_1 は、.space により確保された空間の最初のワードを指します。

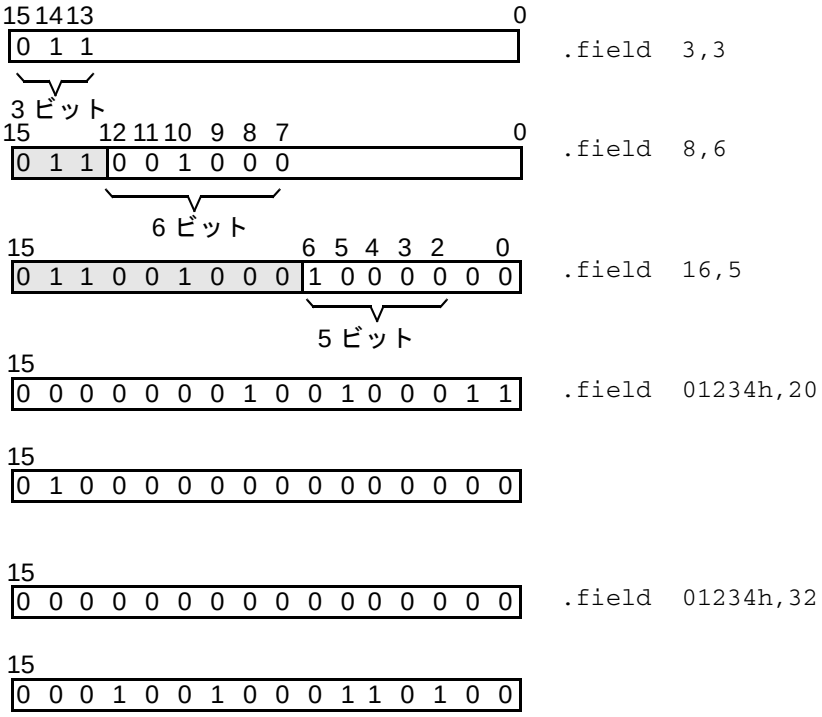
- **.byte**、**.ubyte**、**.char**、および **.uchar** の疑似命令は、1 つ以上の 8 ビット値を現行のセクションの連続するワードに格納します。これらの疑似命令は .word と .uword に似ていますが、各値の幅が 8 ビットに制限されている点が異なります。

- **.field** 疑似命令は、1 つの値を（データ・セクション内の）ワードの指定された数のビットに入れます。.field を使用すると、複数のフィールドを単一のワードにパックできます。アセンブラは、1 つのワードが完全に埋まるまで SPC をインクリメントしません。1 つの値が 1 ワードに収まる場合、ワード・アドレス境界にまたがらないことをアセンブラが保証します。

図 4-1 は、フィールドがどのようにワードにパックされるかを示しています。この例では、次のコードをアセンブルしています。最初の 3 つのフィールドで SPC は変わらないことに注目してください（つまり、これらのフィールドは同じワードにパックされます）。

```
3 000000              .data
4 000000 6000         .field          3, 3
5 000000 6400         .field          8, 6
6 000000 6440         .field          16, 5
7 000001 0123         .field          01234h,20
  000002 4000
8 000003 0000         .field          01234h,32
  000004 1234
```

図 4-1. .field 疑似命令



- **.float** と **.xfloat** は、1 つの浮動少数点値の単精度（32 ビット）IEEE 浮動小数点表記を計算し、その結果を現行のセクションの連続する 2 ワードに格納します。最上位のワードを最初に格納します。.float 疑似命令は最も近くの倍長ワード境界に自動的に位置合わせを行いますが、.xfloat はこの位置合わせを行いません。

- **.int**、**.uint**、**.half**、**.uhalf**、**.short**、**.ushort**、**.word**、および **.uword** は、1 つ以上の 16 ビット値を現行のセクション内の連続するワードに入れます。

- **.double** と **.ldouble** は、1 つ以上の浮動少数点値の倍精度（64 ビット）IEEE 浮動小数点表記を計算し、その結果を現行のセクションの連続する 4 ワードに格納します。.double 疑似命令は、自動的に倍長ワード境界に位置合わせします。

- **.ivec** 疑似命令を使用して、割り込みベクター・テーブルの入力を初期化します。

- **.long**、**.ulong** および **.xlong** は、32 ビット値を現行のセクション内の連続した 2 ワードに格納します。最上位のワードを最初に格納します。.long 疑似命令は倍長ワード境界に自動的に位置合わせを行いますが、.xlong は位置合わせを行いません。

- **.string** および **.pstring** は、1 つ以上の文字列内の 8 ビット文字を現行のセクションに格納します。.string 疑似命令は .byte に似ています。.string は、8 ビット文字をカレント・データ・セクション内の連続したワードに格納します。.pstring も 8 ビットの幅を持っていますが、1 ワードあたり 2 文字をパックします。.pstring の場合、文字列内の最終ワードには必要に応じてヌル文字（0）が埋め込まれます。

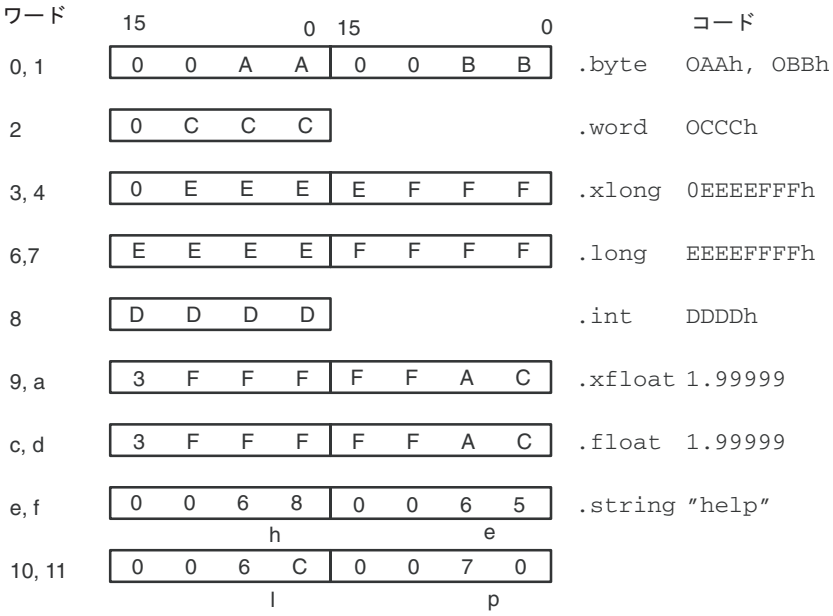
注 ： .struct/.endstruct シーケンスにおけるこれらの疑似命令

上記の疑似命令が .struct/.endstruct シーケンスの一部である場合、メモリは初期化されず、メンバのサイズが定義されます。.struct/.endstruct 疑似命令の詳細は、4-22 ページの 4.9 節「アセンブル時シンボル疑似命令」を参照してください。

図 4-2 では、.byte、.int、.long、.xlong、.float、.xfloat、.word、および .string 疑似命令を比較しています。この例では、次のコードをアセンブルしています。

```
1 000000              .data
2 000000 00AA         .byte           0AAh, 0BBh
  000001 00BB
3 000002 0CCC         .word           0CCCh
4 000003 0EEE         .xlong          0EEEEFFFh
  000004 EFFF
5 000006 EEEE         .long           0EEEEFFFFh
  000007 FFFF
6 000008 DDDD         .int            0DDDDh
7 000009 3FFF         .xfloat         1.99999
  00000a FFAC
8 00000c 3FFF         .float          1.99999
  00000d FFAC
9 00000e 0068         .string         "help"
  00000f 0065
  000010 006c
  000011 0070
```

図 4-2.　　初期化疑似命令


ワード
15
0
15
0
コード
0, 1
0 0 A A
0 0 B B
.byte OAAh, OBBh
2
0 C C C
.word OCCCh
3, 4
0 E E E
E F F F
.xlong 0EEEEFFFh
6,7
E E E E
F F F F
.long EEEEFFFFh
8
D D D D
.int DDDDh
9, a
3 F F F
F F A C
.xfloat 1.99999
c, d
3 F F F
F F A C
.float 1.99999
e, f
0 0 6 8
0 0 6 5
.string "help"
h
e
10, 11
0 0 6 C
0 0 7 0
l
p

4.4 セクション・プログラム・カウンタの位置合わせを行う疑似命令

これらの疑似命令は、セクション・プログラム・カウンタ（SPC）の位置合わせまたは位置合わせの処理のいずれかを実行します。

- **.align** 疑似命令は、SPC をコード・セクションのバイト境界またはデータ・セクションのワード境界で位置合わせします。SPC が、すでに選択した境界の上に位置合わせされている場合、SPC はインクリメントされません。.align 疑似命令のオペランドは、2^0 ～ 2^{16} の間の 2 の累乗でなくてはなりません。

  .align 疑似命令にオペランドがない場合、デフォルトの位置合わせは、コード・セクションでは 128 バイト境界に、データ・セクションでは 128 ワード（ページ）境界に設定されます。

- **.even** 疑似命令は、SPC を、次のワード境界（コード・セクション内）または倍長ワード（データ・セクション内）を指すように位置合わせします。これは、オペランドが 2 の .align 疑似命令を指定するのと同じことです。現行のバイトまたは現行のワード内の未使用ビットには、ゼロ（0）が埋め込まれます。

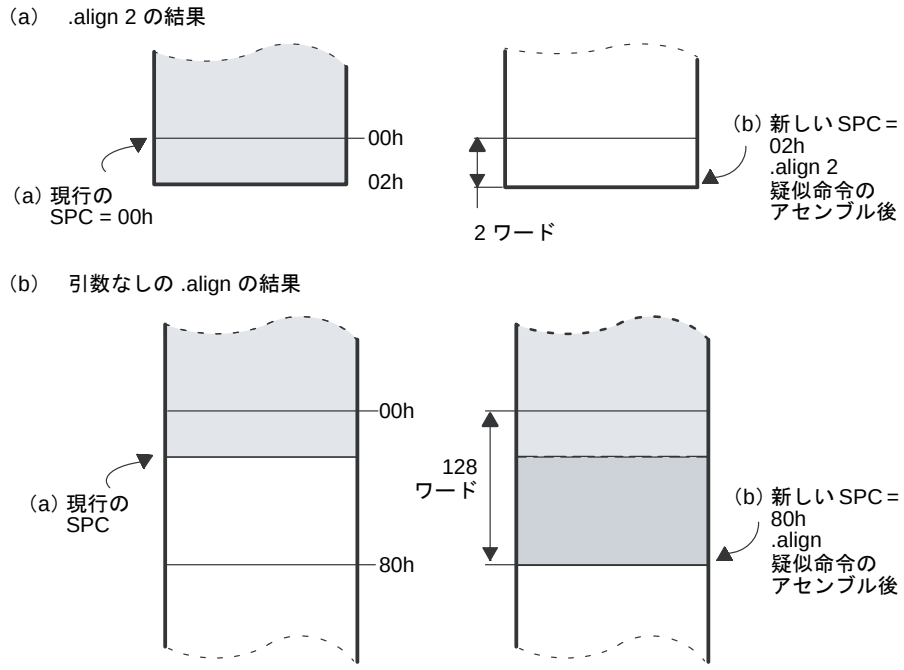
- The **.localalign** 疑似命令は、特定ループの最大の localrepeat ループ・サイズを許容します。

- **.sblock** 疑似命令は、ブロック化するセクションを指定します。ブロック化は、ページ位置合わせに似た（ただしそれより制約の弱い）機能をもつアドレス位置合わせメカニズムです。コード・セクションでは、ブロック化されたコードは、128 バイトより小さい場合は 128 バイト にまたがらないように設定され、128 バイトより大きい場合は 128 バイトから始まるように設定されます。データ・セクションでは、ブロック化されたコードは、1 ページより小さい場合は 128 ワード（ページ）境界にまたがらないように設定され、1 ページより大きい場合は ページ境界から始まるように設定されます。この疑似命令は、初期化されたセクションについてのみブロック化を指定できます。.usect または .bss セクションで宣言された初期化されないセクションは指定できません。

図 4-3 は .align 疑似命令を示しています。この例では、次のコードをアセンブルしています。

```
1 000000                .data
2 000000 4000           .field      2, 3
3 000000 4160           .field      11, 8
4                       .align      2
5 000002 0045           .string     "Errorcnt"
  000003 0072
  000004 0072
  000005 006f
  000006 0072
  000007 0063
  000008 006e
  000009 0074
6                       .align
7 000080 0004           .word       4
```

図 4-3. .align 疑似命令



4.5 出力リストのフォーマットを設定する疑似命令

リスト・ファイルのフォーマットを設定する疑似命令には、次のものがあります。

- **.drnolist** 疑似命令を使用すると、リスト・ファイルへの次の疑似命令の出力を抑止できます。

.asg	.eval	.length	.mnolist	.var
.break	.fclist	.mlist	.sslist	.width
.emsg	.fcnolist	.mmsg	.ssnolist	.wmsg

  リスト出力を再開させるには **.drlist** 疑似命令使用します。

- リスト・ファイルは、コードを作成しない偽の条件付きブロックのリストを含んでいます。**.fclist** および **.fcnolist** 疑似命令は、それぞれにこのリスト作成の始動と停止を行います。.fclist 疑似命令を使用すると、ソース・コードどおりに偽の条件ブロックのリストを作成できます。これはアセンブラのデフォルトの動作です。.fcnolist 疑似命令を使用すると、実際にアセンブルされる条件ブロックのみのリストを作成できます。

- **.length** 疑似命令を使用すると、リスト・ファイルのページの長さを制御できます。この疑似命令を使用すると、さまざまな出力デバイス用にリストを調整できます。

- **.list** および **.nolist** の疑似命令は、それぞれに出力リストの始動と停止を行います。.nolist 疑似命令を使用すると、アセンブラが選択したソース文をリスト・ファイルに出力しないようにできます。.list 疑似命令を使用すると、リスト作成を再開させることができます。

- リスト・ファイルは、マクロ展開とループ・ブロックのリストを含んでいます。**.mlist** および **.mnolist** 疑似命令は、それぞれにこのリスト作成の始動と停止を行います。.mlist 疑似命令を使用すると、すべてのマクロ展開とループ・ブロックをリストに出力（アセンブラのデフォルト動作です）できます。.mnolist 疑似命令を使用すると、このリスト作成を抑止できます。

- **.option** 疑似命令を使用すると、リスト・ファイルの特定の機能を設定できます。この疑似命令には次のオペランドを指定できます。

A	すべての疑似命令とデータ、および以降の拡張子、マクロ、ブロックのリスト作成を再開します。
B	.byte 疑似命令によるリスト作成を 1 行に制限します。
D	特定の疑似命令（.drnolist と同じ働き）によるリスト作成を停止します。
H	.half および .short 疑似命令のリスト作成を 1 行に制限します。
L	.long 疑似命令によるリスト作成を 1 行に制限します。
M	リストでのマクロ展開を停止します。
N	リスト作成（.nolist を実行する）を停止します。

O リスト作成（.list を実行する）を再開します。

R B、M、T および W オプションをリセットします。

T .string 疑似命令によるリスト作成を 1 行に制限します。

W .word 疑似命令によるリスト作成を 1 行に制限します。

X シンボルのクロスリファレンス・リストを作成します（アセンブラを起動するときに -x オプションを指定して、クロスリファレンス・リストを取得することもできます)。

- **.page** 疑似命令により、出力リストのページ替えを行うことができます。
- **.sslist** および **.ssnolist** 疑似命令は、展開された置換シンボルのリスト作成を許可および禁止します。この 2 つの疑似命令は、置換シンボルの展開のデバッグを行うときに便利です。
- **.tab** 疑似命令 は、タブサイズを定義します。
- **.title** 疑似命令は、アセンブラが各ページ上部に出力するタイトルを提供します。
- **.width** 疑似命令を使用すると、リスト・ファイルのページの幅を設定できます。この疑似命令を使用すると、さまざまな出力デバイス用にリストを調整できます。

4.6 他のファイルを参照する疑似命令

.copy および **.include** の疑似命令は、アセンブラに対して、他のファイルからソース文の読み取りを開始するように指示します。アセンブラは、コピー／インクルード・ファイルのソース文の読み取りを終了すると、.copy または .include 疑似命令が発生した地点にすぐに続く現行のファイルからのソース文の読み取りを再開します。コピー・ファイルから読み取られた文は、リスト・ファイルに出力されます。インクルード・ファイルから読み取られた文は、リスト・ファイルに出力されません。

4.7 シンボル・リンク疑似命令

これらの疑似命令は、シンボルの有効範囲または可能性について参照します。

- **.def** 疑似命令は、現行のモジュールで定義され、別のモジュールで使用できるシンボルを特定します。アセンブラは、このようなシンボルをシンボル・テーブルに入れます。

- **.global** 疑似命令は、シンボルを外部シンボルとして宣言し、リンク時に他のモジュールを使えるようにします（グローバル・シンボルに関する詳細は、2-19 ページの 2.7.1 項「外部シンボル」を参照してください。.global 疑似命令は、定義されたシンボルに対しては .def として機能し、定義されていないシンボルに対しては .ref として機能するという二重の役割を果たします。リンカは、定義されていないグローバル・シンボルがプログラム内で使われている場合にのみ、そのグローバル・シンボルを解決します。

- **.ref** 疑似命令は、他のモジュールで定義され、現行のモジュールからアクセス可能なシンボルを特定します。アセンブラは、このようなシンボルを定義されていない外部シンボルとしてマークを付け、リンカがその定義を解決できるようにオブジェクト・シンボル・テーブルに入れます。

4.8 条件付きアセンブラ疑似命令

条件付きアセンブラ疑似命令を使用すると、アセンブラに対して、式の評価結果が真か偽かに応じてそれに対応するコードのブロックをアセンブルさせることができます。2組の疑似命令を使用して、条件付きのコード・ブロックをアセンブルできます。

- **.if /.elseif /.else /.endif** 疑似命令はアセンブラに対して、式の計算結果に応じてコード・ブロックを条件付きでアセンブルするように指示します。式は、疑似命令と同じ行に全体を指定する必要があります。

.if 式	条件ブロックの始まりにマークを付け、.if 条件が真の場合にコードをアセンブルします。
.elseif 式	.if が偽で、かつ .elseif が真の場合に、コード・ブロックにアセンブルされるようにマークを付けます。
.else	.if が偽の場合に、コード・ブロックにアセンブルされるようにマークを付けます。
.endif	条件ブロックの終わりにマークを付け、そのブロックを終了します。

- **.loop/.break/.endloop** 疑似命令はアセンブラに対して、式の計算結果に応じてコード・ブロックを繰り返してアセンブルするように指示します。式は、疑似命令と同じ行に全体を指定する必要があります。

.loop 式	式により何回も示されるまで繰り返してアセンブルされるコード・ブロックの始まりにマークを付けます。式はループ・カウントです。
.break 式	アセンブラに対して、.break 式が偽のときにアセンブルを繰り返し、式が真のときは .endloop 直後のコードに移るように指示します。
.endloop	反復使用が可能なブロックの終わりにマークを付けます。

アセンブラは、条件式に対して便利な関係演算子をいくつかサポートしています。関係演算子の詳細は、3-38 ページの 3.11.4 項「条件式」を参照してください。

4.9 アセンブル時シンボル疑似命令

アセンブル時シンボル疑似命令は、意味のあるシンボル名を定数値、または文字列と等価にします。

- **.asg** 疑似命令は、置換シンボルに文字列を割り当てます。値は置換シンボル・テーブルに格納されます。アセンブラは置換シンボルを検出すると、そのシンボルを対応する文字列の値に置換します。置換シンボルは定義し直すことができます。

```
.asg    "10, 20, 30, 40", coefficients

.byte   coefficients
```

- **.cstruct/.cunion** 疑似命令は、アセンブリおよび C コード間で共通データ構造を共有しやすいようサポートします。.cstruct/.cunion 疑似命令は、既存の .struct と .union と全く同様に使用できます。ただしこれらの疑似命令が、C struct および union のデータ・タイプ用の C コンパイラの使用するレイアウトに対応したデータ・レイアウトが保障される場合を除きます。特に、このようなタイプが複合的なデータ構造内にネストされている場合、.cstruct/.cunion 疑似命令は C コンパイラが使用するのと同じ位置合わせおよび埋め込みを強制します。

- **.eval** 疑似命令は、式を計算し、その結果を文字に変換し、その文字列を置換シンボルに割り当てます。この疑似命令は、カウンタを操作する上でとても便利です。

```
.asg    1 , x
.loop
.byte   x*10h
.break  x = 4
.eval   x+1, x
.endloop
```

- **.label** 疑似命令は、現行のセクション内のロード・アドレスを参照する特殊シンボルを定義します。この疑似命令は、セクションをロードするアドレスと実行するアドレスが異なる場合に使用すると有効です。たとえば、パフォーマンスを左右するコードのブロックを低速のオフチップ・メモリにロードしてメモリ空間を節約し、それを実行するときには高速のオンチップ・メモリに移動するような場合です。

- **.set** および **.equ** の疑似命令は、値をシンボルに設定します。このシンボルはシンボル・テーブルに格納され、定義し直すことはできません。次に例を示します。

```
bval  .set            0100h
      .int            bval, bval*2, bval+12
      B       bval
```

.set および .equ の疑似命令はオブジェクト・コードを作成しません。この 2 つの疑似命令は同じ働きをし、交換して使用できます。

- **.struct/.endstruct** の疑似命令は、C に似た構造体定義を設定し、**.tag** 疑似命令は、C に似た構造体特性をラベルに割り当てます。

  .struct/.endstruct 疑似命令を使用すると、情報から構造体を組織することにより、類似の要素をまとめてグループ化できます。要素オフセット計算はアセンブラに委ねられます。.struct/.endstruct 疑似命令はメモリの割り当てを行いません。これらの疑似命令は、単に反復使用が可能なシンボリックなテンプレートを作成するだけです。

  .tag 疑似命令は、構造体特性をラベル・シンボルに関連付けます。これによりシンボリック表記は簡略化され、さらに他の構造体をもつ構造体も定義できるようになります。.tag 疑似命令はメモリの割り当ては行いません。構造体タグ（stag）を使用するには、最初にそれを定義しておく必要があります。

```
      .data
type  .struct           ; structure tag definition
X     .int
Y     .int
T_LEN .endstruct

COORD .tag type         ; declare COORD (coordinate)
      .bss COORD, T_LEN ; actual memory allocation
              .text
       ADD @(COORD.Y),AC0,AC0
```

- **.union/.endunion** の疑似命令は、反復使用が可能なシンボリックなテンプレートを作成します。これにより、同じ記憶域内の異なる種類のデータを操作できるようになります。共用体は C に似た共用体定義を設定します。共同体はメモリを割り当てませんが、サイズとタイプの代替定義を一時的に同じメモリ空間に格納できるようにします。

  .tag 疑似命令は、共用体特性をラベル・シンボルに関連付けます。そして、.tag 疑似命令はその共用体の開始点に関連付けられます。その後 .tag 疑似命令を使用して、その共用体を構造体のメンバとして宣言できます。共用体はタグを付けずに宣言することもできますが、その場合はすべてのメンバがシンボル・テーブルに格納されるので、各メンバには必ず固有の名前を付けなければなりません。

また、共用体は構造体内で定義することも可能です。共用体への参照はその共用体をもつ構造体を介して作成されなければなりません。次に例を示します。

```
       .data
s2_tag .struct    ;structure tag definition
       .union     ;union is first structure member
       .struct    ;structure is union member
h1     .half      ;h1, h2, and w1
h2     .uhalf     ;exist in the same memory
       .endstruct
w1     .word      ;word is another union member

       .endunion
w2     .word      ;second structure member
s2_len .endstruct

XYZ    .tag   s2_tag
       .bss   XYZ,s2_len   ;declare instance of structure

       .text
       ADD @(XYZ.h2),AC0,AC0
```

4.10 実行時環境の詳細を伝える疑似命令

これらの疑似命令は、コード処理中にアセンブラの前提条件に影響します。これらの疑似命令がマークした範囲で、アセンブラのデフォルトの動作は指定通り変更します。

- **.dp** 疑似命令は、DP レジスタの値を指定します。アセンブラは DP レジスタの値を追跡できませんが、直接メモリ・アクセス・オペランドをアセンブルするために DP 値を知っていなくてなりません。ですから、この疑似命令は、DP レジスタの値を変更する命令の直後に配置する必要があります。DP レジスタの値について、アセンブラが何の情報も提供しない場合、直接メモリ・オペランドのエンコード時の値は 0 とされます。

- **.lock_on** 疑似命令は、アセンブラが lock() 修飾子を許容するコード・ブロックを開始します。**.lock_off** 疑似命令は、このコード・ブロックを終了して、アセンブラのデフォルトの動作を再開します。

- **.vli_off** 疑似命令は、アセンブラが特定の可変長命令の最大書式（P24）を使用するコード・ブロックの開始点を特定します。デフォルトでは、アセンブラは可変長の命令を最小の形に分解しようとします。**.vli_on** 疑似命令は、このコード・ブロックを終了して、アセンブラのデフォルトの動作を再開します。

以下の疑似命令は、C55x アドレッシング・モードに関係しています。

- **.arms_on** 疑似命令は、アセンブラがコード・サイズの最適化用の間接アクセス修飾子を使用するコード・ブロックを開始します。この修飾子は、ショート・オフセット修飾子です。**.arms_off** 疑似命令は、コード・ブロックを終了します。

- **.c54cm_on** 疑似命令は、このあとに続くコード・ブロックが C54x コードから変換されたものであることをアセンブラに伝えます。**.c54cm_off** 疑似命令は、コード・ブロックを終了します。

- **.cpl_on** 疑似命令は、直接メモリ・アドレッシング（DMA）がスタック・ポインタに対して相対的であるコード・ブロックを開始します。デフォルトでは、DMA はデータ・ページに対して相対的です。**.cpl_off** 疑似命令は、コード・ブロックを終了します。

以下の疑似命令は、C54x コードの移植に関係しています。

- **.port_for_speed** 疑似命令は、アセンブラが移植された C54x コードをコードを速くする目標値にエンコードするコード・ブロックを開始します。デフォルトでは、アセンブラは C54x コードをコードを少なくする目標値にエンコードします。**.port_for_size** 疑似命令は、コード・ブロックを終了します。

- **.sst_off** 疑似命令は、アセンブラが SST ステータス・ビットを 0 に設定すると仮定するコード・ブロックを開始します。デフォルトでは、アセンブラは SST ビットを 1 に設定するものとします。**.sst_on** 疑似命令は、コード・ブロックを終了します。

4.11 その他の疑似命令

次の疑似命令は、その他のさまざまな機能を実行します。

- **.asmfunc** 疑似命令は、関数を含むコード・ブロックを開始します。**.endasmfunc** 疑似命令は、関数コードを終了して、アセンブラのデフォルトの動作を再開します。これらの疑似命令は、コンパイラ -gw オプションとともに使われ、別々の関数についてのデバッグ情報を生成します。

- **.end** 疑似命令は、アセンブリを終了させます。この疑似命令は、プログラムの最後のソース文でなければなりません。この疑似命令は、end-of-file と同じ働きをします。

- **.newblock** 疑似命令は、ローカル・ラベルをリセットします。ローカル・ラベルは、$n または name? という書式のシンボルです。ローカル・ラベルは、ラベル・フィールドに現れた時点で定義されます。ローカル・ラベルは、ジャンプ命令のオペランドとして使用できる一時的なラベルです。.newblock 疑似命令は、ローカル・ラベルの定義を解除し、それによってローカル・ラベルの有効範囲を制限します。ローカル・ラベルに関する詳細は、3-33 ページの 3.10.6 項「ローカル・ラベル」を参照してください。

- **.noremark** 疑似命令は、アセンブラが特定のアセンブラ注釈を抑止するコード・ブロックを開始します。注釈はアセンブラの情報メッセージであり、警告ほど深刻ではありません。**.remark** 疑似命令は、.noremark により事前に抑止された注釈の出力を再び可能にします。

- **.warn_on/.warn_off** 疑似命令は、アセンブラによる警告メッセージの発行を有効および無効にします。デフォルトでは、警告は有効です（.warn_on）。

次の 3 つの疑似命令を使用すると、独自のエラー・メッセージや警告メッセージを定義できます。

- **.emsg** 疑似命令は、エラー・メッセージを標準出力デバイスに送ります。.emsg 疑似命令はアセンブラと同様にエラーを生成させます。つまりエラー回数を 1 つずつインクリメントさせ、アセンブラがオブジェクト・ファイルを作成するのを停止させます。

- **.mmsg** 疑似命令は、アセンブル時メッセージを標準出力デバイスに送ります。.mmsg 疑似命令は .emsg 疑似命令および .wmsg 疑似命令と同じ機能を果たしますが、エラー回数も警告回数もインクリメントさせません。またオブジェクト・ファイルの作成に影響しません。

- **.wmsg** 疑似命令は、警告メッセージを標準出力デバイスに送ります。.wmsg 疑似命令は .emsg 疑似命令と同じ機能を果たしますが、エラー回数ではなく警告回数をインクリメントさせます。またオブジェクト・ファイルの作成に影響しません。

4.12 疑似命令のリファレンス

以降のページは疑似命令のリファレンスです。原則として、疑似命令はアルファベット順に並べられており、1 ページにつき 1 つの疑似命令について説明します。ただし、関連性のある複数の疑似命令（たとえば .if/.else/.endif）は 1 ページにまとめてあります。

.align/.even　セクション・プログラム・カウンタ（SPC）の境界位置合わせ

構文

.align [*size*]
.even

説明

.align 疑似命令は、*size* パラメータによって、セクション・プログラム・カウンタ（SPC）を次の境界で位置合わせします。*size* には 2 の累乗であればどんな値でも指定できますが、位置合わせに有効なのは特定の値に限られます。

size パラメータは、コード・セクションの場合はバイト単位で、データ・セクションの場合はワード単位で指定する必要があります。*size* を指定しないと、SPC は、コード・セクションの場合は次の 128 バイト境界で位置合わせされ、データ・セクションの場合は次の 128 ワード（ページ）境界で位置合わせされます。

.align 疑似命令が発生した点で、SPC が必要なバイト境界またはワード境界上にない場合、この疑似命令によりホールが作成されることがあります。データ・セクションでは、アセンブラが .align により作成されたホールをゼロで埋めます。コード・セクションでは、ホールは NOP 命令が埋め込まれます。

.even 疑似命令は、SPC をワード境界（コード・セクション）または倍長ワード境界（データ・セクション）で位置合わせします。この疑似命令は、オペランドが 2 の .align 疑似命令と同じです。

.align 疑似命令を使用すると、次の 2 つの効果があります。

- ❑ アセンブラは SPC をカレント・セクション内の境界上に位置合わせします。
- ❑ アセンブラはリンカに対して、セクションの位置合わせをさせるためのフラグを設定します。これにより、セクションがメモリにロードされたときに個々の位置合わせに影響が及ばないようにします。

例

この例は、.even、.align 4、デフォルトの .align などのいくつかの位置合わせのタイプを表しています。

```
 1 000000                       .data
 2 000000 0004          .word      4
 3                      .even
 4 000002 0045          .string    "Errorcnt"
   000003 0072
   000004 0072
   000005 006F
   000006 0072
   000007 0063
   000008 006E
   000009 0074
 5                      .align
 6 000080 6000          .field     3,3
 7 000080 6A00          .field     5,4
 8                      .align     2
 9 000082 6000          .field     3,3
10                      .align     8
11 000088 5000          .field     5,4
12                      .align
13 000100 0004          .word      4
```

.arms_on/ .arms_off　選択アドレスにおけるコードの表示

構文

.arms_on
.arms_off

説明

.arms_on 疑似命令と **.arms_off** 疑似命令は、ARMS ステータス・ビットをモデル化します。

アセンブラは、ARMS ステータス・ビットの値は追跡できません。このモードの値をアセンブラに通知するには、アセンブラ疑似命令とコマンド行オプション、あるいはそのいずれかを使用する必要があります。ARMS ステータス・ビットの値を変更する命令の直後には、該当するアセンブラ疑似命令が必要です。

.arms_on 疑似命令は、1 に設定された ARMS ステータス・ビットをモデル化します。これは、-ma コマンド行オプションを使用するのと同等です。.arms_off 疑似命令は、0 にセットされた ARMS ステータス・ビットをモデル化します。コマンド行オプションと疑似命令の間でコンフリクトが起こった場合、疑似命令が優先します。

デフォルト（.arms_off）では、アセンブラはアセンブリ・コード用の間接メモリ・アクセス修飾子を使用します。

ARMS モード（.arms_on）では、アセンブラは間接メモリ・アクセスにショート・オフセット修飾子を使用します。これらの修飾子は、コード・サイズの最適化により効果的です。

.arms_on および .arms_off 疑似命令の有効範囲は静的であり、アセンブリ・プログラムの制御フローには影響されません。.arms_on 行と .arms_off 行との間にあるすべてのアセンブリ・コードは、ARMS モードでアセンブルできます。

.asg/.eval 置換シンボルの割り当て

構文

.asg [**"**]*character string*[**"**]*, substitution symbol*
.eval *well-defined expression, substitution symbol*

説明

.asg 疑似命令は、置換シンボルに文字列を割り当てます。置換シンボルは、置換シンボル・テーブルに格納されます。.asg 疑似命令は .set 疑似命令とほとんど同じ使い方ができます。ただし、.set は（再定義できない）定数値をシンボルに割り当てますが、.asg は（再定義できる）文字列を置換シンボルに割り当てる点が異なります。

- ❑ アセンブラは、*character string* を置換シンボルに割り当てます。引用符の使用は任意です。引用符がない場合、アセンブラは最初のコンマまでの文字を読み取り、その前後の空白を取り除きます。どちらの場合でも、文字列が読み取られて置換シンボルに割り当てられます。

- ❑ *substitution symbol* は、有効なシンボル名でなくてはなりません。置換シンボルは長さが 32 文字までで、先頭を文字で始める必要があります。シンボルの残りの文字には、英数字、下線（_）、およびドル記号（$）を組み合わせて使用できます。

.eval 疑似命令は、提供された式の算術計算を行い、式の結果の文字列表記を置換シンボルに割り当てます。この疑似命令は、式を計算し、結果の文字列値を置換シンボルに割り当てます。.eval 疑似命令は、.loop/.endloop のブロックのカウンタとして使用すると特に便利です。

- ❑ *well-defined expression* は、事前に定義された正しい値で構成されている、つまり絶対的な結果を得られる英数字式のことをいいます。

- ❑ *substitution symbol* は、有効なシンボル名でなくてはなりません。置換シンボルは長さが 32 文字までで、先頭を文字で始める必要があります。シンボルの残りの文字には、英数字、下線（_）、およびドル記号（$）を組み合わせて使用できます。

例　　.asg と .eval の使用例を以下に示します。

```
       1                        .sslist;show expanded sub. symbols
       2              *
       3              *         .asg/.eval example
       4              *
       5                        .asg    *+, INC
       6                        .asg    AR0, FP
       7
       8 000000 7b00            ADD #100,AC0
         000002 6400
       9 000004 b403            AMAR     (*FP+)
#                               AMAR     (AR0+)
      10
      11
      12 000000                 .data
      13                        .asg    0, x
      14                        .loop   5
      15                        .eval   x+1, x
      16                        .word   x
      17                        .endloop
1                               .eval   x+1, x
#                               .eval   0+1, x
1        000000 0001            .word   x
#                               .word   1
1                               .eval   x+1, x
#                               .eval   1+1, x
1        000001 0002            .word   x
#                               .word   2
1                               .eval   x+1, x
#                               .eval   2+1, x
1        000002 0003            .word   x
#                               .word   3
1                               .eval   x+1, x
#                               .eval   3+1, x
1        000003 0004            .word   x
#                               .word   4
1                               .eval   x+1, x
#                               .eval   4+1, x
1        000004 0005            .word   x
#                               .word   5
```

.asmfunc/ .endasmfunc 関数境界のマーク

構文

symbol **.asmfunc**
.endasmfunc

説明

.asmfunc および **.endasmfunc** 疑似命令は、関数境界をマークします。コンパイラ -g オプション（--symdebug:DWARF）とともにこれらの疑似命令を使うと、セクション・アセンブル・コードを C/C++ 関数と同じ方法でデバッグできます。

コンパイラの生成した同じ疑似命令を使用してアセンブル・デバッグを行ってはいけません（付録 B 参照）。これらの疑似命令は、C/C++ ソース・ファイルのシンボル・デバッグ情報を生成するためにコンパイラだけが使用できます。

.asmfunc および .endasmfunc 疑似命令は、下位互換性のあるコンパイラ --symdebug:coff オプションを起動するときには使えません。このオプションは、コンパイラに旧式の COFF シンボリック・デバッグ・フォーマットを使うよう指示します。このフォーマットはこれらの疑似命令をサポートしていません。

symbol は、ラベル・フィールドに表示されるラベルです。

.asmfunc および .endasmfunc 疑似命令で囲まれていない、連続するアセンブリ・コードの範囲は、以下のフォーマットでデフォルトの名前が付きます。

$*filename***:***beginning source line***:***ending source line***$**

例　この例では、アセンブリ・ソースが user_func セクションのデバッグ情報を生成します。

```
 1 000000                       .sect   ".text"
 2                              .align 4
 3                              .global userfunc
 4                              .global _printf
 5
 6              user_func:.asmfunc
 7 000000 4EFD                  AADD #-3, SP
 8 000002 FB00                  MOV #(SL1 & 0xffff), *SP(#0)
   000004 0000%
 9 000006 6C00                  CALL #_printf
   000008 0000!
10 00000a 3C04                  MOV #0, T0
11 00000c 4E03                  AADD #3, SP
12 00000e 4804                  RET
13                              .endasmfunc
14
15 000000                       .sect   ".const"
16 000000 0048  SL1:            .string "Hello World!",10,0
   000001 0065
   000002 006C
   000003 006C
   000004 006F
   000005 0020
   000006 0057
   000007 006F
   000008 0072
   000009 006C
   00000a 0064
   00000b 0021
   00000c 000A
   00000d 0000
```

.bss .bss セクションの空間確保

構文

.bss *symbol, size in words*[**,**[*blocking flag*][**,** *alignment flag*]]

説明

.bss 疑似命令は、.bss セクションに変数用の空間を確保します。この疑似命令は、通常は RAM に変数を割り当てるときに使用します。

- *symbol* は、必須パラメータです。このパラメータは、疑似命令により確保される最初の位置を指すラベルを定義します。このシンボル名は、空間を確保する対象となる変数と対応していなければなりません。
- *size in words* は、必須パラメータです。絶対式でなければなりません。アセンブラは、.bss セクションに指定されたワード数を確保します。デフォルトのサイズはありません。
- *blocking flag* は、任意のパラメータです。このパラメータに 0 以外の値を指定すると、アセンブラは指定されたサイズの空間を連続した空間に確保します。つまり、空間のサイズが 1 ページを越えていない限り、確保された空間がページ境界にまたがることはありません。1 ページを越える場合は、オブジェクトはページ境界から開始されます。
- *alignment* は、任意のパラメータです。*alignment* は 2 の累乗です。この値は、この .bss 疑似命令によって確保された空間が特定のワード･アドレス境界に割り当てられるよう指定します。

注 : 位置合わせフラグのみを指定する

ブロック化フラグを指定しないで位置合わせフラグのみを指定するには、位置合わせフラグの前にコンマを 2 つ挿入するか、ブロックかフラグに 0 を指定します。

アセンブラは、次の 2 つの規則に従って .bss セクションに空間を確保します。

規則 1 （図 4-4 に示すように）メモリにホール（穴）が残っていると、.bss 疑似命令は必ずそのホールを埋めようとします。アセンブラは、.bss 疑似命令をアセンブルするときに、前の .bss 疑似命令により残されたホールのリストを検索して、ホールの 1 つに現行のブロックを割り当てようとします（これは、連続割り当てオプションが指定されているかどうかに関係なく実行される標準の手順です）。

規則 2 アセンブラは、要求された空間を確保できる大きさのホールが見つからなかった場合、ブロック化オプションが指定されているかどうかをチェックします。

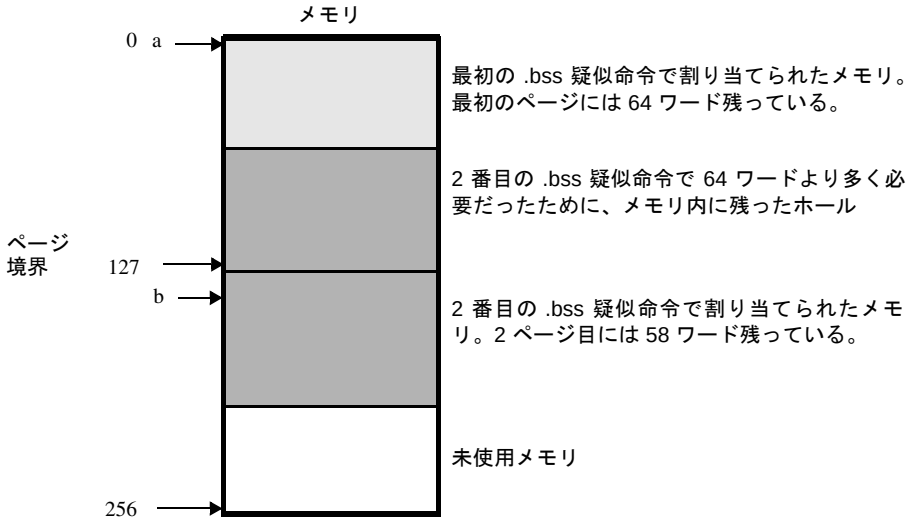
- ブロック化を指定しないと、メモリは現行の SPC の位置に割り当てられます。
- ブロック化を指定すると、アセンブラは、現行の SPC からページ境界までに十分な空間があるかどうかを調べます。ここに十分な空間がなければ、アセンブラはそこに別のホールを作成し、次のページの始まりに空間を割り当てます。

ブロック化オプションを使用すると、.bss セクションに最大 128 ワードを確保して、それがメモリの 1 ページに収まるようにできます（もちろん一度に 128 ワード以上確保できますが、その場合は 1 ページには収まりません）。次のコード例では、.bss セクションに 2 ブロックの空間が確保されます。

```
memptr:    .bss     A,64,1
memptr1:   .bss     B,70,1
```

各ブロックは 1 ページの境界内部に入っていなければなりません。しかし、最初のブロックを割り当てた後では、2 番目のブロックは現行のページ内には収まりません。図 4-4 に示すとおり、2 番目のブロックは次のページに割り当てられます。

図 4-4.　ページ内への .bss ブロックの割り当て



初期化されたセクション用のセクション疑似命令（.text、.data、.sect）は、現行のセクションを終了させ、別のセクションへのアセンブルを始めるように指示します。ただし、.bss 疑似命令は現行のセクションに影響を与えません。アセンブラは、.bss 疑似命令を使って .bss セクションに空間を確保しますが、その後現行のセクションにコードのアセンブルを再開します（.bss の処理後）。詳細は、第 2 章「共通オブジェクト・ファイル・フォーマットの概要」を参照してください。

例

この例では、.bss 疑似命令を使用して、2 つの変数 TEMP と ARRAY のための空間を確保します。シンボル TEMP は、(.bss の SPC = 0 にある) 初期化されない 4 ワードの空間を指します。シンボル ARRAY は、(.bss の SPC = 04h にある) 初期化されない 100 ワードの空間を指します。この空間は、1 ページ内に連続して配置する必要があります。.bss 疑似命令で宣言されたシンボルは、他のシンボルと同様に参照でき、.global 疑似命令を使って外部でも宣言できることに注意してください。

```
 1              ****************************************
 2              **  Assemble into the .text section.  **
 3              ****************************************
 4 000000               .text
 5 000000 3C00          MOV #0,AC0
 6              ****************************************
 7              **  Allocate 4 words in .bss for TEMP.**
 8              ****************************************
 9 000000       Var_1:.bss    TEMP, 4
10
11              ****************************************
12              **           Still in .text           **
13              ****************************************
14 000002 7B00          ADD #86,AC0,AC0
   000004 5600
15 000006 5272          MOV  T3,HI(AC2)
16 000008 1E73          MPYK #115,AC2,AC0
   00000a 80
17
18              ****************************************
19              **  Allocate 100 words in .bss for the**
20              **  symbol named ARRAY; this part of  **
21              **  .bss must fit on a single page.   **
22              ****************************************
23 0000004               .bss    ARRAY, 100, 1
24
25              ****************************************
26              **  Assemble more code into .text.    **
27              ****************************************
28 00000b C000-         MOV AC0,Var_1
29
30              ****************************************
31              **  Declare external .bss symbols.    **
32              ****************************************
33                      .global ARRAY, TEMP
34                      .end
```

.byte/.ubyte/ .char/.uchar バイトの初期化

構文

.byte *value1* [**,** *...*, *valuen*]
.ubyte *value1* [**,** *...*, *valuen*]
.char *value1* [**,** *...*, *valuen*]
.uchar *value1* [**,** *...*, *valuen*]

説明

.byte、**.ubyte**、**.char**、および **.uchar** の疑似命令は、1 つ以上の 8 ビット値を現在のデータ・セクションの連続するワードに格納します。

注 : これらの疑似命令はデータ・セクションで使用してください。

コード・セクションとデータ・セクションのアドレス指定のしかたは異なるので、C55x 命令が組み込まれているセクション内で .byte、.ubyte、.char および .uchar 疑似命令を使用すると、実行時におけるデータへの不正アクセスにつながります。ですから、これらの疑似命令は、データ・セクション内でのみ使用されるようにすることをお勧めします。

データ・セクションでは、各 8 ビット値は単独で 1 つのワードに挿入されます。8 MSB にはゼロ（0）が埋め込まれます。*value* には、次のものを指定できます。

- ❑ アセンブラが計算して、8 ビットの符号付きまたは符号なしの数値として扱う式。
- ❑ 二重引用符に囲まれた文字列。文字列内の各文字は、別々の値を表します。

値は、パックも符号拡張もされません。ワード・アドレス可能なデータ・セクションでは、各バイトは、完全な 16 ビット・ワードの LSB8 ビットを使います。アセンブラは、8 ビットを超える値は切り捨てます。

ラベルを使用する場合、ラベルはアセンブラが最初のバイトを置く位置を指します。

これらの疑似命令を .struct/.endstruct シーケンスで使用した場合、その疑似命令はメンバのサイズを定義し、メモリは初期化しないことに注意してください。.struct/.endstruct の詳細は、4.9 節「アセンブル時シンボル疑似命令」（4-22 ページ）を参照してください。

例

この例では、8 ビット値（10、-1、abc、および a）はメモリ内の連続するワードに配置されます。ラベル strx の値は 100h で、これは初期化されたワードの先頭位置を表します。

```
1 000000                .data
2 000000                .space   100h * 16
3 000100 000a STRX      .byte    10, -1, "abc", 'a'
  000101 00ff
  000102 0061
  000103 0062
  000104 0063
  000105 0061
```

.c54cm_on/ .c54cm_off 選択アドレスにおけるコードの表示

構文

.c54cm_on
.c54cm_off

説明

.c54cm_on および **.c54cm_off** の疑似命令は、コード領域が C54x コードから変換されたものであることを伝えます。.c54cm_on および .c54cm_off 疑似命令は、C54CM ステータス・ビットをモデル化します。.c54cm_on 疑似命令は、1 に設定された C54CM ステータス・ビットをモデル化します。これは、-ml コマンド行オプションを使用することと同等です。.c54cm_off 疑似命令は、0 にセットされた C54CM ステータス・ビットをモデル化します。コマンド行オプションと疑似命令の間でコンフリクトが起こった場合、疑似命令が優先します。

.c54cm_on および .c54cm_off 疑似命令の有効範囲は静的であり、アセンブリ・プログラムの制御フローには影響されません。.c54cm_on 疑似命令と .c54cm_off 疑似命令との間にあるすべてのアセンブリ・コードは、C54x 互換モードでアセンブルされます。

C54x 互換モードでは、メモリ・オペランドで T0 ではなく AR0 が使われます。たとえば C54x 互換モードでは、*（AR5 + T0）は無効であり、*（AR5 + AR0）を使わなければなりません。

.clink　COFF 出力から条件付きでセクションを除去する

構文　　**.clink** [*"section name"*]

説明

.clink 疑似命令は、リンカがデッド・コードを除去する場合に、現行のセクションまたは名前付きセクションを除去の候補とします。.clink 疑似命令は、*section name* で指定されるセクションのタイプ・フィールドに STYP_CLINK フラグを設定して、セクションの条件付きリンクを設定します。.clink 疑似命令は、初期化されたセクションまたは初期化されていないセクションのいずれにも適用できます。

セクション名を指定しないで .clink を使用する場合、.clink は現在の初期化されたセクションに適用されます。.clink を初期化されないセクションに適用する場合は、セクション名を指定する必要があります。セクション名は二重引用符で囲まなければなりません。セクション名は、*section name*:*subsection name* の書式でサブセクション名を持つことができます。

STYP_CLINK フラグは、リンカに対して、そのセクション内のシンボルへの参照が 1 つも検出されない場合は、そのセクションをリンカの最終 COFF 出力から除去するようにします。

C プログラムのエントリ・ポイントが定義されているセクションや、割り込みサービス・ルーチンのアドレスを含むセクションには、条件付きでリンクされるセクションとしてマークを付けることはできません。

例　次の例では、Vars セクションと Counts セクションが条件リンクに設定されています。

```
1 000000                .sect "Vars"
2               ; Vars section is conditionally linked
3                       .clink
4
5 000000 001A  X:.word 01Ah
6 000001 001A  Y:.word 01Ah
7 000002 001A  Z:.word 01Ah
8 000000                .sect "Counts"
9               ; Counts section is conditionally linked
10                       .clink
11
12 000000 001A  Xcount:.word 01Ah
13 000001 001A  Ycount:.word 01Ah
14 000002 001A  Zcount:.word 01Ah
15          ; By default, .text is unconditionally linked
16 000000                .text
17          ; Reference to symbol X cause the Vars section
18          ; to be linked into the COFF output
19 000000 3C00           MOV #0,AC0
20 000002 C000+          MOV AC0,X
```

.copy/.include ソース・ファイルのコピー

構文

.copy ["]*filename*["]
.include ["]*filename*["]

説明

.copy および **.include** の疑似命令は、アセンブラに対して、別のファイルからソース文を読み取るように指示します。コピー・ファイルからアセンブルされた文は、アセンブリ・リストに出力されます。インクルード・ファイルからアセンブルされた文は、アセンブリ・リストに出力されません。アセンブラは .copy/.include 疑似命令を検出すると、

1） 現行のソース・ファイルの文のアセンブルを中止します。

2) コピー・ファイルまたはインクルード・ファイルの文をアセンブルします。

3) コピー・ファイルまたはインクルード・ファイルの最後に到達すると、メイン・ソース・ファイルの文のアセンブルを、.copy または .include の疑似命令に続く文から再開します。

filename は、ソース・ファイルの名前を指定する必須パラメータです。二重引用符で囲むことができ、オペレーティング・システムの規則に従っていなければなりません。*filename* が数字で始まる場合、二重引用符が必要です。

完全なパス名（c:\dsp\file1.asm など）を指定することもできます。完全なパス名を指定しない場合、アセンブラは次の順序でファイルを検索します。

1） 現行のソース・ファイルが入っているディレクトリ

2） -i アセンブラ・オプションを使って指定したすべてのディレクトリ

3） 環境変数 A_DIR を使って指定したすべてのディレクトリ

-i オプションと A_DIR の詳細は、3.6 節「アセンブラ入力のための代替ファイルと代替ディレクトリの命名方法」（3-19 ページ）を参照してください。

.copy および .include の疑似命令は、コピーまたはインクルードされるファイル内にネストできます。アセンブラは、このようなネストを 32 段階まで許容します。さらに、ホスト・オペレーティング・システムは別の制限を設定している場合があります。アセンブラは、コピーされたファイルの行番号の前にコピーのレベルを識別する文字コードを置きます。たとえば、A は最初にコピーされたファイルを示し、B は 2 番目にコピーされたファイルを示します。

例 1

この例では、.copy 疑似命令を使用して他のファイルからソースを読み取ってアセンブルした後、アセンブラは現行のファイルへのアセンブルを再開します。

オリジナル・ファイル copy.asm は、ファイル byte.asm をコピーする .copy 文を含んでいます。copy.asm をアセンブルする際、アセンブラは byte.asm をリスト内の該当場所にコピーします（次のリストを参照してください）。コピー・ファイル byte.asm には、2 番目のファイル word.asm に対する .copy 文が入っています。

アセンブラは word.asm に対する .copy 文を検出すると、word.asm に切り替えてコピーおよびアセンブリの作業を続行します。その後、アセンブラは byte.asm 内の元の場所に戻って、コピーおよびアセンブリの作業を続行します。byte.asm のアセンブリが完了すると、アセンブラは copy.asm に戻って残りの文をアセンブルします。

copy.asm （ソース・ファイル）	**byte.asm （最初のコピー・ファイル）**	**word.asm （2 番目のコピー・ファイル）**
.data .space 29 **.copy "byte.asm"** ** Back in original file .pstring "done"	** In byte.asm .data .byte 32,1+ 'A' **.copy "word.asm"** ** Back in byte.asm .byte 67h + 3q	** In word.asm .data .word 0ABCDh, 56q

リスト・ファイル：

```
      1 000000          .data
      2 000000          .space 29
      3                 .copy "byte.asm"
A     1             ** In byte.asm
A     2 000001          .data
A     3 000002 0020     .byte 32,1+ 'A'
        000003 0042
A     4                 .copy "word.asm"
B     1             * In word.asm
B     2 000004          .data
B     3 000004 ABCD     .word 0ABCDh, 56q
        000005 002E
A     5             ** Back in byte.asm
A     5 000006 006A     .byte 67h + 3q
      4
      5             ** Back in original file
      6 000007 646F     .pstring "done"
        000008 6E65
```

例 2　この例では、.include 疑似命令を使用して他のファイルからソース文を読み取ってアセンブルした後、アセンブラは現行のファイルへのアセンブルを再開します。このメカニズムは、文がリスト・ファイルに出力されない点を除いて .copy 疑似命令に似ています。

include.asm （ソース・ファイル）	byte2.asm （最初のインクルード・ファイル）	word2.asm （2番目のインクルード・ファイル）
.data .space 29 **.include "byte2.asm"** ** Back in original file .string "done"	** In byte2.asm .data .byte 32,1+ 'A' **.include "word2.asm"** ** Back in byte2.asm .byte 67h + 3q	** In word2.asm .data .word 0ABCDh, 56q

リスト・ファイル：

```
1 000000          .data
2 000000          .space 29
3                 .include "byte2.asm"
4
5              ** Back in original file
6 000007 0064     .string "done"
  000008 006F
  000009 006E
  00000a 0065
```

.cpl_on/.cpl_off　直接アドレッシング・モードの選択

構文

.cpl_on
.cpl_off

説明

.cpl_on および **.cpl_off** 疑似命令は、CPL ステータス・ビットをモデル化します。

アセンブラは、CPL ステータス・ビットの値は追跡できません。このモードをアセンブラにモデル化するには、アセンブラ疑似命令とコマンド行オプション、あるいはそのいずれかを使用する必要があります。CPL ステータス・ビットの値を変更する命令の直後には、該当するアセンブラ疑似命令が必要です。

.cpl_on 疑似命令は、CPL ステータス・ビットを 1 にセットします。オブジェクト・コードを定義する他の命令や疑似命令の前に .cpl_on 疑似命令が指定されると、これは -mc コマンド行オプションを使うのと同等です。.cpl_off 疑似命令は、CPL ステータス・ビットを 0 にセットします。コマンド行オプションと疑似命令の間でコンフリクトが起こった場合、疑似命令が優先します。

.cpl_on および .cpl_off 疑似命令は、引数を取りません。

CPL モード（.cpl_on）では、直接メモリ・アドレッシングはスタック・ポインタ（SP）に対して相対的です。dma 構文は *SP（ *dma* ）で、*dma* には、定数またはリンク時に認識されたシンボル式が可能です。アセンブラは、*dma* の値を出力ビットにコード化します。

デフォルトでは（.cpl_off）、直接メモリ・アドレッシング（dma）は、データ・メモリ・ローカル・ページ・ポインタ・レジスタ（DP）に対して相対的です。dma 構文は @*dma* で、*dma* には、定数または再配置可能なシンボル式が可能です。アセンブラは、*dma* と DP レジスタの値の差を計算し、この差を出力ビットにコード化します。

アセンブラは、DP レジスタの値を追跡できません。しかし、直接メモリ・アクセス・オペランドをアセンブルするには、DP の値を仮定しなければなりません。したがって、アセンブラに DP 値をモデル化するには、.dp 疑似命令を使う必要があります。この疑似命令を発効した直後に、DP レジスタの値を変更する命令を続けます。

.cpl_on および .cpl_off 疑似命令の有効範囲は静的であり、アセンブリ・プログラムの制御フローには影響されません。.cpl_on 行と .cpl_off 行との間にあるすべてのアセンブリ・コードは、CPL モードでアセンブルされます。

.cstruct/ .endstruct/.tag C 構造タイプの宣言

構文

[*stag*]	**.cstruct**	[*expr*]
[*mem0*]	*element*	[*expr0*]
[*mem1*]	*element*	[*expr1*]
.	.	.
.	.	.
.	.	.
[*memn*]	**.tag** *stag*	[, *exprn*]
.	.	.
.	.	.
.	.	.
[*memN*]	*element*	[*exprN*]
[*size*]	**.endstruct**	
label	**.tag**	*stag*

説明

.cstruct 疑似命令は、(4-45 ページの「.cunion」とともに）アセンブリおよび C コード間で共通データ構造を共有しやすいようサポートします。.cstruct 疑似命令は、.struct 疑似命令と全く同様に使用できます。ただしこの疑似命令が、C struct のデータ・タイプ用の C コンパイラの使用するレイアウトに対応したデータ・レイアウトを保障される場合を除きます。特に、このようなタイプが複合的なデータ構造内にネストされている場合、.cstruct 疑似命令は C コンパイラが使用するのと同じ位置合わせおよび埋め込みを強制します。

.endstruct 疑似命令は、構造体の定義の終わりにマークを付けます。

.tag 疑似命令は、構造体の特性を *label* に付与します。これによりシンボル表記を簡略化し、他の構造体を含む構造体を定義できるようにします。.tag 疑似命令はメモリの割り当ては行いません。.tag 疑似命令の構造体タグ（stag）は、事前に定義しておかなければなりません。

以下は、.cstruct、.endstruct、および .tag 疑似命令で使われるパラメータの説明です。

- *stag* は構造体のタグです。値は、その構造体の始まりを示します。stag が指定されていない場合、アセンブラは構造体のメンバを、構造体の最上部からの絶対オフセットを値とするグローバル・シンボル・テーブルに入れます。.stag は、.struct の場合は任意ですが、.tag の場合は必須です。
- *expr* は構造体の始まりのオフセットを示す任意の式です。デフォルトの構造体の開始点は 0 です。
- $mem_{n/N}$ は、構造体のメンバを示す任意のラベルです。このラベルは絶対的で、構造体の始まりからのオフセットと等価です。構造体メンバに対するラベルはグローバル宣言ができません。

- *element* は、次のいずれかの記述子です。.string、.byte、.char、.int、.half、.short、.word、.long、.double、.float、.tag、または .field。.tag を除くすべては、メモリを初期化する典型的な疑似命令です。これらの疑似命令は、.struct 疑似命令の後に使用すると構造の要素サイズを表します。メモリの割り当ては行いません。stag を使う必要があるため、.tag 疑似命令は特別なケースです（stag の定義のように）。

- $expr_{n/N}$ は、記述される要素数を指定する任意の式です。この値はデフォルトで 1 となります。.string 要素のサイズは 1 バイト、.field 要素は 1 ビットと見なされます。

- *size* は、構造体全体のサイズを示す、任意のラベルです。

注 : .cstruct/.endstruct シーケンス内で使用できる疑似命令

.cstruct/.endstruct シーケンス内で使用できる疑似命令は、要素記述子、構造体および共用体のタグ、条件付きアセンブリ疑似命令、およびメンバ・オフセットをワード境界で位置合わせする .align 疑似命令だけです。空の構造体は正しくありません。

.cunion .endstruct/.tag C 構造タイプの宣言

構文

[*stag*]	**.cunion**	[*expr*]
[*mem0*]	*element*	[*expr0*]
[*mem1*]	*element*	[*expr1*]
.	.	.
.	.	.
.	.	.
[*memn*]	**.tag** *stag*	[, *exprn*]
.	.	.
.	.	.
.	.	.
[*memN*]	*element*	[*exprN*]
[*size*]	**.endstruct**	
label	**.tag**	*stag*

説明

.cunion 疑似命令（4-44 ページの「.cstruct」とともに）は、アセンブリおよび C コード間で共通データ構造を共有しやすいようサポートします。.cunion 疑似命令は、.union 疑似命令と全く同様に使用できます。ただし .cunion 疑似命令が、C union のデータ・タイプ用の C コンパイラの使用するレイアウトに対応したデータ・レイアウトを保障される場合を除きます。特に、union タイプが複合的なデータ構造内にネストされている場合、.cunion 疑似命令は C コンパイラが使用するのと同じ位置合わせおよび埋め込みを強制します。

.cstruct 定義には .cunion 定義を含めることができます。また、.cstructs と .cunions をネストすることもできます。

.endunion 疑似命令は、共用体の定義を終了します。

.tag 疑似命令は、構造体または共用体の特性を *label* に付与します。これによりシンボル表記を簡略化し、他の構造体または共用体を含む構造体または共用体を定義できるようにします。.tag 疑似命令はメモリの割り当ては行いません。.tag 疑似命令の構造体または共用体のタグは、事前に定義しておく必要があります。

- *utag* は共用体のタグです。値は、その共用体の始まりを示します。utag が指定されていない場合、アセンブラは共用体のメンバを、共用体の最上部からの絶対オフセットを値とするグローバル・シンボル・テーブルに入れます。この場合、各メンバには固有の名前が付いていなくてはなりません。

- *expr* は共用体の始まりのオフセットを示す任意の式です。共用体は、デフォルトでは 0 から開始するように設定されます。このパラメータはトップレベルの共用体でのみ使用できます。ネストされた構造体を定義しているときには使用できません。

- $mem_{n/N}$ は、共用体のメンバを示す任意のラベルです。このラベルは絶対的で、共用体の始まりからのオフセットと等価です。共用体のメンバに対するラベルはグローバル宣言できません。

- *element* は、次のいずれかの記述子です。.byte、.char、.double、field、.float、.half、.int、.long、.short、.string、.ubyte、.uchar、.uhalt、.uint、.ulong、.ushort、.uword 、および .word。また、element（要素）は、ネストされた構造体か共用体の完全な宣言、またはタグで宣言された構造体か共用体の完全な宣言でも構いません。これらの疑似命令は、.union 疑似命令の後に使用すると、要素のサイズを表します。メモリの割り当ては行いません。

- $expr_{n/N}$ は、記述される要素数を指定する任意の式です。この値はデフォルトで 1 となります。.string 要素のサイズは 1 バイト、.field 要素は 1 ビットと見なされます。

- *size* は、共用体全体のサイズを示す、任意のラベルです。

注 : .union/.endunion シーケンス内で使用できる疑似命令

.union/.endunion シーケンス内で使用できる疑似命令は、要素記述子、構造体および共用体のタグ、および条件付きアセンブリ疑似命令だけです。空の構造体は正しくありません。

.data .data セクションへのアセンブル

構文 **.data**

説明 **.data** 疑似命令はアセンブラに対して、.data セクションへのソース・コードのアセンブルを開始するように指示します。このとき .data セクションが現行のセクションになります。.data セクションは通常、データまたは事前に初期化された変数を含めるために使用されます。

C55x では、データはワード・アドレス可能です。

アセンブラは、.text をデフォルトのセクションとみなします。したがって、セクション制御疑似命令を使用しなければ、アセンブル開始時にアセンブラは .text セクションにコードをアセンブルします。

COFF セクションについての詳細は、第 2 章「共通オブジェクト・ファイル・フォーマットの概要」を参照してください。

例 この例では、コードを .data（ワード・アドレス可能）セクションと .text（バイト・アドレス可能）セクションへアセンブルします。

```
 1                ****************************************
 2                **  Reserve space in .data.           **
 3                ****************************************
 4 000000                .data
 5 000000                .space          0CCh
 6
 7                ****************************************
 8                **  Assemble into .text.              **
 9                ****************************************
10 000000                .text
11                INDEX   .set            0
12 000000 3C00          MOV #INDEX,AC0
13
14                ****************************************
15                **  Assemble into .data.              **
16                ****************************************
17 00000c                .data
18 00000d ffff Table: .word   -1   ; Assemble 16-bit
19                                 ; constant into .data.
20 00000e 00ff          .byte   0FFh ; Assemble 8-bit
21                                 ; constant into .data
22                ****************************************
23                **  Assemble into .text.              **
24                ****************************************
25 000002                .text
26 000002 D600          ADD Table,AC0,AC0
   000004 00"
27
28                ****************************************
29                **  Resume assembling into the .data  **
30                **  section at address 0Fh.           **
31                ****************************************
32 00000f                .data
```

.double/ .ldouble 倍精度浮動小数点値の初期化

構文

.double *value* [**, ...,** *valuen*]
.ldouble *value* [**, ...,** *valuen*]

説明

.double 疑似命令と **.ldouble** 疑似命令は、1 つ以上の IEEE 倍精度浮動小数点表示の浮動小数点値を現行のセクションに挿入します 。各 *value* は浮動小数点定数、または浮動小数点定数と等価なシンボルでなくてはなりません。各定数は、IEEE 倍精度 64 ビット形式の浮動小数点値に変換されます。浮動小数点定数は、ワード境界に位置合わせされます。

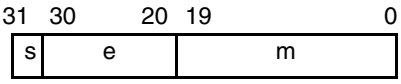
注 : これらの疑似命令はデータ・セクションで使用してください。

コード・セクションとデータ・セクションのアドレス指定のしかたは異なるので、C55x 命令が組み込まれているセクション内で .double および .ldouble 疑似命令を使用すると、実行時におけるデータへの不正アクセスにつながります。ですから、これらの疑似命令は、データ・セクション内でのみ使用されるようにすることをお勧めします。

値は、次の 3 つのフィールドで構成されます。

フィールド	意味
s	1 ビットの符号フィールド
e	11 ビットのバイアス指数
m	52 ビットの小数部

値は次の形式で、最上位のワードが最初に格納され、最下位のワードが 2 番目に格納されます。



.double または .ldouble の疑似命令を .struct/.endstruct のシーケンスで使用すると、その疑似命令はメンバのサイズを定義し、メモリの初期化は行いません。.struct/.endstruct の詳細は、4.9 節「アセンブル時シンボル疑似命令」（4-22 ページ）を参照してください。

例

この例は、.double 疑似命令と .ldouble 疑似命令を示しています。

```
1 000000             .data
2 000000 C520        .double  -1.0e25
  000001 8B2A
  000002 2C28
  000003 0291
2 000004 407C        .ldouble  456.0
  000005 8000
  000006 0000
  000007 0000
```

.dp　DP 値の指定

構文

.dp *dp_value*

説明

.dp 疑似命令は、DP レジスタの値を指定します。*dp_value* には、定数または再配置可能なシンボル式を指定できます。

デフォルトでは、直接メモリ・アドレッシング（dma）はデータ・メモリのローカル・ページ・ポインタ・レジスタ（DP）に対して相対的です。dma 構文は *@dma* で、*dma* には、定数または再配置可能なシンボル式が可能です。アセンブラは、*dma* と DP レジスタの値の差を計算し、この差を出力ビットにコード化します。

アセンブラは、DP レジスタの値を追跡できません。しかし、直接メモリ・アクセス・オペランドをアセンブルするには、DP の値を仮定しなければなりません。したがって、DP 値をモデル化するには、.dp 疑似命令を使う必要があります。この疑似命令を発効した直後に、DP レジスタの値を変更する命令を続けます。DP レジスタの値について、アセンブラに何の情報も提供されない場合、値は 0 とされます。

.drlist/ .drnolist　疑似命令のリスト出力の制御

構文

.drlist
.drnolist

説明

この 2 つの疑似命令を使用すると、アセンブラの疑似命令をリスト・ファイルに出力することを制御できます。

.drlist 疑似命令は、すべての疑似命令をリスト・ファイルに出力させます。

.drnolist 疑似命令は、以下の疑似命令をリスト・ファイルに出力させるのを抑止できます。.drnolist 疑似命令はマクロ内には影響しません。

- ❏ .asg
- ❏ .break
- ❏ .emsg
- ❏ .eval
- ❏ .fclist
- ❏ .fcnolist
- ❏ .mlist
- ❏ .mmsg
- ❏ .mnolist
- ❏ .sslist
- ❏ .ssnolist
- ❏ .var
- ❏ .wmsg

デフォルトでは、アセンブラは .drlist 疑似命令が指定されているものとして動作します。

例

この例は、指定された疑似命令のリスト作成を .drnolist がどのように禁止するかを示したものです。

ソース・ファイル :

```
.asg    0, x
.loop   2
.eval   x+1, x
.endloop

.drnolist

.asg    1, x
.loop   3
.eval   x+1, x
.endloop
```

リスト・ファイル :

```
       1                .asg    0, x
       2                .loop   2
       3                .eval   x+1, x
       4                .endloop
1                       .eval   0+1, x
1                       .eval   1+1, x
       5
       6                .drnolist
       7
       9                .loop   3
      10                .eval   x+1, x
      11                .endloop
```

.emsg/.mmsg/ .wmsg メッセージの定義

構文

.emsg *string*
.mmsg *string*
.wmsg *string*

説明

この 3 つの疑似命令を使用すると、独自のエラー・メッセージと警告メッセージを定義できます。アセンブラはエラーと警告の発行回数を記録し、リスト・ファイルの最後の行にその回数を出力します。

.emsg 疑似命令はアセンブラと同様にエラー・メッセージを標準出力デバイスに送ります。つまりエラー回数を 1 つずつインクリメントさせ、アセンブラがオブジェクト・ファイルの作成を禁止します。

.mmsg 疑似命令は、.emsg および .wmsg の疑似命令が行うのと同様に、アセンブル時メッセージを標準出力デバイスに送ります。ただし、エラー回数も警告回数も設定しません。また、アセンブラによるオブジェクト・ファイルの作成も禁止しません。

.wmsg 疑似命令は、.emsg 疑似命令が行うのと同様に、警告メッセージを標準出力デバイスに送ります。ただし、エラー回数でなく警告回数をインクリメントさせます。また、アセンブラによるオブジェクト・ファイルの作成も禁止しません。

例

この例では、メッセージ ERROR -- MISSING PARAMETER が標準出力デバイスに送られます。

ソース・ファイル :

```
          .global     PARAM
MSG_EX    .macro parm1
          .if    $symlen(parm1) = 0
          .emsg  "ERROR -- MISSING PARAMETER"
          .else
          ADD parm1,AC0,AC0
          .endif
          .endm

          MSG_EX PARAM

          MSG_EX
```

リスト・ファイル :

```
        1                .global PARAM
        2       MSG_EX   .macro  parm1
        3                .if     $symlen(parm1) = 0
        4               .emsg  "ERROR -- MISSING PARAMETER"
        5                .else
        6                ADD parm1,AC0,AC0
        7                .endif
        8                .endm
        9
       10 000000         MSG_EX PARAM
1                        .if     $symlen(parm1) = 0
1                       .emsg   "ERROR -- MISSING PARAMETER"
1                        .else
1         000000 D600    ADD PARAM,AC0,AC0
          000002 00!
1                        .endif
       11
       12 000003         MSG_EX
1                        .if     $symlen(parm1) = 0
1                       .emsg   "ERROR -- MISSING PARAMETER"
 "emsg.asm", ERROR! at line 12:[***** USER ERROR ***** -]
 ERROR -- MISSING PARAMETER
1                        .else
1                        ADD parm1,AC0,AC0
1                        .endif

 1 Error,  No Warnings
```

さらに、アセンブラにより次のメッセージが標準出力デバイスに送られます。

```
TMS32055xx COFF Assembler       Version x.xx
Copyright (c) 2001    Texas Instruments Incorporated
 PASS 1
 PASS 2
"emsg.asm", ERROR! at line 12:[***** USER ERROR ***** -] ERROR --
MISSING
  PARAMETER
                    .emsg "ERROR -- MISSING PARAMETER"


 1 Error,  No Warnings

Errors in source - Assembler Aborted
```

.end　アセンブリの終了

構文　　**.end**

説明　　**.end** 疑似命令は、アセンブルを終了させるためのオプションの疑似命令です。アセンブラは、.end 疑似命令以降のソース文はすべて無視します。

この疑似命令は、end-of-file と同じ働きをします。.end を使用して、デバッグ時にコードの途中の指定のポイントでアセンブルを停止させることができます。

例　　この例は、.end 疑似命令を使用したアセンブルの終了方法を示しています。アセンブラは、.end 疑似命令以降の .byte および .word 文を無視します。

ソース・ファイル :

```
        .data
START:  .space  300
TEMP    .set    15
        .bss    LOC1, 48h
        .data
        ABS AC0,AC0
        ADD #TEMP,AC0,AC0
        MOV AC0,LOC1
        .end
        .byte   4
        .word   CCCh
```

リスト・ファイル :

```
       1 000000                .data
       2 000000        START:  .space  300
       3               TEMP    .set    15
       4 000000                .bss    LOC1, 48h
       5 000000                .text
       5 000000 3200           ABS AC0,AC0
       6 000002 40F0           ADD #TEMP,AC0,AC0
       7 000004 C000-          MOV AC0,LOC1
       8                      .end
```

.fclist/ .fcnolist 偽の条件付きブロックのリスト作成の制御

構文

.fclist
.fcnolist

説明

次の 2 つの疑似命令を使用すると、偽の条件付きブロックのリスト作成を制御できます。

.fclist 疑似命令を使用すると、偽の条件付きブロック（コードを生成しない条件ブロック）のリストを作成できます。

.fcnolist 疑似命令は、.fclist 疑似命令が検出されるまで偽の条件付きブロックのリスト作成を禁止します。.fcnolist を使用すると、実際にアセンブルを行う条件ブロック内のコードのみがリストに表示されます。.if、.elseif、.else、および .endif 疑似命令はリストには表示されません。

デフォルトでは、すべての条件付きブロックのリストが作成されます。アセンブラは .fclist 疑似命令が使われているものとして動作します。

例

この例は、条件付きブロックをリストする場合とリストしない場合のアセンブリ・コードに対するソース・ファイルとリスト・ファイルを示しています。

ソース・ファイル :

```
AAA     .set  1
BBB     .set  0
        .fclist
        .if   AAA
        ADD #1024,AC0,AC0
        .else
        ADD #(1024*10),AC0,AC0
        .endif

        .fcnolist
        .if   AAA
        ADD #1024,AC0,AC0
        .else
        ADD #(1024*10),AC0,AC0
        .endif
```

リスト・ファイル :

```
       1                AAA    .set  1
       2                BBB    .set  0
       3                      .fclist
       4                      .if   AAA
       5 000000 7B04          ADD #1024,AC0,AC0
         000002 0000
       6                      .else
       7                      ADD #(1024*10),AC0,AC0
       8                      .endif
       9
      10                      .fcnolist
      11
      13 000004 7B04          ADD #1024,AC0,AC0
         000006 0000
```

.field フィールドの初期化

構文

.field *value* [**,***size in bits*]

説明

.field 疑似命令は、データ・セクションの単一ワード内の複数ビットで構成されているフィールドを初期化します。

注 : これらの疑似命令はデータ・セクションで使用してください。

コード・セクションとデータ・セクションのアドレス指定のしかたは異なるので、C55x 命令が組み込まれているセクション内で .field 疑似命令を使用すると、実行時におけるデータへの不正アクセスにつながります。ですから、これらの疑似命令は、データ・セクション内でのみ使用されるようにすることをお勧めします。

この疑似命令には 2 つのオペランドがあります。

- *value* は、必須パラメータです。ここには、評価されてフィールドに入れられる式が入ります。この値が再配置可能な場合、*size* の値は 16 または 24 でなければなりません。
- *size* は任意のパラメータです。ここにはフィールドを構成するビット数（1 ～ 32）が入ります。サイズを指定しないと、アセンブラはサイズが 16 ビットであるとみなします。16 以上の値を指定すると、フィールドはワード境界から始まります。*size* のビットに合わない値を指定すると、アセンブラはその値を切り捨てて、警告メッセージを発行します。たとえば .field 3,1 と指定した場合、アセンブラは 3 を 1 に切り捨てます。また、アセンブラは次の警告メッセージを出力します。

```
***warning - value truncated.
```

連続する .field 疑似命令は、現行のワードから始まる指定されたビット数に値をパックします。さらにフィールドが追加されるに従って、フィールドは、ワードの最上位から最下位に向かって順番にパックされます。アセンブラは、現行のワードに入りきらないサイズのフィールドを検出すると、そのワードを書き出して、SPC をインクリメントし、次のワードへのフィールドのパックを開始します。オペランドに 1 を指定して .align 疑似命令を使用すると、次の .field 疑似命令は新しいワードへのパックを開始します。

ラベルを使用する場合、ラベルは指定されたフィールドを含んだワードを指します。

.field を .struct/.endstruct のシーケンスで使用すると、.field はメンバのサイズを定義し、メモリは初期化しません。.struct/.endstruct の詳細は、4.9 節「アセンブル時シンボル疑似命令」（4-22 ページ）を参照してください。

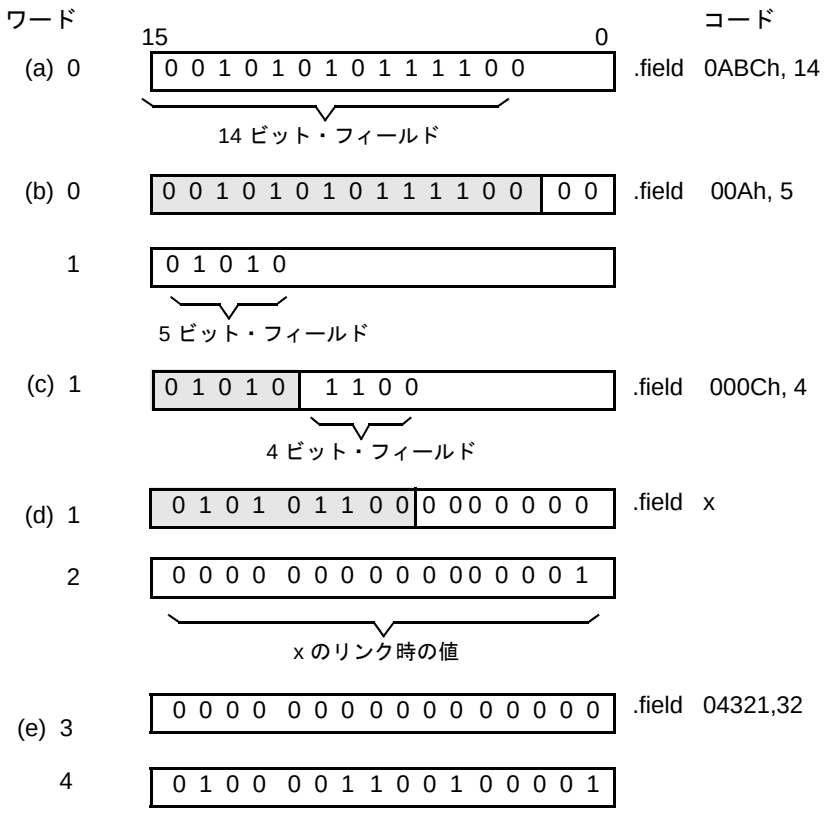
例

この例では、フィールドがどのようにしてワードにパックされるかを示しています。ワードが完全に埋められて次のワードへのパックが開始されるまで、SPC 値は変わらないことに注意してください。

```
 1 000000               .data
 2                ***********************************
 3                **    Initialize a 14-bit field.  **
 4                ***********************************
 5 000000 2AF0          .field  0ABCh, 14
 6
 7                ***********************************
 8                **    Initialize a 5-bit field    **
 9                **         in a new word.         **
10                ***********************************
11 000001 5000  L_F:.field  0Ah, 5
12
13                ***********************************
14                **    Initialize a 4-bit field    **
15                **         in the same word.      **
16                ***********************************
17 000001 5600  x:.field  0Ch, 4
18
19                ***********************************
20                **    16-bit relocatable field    **
21                **         in the next word.      **
22                ***********************************
23 000002 0001"         .field  x
24
25                ***********************************
26                **   Initialize a 32-bit field.   **
27                ***********************************
28 000003 0000          .field  04321h, 32
   000004 4321
```

図 4-4 は、この例の疑似命令がどのような効果をメモリに与えるかを示しています。

図 4-4. .field 疑似命令



.float/.xfloat 単精度浮動小数点値の初期化

構文

.float *value1* [**,** *...*, *valuen*]
.xfloat *value1* [**,** *...*, *valuen*]

説明

.float 疑似命令と **.xfloat** 疑似命令は、1 つ以上の浮動小数点定数の浮動小数点表現を現行のデータ・セクションに挿入します。value は、浮動小数点定数または浮動小数点定数と等価なシンボルでなくてはなりません。各定数は、IEEE 単精度 32 ビット形式の浮動小数点値に変換されます。

浮動小数点定数は、xfloat 疑似命令が使用されない限りは倍長ワード境界で位置合わせされます。.xfloat 疑似命令は .float 疑似命令と同じ働きをしますが、結果を倍長ワード境界に位置合わせしません。

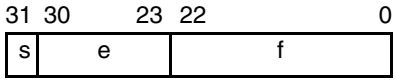
注 : これらの疑似命令はデータ・セクションで使用してください。

コード・セクションとデータ・セクションのアドレス指定のしかたは異なるので、C55x 命令が組み込まれているセクション内で .float および .xfloat 疑似命令を使用すると、実行時におけるデータへの不正アクセスにつながります。ですから、これらの疑似命令は、データ・セクション内でのみ使用されるようにすることをお勧めします。

32 ビット値は、次の 3 つのフィールドで構成されます。

フィールド	意味
s	1 ビットの符号フィールド
e	8 ビットのバイアス指数
m	23 ビットの小数部

値は次の形式で、最上位のワードが最初に格納され、最下位のワードが 2 番目に格納されます。



.float を .struct/.endstruct のシーケンスで使用すると、.float はメンバのサイズを定義し、メモリは初期化しません。.struct/.endstruct の詳細は、4.9 節「アセンブル時シンボル疑似命令」（4-22 ページ）を参照してください。

例

.float 疑似命令の例を以下に示します。

```
1 000000             .data
2 000000 E904        .float  -1.0e25
  000001 5951
3 000002 4040        .float  3
  000003 0000
4 000004 42F6        .float  123
  000005 0000
```

.global　グローバル・シンボルの識別

構文

.global *symbol1* [**,** *...*, *symboln*]
.def *symbol1*[**,** *...*, *symboln*]
.ref *symbol1*[**,** *...*, *symboln*]

説明

.global、**.def**、および **.ref** 疑似命令は、外部で定義されたグローバル・シンボル、または外部から参照できるグローバル・シンボルを識別します。

.def 疑似命令は、現行のモジュールで定義され、別のファイルでアクセスできるシンボルを特定します。アセンブラは、そのようなシンボルをシンボル・テーブルに入れます。

.ref 疑似命令は、現行のモジュールで使用され、他のモジュールで定義されるシンボルを特定します。リンカは、このようなシンボルの定義をリンク時に解決します。

.global 疑似命令は、必要に応じて .ref または .def として機能します。

グローバル・シンボルは、他のシンボルと同じ方法で定義されます。つまり、ラベルとして使用されるか、.set、.bss、.usect の疑似命令のいずれかにより定義されます。他のすべてのシンボルと同じように、グローバル・シンボルを 2 回以上定義すると、リンカは重複定義エラーを発行します。.ref は、モジュールがシンボルを使用するかどうかにかかわらず、必ずシンボル・テーブル・エントリを作成します。ただし .global は、モジュールが実際にシンボルを使用するときだけシンボル・テーブル・エントリを作成します。

シンボルは、次の 2 つの理由でグローバルと宣言されます。

- ❏ シンボルが現行のモジュールで定義されていない場合（マクロ、コピー、およびインクルード・ファイルを含む）、.global または .ref 疑似命令は、アセンブラに対してそのシンボルが外部モジュールで定義されていることを通知します。これにより、アセンブラは解決されていない参照のエラーを発行しません。リンク時にリンカは、他のオブジェクト・モジュールでそのシンボルが定義されていないかを調べます。
- ❏ シンボルが現行のモジュールで定義されている場合、.global または .def の疑似命令は、そのシンボルとその定義を外部の他のモジュールから使用できることを宣言します。このようなタイプの参照はリンク時に解決されます。

例

この例では 4 つのファイルを使います。

file1.lst と **file3.lst** は同等なファイルです。どちらのファイルもシンボル Init を定義し、他のモジュールに使用できるようにします。どちらのファイルも外部シンボル x、y、および z を使用します。file1.lst では、.global 疑似命令を使用してこれらのグローバル・シンボルを識別します。file3.lst では、.ref と .def を使用してこれらのシンボルを識別します。

file2.lst と **file4.lst** は同等なファイルです。どちらのファイルもシンボル x、y、および z を定義し、他のモジュールに使用できるようにします。どちらのファイルも外部シンボル Init を使用します。file2.lst では、.global 疑似命令を使用してこれらのグローバル・シンボルを識別します。file4.lst では、.ref と .def を使用してこれらのシンボルを識別します。

file1.lst:

```
 1            ; Global symbol defined in this file
 2                      .global INIT
 3            ; Global symbols defined in file2.lst
 4                      .global X, Y, Z
 5 000000       INIT:
 6 000000 7B00           ADD #86,AC0,AC0
   000002 5600
 7 000000                .data
 8 000000 0000!          .word   X
 9            ;        .
10            ;        .
11            ;        .
12                      .end
```

file2.lst:

```
 1            ; Global symbols defined in this file
 2                      .global X, Y, Z
 3            ; Global symbol defined in file1.lst
 4                      .global INIT
 5              X:      .set    1
 6              Y:      .set    2
 7              Z:      .set    3
 8 000000                .data
 9 000000 0000!          .word   INIT
10            ;        .
11            ;        .
12            ;        .
13                      .end
```

file3.lst:

```
 1            ; Global symbol defined in this file
 2                      .def    INIT
 3            ; Global symbols defined in file4.lst
 4                      .ref    X, Y, Z
 5 000000       INIT:
 6 000000 7B00           ADD #86,AC0,AC0
   000002 5600
 7 000000                .data
 8 000000 0000!          .word   X
 9            ;        .
10            ;        .
11            ;        .
12                      .end
```

file4.lst:

```
 1            ; Global symbols defined in this file
 2                      .def    X, Y, Z
 3            ; Global symbol defined in file3.lst
 4                      .ref    INIT
 5              X:      .set    1
 6              Y:      .set    2
 7              Z:      .set    3
 8 000000               .data
 9 000000 0000!         .word   INIT
10            ;        .
11            ;        .
12            ;        .
13                     .end
```

.half/.uhalf/ .short/.ushort 16 ビット整数の初期化

構文

.half*value1* [**,** *...*, *valuen*]
.uhalf *value1* [**,** *...*, *valuen*]
.short*value1* [**,** *...*, *valuen*]
.ushort *value1* [**,** *...*, *valuen*]

説明

.half、**.uhalf**、**.short**、および **.ushort** の疑似命令は、1 つ以上の値を現行のセクション内の連続した 16 ビット・フィールドに格納します。*value* には、次のものを指定できます。

- ❑ アセンブラが計算して、16 ビットの符号付きまたは符号なしの数値として扱う式。
- ❑ 二重引用符に囲まれた文字列。文字列内の各文字は、別々の値を表します。

注 : これらの疑似命令はデータ・セクションで使用してください。

コード・セクションとデータ・セクションのアドレス指定のしかたは異なるので、C55x 命令が組み込まれているセクション内で .half、.uhalf、.short および .ushort 疑似命令を使用すると、実行時におけるデータへの不正アクセスにつながります。ですから、これらの疑似命令は、データ・セクション内でのみ使用されるようにすることをお勧めします。

values は絶対式または再配置可能式のどちらかです。再配置可能式の場合、アセンブラは、適切なシンボルを参照する再配置エントリを作成します。これでリンカが参照を正しくパッチ（再配置）できるようになります。ユーザーは、変数またはラベルを指すポインタを使用してメモリを初期化できます。

アセンブラは、16 ビットを超える値は切り捨てます。ラベルを使用する場合、ラベルは最初に初期化されたワードを指します。

.half、.uhalf、.short または .ushort 疑似命令を .struct/.endstruct シーケンスで使用した場合、その疑似命令はメンバのサイズを定義し、メモリの初期化は行いません。.struct/.endstruct の詳細は、4.9 節「アセンブル時シンボル疑似命令」(4-22 ページ) を参照してください。

例

この例では、.half 疑似命令を使用して 16 ビット値（10、-1、abc、および a）をメモリに格納します。また .short 疑似命令を使用して、16 ビット値（8、-3、def、および b）をメモリに格納します。ラベル STRN の値は 106h で、これは初期化されたワードの先頭位置を表します。

```
 1 000000                .data
 2 000000                .space 100h * 16
 3
 4 000100 000A           .half  10, -1, "abc", 'a'
   000101 FFFF
   000102 0061
   000103 0062
   000104 0063
   000105 0061
 5 000106 0008  STRN     .short  8, -3, "def", 'b'
   000107 FFFD
   000108 0064
   000109 0065
   00010a 0066
   00010b 0062
```

.if/.elseif/ .else/ .endif　条件付きブロックのアセンブル

構文

.if *Boolean expression*
.elseif*Boolean expression*
.else
.endif

説明

これらの疑似命令を使用して、条件付きのコード・ブロックをアセンブルできます。条件付きアセンブリ・ブロックをネストできます。

.if 疑似命令は、条件付きブロックの始まりにマークを付けます。*Boolean expression* は必須パラメータで、疑似命令と同じ行に全体を指定しなければなりません。

- ❑ 式の計算結果が真（ゼロ以外）の場合、アセンブラは式に続くコードをアセンブルします（.elseif、.else、.endif のいずれかが同じ字句レベルで検出されるまで)。
- ❑ 式の計算結果が偽（ゼロ）のとき、アセンブラは .elseif（もしあれば）、.else（もしあれば)、または .endif（ .elseif も .else もなければ)に続くコードをアセンブルします。

.elseif 疑似命令は、.if 式が偽（0）で、かつ .elseif 式が真（ゼロ以外）の場合に、アセンブルするコード・ブロックを識別します。.elseif 式が偽のとき、アセンブラは、次の .elseif（もしあれば）、else（もし あれば）、または .endif（ .elseif も .else もなければ）へ処理を進めます。条件付きブロックでは .elseif 疑似命令は任意であり、複数の .elseif を使用することもできます。式が偽であり、かつ .elseif 文がない場合、アセンブラは、.else（もしあれば）または .endif の後に続くコードへと処理を進めます。

.else 疑似命令は、.if 式とそれまでのすべての .elseif 式が偽（0）のとき、アセンブラがアセンブルするコードのブロックを識別します。この疑似命令の使用は、条件付きブロックでは任意です。式が偽で .else 文がない場合は、アセンブラは .endif に続くコードへと処理を続けます。

.endif 疑似命令は、条件付きブロックの終わりにマークを付けます。

関係演算子の詳細は、3.11.4 項「条件式」(3-38 ページ）を参照してください。

例　条件付きアセンブリの例を以下に示します。

```
 1                SYM1   .set    1
 2                SYM2   .set    2
 3                SYM3   .set    3
 4                SYM4   .set    4
 5 000000               .data
 6                If_4:  .if     SYM4 = SYM2 * SYM2
 7 000000 0004          .byte   SYM4        ; Equal values
 8                      .else
 9                      .byte   SYM2 * SYM2 ; Unequal values
10                      .endif
11
12                If_5:.if     SYM1 <= 10
13 000001 000a         .byte   10         ; Less than / equal
14                      .else
15                      .byte   SYM1       ; Greater than
16                      .endif
17
18                If_6:.if     SYM3 * SYM2 != SYM4 + SYM2
19                      .byte   SYM3 * SYM2  ; Unequal value
20                      .else
21 000002 0008         .byte   SYM4 + SYM4  ; Equal values
22                      .endif
23
24                If_7:.if SYM1 = 2
25                      .byte   SYM1
26                      .elseif SYM2 + SYM3 = 5
27 000003 0005         .byte   SYM2 + SYM3
28                      .endif
```

.int/.uint/ .word/ .uword　16 ビット整数の初期化

構文

.int*value1* [**,** *...*, *valuen*]
.uint *value1* [**,** *...*, *valuen*]
.**word***value1* [**,** *...*, *valuen*]
.uword *value1* [**,** *...*, *valuen*]

説明

.int、**.uint**、**.word**、および **.uword** の疑似命令の機能は同じであり、1 つ以上の値を現行のセクション内の連続した 16 ビット・フィールドに格納します。*value* には、次のいずれかを指定できます。

- ❑ アセンブラが計算して、16 ビットの符号付きまたは符号なしの数値として扱う式。
- ❑ 二重引用符に囲まれた文字列。文字列内の各文字は、別々の値を表します。

注 : これらの疑似命令はデータ・セクションで使用してください。

コード・セクションとデータ・セクションのアドレス指定のしかたは異なるので、C55x 命令が組み込まれているセクション内で .int、.uint、.word および .uword 疑似命令を使用すると、実行時におけるデータへの不正アクセスにつながります。ですから、これらの疑似命令は、データ・セクション内でのみ使用されるようにすることをお勧めします。

values は絶対式または再配置可能式のどちらかです。再配置可能式の場合、アセンブラは、適切なシンボルを参照する再配置エントリを作成します。これでリンカが参照を正しくパッチ（再配置）できるようになります。ユーザーは、変数またはラベルを指すポインタを使用してメモリを初期化できます。

ラベルを使用すると、ラベルは初期化される最初のワードを指します。

これらの疑似命令を .struct/.endstruct シーケンスで使用した場合、その疑似命令はメンバのサイズを定義し、メモリの初期化は行いません。.struct/.endstruct の詳細は、4.9 節「アセンブル時シンボル疑似命令」（4-22 ページ）を参照してください。

例 1 この例では、.int 疑似命令を使用してワードを初期化します。

```
 1 000000             .data
 2 000000             .space 73h
 3 000000             .bss   PAGE, 128
 4 000080             .bss   SYMPTR, 3
 5 000000          .text
 6 0000007600 INST: MOV #86,AC0
   0000025608
 7 000007          .data
 8 000008 000A       .int   10, SYMPTR, -1, 35 + 'a'
   000009 0080-
   00000a FFFF
   00000b 0084
```

例 2 この例では、.word 疑似命令を使用してワードを初期化します。シンボル WordX は、最初に確保されたワードを指します。

```
 1 000000               .data
 1 000000 0C80  WORDX: .word  3200, 1 + 'AB', -0AFh, 'X'
   000001 4143
   000002 FF51
   000003 0058
```

.ivec 割り込みテーブル・エントリの初期化

構文 [*label*:]**.ivec** [*address* [, *stack mode*]]

説明 **.ivec** 疑似命令を使用して、割り込みベクター・テーブルの入力を初期化します。

この疑似命令には次のオペランドを指定できます。

- *label* を指定する場合、疑似命令に関連付けられているコード（バイト）アドレスに割り当てられます。他の疑似命令のようにデータ (ワード) アドレスではありません。
- *address* は、割り込みサービス・ルーチンのアドレスを指定します。アドレスが指定されていない場合、0 が使われます。
- リセット・ベクターに限り *stack mode* を指定できます。リセット・ベクターは割り込みベクター・テーブル内で最初の .ivec にならなければなりません。スタック・モードは以下のように識別できます。

C54X_STK	この値は、変換された C54x コードが必要とする 32 ビット・スタックを指定します。スタック・モードに値がない場合、これがデフォルトです。
USE_RETA	この値は、16 ビット・プラス・レジスタの高速リターン・モードを指定します。
NO_RETA	この値は、16 ビットの低速リターン・モードを指定します。

スタック・モードについての詳細は、TMS320C55x DSP CPU リファレンス・ガイドを参照してください。これらのシンボル名は、大文字小文字のどちらでも書き込めます。

.ivec 疑似命令は、8 バイト境界に SPC を位置合わせします。従って、2 つの .ivec エントリの間に命令を挿入する必要はありません。この位置合わせのために追加される空間にはすべて NOP 命令が埋め込まれます。

一般的に、他のデータ定義疑似命令（.word など）を含むセクションは、データ・セクションとしての特徴を持っています。データ・セクションはワード・アドレス可能で、コードを持つことができません。.ivec 疑似命令をもつセクションは、コード・セクションとしての特徴を持ち（バイト・アドレス可能）、他の命令を含む場合もあります。命令と同様に、.ivec は他のデータ定義疑似命令と混合できません。

アセンブラは、.ivec 疑似命令および 4 バイトより大きな命令をもつセクションを検出すると、警告を発行します。これにより、割り込みベクターの持つ命令が大きすぎる場合、最後の 4 バイトがはみ出すのを防ぎます。

アセンブラは、.ivec の直後に複数の命令を検出した場合にも警告を発行します。ISR への分岐前には、1 つの命令だけしか実行されません。

.ivec 疑似命令を含むセクションは、割り込みベクター・セクションとしてマークが付けられます。リンカはこのようなセクションを認識し、通常のコード・セクションに行うように、セクションの最後に非パラレル NOP を追加しません。

例　この例は、.ivec 疑似命令の使用法を示しています。

```
       .sect  "vectors"          ; start vectors section
       .ref   start,nmi_isr,isr2 ; symbols referenced
                                 ; from other files
       .def   rsv,no_isr         ; symbols defined in this
                                 ; file
rsv:   .ivec  start,c54x_stk     ; C54x compatibility
                                 ; stack mode
nmi    .ivec  nmi_isr            ; standard usage
int3   .ivec                     ; one way to skip a vector
int4   .ivec  no_isr             ; better way to skip a vector
; ... and so on.  Fill out all 32 vectors.
int31  .ivec  no_isr             ; last vector
       .text                     ; change to text section
no_isr B      no_isr             ; default ISR
```

int3 と int4 の違いに注意してください。int 3 ベクトルが返される場合、この例では 0 に分岐し、結果は予想不能です。しかし int 4 ベクトルが返される場合、この例では no_isr スピン・ループに分岐し、予想可能な結果を生成します。

.label ロード時アドレス・ラベルの作成

構文

.label*symbol*

説明

.label 疑似命令は、カレント・セクション内の実行時アドレスではなく、ロード時のアドレスを参照する特殊 *symbol* を定義します。アセンブラで作成されるほとんどのセクションは、再配置可能なアドレスを持っています。アセンブラは、各セクションがゼロで開始されているとしてセクションをアセンブルし、リンカは、それらをロードおよび実行されるアドレスに再配置します。

一部のアプリケーションでは、セクションがロードされるアドレスと実行されるアドレスを別にした方がよい場合があります。たとえば、パフォーマンスを左右するコードのブロックを低速のオフチップ・メモリにロードしてメモリ空間を節約し、それを実行するときには高速のオンチップ・メモリに移動するような場合です。

このようなセクションは、リンク時に 2 つのアドレスに割り当てられます。ロード・アドレスと実行アドレスです。このセクションで定義されるすべてのラベルは実行時アドレスを参照するように再配置され、コード実行時にこのセクションに対する参照（ブランチなど）は正しくなります。

.label 疑似命令は、ロード時のアドレスを参照する特殊ラベルを作成します。この機能は主として、セクションをロード時の位置から実行時の位置へ移動するコードに対して、セクションがロードされたときの位置を設定するときに便利です。

例

この例は、ロード・アドレス・ラベルの使用法を示しています。

```
      .sect  ".EXAMP"
      .label EXAMP_LOAD ; load address of section.
START:                  ; run address of section.
      <code>
FINISH:                 ; run address of section end.
      .label EXAMP_END  ; load address of section end.
```

リンカにおける実行時アドレスおよびロード・アドレスの詳細については、8.10 節「セクションのロード時および実行時アドレスの指定方法」（8-45 ページ）を参照してください。

.length/.width リスト・ページ・サイズの設定

構文

.length*page length*
.width*page width*

説明

.length 疑似命令を使用すると、出力リスト・ファイルのページの長さを設定できます。この疑似命令は、現行のページとそれに続くページに影響を及ぼします。ページの長さを再設定するには、別の .length 疑似命令を使用します。

- ❑ デフォルト長さ : 60 行
- ❑ 最小長さ : 1 行
- ❑ 最大長さ : 32767 行

.width 疑似命令を使用すると、出力リスト・ファイルのページの幅を設定できます。この疑似命令は、次にアセンブルされる行とそれに続く行に影響を及ぼします。ページの幅を再設定するには、別の .width 疑似命令を使用します。

- ❑ デフォルト幅 : 80 文字
- ❑ 最小幅 : 80 文字
- ❑ 最大幅 : 200 文字

ここで言う幅とは、リスト・ファイルの完全な 1 行を指します。行カウンタの値、SPC 値、およびオブジェクト・コードは 1 行の幅の一部となります。ページ幅を超えるコメント及びソース文の他の部分は、リスト内で切り捨てられます。

アセンブラは、.width および .length 疑似命令のリストを作成しません。

例

この例では、ページ長とページ幅を変える方法を示しています。

```
*****************************************
**        Page length = 65 lines.          **
**        Page width  = 85 characters.     **
*****************************************
          .length    65
          .width     85

*****************************************
**        Page length = 55 lines.          **
**        Page width  = 100 characters.    **
*****************************************
          .length    55
          .width     100
```

.list/.nolist　ソース・リスト出力の開始／停止

構文

.list
.nolist

説明

2 つの疑似命令を使用して、ソース・リストの出力を制御できます。

.list 疑似命令によって、ソース・リストを出力できます。

.nolist 疑似命令は、.list 疑似命令を検出するまでソース・リストの出力を抑止します。.nolist 疑似命令を使用すると、アセンブル時間とソース・リストのサイズを削減できます。また、この疑似命令をマクロ定義で使用すると、リストへのマクロ展開の出力を抑止できます。

アセンブラは、.list および .nolist 疑似命令、または .nolist 疑似命令より後に現れるソース文を出力しません。ただし、行カウンタはインクリメントしていきます。.list/.nolist 疑似命令はネストできます。リスト出力を回復するには、各 .nolist に対応する .list が必要です。

デフォルトでは、ソース・リストはリスト・ファイルに出力されます。アセンブラは .list 疑似命令が指定されているものとして動作します。ただし、アセンブラを起動する際にリスト・ファイルを要求しないと、アセンブラは .list 疑似命令を無視します。

例

この例では、.copy 疑似命令を使用して他のファイルからのソース文を挿入します。アセンブラは、最初にこの疑似命令を検出すると、コピーされたソース行をリスト・ファイルに出力します。2 回目に .copy を検出したときには、.nolist 疑似命令はアセンブルされているので、コピーされたソース行をリストに出力しません。.nolist 疑似命令、2 番目の .copy 疑似命令、.list 疑似命令は、リスト・ファイルには出力されません。また、ソース文がリストに出力されない場合でも行カウンタはインクリメントされることに注意してください。

ソース・ファイル :

```
   .copy  "copy2.asm"
* Back in original file
   NOP
   .nolist
   .copy  "copy2.asm"
   .list
* Back in original file
   .string  "Done"
```

リスト・ファイル：

```
      1                    .copy   "copy2.asm"
A     1             * In copy2.asm (copy file)
A     2 000000              .data
A     3 000000 0020         .word 32, 1 + 'A'
      4 000001 0042
      2             * Back in original file
      3 000000              .text
      4 000000 90           NOP
      9             * Back in original file
     10 000004              .data
     11 000004 0044         .string "Done"
        000005 006F
        000006 006E
        000007 0065
```

.localalign　ロード時アドレス・ラベルの作成

構文　　**.localalign**

説明　アセンブラ疑似命令 .localalign は、localrepeat 命令の直前に置かれることを意味します。これにより、ループ本体の最初の命令は、4 バイト境界に位置合わせされ、これにより最大の localrepeat ループ・サイズが可能になります。これは、充分なシングル・サイクルの NOP 命令を挿入することにより、位置合わせを正しくします。また、4 バイト境界の位置合わせを現行のセクションに適用し、リンカは該当するループ本体に必要な位置合わせを行います。パラメータは取りません。

例 1　この例は、.localalign 疑似命令を使わない localrepeat ループの動作を示しています。

```
main:nop
      nop
      nop
      localrepeat {
          ac1 = #5
          ac2 = ac1
      }
```

上記ソース・コードはこの出力を作成します。

```
       1 000000 20    main:nop
       2 000001 20             nop
       3 000002 20             nop
       4 000003 4A82           localrepeat {
       5 000005 3C51                   AC1 = #5
       6 000007 2212                   AC2 = AC1
       7                       }
```

例 2　この例は、.localalign を追加した後、例 1 のソース・コードを示しています。

```
main:nop
      nop
      nop
      .localalign
      localrepeat {
          ac1 = #5
          ac2 = ac1
      }
```

この例は、5 行目の localrepeat の前に位置合わせしたループ本体を作成しています。これにより、6 行目で開始するループ本体は 4 バイト境界に位置合わせされます。アドレスは 0x5 から 0x8 になります。

```
 1 000000 20    main:   nop
 2 000001 20            nop
 3 000002 20            nop
 4                      .localalign
 5 000006 4A82          localrepeat {
 6 000008 3C51                  AC1 = #5
 7 00000a 2212                  AC2 = AC1
 8                      }
```

ディスアセンブリは、NOP の挿入方法を示しています。

```
TEXT Section .text, 0xC bytes at 0x0
000000: 20              NOP
000001: 20              NOP
000002: 20              NOP
000003: 5e80_21         NOP_16 || NOP
000006: 4a82            RPTBLOCAL 0xa
000008: 3c51               MOV #5,AC1
00000a: 2212               MOV AC1,AC2
```

.localalign 疑似命令を使って（または手動で）ループを位置合わせすることにより、localrepeat ループのサイズを最大にすることができます。この位置合わせをしないと、C55x プロセッサの命令バッファ・キュー（IBQ）にロードするために、ループを数バイト短くしなければならない場合があります。

短いループで疑似命令を使える一方、.localalign は localrepeat サイズの限界に近い localrepeat ループで使用するだけで構いません。

.lock_on/ .lock_off　リード・モディファイ・ライト命令の範囲を有効化

構文

.lock_on
.lock_off

説明

.lock_on および **.lock_off** の疑似命令は、リード・モディファイ・ライト命令を使用する範囲を識別します。この範囲内では lock 修飾子をすべてのリード・モディファイ・ライト命令とともに指定できます。lock 修飾子が .lock_on ブロック内にあるリード・モディファイ・ライト命令とともに指定されない場合、アセンブラは診断の注釈を発行し、操作がアトミック処理されないことを示します。.lock_on および .lock_off 疑似命令の範囲外では、lock 修飾子は不正で、リード・モディファイ・ライト命令にはフラグが立てられません。

これらの疑似命令は、メモリ位置へのアトミック・アクセスを必ず実行する必要のあるコードの重要な領域（通常セマフォ）に挿入されます。

デフォルトでは、アセンブラはすべてのコードを .lock_on/.lock_off 領域の外側にあるものとして処理します。

.long/.ulong/ .xlong　32 ビット整数の初期化

構文

.long*value1*[, ...,*valuen*]
.ulong *value1*[, ...,*valuen*]
.xlong*value1*[, ...,*valuen*]

説明

.long、**.ulong** および **.xlong** 疑似命令は、1 つ以上の 32 ビット値を現行のセクション内の連続したワードに格納します。最上位のワードを最初に格納します。.long 疑似命令および .ulong 疑似命令は結果を倍長ワード境界で位置合わせしますが、.xlong 疑似命令はそのような位置合わせは行いません。*value* には、次のものを指定できます

- ❏ アセンブラが計算して、32 ビットの符号付きまたは符号なしの数値として扱う式。
- ❏ 二重引用符に囲まれた文字列。文字列内の各文字は、別々の値を表します。

注 : これらの疑似命令はデータ・セクションで使用してください。

コード・セクションとデータ・セクションのアドレス指定のしかたは異なるので、C55x 命令が組み込まれているセクション内で .long、.ulong および .xlong 疑似命令を使用すると、実行時におけるデータへの不正アクセスにつながります。ですから、これらの疑似命令は、データ・セクション内でのみ使用されるようにすることをお勧めします。

value は絶対式または再配置可能式のどちらかです。再配置可能式の場合、アセンブラは、適切なシンボルを参照する再配置エントリを作成します。これでリンカが再配置された値を使って参照を正しくパッチ（再配置）できるようになります。ユーザーは、変数またはラベルを指すポインタを使用してメモリを初期化できます。

ラベルを使用すると、ラベルは初期化される最初のワードを指します。

これらの疑似命令を .struct/.endstruct シーケンスで使用した場合、その疑似命令はメンバのサイズを定義し、メモリの初期化は行いません。.struct/.endstruct の詳細は、4.9 節「アセンブル時シンボル疑似命令」（4-22 ページ）を参照してください。

例

この例は、.long 疑似命令と .xlong 疑似命令が倍長ワードをどのように初期化するかを示しています。

```
1 000000                .data
2 000000 0000 DAT1: .long  0ABCDh, 'A' + 100h, 'g', 'o'
  000001 ABCD
  000002 0000
  000003 0141
  000004 0000
  000005 0067
  000006 0000
  000007 006F
3 000008 0000       .xlong  DAT1, 0AABBCCDDh
  000009 0000"
  00000a AABB
  00000b CCDD
4 00000c     DAT2:
```

.loop/.break/ .endoop　コード・ブロックの反復アセンブル

構文

.loop[*well-defined expression*]
.break[*Boolean expression*]
.endloop

説明

これらの疑似命令を使用すると、コードのブロックを反復してアセンブルできます。

.loop 疑似命令は、反復使用が可能なコード・ブロックを開始します。任意の式には、ループ・カウント（ループに含まれるコードがアセンブリを繰り返す回数）を指定します。式の指定がない場合は、アセンブラが真（ゼロ以外）の式をもった .break 疑似命令か、式をもたない .break 疑似命令を最初に検出しない限り、ループ回数はデフォルトの 1024 となります。

.break 疑似命令とその式は任意です。式が偽（0）のとき、ループは継続します。式が真（ゼロ以外）または省略されたときには、アセンブラはループを抜け出し、.endloop 疑似命令の後のコードのアセンブルを開始します。

.endloop 疑似命令は、反復使用が可能なコード・ブロックの最後をマークします。アセンブラは、ループを抜け出すか、またはループの最後の反復が終了すると、.endloop の後のコードのアセンブルを続けます。

例

この例は、.loop、.break、および .endloop 疑似命令を .eval 疑似命令と一緒に使用する方法を示しています。

```
       1 000000           .data
       2                  .eval       0,x
       3             LAB_1 .loop
       4                  .word       x*100
       5                  .eval       x+1, x
       6                  .break      x = 6
       7                  .endloop
1        000000 0000      .word       0*100
1                         .eval       0+1, x
1                         .break      1 = 6
1        000001 0064      .word       1*100
1                         .eval       1+1, x
1                         .break      2 = 6
1        000002 00C8      .word       2*100
1                         .eval       2+1, x
1                         .break      3 = 6
1        000003 012C      .word       3*100
1                         .eval       3+1, x
1                         .break      4 = 6
1        000004 0190      .word       4*100
1                         .eval       4+1, x
1                         .break      5 = 6
1        000005 01F4      .word       5*100
1                         .eval       5+1, x
1                         .break      6 = 6
```

.macro/.endm マクロの定義

構文

macname **.macro** [*parameter1*] [, ... *parametern*]
model statements or macro directives
.endm

説明

マクロの定義には、**.macro** 疑似命令を使用します。

マクロはプログラム内のどこにでも定義できますが、使用する前に必ず定義する必要があります。マクロは、ソース・ファイルの先頭でも、.include/.copy ファイルでも、またはマクロ・ライブラリでも定義できます。

macname	マクロに名前を付けます。名前は、必ずソース文のラベル・フィールドに入れます。
.macro	ソース文がマクロ定義の第 1 行目であることを特定します。.macro は、必ず命令コード・フィールドに置かなければなりません。

[parameters]	.macro 疑似命令のオペランドとして使用される、指定が任意の置換シンボルです。
model statements	マクロが呼び出されるたびに実行される命令、またはアセンブラ疑似命令です。
macro directives	マクロ展開の制御に使用されます。
.endm	マクロ定義の終わりにマークを付けます。

マクロについては、第 5 章「マクロ言語」で詳しく説明します。

.mlib マクロ・ライブラリの定義

構文

.mlib["]*filename*["]

説明

.mlib 疑似命令は、アセンブラにマクロ・ライブラリ名を提供します。マクロ・ライブラリは、マクロ定義を含んだファイルの集合です。マクロ定義ファイルは、アーカイバにより（アーカイブまたはライブラリと呼ばれる）1 つのファイルに結合されます。

マクロ・ライブラリのファイルには、そのファイル名に対応した 1 つのマクロ定義が含まれます。マクロ・ライブラリ・メンバの *filename* はマクロ名と同じで、かつ拡張子は .asm でなければなりません。*filename* は、ホスト・オペレーティング・システムの規則に従って指定しなければなりません。*filename* は二重引用符で囲むこともできます。完全なパス名（c:\320tools\macs.lib など）を指定することもできます。完全なパス名を指定しない場合、アセンブラは次の順序でファイルを検索します。

1） 現行のソース・ファイルが入っているディレクトリ

2） アセンブラ・オプション -i を使って指定したすべてのディレクトリ

3） 環境変数 C55X_A_DIR または A_DIR を使って指定したすべてのディレクトリ

-i オプション、C55X_A_DIR、および A_DIR の詳細は、3.6 節「アセンブラ入力のための代替ファイルと代替ディレクトリの命名方法」（3-19 ページ）を参照してください。

.mlib 疑似命令を検出すると、アセンブラは *filename* の指定したライブラリをオープンし、その内容のテーブルを作成します。アセンブラは、個々のライブラリ・メンバ名をライブラリ・エントリとして命名コード・テーブルに入れます。同じ名前の命令コードやマクロがすでに存在する場合には再定義します。これらのマクロの中の 1 つが呼び出されると、アセンブラはライブラリからそのエントリを抽出し、マクロ・テーブルにロードします。アセンブラはライブラリ・エントリを他のマクロと同じように展開しますが、そのソース・コードをリストに出力しません。抽出されるのは、ライブラリから実際に呼び出されたマクロだけです。

マクロおよびマクロ・ライブラリの詳細は、第 5 章「マクロ言語」を参照してください。

例

この例では、incr および decr の 2 つのマクロを定義するマクロ・ライブラリを作成します。ファイル incr.asm には incr の定義が、ファイル decr.asm には decr の定義がそれぞれ含まれます。

incr.asm	decr.asm

* Macro for incrementing
incr .macro
ADD #1,AC0,AC0
ADD #1,AC1,AC1
ADD #1,AC2,AC2
ADD #1,AC3,AC3
.endm
	* Macro for decrementing
	decr .macro
	SUB #1,AC0,AC0
	SUB #1,AC1,AC1
	SUB #1,AC2,AC2
	SUB #1,AC3,AC3
	.endm

アーカイバを使用してマクロ・ライブラリを作成します。

```
ar55 -a mac incr.asm decr.asm
```

これで、.mlib 疑似命令を使用してマクロ・ライブラリを参照し、マクロ incr と decr を定義できるようになります。

```
       1                .mlib    "mac.lib"
       2 000000         incr              ; Macro call
1        000000 4010       ADD #1,AC0,AC0
1        000002 4011       ADD #1,AC1,AC1
1        000004 4012       ADD #1,AC2,AC2
1        000006 4013       ADD #2,AC3,AC3
       3 000008         decr              ; Macro call
1        000008 4210       SUB #1,AC0,AC0
1        00000a 4211       SUB #1,AC1,AC1
1        00000c 4212       SUB #1,AC2,AC2
1        00000e 4213       SUB #1,AC3,AC3
```

.mlist/.mnolist マクロ展開リスト作成の開始／停止

構文

```
.mlist
.mnolist
```

説明

この 2 つの疑似命令を使用して、リスト・ファイルにあるマクロ展開および反復使用可能なブロック展開のリスト作成を制御できます。

.mlist 疑似命令を使用すると、マクロと .loop/.endloop ブロックの展開をリスト・ファイルに作成します。

.mnolist 疑似命令は、マクロと .loop/.endloop ブロックの展開のリスト・ファイルへの出力を抑止します。

デフォルトでは、アセンブラは、.mlist 疑似命令がすでに指定されているものとして動作します。

マクロおよびマクロ・ライブラリの詳細は、第 5 章「マクロ言語」を参照してください。

例

この例では、*STR_3* という名前のマクロを定義します。マクロが 2 回目に呼び出されるときには .mnolist 疑似命令がアセンブルされているので、マクロ展開はリストに出力されません。3 回目に呼び出されるときには .mlist 疑似命令がアセンブルされているので、マクロ展開はリストに出力されます。

```
                1            STR_3  .macro   P1, P2, P3
                2                   .data
                3                   .string ":p1:", ":p2:", ":p3:"
                4                   .endm
                5
                6 000000            STR_3 "as", "I", "am"
1                 000000            .data
1                 000000 003A       .string ":p1:", ":p2:", ":p3:"
                  000001 0070
                  000002 0031
                  000003 003A
                  000004 003A
                  000005 0070
                  000006 0032
                  000007 003A
                  000008 003A
                  000009 0070
                  00000a 0033
                  00000b 003A
                7                   .mnolist
                8 00000c            STR_3 "as", "I", "am"
                9                   .mlist
               10 000018            STR_3 "as", "I", "am"
1                 000018            .data
1                 000018 003A       .string ":p1:", ":p2:", ":p3:"
                  000019 0070
                  00001a 0031
                  00001b 003A
                  00001c 003A
                  00001d 0070
                  00001e 0032
                  00001f 003A
                  000020 003A
                  000021 0070
                  000022 0033
                  000023 003A
```

.newblock　ローカル・シンボル・ブロックの終了

構文　　**.newblock**

説明

.newblock 疑似命令を使用すると、現在定義されているローカル・ラベルの定義が取り消されます。ローカル・ラベルは本来一時的なものですが、.newblock 疑似命令はローカル・ラベルをリセットし、スコープを終了させます。

ローカル・ラベルは、$n という形式のラベルです。ここで、n は 1 桁の 10 進数字を表します。ローカル・ラベルは、ほかのラベルと同様に命令ワードを指します。しかし、ほかのラベルと違って式の中で使用することはできません。ローカル・ラベルはシンボル・テーブルには含まれません。

ローカル・ラベルは、ワイルドカード ? を使って定義することもできます。ラベル ? の書式のローカル・ラベルは、アセンブラが ? を固有のラベル識別子に置き換えます。

ローカル・ラベルを定義し、使用した後、.newblock 疑似命令を使用してローカル・ラベルをリセットする必要があります。.text、.data、および指定されたセクションもローカル・ラベルをリセットします。インクルード・ファイル内で定義されたローカル・ラベルは、ローカル・ファイルの外部では無効です。

例 この例は、ローカル・ラベル $1 を宣言し、リセットし、再び宣言する方法を示しています。

```
 1                            .ref   ADDRA, ADDRB, ADDRC
 2                 foo    .set   76h
 3
 4   000000 A000! LABEL1: MOV ADDRA,AC0
 5   000002 7C00          SUB #foo,AC0
     000004 7600
 6   000006 62200            BCC $1,AC0 < #0
 7   000008 A000!         MOV ADDRB,AC0
 8   00000a 4A02          B $2
 9
10   00000c A000! $1      MOV ADDRA,AC0
11   000003 D600  $2      ADD ADDRC,AC0,AC0
     000010 00!
12                         .newblock  ; Undefine $1 to reuse
13   000011 6120          BCC $1,AC0 < #0
14   000013 C000!         MOV AC0,ADDRC
15   000015 20    $1      NOP
```

.noremark/ .remark 注釈の制御

構文

.noremark *num*
.remark [*num*]

説明 **.noremark** 疑似命令は、*num* により特定されるアセンブラ注釈を抑止します。注釈はアセンブラの情報メッセージであり、警告ほど深刻ではありません。注釈の説明については、7.7 節「アセンブラ・メッセージ」（7-35 ページ）を参照してください。

この疑似命令は、-r[*num*] アセンブラ・オプションを使用することと同じです。

.remark 疑似命令は、事前に抑止された注釈を再び可能にします。

例 この例は、R5002 注釈の抑止方法を示しています。

オリジナル・リスト・ファイル :

```
 1    000000 20                RSBX CMPT
"file.asm", REMARK  at line 1:[R5002] Ignoring RSBX CMPT instruction
 2
 3    000001 4804              RETF
"file.asm", REMARK  at line 3:[R5004] Translation of RETF correct
only for non-interrupt routine
```

.noremark を使ったリスト・ファイル：

```
 1                                  .noremark 5002
 2  00000020                        RSBX CMPT
 3
 4  0000014804                      RETF
"file.asm", REMARK  at line 4:[R5004] Translation of RETF correct
only for non-interrupt routine
```

.option　リスト作成オプションの選択

構文

.option *option list*

説明

.option 疑似命令では、アセンブラ出力リスト用にいくつかのオプションを選択します。*Option list* には、縦線で区切ったオプションのリストを使用します。各オプションは、リスト作成機能を選択するために使用します。有効なオプションには次のものがあります。

B	.byte 疑似命令によるリスト出力を 1 行に制限します。
L	.long 疑似命令によるリスト出力を 1 行に制限します。
M	リストでのマクロ展開を行わないようにします。
R	B、M、T および W オプションをリセットします。
T	.string 疑似命令によるリスト出力を 1 行に制限します。
W	.word 疑似命令によるリスト出力を 1 行に制限します。
X	シンボルのクロスリファレンス・リストを作成します（アセンブラを起動するときに -x オプションを指定して、クロスリファレンス・リストを取得することもできます)。

このオプションでは、大文字小文字の区別は行われません。

例

この例では、.byte、.word、.long、および .string 疑似命令によるリスト出力をそれぞれ 1 行に制限する方法を示しています。

```
 1            ****************************************
 2            ** Limit the listing of .byte, .word, **
 3            ** .long, and .string directives      **
 4            **         to 1 line each.            **
 5            ****************************************
 6                    .option B, W, L, T
 7 000000             .data
 8 000000 00BD        .byte   -'C', 0B0h, 5
 9 000004 AABB        .long   0AABBCCDDh, 536 + 'A'
10 000008 15AA        .word   5546, 78h
11 00000a 0045        .string "Extended Registers"
12
13            ****************************************
14            **     Reset the listing options.     **
15            ****************************************
16                    .option R
17 00001c FFBD        .byte   -'C', 0B0h, 5
   00001d 00B0
   00001e 0005
18 000020 AABB        .long   0AABBCCDDh, 536 + 'A'
   000021 CCDD
   000022 0000
   000023 0259
19 000024 15AA        .word   5546, 78h
   000025 0078
20 000026 0045        .string "Extended Registers"
   000027 0078
   000028 0074
   000029 0065
   00002a 006E
   00002b 0064
   00002c 0065
   00002d 0064
   00002e 0020
   00002f 0052
   000030 0065
   000031 0067
   000032 0069
   000033 0073
   000034 0074
   000035 0065
   000036 0072
   000037 0073
```

.page　リストのページ替え

構文　　**.page**

説明　　**.page** 疑似命令により、リスト・ファイルのページ替えを行うことができます。.page 疑似命令はソース・リストには出力されませんが、アセンブラは、この疑似命令を検出すると行カウンタをインクリメントします。.page 疑似命令を使用してソース・リストをいくつかの論理部に分割すると、プログラムが読みやすくなります。

例　この例は、.page　疑似命令によって、アセンブラがどのようにソース・リストの新しいページを開始するかを示しています。

ソース・ファイル :

```
        .title   "**** Page Directive Example ****"
;         .
;         .
;         .
        .page
```

リスト・ファイル :

```
TMS320C55x COFF Assembler       Version x.xx
Copyright (c) 2001    Texas Instruments Incorporated
**** Page Directive Example ****                          PAGE   1
      2            ;        .
      3            ;        .
      4            ;        .
TMS320C55x COFF Assembler       Version x.xx
Copyright (c) 2001    Texas Instruments Incorporated
**** Page Directive Example ****                          PAGE   2
```

.port_for_speed/ .port_for_size　スピードまたはサイズに関する C54x 命令のエンコード

構文

.port_for_speed
.port_for_size

説明　**.port_for_speed** および **.port_for_size** の疑似命令は、C55x への移植時にアセンブラが特定の C54x 命令をエンコードする方法に影響を与えます。デフォルトでは、masm55 はコード・サイズを小さくするために C54x 命令をエンコードしようとします（.port_for_size）。アセンブラが高速エンコードを生成するには、.port_for_speed または -mh アセンブラ・オプションを使います。詳細は、7.2.2 項「速度優先の移植（-mh オプション）」（7-6 ページ）を参照してください。

.port_for_size 疑似命令は、アセンブラのデフォルト・エンコードをモデル化します。

.port_for_speed 疑似命令は、-mh アセンブラ・オプションの働きをモデル化します。コマンド行オプションと疑似命令の間でコンフリクトが起こった場合、疑似命令が優先します。

重要なループの直前には .port_for_speed の使用を考えてみましょう。ループの後で、デフォルトのエンコードに戻るためには .port_for_size を使います。

.sblock　初期化されたセクションのブロック化の指定

構文

.sblock["]*section name*["][**,** "*section name*", . . .]

説明　**.sblock** 疑似命令は、ブロック化するセクションを指定します。ブロック化は、ページ位置合わせに似た（ただしそれより制約の弱い）機能をもつアドレス位置合わせメカニズムです。ブロック化されたコード・セクションは、128 バイトより小さければ 128 バイト

境界を越えないように設定されます。128 バイトより大きいと、128 バイト境界から始まります。ブロック化されたセクションは、1 ページより小さい場合は 128 ワード（ページ）を越えないように設定されます。1 ページより大きい場合は、ページ境界から始まります。この疑似命令は、初期化されたセクションについてのみブロック化を指定できます。.usect または .bss 疑似命令で宣言された初期化されないセクションは指定できません。*section names* は、引用符で囲んでも囲まなくても構いません。

例

この例では、.text セクションと .data セクションがブロック化用に指定されています。

```
1 ****************************************
2 ** Specify blocking for the .text     **
3 ** and .data sections.                **
4 ****************************************
5         .sblock       .text, .data
```

.sect 名前付きセクションへのアセンブル

構文

.sect "*section name***"**

説明

.sect 疑似命令は、デフォルトの .text セクションや .data セクションのように使用される名前付きセクションを定義します。.sect 疑似命令は、名前付きセクションへのソース・コードのアセンブルを開始します。

section name は、アセンブラがコードをアセンブルするセクションを特定します。名前は二重引用符で囲まなければなりません。セクション名は、*section name*:*subsection name* の書式でサブセクション名を持つことができます。

COFF セクションについての詳細は、第 2 章「共通オブジェクト・ファイル・フォーマットの概要」を参照してください。

例

この例は、専用セクション Vars を定義し、これらのセクションにコードをアセンブルする方法を示しています。

```
 1              ********************************************
 2              **   Begin assembling into .text section.   **
 3              ********************************************
 4 000000                    .text
 5 000000 7600               MOV #120,AC0 ; Assembled into .text
   000002 7808
 6 000004 7B00               ADD #54,AC0 ; Assembled into .text
   000006 3600
 7              ********************************************
 8              **   Begin assembling into Vars section.    **
 9              ********************************************
10 000000                    .sect   "Vars"
11               WORD_LEN    .set    16
12               DWORD_LEN   .set    WORD_LEN * 2
13               BYTE_LEN    .set    WORD_LEN / 2
14 000000 000E               .byte   14
15              ********************************************
16              **   Resume assembling into .text section.  **
17              ********************************************
18 000008                    .text
19 000008 7B00               ADD #66,AC0 ; Assembled into .text
   00000a 4200
20              ********************************************
21              **   Resume assembling into Vars section.   **
22              ********************************************
23 000001                    .sect   "Vars"
24 000001 000D               .field  13, WORD_LEN
25 000002 0A00               .field  0Ah, BYTE_LEN
26 000003 0000               .field  10q, DWORD_LEN
   000004 0008
27
```

.set/.equ アセンブル時定数の定義

構文

symbol.**set***value*
symbol.**equ***value*

説明

.set および **.equ** の疑似命令は、値をシンボルと等価にします。このオプションで定義したシンボルは、アセンブリ・ソースの中で値の代わりに使用できます。この方法で、意味のある名前を定数や他の値と等価にすることができます。

- *symbol* は、ラベル・フィールドに表示されるラベルです。
- *value* は、整合定義式でなくてはなりません。つまり、式の中のすべてのシンボルは、前もって現行のソース・モジュール内で定義されていなくてはなりません。

未定義の外部シンボル、またはモジュールで、後で定義されるシンボルを式で使用することはできません。式が再配置可能な場合は、その式に割り当てられるシンボルも再配置可能となります。

式の値は、リストのオブジェクト・フィールドに示されます。この値は実際のオブジェクト・コードの一部ではなく、出力ファイルには書き込まれません。

.set または .equ を使用して定義されたシンボルは、.def または .global 疑似命令を使用して外部から参照できます。この方法で、グローバルな絶対定数を定義できます。

例 この例は、.set と .equ を使用したシンボルの割り当て方法を示しています。

```
 1              ********************************************
 2              **   Set symbol index to an integer expr.   **
 3              **    and use it as an immediate operand.   **
 4              ********************************************
 5               INDEX   .equ    100/2 +3
 6 000000 7B00          ADD #INDEX,AC0,AC0
   000002 3500
 7
 8              ********************************************
 9              ** Set symbol SYMTAB to a relocatable expr. **
10              **   and use it as a relocatable operand.   **
11              ********************************************
12 000000                .data
13 000000 000A  LABEL   .word   10
14               SYMTAB  .set    LABEL + 1
15
16              ********************************************
17              **   Set symbol NSYMS equal to the symbol   **
18              **   INDEX and use it as you would INDEX.   **
19              ********************************************
20                NSYMS   .set    INDEX
21 000001 0035              .word   NSYMS
```

.space 空間の確保

構文 **.space** *size in bits*

説明 **.space** 疑似命令は、現行のセクションに *size in bits* に指定された数のビットを確保し、それを 0 で埋めます。

> **注 : この疑似命令はデータ・セクションで使用してください。**
>
> コード・セクションとデータ・セクションのアドレス指定のしかたは異なるので、C55x 命令が組み込まれているセクション内で .space 疑似命令を使用すると、実行時におけるデータへの不正アクセスにつながります。ですから、これらの疑似命令は、データ・セクション内でのみ使用されるようにすることをお勧めします。

.space 疑似命令でラベルを使用すると、そのラベルはデータ・セクションで確保された最初のワードを指します。

例　この例は、.space 疑似命令を使用してメモリを確保する方法を示しています。

```
 1                ******************************************
 2                **  Begin assembling into .data section.   **
 3                ******************************************
 4 000000                    .data
 5 000000 0049               .string "In .data"
   000001 006E
   000002 0020
   000003 002E
   000004 0064
   000005 0061
   000006 0074
   000007 0061
 6                ******************************************
 7                **  Reserve 100 bits in the .data section; **
 8                **   RES_1 points to the first word that   **
 9                **         contains reserved bits.         **
10                ******************************************
11 000008          RES_1:.space  100
12 00000f 000F             .word   15
13 000010 0008"            .word   RES_1
14
```

.sslist/.ssnolist 置換シンボルのリスト作成の制御

構文

```
.sslist
.ssnolist
```

説明　この 2 つの疑似命令を使用すると、リスト・ファイル内の置換シンボルの展開の詳細を含めることを制御できます。

.sslist 疑似命令は、リスト・ファイルでの置換シンボルの展開の詳細を提供します。展開された行は、実際のソース行の下に示されます。

.ssnolist 疑似命令は、リスト・ファイルでの置換シンボル展開の詳細を抑止します。

デフォルトでは、リスト・ファイルでのすべての置換シンボルの展開は抑止されます。接頭部に（#）の付いた行は、展開された置換シンボルの詳細を示しています。

例

この例は、デフォルトで置換シンボルの展開のリスト作成を禁止するコードを示しています。また .sslist 疑似命令をアセンブルし、アセンブラに対して置換シンボル・コードの展開の詳細をリストに出力するように指示しています。

(a) ニーモニックの例

```
      1   000000              .bss    ADDRX, 1
      2   000001              .bss    ADDRY, 1
      3   000002              .bss    ADDRA, 1
      4   000003              .bss    ADDRB, 1
      5             ADD2      .macro  ADDRA, ADDRB
      6                       MOV ADDRA,AC0
      7                       ADD ADDRB,AC0,AC0
      8                       MOV AC0,ADDRB
      9                       .endm
     10
     11   000000C083          MOV AC0,*AR4+
     12   000002              ADD2    ADDRX, ADDRY
1          000002A000-        MOV ADDRX,AC0
1          000004D600         ADD ADDRY,AC0,AC0
           00000600-
1          000007C000-        MOV AC0,ADDRY
     13
     14                       .sslist
     15
     16   000009C083          MOV AC0,*AR4+
     17   00000bC003          MOV AC0,*AR0+
     18
     19   00000d              ADD2    ADDRX, ADDRY
1          00000dA000-        MOV ADDRA,AC0
#                             MOV ADDRX,AC0
1          00000fD600         ADD ADDRB,AC0,AC0
#                             ADD ADDRY,AC0,AC0
           00001100-
1          000012C000-        MOV AC0,ADDRB
#                             MOV AC0,ADDRY
```

(b) 代数表記の例

```
      1  000000              .bss    ADDRX, 1
      2  000001              .bss    ADDRY, 1
      3  000002              .bss    ADDRA, 1
      4  000003              .bss    ADDRB, 1
      5            ADD2      .macro  ADDRA, ADDRB
      6                      AC0 = @(ADDRA)
      7                      AC0 = AC0 + @(ADDRB)
      8                      @(ADDRB) = AC0
      9                      .endm
     10
     11  000000C083          *AR4+ = AC0
     12  000002              ADD2    ADDRX, ADDRY
1         000002A000-        AC0 = @(ADDRX)
1         000004D600         AC0 = AC0 + @(ADDRY)
          00000600-
1         000007C000-        @(ADDRY) = AC0
     13
     14                      .sslist
     15
     16  000009C083          *AR4+ = AC0
     17  00000bC003          *AR0+ = AC0
     18
     19  00000d              ADD2    ADDRX, ADDRY
1         00000dA000-        AC0 = @(ADDRA)
#                            AC0 = @(ADDRX)
1         00000fD600         AC0 = AC0 + @(ADDRB)
#                            AC0 = AC0 + @(ADDRY)
          00001100-
1         000012C000-        @(ADDRB) = AC0
#                            @(ADDRY) = AC0
```

.sst_off/.set_on SST モードの指定

構文

.sst_off
.sst_on

説明

.sst_off および **.sst_on** の疑似命令は、C55x への移植時にアセンブラが特定の C54x 命令をエンコードする方法に影響を与えます。デフォルトでは、masm55 は SST ビット（格納時は飽和）を有効とします（.sst_on）。このビットが実際に有効であるかどうかに関わらず、アセンブラの生成するデフォルトのエンコードは機能します。しかし、ユーザーのコードが SST ビットを有効にしない場合、アセンブラがより有効なエンコードを生成できるようにするには、.sst_off または -mt アセンブラ・オプションを使うことができます。詳細は、7.2.1 項「SST の無効化（-mt オプション）」（7-5 ページ）を参照してください。

.sst_on 疑似命令は、1 に設定された SST ステータス・ビットをモデル化します。これはアセンブラのデフォルトの設定です。.sst_off 疑似命令は、0 に設定された SST ステータス・ビットをモデル化します。これは、-mt アセンブラ・オプションを使用することと同等です。コマンド行オプションと疑似命令の間でコンフリクトが起こった場合、疑似命令が優先します。

.sst_on および .sst_off 疑似命令の有効範囲は静的であり、アセンブリ・プログラムの制御フローには影響されません。.sst_off 疑似命令と .sst_on 疑似命令との間にあるすべてのアセンブリ・コードはすべて、SST が無効であるという設定でアセンブルされます。

.string/.pstring テキストの初期化

構文

.string"*string1*"[, ...,"*stringn*"]
.pstring"*string1*"[, ..., "*stringn*"]

説明

.string および **.pstring** 疑似命令は、文字列からの 8 ビット文字を現行のセクションに格納します。.string 疑似命令は、8 ビット文字を現行のデータ・セクション内の連続したワードに格納します。.pstring 疑似命令は、8 ビット領域を初期化しますが、各文字列の内容はワードごと 2 文字にパックします。各 *string* は以下のいずれかです。

- ❑ アセンブラが計算し、8 ビットまたは 16 ビットの符号付き数値として取り扱う式。
- ❑ 二重引用符に囲まれた文字列。文字列内の各文字は、別々のバイトを表します。

注 : これらの疑似命令はデータ・セクションで使用してください。

コード・セクションとデータ・セクションのアドレス指定のしかたは異なるので、C55x 命令が組み込まれているセクション内で .string および .pstring 疑似命令を使用すると、実行時におけるデータへの不正アクセスにつながります。ですから、これらの疑似命令は、データ・セクション内でのみ使用されるようにすることをお勧めします。

.pstring では、値がワードにパックされる際に、各ワードの最上位のバイトから埋められていきます。未使用の空間にはヌル・バイトが埋め込まれます。

オペランドを 8 ビット定数として指定できますが、アセンブラは 8 ビットより大きな値はすべて切り捨てます。

ラベルを使用する場合、ラベルは、（データ・セクション内の）初期化される最初のワードを指します。

文字列を .struct/.endstruct のシーケンスで使用すると、.string はメンバのサイズを定義し、メモリは初期化しないことに注意してください。.struct/.endstruct の詳細は、4.9 節「アセンブル時シンボル疑似命令」（4-22 ページ）を参照してください。

例　この例では、現行のセクションのワードに 8 ビット値が格納されています。

```
  1 000000                    .data
  2 000000 0041               .string  41h, 42h, 43h, 44h
    000001 0042
    000002 0043
    000003 0044
  3 000004 0041  Str_Ptr:.string  "ABCD"
    000005 0042
    000006 0043
    000007 0044
  4 000008 4175                 .pstring "Austin", "Houston"
    000009 7374
    00000a 696E
    00000b 486F
    00000c 7573
    00000d 746F
    00000e 6E00
  5 00000f 0030               .string  36 + 12
```

.struct/ .endstruct/.tag　構造体型の宣言

構文

[*stag*]	**.struct**	[*expr*]
[*mem0*]	*element*	[*expr0*]
[*mem1*]	*element*	[*expr1*]
.	.	.
.	.	.
.	.	.
[*memn*]	**.tag** *stag*	[, *exprn*]
.	.	.
.	.	.
.	.	.
[*memN*]	*element*	[*exprN*]
[*size*]	**.endstruct**	
label	**.tag**	*stag*

説明　**.struct** 疑似命令は、シンボリック・オフセットをデータの構造体定義の要素に割り当てます。これにより、類似するデータ要素をグループ化してから、アセンブラに要素のオフセットを計算するように指示できます。この方式は、C 構造体または Pascal レコードに似ています。.struct 定義には .union 定義を含めることができます。また、.structs と .unions をネストすることもできます。.struct 疑似命令はメモリの割り当ては行いません。反復使用が可能なシンボリックなテンプレートを作成するだけです。

.endstruct 疑似命令は、構造体定義の終わりにマークを付けます。

.tag 疑似命令は、構造体の特性をラベルに付与します。これによりシンボル表記を簡略化し、他の構造体を含む構造体を定義できるようにします。.tag 疑似命令はメモリの割り当ては行いません。.tag 疑似命令の構造体タグ（stag）は、事前に定義しておかなければなりません。

stag 構造体のタグです。値は、その構造体の始まりを示します。stag が指定されていない場合、アセンブラは構造体のメンバを、構造体の最上部からの絶対オフセットを値とするグローバル・シンボル・テーブルに入れます。stag は、.struct の場合は任意ですが、.tag の場合は必須です。

expr 構造体の始まりのオフセットを示す任意の式です。構造体は、デフォルトでは 0 から開始するように設定されます。このパラメータはトップレベルの構造体でのみ使用できます。ネストされた構造体を定義しているときには使用できません。*expr* は、.struct の上部と .struct の最初のメンバとの間に埋め込みを指定します。

mem_n 構造体のメンバを示す任意のラベルです。このラベルは絶対的で、構造体の始まりからのオフセットと等価です。メンバ構造体に対するラベルはグローバル宣言ができません。

element 次の疑似命令のどれか 1 つを指定します。.byte、.char、.double、field、.float、.half、.int、.long、.short、.string、.ubyte、.uchar、.uhalt、.uint、.ulong、.ushort、.uword 、および .word。また、element（要素）は、ネストされた構造体か共用体の完全な宣言、またはタグで宣言された構造体か共用体でも構いません。これらの疑似命令は、.struct 疑似命令の後に使用すると要素のサイズを表します。メモリの割り当ては行いません。

$expr_n$ 記述される要素数を指定する任意の式です。この値はデフォルトで 1 となります。.string 要素のサイズは 1 ワード、.field 要素は 1 ビットと見なされます。

size 構造体全体のサイズを示す、任意のラベルです。

注 : .struct /.endstruct シーケンス内で使用できる疑似命令

.struct/.endstruct シーケンス内で使用できる疑似命令は、要素記述子、構造体および共用体のタグ、条件付きアセンブリ疑似命令、およびメンバ・オフセットをワード境界で位置合わせする .align 疑似命令だけです。空の構造体は正しくありません。

次の例では、.struct、.tag、および .endstruct の疑似命令のさまざまな使用方法を示しています。

例 1

```
 1 000000                    .data
 2                REAL_REC   .struct                  ; stag
 3         0000   NOM        .int                     ; member1 = 0
 4         0001   DEN        .int                     ; member2 = 1
 5         0002   REAL_LEN   .endstruct               ; real_len = 2
 6 000000                    .text
 7 000000 D600               ADD @(REAL + REAL_REC.DEN),AC0,AC0
   000002 00-
 8                                                  ; access structure element
 9
10 000000                    .bss REAL, REAL_LEN      ; allocate mem rec
```

例 2

```
11                          .data
12              CPLX_REC    .struct
13       0000   REALI       .tag REAL_REC            ; stag
14       0002   IMAGI       .tag REAL_REC            ; member1 = 0
15       0004   CPLX_LEN    .endstruct               ; cplx_len = 4
16
17              COMPLEX     .tag CPLX_REC            ; assign structure attrib
18
19 000002                   .bss COMPLEX, CPLX_LEN
20 000003                   .text
21 000003 D600              ADD @(COMPLEX.REALI),AC0,AC0     ; access structure
   000005 00-
22 000006 C000-             MOV AC0,@(COMPLEX.REALI)
23
24 000008 D600              ADD @(COMPLEX.IMAGI),AC1,AC1     ; allocate space
   00000a 11-
```

例 3

```
 1 000000                    .data
 2                           .struct                  ; no stag puts mems into
 3                                                    ; global symbol table
 4        0000   X           .int                     ; create 3 dim templates
 5        0001   Y           .int
 6        0002   Z           .int
 7        0003               .endstruct
```

例 4

```
1  000000                .data
1              BIT_REC   .struct                  ; stag
2         0000 STREAM    .string 64
3         0040 BIT7      .field  7                ; bits1 = 64
4         0040 BIT9      .field  9                ; bits2 = 64
5         0041 BIT10     .field  10               ; bits3 = 65
6         0042 X_INT     .int                     ; x_int = 66
7         0043 BIT_LEN   .endstruct               ; length = 67
8
9              BITS      .tag BIT_REC
10 000000                .text
11 000000 D600           ADD @(BITS.BIT7),AC0,AC0   ; move into acc
   000002 00%
12 000003 187F           AND #127,AC0               ; mask off garbage bits
   000005 00
13
14 000000                .bss BITS, BIT_REC
```

.tab　タブ・サイズの定義

構文　　**.tab** *size*

説明　　**.tab** 疑似命令は、タブサイズを定義します。ソース入力で検出されたタブは、リスト内で *size* に指定された数の空白文字に変換されます。デフォルトのタブのサイズは、空白文字 8 個分です。

例　　次のそれぞれの行は、タブ文字 1 つと、それに続く NOP 命令から構成されています。

ソース・ファイル :

```
; default tab size
     NOP
     NOP
     NOP

       .tab 4
     NOP
     NOP
     NOP

       .tab 16
     NOP
     NOP
     NOP
```

リスト・ファイル：

```
 1            ; default tab size
 2 000000 20          NOP
 3 000001 20          NOP
 4 000002 20          NOP
 5
 7 000003 20      NOP
 8 000004 20      NOP
 9 000005 20      NOP
10
12 000006 20                  NOP
13 000007 20                  NOP
14 000008 20                  NOP
```

.text　.text セクションへのアセンブル

構文　　**.text**

説明

.text 疑似命令は、アセンブラに対して .text セクションへのアセンブルを開始するように指示します。アセンブリは、*.text* セクションに実行可能コードが含まれているものとみなします。.text セクションへのアセンブルが行われていない場合、セクション・プログラム・カウンタは 0 に設定されます。すでにコードが .text セクションにアセンブルされている場合、セクション・プログラム・カウンタはそのセクションの前の値からカウントを再開します。

.text セクションはコード・セクションなので、バイト・アドレス可能です。データ・セクションはワード・アドレス可能です。

.text はデフォルトのセクションです。したがって、異なるセクション疑似命令（.data または .sect）を指定しなければ、アセンブル開始時にアセンブラは .text セクションにコードをアセンブルします。

COFF セクションについての詳細は、第 2 章「共通オブジェクト・ファイル・フォーマットの概要」を参照してください。

例

この例では、コードを .text セクションと .data セクションにアセンブルします。.data セクションには整数定数が書き込まれ、.text セクションには実行可能コードが書き込まれます。

```
 1            ****************************************
 2            ** Begin assembling into .data section.**
 3            ****************************************
 4 000000               .data
 5 000000 0041 START:   .string "A","B","C"
   000001 0042
   000002 0043
 6 000003 0058 END:     .string "X","Y","Z"
   000004 0059
   000005 005a
 7            ****************************************
 8            ** Begin assembling into .text section. **
 9            ****************************************
10 000000               .text
11 000000 D600          ADD START,AC0,AC0
   000002 00"
12 000003 D600          ADD END,AC0,AC0
   000005 00‟
13            ****************************************
14            **  Resume assembling into .data section.**
15            ****************************************
16 000006               .data
17 000006 000a          .byte   0Ah, 0Bh
   000007 000b
18 000008 000c          .byte   0Ch, 0Dh
   000009 000d
19            ****************************************
20            **  Resume assembling into .text section.**
21            ****************************************
22 000006               .text
23 000006 2201          MOV AC0,AC1
```

.title　ページ・タイトルの定義

構文

.title "*string***"**

説明

.title 疑似命令は、各リスト・ページのヘッダ部分に出力されるタイトルを提供します。.title 文自体は出力されませんが、行カウンタはインクリメントされます。

string には、引用符で囲まれたタイトルを 65 文字まで指定できます。65 文字を越えると、アセンブラは文字列を切り捨て、警告を発行します。

アセンブラは、.title 疑似命令に続くページにタイトルを出力します。そして別の .title 疑似命令が処理されるまで、以降のページに出力し続けます。リストの最初のページにタイトルを出力する場合は、最初のソース文に .title 疑似命令を指定しなければなりません。

例

この例では、あるタイトルは 1 ページ目に出力され、別のタイトルは後続ページに出力されます。

ソース・ファイル :

```
          .title  "**** Fast Fourier Transforms ****"
;         .
;         .
;         .
          .title  "**** Floating-Point Routines ****"
          .page
```

リスト・ファイル :

```
COFF Assembler       Version x.xx
Copyright (c) 2001      Texas Instruments Incorporated
**** Fast Fourier Transforms ****                          PAGE   1

      2             ;         .
      3             ;         .
      4             ;         .
COFF Assembler       Version x.xx
Copyright (c) 2001      Texas Instruments Incorporated
**** Floating-Point Routines ****                          PAGE   2
```

.union/ .endunion/.tag 共用体型の宣言

構文

[*utag*]	**.union**	[*expr*]
[*mem0*]	*element*	[*expr0*]
[*mem1*]	*element*	[*expr1*]
.	.	.
.	.	.
.	.	.
[*memn*]	**.tag**	*utagn*[, *exprn*]
.	.	.
.	.	.
.	.	.
[*memN*]	*element*	[*exprN*]
[*size*]	**.endunion**	
label	**.tag**	*utag*

説明

.union 疑似命令は、シンボリックなオフセットを、同じメモリ空間に割り当てられる代替データ構造体定義の要素に割り当てます。この疑似命令を使用すると、いくつかの代替構造体を定義して、アセンブラに要素オフセットを計算させることができます。この機能は C の共用体に似ています。.union 疑似命令は、メモリの割り当ては行いません。反復使用が可能なシンボリックなテンプレートを作成するだけです。

.struct 定義には .union 定義を含めることができます。また、.structs と .unions をネストすることもできます。

.endunion 疑似命令は、共用体定義の終わりにマークを付けます。

.tag 疑似命令は、構造体または共用体の特性を *label* に付与します。これによりシンボル表記を簡略化し、他の構造体または共用体を含む構造体または共用体を定義できるようにします。.tag 疑似命令はメモリの割り当ては行いません。.tag 疑似命令の構造体または共用体のタグは、事前に定義しておく必要があります。

utag 共用体のタグです。値は、その共用体の始まりを示します。utag が指定されていない場合、アセンブラは共用体のメンバを、共用体の最上部からの絶対オフセットを値とするグローバル・シンボル・テーブルに入れます。この場合、各メンバには固有の名前が付いていなくてはなりません。

expr 共用体の始まりのオフセットを示す任意の式です。共用体は、デフォルトでは 0 から開始するように設定されます。このパラメータはトップレベルの共用体でのみ使用できます。ネストされた構造体を定義しているときには使用できません。

mem_n 共用体のメンバを示す任意のラベルです。このラベルは絶対的で、共用体の始まりからのオフセットと等価です。共用体のメンバに対するラベルはグローバル宣言できません。

element 次の疑似命令のどれか 1 つを指定します。.byte、.char、.double、field、.float、.half、.int、.long、.short、.string、.ubyte、.uchar、.uhalt、.uint、.ulong、.ushort、.uword 、および .word。また、element（要素）は、ネストされた構造体か共用体の完全な宣言、またはタグで宣言された構造体か共用体でも構いません。これらの疑似命令は、.union 疑似命令の後に使用すると、要素のサイズを表します。メモリの割り当ては行いません。

$expr_n$ 記述される要素数を指定する任意の式です。この値はデフォルトで 1 となります。.string 要素のサイズは 1 ワード、.field 要素は 1 ビットと見なされます。

size 共用体全体のサイズを示す、任意のラベルです。

注 : .union /.endunion シーケンス内で使用できる疑似命令

.union/.endunion シーケンス内で使用できる疑似命令は、要素記述子、構造体および共用体のタグ、および条件付きアセンブリ疑似命令だけです。空の共用体定義は正しくありません。

以下の例では、タグがある共用体とタグがない共用体を示しています。

例 1

```
1                       .global employid
2  000000               .data
3              xample   .union                    ; utag
4         0000 ival     .word                     ; member1 = 0
5         0000 fval     .float                    ; member2 = 0
6         0000 sval     .string                   ; member3 = 0
7         0002 real_len .endunion                 ; real_len = 4
8
9  000000               .bss  employid, real_len  ;allocate memory
10
11             employid .tag  xample
12 000000               .text
13 000000 D600          ADD @(employid.fval),ADD,ADD ; access union element
   000002 00-
```

例 2

```
1  000000               .data
2                       .union                    ; utag
3         0000 x        .long                     ; member1 = long
4         0000 y        .float                    ; member2 = float
5         0000 z        .word                     ; member3 = word
6         0002 size_u   .endunion                 ; real_len = 4
7
```

.usect　初期化されない空間の確保

構文

symbol **.usect "***section name***",** *size in words* **[,** [*blocking flag*] [, *alignment*]]

説明

.usect は、初期化されていない名前付きセクションに変数のための空間を確保します。この疑似命令は .bss 疑似命令に似ています。両方ともデータ用の空間を確保するだけで、内容はありません。ただし .bss セクションとは無関係に、.usect はメモリ内の任意の場所に配置できる追加のセクションを定義します。

symbol　.usect 疑似命令の実行により確保される最初の位置を指します。symbol は、空間を確保する変数の名前に対応します。

section name　二重引用符で囲まなければなりません。このパラメータは、初期化されないセクションに名前を付けます。COFF0 および COFF1 フォーマットのファイルの場合は、最初の 8 文字のみが有効です。セクション名は、*section name*:*subsection name* の書式でサブセクション名を持つことができます。

size in words　*section name* で確保されたワード数を定義する式です。

blocking flag 任意のパラメータです。指定した値がゼロ以外の場合、このフラグはこのセクションがブロック化されることを表します。ブロック化は、位置合わせに似た（ただしそれより制約の弱い）機能をもつアドレス指定メカニズムです。セクションは、1 ページより小さい場合はページ境界（128 ワード）にまたがらないように設定され、1 ページより大きい場合はページ境界から始まるように設定されます。このブロック化はセクションに適用されるのであり、.usect 疑似命令のこのインスタンスで宣言したオブジェクトに適用されるものではありません。

alignment 任意のパラメータで、シンボルに割り当てられた空間を指定の境界で開始させます。この境界は、スロット・サイズをワードで示し、2 の累乗に設定できます。

注 : 位置合わせフラグのみを指定する

ブロック化フラグを指定しないで位置合わせフラグのみを指定するには、構文が示すように、位置合わせフラグの前にコンマを 2 つ挿入する必要があります。

その他のセクション疑似命令（.text、.data、および .sect）は、現行のセクションを終了して、アセンブラに別のセクションへのアセンブルを開始するように伝えます。ただし、.usect および .bss 疑似命令は現行のセクションに影響を与えません。アセンブラは、.usect および .bss 疑似命令をアセンブルしてから現行のセクションへのアセンブルを再開します。

メモリ内に隣接して配置される変数は、指定した同じセクション内に定義できます。これをするには、同じセクション名を使用して .usect 疑似命令を繰り返します。

COFF セクションについての詳細は、第 2 章「共通オブジェクト・ファイル・フォーマットの概要」を参照してください。

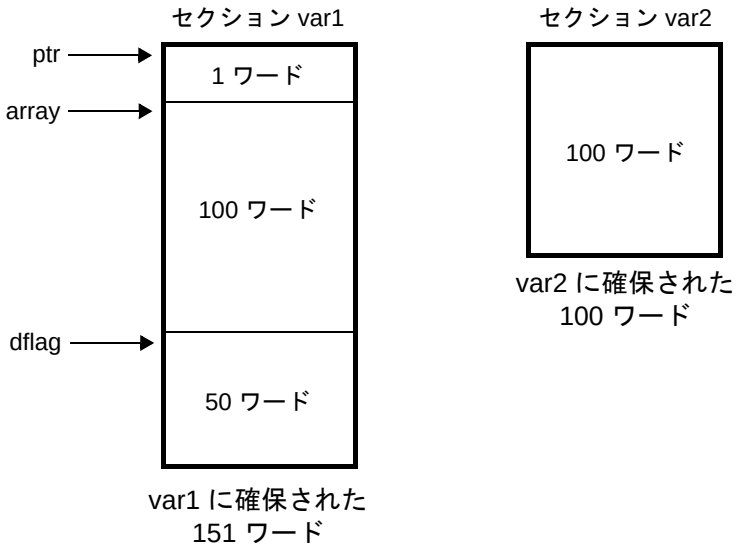
例

この例では、.usect 疑似命令を使用して、2 つの初期化されない名前付きセクション var1 と var2 を定義します。シンボル ptr は、var1 セクションに確保された最初のワードを指します。シンボル配列は、var1 に確保された 100 ワード・ブロック内の最初のワードを指します。また、dflag は var1 内の 50 ワード・ブロックの最初のワードを指します。シンボル vec は、var2 セクションに確保された最初のワードを指します。

図 4-4（4-100 ページ）では、この例で 2 つの初期化されないセクション var1 と var2 で空間を確保する方法を示します。

```
 1              ****************************************
 2              **     Assemble into .text section.     **
 3              ****************************************
 4 000000               .text
 5 000000 3C30          MOV #3,AC0
 6
 7              ****************************************
 8              **       Reserve 1 word in var1.        **
 9              ****************************************
10 000000     ptr      .usect  "var1", 1
11
12              ****************************************
13              **      Reserve 100 words in var1.      **
14              ****************************************
15 000001     array    .usect  "var1", 100
16
17 000002 7B00         ADD #55,AC0,AC0  ; Still in .text
   000004 3700
18
19              ****************************************
20              **      Reserve 50 words in var1.       **
21              ****************************************
22 000065     dflag    .usect  "var1", 50
23
24 000006 7B06         ADD #dflag,AC0,AC0 ; Still in .text
   000008 5000-
25
26              ****************************************
27              **      Reserve 100 words in var2.      **
28              ****************************************
29 000000     vec      .usect  "var2", 100
30
31 00000a 7B00         ADD #vec,AC0,AC0   ; Still in .text
   00000c 0000-
32              ****************************************
33              **  Declare an external .usect symbol.  **
34              ****************************************
35                     .global array
```

図 4-4.　　.usect 疑似命令



.var　置換シンボルをローカルな変数として使用

構文　　**.var**　*sym1 [,sym2, ..., symn]*

説明

.var　疑似命令を使用すると、マクロ内で置換シンボルをローカル変数として使用できます。この疑似命令を使用すると、パラメータを含めて、1 つのマクロにつき 32 個までのローカル・マクロ置換シンボルを定義できます。

.var 疑似命令は、ヌル文字列の初期値をもつ一時的な置換シンボルを作成します。このシンボルはパラメータとして渡されることはなく、展開後は消滅します。

マクロの詳細は、第 5 章「マクロ言語」を参照してください。

.vli_off/.vli_on 可変長命令のサイズ解決の抑止

構文

.vli_off
.vli_on

説明

.vli_off 疑似命令と **.vli_on** 疑似命令は、アセンブラによる可変長命令の処理のしかたに影響します。.vli_off 疑似命令は、-mv コマンド行オプションを使用することと同じです。コマンド行オプションと疑似命令の間でコンフリクトが起こった場合、疑似命令が優先します。

デフォルト（.vli_on）では、アセンブラは、スタンドアロンの可変長命令を、可能な限り最小のサイズまで解決しようとします。

以下の命令グループでサイズ解決が行われます。

```
goto L7, L16, P24
if (cond) goto l4
if (cond) goto L8, L16, P24
call L16, P24
if (cond) call L16, P24
```

場合によっては、ある命令の最大書式（P24）を使用したいことがあります。特定の可変長命令の P24 バージョンは、同じ命令のより小さいバージョンよりも少ないサイクルで実行されます。次の命令の最大書式を保持するには、.vli_off 疑似命令を使用します。

```
goto P24
call P24
```

.vli_off 疑似命令と .vli_on 疑似命令を使用して、アセンブリ・ファイルの領域に対するこの動作を切り替えることができます。.vli_off 疑似命令を使用しても、他のすべての可変長命令は、アセンブラによって最小可能サイズに分解されます。

.vli_off および .vli_on 疑似命令の有効範囲は静的であり、アセンブリ・プログラムの制御フローには影響されません。

.warn_off/ .warn_on 警告メッセージの抑止

構文

```
.warn_on
.warn_off
```

説明

.warn_off および **.warn_on** 疑似命令は、アセンブラ警告メッセージのレポートを制御します。デフォルト（.warn_on）では、アセンブラは警告メッセージを生成します。.warn_off 疑似命令は、アセンブラ警告メッセージを抑制します。この疑似命令は、-mw コマンド行オプションを使用することと同じです。コマンド行オプションと疑似命令の間でコンフリクトが起こった場合、疑似命令が優先します。

.warn_off 疑似命令と .warn_on 疑似命令を使用して、アセンブリ・ファイルの領域に対するこの動作を切り替えることができます。

.warn_off および .warn_on 疑似命令の有効範囲は静的であり、アセンブリ・プログラムの制御フローには影響されません。ファイル内の .warn_off 疑似命令と .warn_on 疑似命令との間にあるアセンブリ・コードについては、警告メッセージはレポートされません。

第 5 章

マクロ言語

アセンブラでは、ユーザー独自の「命令」を作ることができるマクロ言語をサポートしています。特に、プログラムに何度も同じような作業を実行させる場合にこの機能を使用すると便利です。マクロ言語を使用すると次のことができます。

- ❑ 独自のマクロを定義し、既存のマクロを再定義する。
- ❑ 長い、または複雑なアセンブリ・コードを単純化する。
- ❑ アーカイバを使って作成したマクロ・ライブラリにアクセスする。
- ❑ マクロ内に、条件付きブロックおよび繰り返しブロックを定義する。
- ❑ マクロ内の文字列を操作する。
- ❑ 展開リスト出力を制御する。

項目 **ページ**

5.1 マクロの使用方法

プログラムには、何回か実行されるルーチンが含まれていることがしばしばあります。このような場合、ルーチンのソース文を繰り返す代わりにルーチンをマクロとして定義し、通常はルーチンを繰り返す場所でそのマクロを呼び出すことができます。この方法により、短く簡潔なソース・プログラムを書くことができます。

同じマクロを何度か呼び出す場合、ただし呼び出しごとに別のデータを使用するときには、マクロ・パラメータを介してマクロに引数を渡すことができます。パラメータを割り当てることにより、マクロを呼び出すたびに異なる情報をマクロに渡すことができます。マクロ言語では、置換シンボルと呼ばれる特殊シンボルをサポートしています。マクロ・パラメータにこの置換シンボルを使用します。

マクロの使用には、次の 3 つの手順があります。

ステップ 1: **マクロの定義。**プログラムでマクロを使用するには、まずマクロを定義する必要があります。マクロを定義するには、次の 2 つの方法があります。

- ❏ マクロはソース・ファイルまたは .copy/.include ファイル内で定義できます。詳細は、5.2 節「マクロの定義」を参照してください。
- ❏ マクロはマクロ・ライブラリで定義することができます。マクロ・ライブラリとは、アーカイバを使用して作成したアーカイブ・フォーマットのファイルの集合のことです。アーカイブ・ファイル（マクロ・ライブラリ）の個々のメンバには、そのメンバ名に対応したマクロ定義が 1 つ入っています。マクロ・ライブラリにアクセスするには、.mlib 疑似命令を使用します。詳細は、5.4 節「マクロ・ライブラリ」（5-14 ページ）を参照してください。

ステップ 2: **マクロの呼び出し。**いったんマクロが定義されると、そのマクロ名は、ソース・プログラムの中でニーモニックとして使用し、マクロを起動できます。これを「マクロ呼び出し（マクロ・コール）」と言います。

ステップ 3: **マクロの展開。**ソース・プログラムがマクロを呼び出すと、アセンブラはそのマクロを展開します。展開時、アセンブラは変数によって引数をマクロ・パラメータに渡し、マクロ呼び出し文をマクロ定義に置き換え、ソース・コードをアセンブルします。デフォルトでは、マクロ展開はリスト・ファイルに出力されます。.mnolist 疑似命令を使用すると、展開リストの出力を止めることができます。詳細は、5.8 節「出力リストをフォーマットする方法」（5-21 ページ）を参照してください。

アセンブラはマクロ定義を見つけると、そのマクロ名を命令コード・テーブルに記録します。すでに定義されているマクロ、ライブラリ・エントリ、疑似命令、または命令ニーモニックは、見つかったマクロと同じ名前のものがあれば再定義されます。これにより、新しい命令を追加するだけでなく、既存の疑似命令および命令の関数を展開することができます。

5.2 マクロの定義

マクロはプログラム内のどこにでも定義できますが、使用する前に必ず定義する必要があります。マクロは、ソース・ファイルでも、.include/.copy ファイルでも、またはマクロ・ライブラリでも定義できます。マクロ・ライブラリの詳細は、5.4 節「マクロ・ライブラリ」（5-14 ページ）を参照してください。

マクロ定義はネストさせることも、別のマクロを呼び出すこともできます。ただし、マクロのすべての要素が同じファイルの中で定義されていなければなりません。ネストされたマクロの詳細は、5.9 節「再帰的なマクロとネストされたマクロの使用方法」（5-23 ページ）を参照してください。

マクロの定義は、次のようなフォーマットの一連のソース文です。

macname **.macro** [*parameter*$_1$] [**,** ... **,** *parameter*$_n$]
model statements or macro directives
[.mexit]
.endm

macname	マクロに名前を付けます。名前は、必ずソース文のラベル・フィールドに入れます。マクロ名の最初の 32 文字が有効です。アセンブラはマクロ名を内部命令コード・テーブルに記録し、すでに同じ名前をもった命令やマクロ定義があればそれを置き換えます。
.macro	ソース文がマクロ定義の第 1 行目であることを特定します。.macro は、必ずニーモニック・フィールドに入れます。
[parameters]	.macro 疑似命令のオペランドとして使用される、指定が任意の置換シンボルです。パラメータについては、5.3 節「マクロ・パラメータ／置換シンボル」（5-6 ページ）を参照してください。
model statements	マクロが呼び出されるたびに実行される命令、またはアセンブラ疑似命令です。
macro directives	マクロ展開の制御に使用されます。
.mexit	goto .endm 文として機能します。エラー・テストでマクロ展開が失敗すること、および残りのマクロが不要であることが分かった場合には、.mexit 疑似命令を使用すると便利です。
.endm	マクロ定義を終了します。

マクロ定義にコメントを含め、そのコメントをマクロ展開には表示したくない場合には、コメントの前に感嘆符を付けます。コメントをマクロ展開にも表示する場合には、アスタリスクかセミコロンを付けます。マクロのコメントの詳細は、5.7 節「マクロでのメッセージの作成方法」（5-19 ページ）を参照してください。

例 5–1 は、マクロの定義、呼び出し、および展開を示します。

例 5–1. マクロの定義、呼び出し、展開

(a) ニーモニックの例

```
       1              *
       2
       3              *        add3
       4              *
       5              *        ADDRP = P1 + P2 + P3
       6
       7              add3     .macro P1, P2, P3, ADDRP
       8
       9                       MOV P1,AC0
      10                       ADD P2,AC0,AC0
      11                       ADD P3,AC0,AC0
      12                       MOV AC0,ADDRP
      13                       .endm
      14
      15
      16                       .global abc, def, ghi, adr
      17
      18 000000                add3 abc, def, ghi, adr
1
1        000000 A000!          MOV abc,AC0
1        000002 D600           ADD def,AC0,AC0
         000004 00!
1        000005 D600           ADD ghi,AC0,AC0
         000007 00!
1        000008 C000!          MOV AC0,adr
```

例 5–1. マクロの定義、呼び出し、展開（続き）

（b）代数表記の例

```
      1              *
      2
      3              *         add3
      4              *
      5              *         ADDRP = P1 + P2 + P3
      6
      7              add3      .macro P1, P2, P3, ADDRP
      8
      9                        AC0 = @(P1)
     10                        AC0 = AC0 + @(P2)
     11                        AC0 = AC0 + @(P3)
     12                        @(ADDRP) = AC0
     13                        .endm
     14
     15
     16                        .global abc, def, ghi, adr
     17
     18 000000                 add3 abc, def, ghi, adr
1
1       000000 A000!           AC0 = @(abc)
1       000002 D600            AC0 = AC0 + @(def)
        000004 00!
1       000005 D600            AC0 = AC0 + @(ghi)
        000007 00!
1       000008 C000!           @(adr) = AC0
```

5.3 マクロ・パラメータ／置換シンボル

同じマクロを何度か呼び出す場合、ただし呼び出しごとに別のデータを使用するときは、そのマクロ内部でパラメータを割り当てることができます。マクロ言語では、置換シンボルと呼ばれる特殊シンボルをサポートしています。マクロ・パラメータにこの置換シンボルを使用します。

マクロ・パラメータは文字列を表す置換シンボルです。これらのシンボルは、文字列をシンボル名で代用するという用途で、マクロ以外の場所で使用することもできます。

有効な置換シンボルは長さが 32 文字までで、先頭を文字で始める必要があります。先頭の文字に続く部分には、英数字、下線、ドル記号を使用できます。

マクロ・パラメータとして使用される置換シンボルは、それが定義されているマクロ内でしか効果がありません。.var 疑似命令を使用して定義する置換シンボルを含めて、1 つのマクロにつき 32 個までのローカルな置換シンボルを定義できます。.var 疑似命令の詳細は、5.3.6 項「マクロでローカル変数として使われる置換シンボル」（5-13 ページ）を参照してください。

マクロ展開時に、アセンブラは引数をマクロ・パラメータに渡します。各引数の文字列は、対応するマクロ・パラメータに割り当てられます。対応する引数をもたないパラメータは、ヌル文字列に設定されます。引数の数がパラメータの数よりも多い場合は、最後のパラメータに残りのすべての引数の文字列が割り当てられます。

引数のリストを 1 つのパラメータに渡す場合、またはコンマやセミコロンをパラメータに渡す場合は、その語を引用符で囲みます。

アセンブル時に、アセンブラはまずマクロ定義内のマクロ・パラメータをそれに対応する文字列に置き換え、次にそのソース・コードをオブジェクト・コードに変換します。

例 5–2 は、引数の数が異なるマクロ展開の例を示しています。

例 5–2. 引数の数が異なるマクロ呼び出しの例

マクロ定義

```
Parms  .macro a,b,c
;         a = :a:
;         b = :b:
;         c = :c:
          .endm
```

マクロの呼び出し:

```
       Parms  100,label            Parms 100,label,x,y
;         a = 100                        ;   a = 100
;         b = label                      ;   b = label
;         c = " "                        ;   c = x,y

       Parms  100, , x             Parms "100,200,300",x,y
;         a = 100                        ;   a = 100,200,300
;         b = " "                        ;   b = x
;         c = x                          ;   c = y

       Parms  """string""",x,y
;         a = "string"
;         b = x
;         c = y
```

5.3.1 置換シンボルを定義するための疑似命令

置換シンボルは、**.asg** 疑似命令と **.eval** 疑似命令を使用して操作できます。

.asg 疑似命令は、置換シンボルに文字列を割り当てます。

.asg 疑似命令の構文は次のとおりです。

.asg ["]*character string*["], *substitution symbol*

引用符の使用は任意です。引用符がない場合、アセンブラは最初のコンマまでの文字を読み取り、その前後の空白を取り除きます。どちらの場合でも、文字列が読み取られて置換シンボルに割り当てられます。

例 5–3 は、置換シンボルに割り当てられた文字列を示しています。

例 5–3. .asg 疑似命令

```
   .asg  AR0,FP              ;   frame pointer
   .asg  *AR1+,Ind           ;   indirect addressing
   .asg  *AR1+0b,Rc_Prop     ;   reverse carry propagation
   .asg  """string""",strng  ;   string
   .asg  "a,b,c",parms       ;   parameters
```

.eval 疑似命令は、数値置換シンボルに対して算術を実行します。

.eval 疑似命令の構文は次のとおりです。

.eval *well-defined expression, substitution symbol*

.eval 疑似命令は、式を計算し、結果の文字列値を置換シンボルに割り当てます。式の定義が不完全な場合、アセンブラはエラーを発行し、ヌル文字列をシンボルに割り当てます。

例 5–4 は、置換シンボルについて実行する算術計算を示します。

例 5–4. .eval 疑似命令

```
.asg   1,counter
.loop  100
.word  counter
.eval  counter + 1,counter
.endloop
```

例 5–4 では、.asg 疑似命令を .eval 疑似命令と置き換えても、出力結果は変わりません。このような単純なケースでは、.eval と .asg のどちらを使用しても構いません。ただし、式から値を計算する場合は .eval 疑似命令を使用する必要があります。.asg 疑似命令は文字列を置換シンボルに割り当てるだけですが、.eval 疑似命令は式を計算し、置換シンボルに等価の文字列を割り当てます。

5.3.2 ビルトイン置換シンボル関数

以下のビルトイン置換シンボル関数を使用すると、置換シンボルの文字列値に基づいて判断を行うことができます。これらの関数は必ず値を戻し、式の中で使用することもできます。ビルトイン置換シンボル関数は、アセンブリの条件式に使用すると非常に便利です。これらの関数のパラメータは、置換シンボルか文字列定数です。

表 5–1 の関数定義では、*a* と *b* はパラメータで、置換シンボルまたは文字列定数を表します。文字列という用語は、パラメータの文字列値を表します。シンボル *ch* は、文字定数を表します。

表 5–1. 関数と戻り値

関数	戻り値
$symlen*(a)*	文字列 *a* の長さ
$symcmp*(a,b)*	< 0: *a* < *b* の場合 0: *a* = *b* の場合 > 0: *a* > *b* の場合
$firstch*(a,ch)*	文字列 *a* の中で文字定数 *ch* が最初に現れる位置
$lastch*(a,ch)*	文字列 *a* の中で文字定数 *ch* が最後に現れる位置
$isdefed*(a)*	1: 文字列 *a* がシンボル・テーブルで定義されている場合 0: 文字列 *a* がシンボル・テーブルで定義されていない場合
$ismember*(a,b)*	リスト *b* の先頭メンバが文字列 *a* に割り当てられる。 0: *b* がヌル文字列の場合
$iscons*(a)*	1: 文字列 *a* が 2 進数定数の場合 2: 文字列 *a* が 8 進数定数の場合 3: 文字列 *a* が 16 進数定数の場合 4: 文字列 *a* が文字定数の場合 5: 文字列 *a* が 10 進数定数の場合
$isname*(a*)	1: 文字列 *a* が有効なシンボル名である場合 0: 文字列 *a* が有効なシンボル名でない場合
$isreg*(a)* †	1: 文字列 *a* が有効な事前定義レジスタ名である場合 0: 文字列 *a* が有効な事前定義レジスタ名でない場合、0
$structsz(a)	構造体タグ *a* が表わす構造体のサイズ
$structacc(a)	構造体タグ *a* が表わす構造体の参照点

† 定義済みレジスタ名についての詳細は、3.10 節「シンボル」(3-30 ページ）を参照してください。

例 5–5 は、ビルトイン置換シンボル関数を示しています。

例 5–5. ビルトイン置換シンボル関数の使用

```
.asg   label, ADDR                ; ADDR = label
.if    ($symcmp(ADDR,"label") = 0); evaluates to true
SUB ADDR,AC0,AC0
.endif
.asg   "x,y,z" , list             ; list = x,y,z
.if    ($ismember(ADDR,list))     ; addr = x, list = y,z
SUB ADDR,AC0,AC0                  ; sub x
.endif
```

5.3.3 再帰的置換シンボル

アセンブラは置換シンボルを見つけると、対応する文字列の値に置換しようとします。その文字列も置換シンボルである場合は、アセンブラは再度置換を行います。アセンブラは、置換シンボルではないトークンを検出するまで、あるいはすでに評価の過程で検出されたことのある置換シンボルを検出するまで、置換を続けます。

例 5–6 では、x が z に置換されます。z は y に置換され、y は x に置換されます。アセンブラはこれが無限に再帰的であると判断し、置換を中止します。

例 5–6. 再帰的置換

```
      .asg   "x",z  ; declare z and assign z = "x"
      .asg   "z",y  ; declare y and assign y = "z"
      .asg   "y",x  ; declare x and assign x = "y"
       ADD x,AC0,AC0   ; recursive expansion
```

5.3.4 強制置換

場合によっては、アセンブラは置換シンボルを認識できないことがあります。1 対のコロンで示される強制置換演算子を使用すると、シンボルの文字列を強制的に置換できます。単純にシンボルをコロンで囲めば、置換を強制できます。コロンとシンボルの間には空白は入れないでください。

強制置換演算子の構文は次のとおりです。

```
:symbol:
```

アセンブラは、まずコロンで囲まれた置換シンボルを拡張してから、他の置換シンボルを拡張します。

強制置換演算子はマクロの中でのみ使用できますが、別の強制置換演算子でネストすることはできません。

例 5–7 は、強制置換演算子の使用方法を示しています。

例 5–7. 強制置換演算子の使用例

```
force  .macro x
       .loop  8
AUX:x: .set   x
       .eval  x+1,x
       .endloop
       .endm
       force 0
```

強制マクロは次のソース・コードを生成します。

```
AUX0   .set 0
AUX1   .set 1
   .
   .
   .
AUX7   .set 7
```

5.3.5 添字付き置換シンボルの個々の文字へアクセスする方法

マクロでは、添字付き置換シンボルを使用して置換シンボルの個々の文字（部分文字列）にアクセスできます。その際は、明確に指示するために強制置換演算子を使用する必要があります。

部分文字列へアクセスする方法は 2 つあります。

- :*symbol* (*well-defined expression*):

  この方法で添字を付けると、計算結果は 1 文字の文字列に表されます。

- :*symbol* (*well-defined expression*$_1$, *well-defined expression*$_2$):

  この方法では、expression$_1$ は部分文字列の開始位置を示し、expression$_2$ は部分文字列の長さを表します。添字を付ける正確な位置と、結果の文字列の正確な長さを指定できます。部分文字列のインデックスは、0 ではなく 1 から始まります。

例 5–8 と 例 5–9 は、添字付き置換シンボルとともに使われるビルトイン置換シンボル関数の例を示しています。

例 5–8 では、添字付き置換シンボルは、add 命令を再定義してこの命令がショート型の即値を処理できるようにしています。

例 5–8. 添字付き置換シンボルを使って命令を再定義する例

```
ADDX      .macro    ABC
          .var      TMP
          .asg      :ABC(1):,TMP
          .if       $symcmp(TMP,"#") = 0
          ADD ABC,AC0,AC0
          .else
          .emsg     "Bad Macro Parameter"
          .endif
          .endm

          ADDX      #100          ;macro call
          ADDX      *AR1          ;macro call
```

例 5–9 では、添字付き置換シンボルは、文字列 strg2 内の start 位置から始めて、部分文字列 strg1 を検索するために使われています。部分文字列 strg1 の位置は、置換シンボル pos に割り当てられています。

例 5–9. 添字付き置換シンボルを使用して部分文字列を見つける例

```
substr    .macro    start,strg1,strg2,pos
          .var      LEN1,LEN2,I,TMP
          .if       $symlen(start) = 0
          .eval     1,start
          .endif
          .eval     0,pos
          .eval     1,i
          .eval     $symlen(strg1),LEN1
          .eval     $symlen(strg2),LEN2
          .loop
          .break    i = (LEN2 - LEN1 + 1)
          .asg      ":strg2(i,LEN1):",TMP
          .if       $symcmp(strg1,TMP) = 0
          .eval     i,pos
          .break
          .else
          .eval     i + 1,i
          .endif
          .endloop
          .endm

          .asg      0,pos
          .asg      "ar1 ar2 ar3 ar4",regs
          substr    1,"ar2",regs,pos
          .data
          .word     pos
```

5.3.6 マクロでローカル変数として使われる置換シンボル

置換シンボルをマクロ内でローカル変数として使用する場合は、**.var** 疑似命令を使用します。マクロ 1 つにつき（パラメータを含めて）32 個までのローカル・マクロ置換シンボルを定義できます。.var 疑似命令は、ヌル文字列の初期値をもつ一時的な置換シンボルを作成します。このシンボルはパラメータとして渡されることはなく、展開後は消滅します。

.var sym_1 [,sym_2] ... [,sym_n]

.var 疑似命令は例 5–8 および例 5–9 で使用されています。

5.4 マクロ・ライブラリ

マクロを定義する方法の 1 つは、マクロ・ライブラリを作成することです。マクロ・ライブラリは、マクロ定義を含んだファイルの集合です。このようなファイルつまりメンバを集めて 1 つのファイル（アーカイブと呼ばれる）にまとめるには、アーカイバを使用する必要があります。マクロ・ライブラリのどのメンバにもマクロ定義が 1 つ含まれます。マクロ・ライブラリ内のファイルは、アセンブルされていないソース・ファイルでなければなりません。マクロ名とそのメンバ名は同一で、マクロ・ファイル名の拡張子は .asm でなければなりません。次に例を示します。

マクロ名	マイクロ・ライブラリ内のファイル名
`simple`	`simple.asm`
`add3`	`add3.asm`

マクロ・ライブラリにアクセスするには、.mlib アセンブラ疑似命令を使用します（4-74 ページで説明）。.mlib の構文は次のとおりです。

.mlib *macro library filename*

アセンブラは、ライブラリ・エントリを他のマクロと同じ方法で展開します。ライブラリ・エントリ展開のリスト出力は、.mlist 疑似命令を使用して制御できます。.mlist 疑似命令の詳細は、5.8 節「出力リストをフォーマットする方法」（5-21 ページ）を参照してください。抽出されるのは、ライブラリから実際に呼び出されたマクロだけです。

アーカイバを使用して必要なファイルをアーカイブに入れるだけで、マクロ・ライブラリを作成できます。アセンブラが、マクロ・ライブラリにはマクロ定義が入っているものとみなす点を除けば、マクロ・ライブラリは他のアーカイブと違いはありません。アセンブラは、マクロ・ライブラリの中にはマクロ定義だけが入っているものと考えます。ライブラリにオブジェクト・コードやいろいろなソース・ファイルを入れると、結果は保証されません。

5.5 マクロでの条件付きアセンブリの使用方法

条件付きアセンブリ疑似命令は、**.if/.elseif/.else/.endif** および **.loop/ .break/.endloop** です。条件付きアセンブリ疑似命令は、32 個のレベルまでネストできます。条件ブロックのフォーマットは次のとおりです。

.if *Boolean expression*
[**.elseif** *Boolean expression*]
[**.else**]
.endif

条件付きアセンブリ内では **.elseif** および **.else** の疑似命令の指定は任意です。.elseif 疑似命令は、条件付きアセンブリ・コード・ブロック内で 2 回以上使用できます。この 2 つの疑似命令が省略されて、かつ .if 式が偽（0）の場合、アセンブラは .endif 疑似命令に続くコードのアセンブルへと進みます。.if/ .elseif/.else/.endif 疑似命令の詳細は、4-61 ページを参照してください。

.loop/.break/.endloop 疑似命令を使用すると、コード・ブロックを繰り返してアセンブルできます。繰り返しが可能なブロックのフォーマットは、次のとおりです。

.loop *[well-defined expression]*
[**.break** [*Boolean expression*]]
.endloop

.loop 疑似命令の任意の式が計算されて、ループ・カウント（実行するループの数）に入ります。この式を省略すると、式が真（非ゼロ）になる .break 疑似命令をアセンブラが検出しない限り、ループ・カウントはデフォルトの 1024 となります。.loop/.break/.endloop 疑似命令の詳細は、4-72 ページを参照してください。

.break 疑似命令とその式の使用は任意です。式が計算されて偽となった場合、ループは継続します。アセンブラは、.break 式が真と計算された場合または .break 式が省略されている場合は、ループを終了します。ループを抜け出すと、アセンブラは .endloop 疑似命令の後のコードのアセンブルに移ります。

例 5–10、例 5–11 および例 5–12 は、.loop/.break/.endloop 疑似命令、適切なネストが行われている条件付きアセンブリ疑似命令、およびビルトイン置換シンボル関数が、条件付きアセンブリ・コード・ブロックでどのように使われているかを示しています。

例 5–10. *.loop/.break/.endloop* 疑似命令

```
       .asg   1 , x
       .loop

       .break(x == 10)  ;   if x == 10, quit loop/break with
                        ;   expression

       .eval  x+1,x
       .endloop
```

例 5–11. ネストされた条件付きアセンブリ疑似命令

```
       .asg   1,x
       .loop

       .if (x == 10);   if x == 10 quit loop
       .break        ;   force break
       .endif

       .eval  x+1,x
       .endloop
```

例 5–12. 条件付きアセンブリ・コード・ブロックと組み合わせて使用するビルトイン置換シンボル関数

```
       .ref OPZ
       .fcnolist
*
*Double Add or Subtract
*
DB     .macro ABC, ADDR, dst      ; add or subtract double

       .if $symcmp(ABC,"+") == 0
       ADD dbl(ADDR),dst       ; add double

       .elseif $symcmp(ABC,"-") == 0
       SUB dbl(ADDR),dst       ; subtract double

       .else
       .emsg  "Incorrect Operator Parameter"

       .endif

       .endm

*Macro Call
       DB  -, @OPZ, AC0
```

条件付きアセンブリ疑似命令の詳細は、4.8 節「条件付きアセンブラ疑似命令」(4-21 ページ）を参照してください。

5.6 マクロでのラベルの使用方法

アセンブリ言語プログラムで使用されるラベルは、どれも固有になる必要があります。マクロで使用されるラベルについても同様に固有にします。マクロが 2 回以上展開されていれば、ラベルも 2 回以上定義されています。ただし、2 回以上定義したラベルは不正です。マクロ言語は、マクロで定義するラベルが必ず固有になる方法を提供しています。ラベルの最後に疑問符を付ければ、アセンブラはその疑問符を固有の数字に変更します。マクロを展開しても、リスト・ファイル上でこの固有番号を参照することはできません。ラベルは、マクロ定義で行われたように疑問符が付いて表示されます。このラベルをグローバルとして宣言することはできません。

この固有の添字をもつラベルは、クロスリファレンス・リスティング・ファイルに出力されます。

固有のラベルの構文は次のとおりです。

```
label?
```

例 5–13 は、マクロでの固有のラベルの作成方法を示しています。

例 5–13. マクロでの固有のラベル

(a) ニーモニックの例

```
     1              ; define macro
     2              MLAB     .macro AVAR, BVAR ; find minimum
     3
     4                       MOV AVAR,AC0
     5                       SUB #BVAR,AC0,AC0
     6                       BCC M1?,AC0 < #0
     7                       MOV #BVAR,AC0
     8                       B M2?
     9              M1?      MOV AVAR,AC0
    10              M2?
    11                       .endm
    12
    13              ; call macro
    14 000000                MLAB 50, 100
1
1      000000  A064          MOV 50,AC0
1      000002  7C00          SUB #100,AC0,AC0
       000004  6400
1      000006  6320          BCC M1?,AC0 < #0
1      000008  7600          MOV #100,AC0
       00000a  6408
1      00000c  4A02          B M2?
1      00000e  A064 M1?      MOV 50,AC0
1      000010       M2?
```

例 5–13. マクロでの固有のラベル（続き）

(b) 代数表記の例

```
     1              ; define macro
     2              MLAB     .macro AVAR, BVAR ; find minimum
     3
     4                       AC0 = @(AVAR)
     5                       AC0 = AC0 - #(BVAR)
     6                       if (AC0 < #0) goto #(M1?)
     7                       AC0 = #(BVAR)
     8                       goto #(M2?)
     9              M1?      AC0 = @(AVAR)
    10              M2?
    11                       .endm
    12
    13              ; call macro
    14 000000                MLAB 50, 100
1
1      000000  A064          AC0 = @(50)
1      000002  7000          AC0 = AC0 - #(100)
       000004  6400
1      000006  7B20          if (AC0 < #0) goto #(M1?)
1      000008  6B00          AC0 = #(100)
       00000a  6480
1      00000c  0082          goto #(M2?)
1      00000e  A064 M1?      AC0 = @(50)
1      000010       M2?
```

5.7 マクロでのメッセージの作成方法

マクロ言語では、独自のアセンブル時エラー・メッセージと警告メッセージを定義できるように、3 つの疑似命令をサポートしています。この 3 つの疑似命令は、ユーザーが独自に必要とする特別のメッセージを作成するのに便利です。リスト・ファイルの最後の行に、エラーと警告の発行された回数が示されます。この回数によりコードの問題点を知ることができるので、特にデバッグ中に役立ちます。

.emsg	エラー・メッセージをリスト・ファイルに送ります。.emsg 疑似命令はアセンブラと同様にエラーを生成させます。つまりエラー回数を 1 つずつインクリメントさせ、アセンブラがオブジェクト・ファイルを作成するのを禁止します。
.mmsg	アセンブル時メッセージをリスト・ファイルに送ります。.mmsg 疑似命令は .emsg 疑似命令と同じ機能を果たしますが、エラー回数の設定をせず、かつオブジェクト・ファイルの作成を禁止しない点が異なります。
.wmsg	警告メッセージをリスト・ファイルに送ります。.wmsg 疑似命令は .emsg 疑似命令と同じ機能を果たしますが、警告の回数をインクリメントさせる点と、オブジェクト・ファイルの作成を禁止しない点が異なります。

マクロ・コメントはマクロの定義で使用されるコメントですが、マクロ展開の中には表示されません。1 カラム目の感嘆符（!）は、その行がマクロ・コメントであることを示しています。コメントをマクロ展開にも表示する場合には、コメントの前にアスタリスクかセミコロンを付けます。

例 5–14 は、マクロでのユーザー・メッセージを示しています。

例 5–14. マクロでのメッセージの作成

```
   1             testparam   .macro x,y
   2
   3                         .if ($symlen(x) == 0)
   4                         .emsg "ERROR -- Missing Parameter"
   5                          .mexit
   6                         .elseif ($symlen(y) == 0)
   7                          .emsg "ERROR == Missing Parameter"
   8                           .mexit
   9                          .else
   10                           MOV y,AC0
   11                           MOV x,AC0
   12                           ADD AC0,AC1
   13                         .endif
   14                         .endm
   15
   16 000000                  testparam 1,2
1
1                            .if ($symlen(x) == 0)
1                            .emsg "ERROR -- Missing Parameter"
1                             .mexit
1                            .elseif ($symlen(y) == 0)
1                            .emsg "ERROR == Missing Parameter"
1                             .mexit
1                            .else
1     000000 A004              MOV 2,AC0
1     000002 A102              MOV 1,AC1
1     000004 2401              ADD AC0,AC1
1                            .endif
   17
   18 000006                  testparam
1
1                            .if ($symlen(x) == 0)
1                            .emsg "ERROR -- Missing Parameter"
  ***** USER ERROR ***** - : ERROR -- Missing Parameter
1                             .mexit
 1 Error, No Warnings
```

5.8 出力リストをフォーマットする方法

マクロ、置換シンボル、および条件付きアセンブリ疑似命令は、情報を隠すことができます。この隠された情報を参照しなければならない場合があるので、マクロ言語は展開リスト作成機能をサポートしています。

デフォルトではアセンブラは、リスト出力ファイルにマクロ展開と偽の条件ブロックを出力します。リスト・ファイル内で、このリスト出力のオン／オフを切り換えることができます。4 組の疑似命令を使用して、この情報のリスト出力を制御できます。

- ❑ マクロ展開とループ展開のリスト

  .mlist マクロと .loop/.endloop ブロックを展開します。.mlist 疑似命令は、これらのブロックで検出したすべてのコードをリストに出力します。

  .mnolist マクロ展開と .loop/.endloop ブロックのリスト作成が抑止されます。

  マクロ展開とループ展開のリスト出力に関しては、.mlist がデフォルトです。

- ❑ 偽の条件ブロックのリスト

  .fclist アセンブラがコードを生成しないすべての条件付きブロック（偽の条件付きブロック）をリスト・ファイルに含めるようにします。条件付きブロックは、ソース・コード内と全く同じようにリストに出力されます。

  .fcnolist 偽の条件付きブロックのリスト出力を抑止します。条件付きブロック内の、実際にアセンブルを行うコードのみがリストに出力されます。.if、.elseif、.else、および .endif 疑似命令はリストには出力されません。

  偽の条件付きブロックのリスト出力に関しては、.fclist がデフォルトです。

- ❑ 置換シンボル展開リスト

  .sslist リスト内で置換シンボルを展開します。置換シンボルの展開のデバッグに有効です。展開された行は、実際のソース行の下に示されます。

  .ssnolist リスト内の置換シンボルの展開を抑止します。

  置換シンボル展開のリスト出力に関しては、.ssnolist がデフォルトです。

- ❑ 疑似命令のリスト

  .drlist アセンブラが、すべての疑似命令行をリスト・ファイルに出力するようにします。

  .drnolist リスト・ファイルへの次の疑似命令の出力を抑止します。.asg、.eval、.var、.sslist、.mlist、.fclist、.ssnolist、.mnolist、.fcnolist、.emsg、.wmsg、.mmsg、.length、.width、および .break。

  疑似命令のリスト出力に関しては、.drlist がデフォルトです。

5.9 再帰的なマクロとネストされたマクロの使用方法

マクロ言語では、再帰的なマクロ呼び出しとネストされたマクロ呼び出しをサポートしています。したがって、1 つのマクロ定義の中から別のマクロを呼び出すことができます。マクロは 32 個のレベルまでネストできます。再帰的なマクロを使用すると、マクロをその定義自体（マクロ呼び出し自体）から呼び出すことができます。

再帰的なマクロやネストされたマクロを作成する際、アセンブラではパラメータの動的な範囲設定が使用されるので、マクロ・パラメータに渡す引数には十分な注意が必要です。これは、呼び出されたマクロは、それを呼び出したマクロの環境を利用するということです。

例 5–15 は、ネストされたマクロを示しています。in_block マクロにある y が out_block マクロにある y を隠すことに注意してください。ただし、out_block マクロにある x と z には in_block マクロからアクセスできます。

例 5–15. ネストされたマクロの使用方法

```
in_block  .macro y,a
             .             ; visible parameters are y,a and
             .             ;   x,z from the calling macro
             .endm

out_block .macro x,y,z
             .             ; visible parameters are x,y,z
             .
          in_block x,y     ; macro call with x and y as
                           ;   arguments
             .
             .
          .endm
          out_block        ; macro call
```

例 5–16 は、再帰的なマクロを示しています。fact マクロは、n の階乗を計算するのに必要なアセンブリ・コードを作成します。ここで n は即値です。結果はデータ・メモリ・アドレス loc に入ります。fact マクロは fact1 を呼び出すことによりこれを実現しますが、fact1 は自分自身を再帰的に呼び出します。

例 5–16. 再帰的なマクロの使用方法

(a) ニーモニックの例

```
fact   .macro N, loc    ; n is an integer constant
                        ; loc memory address = n!
       .if N < 2        ; 0! = 1! = 1

       MOV #1,loc
       .else
       MOV #N,loc       ; n >= 2 so, store n at loc
                        ; decrement n, and do the
       .eval N - 1, N   ;      factorial of n - 1
       fact1            ; call fact1 with current
                        ;                 environment
       .endif

       .endm

fact1  .macro

       .if N > 1
       MOV loc,T3       ; multiply present factorial
       MOV T3,HI(AC2)   ; by present position
       MPYK #N,AC2,AC0
       MOV AC0,loc      ; save result
       .eval N - 1, N   ; decrement position
       fact1            ; recursive call
       .endif

       .endm
```

例 5–16. 再帰的なマクロの使用方法（続き）

(b) 代数表記の例

```
fact   .macro N, loc   ; n is an integer constant
                       ; loc memory address = n!
       .if N < 2       ; 0! = 1! = 1

       loc = #1
       .else
       loc = #N        ; n >= 2 so, store n at loc
                       ; decrement n, and do the
       .eval N - 1, N  ;      factorial of n - 1
       fact1           ; call fact1 with current
                       ;                 environment
       .endif

       .endm

fact1  .macro

       .if N > 1
       T3 = loc    ;    multiply present factorial
       HI(AC2) = T3    ;  by present position
       AC0 = AC2 * #(N)
       loc = AC0   ;    save result
       .eval N - 1, N  ; decrement position
       fact1           ; recursive call
       .endif

       .endm
```

5.10 マクロ疑似命令のまとめ

表 5–2. マクロの作成

ニーモニックと構文	説明
macname **.macro** [*parameter*$_1$]...[*parameter*$_n$]	マクロを定義します。
.mlib *filename*	マクロ定義を含むライブラリを特定します。
.mexit	.endm に移動します。
.endm	マクロ定義を終了します。

表 5–3. 置換シンボルの操作

ニーモニックと構文	説明
.asg [**"**]*character string*[**"**], *substitution symbol*	文字列を置換シンボルに割り当てます。
.eval *well-defined expression, substitution symbol*	数値置換シンボルの算術計算を実行します。
.var *substitution symbol*$_1$...[*substitution symbol*$_n$]	ローカル・マクロ・シンボルを定義します。

表 5–4. 条件付きアセンブリ

ニーモニックと構文	説明
.if *Boolean expression*	条件付きアセンブリを開始します。
.elseif *Boolean expression*	任意の条件付きアセンブリ・ブロックです。
.else	任意の条件付きアセンブリ・ブロックです。
.endif	条件付きアセンブリを終了します。
.loop [*well-defined expression*]	反復使用が可能なブロック・アセンブリを開始します。
.break[*Boolean expression*]	任意の反復使用が可能なブロック・アセンブリです。
.endloop	反復使用が可能なブロック・アセンブリを終了します。

表 5–5. アセンブル時メッセージの作成

ニーモニックと構文	説明
.emsg	エラー・メッセージを標準出力に送ります。
.wmsg	警告メッセージを標準出力に送ります。
.mmsg	警告メッセージまたはアセンブル時メッセージを標準出力に送ります。

表 5–6. リストのフォーマット

ニーモニックと構文	説明
.fclist	偽の条件付きコード・ブロックのリスト出力を許可します（デフォルト）。
.fcnolist	偽の条件付きコード・ブロックのリスト出力を抑止します。
.mlist	マクロ・リストの出力を許可します（デフォルト）。
.mnolist	マクロ・リストの出力を抑止します。
.sslist	置換シンボルの展開リスト出力を許可します。
.ssnolist	置換シンボルの展開リスト出力を抑止します（デフォルト）。

第 6 章

C55x における C54x コードの実行

TMS320C55x ™ ソース・コードに加え、C55x ニーモニック・アセンブラは TMS320C54x ™ ニーモニック・アセンブリも受け入れます。C54x 命令セットは、211 の命令を含んでいます。C55x ニーモニック命令セットは、C54x 命令セットのスーパーセットです。以下の表には、C54x 命令が masm55 を使ってアセンブルする方法について、その統計を示しています。

オリジナルの C54x 命令のアセンブル方法	C54x 命令セット全体の割合い	通常使用される C54x 命令の割合い
1 つの C55x 命令	85	95-99
2 つの C55x 命令	10	1-3
3 つ以上の C55x 命令	5	0-2

2 列目のデータは、あらゆる C54x 命令のインスタンスを含む仮想ファイルのアセンブリの特徴を決定します。しかし、2 つ以上の命令をアセンブルする命令は、通常使われません。3 列目のデータは、最も一般的に使われる C54x 命令を含むファイルのアセンブリの特徴を決定します。正確なパーセンテージは、使用する個別のソース・ファイルにより異なります。

この互換性により、C55x ニーモニック・アセンブラは C54x コードをアセンブルし、C55x オブジェクト・コードを生成できます。このオブジェクト・コードは、実行中全く同じ結果を計算します。このアセンブラの機能により、ユーザーが C55x に変換する際、ユーザーの C54x ソース・コードへの投資は保持されます。

本章では、C55x の新しいアーキテクチャ機能の利点については説明しません。この点については、「TMS320C55x DSP プログラマーズ・ガイド」を参照してください。

項目 **ページ**

6.1 C54x から C55x への開発フロー

C55x において C54x アプリケーションを実行するには、以下の操作が必要です。

- ❏ cl55 で各関数のアセンブル。C54x アプリケーションは、cl500 アセンブラを使い、エラーの発生なくアセンブルを完了している必要があります。C54x コードの移植をサポートする cl55 オプションについての詳細は、7.2 節（7-5 ページ）を参照してください。
- ❏ スタック・ポインタ SP および SSP の初期化。6.1.1 項を参照してください。
- ❏ メモリ配置における違いの処理。6.1.2 項を参照してください。
- ❏ C55x の C54x リンカ・コマンド・ファイルの更新。6.1.3 項を参照してください。

ネイティブ C55x 関数とともに、移植された C54x 関数を使用するには、7.3 節「ネイティブ C55x 関数と移植された C54x 関数を混在して使用する方法」（7-10 ページ）を参照してください。

6.1.1 スタック・ポインタの初期化

リセットから移植された C54x コードを実行する場合、適切な実行環境は既に整っています。しかし、スタック・ポインタ SP（プライマリ・スタック）および SSP（二次システム・スタック）を初期化する必要があります。次に例を示します。

```
stack_size        .set  0x400
stack:            .usect "stack_section", stack_size
sysstack:         .usect "stack_section", stack_size

                  AMOV #(stack+stack_size), XSP
                  MOV #(sysstack+stack_size), SSP
```

スタックは、高いアドレスから低いアドレスへと変化します。そのため、スタック・ポインタは最も高いアドレスに初期化する必要があります。プライマリ・スタックおよび二次システム・スタックは、メモリの同じ 64K ワード・ページ内になければなりません。

SP を変更するコードは、移植できます。このような変更は、直接的にも間接的にも行えます。場合によっては、SSP も変更しなければならないという警告を受けません。

6.1.2 メモリ配置における違いの処理

この節では、メモリ内でコードを配置できる位置について、その制約を説明します。データはすべて、最初の 64K ワードに配置する必要があります。

C54x コードに以下の内容が含まれている場合、コードはすべて 64K バイトに配置しなければなりません。

- ❏ CALA を使った間接コール
- ❏ リピート・ブロック・アドレス・レジスタ REA または RSA の変更

- ❑ デバイス・リビジョンを指定するために -v オプションを使わない場合、BACC を使った間接分岐。

C54x コードに以下の内容のいずれかが含まれている場合、コードは 64K バイト境界を超えずにいずれかの 64K バイト・ブロックに配置できます。

- ❑ デバイス・リビジョンを指定するために適切な -v オプションを使って作成する場合、BACC を使った間接分岐。
- ❑ 非標準的な方法（スタックの展開）による、スタック上の関数リターン・アドレスの変更または使用。

上記の内容が含まれていない場合、コードはメモリ内のどこにでも配置できます。

6.1.3 C54x リンカ・コマンド・ファイルの更新

C54x リンカ・コマンド・ファイルを C55x システムで使用するには、以下の情報を考慮する必要があります。

- ❑ C55x　リンカ・コマンド・ファイルでは、すべてのアドレスと長さ（コードおよびデータの両方）はバイト単位で表記されます。データは、プロセッサ上のアドレッシングはワード単位ですが、表記はバイト単位なので注意してください。したがって、-heap および -stack オプションの割り当てはワードではなくバイトで指定します。

- ❑ C54x では、メモリは 2 つの異なるページに別れます。コードはページ 0、データはページ 1 です。各ページのアドレス空間の範囲は 0 から 0xFFF（ワード単位）です。C55x には、0 から 0xFFFFFF の範囲にわたる、単一のアドレス空間があります。

- ❑ C55x では、すべてのセクションは固有のアドレスを持つ必要があり、重なってはいけません。C54x ではコードとデータは別のページにあるため、セクションは同じアドレスを持つことができ、重なっても構いません。

- ❑ DP に基づいた直接メモリ・アドレス（DMA）を使用する場合、DP 境界と DMA を使ってアクセスする変数の間の関係を変えないように注意して下さい。C54x では、DP ページは 128 ワードの長さです。128 ワード境界で開始しなければなりません。cl55 によって移植された C54x コードにも、同じ制約があります。しかし、リンカ・コマンド・ファイルにおいてこの制約の表記は異なっています。リンカはバイト・アドレスを使用するため、DP ページは 256 バイトの長さであり、256 バイト境界で開始しなければなりません。

  .bss または .usect アセンブラ疑似命令のブロッキング・パラメータを使うことにより、同じ DP ページ上に変数を配置できます。ブロッキング・パラメータを使う場合、リンカ・コマンド・ファイルを変更する必要はありません。

  リンカ・コマンド・ファイルを使って同じ DP ページ上に変数を配置するには、128 ワードの指定を 256 バイトに変更しなければなりません。たとえば、以下のような指定は変更する必要があります。

  output_section `ALIGN(128) {` *list of input sections* `}`

変更

output_section `ALIGN(256) {` *list of input sections* `}`

6.2 リスト・ファイルについて

アセンブラのリスト・ファイル（-al オプションを使って cl55 を起動する際に作成されます）は、C54x 命令を C55x にマッピングする方法について詳細を提供します。

次の C54x ソース・ファイルの例を見てみましょう。

```
          .global name

          ADD     *AR2, A
          LD      *AR3, B

          RPT     #10
          MVDK    *AR4+, name

  subm    .macro  mem1, mem2, reg
          LD      mem1, reg
          SUB     mem2, reg
          .endm

          subm    name, *AR6, B

          MOV     T1, AC3       ; native C55x instruction
```

以下に示すリスト・ファイルには、分類のために説明が挿入されています。

ファイルは、ファイルの移植に使われる C55x 一時レジスタのコメントで始まっています。

```
16          ; Temporary Registers Used: XCDP
```

このコメントは、コードの移植に一時レジスタが必要な場合にのみ表示されます。一時レジスタは、ファイルの後半で !REG! というコメントで開始するエンコードで使われます（この例は 7 行目に示しています）。

C55x と同じ構文で使われる C54x 命令（以下の ADD 命令など）は、特別な注記なしに表示されます。

```
1                    .global name
2
3   000000  D641     ADD *AR2,A
    000002  00
```

上記の例の A は、C55x で AC0 にマッピングされますが、受け入れられます。

C55x と異なる構文で使われる C54x 命令でも、単一行のマッピングの場合は、特別な注記なしに表示されます。

```
4   000003  A161     LD *AR3, B
```

上記の LD 命令は次のように表記できます。

```
                     MOV *AR3, AC1
```

以下のコードは、C55x 命令の順序をオリジナルのソースと変える必要のある複数行命令マッピングを示しています。この複数行エンコードは C55x 一時レジスタを使う必要があるため、オリジナルのソースを繰り返す !REG! 行で開始します。マッピングに対応する複数行は、オリジナルのソース行番号（この場合は 7）で開始し、終了します。

```
7   ******  !REG!    MVDK *AR4+, name
7   000005  EC31     AMAR *(#(name)), XCDP  ; port of
    000007  7E00                            ; MVDK *AR4+, name
    000009 0000!
5
6   00000b  4C0A     RPT #10
7   00000d  EF83     MOV *AR4+, coef(*CDP+) ; port of
            00000f   05                     ; MVDK *AR4+, name
```

まとめると、上記例ではオリジナルの C54x コードは、

```
                     RPT #10
                     MVDK *AR4+, name
```

が以下のようにマッピングされました。

```
                     AMAR *(#(name)),XCDP
                     RPT #10
                     MOV *AR4+, coef(*CDP+)
```

一時レジスタが不要の複数行マッピングは、PORT コメントでマークされています。

マクロ定義は単純に繰り返されます。

```
8
9             subm    .macro  mem1, mem2, reg
10                    LD      mem1, reg
11                    SUB     mem2, reg
12                    .endm
```

マクロ起動は MACRO 行でマークされています。マクロ展開では、上記で説明した例のいずれかになります。

```
13
14  ******  MACRO    subm name, *AR6, B
14  000010  A100%    LD  name, B
14  000012  D7C1     SUB *AR6, B
    000014  11
```

ネイティブ C55x 命令は、特別な注記なしに表示されます。ネイティブ C55x コードとともに、移植された C54x 関数を使用する場合の詳細については、7.3 節「ネイティブ C55x 関数と移植された C54x 関数を混在して使用する方法」（7-10 ページ）を参照してください。

```
15
16  000015  2253     MOV T1, AC3  ; native C55x
```

6.3 C55x 予約済み名前の処理

新しい C55x ニーモニックおよびレジスタは、予約語です。C54x コードに、C55x ニーモニックまたはレジスタとして使われているシンボル名が含まれてはいけません。たとえば、シンボル名として T3 は使えません。

C54x コードには、C55x 代数表記構文の予約語であるシンボル名も含まれてはいけません。たとえば、return と名付けられたラベルは使えません。

C55x ニーモニック・アセンブラは、シンボル名コンフリクトを検出するとエラー・メッセージを発行します。

第 7 章

C54x システムから C55x システムへの移行

第 6 章の説明に従い TMS320C54x™ コードを移植した後、C54x コードを TMS320C55x™ に移動した場合の様々なシステムレベルについて考慮する必要があります。本章は以下の内容を説明します。

- ❑ 割り込みに関する違いの処理方法
- ❑ ネイティブ C55x 関数を使い、移植された C54x 関数を使用する方法
- ❑ 移植できない C54x コーディングの例

項目 **ページ**

7.1 割り込みの処理

この節では、割り込みに関する処理について説明します。

7.1.1 割り込みベクター・テーブルの違い

C54x 割り込みテーブルは、32 のベクターにより構成されています。各ベクターには 4 ワードの実行可能なコードが含まれています。C55x ベクター・テーブルも、32 のベクターにより構成されています。両方のテーブルのベクターは同じ長さですが、C55x では長さは 8 バイトと数えられます。

割り込みベクター・テーブルにおけるベクターの順序は、ユーザーのシステム固有のデバイスごとに、データ・シートに記載されています。ベクターの順序はデバイス固有のため、IMR または IFR レジスタへのアクセスはデバイスに応じて更新する必要があります。同様に、TRAP 命令を使う場合、オペランドを更新しなければならない場合があります。

C54x と C55x では、ベクターの内容の処理方法が異なります。これらの違いに対応するには、C54x ベクターそのものを変更しなければなりません。

C55x ベクター・テーブルでは、最初のバイトは無視され、次の 3 バイトが割り込みサービス・ルーチン（ISR）のアドレスとして割り込まれます。以下の例に示すように、.ivec アセンブラ疑似命令を使い、C55x ベクター入力を初期化します。.ivec 疑似命令の詳細は、4-64 ページを参照してください。

ISR への単純分岐

C54x ベクターが以下を含んでいる場合、

```
B isr
```

対応する C55x ベクターを以下のように変更します。

```
.ivec isr
```

ISR への遅延分岐

C54x ベクターが以下を含んでいる場合、

```
BD isr
inst_1      ; two instruction words of code
inst_2
```

最も簡単な解決法として、ベクターを以下のように書きます。

```
.ivec isr
```

そして、命令 *inst1* および *inst2* を、ISR の最初に移動します。*inst1* の変換が、4 バイト以下の単一の C55x 命令の場合、ベクターに配置できます。しかし、*inst2* は必ず ISR に移動しなければなりません。

ベクターが ISR 全体を含んでいる場合

以下の例のように、C54x ベクターが 4 ワードの ISR 全体を含んでいる場合、独立したルーチンとして 4 ワードの ISR を作成しなければなりません。

```
; example 1
inst1
inst2
inst3
RETF

; example 2
inst1
RETFD
inst2
inst3

; example 3
CALL routine1
RETE
nop
```

さらに、C55x ベクター・テーブルに該当するルーチンのアドレスを提供しなければなりません。

```
.ivec new_isr
```

7.1.2 割り込みサービス・ルーチンの処理

割り込みサービス・ルーチンを変更する必要があるのは、複数の C55x 命令にマッピングされる C54x 命令を含む場合、または C55x 命令の 1 つが C55x レジスタまたはビットを一時的に使用しなければならない場合に限ります。

この場合、新しい C55x レジスタはルーチンによって保持される必要があります。

複数行命令マッピングで一時的に使用できる C55x レジスタのリストについては、7.3.2 項「一時的に使用される C55x レジスタ」（7-11 ページ）を参照してください。

割り込みを動作させるために、レジスタの全リストを確保できます。または、単に使用するレジスタを確保することもできます。

1） -al オプションで、cl55 を使って ISR をアセンブルし、リスト・ファイルを作成します。

2） リストをチェックし、以下のように、ファイルの上部に一時レジスタのコメントが含まれているかどうかを確認します。

```
16              ; Temporary Registers Used:XCDP
```

このコメントでは、ファイルの移植に使われるすべての一時レジスタのリストが提供されます。詳細は、6.2 節「リスト・ファイルについて」（6-4 ページ）を参照してください。

3） 一時レジスタが使われる場合、適切なレジスタまたはビットを ISR の最初でスタックにプッシュし、最後でスタックからポップする必要があります。

7.1.3 割り込みに関するその他の注意

以下に説明する割り込みについて注意する必要があります。

- ❑ アセンブラが RETE、RETED、FRETE、FRETED、RETF または RETFD を検出すると、警告が発行されます。これらの命令がある場合、アセンブラは割り込みサービス・ルーチンまたは割り込みベクター・テーブル本体を処理するため、命令を正しく移植できない場合があります。

- ❑ INTR は、C54x および C55x の両方について、同じニーモニック構文を持っています。したがって、使用可能なネイティブ C55x 割り込みなのか、もしくは（割り込み番号が正しくない可能性のある）C54x 割り込みなのか、アセンブラは識別することができません。

- ❑ 割り込みベクター･テーブルの位置を示す PMST の 9 ビット･フィールドである IPTR に対し、コードが値を書き込む場合、コードを変更し、C55x システムの変更を反映させる必要があります。

7.2 C54x コードのアセンブラ・オプション

cl55 アセンブラは、いくつかのオプションを提供し、C54x アセンブリ・コードの C55x への移植をさらにサポートします。これらのオプションを使うと、アセンブラは以下のことができます。

- ❏ SST の無効化（-mt オプション）
- ❏ 速度優先の移植（-mh オプション）
- ❏ C54x 固有のサーキュラ・アドレッシングのエンコード（--purecirc オプション）
- ❏ 遅延スロットからの NOP の除去（-mn オプション）

7.2.1 SST の無効化（-mt オプション）

デフォルトでは、アセンブラは SST ビット（格納時は飽和）を有効とします。たとえば、SST の設定により、アセンブラは STH および STL 命令を以下のように移植します。

C54x 命令	デフォルトの C55x エンコード	バイト
`STH src, Smem`	`MOV HI(ACx << #0), Smem`	3
`STL src, Smem`	`MOV ACx << #0, Smem`	3

シフト（<< #0）を使うと、C54x が提供するのと同じ格納時飽和状態動作を達成します。コードで SST が無効化されている場合でも、このエンコードは機能します。

しかし、飽和動作が必要でない場合、-mt アセンブラ・オプションを使ってさらに最適なエンコードを生成します。

C54x 命令	-mt を使った C55x エンコード	バイト
`STH src, Smem`	`MOV HI(ACx), Smem`	2
`STL src, Smem`	`MOV ACx, Smem`	2

-mt オプションはファイル全体に影響を与えます。ファイル内の SST モードを切り替えるには、.sst_on および .sst_off アセンブラ疑似命令を使用します。

.sst_on 疑似命令は、1 に設定された SST ステータス・ビットをモデル化します。これはアセンブラのデフォルトの設定です。.sst_off 疑似命令は、0 に設定された SST ステータス・ビットをモデル化します。これは、-mt アセンブラ・オプションを使用することと同等です。コマンド行オプションと疑似命令の間でコンフリクトが起こった場合、疑似命令が優先します。

.sst_on および .sst_off 疑似命令の有効範囲は静的であり、アセンブリ・プログラムの制御フローには影響されません。.sst_off 疑似命令と .sst_on 疑似命令との間にあるすべてのアセンブリ・コードはすべて、SST が無効であるという設定でアセンブルされます。コマンド行オプションを使わずに SST ビットを無効化するよう指示するには、.set_off 疑似命令を各ソース・ファイルの上部に配置します。

7.2.2 速度優先の移植（-mh オプション）

デフォルトでは、アセンブラは C54x コードをコードサイズが小さくなることを目標にエンコードします。たとえば、AR *x* レジスタを書き込む MVMM および STM 命令のエンコードを考えてみましょう（以下の STM 命令では、*const* は -15 ～ 15 の範囲の定数です）。

C54x 命令	デフォルトの C55x エンコード	バイト
`MVMM ARx, ARy`	`MOV ARx, ARy`	2
`STM #const, ARx`	`MOV #const, ARx`	2

-mh アセンブラ・オプションを使って、より「速い」エンコードを生成できます。

C54x 命令	デフォルトの C55x エンコード	バイト
`MVMM ARx, ARy`	`AMOV ARx, ARy`	3
`STM #const, ARx`	`AMOV #const, ARx`	3

MOV 命令は、パイプラインの実行フェーズにおいて AR*y* を書き込みます。AMOV は、アドレス・フェーズにおいて AR*y* を書き込みます。これは 4 サイクル前です。MVMM または STM に続く命令が AR*y* を逆参照する場合（たとえば *AR3+）、MOV は 4 サイクル・ストールを課し、AR*y* が書き込まれるのを待ちます。AMOV はストールを課しません。AMOV エンコードは、1 バイトのエンコード・スペースを犠牲にして、スピードにおける重要なゲインを提供します。

-mh オプションはファイル全体に影響を与えます。ファイル内で「速度優先の移植」モードを切り替えるには、.port_for_speed および .port_for_size アセンブラ疑似命令を使います。

.port_for_size 疑似命令は、アセンブラのデフォルト・エンコードをモデル化します。.port_for_speed 疑似命令は、-mh アセンブラ・オプションの働きをモデル化します。コマンド行オプションと疑似命令の間でコンフリクトが起こった場合、疑似命令が優先します。

重要なループの直前には .port_for_speed の使用を考えてみましょう。ループの後で、デフォルトのエンコードに戻るためには .port_for_size を使います。

7.2.3 C54x サーキュラ・アドレッシングの最適なエンコード（--purecirc オプション）

移植した C54x コードが、C55x リニア／サーキュラ・アドレッシング・ビットを使わず C54x サーキュラ・アドレッシングを使っている場合、--purecirc オプションを使います。このオプションを使うと、アセンブラはサーキュラ・アドレッシング・コードに最も適したエンコードを生成できます。

次の C54x コードには

```
      RPTB   end-1
      NOP ; 1
      MAC   *AR5+, *AR3+0%, A
      NOP ; 2
   end
```

--purecirc を使わずに作成すると、このコードが生成されます。

```
      RPTB   end-1
      NOP ; 1
      BSET   AR3LC
      MACM   T3 = *AR5+, *(AR3+AR0), AC0, AC0
      BCLR   AR3LC
      NOP ; 2
   end:
```

AR3 のリニア／サーキュラ・ビットの切り替え命令がループ内部にあることに注意してください。--purecirc を使って作成すると、このコードが生成されます。

```
      BSET   AR3LC
      RPTB   P04_3
      NOP ; 1
      MACM   T3 = *AR5+, *(AR3+AR0), AC0, AC0
   P04_3:
      NOP ; 2
      BCLR   AR3LC
   end:
```

今回は AR3 のリニア／サーキュラ・ビットの切り替え命令がループの外にあります。

特定のコーディングの例により、--purecirc オプション使用時にも最適なサーキュラ・アドレッシング・コードが妨害される場合があります。

❑ 未使用ラベル

以下のコードで、ラベル「middle」は未使用です。

```
start:
         RPTB   end-1
            LD   *AR4, A
middle:       ; unused label
            MAR   *AR4-0%
end:
```

未使用ラベルがループから削除されると、アセンブラは MAR 命令のサーキュラ・ビット・オペレーションをループの外に移動できます。未使用ラベルが削除されない場合、サーキュラ命令はループ内にとどまり、ループは 4 バイト大きく、4 サイクル長くなります。

- 同じループ内におけるサーキュラおよび非サーキュラのためのレジスタの使用

  次のコードを考えてみましょう。

```
        RPTB    end-1
        ; reference to AR3 (circular)
        MAC     *AR5+, *AR3+0%, A

        ...

        ; reference to AR3 (non-circular)
        ST      A, *AR3+
   ||   SUB     *AR2, B

        ...
end:
```

  2 番目の AR3 の参照が非サーキュラのため、MAC 命令のサーキュラ・ビット・オペレーションはループの外に移動できません。ループ内における AR*x* の間接参照 1 つがサーキュラ・アドレッシングを使用している場合、そのループ内におけるレジスタの間接参照すべてもまたサーキュラ・アドレッシングを使用していれば、移動は可能となります。

7.2.4 遅延スロットの NOP 除去（-atn および -mn オプション）

-atn または -mn オプションが指定された場合、アセンブラは C54x の遅延分岐または呼び出し命令の遅延スロットに位置する NOP を除去します。

たとえば、-mn オプションを使った次の C54x コードは :

```
CALLD   func
LD      *AR2, A
NOP
; call occurs here
```

cl55 リスト・ファイルには次のように表示されます。

```
4   000000   A041   LD   *AR2, A
2
3   000002   6C00   CALLD   func
   000004 0000!
5   ****** DEL   NOP
6               ; call occurs here
```

命令コード・フィールドの DEL は、削除された NOP を示します。

7.3 ネイティブ C55x 関数と移植された C54x 関数を混在して使用する方法

C54x アプリケーションを完全なネイティブ C55x コードに書き換える場合、一度に 1 つの機能しか働かないことを考慮してテストを続けます。問題を検出した場合、直前に行われた変更の中で簡単に見つけることができます。この処理を通じて、移植された C54x コードおよびネイティブ C55x コードの両方を使って作業できます。以下の点に注意してください。

- ❏ 同じ関数内に C54x および C55x の命令を混在させないこと。
- ❏ 移植された C54x 命令とネイティブ C55x 命令間の変わり目は、関数呼び出しおよびリターンの時にのみ行うこと。
- ❏ C コンパイラは、C コード呼び出しアセンブリに C54X_CALL プラグマを提供します。しかし、7.3.7 項の例で、移植された C54x アセンブリ関数を C コードから呼び出す場合の veneer 関数の使用に関する説明を見てください。C54X_CALL についての詳細は、TMS320C55x C コンパイラの最適化入門マニュアルを参照してください。

7.3.1 移植された C54x コードの実行環境

実行環境とは、レジスタ、ステータス・レジスタ・ビット設定、およびスタックなどの機械リソースの使用を管理する、仮定および規則です。移植された C54x コードが使用する実行環境は、ネイティブ C55x コードの使用する環境とは異なります。リセットから移植された C54x コードを実行する場合、適切な実行環境は既に整っています。しかし、一方のコードから他のコードへ変わる場合、特定の環境を構成するステータス・ビットおよびレジスタ設定を確認することは重要です。

以下の CPU 環境は、移植された C54x 関数の実行に要求される環境です。

- ❏ 32 ビット・スタック・モード。
- ❏ SP および SSP は初期化し、スタックのために確保されたメモリに指定する必要があります。6.1.1 項「スタック・ポインタの初期化」（6-2 ページ）を参照してください。

- ステータス・ビットは以下のように設定する必要があります。

ステータス・ビット	設定値
C54CM	1
M40	0
ARMS	0
RDM	0
ST2[7:0]（サーキュラ・アドレッシング・ビット）	0

- アドレッシング・レジスタの上位ビット（DPH、CDPH、AR*n*H、SPH）は、0 に設定する必要があります。

- BSA*xx* レジスタは、0 に設定する必要があります。

7.3.2 一時的に使用される C55x レジスタ

以下の C55x レジスタは、cl55 が生成する複数行マッピングでは一時的に使用される場合があります。

- T0
- T1
- AC2
- AC3
- CDP
- CSR
- ST0_55（TC1 ビットのみ）
- ST2_55

これらのレジスタを使用する割り込みルーチンは、レジスタを保存および復元する必要があります。詳細は、7.1.2 項「割り込みサービス・ルーチンの処理」（7-3 ページ）を参照してください。

移植された C54x コードを呼び出すネイティブ C55x コードは、移植されたコードがこれらのレジスタを上書きする可能性があることを考慮する必要があります。

7.3.3 C54x から C55x へのレジスタ・マッピング

以下の C54x レジスタは、下記のように C55x レジスタにマッピングされます。

C54x レジスタ	C55x レジスタ
T	T3
A	AC0
B	AC1
AR*n*	AR*n*
IMR*n*	IER*n*
ASM（ST1 におけるステータス・ビット）	T2

7.3.4 T2 レジスタ使用時の警告

C54CM モードは、cl55 によって移植された C54x コードを自動的に実行する際に必要になるモードですが、このモードでは、cl55 による移植されたコードとまったく同じように ST1 の ASM フィールドを正確にモデル化する以外の目的で T2 レジスタを使用することはできません。C54CM では、ステータス・レジスタ ST1_55 が書き込まれるときはいつでも、下位 5 ビット（ASM フィールド）に符号拡張されて T2 に自動的にコピーされます。

割り込みが発生すると、ST1_55 は自動的に保存および復元されます。復元されると、T2 への自動コピーが再開されます。このような割り込み時の自動上書きのため、ASM フィールドを使わない C54x コードのセクションでも、T2 を汎用レジスタとして使うことはできません。

7.3.5 ステータス・ビット・フィールド・マッピング

C55x ステータス・ビット・フィールドは、以下のように C54x ステータス・ビット・フィールドにマッピングされます。

表 7–1. ST0_55 ステータス・ビット・フィールド・マッピング

ビット数	C55x フィールド	C54x フィールド（ST0 において）
15	ACOV2	なし
14	ACOV3	なし
13	TC1	なし
12	TC2	TC
11	CARRY	C
10	ACOV0	OVA
9	ACOV1	OVB
8-0	DP	DP

表 7–2. ST1_55 ステータス・ビット・フィールド・マッピング

ビット数	C55x フィールド	C54x フィールド（ST1 において）
15	BRAF	BRAF
14	CPL	CPL
13	XF	XF
12	HM	HM
11	INTM	INTM
10	M40	なし
9	SATD	OVM
8	SXMD	SXM
7	C16	C16
6	FRCT	FRCT
5	C54CM	なし
4-0	ASM	ASM

表 7–3. ST2_55 ステータス・ビット・フィールド・マッピング

ビット数	C55x フィールド	C54x フィールド
15	ARMS	なし
14-13	Reserved	なし
12	DBGM	なし
11	EALLOW	なし
10	RDM	なし
9	Reserved	なし
8	CDPLC	なし
7-0	AR*n*LC	なし

表 7–4. ST3_55 ステータス・ビット・フィールド・マッピング

ビット数	C55x フィールド	C54x フィールド（PMST において）
15-8	Reserved	なし
7	CBERR	なし
6	MPNMC	MP/MC_
5	SATA	なし
4	Reserved	なし
3	Reserved	なし
2	CLKOFF	CLKOFF
1	SMUL	SMUL
0	SST	SST

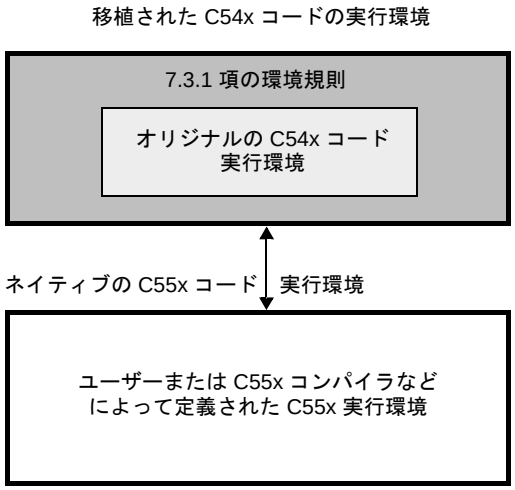
7.3.6 実行環境の切り替え

7.3.1 項で定義されている実行環境は、C55x について新しいレジスタおよびステータス・ビットを定義しているに過ぎないため、完全ではありません。C55x について新しくないレジスタおよびステータス・ビットは、オリジナルの C54x コードから規則を継承しています（7.3.3 項に示すとおり、一部のレジスタには新しい名前が付いています）。

ネイティブ C55x コードの実行環境が、移植された C54x コードのために定義された環境とは異なる場合、環境の切り替え時に以下の項目について適切な調整をする必要があります。

- ステータス・ビット・フィールド値の確保
- レジスタの確保
- 引数の渡し方
- 結果の返し方

図 7-1. 移植された C54x コードおよびネイティブ C55x コードのための実行環境



7.3.7 C54x アセンブリを呼び出す C コードの例

この例では、コンパイルされた C コードから C54x アセンブリ・ルーチンへの呼び出しを処理する技術について説明しています。この例では、ネイティブの C55x コードと移植された C54x コードの間に追加の関数が挿入されています。この関数は、*veneer* 関数と呼ばれ、2 つの実行環境を切り替えるためのコードを提供します。

注 : コンパイラ・プラグマ

コンパイラは、2 つのプラグマを提供し、C54X_CALL および C54X_FAR_CALL の作業を行います。これらのプラグマを使用している場合、veneer を自分で書く必要はありません。C54x および C55x の C コンパイラの実行環境はどちらも明確に定義されており、この例で示す技術をより具体的にし、ユーザーの状況に適用し易くします。

例 7–1. 呼び出された関数の C プロトタイプ

```
short firlat(short *x, short *k, short *r, short *dbuffer,
             unsigned short nx, unsigned short nk);
```

例 7–2. アセンブリ関数 _firlat_veneer

```
      .def   _firlat_veneer
      .ref   _firlat

_firlat_veneer:

; Saving Registers -------------------------
      PSH      AR5
      ; PSH   AR6      ; saved in ported C54x environment
      ; PSH   AR7      ; ditto
      PSH      T2
      PSH      T3

; Passing Arguments ------------------------
      PSH      T1      ; push rightmost argument first
      PSH      T0      ; then the next rightmost
      PSH      AR3      ; and so on
      PSH      AR2
      PSH      AR1

      MOV      AR0, AC0   ; leftmost argument goes in AC0

; Change Status Bits -----------------------
      BSET   C54CM
      BCLR   ARMS
      BCLR   C16

; Call -------------------------------------
      CALL   _firlat

; Restore Status Bits ----------------------
      BCLR   C54CM
      BSET   ARMS
      BSET   SXMD

; Capture Result ---------------------------
      MOV      AC0, T0

; Clear Arguments From the Stack -----------
      AADD      #5, SP

; Restore Registers and Return -------------
      POP      T3
      POP      T2
      ; POP   AR7
      ; POP   AR6
      POP      AR5

      RET
```

veneer 関数を以下に示します。各セグメントの説明に合わせて、いくつかの部分に分かれています。

例 7–2. アセンブリ関数 _firlat_veneer（続き）

(a) レジスタの保存

```
      PSH      AR5
      ; PSH   AR6       ; saved in ported C54x environment
      ; PSH   AR7       ; ditto
      PSH      T2
      PSH      T3
```

C55x 実行環境で、関数呼び出しによって特定のレジスタが変更されないことが想定される場合、これらのレジスタは保存する必要があります。C55x C コンパイラ環境の場合、レジスタ XAR5-XAR7、T2、および T3 は保存する必要があります。C54x コードは XAR*n* レジスタの上位ビットを変更できないため、下位ビットだけを確保する必要があります。C54x の移植されたコードの実行環境が（C54x C コンパイラによって定義されたとおり）これらのレジスタを保存すると思われるため、AR6 および AR7 を PUSH する命令がコメント化されます。さらにより確実なアプローチとしては、これらのレジスタを保存することです。

(b) 引数の引渡し

```
      PSH      T1       ; push right-most argument first
      PSH      T0       ; then the next argument
      PSH      AR3       ; and so on
      PSH      AR2
      PSH      AR1

      MOV      AR0, AC0    ; left-most argument goes in AC0
```

ネイティブの C55x コードから渡された引数は、移植された C54x コードの想定する場所に配置する必要があります。この場合、すべての引数はレジスタに渡されます。C55x C コンパイラの呼び出し規則に応じ、以下に示すレジスタに firlat() 関数への引数が渡され、結果が返されます。

```
T0           AR0        AR1        AR2        AR3
short firlat(short *x, short *k, short *r, short *dbuffer,
                 T0                 T1
            unsigned short nx, unsigned short nk);
```

C コンパイラの呼び出し規則についての詳細は、TMS320C55x オプティマイジング（最適化）C コンパイラ ユーザーズ・ガイドの「実行環境」の章を参照してください。

移植された C54x 環境では、最初の引数を A（C55x の AC0）に、残りの引数をスタックに、引数リストの表示順の逆に配置するよう想定されます。一番右の引数（T1）は最初にスタック上に PUSH されます。次の引数（T0）は、次にスタック上に PUSH されます。引数の配置は、一番左の引数（AR0）まで続きます。この引数は AC0 にコピーされます。

例 7–2. アセンブリ関数 _firlat_veneer（続き）

(c) ステータス・ビットの変更

```
      BSET   C54CM
      BCLR   ARMS
      BCLR   C16
```

ネイティブ C55x コードのステータス設定を、移植された C54x コードで必要な設定に変更する必要があります。この設定は、7.3.1 項「移植された C54x コードの実行環境」（7-10 ページ）に示しています。この場合、変更する必要があるのは C54CM および ARMS ビットだけです。

オリジナルの C54x コードの実行の仕様により、C16 ビットを 0 に設定しなければならない場合があります。このビットは、C55x コンパイル・コードには無視されますが、C54x コンパイラにより 0 になると想定されます。ビットを 0 に設定するのは、この想定に合わせるためのより確実なアプローチです。

(d) 関数呼び出し

```
      CALL   _firlat
```

これでレジスタが保存され、ステータス・ビットが設定されたため、移植された C54x コードへの呼び出しが行えます。

(e) ステータス・ビットの復元

```
      BCLR   C54CM
      BSET   ARMS
      BSET   SXMD
```

呼び出しの後、ステータスビットをネイティブの C55x 環境で必要となる設定に復元します。移植された C54x コードは、関数の呼び出しの後、SXMD ビット（C54x の SXM）について何も想定していません。しかし、C55x コンパイル・コードは、このビットが 1 に設定されると想定しています。

(f) 結果のキャプチャ

```
       MOV      AC0, T0
```

移植された C54x 環境では、AC0 に結果を返します。一方ネイティブの C55x 環境では、結果は T0 に返されると仮定しています。したがって、結果は AC0 から T0 にコピーしなければなりません。

例 7–2. アセンブリ関数 _firlat_veneer（続き）

(g) スタックから引数の消去

```
      AADD      #5, SP
```

この時点で、関数の渡した引数を PUSH するのに元々必要だったワード数分、スタックを減らさなければなりません。この場合は 5 ワードです。スタックは上位アドレスから下位アドレスに進むにつれて多くなるため、加算を使い、下位アドレスから上位アドレスにスタック・ポインタを変更します。

(h) レジスタの復元およびリターン

```
      POP       T3
      POP       T2
      ; POP    AR7
      ; POP    AR6
      POP       AR5

      RET
```

関数の最初に保存されたレジスタを復元し、リターンします。

7.3.8 C コードを呼び出す C54x アセンブリの例

この例は、コンパイルされた C ルーチンを呼び出す C54x アセンブリ・ルーチンです。C ルーチンは、C55x C コンパイラを使って再コンパイルされるため、アセンブリ・ルーチンは移植された C54x 実行環境と C55x コンパイラで使われる実行環境の違いを処理する必要があります。

C55x コードに異なる実行環境を使用する場合、コードの変更は、この例と若干異なります。しかし、ここで述べた点は考慮する必要があります。

例 7–3. 呼び出された C 関数のプロトタイプ

```
int C_func(int *buffer, int length);
   ...
```

アセンブリ関数は、この例には示されていない計算を行い、C 関数を呼び出します。返された結果は、`result` と呼ばれる C グローバル変数にコピーされます。その他の計算もここには示されていませんが、その後に行われます。

例 7–4. オリジナル C54x アセンブリ関数

```
; Declare some data   -----------------

          .data
buffer:    .word 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
BUFLEN       .set 11
          .text

; Assembly routine starts -------------

callsc:
; original C54x code ...

; Call C function (original C54x code) -------------

          ST #BUFLEN, *SP(0)  ; pass 2nd arg on stack
          CALLD #_C_func
          LD #buffer, A        ; pass 1st arg in A

; Effects of calling C:
;  May modify A, B, AR0, AR2-AR5, T, BRC
;  Will not modify AR1, AR6, AR7
;  May modify ASM, BRAF, C, OVA, OVB, SXM, TC
;  Will not modify other status bits
;  Presume CMPT = 0, CPL = 1

          STL A, *(_result) ; Result is in accumulator A

; original C54x code ...

          RET
```

C55x でこの関数を使うには、C 関数への呼び出しを変更する必要があります。

例 7–5. 変更されたアセンブリ関数

```
; declare data as shown previously

; Assembly routine starts -----------

callsc:
;  ported C54x code ...

; Call C function (Change to C55x compiler environment)

         AMOV #buffer,AR0  ; pass 1st ptr arg in AR0
         MOV #BUFLEN,T0    ; pass 1st int arg in T0
  ; compiler code needs C54CM=0, ARMS=1
         BCLR C54CM  ; clear C54x compatibility mode
         BSET ARMS   ; set AR mode
         BSET SXM     ; set sign extension mode
         CALL _C_func  ; no delayed call instruction

; Effects of calling C:
;  May modify AC0-AC3, XAR0-XAR4, T0-T1
;  May modify RPTC,CSR,BRCx,BRS1,RSAx,REAx
;  Will not modify XAR5-XAR7,T2-T3,RETA
;  May modify ACOV[0-3],CARRY,TC1,TC2,SATD,FRCT,ASM,
;   SATA,SMUL
;  Will not modify other status bits

         MOV T0, *(_result) ; Result is in T0

; could use *abs16(_result) if all globals are in the
; same 64K word page of data

; Change back to ported C54x environment ----------

         BSET C54CM  ; reset C54x compatibility mode
         BCLR ARMS   ; disable AR mode

; ported C54x code ...

         RET
```

呼び出し規則に応じて渡される引数についての詳細は、TMS320C55x C コンパイラの最適化入門マニュアルの「実行環境」の章を参照してください。変更されたステータス・ビットは、C54x の移植された実行環境とネイティブの C55x 環境（この場合 C55x C コンパイラによって定義されたとおり）で異なるものだけです。

C を呼び出す働きについてのコメント（レジスタおよびステータス・ビットが変更できるかできないか）は、表示されるコードに影響を与えません。しかし、これらの働きは呼び出しなどのコードに影響を与える場合があります。

たとえば、XAR1 レジスタを見てみましょう。C54x コンパイラ環境では、AR1 は呼び出しによって変更されません。C55x コンパイラ環境では、XAR1 は変更される場合があります。C_func への呼び出し前のコードが AR1 に値をロードし、呼び出し後にコードがその値のために AR1 をリードする場合、コードは書かれたとおり、C55x で機能しません。代替案として、XAR5 など、C ルーチンが保存している XAR*n* レジスタを使う方法が最適です。

7.4 C55x ソースの出力

この節では、C54x ソース・コードを オブジェクト・コードではなく C55x ソース・コードに直接変換する方法を説明します。この変換では、同じフォーマット、間隔、コメント、および（しばしば使われる）オリジナル・ソースのシンボリック参照を使うことにより、C54x アセンブリ・コードへの投資は保持されます。

7.4.1 コマンドライン・オプション

表 7–5 に、cl55 コマンドライン・オプションを示します。

表 7–5. cl55 コマンドライン・オプション

オプション	意味
--mnem	ニーモニック出力
--alg	代数表記出力
--incl	インクルード・ファイルを出力に書き込む
--nomacx	マクロを展開しない
-fr *dir*	出力ファイルにディレクトリ名を付ける
-eo *.ext*	出力ファイルの拡張子に名前を付ける

ソース出力を取得するには、--mnem または --alg を使う必要があります。使わない場合、通常のオブジェクト・ファイルを作成します。この節では、ほとんどの例で *cl55 --mnem* が使用されていますが、*cl55 --alg* も使用することができます。

出力ファイルの拡張子を指定しない場合、出力ファイル名は対応する入力ファイルと同じになり、別の拡張子が付きます。入力ファイル拡張子の最初の文字が「a」または「s」の場合、または拡張子がない場合、出力ファイル拡張子は *.s55* になります。そうでない場合、ファイルはインクルード・ファイルと想定され、出力ファイル拡張子は *.i55* になります。

--incl についての詳細は、7.4.2 項および 7.4.3 項を参照してください。--nomacx についての詳細は、7.4.7 項「マクロの処理」を参照してください。

次に例を示します。

```
cl55 -q --mnem --incl -fr c55x_asm -eo .asm *.asm
```

すべてのアセンブリ・ファイルが処理され、ディレクトリ *c55x_asm* に出力が格納され、拡張子にデフォルトの .s55 でなく .asm が付けられたことを示しています。作成されたインクルード・ファイルはどれも c55x_asm の中にありますが、上述したデフォルトの出力ファイル名規定に応じて名前が付けられます。

注 : -fr および -eo オプション

cl55 を使ってファイルをオブジェクトにコンパイルまたはアセンブルする際、-fr および -eo オプションは、.obj ファイルに対して同じ意味を持ちます。

オブジェクト・ファイルが作成されていないため、通常アセンブラに影響を与える一部のコンパイラ・オプションは、適用されません。たとえば、-al はリスト・ファイルを作成しません。

表 7–6 に、アセンブラに影響を与えるコンパイラ・オプションのリストを示します。

表 7–6. アセンブラに影響を与えるコンパイラ・オプション

オプション	意味	影響
-aa	絶対リスト出力の有効	なし
-ac	大文字と小文字を区別しない	あり
-ad*name*	*name* の事前定義	あり
-ahc<f>	.copy file f	あり
-ahi<f>	.include file f	あり
-al	（L の小文字）asm リスト・ファイルの作成	なし
-ar[#]	注釈の抑止 [#]	あり
-as	ローカル・シンボルの保持	なし
-ata	ARMS 有効と初期設定されていることの表明	なし
-atc	CPL 有効と初期設定されていることの表明	なし
-ath	速度優先の移植	あり
-atl	C54x 有効と初期設定されていることの表明	なし
-atn	遅延スロットからの NOP の除去	あり
-atp	プロファイル .prf ファイルの作成	なし
-att	SST は常時ゼロであるという表明	あり
-atv	すべての分岐／呼び出しは 24 ビット・オフセットとしてエンコード	なし
-atw	すべての警告の抑止	あり
-au*name*	*name* の未定義	あり
-ax	クロスリファレンス・ファイルの作成	なし

影響ないとリストされたオプションを使うと、ただ無視されます。

7.4.2 .include ファイルおよび .copy ファイルの処理

この節で、インクルード・ファイルという用語は .include または .copy 疑似命令のどちらかによってインクルードされたファイルを意味します。インクルードしているファイルを正しく処理するためには、インクルード・ファイルを読み出す必要があります。この節では、インクルード・ファイルが対応する出力ファイルに含まれるかどうかについて説明します。

デフォルトでは、出力ファイルにインクルード・ファイルは書き出されず、.include 文自体は変更されずに残ります。新しい .s55 ファイルが再アセンブルされると、それには処理されていないファイルが含まれます。アセンブラが C54x の構文を読み出せるため、このことは正確性には影響しません。

--incl オプションはこの動作を変更します。--incl を使うと、出力ファイルに各インクルード・ファイルが書き出されます。新しいファイルの名前は、使用しているコマンドライン・オプションと指定する拡張子によって決められます。拡張子は、7.4.1 項「コマンドライン・オプション」に説明されているとおりです。さらに、.include 文は変更され、新しい出力ファイルを含みます。

たとえば、--incl を使うと、

```
.include    i1.inc
```

上記により、il.i55 が作成され、この文は以下のように変更されます。

```
.include    i1.i55
```

インクルード・ファイルの名前を付ける際、1 つ特殊なケースがあります。-fr <dir> オプションを使って cl55 が起動され、インクルード・ファイルの名前にディレクトリ情報が入らない場合、新しいインクルード・ファイルが -fr オプションによって与えられたディレクトリに書き出されます。

たとえば、次のようになります。

```
cl55 --mnem --incl input.asm -fr outdir
```

これにより、新しい il.i55（および input.i55）ファイルがディレクトリ *outdir* に格納されます。

7.4.3 --incl オプションを使う場合の問題点

以下の 3 つのファイルで例を見てみましょう。

```
;   i1.inc----------------------
      .word   x
;   file1.asm-------------------
x       .set   0
      .include   i1.inc
;   file2.asm-------------------
x       .set   1
      .include   i1.inc
```

file1.asm および file2.asm の両方をビルド時、-incl を使用するとします。新しい il.i55 に *.word 0* または *.word 1* が含まれるかどうかは、どちらのファイルが最後に作成されるかによって決まります（どちらか一方は必ず間違いです）。

-incl オプションは、作成されるそれぞれのインクルード・ファイルにコンテキストがない場合に限り動作します。つまり、それを含むファイルに対する依存関係はありません。この場合、*i1.inc* は、fail1.asm および file2.asm の両方で定義されたとおり、異なる x 値に依存しています。

i1.inc が複数の異なるファイルにインクルードされている場合、--incl を使うと cl55 --mnem が該当するファイルで起動されるたびに il.i55 が書き出されます。複数の開発者が同じディレクトリ上の異なるファイルで作業をしている場合、*cl55 --mnem --incl* が実行されるたびに新しい *i1.i55* ファイルが作成されていることに注意する必要があります。

--incl を使う場合のもう一つの問題点は、パラレル・メイクに関連しています（パラレル・メイクについて分からない場合は、この段落は飛ばして読んで構いません）。*i1.inc* を持つ、一連の .asm ファイルがあるとします。さらに、*cl55 --mnem* を使ってこれらのファイルを C55x 構文に変更し、結果である .s55 ファイルを *cl55* だけを使ってオブジェクトにアセンブルするメイクファイルがあるものとします。この動作を並行して行うと、il.i55 への同時書き込みおよび読み出しで終了します。これは機能しないため、シリアル・メイクを使って .i55 ファイルを作成しなければなりません。

7.4.4 .asg および .set の処理

.asg および .set 行は、変更されずにコピーされます。.asg および .set が定義するシンボルの使用は、大部分が保持されます。一般的に、古い C54x 命令から新しい C55x 命令へオペランド全体を変更せずにコピーできる場合、そのオペランドはすべてコピーされます。しかし、そのオペランドが何らかの方法で変更される場合、シンボリック参照は表示されない場合があります。

7.4.5 .tab 疑似命令を使ったスペーシング幅の確保

アセンブラは、オリジナルのソース行のスペーシング幅を、ソース・ファイルからコピーすることによって確保します。しかし、C55x ニーモニックまたはオペランドのフィールド幅がオリジナルより広い場合、一部のオリジナルのスペーシング幅は省略されます。この操作を正確に処理するには、タブが使用するスペーシング幅をアセンブラが認識している必要があります。デフォルトは 8 です。このデフォルト値は、**.tab** *number* の疑似命令を使って変更することができます。この *number* とは、システム内でタブが使用するスペーシング幅です。

7.4.6 アセンブラ生成コメント

ソース行が削除または追加されると必ず、アセンブラは特別な接頭部または接尾部のコメントを使ってこの行にマークを付けます。これらの行は、通常テキスト・エディタや Perl および　awk などのスクリプト言語に見られる検索機能を使って簡単に見つけることができます。

コメントの一般的な形式は ";+XX" です。XX は 2 文字のコードで、コメントが実行する関数を指定するために使われます。これらについては、次の節で説明します。

7.4.6.1 複数行リライト

複数行リライトは、コメント・アウトされたオリジナル・ソース行の前に表示されます。これらは、オリジナル・ソース行に関連した各行の終わりにも付け加えられます。

2 種類の複数行リライトは以下のとおりです。

- ML — 複数行リライト
- RL — 一時レジスタを使う複数行リライト

7.4.6.2 展開されたマクロの呼び出し

展開されたマクロの呼び出しは、マクロ展開内のマクロの呼び出し（常に MI）または複数行リライトの前に表示されます。これらは、マクロ展開内の各行の終わりにも付け加えられます。

3 種類の展開されたマクロの呼び出しは以下のとおりです。

- ❏ MI — 展開されたマクロ内の単一行リライト
- ❏ MM — 展開されたマクロ内の複数行リライト
- ❏ RM — 展開されたマクロ内の一時レジスタを使う複数行リライト

7.4.6.3 接頭部コメント

以下のコメントは、コメント・アウトされた行の前に表示されます。

- ❏ NP — 削除された NOP
- ❏ IF — .if/.endif 疑似命令および関連する疑似命令。関連する false のブロックと同様。
- ❏ LP — .loop/.break/.endloop および囲んでいる行
- ❏ FN — ファイルの命名
- ❏ MS — その他

7.4.6.4 接尾部コメント

以下のコメントは、変更された行の終わりに付け加えられます。

- ❏ SA — .asg に変更された .set（代数表記出力においてのみ）

以下のコメントは、追加された行の終わりに付け加えられます。

- ❏ RK — .mmsg 疑似命令を使って挿入された注釈

7.4.6.5 アセンブラ生成コメント用のコード例

例 7–6 に、アセンブラ生成コメント用のコード例を示します。例 7–7 に、例 7–6 の C55x 出力を表示します。

例 7–6. オリジナル C54x アセンブリ・ファイル

```
      .global name

      ADD      *AR2, A          ; same mnemonic
      LD       *AR3, B          ; different mnemonic

      RPT      #10
      MVDK     *AR4+, name      ; multi-line rewrite

subm  .macro   mem1, mem2, reg      ; macro definition
                                    ; copied through
      LD       mem1, reg
      SUB      mem2, reg
      endm

      subm     name, *AR6, B       ; macro invocation
                                   ; expanded

      MOV      T1, AC3          ; native LEAD3 instruction
```

例 7–7. 例 7–6 における C54x コード例の C55x 出力

```
;+MS translation of try1.asm
;+MS Temporary Registers Used:XCDP
      trans_count 1            ;+MS do NOT remove!
      global      name

      ADD      *AR2, AC0, AC0          ; same mnemonic
      MOV      *AR3, AC1              ; different mnemonic
;+RL  MVDK     *AR4+, name     ; multi-line rewrite
      AMAR     *(#name), XCDP  ; +RL port of MVDK *AR4+, name

      RPT      #10
      MOV      *AR4+, *CDP+    ; +RL port of MVDK *AR4+, name

subm  .macro   mem1, mem2, reg      ; macro definition
                                    ; copied through

      LD       mem1, reg
      SUB      mem2, reg
      .endm

;+MI  subm     name, *AR6, B         ; macro invocation
                                     ; expanded

      MOV      @#name, AC1    ;+MI
      SUB      *AR6, AC1, AC1 ;+MI

      MOV      T1, AC3        ; native LEAD3 instruction
```

7.4.7 マクロの処理

マクロ定義は常に変更されずにコピーされます。

デフォルトでは、マクロの呼び出しは展開されます。オプション *--nomacx* を使用するとこの展開を無効にできます。

--alg --nomacx を組み合わせると、出力ファイルは、ファイルのニーモニック構文を使うマクロを呼び出します。このファイルは、その他の場合は代数表記構文を使います。従って、このようなファイルの上部には以下のエラー・メッセージが見られます。

例 7–8. --alg & --nomacx を組み合わせて作成する C55x 出力

```
.emsg "This file will not assemble because it combines algebraic syntax with
     invocations of macros in mnemonic syntax.Please see the comment at
     the top of <output file> for more information."

.end ;  stops assembler processing

Also note that because this file cannot be assembled, this combination of
features cannot be tested.To attempt to assemble this file you must rewrite
the macros in C55x algebraic syntax, and remove this .emsg, .end, and
associated comment block.
```

このファイルをアセンブルする場合、.emsg にあるエラー・メッセージが表示されます。メッセージが表示された後、アセンブラは実行を停止します。続行するには、エラー・メッセージで指示されたとおりにファイルを編集する必要があります。

7.4.8 .if および .loop 疑似命令の処理

.if および .loop 疑似命令により制御されるコード・ブロックを使った場合の問題点は、変換時にこれらのブロックが必ずしも一緒ではないということです。遅延分岐や呼び出しは、変換の一部として動作しなければなりません。これらの遅延分岐や呼び出しが上記のようなブロックの直前で発生した場合、これらはブロック内に移動します。終了近くにある場合には、ブロックを出る場合もあります。

アセンブラは条件式を演算し、false のブロック内のコードと同様に .if、.else、.elseif、および .endif 疑似命令をコメント・アウトします。.loop についての解決策は、.loop から .endloop にすべてをコメント・アウトすることです。これには、.break 疑似命令も含まれます。続いて .loop ブロックを必要な回数だけ反復することです。

注 : ループ・カウントは変換されたソース・サイズに影響します。
ループ・カウントの値が大きい場合、オリジナルのソースに対して変換されたソースのサイズが大きくなります。

7.4.9 Code Composer Studio への統合

C54x から C55x へのソース変換は、Code Composer Studio（CCStudio）内で統合された処理ではありません。この節で説明したコマンド行オプションは、いずれも CCStudio 内からは利用できません。cl55 へのコマンド行インタフェースを使って C54x ソースを C55x に変換した後、新しい C55x ソース・ファイルを C55x CCStudio プロジェクトに追加します。

7.5 移植できない C54x コーディングの例

一部の C54x コーディングの例は、C55x に移植できません。アセンブラは検出可能な点については警告しますが、すべてを検出することはできません。以下に示すコーディングの例は、移植できません。

- メモリ・アドレスとして使用される定数。次に例を示します。

```
B 42
ADD @42,A
SUB @symbol+10,b
```

- 後にコード・アドレスとして解釈される定数を使って初期化されるメモリ。次に例を示します。

```
table:.word  10, 20, 30
...
LD      @table,A
CALA
```

- 命令として使われるデータ。次に例を示します。

```
function:
      .word 0xabcd  ; opcode for ???
      .word 0xdef0  ; opcode for ???
...
      CALL function
```

- out-of -order 実行、別名パイプラインのトリック。アセンブラは out-of -order 実行のインスタンスを検出します。ある命令が C54x XC 命令の 2 命令ワード前の条件を変更した場合です。この場合、アセンブラは注釈を発行します。その他の out-of -order 実行の場合は、アセンブラによって検出されません。

- コードを作成または変更するコード。

- 複数のファイルにまたがるリピート・ブロック。

- ラベルを定義されていない、分岐または呼び出しの位置。または、ラベルが定義されていない位置に返すリターン・アドレスの変更。これには以下のような命令が含まれます。

```
B $+10
```

- データではなく命令にアクセスするために使用する READA および WRITA 命令。詳細は、7.6.1 項「プログラム・メモリ・アクセスの処理」(7-33 ページ) を参照してください。

- 上位ビットがゼロでないアキュムレータを使った READA または WRITA の使用。

  C54x デバイス（'C548 以降でないもの）の READA/WRITA 命令は、アキュムレータの下位 16 ビットを使い、上位 16 ビットは無視します。しかし 'C548 以降のデバイスは、下位 23 ビットを使います。アセンブラは、コードが指定されているデバイスを簡単に知ることはできません。それは 'C548 以降と考えられます。したがって、'C548 以降のデバイスのコードは、問題なくマッピングされます。これら以外のデバイスのコードは実行されません。

- 条件付きアセンブリ表現では、ラベルの違いは許されません。

7.6 C54x についての追加事項

このセクションでは、システムの追加事項を説明します。

C54x コードが以下のいずれかを行う場合、このコードを変更し、ネイティブの C55x 命令を使う必要があるかもしれません。

- ❏ LDM のような MMR 命令の MMR スロットで *SP（オフセット）オペランドを使用。
- ❏ コードのブロックをコピー。通常はオフチップ・メモリからオンチップメモリへ。
- ❏ 周辺機器に対し、メモリマッピングされたアクセスの使用。
- ❏ C55x へのマッピングの後、32K より大きなリピート・ブロックの使用。
- ❏ 分岐条件 BIO/NBIO の使用。

以下の点についても注意する必要があります。

- ❏ C54x の C5x 互換機能は、C55x でサポートされていない。
- ❏ C54x で割り込み不能な RPT 命令は、C55x で割り込むことができる。
- ❏ 式がガード・ビットにオーバーフローし、左シフトがガード・ビットを消去した際、C54x の値がゼロで、C55x の値が飽和している。
- ❏ C54x および C55x のニーモニック・アセンブリ言語が、命令の並列表記の点で著しく異なっている。

  C55x は、2 種類の並列表記、1 命令であらかじめ定義された並列表記（:: 演算子を使用）および 2 つの命令間におけるユーザー定義の並列表記（|| 演算子を使用）を実装しています。C54x は、1 種類の並列表記しか示しません。それは、C55x であらかじめ定義された並列表記に似ています。しかし、C54x 並列表記は、演算子として並行の棒（||）を使っています。C55x 並列表記は、TMS320C55x DSP ニーモニック命令設定リファレンス・ガイドで説明されています。

- 以下のようなメモリ・マップド・アクセス命令を使って間接アクセスを使用する際、

```
STM #0x1234, *AR2+
```

C54x は、AR*n* レジスタの上位 9 ビットをマスキングします。このマスキングは、AR2 へのポスト-インクリメントの前後双方に影響があります。次に例を示します。

```
; AR2 = 0x127f
STM #0x1234, *AR2+ ; access location 0x7f
; AR2 = (0x7f + 1) & ~7f ==> 0
```

しかし、C55x アセンブラは次のようにマッピングします。

```
AND #0x7f, AR2
MOV #0x1234, *AR2+  ; note no masking afterward
```

そして AR2 のメモリ・マップド・アドレスの可能性を報告します。

7.6.1 プログラム・メモリ・アクセスの処理

cl55 アセンブラは、コードでなくデータにアクセスするために C54x プログラム・メモリ・アクセス命令（FIRS、MACD、MACP、MVDP、MVPD、READA、WRITA）をサポートします。アセンブラがこれらの命令の 1 つを検出すると、注釈（R5017）を発行します。C54x ではコード・アドレスはワード単位ですが、C55x ではバイト単位です。プログラム・メモリ・アクセス命令を処理する際、この違いに対応するために、アセンブラは以下の動作を行います。

- C54x プログラム・メモリ・アクセス・オペランドがコード（バイト）アドレスでなくデータ（ワード）アドレスを示していると想定し、C55x 命令シーケンスを生成します。
- コード・セクションで見つけたデータ宣言はすべて、そのデータ・セクションに配置します。そのためには、リンカ・コマンド・ファイルへの変更が最も必要になってきます。

たとえば、次の C54x 入力は、

```
      .global ext
      MVDP   *AR2, ext
table:
      .word   10
```

cl55 によって以下のように移植されます。

```
      .global ext
      AMOV   #ext, XCDP
      MOV      *AR2, *CDP
      .sect   ".data:.text"
table:
      .word   10
```

この例では、MVDP のために生成される命令は、ext がデータ（ワード）アドレスであると想定しています。コード内で使われるメモリ・アドレスが実際にコード・アドレスである場合、C55x 命令は機能しません。この場合、関数を書き直して、ネイティブの C55x 命令を使う必要があります。移植された C54x コードとともにネイティブの C55x 命令を使う方法についての詳細は、7.3 節「ネイティブ C55x 関数と移植された C54x 関数を混在して使用する方法」（7-10 ページ）を参照してください。

この例の .word 疑似命令は、.data:.text と呼ばれる新しいセクションに配置されます。一般的に、1 つのコード・セクション内にあるデータのグループ化は、.data:*root_section* の名前が付いたサブセクションに配置されます。ここでは、*root_section* は C54x で使用されるオリジナルのコード・セクションの名前です。この変更に対応するため、リンカ・コマンド・ファイルを変更する必要があります。サブセクションは個別に割り当てることも、同じベース・セクション名をもつ他のセクションとグループ化して割り当てることもできます。たとえば、すべてのデータ・セクションとサブセクションをグループ化するには、

```
.data > RAM  ; allocates all .data sections / subsections
```

サブセクションについての詳細は、2.2.4 項「サブセクション」（2-8 ページ）を参照してください。

7.7 アセンブラ・メッセージ

C54x コードのアセンブル時、cl55 は以下の注釈のいずれかを生成する場合があります。特定の注釈またはすべての注釈を抑止するには、-r アセンブラ・オプションまたは .noremark 疑似命令を使います。詳細は、4-78 ページの .noremark についての説明を参照してください。

(R5001) Possible dependence in delay slot of RPTBD--be sure delay instructions do not modify repeat control registers.

説明 このメッセージは、C54x RPTBD 命令の遅延スロットの命令が、間接メモリ参照を実行する場合に発生します。

動作 これらの命令により REA または RSA リピート・アドレス制御レジスタが変更する場合、RPTBD をインプリメントするために使われる C55x 命令は機能しません。この命令が REA または RSA を変更しない場合、このメッセージを無視しても、コードを書き直して RTPB を使っても構いません。

(R5002) Ignoring RSBX CMPT instruction

説明 この C54x 命令は、C54x の 'C5x 互換モードを無効化します。C55x は 'C5x 互換モードをサポートしないため、この命令は無視されます。

動作 この命令をコードから削除するか、単にこのメッセージを無視します。

(R5003) C54x does not modify AR*n*, but C55x does

説明 このメッセージは、ADD または SUB 命令の両方のメモリ・オペランドが同じ AR*n* レジスタを使い、2 番目のオペランドだけがレジスタを変更する場合に発生します。次に例を示します。

```
SUB  *AR3, *AR3+, A
```

動作 C54x では、このような命令は 1 つ追加して AR3 を変更することはありません。C55x では、同じ命令が AR3 に 1 つ追加します。この動作の違いは、コードに影響を与える場合と与えない場合があります。このメッセージが発行されるのを防ぐためには、AR*n* の変更を最初のオペランドに移動します。

```
SUB  *AR3+, *AR3, A
```

(R5004) Port of RETF correct only for non-interrupt routine.

説明 このメッセージは、アセンブラが RETF および RETFD を検出し、C54x 高速割り込みが命令を返すと発生します。これらの命令を非割り込みルーチンで適切に使うことができるため、RETF 命令は C55x RET 命令にマッピングされます。

動作 RETF または RETFD のこのインスタンスを実際に使う場合、R5005 に記載されている点を考慮する必要があります。そして C55x RDTI 命令を使ってこの命令を書き直します。

(R5005) Port of [F]RETE is probably not correct.Consider rewriting to use RETI instead.

説明 このメッセージは、アセンブラが C54x RETE、RETFD、FRETE および FRETED 命令を検出すると発生します。これらの命令は C55x RETI 命令にマッピングされます。

動作 RETI 命令の結果は、RETE 命令の結果と異なります。たとえば、RETI は ST1_55、ST2_55 および ST0_55 の一部を自動的に復元します。RETE は復元しません。それに応じてコードを調整する必要があります。さらに、C55x 一時レジスタを使い、C54x 割り込みサービス・ルーチンに複数行マッピングを含めるかどうかを決定する必要があります。含める場合、レジスタを確保する必要があります。詳細は、7.1.2 項「割り込みサービス・ルーチンの処理」（7-3 ページ）を参照してください。

(R5006) This instruction loads the memory address itself, and not the contents at that memory address

説明 このメッセージは、AMOV 命令の最初のオペランドがオペランド接頭部のないシンボルである場合に発生します。次に例を示します。

```
AMOV symbol, XAR3     ; not written as #symbol
```

動作 この命令は、*symbol* の表示するメモリ・アドレスに内容をロードするように見える場合があります。しかし、シンボルのアドレス自体がロードされます。# 接頭部を使い、この点を修正します。

```
AMOV #symbol, XAR3
```

(R5007) C54x and C55x port numbers are different

説明 このメッセージは、アセンブラが C54x PORTR および PORTW 命令を検出すると発生します。C55x 命令シーケンスがエンコードされ、同じ関数を実行します。しかし使用されるポート番号は、C55x には不適切である場合が大半です。

動作 コードを書き直し、ポート・アドレスの内容をレジスタにロードまたは格納する、類似の C55x 命令の使用を考えます。

```
MOV port(#100), AC0   ; for PORTR
MOV AC1, port(#200)   ; for PORTW
```

(R5008) C54x directive ignored

説明 一部の C54x アセンブラ疑似命令は、C55x では必要ありません。このメッセージは、このような疑似命令（.version、c_mode、.far_mode）を使う場合に発生します。

動作 この疑似命令をコードから削除するか、単にこのメッセージを無視します。

(R5009) Modifying C54x IPTR in PMST will not update C55x IVPD/IVPH.Replace with native C55x mnemonic (e.g., MOV #K, mmap(IVPD)).

説明 このメッセージは、アセンブラが PMST レジスタへの書き込みを検出すると発生します。C54x では、PMST のビット 15 から 7 に、割り込みベクトル・テーブルのアドレスのうち上位 9 ビットが含まれます。C55x は、この役割に IVPD/IVPH レジスタを使います。IVPD/IVPH についての詳細は、TMS320C55x DSP CPU リファレンス・ガイドを参照してください。

動作 C54x 命令をネイティブの C55x 命令と置き換えます。

(R5010) C54x and C55x interrupt enable/flag registers and bit mapping are different.Replace with native C55x mnemonic.

説明 このメッセージは、アセンブラが IFR または IMR レジスタへの書き込みを検出すると発生します。C55x IFR および IER（C54x の IMR）レジスタのビット・マッピングは、C54x のマッピングと異なります。レジスタについての詳細は、TMS320C55x DSP CPU リファレンス・ガイドを参照してください。

動作 C54x 命令をネイティブの C55x 命令と置き換えます。

(R5011) C55x requires setting up the system stack pointer (SSP) along with the usual C54x SP setup.

説明 このメッセージは、アセンブラが SP レジスタへの書き込みを検出すると発生します。C55x では、プライマリ・システム・スタックが SP によって管理され、二次システム・スタックは SSP によって管理されます。この注釈は、SP が初期化されると必ず SSP も初期化されなければならないということの注意です。

動作 SSP レジスタを初期化してください。

(R5012) This instruction requires the use of C55x 32-bit stack mode.

説明 このメッセージは、アセンブラが FCALL[D] または FCALA[D] 命令を検出すると発生します。これらの命令は、32 ビット・スタック・モードでのみ機能します。スタックの設定についての詳細は、TMS320C55x DSP CPU リファレンス・ガイドを参照してください。32 ビット・スタック・モードは、デバイス・リセットにおけるデフォルトのモードです。異なるスタック・モードを使うには、リセット・ベクターを明確に設定する必要があります。詳細は、4-64 ページの .ivec 疑似命令についての説明を参照してください。

動作 必要に応じてスタック設定を行います。

(R5013) C55x peripheral registers are in I/O space.Use C55x port() qualifier.

説明 このメッセージは、アセンブラが C54x ペリフェラル・レジスタ名の使用を検出すると発生します。これらのレジスタは、C55x にメモリマッピングされていません。その代わり、I/O スペースに配置しています。C55x I/O スペースにアクセスするには、port() オペランド修飾子を使う必要があります。詳細については、TMS320C55x DSP ニーモニック命令設定リファレンス・ガイドを参照してください。

動作 必要に応じて port() 修飾子を使います。

(R5014) On C54x, the condition set in the two instruction words before an XC does not affect that XC.The opposite is true on C55x.

説明 このメッセージは、アセンブラが C54x XC 命令の 2 命令ワード前の条件を変更する命令を検出した場合に発生します。C54x では、このコードはパイプラインの out-of -order 実行に依存します。しかし、この out-of -order 実行は C55x では発生しないため、結果は同じではありません。out-of -order 実行は、7.5 節（7-30 ページ）にあるとおり、移植できない C54x コーディングの例と考えられます。out-of -order 実行の場合が多くありますが、これはアセンブラによって検出されるものだけです。

動作 C55x における違いに対応するため、コードを変更します。

(R5015) Using hard-coded address for branch/call destination is not portable from C54x.

説明 このメッセージは、アセンブラが非シンボリック、つまり数値表記のアドレスへの分岐または呼び出しを含む C54x 命令を検出した場合に発生します。コード・アドレスは、C54x ではワード、C55x ではバイトのため、アセンブラはアドレスがバイトとワードの違いに対応するかどうか認識できません。

動作 C55x における違いに対応するため、コードを変更します。

(R5016) Using expression for branch/call destination is not portable from C54x.

説明 このメッセージは、アセンブラが算術演算子（sym+1 など）を含む式を使って C54x 分岐または呼び出し命令を検出する場合に発生します。コード・アドレスは、C54x ではワード、C55x ではバイトのため、アセンブラはコードがバイトとワードの違いに対応するかどうか認識できません。

動作 C55x における違いに対応するため、コードを変更します。

(R5017) Program memory access is supported when accessing data, but not when accessing code.In addition, changes to your linker command file are typically required.

説明　このメッセージは、アセンブラが C54x プログラム・メモリ・アクセス命令（FIRS、MACD、MACP、MVDP、MVPD、READA、WRITA）を検出すると発生します。詳細は、7.6.1 項（7-33 ページ）を参照してください。

動作　C55x における違いに対応するため、コードおよびリンカ・コマンド・ファイルを変更します。

❑ 1 命令内のビルトイン並列表記

一部の命令は、2 つの異なる演算を並行して実行します。二重のコロン（::）を使って 2 つの演算を分けます。この種の並列表記は、あらかじめ定義された並列表記とも呼ばれます。これらの命令は、デバイスによって直接提供されます。TMS320C55x DSP ニーモニック命令設定リファレンス・ガイドで説明されています。自分であらかじめ定義された並列表記を作成することはできません。

❑ 2 つの独立した命令間のユーザー定義並列表記

TMS320C55x DSP ニーモニック命令設定リファレンス・ガイドで説明されている、並列表記規則で許容されているように、2 つの命令を自分で並列にすることもできます。平行の棒（||）を使って 2 つの命令を分け、並行して実行させます。

C54x は、1 種類の並列表記だけを示しています。C55x であらかじめ定義された並列表記に似ています。しかし、C54x 並列表記は、演算子として並行の棒（||）を使っています。

以下の表に、C54x および C55x の並列表記演算子をまとめました。

並列表記の種類	C54x 演算子	C55x 演算子
あらかじめ定義	‖	::
ユーザー定義	該当なし	‖

第 8 章

リンカの説明

TMS320C55x™ リンカは、複数の COFF オブジェクト・ファイルを結合して実行可能モジュールを作成します。COFF セクションの概念は、リンカの動作の基本になっています。第 2 章「共通オブジェクト・ファイル・フォーマットの概要」に COFF フォーマットについて詳しく説明します。

項目 **ページ**

8.1 リンカの概要

TMS320C55x リンカを使用すると、出力セクションを効率良くメモリ・マップに割り当てることによりシステム・メモリを構成できます。リンカは、オブジェクト・ファイルを結合するときに次の作業を実行します。

- ❏ 入力ファイル間の未定義の外部参照を解決する。
- ❏ セクションをターゲット・システムの構成メモリに配置する。
- ❏ シンボルとセクションを再配置して、最終アドレスに割り当てる。

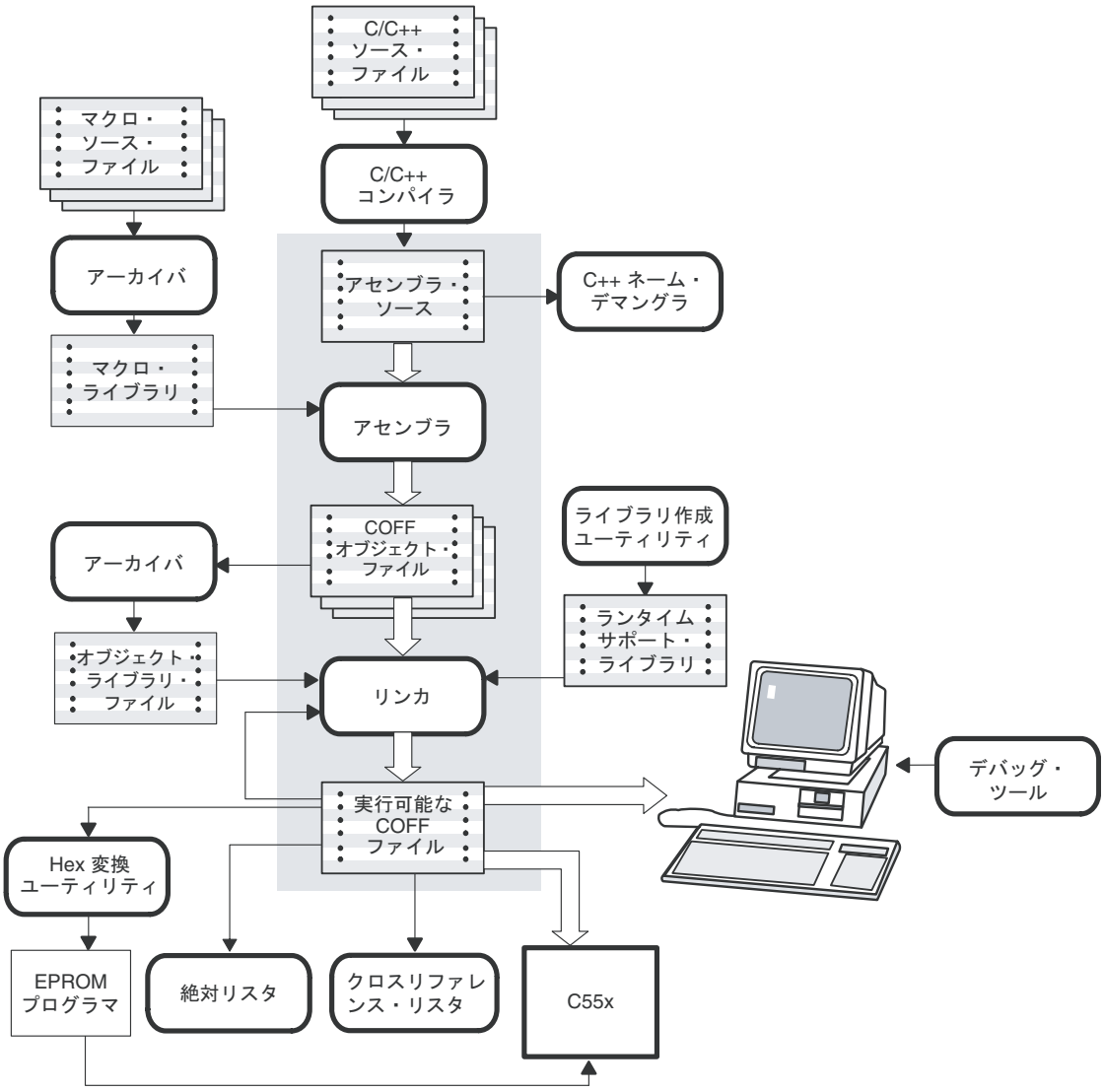
リンカ・コマンド言語は、メモリ構成、出力セクション定義、およびアドレスのバインディングを制御します。この言語は、式の代入と計算をサポートします。設計したメモリ・モデルを定義し作成することにより、システム・メモリを構成できます。2 つの強力な疑似命令である MEMORY および SECTIONS 疑似命令を使用すると、次の操作を行うことができます。

- ❏ セクションをメモリの特定の領域に配置する。
- ❏ オブジェクト・ファイルのセクションを結合する。
- ❏ リンク時にグローバル・シンボルを定義または再定義する。

8.2 リンカ開発のフロー

図 8-1 は、アセンブリ言語開発プロセスにおけるリンカの役割を示したものです。リンカは、オブジェクト・ファイル、コマンド・ファイル、ライブラリ、部分的にリンクされたファイルなど数種類のファイルを入力として受け入れます。リンカが作成する実行可能な COFF オブジェクト・モジュールは、いくつかの開発ツールの 1 つにダウンロードでき、または TMS320C55x デバイスで実行できます。

図 8-1. リンカ開発のフロー



8.3 リンカの起動

リンカを起動する一般的な構文は次のとおりです。

cl55 -z [*-options*] *filename*$_1$*.... filename*$_n$

cl55 -z	リンカを起動するコマンドです。
options	コマンド行またはリンカ・コマンド・ファイル内の任意の場所に指定できます（オプションについては、8.4 節「リンカ・オプション」（8-5 ページ）のを参照してください）。
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・ライブラリが入ります。すべての入力ファイルのデフォルト拡張子は *.obj* であり、それ以外の拡張子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・ファイルか、リンカ・コマンドを含む ASCII ファイルかを判断します。デフォルトの出力ファイル名は *a.out* です。

リンカを起動するには、次の 2 つの方法のいずれかを使用します。

- コマンド行でオプションとファイル名を指定します。この例では、file1.obj と file2.obj の 2 つのファイルをリンクして link.out という名前の出力モジュールを作成します。

  ```
  cl55 -z file1.obj file2.obj -o link.out
  ```

- ファイル名とオプションをリンカ・コマンド・ファイルに入れます。リンカ・コマンド・ファイル内に指定されたファイル名は、先頭を文字で始める必要があります。たとえば、ファイル linker.cmd に次の行があるとします。

  ```
  -o link.out
  file1.obj
  file2.obj
  ```

  これで、コマンド行からリンカを起動できます。コマンド・ファイル名を入力ファイルとして指定してください。

  ```
  cl55 -z linker.cmd
  ```

  コマンド・ファイルを使用するときには、コマンド行で別のオプションとファイルを指定することもできます。たとえば、次のように入力できます。

  ```
  cl55 -z -m link.map linker.cmd file3.obj
  ```

  リンカは、コマンド行でコマンド・ファイルを検出すると直ちにそのファイルを読み取って処理します。したがって、ファイルは file1.obj、file2.obj、file3.obj の順序でリンクされます。この例では、link.out という出力ファイルと link.map というマップ・ファイルが作成されます。

C/C++ ファイルでリンカを起動する方法の詳細は、8.19 節「C/C++ コードのリンク」（8-93 ページ）を参照してください。

8.4 リンカ・オプション

リンカ・オプションはリンク操作を制御します。オプションで、コマンド行かコマンド・ファイルに置くことができます。リンカ・オプションの前には必ずハイフン（-）を付けなければなりません。-l（小文字の L）と -i オプションを除き、オプションはどのような順序で指定してもかまいません。リンカ・オプションの要約を次に示します。

-a 絶対的な実行可能モジュールを作成します。このオプションはデフォルトです。-a も -r も指定されていない場合、リンカは -a が指定されているものとして動作します。

-abs 絶対リスト・ファイルを作成します。リンカ・コマンド・ファイルで -O オプションを使用している場合でも、-O オプション（-z の後に）を使用して、絶対リスタの .out ファイルを指定する必要があります。

-ar 再配置可能で実行可能なオブジェクト・モジュールを作成します。

--args=*size* ローダが引数を渡すために使用するメモリを割り当てます。

-b シンボリック・デバッグ情報のマージを抑制します。

-c TMS320C55x C/C++ コンパイラの ROM 自動初期化モデルにより定義されるリンク規則を使用します。

-cr TMS320C55x C/C++ コンパイラの RAM 自動初期化モデルにより定義されるリンク規則を使用します。

-e=*global_symbol* 出力モジュールのエントリ・ポイントを指定する *global_symbol* を定義します。-c または -cr オプションを使用している場合、デフォルト・エントリ・ポイントとして _c_int00 が使用されます。

-f=*fill_value* 出力セクション内のホールの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* *global_symbol* をグローバルに保持します（-h を無効にします）。

-h すべてのグローバル・シンボルを静的にします。

-help
-? ヘルプ・メニューを出力します。

-heap=*size* ヒープ・サイズ（C/C++ の動的メモリ割り当て用）を *size* バイトに設定し、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・サイズは 2K バイトです。

-I=*pathname* ファイル検索アルゴリズムを変更し、デフォルト・ロケーション内を検索する前に *pathname* の中を検索するようにします。-l オプションより前にすべての -I オプションを指定する必要があります。ディレクトリまたはファイル名はオペレーティング・システムの規則に従っていなければなりません。

-j	条件付きリンクを抑制します。
-l=*filename*	リンカの入力としてファイルの名前を指定します。*filename* は、アーカイブ、オブジェクト・ファイル、またはリンカ・コマンド・ファイルを指定できます。-I オプションを使用して検索パスを設定した後にのみ、-l オプションを使用してファイルを指定する必要があります。ディレクトリまたはファイル名はオペレーティング・システムの規則に従っていなければなりません。
-m=*filename*	ホールおよびシンボルを含む入出力セクションのマップ・ファイル・リストを生成します。生成されたファイルは *filename* と命名されます。
-o=*filename*	実行可能な出力モジュールに名前を付けます。デフォルトのファイル名は a.*out* です。ディレクトリまたはファイル名はオペレーティング・システムの規則に従っていなければなりません。
-priority	リンカがシンボル参照を解決しようとするときに指定した順序でライブラリを検索するようにします。
-r	再配置可能な出力モジュールを生成します。
-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取り去ります。
-stack=*size*	プライマリ・スタック・サイズを *size* バイトに設定し、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・サイズは 1K バイトです。
-sysstack=*size*	二次システム・スタック・サイズを *size* バイトに設定し、二次システム・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・サイズは 1000 バイトです。
-u=*symbol*	未解決の外部 *symbol* を、出力モジュールのシンボル・テーブルに挿入します。これにより、リンカは、リンクを完了するためにシンボルの定義を検出するように強制されます。
-v*n*	COFF 出力フォーマットを指定します。ただし、*n* は 0、1、または 2 です。デフォルトのフォーマットは COFF2 です。
-w	リンカが未定義の出力セクションを作成したときに、メッセージを表示します。
-x	後方参照を解決するために、ライブラリの再読み取りを強制実行します。
--xml_link_info=*file*	リンク結果の詳細情報を含む見やすい XML ファイルを生成します。

8.4.1 再配置機能（-a オプションと -r オプション）

リンカは再配置を実行します。再配置は、あるシンボルのアドレスが変わったときに、そのシンボルに対するすべての参照を調整するプロセスです。リンカは 2 つのオプション（-a と -r）をサポートしており、それを使用して絶対出力モジュールや再配置可能な出力モジュールを作成できます。

❑ 絶対的な出力モジュールの作成（-a オプション）

-r オプションを使用せずに -a オプションを使用すると、リンカは実行可能な絶対出力モジュールを作成します。絶対ファイルは、再配置情報を何も保持していません。実行可能なファイルには次のものが含まれます。

- リンカによって定義された特別シンボル（これらのシンボルについては、8.15.4 項「リンカで定義されるシンボル」（8-71 ページ）に説明があります）。
- プログラム・エントリ・ポイントなどの情報を記述するオプションのヘッダ。
- 未解決のシンボルの参照なし。

次の例では、file1.obj と file2.obj がリンクされて、a.out という絶対出力モジュールが作成されます。

```
cl55 -z -a file1.obj file2.obj
```

注 : - a オプションと - r オプション

-a と -r のどちらのオプションも使用しなかった場合、リンカは -a が指定されたものとして動作します。

❑ 再配置可能な出力モジュールの作成（-r オプション）

-r オプションを使用すると、リンカは出力モジュールに再配置エントリを保持します。出力モジュールが（ロード時に）再配置されるか、（他のリンカの実行により）再リンクされる場合は、-r オプションを指定して再配置エントリを保持します。

-a オプションを使用せずに -r オプションを使用すると、リンカは実行不可能なファイルを作成します。実行できないファイルには、特別なリンカ・シンボルもオプションのヘッダも含まれません。このファイルには未解決の参照が含まれていることもありますが、これらの参照により出力モジュールが作成されなくなることはありません。

次の例では、file1.obj と file2.obj がリンクされて、a.out という再配置可能な出力モジュールが作成されます。

```
cl55 -z -r file1.obj file2.obj
```

出力ファイル a.out は、他のオブジェクト・ファイルと再リンクしたり、またはロード時に再配置したりできます（他のファイルと再リンクされるファイルをリンクすることを、部分リンクまたはインクリメンタル・リンクといいます）。詳細は、8.20 節「リンカの例」（8-98 ページ）を参照してください。

❑ **実行可能で再配置可能な出力モジュールの作成（-ar オプションの組み合わせ）**

-a オプションと -r オプションの両方を指定してリンカを起動すると、リンカは実行可能で再配置可能なオブジェクト・モジュールを作成します。この出力ファイルには、特別なリンカ・シンボル、オプションのヘッダ、および解決されたすべてのシンボル参照が含まれています。ただし、再配置情報は保持されます。

次の例では、file1.obj と file2.obj がリンクされて、xr.out という名前の実行可能で再配置可能な出力モジュールが作成されます。

```
cl55 -z -ar file1.obj file2.obj -o xr.out
```

2 つのオプションを結合して入力する（cl55 -z -ar）ことも、または個別に入力することができます（cl55 -z -a -r）。

❑ **絶対的な出力モジュールの再配置または再リンク**

リンカは、再配置情報またはシンボル・テーブル情報が含まれていないファイルを検出すると、警告メッセージを発行します（ただし実行は継続します）。絶対的なファイルを再リンクすることは、各入力ファイルに再配置する必要がある情報が含まれていないときのみ可能です（つまり、どのファイルにも未解決の参照がなく、リンカがそのファイルを作成したときにバインドされた同じ仮想アドレスにバインドされるときのみです）。

8.4.2 絶対リスト・ファイルの作成（-abs オプション）

-abs オプションは、リンクされた各ファイルにつき 1 つの出力ファイルを作成します。出力ファイルの名前は、入力ファイルの名前に拡張子 .abs を付けたものとなります。ただし、ヘッダ・ファイルについては対応する .abs ファイルは作成されません。

8.4.3 ローダが引数を渡すために使用するメモリの割り当て（--args オプション）

args オプションは、リンカに対して、ローダがローダのコマンド行からプログラムに引数を渡すために使用するメモリを割り当てるよう指示します。--args オプションの構文は次のとおりです。

-args=*size*

size は、コマンド行の引数用にターゲット・メモリに割り当てるバイト数を表す数値です。

デフォルトで、リンカは _ _c_args_ _ シンボルを作成し、このシンボルに -1 を設定します。--args=*size* を指定すると、以下が実行されます。

❑ リンカは、*size* バイトの初期化されない .args という名前のセクションを作成します。
❑ _ _c_args_ _ シンボルに .args セクションのアドレスが含まれます。

ローダとターゲット・ブート・コードは、.args セクションと _ _c_args_ _ シンボルを使用して、ホストからターゲット・プログラムに引数を渡すかどうかと引数を渡す場合はその方法を決定します。

8.4.4 シンボリック・デバッグ情報のマージの抑止（-b オプション）

リンカは、デフォルトでは、シンボリック・デバッグ情報に重複エントリができないようにします。このような重複情報が作成されるのは、通常は、C プログラムをデバッグ用にコンパイルした場合です。次に例を示します。

```
-[ header.h ]-
typedef struct
{
   <define some structure members>
} XYZ;

-[ f1.c ]-
#include "header.h"
...
-[ f2.c ]-
#include "header.h"
...
```

この例のファイルをデバッグ用にコンパイルすると、f1.obj と f2.obj の両方にタイプ XYZ を記述するシンボリック・デバッグ・エントリができます。最終出力ファイルで必要なのは、これらのエントリのセットのうち 1 つだけです。リンカは、重複エントリを自動的に削除します。

-b オプションはリンカに対して、このような重複エントリを保存するように指定する場合に使用されます。ローダは多くの情報を読み取り、保持する必要があるため、-b を使用するとローダは低速になります。

8.4.5 C 言語オプション（-c オプションと -cr オプション）

-c および -cr のオプションを使用すると、リンカは C/C++ コンパイラが必要なリンク規則を使用するようになります。

- ❏ -c オプションはリンカに対して、ROM 自動初期化モデルを指定するように指示します。
- ❏ -cr オプションはリンカに対して、RAM 初期化モデルを指定するように指示します。

-e オプションが指定されない場合、-c と -cr オプションは未解決の参照を _c_int00 に挿入します。

C/C++ コードのリンクの詳細は、8.19 節「C/C++ コードのリンク」（8-93 ページ）、および 8.19.6 項「-c リンカ・オプションと -cr リンカ・オプション」（8-97 ページ）を参照してください。

8.4.6 エントリ・ポイントの定義（-e *global_symbol* オプション）

プログラムが実行を開始するメモリ・アドレスをエントリ・ポイントと呼びます。ローダがプログラムをターゲット・メモリにロードするとき、プログラム・カウンタをエントリ・ポイントに初期化する必要があります。これにより、プログラム・カウンタはプログラムの始まりを指示します。

リンカは、エントリ・ポイントに 4 種類の値を割り当てることができます。次に示した 4 つの値について、リンカはこの順に使おうとします。最初の 3 つの値のうちのいずれかを使用する場合、その値はシンボル・テーブル内の外部シンボルでなければなりません。

- -e オプションで指定された値。構文は次のとおりです。

  -e *global_symbol*

  global_symbol はエントリ・ポイントを定義し、入力ファイル内で外部シンボルとして定義されなければなりません。

- シンボル _c_int00 の値（指定されている場合）。C/C++ コンパイラによって生成されたコードをリンクしている場合は、_c_int00 は必ずエントリ・ポイントでなければなりません。

- シンボル _main の値（指定されている場合）

- ゼロ（デフォルト値）

この例では、file1.obj と file2.obj をリンクします。シンボル begin はエントリ・ポイントです。begin は、file1 か file2 で定義され、外部から見えている（アクセスできる）必要があります。

```
cl55 -z -e begin file1.obj file2.obj
```

8.4.7 デフォルトの埋め込み値の設定（-f *cc* オプション）

-f オプションは、出力セクション内に形成されたホールを埋めます。または、初期化されないセクションと初期化されたセクションを結合するときに、初期化されないセクションを初期化します。この機能を使用すると、ソース・ファイルを再アセンブルせずに、リンク時にメモリ領域を初期化できます。-f オプションの構文は次のとおりです。

-f *cc*

引数 *cc* は 16 ビットの定数です（最大 4 桁の 16 進数字）。-f を指定しないと、リンカはデフォルトの埋め込み値として 0 を使用します。

この例では、ホールは 16 進値 ABCD で埋められます。

```
cl55 -z -f 0ABCDh file1.obj file2.obj
```

8.4.8 シンボルのグローバル化（-g *global_symbol* オプション）

-h オプションは、すべてのグローバル・シンボルを静的にします。グローバルにして残したいシンボルがあり、-h オプションを使用する場合は、-g オプションを使用してそのシンボルをグローバルに宣言できます。-g オプションは、指定したシンボルについて、-h オプションによる効果を無効にします。-g オプションの構文は次のとおりです。

-g *global_symbol*

8.4.9 すべてのグローバル・シンボルの静的化（-h オプション）

-h オプションは、.global アセンブラ疑似命令で定義されたすべてのグローバル・シンボルを静的にします。静的シンボルは、外部からリンクしたモジュールには見えません。グローバル・シンボルを静的にすると、グローバル・シンボルは実質的に隠されます。これにより、同じ名前の（別のファイル内にある）外部シンボルを固有のものとして扱うことができます。

-h オプションは、すべての .global アセンブラ疑似命令を実質的に無効にします。すべてのシンボルは、そのシンボルが定義されているモジュールのローカル・シンボルになるので、外部参照が不可能になります。たとえば、b1.obj、b2.obj、および b3.obj は互いに関連していて、グローバル変数 GLOB を参照しているとします。また、d1.obj、d2.obj、および d3.obj は互いに関連していて、別のグローバル変数 GLOB を参照しているとします。この場合、-h オプションと部分リンクを使用すれば、関連ファイルを競合なしにリンクできます。

```
cl55 -z -h -r b1.obj b2.obj b3.obj -o bpart.out
cl55 -z -h -r d1.obj d2.obj d3.obj -o dpart.out
```

-h オプションにより、bpart.out と dpart.out はグローバル・シンボルをもたないこと、つまり GLOB は 2 つの別々の GLOB として取り扱われることが保証されます。-r オプションを指定すると、bpart.out と dpart.out はそれぞれの再配置エントリを保持できます。こうすると、この 2 つの部分的にリンクされたファイルは、次のコマンドを使用してリンクしても安全です。

```
cl55 -z bpart.out dpart.out -o system.out
```

8.4.10 ヒープ・サイズの定義（-heap *constant* オプション）

C/C++ コンパイラは、.sysmem と呼ばれる初期化されないセクションを、malloc() の使用する C ランタイム・メモリ・プールに使用します。このメモリ・プールのサイズは、リンク時に -heap オプションを使用して設定できます。-heap オプションの構文は次のとおりです。

-heap *size*

このオプションのすぐ後に、定数としてバイト単位でサイズを指定します。

```
cl55 -z -heap 0x0400 /* defines a heap size */
```

リンカは、入力ファイルのいずれかに .sysmem セクションがある場合のみ、.sysmem セクションを作成します。

リンカはまた、グローバル・シンボル __SYSMEM_SIZE を作成し、このシンボルにヒープ・サイズを割り当てます（バイト単位）。デフォルト・サイズは 2000 バイトです。

C コードのリンク方法の詳細は、8.19 節「C/C++ コードのリンク」（8-93 ページ）を参照してください。

8.4.11 ファイル検索アルゴリズムの変更（-l オプション、-i オプション、および C55X_C_DIR/C_DIR 環境変数）

通常、ファイルをリンカ入力として指定する場合、単にファイル名を入力します。リンカは、現行ディレクトリのファイルを検索します。たとえば、現行のディレクトリにライブラリ object.lib があるとします。このライブラリに、file1.obj で参照されているシンボルが定義されているものとします。このファイルのリンク方法を次に示します。

```
cl55 -z file1.obj object.lib
```

現行のディレクトリにないファイルを使用する場合は、-l（小文字の L）リンカ・オプションを使用します。このオプションの構文は次のとおりです。

-l [*pathname*] *filename*

filename は、アーカイブ、オブジェクト・ファイル、またはリンカ・コマンド・ファイルの名前です。-l とファイル名との間の空白は、入れても入れなくてもかまいません。

出力セクションへの入力に、オブジェクト・ライブラリの 1 つ以上のメンバが指定された場合、-l オプションは必要ありません。詳細は、8.9.4 項「アーカイブ・メンバを出力セクションに割り当てる方法」（8-40 ページ）を参照してください。

-i リンカ・オプションまたは C55X_C_DIR 環境変数を使用して、リンカのディレクトリ検索アルゴリズムを補強させることができます。リンカは、次の順序で入力ファイルを検索します。

1） -i リンカ・オプションを使用して指定されたディレクトリを検索します。

2） C_DIR と C55X_C_DIR を使用して指定されたディレクトリを検索します。

3） C_DIR と C55X_C_DIR が設定されていない場合は、アセンブラの環境変数 C55X_A_DIR と A_DIR を使用して指定されたディレクトリを検索します。

4） 現行のディレクトリを検索します。

8.4.11.1 代替ファイル・ディレクトリの名前の指定（-i オプション）

-i オプションは、入力ファイルが入っている代替ディレクトリの名前を指定します。このオプションの構文は次のとおりです。

-I *pathname*

pathname は、入力ファイルが入っているディレクトリの名前を指定します。-i とディレクトリ名の間の空白は、入れても入れなくてもかまいません。

リンカは、-l オプションにより指定された入力ファイルを検索するときは、-I で指定されたディレクトリ内を最初に検索します。各 -I オプションは、1 つのディレクトリしか指定できませんが、1 回の起動につき複数の -I オプションを指定できます。-I オプションを使用して代替ディレクトリを指定する場合は、コマンド行またはコマンド・ファイルで -l オプションの前に指定する必要があります。

たとえば、r.lib と lib2.lib という 2 つのアーカイブ・ライブラリがあるとします。次の表は、r.lib と lib2.lib があるディレクトリ、環境変数の設定方法、およびリンク時における 2 つのライブラリの指定方法を示しています。使用しているオペレーティング・システムに従って、次のように選択してください。

O.S.	パス名	入力するコマンド
Windows	\ld と \ld2	`cl55 -z f1.obj f2.obj -I\ld -I\ld2 -lr.lib -llib2.lib`
UNIX（Bourne シェル）	/ld と /ld2	`cl55 -z f1.obj f2.obj -I/ld -I/ld2 -lr.lib -llib2.lib`

8.4.11.2 代替ファイル・ディレクトリの名前の指定（C_DIR 環境変数）

環境変数は、ユーザー定義により文字列を割り当てるオペレーティング・システムのシンボルです。リンカは、C_DIR と C55X_C_DIR という環境変数を使用して、入力ファイルが含まれる代替ディレクトリを指定します。次のコマンド構文を使用して環境変数を割り当てます。

O.S.	入力するコマンド
Windows	**set C_DIR=** *pathname*$_1$**;***pathname*$_2$**; . . .**
UNIX（Bourne シェル）	**C_DIR="***pathname*$_1$**;***pathname*$_2$**; . . .";** **export C_DIR**

pathnames は、入力ファイルが入っているディレクトリです。コマンド行またはコマンド・ファイルで -l オプションを使用して、特定の入力ファイルを検索するために、いつファイル検索ディレクトリのリストを使用するかを、リンカに指示します。

次の例では、r.lib と lib2.lib という 2 つのアーカイブ・ライブラリが、ld と ld2 というディレクトリに入っているとします。次の表は、r.lib と lib2.lib が入っているディレクトリ、環境変数の設定方法、およびリンク時における 2 つのライブラリの使用方法を示しています。使用しているオペレーティング・システムに従って、次のように選択してください。

O.S.	パス名	起動コマンド
Windows	\ld と \ld2	`set  C_DIR=\ld;\ld2` `cl55 -z f1.obj f2.obj -l r.lib -l lib2.lib`
UNIX（Bourne シェル）	/ld と /ld2	`C_DIR="/ld;/ld2"; export C_DIR` `cl55 -z f1.obj f2.obj -l r.lib -l lib2.lib`

システムをリブートするか、または次のコマンドを入力して変数をリセットするまで、環境変数は設定されたままになっています。

O.S.	入力するコマンド
Windows	`set C_DIR=`
UNIX（Bourne シェル）	`unset C_DIR`

アセンブラは A_DIR という環境変数を使用して、コピー / インクルード・アセンブリ・ソース・ファイルまたはマクロ・ライブラリを含んだ代替ディレクトリの名前を指定します。C_DIR が設定されていない場合、リンカは、A_DIR で指定されたディレクトリ内の入力ファイルを検索します。オブジェクト・ライブラリの詳細は、8.7 節「オブジェクト・ライブラリ」（8-26 ページ）にあります。

8.4.12 条件付きリンクの無効化（-j オプション）

-j オプションは、参照されないセクションの削除を抑制します。.clink アセンブラ疑似命令により、削除する候補としてマーク付けされたセクションだけが条件付きリンクの影響を受けます。.clink 疑似命令を使用して条件付きリンクを設定する方法の詳細は、4-39 ページを参照してください。

8.4.13 マップ・ファイルの作成（-m *filename* オプション）

-m オプションを指定すると、リンカ・マップ・リストが作成され、そのマップは *filename* に入れられます。-m オプションの構文は次のとおりです。

-m *filename*

データ・セクションで定義されたシンボルはワード・アドレス値をもち、コード・セクションで定義されたシンボルはバイト・アドレス値をもちます。

リンカ・マップには次の事項が記述されます。

- ❏ メモリ構成
- ❏ 入出力セクションの割り当て
- ❏ 外部シンボルの再配置後のアドレス

マップ・ファイルには、出力モジュール名とエントリ・ポイントが入っています。また、次の 3 つのテーブルが入っていることもあります。

- ❏ デフォルトではないメモリが指定されている場合には、新しいメモリ構成を示すテーブル
- ❏ 各出力セクションと出力セクションを構成する入力セクションのリンクされたアドレスを示すテーブル
- ❏ 各外部シンボルとそのアドレスを示すテーブル。このテーブルは 2 つに分けてリストされ、左側のリストには名前順にソートされたシンボルが入り、2 番目のリストにはアドレス順にソートされたシンボルが入ります。

この例では、file1.obj と file2.obj がリンクされて、file.map というマップ・ファイルが作成されます。

```
cl55 -z file1.obj file2.obj -m file.map
```

例 8–24「出力マップ・ファイル demo.map」（8-100 ページ）は、マップ・ファイルの例を示します。

8.4.14 出力モジュールの名前の指定（-o *filename* オプション）

エラーが検出されないと、リンカは出力モジュールを作成します。出力モジュールのファイル名を指定しない場合、リンカはデフォルト名の a.out を使用します。出力モジュールを別のファイルに書き込む場合は、-o オプションを指定します。-o オプションの構文は次のとおりです。

-o *filename*

filename は、新しい出力モジュールの名前です。

この例では、file1.obj と file2.obj をリンクして run.out という名前の出力モジュールを作成します。

```
cl55 -z -o run.out file1.obj file2.obj
```

8.4.15 シンボル情報の除去（-s オプション）

-s オプションを指定すると、シンボル・テーブル情報と行番号エントリが省略され、出力モジュールのサイズを小さくできます。実働アプリケーションでは、コンシューマにシンボル情報を開示したくないときに -s オプションを使用すると便利です。

この例では、file1.obj と file2.obj をリンクして、行番号とシンボル・テーブル情報が除去された nosym.out という名前の出力モジュールを作成します。

```
cl55 -z -o nosym.out -s file1.obj file2.obj
```

-s オプションを使用する場合、後でシンボリック・デバッガの使用が制限されます。

8.4.16 スタック・サイズの定義（-stack *constant* オプション）

TMS320C55x C/C++ コンパイラは、初期化されない .stack セクションを使用してランタイム・スタックの空間を割り当てます。リンク時に、-stack オプションを指定して .stack セクションのサイズを設定できます。-stack オプションの構文は次のとおりです。

-stack *size*

このオプションのすぐ後に、定数としてバイト単位でサイズを指定します。

```
cl55 -z -stack 0x1000 /* defines a stack size */
```

入力セクションに異なるスタック・サイズを指定した場合、入力セクションのスタック・サイズは無視されます。入力セクションで定義されているシンボルは、すべて有効なままです。スタック・サイズのみが変わります。

リンカは .stack セクションを定義するときにグローバル・シンボル __STACK_SIZE を定義し、このシンボルにセクションのサイズを割り当てます（バイト単位）。デフォルト・スタック・サイズは 1000 バイトです。

注 : .stack および .sysstack セクションの割り当て

.stack と .sysstack セクションは、同じ 64K ワードのデータ・ページに割り当てる必要があります。

8.4.17 セカンダリ・スタック・サイズの定義（-sysstack *constant* オプション）

TMS320C55x C/C++ コンパイラは、初期化されない .sysstack セクションを使用してセカンダリ・ランタイム・スタックの空間を割り当てます。この .sysstack セクションのサイズは、リンク時に -sysstack オプションを使用して設定できます。-sysstack オプションの構文は次のとおりです。

-sysstack *size*

このオプションのすぐ後に、定数としてバイト単位でサイズを指定します。

```
cl55 -z -sysstack 0x1000 /* defines secondary stack size */
```

リンカは .sysstack セクションを定義するときにグローバル・シンボル __SYSSTACK_SIZE を定義し、このシンボルにセクションのサイズを割り当てます（バイト単位）。デフォルトのセカンダリ・スタック・サイズは 1000 バイトです。

注 : .stack および .sysstack セクションの割り当て

.stack と .sysstack セクションは、同じ 64K ワードのデータ・ページに割り当てる必要があります。

8.4.18 未解決のシンボルの導入（-u *symbol* オプション）

-u オプションを指定すると、未解決のシンボルをリンカのシンボル・テーブルに入れることができます。これにより、リンカは、リンカへのオブジェクト・ファイルとライブラリの入力に混じったそのシンボルの定義を検索するように強制されます。リンカがシンボルを定義するメンバをリンクするには、リンカは -u オプションを検出しなければなりません。

たとえば、rts.lib という名前のライブラリに symtab というシンボルを定義したメンバがあり、リンクするどのオブジェクト・ファイルも symtab を参照しないとします。しかし、出力モジュールを再リンクする必要があり、このリンクに symtab を定義したライブラリ・メンバをインクルードしようとしているとします。次に示すように -u オプションを指定すると、リンカは、rts.lib で symtab を定義したメンバを検索し、そのメンバをリンクするように強制されます。

```
cl55 -z -u symtab file1.obj file2.obj rts.lib
```

-u オプションを指定しない場合、file1.obj または file2.obj のどちらからもこのメンバを明示的に参照しないので、このメンバはインクルードされません。

8.4.19 COFF フォーマットの指定（-v オプション）

-v オプションは、COFF オブジェクト・ファイルを作成するためにリンカにより使用されるフォーマットを指定します。COFF オブジェクト・ファイルは、リンカの出力です。このフォーマットは、オブジェクト・ファイル内に情報を配置する方法を指定します。

リンカは、COFF0、COFF1、および COFF2 の各フォーマットを読み書きできます。デフォルトで、リンカは COFF2 ファイルを作成します。異なる出力フォーマットを作成するには、-v オプションを使用します。ただし、*n* は、COFF0 の場合には 0、COFF1 の場合には 1 です。

COFF の詳細は、第 2 章「共通オブジェクト・ファイル・フォーマットの概要」、および付録 A 「共通オブジェクト・ファイル・フォーマット」を参照してください。

注 : DWARF デバッグの COFFO および COFF1 との非互換性

デフォルトで、コード生成ツールは DWARF デバッグ情報を生成します。この結果、コンパイラは、COFF0 および COFF1 フォーマットと互換性のない名前をもつデバッグ・セクションを生成します。-v0 または -v1 リンカ・オプションを指定すると、リンク時エラーが発生します。

8.4.20 出力セクション情報のメッセージの表示（-w オプション）

-w オプションを指定すると、デフォルトの出力セクションの作成に関連した追加メッセージが表示されます。追加メッセージは、次の環境で表示されます。

- リンカ・コマンド・ファイルの中で、入力セクションをどのように結合して出力セクションにするかを記述した SECTIONS 疑似命令を設定できます。しかし、リンカは、SECTIONS 疑似命令の中で、対応する出力セクションが定義されていない入力セクションを 1 つ以上検出すると、同じ名前をもつ複数の入力セクションを結合して、その名前が付いた 1 つの出力セクションにします。デフォルトでは、リンカは、それが発生したことを知らせるメッセージを表示しません。

  この状況が発生し、-w オプションを使用している場合、リンカは、新しい出力セクションを作成したときにメッセージを表示します。

- -heap、-stack、および -sysstack オプションを使用しなかった場合、リンカは自動的に、それぞれ .sysmem、.stack、および .sysstack セクションを作成します。.sysmem セクションのデフォルト・サイズは 2000 バイトです。.stack と .sysstack セクションのデフォルト・サイズは 1000 バイトです。これらのセクションのどちらか、またはすべてに割り当てる十分なメモリが存在しない場合も考えられます。その場合、リンカは、セクションの割り当てができなかったことを知らせるエラー・メッセージを発行します。

  -w オプションを使用した場合、リンカはさらに詳しい別のメッセージを表示します。そのメッセージには、ユーザー自身が .system または .stack セクションを割り当てるための疑似命令の名前が含まれています。

注 : .stack および .sysstack セクションの割り当て

.stack と .sysstack セクションは、同じ 64K ワードのデータ・ページに割り当てる必要があります。

SECTIONS 疑似命令の詳細は、8.9 節「SECTIONS 疑似命令」（8-32 ページ）を参照してください。リンカのデフォルト動作の詳細は、8.13 節「デフォルトの割り当てアルゴリズム」（8-64 ページ）を参照してください。

8.4.21 ライブラリの強制読み取りと検索（-x および -priority オプション）

未解決のシンボルを強制的に検索するには、次の 2 つの方法があります。

- ❏ シンボル参照を解決できない場合、ライブラリを再度読み取ります（-x）。
- ❏ ライブラリを指定した順序で検索します（-priority）。

リンカは通常、入力ファイル（アーカイブ・ライブラリを含む）を、コマンド行またはコマンド・ファイルで検出したときに 1 回だけ読み取ります。アーカイブが読み取られると、未定義のシンボルに対する参照を解決するすべてのメンバがリンクに入れられます。その後、入力ファイルがすでに読み取ったアーカイブ・ライブラリで定義されているシンボルを参照すると、その参照は解決されません。

-x オプションを使用すると、リンカに対してすべてのライブラリを再読み取りするように強制できます。リンカは、解決されていない参照がなくなるまでライブラリを再度読み取ります。-x オプションを使用すると、リンクの速度が低下します。したがって、使用するのは必要なときだけにしてください。たとえば、a.lib には b.lib で定義されているシンボルに対する参照が含まれていて、b.lib には a.lib で定義されているシンボルへの参照が含まれている場合、次に示すようにどちらかのライブラリを 2 回指定することにより、この相互依存関係を解決できます。

```
cl55 -z -la.lib -lb.lib -la.lib
```

または、次のようにして強制的にリンカに解決させることもできます。

```
cl55 -z -x -la.lib -lb.lib
```

-priority オプションは、ライブラリの代替検索メカニズムを提供します。-priority オプションを使用すると、そのシンボルの定義が含まれる最初のライブラリで、個々の未解決の参照が解決されます。次に例を示します。

```
objfile   references A
lib1      defines B
lib2      defines A, B; obj defining A references B

% cl55 -z objfile lib1 lib2
```

既存のモデルの場合、objfile は lib2 の A への参照を解決し、B への参照を制御して、lib2 の B への参照を解決します。

-priority では、objfile は lib2 の A への参照を解決し、B への参照を制御します。しかし、ここでは順番にライブラリを検索することによって見つかる最初の定義、つまり lib1 の B として解決します。

-priority オプションは、すべてのライブラリの完全なバージョンを用意することなく、他のライブラリの一連の関連する関数の定義を無効化するライブラリに便利です。

たとえば、rts55.lib をすべて交換せずに、rts55.lib で定義された malloc と free のバージョンを無効にしようとしているとします。rts55.lib が malloc と free へのすべての参照を確保する前に、-priority を使用し、新しいライブラリにリンクすることによって、新しいライブラリへの参照を解決します。

-priority オプションは、上記で説明したような状況が発生する場合に、プログラムの DSP/BIOS とのリンクをサポートすることを意図しています。

8.4.22 XML リンク情報ファイルの生成（--xml_link_info オプション）

リンカは、--xml_link_info *file* オプションを介して、XML リンク情報ファイルの生成をサポートします。このオプションは、リンカがリンク結果の詳細情報を含む見やすい XML ファイルを生成するようにします。このファイルに入れられる情報には、リンカが生成するマップ・ファイルに現在生成されているすべての情報が含まれます。

生成するファイルの内容を指定する方法の詳細は、付録 C「XML リンク情報ファイルの説明」を参照してください。

8.5 バイト / ワード・アドレッシング

C55x メモリは、コード用にバイト・アドレス、データ用にワード・アドレスです。アセンブラとリンカは、アドレス、相対オフセット、およびビットのサイズを、指定したセクションに適した単位（データ・セクションではワード、コード・セクションではバイト単位）で記録します。

注 : リンカ・コマンド・ファイルでのバイト・アドレスの使用
リンカ・コマンド・ファイル内で指定するすべてのアドレスとサイズは、コードとデータのどちらのセクションでもバイト・アドレスでなければなりません。

プログラム・ラベルの場合、無変更のバイト・アドレスがエンコードされて実行可能出力となり、実行中にプログラム・アドレス・バスを通じて送信されます。データ・ラベルの場合、バイト・アドレスは実行可能出力へとコード化される前に、（それらのアドレスをワード・アドレスに変換する）リンカによって 2 で除算され、データ・アドレス・バスを通じて送信されます。

リンカが生成する .map ファイルは、コードのアドレスとサイズをバイト単位で示し、データのアドレスとサイズをワード単位で示します。

8.6 リンカ・コマンド・ファイル

リンカ・コマンド・ファイルを使用すると、リンク情報をファイルに組み込むことができます。リンカを同じ情報で何度も起動するときに、この機能を使用すると便利です。また、リンカ・コマンド・ファイルを使用すると、MEMORY および SECTIONS の疑似命令を使用してアプリケーションをカスタマイズできるので便利です。これらの疑似命令は、リンカ・コマンド・ファイルでのみ使用できます。

注 : リンカ・コマンド・ファイルでのバイト・アドレスの使用
リンカ・コマンド・ファイル内で指定するすべてのアドレスとサイズは、コードとデータのどちらのセクションでもバイト・アドレスでなければなりません。

リンカ・コマンド・ファイルは、次の要素を 1 つか複数含む ASCII ファイルです。

- ❑ 入力ファイル名。オブジェクト・ファイル、アーカイブ・ライブラリ、または他のコマンド・ファイルを指定します。
- ❑ リンカ・オプション。このオプションは、コマンド行で使用するのと同様にコマンド・ファイルで使用できます。
- ❑ MEMORY および SECTIONS リンカ疑似命令。MEMORY 疑似命令を使用すると、ターゲット・メモリの構成を定義できます。SECTIONS 疑似命令を使用すると、セクションの構築と割り当ての方法が制御できます。
- ❑ 代入文。グローバル・シンボルを定義し、それに値を割り当てます。

コマンド・ファイルでリンカを起動するには、コマンド **cl55 -z** を入力し、その後にコマンド・ファイル名を入力します。

```
cl55 -z command_filename
```

リンカは、検出した順に入力ファイルを処理します。ライブラリ名が指定されている場合、リンカはライブラリを検索して、未解決のシンボルの定義を検出します。リンカは、検出したファイルをオブジェクト・ファイルとして認識すると、そのファイルをリンクに組み込みます。検出したファイルがオブジェクト・ファイルでない場合、リンカは、そのファイルをコマンド・ファイルとみなし、その中のコマンドを読み取って処理します。コマンド・ファイル名は、使用しているシステムに関係なく大文字と小文字を区別します。

例 8–1 は、link.cmd という名前のサンプルのリンカ・コマンド・ファイルを示しています（2.3.2 項「セクションのメモリ・マップへの配置」（2-14 ページ）に、リンカ・コマンド・ファイルの別の例があります）。

例 8–1. リンカ・コマンド・ファイル

```
a.obj         /*  First input filename          */
b.obj         /*  Second input filename         */
-o prog.out   /*  Option to specify output file */
-m prog.map   /*  Option to specify map file    */
```

例 8–1 のサンプル・ファイルにはファイル名とオプションだけが含まれています。コマンド・ファイルには、/* と */ で囲むことによりコメントを指定できます。このコマンド・ファイルでリンカを起動するには、次のように入力します。

```
cl55 -z link.cmd
```

コマンド・ファイルを使用するときには、コマンド行に別のパラメータを入れることができます。

```
cl55 -z -r link.cmd c.obj d.obj
```

リンカは、link.cmd を検出するとそのコマンド・ファイルを直ちに処理します。したがって、a.obj と b.obj は c.obj と d.obj よりも先に出力モジュールにリンクされます。

複数のコマンド・ファイルを指定できます。たとえば、ファイル名が入っている names.lst という名前のファイルと、リンカ疑似命令が入っている dir.cmd という名前の別のファイルがある場合には、次のように入力します。

```
cl55 -z names.lst dir.cmd
```

コマンド・ファイルから別のコマンド・ファイルを呼び出すことができます。このようなネストは 16 レベルまでに制限されています。

ファイル名とオプション・パラメータを除き、コマンド・ファイル内の空白および空白行は、区切り文字として有効である以外は意味が有りません。これは、コマンド・ファイルのリンカ疑似命令のフォーマットについても同様です。

注 : スペースまたはハイフンを含むファイル名とオプション・パラメータ

コマンド・ファイル内では、組み込みのスペースまたはハイフンを含んでいるファイル名とオプション・パラメータは、引用符で囲まれる必要があります。次に例を示します。
“this-file.obj”

例 8–2 は、リンカ疑似命令が入ったコマンド・ファイルの例を示しています（リンカ疑似命令のフォーマットについては、この後の節で説明します）。

例 8–2. リンカ疑似命令を含むコマンド・ファイル

```
a.obj b.obj c.obj          /* Input filenames       */
-o prog.out -m prog.map    /* Options               */

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

SECTIONS                   /* SECTIONS directive    */
{
  .text:> ROM
  .data:> RAM
  .bss:> RAM
}
```

8.6.1 リンカ・コマンド・ファイル内で予約済みの名前

次の名前は、リンカ疑似命令のキーワードとして予約されています。これらの名前は、コマンド・ファイル内でシンボル名やセクション名として使用することはできません。

align	GROUP	origin
ALIGN	l（小文字の L）	ORIGIN
attr	len	page
ATTR	length	PAGE
block	LENGTH	range
BLOCK	load	run
COPY	LOAD	RUN
DSECT	MEMORY	SECTIONS
f	NOLOAD	spare
fill	o	type
FILL	org	TYPE
group		UNION

8.6.2 コマンド・ファイルの定数

定数は、2 つの構文形式のどちらかを使用して指定できます。1 つはアセンブラで使用される 10 進、8 進、16 進のいずれかの定数を指定するために使用する形式（3.8 節「定数」（3-26 ページ）を参照）で、もう 1 つは、C の構文で整数定数に使用する形式です。

例 :

	10 進数	8 進数	16 進数
アセンブラ・フォーマット	32	40q	20h
C フォーマット :	32	040	0x20

8.7 オブジェクト・ライブラリ

オブジェクト・ライブラリは区画に分割されたアーカイブ・ファイルで、完全なオブジェクト・ファイルをメンバとして組み込みます。通常は、関連性のあるモジュールはグループとしてライブラリに集められています。オブジェクト・ライブラリをリンカ入力として指定すると、リンカは、ライブラリのメンバで既存の未解決のシンボルに対する参照を定義しているメンバをすべて組み込みます。アーカイバを使用して、ライブラリを作成し、保守することができます。アーカイバの詳細は、第 9 章「アーカイバの説明」にあります。

オブジェクト・ライブラリを使用すると、リンク時間を短縮し、実行可能なモジュールのサイズを小さくできます。通常は、関数を含むオブジェクト・ファイルがリンク時に指定されると、そのファイルは使用されるかどうかにかかわらずリンクされます。ただし、その同じ関数がアーカイブ・ライブラリに入っている場合は、参照されるときだけ組み込まれます。

ライブラリを指定する順序には注意を払う必要があります。これは、リンカは、ライブラリの検索時に未定義のシンボルを解決するメンバのみを組み込むからです。同じライブラリは、必要に応じて何度でも指定できます。ライブラリは組み込まれるたびに検索されます。この代わりに、-x オプションを指定することもできます。ライブラリには、このライブラリで定義されているすべての外部シンボルを格納したテーブルがあります。リンカは、このライブラリがこれ以上参照を解決するのに役に立たないと判断するまで、このテーブルを検索します。

次の例では、いくつかのオブジェクト・ファイルがライブラリとリンクされます。次の条件を前提とします。

- ❏ 入力ファイル f1.obj と f2.obj は、両方とも clrscr という名前の外部関数を参照している。
- ❏ 入力ファイル f1.obj は、シンボル origin を参照している。
- ❏ 入力ファイル f2.obj は、シンボル fillclr を参照している。
- ❏ ライブラリ libc.lib のメンバ 0 は、origin の定義を含んでいる。
- ❏ ライブラリ liba.lib のメンバ 3 は、fillclr の定義を含んでいる。
- ❏ 両方のライブラリのメンバ 1 は、clrscr を定義している。

たとえば、次のように入力すると、参照は次に示すように解決されます。

```
cl55 -z f1.obj liba.lib f2.obj libc.lib
```

- ❏ liba.lib のメンバ 1 は、clrscr に対する両方の参照を満たす。これは、このライブラリの検索が行われ、clrscr は f2.obj が参照する前に定義されるからです。
- ❏ libc.lib のメンバ 0 は、origin に対する参照を満たす。
- ❏ liba.lib のメンバ 3 は、fillclr に対する参照を満たす。

ただし、次のように入力した場合、clrscr へのすべての参照は、libc.lib のメンバ 1 によって満たされます。

```
cl55 -z f1.obj f2.obj libc.lib liba.lib
```

リンクされたどのファイルもライブラリで定義されたシンボルを参照していない場合は、-u オプションを指定することにより、リンカに強制的にライブラリ・メンバを組み込ませることができます。リンカのグローバル・シンボル・テーブルに未定義シンボル rout1 を作成する例を、次に示します。

```
cl55 -z -u rout1 libc.lib
```

rout1 を定義しているメンバが libc.lib にある場合は、リンカはそのメンバを組み込みます。

リンカを使用すると、アーカイブ・ライブラリの個々のメンバを特定の出力セクションに割り当てることができます。詳細は、8.9.4 項「アーカイブ・メンバを出力セクションに割り当てる方法」（8-40 ページ）を参照してください。

8.4.11 項「ファイル検索アルゴリズムの変更（-l オプション、-i オプション、および C55X_C_DIR/C_DIR 環境変数）」（8-12 ページ）では、オブジェクト・ライブラリを含むディレクトリを指定する方法について説明します。

8.8 MEMORY 疑似命令

リンカは、出力セクションをメモリのどこに割り当てるべきかを決定します。この作業を実行するには、リンカにターゲット・メモリのモデルを指定する必要があります。MEMORY 疑似命令を使用すると、ターゲット・メモリのモデルを指定することができ、使用しているシステムで保持するメモリの種類と、メモリが占めるアドレス範囲を定義できます。リンカは、出力セクションを割り当てるときにモデルを保持し、そのモデルを使用して、どのメモリ・ロケーションにオブジェクト・コードを配置するかを決定します。モデルがリンカ・コマンド・ファイルで指定されていない場合、リンカはデフォルトのメモリ構成を使用します。

TMS320C55x システムのメモリ構成は、アプリケーションによって異なります。
MEMORY 疑似命令を使用すると、さまざまな構成を指定できます。MEMORY 疑似命令を使用してメモリ・モデルを指定した後で、SECTIONS 疑似命令を使用して、定義されたメモリに出力セクションを割り当てることができます。

リンカがセクションを処理する方法の詳細は、2.3 節「リンカによるセクションの処理」（2-12 ページ）を参照してください。セクションの再配置については、2.4 節「再配置」（2-15 ページ）を参照してください。

8.8.1 デフォルトのメモリ・モデル

アセンブラを使用すると、コードを TMS320C55x デバイス用にアセンブルできます。アセンブラは、このデバイスを識別するフィールドを出力ファイルのヘッダに挿入します。リンカは、この情報をオブジェクト・ファイルのヘッダから読み取ります。MEMORY 疑似命令を指定しない場合、リンカはデフォルトのメモリ・モデルを使用します。デフォルト・メモリ・モデルの詳細は、8.13.1 項「割り当てアルゴリズム」（8-64 ページ）を参照してください。

8.8.2 MEMORY 疑似命令の構文

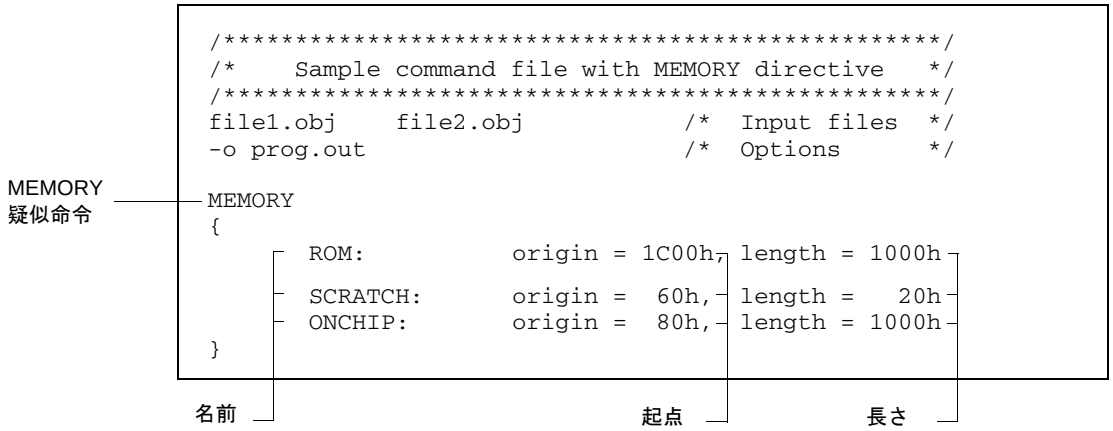
MEMORY 疑似命令は、ターゲット・システム内に物理的に存在し、プログラムが使用できるメモリの範囲を特定します。それぞれのメモリ範囲には、名前、開始アドレス、および長さがあります。

デフォルトで、リンカは PAGE 0 の 1 つのアドレス空間を使用します。ただし、リンカを使用すると、MEMORY 疑似命令の PAGE オプションを使用して、これらのアドレス空間を別々に構成できます。PAGE オプションにより、リンカは、指定されたページを完全に独立したメモリ空間として扱えるようになります。C55x は、最大 255 の PAGE をサポートしますが、使用できる数は選択した構成によって異なります。

MEMORY 疑似命令を使用する場合は、オブジェクト・コード用に使用可能なすべてのメモリ領域を特定する必要があります。MEMORY 疑似命令によって定義されたメモリは構成メモリであり、MEMORY 疑似命令で明示的に定義しなかったメモリは未構成メモリです。リンカは、未構成メモリにはプログラムを配置しません。存在しないメモリ空間を表すには、アドレス範囲を MEMORY 疑似命令文に組み込まないようにするだけです。

MEMORY 疑似命令をコマンド・ファイル内で指定するには、MEMORY（大文字）という語の後に、メモリ範囲指定リストを中括弧に入れて指定します。例 8–3 は、MEMORY 疑似命令を含むサンプル・コマンド・ファイルを示しています。

例 8–3. MEMORY 疑似命令



MEMORY 疑似命令の一般的な構文は次のとおりです。

```
MEMORY
{
    [PAGE 0 :]name 1 [(attr)]  :origin = constant, length = constant;
    [PAGE n :]name n [(attr)]  :origin = constant, length = constant;
}
```

PAGE メモリ空間を特定します。指定した構成により異なりますが、最大 255 ページまで指定できます。通常は、PAGE 0 はプログラム・メモリを、PAGE 2 はペリフェラル・メモリを指定します。PAGE オプションを指定しない場合、リンカは PAGE 0 が指定されているものとして動作します。それぞれの PAGE は、完全に独立したアドレス空間を表します。PAGE 0 の構成メモリは、PAGE 2 の構成メモリと重なってもかまいません。

name メモリ範囲に名前を付けます。メモリ名には 1 から任意の数の文字まで指定できます。使用できる文字は、A ～ Z、a ～ z、$、.、および _ です。メモリ範囲に付けた名前は、リンカにとっては特別な意味がありません。単にメモリ範囲を特定するための記号にすぎません。メモリ範囲名はリンカが内部的に使用するもので、出力ファイルやシンボル・テーブルには保持されません。別のページであれば、メモリ範囲に同じ名前を付けることができます。ただし、同じページではどのメモリ範囲にも固有の名前を付けなければならず、また範囲を重複させることはできません。

attr 名前を付けた範囲に関連する 1 ～ 4 個の属性を指定します。属性の使用は任意ですが、使用するときは括弧に入れなければなりません。属性を使用すると、出力セクションの割り当てを特定のメモリ範囲に制限できます。属性を全く使用しない場合は、どの出力セクションもどのメモリ範囲にでも自由に割り当てることができます。属性が指定されていないメモリは（デフォルトのモデルにあるものを含めて）、4 つの属性すべてをもちます。有効な属性には次のものがあります。

R メモリを読み取れるように指定します。
W メモリに書き込めるように指定します。
X メモリに実行可能なコードを含めるように指定します。
I メモリを初期化できるように指定します。

origin メモリ範囲の開始アドレスを指定します。*origin*、*org*、または *o* を入力します。バイトで指定された値は 24 ビットの定数で、10 進数、8 進数、16 進数のいずれも使用できます。

length メモリ範囲の長さを指定します。*length*、*len*、または *l* を入力します。バイトで指定された値は 24 ビットの定数で、10 進数、8 進数、16 進数のいずれも使用できます。

fill メモリ範囲の埋め込み文字を指定します。*fill* または *f* と入力します。埋め込みの使用は任意です。値は 2 バイトの整数定数で、10 進数、8 進数、16 進数のいずれも使用できます。埋め込み値は、セクションに割り当てられないメモリ範囲の領域を埋めるために使用されます。

注 : メモリ範囲の埋め込み

大きなメモリ範囲に埋め込み値を指定した場合、出力ファイルは非常に大きくなります。その理由は、メモリ範囲の埋め込みを指定すると、(埋め込み値が 0 でも) その範囲内にあるすべての未割り当てメモリ・ブロック用に生データが生成されるからです。

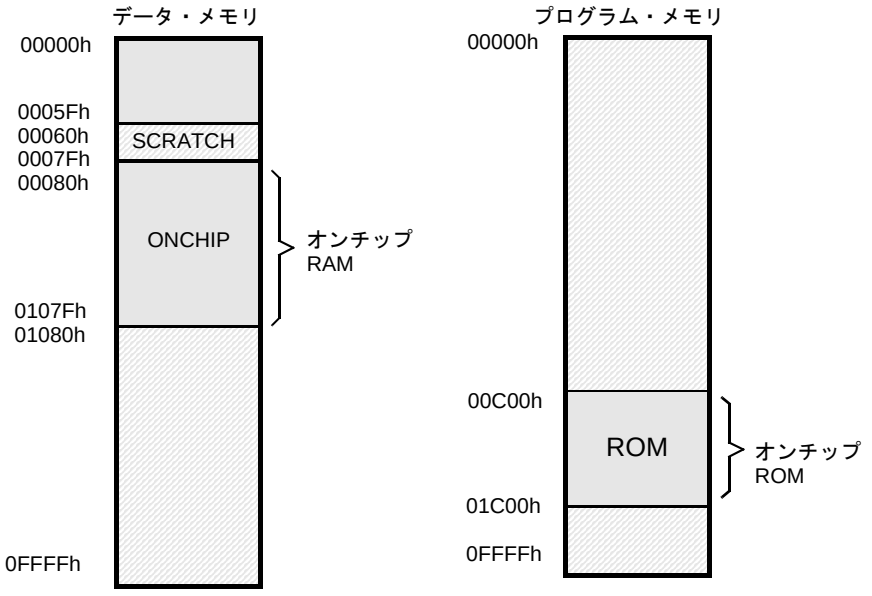
次の例は、R 属性と W 属性、および埋め込み定数 0FFFFh を使用してメモリ範囲を指定しています。

```
MEMORY
{
       RFILE (RW) :o = 02h, l = 0FEh, f = 0FFFFh
}
```

MEMORY 疑似命令は、通常、出力セクションの割り当てを制御するために SECTIONS 疑似命令とともに使用します。MEMORY 疑似命令を使用してターゲット・システムのメモリ・モデルを指定した後、SECTIONS 疑似命令を使用して、出力セクションを特定の名前のメモリ範囲内に割り当てるか、特定の属性をもつメモリ内に割り当てることができます。たとえば、.text および .data セクションを ROM という名前の領域に割り当て、.bss セクションを ONCHIP という名前の領域に割り当てることができます。

図 8-2 は、例 8–3 で示すメモリ・マップを図で示しています。

図 8-2. 例 8–3 で定義されたメモリ・マップ



8.9 SECTIONS 疑似命令

SECTIONS 疑似命令は、次の働きをします。

- ❑ 入力セクションを結合して出力セクションに入れる方法を指示する。
- ❑ 実行可能なプログラム内の出力セクションを定義する。
- ❑ メモリ内の出力セクションを配置する位置を指定する（相互に対照的に、および全メモリ空間について）。
- ❑ 出力セクション名の変更を可能にする。

リンカがセクションを処理する方法の詳細は、2.3 節「リンカによるセクションの処理」（2-12 ページ）を参照してください。セクションの再配置については、2.4 節「再配置」（2-15 ページ）を参照してください。サブセクションの定義については、2.2.4 項「サブセクション」（2-8 ページ）を参照してください。サブセクションを使用すると、セクションをより正確に操作できます。

8.9.1 デフォルト構成

SECTIONS 疑似命令を指定しない場合、リンカはデフォルトのアルゴリズムを使用してセクションを結合し、セクションの割り当てを行ないます。8.13 節「デフォルトの割り当てアルゴリズム」（8-64 ページ）で、このアルゴリズムについて詳しく説明します。

8.9.2 SECTIONS 疑似命令の構文

SECTIONS 疑似命令をコマンド・ファイル内で指定する場合は、SECTIONS（大文字）という文字に続けて、出力セクションの指定を中括弧に入れて指定します。

SECTIONS 疑似命令の一般的な構文は次のとおりです。

```
SECTIONS
{
    name :[property, property, property,...]
    name :[property, property, property,...]
    name :[property, property, property,...]
}
```

各セクションの指定は、*name* で始まり、出力セクションを定義します（出力セクションとは、出力ファイルのセクションです)。セクション名の後ろには、セクションの内容とその割り当て方法を定義する属性のリストが続きます。属性と属性の間はコンマで区切ってもかまいません。セクションに対して指定できる属性には次のものがあります。

- **ロード割り当て**は、メモリ内のセクションをロードする位置を定義します。

  構文 : **load =** *allocation* または
  allocation または
  > *allocation*

- **実行割り当て**は、実行するセクションのメモリ内の位置を定義します。

  構文 : **run =** *allocation* または
  run > *allocation*

- **入力セクション**は、出力セクションを構成する入力セクションを定義します。

  構文 : {input_sections }

- **セクション型**は、特殊なセクション型のフラグを定義します。

  構文 : **type = COPY** または
  type = DSECT または
  type = NOLOAD

  セクション型の詳細は、8.14 節「特別なセクションの型（DSECT、COPY、および NOLOAD)」(8-67 ページ）を参照してください。

- **埋め込み値**は、初期化されないホールに埋め込む値を定義します。

  構文 : **fill =** *value* または
  name:... { ... } **=** *value*

  ホールの作成と埋め込み方法の詳細は、8.16 節「ホールの作成および埋め込みの方法」(8-73 ページ）を参照してください。

例 8–4 は、サンプルのリンカ・コマンド・ファイルでの SECTIONS 疑似命令を示しています。図 8-3 は、これらのセクションがどのようにメモリに割り当てられるかを示しています。

例 8–4. SECTIONS 疑似命令

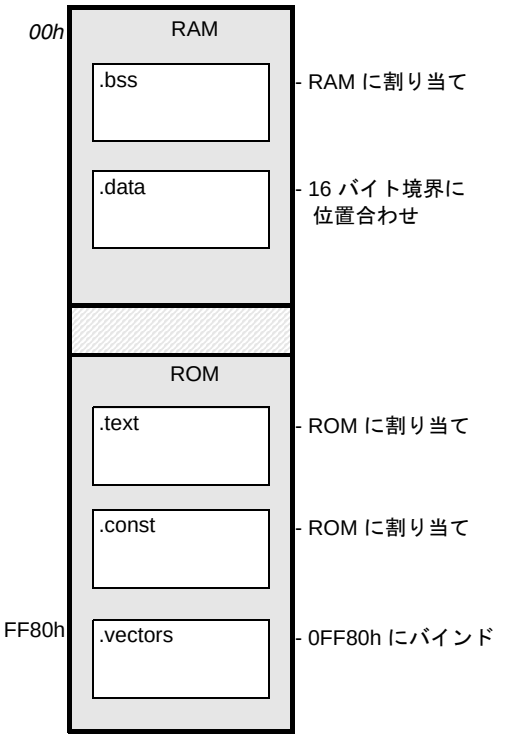
SECTIONS 疑似命令

セクション指定

```
/************************************************/
/*  Sample command file with SECTIONS directive   */
/************************************************/
file1.obj    file2.obj           /*  Input files  */
-o prog.out                      /*  Options      */

SECTIONS
{
  .text:     load = ROM, run = 800h
  .const:    load = ROM
  .bss:      load = RAM
  .vectors:  load = FF80h
     {
        t1.obj(.intvec1)
        t2.obj(.intvec2)
        endvec = .;
     }
  .data:     align = 16
}
```

図 8-3 は、例 8–4 の .vectors、.text、.const、.bss、および .data のセクション疑似命令で定義された出力セクションを示しています。

図 8-3. 例 8–4 で定義されたセクションの割り当て



.bss セクションは、.bss という名前のすべての入力セクションを結合します。

.data セクションは、.data という名前のすべての入力セクションを結合します。リンカはこの .data セクションを十分な空間がある任意の場所に入れ（この図では RAM）、16 バイト境界に位置合わせします。

.text セクションは、.text という名前のすべての入力セクションを結合します。リンカは、.text という名前のすべてのセクションをここに結合します。アプリケーションは、このセクションを 0800h で実行されるように再配置する必要があります。

.const セクションは、.const という名前のすべての入力セクションを結合します。

.vectors セクションは、t1.obj からの .intvec1 セクションと、t2.obj からの .intvec2 セクションにより構成されます。

8.9.3 メモリの配置

リンカは、各出力セクションにターゲット・メモリ内の 2 つの位置を割り当てます。つまり、セクションがロードされる位置と、実行される位置です。通常は、この 2 つの位置は同じで、各セクションは単一のアドレスのみを持っていると考えられます。いずれの場合でも、ターゲット・メモリ内に出力セクションを置き、そのアドレスを割り当てることを「メモリの配置」または「割り当て」と呼びます。別々のロード割り当ておよび実行割り当てを使用する方法の詳細は、8.10 節「セクションのロード時および実行時アドレスの指定方法」（8-45 ページ）を参照してください。

リンカに対してセクションの配置方法を指定しない場合、リンカはデフォルトのアルゴリズムによりセクションの配置を行ないます。一般的には、リンカは構成メモリ内のセクションを配置できる場所ならどこにでもセクションを入れます。このセクションのデフォルトの配置は、SECTIONS 疑似命令内でセクションを定義して、その配置方法を指示することで無効にできます。

1 つ以上の割り当てパラメータを指定して、割り当てを制御します。各パラメータは、キーワード、等価記号（=）や大なり記号（>）、分割演算子（>>）、および括弧で囲んだ値から構成されます。ロードと実行に別々の位置が割り当てられる場合、キーワード LOAD の後ろにあるすべてのパラメータはロード割り当てに適用され、キーワード RUN の後ろにあるすべてのパラメータは実行割り当てに適用されます。使用可能な割り当てパラメータには次のものがあります。

Binding セクションを特定のアドレスに配置します。

```
.text:load = 0x1000
```

Memory セクションを、MEMORY 疑似命令で定義された、指定された名前（ROM など）または属性をもった範囲に配置します。

```
.text:load > ROM
```

Alignment セクションがアドレス境界から始まるように指定するには、align キーワードを使用します。

```
.text:align = 0x80
```

代入を含んでいる出力セクションも強制的に位置合わせするには、.（ドット）に align 式を代入します。たとえば、次のように指定すると、bar.obj が位置合わせされ、outsect が強制的に 0x40 バイト境界で位置合わせされます。

```
SECTIONS
{
   outsect:  {   bar.obj(.bss)
                 . = align(0x40);
             }
}
```

Splitting 分割演算子を使用して、セクションを配置できるメモリ範囲をリストします。

```
.text:>> ROM1|ROM2|ROM3
```

Blocking セクションが 2 つのアドレス境界の間に入っていなければならないことを指定するには、block キーワードを使用します。セクションが大きすぎる場合は、アドレス境界から開始されます。

```
.text: block(0x80)
```

Page 使用されるメモリ・ページを指定します（8.12 節「オーバーレイ・ページ」（8-59 ページ）を参照）。

```
.text:PAGE 0
```

ロード割り当ての場合は（通常はこの割り当てのみ）、単に大なり記号（>）を使用して、LOAD キーワードを省略することができます。

```
.text:> ROM             .text:{...} > ROM
.text:> 0x1000
```

パラメータを 2 つ以上使用する場合は、パラメータを次のように連続させることができます。

```
.text:> ROM align 16 PAGE 2
```

または、読みやすいように括弧を使用することもできます。

```
.text: load = (ROM align(16) page (2))
```

8.9.3.1 バインディング

セクション名の後ろにアドレスを続けると、出力セクションの開始アドレスを指定できます。

```
.text:0x1000
```

この例では、.text セクションが 1000h のバイト位置から始まるように指定しています。バインディング・アドレスは、24 ビットの定数でなければなりません。

出力セクションは構成メモリ（十分な空間があれば）のどこにでもバインドできますが、重複させることはできません。指定したアドレスにセクションをバインドするための十分な空間がない場合、リンカはエラー・メッセージを発行します。

注：バインディングと位置合わせ、または名前付きメモリを同時に使用することはできません。

位置合わせ、または名前付きメモリを使用する場合は、セクションをアドレスにバインドすることはできません。バインドしようとした場合、リンカはエラー・メッセージを発行します。

8.9.3.2 名前付きメモリ

セクションは、MEMORY 疑似命令で定義したメモリ範囲に割り当てることができます（8.8 節「MEMORY 疑似命令」（8-28 ページ）を参照）。この例では、メモリ範囲に名前を付け、セクションをその範囲内にリンクしています。

```
MEMORY
{
   ROM (RIX)  :origin = 0C00h,  length = 1000h
   RAM (RWIX) :origin = 0080h,  length = 1000h
}

SECTIONS
{
   .text  :               > ROM
   .data ALIGN(128) :     > RAM
   .bss   :               > RAM
```

この例では、リンカは .text を ROM という領域に入れています。.data と .bss の出力セクションは RAM に割り当てられています。セクションを名前付きメモリ範囲内で位置合わせすることもできます。この例では、.data セクションは RAM というメモリ範囲内の 128 バイト境界に位置合わせされます。

また、出力セクションを配置するメモリ範囲のリストを指定することもできます。たとえば、次の文は .text を ROM1、ROM2、または ROM3 に配置します。.text が ROM1 に収まらない場合は、ROM2 への配置を試行し、さらに ROM3 への配置を試行します。範囲は必ず指定した順序で試行されます。

```
.text:  > ROM1|ROM2|ROM3
```

同様に、セクションを特定の属性をもつメモリ領域にリンクできます。これを行なうには、メモリ名の代わりに一連の属性を（括弧に入れて）指定します。同じ MEMORY 疑似命令宣言を使用して、次のように指定できます。

```
SECTIONS
{
   .text:> (X)   /* .text --> executable memory      */
   .data:> (RI)  /* .data --> read or init memory    */
   .bss:> (RW)   /* .bss --> read or write memory    */
}
```

この例では ROM 領域と RAM 領域のどちらも X 属性を持っているので、.text 出力セクションを両方の領域にリンクできます。また、ROM と RAM のどちらにも R 属性と I 属性があるので、.data セクションは両方の領域に入れることができます。ただし、W 属性を持っているのは RAM だけなので、.bss 出力セクションは RAM にしか入れることができません。

セクションを名前付きメモリ範囲の中のどの位置に割り当てるかは、制御することができません。ただし、リンカは、まず下位のメモリ・アドレスを使用し、さらに、断片化をできるだけ避けようとします。これまでの例では、衝突する割り当てがないことを前提にして、.text セクションはアドレス 0 から開始します。セクションを特定のアドレスで開始しなければならない場合は、名前付きメモリの代わりにバインディングを使用します。

8.9.3.3 位置合わせとブロック化

リンカに対して、出力セクションを *n* バイト境界上のアドレスに入れるように指示できます。ここで、*n* は 2 の累乗です。たとえば、次の文は 128 バイト境界上にくるように .text を割り当てます。

```
.text: load = align(128)
```

ブロック化は位置合わせの制限を弱くしたもので、セクションはサイズ n のブロック内のどこかに割り当てられます。セクションがブロック・サイズより大きい場合は、セクションはブロックの境界上で始まります。位置合わせの場合と同様に、*n* は 2 の累乗でなければなりません。たとえば、次の文は、セクションが 1 つの 128 バイト・ページに入れられるか、またはページ上で始まるように .bss を割り当てます。

```
bss: load = block(0x80)
```

位置合わせまたはブロック化を単独で使用するか、またはメモリ領域と組み合わせて使用することができます。

8.9.3.4 入力セクションの指定

入力セクションの指定により、出力セクションを形成するために結合される入力ファイルのセクションを特定できます。出力セクションのサイズは、出力セクションを構成する入力セクションのサイズの合計に、指定された入力セクションの位置合わせまたはブロック化によって作成されたホールを加えたものです。リンカは、位置合わせまたはブロック化が入力セクションに指定されない場合、入力セクションを指定された順序で連結することによって結合します。

リンカ・コマンド・ファイルに単純なオブジェクト・ファイル・リファレンス（パス指定のない）を検出すると、リンカは、ファイルを事前に指定した入力ファイルに一致させようとします。リファレンスが入力ファイルのいずれかに一致しない場合、リンカは、カレント・ディレクトリのオブジェクト・ファイルを検索し、検出された場合はロードします。この機能を無効にするには、次のいずれかを実行します。

- ❑ パス指定を、リンカ・コマンド・ファイルのオブジェクト・ファイル・リファレンスに含めます。
- ❑ 入力ファイルの前に -I オプションを指定して、リンカを入力ファイルの検索パスにリンクさせます。

位置合わせまたはブロック化が入力セクションに指定されている場合、出力セクション内の入力セクションは次のように順序付けされます。

1） すべての位置合わせされたセクションについて、最大から最小

2） すべてのブロックされたセクションについて、最大から最小

3） その他すべての入力セクションについて、最大から最小

例 8–5 は、セクションの指定の最も一般的なタイプを示します。入力セクションがリストされていないことに注意してください。

例 8–5. セクション内容を指定する最も一般的な方法

```
SECTIONS
{
    .text:
    .data:
    .bss:
}
```

例 8–5 では、リンカは入力ファイルからすべての .text 出力セクションを取り出し、それらを .text 出力セクションに結合します。リンカは、入力ファイルで検出した順序で .text 入力セクションを連結します。リンカは、.data セクションや .bss セクションについても同じ操作を行います。任意の出力セクションに対して、この指定タイプを使用できます。

出力セクションを形成する入力セクションを、明示的に指定できます。各入力セクションは、ファイル名とセクション名を使用して特定されます。

```
SECTIONS
{
   .text :/* Build .text output section        */
   {
    f1.obj(.text)      /* Link .text section from f1.obj    */
    f2.obj(sec1)       /* Link sec1 section from f2.obj     */
    f3.obj             /* Link ALL sections from f3.obj     */
    f4.obj(.text,sec2) /* Link .text and sec2 from f4.obj   */
   }
}
```

入力セクションは、同じ名前である必要はなく、また形成される出力セクションと同じ名前である必要もありません。ファイルがセクションなしでリストされる場合、すべてのセクションは出力セクションに組み込まれています。追加の入力セクションの名前は出力セクションの名前と同じであるが、SECTIONS 疑似命令により明示的に指定されていない場合は、その入力セクションは自動的に出力セクションの最後にリンクされます。たとえば前の例で、リンカがさらに .text セクションを検出し、その .text セクションが SECTIONS 疑似命令のどこにも 指定されていない 場合は、リンカはこれらの追加のセクションを f4.obj(sec2) の後ろに連結します。

例 8–5 の指定は、実際は次の指定方法を簡略化したものです。

```
SECTIONS
{
   .text:{ *(.text) }
   .data:{ *(.data) }
   .bss:{ *(.bss)  }
}
```

*(.text) という指定は、すべての入力ファイルからの割り当てられていない *.text* セクションを意味します。この形式は次のような場合に便利です。

- ❑ 特定の名前をもつすべての入力セクションを出力セクションに組み込みたいが、出力セクションの名前が入力セクションの名前と違う場合
- ❑ リンカに対して、追加の入力セクションや括弧に入ったコマンドを処理する前に入力セクションを割り当てさせたい場合

次の例は、上記で説明した 2 つの場合を示しています。

```
SECTIONS
{
   .text   :  {
                       abc.obj(xqt)
                          *(.text)
              }
   .data   :  {
                       *(.data)
                       fil.obj(table)
                 }
}
```

この例では、.text 出力セクションには abc.obj ファイルの名前付きセクション xqt が組み込まれ、その後にすべての .text 入力セクションが続いています。.data セクションには、すべての .data 入力セクションが含まれ、その後に fil.obj ファイルの名前付きセクション table が続いています。この方法では、すべての割り当てられていないセクションが組み込まれます。たとえば、リンカが *(.text) を検出したときに .text 入力セクションの 1 つがすでに他の出力セクションに組み込まれている場合は、リンカはその 1 番目の .text 入力セクションを 2 番目の .text 出力セクションに組み込むことはできません。

8.9.4 アーカイブ・メンバを出力セクションに割り当てる方法

リンカ・コマンド・ファイル構文は、出力セクションに入力する 1 つ以上のオブジェクト・ライブラリのメンバを指定するメカニズムを提供するように拡張されました。つまり、リンカを使用すると、1 つ以上のアーカイブ・ライブラリのメンバを特定の出力セクションに割り当てることができます。このような割り当ての構文は次のとおりです。

```
SECTIONS
{
  .output_sec
  {
    [-l]lib_name<obj1 [obj2...objn]> (.sec_name)
  }
}
```

この構文では、*lib_name* がアーカイブ・ライブラリです。< > メカニズムが、メンバを選択するファイルがアーカイブでなければならないと指定するため、通常指定されたファイルのパス検索を行うことを意味する -I オプションの使用は、この構文では任意です。この例では、リンカは、常にパス検索を利用してアーカイブを検出します。ただし、指定した *lib_name* にパス情報が含まれている場合は、ライブラリ・ファイルの検索時にライブラリ・パス検索は実行されません。

-l オプションの詳細は、8.4.11 項「ファイル検索アルゴリズムの変更（-l オプション、-i オプション、および C55X_C_DIR/C_DIR 環境変数）」（8-12 ページ）を参照してください。

不等号括弧（<>）は、アーカイブ・メンバを指定するために使用されます。不等号括弧は、ひとつ以上のオブジェクト・ファイルを含み、スペースで区切られています。*sec_name* は、割り当てられるアーカイブ・セクションです。

次に例を示します。

```
SECTIONS
{
   .boot > BOOT1
   {
     /* This is the new support */
     -l rts55.lib<boot.obj> (.text)
     rts.lib< exit.obj strcpy.obj> (.text)
    }
   .rts > BOOT2
   {
     -l rts55.lib (.text)
   }
   .text > RAM
   {
      * (.text)
   }
}
```

この例では、boot.obj、exit.obj、および strcpy.obj は、ランタイムサポート・ライブラリから抽出され、.boot 出力セクションに配置されます。

参照されるランタイムサポート・ライブラリ・オブジェクトの残りの部分は、.rts 出力セクションに割り当てられます。アーカイブ・メンバまたはメンバのリストは、ライブラリ名の後に < > リストを介して指定できるようになりました。

その他すべての割り当てられない .text セクションは、.text セクション内に置かれることになります。

8.9.5 複数のメモリ領域を使用したメモリの配置

リンカを使用すると、メモリ領域の明示的なリストを、割り当てることのできる出力セクション内に指定できます。次の例を考えてみましょう。

```
MEMORY
{
   P_MEM1 :origin = 02000h,  length = 01000h
   P_MEM2 :origin = 04000h,  length = 01000h
   P_MEM3 :origin = 06000h,  length = 01000h
   P_MEM4 :origin = 08000h,  length = 01000h
}

SECTIONS
{
   .text  :{ } > P_MEM1 | P_MEM2 | P_MEM4
}
```

“|” 演算子は、複数のメモリ領域を指定するために使用します。.text の出力セクションは、メモリ領域が当てはまる最初のメモリ領域内にまるごと割り当てられるようになります。そのメモリ領域は指定された順にアクセスされます。この例では、リンカは P_MEM1 のセクションに割り当てようとします。この試みが失敗すると、リンカはそのセクションを P_MEM2 またはその他に配置しようとします。出力セクションがどの名前付きメモリ領域にも完全に割り当てられない場合、リンカはエラー・メッセージを発行します。

このタイプの SECTIONS 疑似命令の指定では、リンカは、本来割り当てられるメモリ領域の利用可能な空間を超えて成長する出力セクションを均一に取り扱います。リンカ・コマンド・ファイルを変更せずに、リンカに、このセクションを他のどこかの範囲に移動させるようにできます。

8.9.6 不連続なメモリ領域に出力セクションを自動的に分割する方法

リンカは、複数のメモリ領域に出力セクションを分割して、効果的な割り当てを実現します。>> 演算子を使用して、出力セクションが（必要であれば）指定されたメモリ領域の中に分割されることを示します。次に例を示します。

```
MEMORY
{
   P_MEM1 :origin = 02000h,  length = 01000h
   P_MEM2 :origin = 04000h,  length = 01000h
   P_MEM3 :origin = 06000h,  length = 01000h
   P_MEM4 :origin = 08000h,  length = 01000h
}

SECTIONS
{
  .text:{ *(.text) } >> P_MEM1 | P_MEM2 | P_MEM3 | P_MEM4
}
```

この例では、>> 演算子は .text 出力セクションがリストされたすべてのメモリ範囲に分割できることを示します。.text セクションが P_MEM1 内の利用可能なメモリを超えて成長する場合には、入力セクションの境界で分割されます。また、出力セクションの残りの部分は P_MEM2 | P_MEM3 | P_MEM4 に割り当てられます。

"|" 演算子は、複数のメモリ領域のリストを指定するために使用します。

また >> 演算子を使用して、出力セクションが単一のメモリ領域内で分割できることを示すようにできます。この機能は複数の出力セクションが同じメモリ領域内に割り当てられる必要があるときには便利ですが、ひとつの出力セクションによりメモリ領域が区分されるという制限があります。

次の例を考えてみましょう。

```
MEMORY
{
   RAM :origin = 01000h,  length = 08000h
}

SECTIONS
{
  .special:{ f1.obj(.text) } = 04000h
  .text:{ *(.text) } >> RAM
}
```

.special 出力セクションは、RAM メモリ領域の中央付近に割り当てられます。この出力セクションは 01000h から 04000h まで、および f1.obj(.text) の最後から 08000h までの RAM 内の 2 つの未使用範囲を残しておきます。.text セクション用のこの指定により、リンカは .special セクションの周囲にある .text セクションを分割し、および .special のどちらか側の RAM で利用可能な空間を使用します。

>> 演算子はまた、指定された属性の結合にマッチするすべてのメモリ領域の間にある出力セクションを分割するために使用できます。次に例を示します。

```
MEMORY
{
  P_MEM1 (RWX) :origin = 01000h,  length = 02000h
  P_MEM2 (RWI) :origin = 04000h,  length = 01000h
}

SECTIONS
{
  .text:{ *(.text) } >> (RW)
}
```

リンカは、出力セクションのすべてまたは一部を、SECTIONS 疑似命令に指定された属性にマッチする属性をもつあらゆるメモリ領域に割り当てようとします。

この SECTIONS 疑似命令は、次のものと同じ働きをします。

```
SECTIONS
{
  .text:{ *(.text) } >> P_MEM1 | P_MEM2
}
```

特定の出力セクションは分割してはいけません。

- C/C++ プログラム用の自動初期化テーブルを含む .cinit セクション
- C++ プログラム用のグローバル・コンストラクタのリストを含む .pinit セクション
- malloc() 関数およびランタイム・スタックがそれぞれ使用する C メモリ・プールの初期化されないセクションである .sysmem、.stack、および .sysstack セクション
- 単独のロード割り当ておよび実行割り当てをもつ出力セクション。出力セクションのロード時割り当てから実行時割り当てまでの出力セクションをコピーするコードは、その出力セクションで、分割することができません。
- 計算される式を含む入力セクションが指定された出力セクション。その式は、実行時に出力セクションを管理するプログラムで使用される記号を定義する場合があります。
- GROUP メンバである出力セクション。GROUP 疑似命令の目的は、GROUP メンバの出力セクションの連続割り当てを強制的に実行することです。
- 出力セクションに適用する START()、END()、または SIZE() 演算子をもつ出力セクション。これらの演算子は、セクションのロードまたは実行アドレスおよびサイズに関する情報を提供します。セクションが分割されている場合、演算子の完全性が損なわれる場合があります。
- アドレスおよびサイズ演算子の割り当てに使用する GROUP と UNION。

これらの状況で >> 演算子を使用する場合、リンカは警告を発行し、この演算子を無視します。

8.10 セクションのロード時および実行時アドレスの指定方法

ときにはコードをメモリのある領域にロードし、それを別の領域で実行する必要が生ずることがあります。たとえば、ROM ベースのシステムにパフォーマンスを左右するコードがあるとします。そのコードは ROM にロードしなければなりませんが、実行は RAM を使用した方がはるかに速くなります。

リンカを使用すると、この指定を簡単に行うことができます。SECTIONS 疑似命令を使用すると、リンカに同じセクションを 2 回割り当てるように指示できます。つまり、最初はロード・アドレスを、2 回目は実行アドレスを設定するように指示できます。次に例を示します。

```
.fir:load = ROM, run = RAM
```

ロード・アドレスには *load* キーワードを使用し、実行アドレスには *run* キーワードを使用します。

実行時の再配置の概要は、2.5 節「実行時の再配置」（2-17 ページ）を参照してください。

8.10.1 ロード・アドレスと実行アドレスの指定方法

ロード・アドレスにより、ローダはセクションの生データを配置する場所を決定します。そのセクションに対するすべての参照（ラベルの参照など）は実行アドレスを参照します。アプリケーションは、セクションをロード・アドレスから実行アドレスへコピーする必要があります。この作業は、別の実行アドレスを指定しても自動的には行われません。

1 つのセクションに対して 1 回の割り当て（ロードまたは実行）しか行わない場合は、そのセクションは 1 回だけ割り当てられ、ロードも実行も同じアドレスで行われます。2 つの割り当てを指定した場合は、そのセクションは同じサイズの 2 つの異なったセクションであるものとして割り当てられます。これは、両方の割り当てがそれぞれメモリ・マップ内の空間を占め、互いに、または他のセクションとオーバーレイできないことを意味します（UNION 疑似命令を使用すると、セクションをオーバーレイさせることができます。8.11.1 項「UNION 文によるセクションのオーバーレイ」（8-53 ページ）を参照してください）。

ロード・アドレスか実行アドレスに位置合わせやブロック化などの追加のパラメータを指定する場合は、該当するキーワードの後ろに指定してください。*load* キーワードの後ろにあって *run* キーワードが見つかるまでの割り当てに関係があるすべてのものは、ロード・アドレスに影響を与えます。run キーワードの後にあるすべてのものは、実行アドレスに影響を与えます。run を先に指定して、後から load を指定することもできます。括弧を使用すると読みやすくなります。

ロード・アドレスまたは実行アドレスのいずれかに位置合わせを指定すると、位置合わせはロード・アドレスと実行アドレスの両方に影響を与えます。ロード・アドレスと実行アドレスの両方に align オプションを指定すると、リンカは、セクションを両方のアドレスの最大値に再定義します。

8.10.2 初期化されないセクション

初期化されないセクション（.bss など）はロードされないので、有効なアドレスは実行アドレスだけです。リンカは、初期化されないセクションを 1 回だけ割り当てます。ユーザーが実行アドレスとロード・アドレスの両方を指定した場合、リンカは警告を発行し、ロード・アドレスは無視されます。1 つのアドレスのみを指定すると、リンカは、実行アドレスまたはロード・アドレスに関係なくそのアドレスを実行アドレスとして取り扱います。次の例では、初期化されないセクション用のロード・アドレスと実行アドレスを指定しています。

```
.bss:load = 0x1000, run = RAM
```

この例では警告が発行され、ロードは無視され、空間は RAM に割り当てられます。次のすべての例では、同じ結果が得られます。.bss セクションは、RAM に割り当てられます。

```
.bss:load = RAM
.bss:run = RAM
.bss:> RAM
```

8.10.3 リンク時のロード時アドレスおよびサイズの定義

現在、コード生成ツールは（低速）メモリのある領域にプログラム・コードをロードし、別の（高速）領域で実行する機能をサポートしています。これは、リンカ・コマンド・ファイルの出力セクションまたは GROUP のロード・アドレスおよび実行アドレスを個別に指定し、プログラム・コードが必要になる前に、プログラム・コードをロード領域から実行領域に移動する命令（コピー・コード）のシーケンスを実行することにより行われます。

この機能をもつシステムを設定する場合、いくつかの準備を行う必要があります。これらの準備の 1 つに、移動するプログラム・コードのサイズと実行時アドレスを決定する作業があります。これを実行する現行のメカニズムには、例 8–6 で示すとおり、コピー・コードで .label 疑似命令を使用する方法があります。

例 8–6.　.label を使用してロード時アドレスを定義する方法

```
  ; program code

        .sect   ".fir"
        .label  fir_src             ; load address of section
fir:                                ; run address of section
        <.fir section program code>

        .label  fir_end             ; load address of section end

        .text

; copying code
     MOV    #fir_src, AR1
     MOV    #fir
     RPT    #(fir_end - fir_src - 1)
         MOV    *AR1+, *CDP+
     CALL   fir
```

プログラム・コードのサイズおよびロード・アドレスを指定するこの方法には制限があります。1 つのソース・ファイル内にすべて組み込まれる個々の入力ファイルではよく動作しますが、プログラム・コード・セクションがいくつかのソース・ファイルに展開される場合はどうでしょうか。出力セクション全体をロード空間から実行空間にコピーする場合はどうでしょうか。

8.10.4 ドット（.）演算子が必ずしも動作しない理由

ドット（.）演算子は、出力セクション内の特定のアドレスを使用して、リンク時にシンボルを定義するために使用されます。これは PC のように解釈されます。現行のセクション内の現行オフセットがどのようなものであっても、ドット（.）に関連付けられた値です。SECTIONS 疑似命令内の出力セクションの指定について考えてみましょう。

```
outsect:
{
        s1.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.obj(.text)
        end_of_s2 = .;
}
```

この文は次の 3 つのシンボルを生成します。

- ❏ end_of_s1-s1.obj 内の .text の終了アドレス
- ❏ start_of_s2-s2.obj 内の .text の開始アドレス
- ❏ end_of_s2-s2.obj 内の .text の終了アドレス

位置合わせの結果として生成される s1.obj と s2.obj の間に埋め込みがあるとします。この場合、start_of_s2 は s2.obj 内の .text の実際の開始アドレスではありませんが、s2.obj 内の .text セクションを位置合わせするために埋め込みが必要になる前のアドレスです。これは、リンカが、ドット（.）演算子を現行 PC として解釈したことによります。また、ドット（.）演算子が、周囲の入力セクションとは無関係に評価されることにもよります。

上記の例の別の潜在的な問題に、end_of_s2 が出力セクションの終了時に必要となった埋め込みを考慮していない場合があります。end_of_s2 を出力セクションの終了アドレスとして安心して使用することはできません。この問題を回避する 1 つの方法として、問題の出力セクションのすぐ後にダミー・セクションを作成する方法があります。次に例を示します。

```
GROUP
{
        outsect:
        {
                start_of_outsect = .;
                ...
        }
        dummy:{ size_of_outsect = .- start_of_outsect; }
}
```

8.10.5 アドレス演算子とサイズ演算子

リンカ・コマンド・ファイル構文に、次の 6 つの新しい演算子が追加されました。

LOAD_START(*sym*) **START(*sym*)**	関連する割り当てユニットのロード時開始アドレスを使用して、*sym* を定義します。
LOAD_END(*sym*) **END(*sym*)**	関連する割り当てユニットのロード時終了アドレスを使用して、*sym* を定義します。
LOAD_SIZE(*sym*) **SIZE(*sym*)**	関連する割り当てユニットのロード時サイズを使用して、*sym* を定義します。
RUN_START(*sym*)	関連する割り当てユニットの実行時開始アドレスを使用して、*sym* を定義します。
RUN_END(*sym*)	関連する割り当てユニットの実行時終了アドレスを使用して、*sym* を定義します。
RUN_SIZE(*sym*)	関連する割り当てユニットのランタイム・サイズを使用して、*sym* を定義します。

注 : リンカ・コマンド・ファイル演算子の等価性

LOAD_START() と START() は等価であり、同様に LOAD_END()/END() と LOAD_SIZE()/SIZE() は等価です。

新しいアドレス演算子とサイズ演算子は、入力アイテム、出力セクション、GROUP、UNION など、いくつかの異なる種類の割り当てユニットに関連付けることができます。次のセクションでは、各ケースで演算子がどのように使用されるかを示すいくつかの例を説明します。

8.10.5.1 入力アイテム

SECTIONS 疑似命令内の出力セクションの指定について考えてみましょう。

```
outsect:
{
        s1.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.obj(.text)
        end_of_s2 = .;
}
```

この構文は、次のように START および END 演算子を使用して書き換えることができます。

```
outsect:
{
        s1.obj(.text) { END(end_of_s1) }
        s2.obj(.text) { START(start_of_s2), END(end_of_s2) }
}
```

元の例で、ドット（.）演算子を使用したように end_of_s1 と end_of_s2 の値は同じになります。しかし、start_of_s2 は 2 つの .text セクションの間に追加が必要な埋め込みの後で定義されます。ドット（.）演算子は、2 つの入力セクション間に埋め込みの挿入が必要になる前に start_of_s2 を定義することを忘れないでください。

これらの演算子を入力セクションに関連付けて使用する構文では、演算子のリストを囲むために 括弧 { } が必要です。リストの演算子は、リストのすぐ前に行われる入力アイテムに適用されます。

8.10.5.2 出力セクション

また、START、END、および SIZE 演算子は、出力セクションと関連付けることもできます。次に例を示します。

```
outsect:START(start_of_outsect), SIZE(size_of_outsect)
{
        <list of input items>
}
```

この例では、SIZE 演算子は size_of_outsect を定義して、課せられた位置合わせの要件に適合するために出力セクションで必要な埋め込みを実装します。

出力セクションで演算子を指定する構文は、括弧を使用して演算子リストを囲む必要はありません。演算子リストは、単に出力セクションの割り当て指定の一部として組み込まれます。

8.10.5.3 GROUP

GROUP 指定のコンテキストでは、次の例のように START および SIZE 演算子の別の使用方法があります。

```
GROUP
{
         outsect1: { ... }
         outsect2: { ... }
} load = ROM, run = RAM, START(group_start), SIZE(group_size);
```

この構文は、GROUP 全体を 1 つの場所にロードして、別の場所で実行する場合に便利です。コピー・コードは、コピーする場所とコピーする数量を指定するパラメータとして、group_start と group_size を使用できます。この結果、ソース・コードで .label を使用する必要がなくなります。

8.10.5.4 UNION

RUN_SIZE と LOAD_SIZE 演算子は、UNION のロード空間のサイズと UNION を実行する前にその構成がコピーされる空間のサイズを区別するメカニズムを提供します。次に例を示します。

```
UNION:run = RAM, LOAD_START(union_load_addr),
       LOAD_SIZE(union_ld_sz), RUN_SIZE(union_run_sz)
{
         .text1:load = ROM, SIZE(text1_size) { f1.obj(.text) }
         .text2:load = ROM, SIZE(text2_size) { f2.obj(.text) }
}
```

ここで、union_ld_sz は、共用体に配置されるすべての出力セクションの合計サイズと等しくなります。union_run_sz の値は、共用体の最大の出力セクションと等しくなります。これらのシンボルの両方とも、ブロック化または位置合わせの要件に基づいて埋め込みを実装します。

8.10.6 .label 疑似命令を使用してロード・アドレスを参照する方法

8.10.5 項で説明するアドレスおよびサイズ演算子の使用に代わる方法として、.label アセンブラ疑似命令を使用する方法があります。.label 疑似命令は、セクションのロード・アドレスを示す特殊シンボルを定義します。したがって、通常のシンボルが実行アドレスを基準に再配置されるのに対して、.label シンボルはロード・アドレスを基準に再配置されます。.label 疑似命令の詳細は、4-66 ページを参照してください。

例 8–7 は .label 疑似命令の使い方を示しています。

例 8–7. セクションを ROM から RAM にコピーする方法

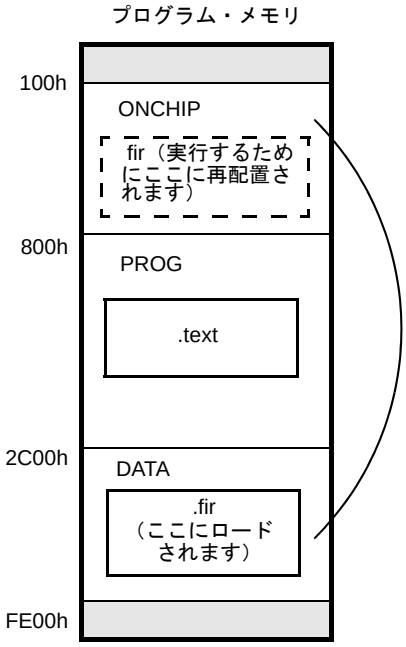
```
;  define a section to be copied from ROM to RAM
     .sect  ".fir"
     .label fir_src        ; load address of section
fir:; run address of section
    <code here>           ; code for the section

     .label fir_end        ; load address of section end

;  copy .fir section from ROM into RAM
     .text

      MOV #fir_src,AR1       ; get load address
      MOV BRC0,T1
      MOV T1,BRC1
      MOV #(fir_end - fir_src - 1),BRC0
      RPTB end
end   MOV *AR1+,*CDP+
      MOV BRC1,T1
      MOV T1,BRC0

;  jump to section, now in RAM
      CALL  fir
```

リンカ・コマンド・ファイル

```
/************************************************/
/*  PARTIAL LINKER COMMAND FILE FOR FIR EXAMPLE   */
/************************************************/

MEMORY
{
   ONCHIP :  origin = 000100h,  length = 000700h
   PROG   :  origin = 000800h,  length = 002400h
   DATA   :  origin = 002C00h,  length = 00D200h
}

SECTIONS
{
   .text: load = PROG
   .fir:  load = DATA, run ONCHIP
}
```

図 8-4 は、例 8–7 の実行時の様子を示しています。

図 8-4.　　例 8–7 の実行時の様子



8.11 UNION および GROUP 文の使用方法

GROUP と UNION という 2 つの SECTIONS 文を使用して、メモリを節約できます。UNION に数多くのセクションを指定すると、リンカは、それらのセクションを同じ実行アドレスに割り当てることができます。セクションを GROUP に配置すると、リンカは、グループ化された複数のセクションをメモリ内で連続して割り当てることができます。

8.11.1 UNION 文によるセクションのオーバーレイ

アプリケーションによっては、同じアドレスで複数のセクションを実行できるように割り当てることもあります。たとえば、プログラム実行のさまざまな段階で、オンチップ RAM にいくつかのルーチンを置きたい場合があります。また、同時にアクティブにはらないことが分かっているいくつかのデータ・オブジェクトに、1 つのメモリ・ブロックを共用させたい場合もあります。SECTIONS 疑似命令の中の UNION 文により、1 つの実行時アドレスにいくつかのセクションを割り当てることができます。

例 8–8 では、file1.obj と file2.obj からの .bss セクションが RAM の同じアドレスに割り当てられます。共用体は、その最大の構成要素と同じ大きさの空間をメモリ・マップ内で占めます。共用体の中の構成要素は、それぞれ独立したセクションのままです。これらのセクションは、単に 1 つのまとまりとして一緒に割り当てられます。

例 8–8. UNION 文

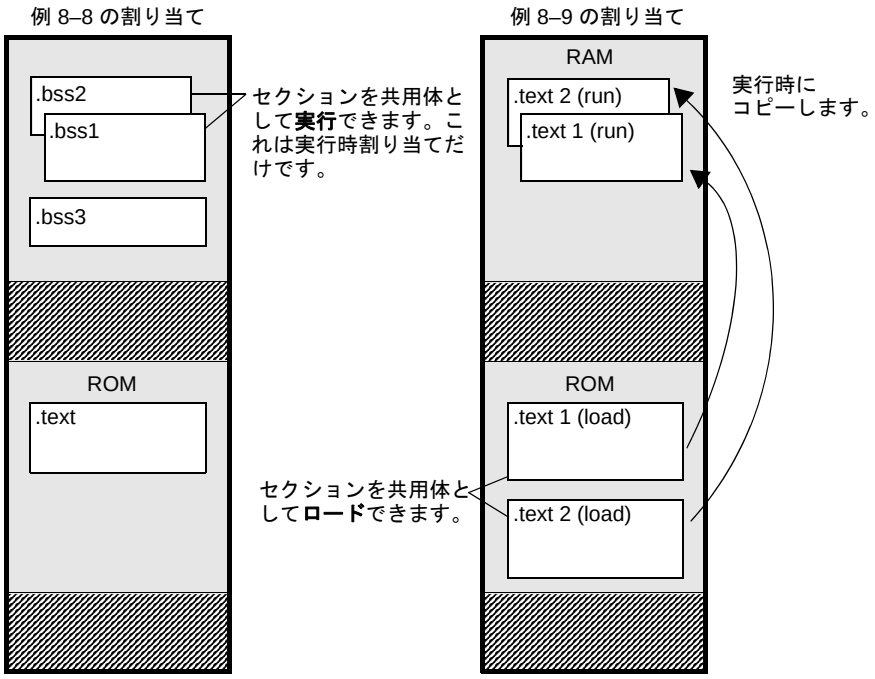
```
SECTIONS
{
   .text:                              load = ROM
   UNION:run = RAM
   {
      .bss1:{ file1.obj(.bss) }
      .bss2:{ file2.obj(.bss) }
   }
   .bss3:run = RAM { globals.obj(.bss) }
 }
```

1 つのセクションを共用体の一部として割り当てることは、その実行アドレスにのみ影響を与えます。どのような場合でも、ロード時に複数のセクションをオーバーレイさせることはできません。初期化されたセクションが共用体のメンバである場合は（初期化されたセクションには .text などの生データがあります）、そのロード割り当ては必ず別々に指定しなければなりません。次に例を示します。

例 8–9. UNION セクションに対する別々のロード・アドレス

```
   UNION:run = RAM
   {
      .text1:load = ROM, { file1.obj(.text) }
      .text2:load = ROM, { file2.obj(.text) }
   }
```

図 8-5.　例 8–8 と例 8–9 で定義されたメモリ割り当て



.text セクションにはデータが入っているので、これを共用体として実行することはできますがロードすることはできません。したがって、各セクションに専用のロード・アドレスが必要となります。共用体の中の初期化されたセクションのロード割り当てに失敗した場合、リンカは警告を発行し、構成メモリ内の可能な場所にロード空間を割り当てます。

初期化されないセクションはロードされず、ロード・アドレスも必要ありません。

UNION 文は実行アドレスの割り当てにのみ使用するので、共用体そのもののロード・アドレスを指定する必要はありません。割り当てに対して、共用体は初期化されないセクションとして取り扱われます。指定された割り当ては、すべて実行アドレスとみなされます。また、両方の割り当てが指定された場合、リンカは警告を発行し、ロード・アドレスを無視します。

共用体の位置合わせとブロック属性は、そのメンバの最大位置合わせとブロック属性です。

注 : UNION とオーバーレイ・ページは同じではありません

UNION 機能とオーバーレイ・ページ機能（8.12 節「オーバーレイ・ページ」（8-59 ページ）を参照）は、どちらもオーバーレイを取り扱うので同じように見えます。しかし、実際にはまったく異なるものです。UNION は、同じメモリ空間内で複数のセクションのオーバーレイを可能にします。一方、オーバーレイ・ページは複数のメモリ空間を定義します。ページ機能を使用して UNION の機能に近づけることは可能ですが、手間がかかります。

8.11.2 出力セクションをグループ化する方法

SECTIONS 疑似命令には、いくつかの出力セクションを連続して割り当てるための GROUP オプションがあります。たとえば、*term_rec* というセクションに .data セクション内のテーブルの終了レコードがあるとします。リンカを使用して、.data と term_rec を同時に割り当てるよう強制できます。

例 8–10. セクション・グループの割り当て

```
SECTIONS
{
   .text          /* Normal output section             */
   .bss           /* Normal output section             */
   GROUP 1000h :  /* Specify a group of sections       */
   {
      .data       /* First section in the group        */
      term_rec    /* Allocated immediately after .data */
   }
}
```

バインディング、位置合わせ、または名前付きメモリを使用して、単一の出力セクションの場合と同じ方法で GROUP を割り当てることができます。上の例では、GROUP はバイト・アドレス 1000h にバインドされています。これは、メモリ内で .data はバイト 1000h に割り当てられ、term_rec はその後に続いて割り当てられるということです。

GROUP の位置合わせとブロック属性は、そのメンバの最大位置合わせとブロック属性です。

GROUP の割り当てルーチンは、8.11.4 項で示す一貫性チェック規則に従います。

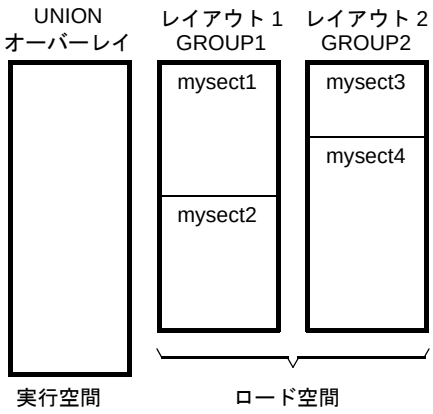
8.11.3 UNION と GROUP のネスト化

リンカを使用すると、SECTIONS 疑似命令とともに GROUP および UNION 文を任意にネストできます。GROUP および UNION 文をネストすることにより、同じメモリ空間の出力セクションのさまざまなレイアウトを表現できます。例 8–11 は、セクションの 2 つのオーバーレイを相互にグループ化する方法を示しています。

例 8–11. UNION 文と GROUP 文のネスト化

```
SECTIONS

   UNION:
   {
      GROUP
      {  mysect1
         mysect2
      }  load = ROM
      GROUP
      {
       mysect3
       mysect4

      }  load = ROM
   }  run = RAM
}
```

図 8-6. 例 8–11 に示すオーバーレイ・メモリ



例 8–11 では、上記のリンカ制御ファイルが与えられていて、そのリンカは次の割り当てを実行します。

- 4 つのセクション（mysect1、mysect2、mysect3、mysect4）は ROM メモリ領域の中で、固有に、およびオーバーラップしないロード・アドレスに指定されています。この指定は、各セクションに与えられている特定のロード割り当てにより決定されます。
- mysect1 および mysect2 セクションは、RAM 上で同じ実行アドレスが指定されています。
- mysect3 および mysect4 セクションは、RAM 上で同じ実行アドレスが指定されています。
- mysect1/mysect2 および mysect3/mysect4 の実行アドレスは、GROUP 文（位置合わせおよびブロックの制限に従って）により方向付けられて隣接して割り当てられています。

グループおよび共用体を参照するために、リンカの診断メッセージは次の注釈を使用します。

```
GROUP_n
UNION_n
```

この注釈では、「*n*」は（1 で始まる）連続番号であり、リンカ制御ファイルの中で、ネストすると見なさずに、グループおよび共用体の語彙の順序を表しています。グループと共用体は、それぞれ固有のカウンタを持っています。

8.11.4 割り付けルーチンの一貫性を調べる方法

リンカは、共用体、グループ、およびセクション用のロード割り当てと実行割り当ての一貫性を調べます。次の規則が使用されます。

- 実行割り当ては、トップレベルのセクション、グループ、または共用体（その他のグループや共用体でネストされていないセクション、グループ、または共用体）でのみ行われます。リンカは、トップレベルの構造体の実行アドレスを使用して、グループと共用体の中で、コンポーネントの実行アドレスを計算します。
- 8.11.1 項の説明にあるように、リンカは UNION 用のロード割り当てを受け入れません。
- 8.11.1 項の説明にあるように、リンカは初期化されないセクション用のロード割り当てを受け入れません。
- ほとんどのケースでは、初期化されたセクション用のロード割り当てを提供する必要があります。ただし、リンカは、ロード割り当てルーチンを既に定義したグループ内に配置された初期化されたセクション用のロード割り当てを受け入れません。

- ショートカットによると、グループ全体用にロード割り当てを指定して、そのグループ内にネストされたすべての初期化されたセクションまたはサブグループ用のロード割り当てを判別できます。ただし、ロード割り当ては次の条件がすべて真である場合にのみ、グループ全体用として受け入れます。
  - グループが初期化されている（たとえば、最低でも 1 つの初期化されたメンバを持っている）。
  - グループがロード割り当てルーチンをもつ他のグループ内でネストされていない。
  - グループが初期化されたセクションを含んでいる共用体を含まない。

  そのグループが初期化されたセクションを持つ共用体を含む場合、そのグループ内にネストされた初期化されたセクションのそれぞれにロード割り当てを指定する必要があります。次の例を考えてみましょう。

```
SECTIONS
{
  GROUP:  load = ROM, run = ROM
  {
   .text1:
   UNION:
   {
    .text2:
    .text3:
   }
  }
}
```

  グループ用に与えられたロード割り当ては、共用体の中の要素 .text2 および .text3 のためのロード割り当てを特別に指定していません。この場合、リンカは診断メッセージを発行して、これらのロード割り当てが明示的に指定されるように要求します。

8.12 オーバーレイ・ページ

ターゲット・システムの中には、メモリ空間の全部または一部がシャドウ・メモリによりオーバーレイされているメモリ構成を使用するものがあります。これにより、システムに、ハードウェア選択信号に従って物理メモリの異なる複数のバンクを 1 つのアドレス範囲に、およびこの範囲外にマップさせることができます。言い換えれば、物理メモリの複数のバンクは 1 つのアドレス範囲で互いにオーバーレイします。リンカを使用して、さまざまな出力セクションをこれらのバンクの各々にロードさせたり、ロード時にマップされないバンクにロードさせたりできます。

リンカは、オーバーレイ・ページを提供することにより、この機能をサポートします。各ページは、MEMORY 疑似命令を使用して別々に構成する必要がある 1 つのアドレス範囲を表します。SECTIONS 疑似命令を使用して、さまざまなページにマップされるセクションを指定できます。

8.12.1 MEMORY 疑似命令を使用してオーバーレイ・ページを定義する方法

リンカにとって、各オーバーレイ・ページは、完全な 24 ビットのアドレス指定可能な位置から構成されている完全に独立したメモリを表します。したがって、複数のセクションが異なるページにある場合には、それらを同じ（またはオーバーラップする）アドレスにリンクできます。

ページには、0 から始まる連続した番号が付けられます。PAGE オプションを指定しない場合、リンカはすべてのセクションを PAGE 0 に割り当てます。

たとえば、システムが、データ・メモリ空間の物理メモリの 2 つのバンク（PAGE 1 の A00h から FFFFh まで、および PAGE 2 の 0A00h から 2BFF までのアドレス範囲）を選択できるとします。一度に 1 つのバンクしか選択できませんが、別のデータを使用して、各バンクを初期化することができます。以降に、この構成を得るための MEMORY 疑似命令の使用方法を示します。

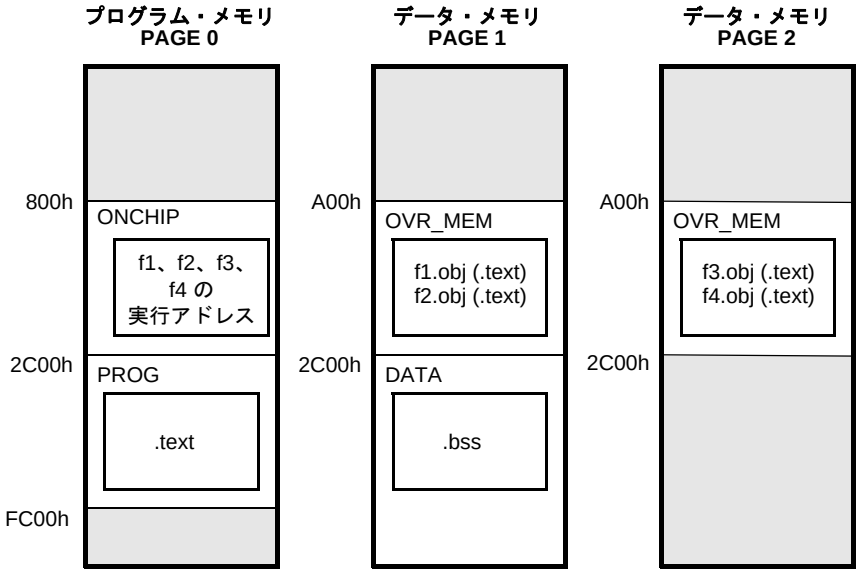
例 8–12. オーバーレイ・ページを使用する MEMORY 疑似命令

```
MEMORY
{
   PAGE 0 :  ONCHIP    :origin = 0800h,   length = 0240h
          :  PROG      :origin = 02C00h,  length = 0D200h
   PAGE 1 :  OVR_MEM   :origin = 0A00h,   length = 02200h
          :  DATA      :origin = 02C00h,  length = 0D400h
   PAGE 2 :  OVR_MEM   :origin = 0A00h,   length = 02200h
}
```

例 8–12 は、3 つの別々のアドレス範囲を定義しています。PAGE 0 は、オンチップ・プログラム・メモリの一領域と残りのプログラム・メモリ空間を定義します。PAGE 1 は、最初のオーバーレイ・メモリ領域と残りのデータ・メモリ空間を定義します。PAGE 2 は、データ空間の別のオーバーレイ・メモリ領域を定義します。両方の OVR_MEM 範囲は、同じアドレス範囲をカバーしています。これは、それぞれの範囲が異なるページにあって、別々のメモリ空間を表しているために可能となるのです。

図 8-7 は、例 8–12 に示す MEMORY 疑似命令と例 8–13 SECTIONS 疑似命令で定義されたオーバーレイ・ページを示しています。

図 8-7. 例 8–12 と例 8–13 で定義されているオーバーレイ・ページ



8.12.2 SECTIONS 疑似命令を使用してオーバーレイ・ページを使用する方法

例 8–12 に示す MEMORY 疑似命令を使用しているとします。さらに、コードには通常のセクションの他に、データ・メモリ空間にロードしたい 4 つのコード・モジュールが存在しているとします。ただし、これらのモジュールをプログラム・メモリ空間のオンチップ RAM 内で実行するとします。例 8–13 は、この場合の SECTIONS 疑似命令によるオーバーレイの使用方法を示しています。

例 8–13. 図 8-7 のオーバーレイ用の SECTIONS 疑似命令定義

```
SECTIONS
{
   UNION :  run = ONCHIP
   {
      S1 :  load = OVR_MEM PAGE 1
      {
         f1.obj (.text)
         f2.obj (.text)

      } LOAD_START(s1_load);
      S2 :  load = OVR_MEM PAGE 2
      {
         s2_load = 0A00h;
         s2_start = .;
         f3.obj (.text)
         f4.obj (.text)
         s2_length = .- s2_start;
      } LOAD_START(s2_load), SIZE(s2_length)

   } RUN_START(union_start)
   .text:  load = PROG PAGE 0
   .data:  load = PROG PAGE 0
   .bss :  load = DATA PAGE 1
}
```

4 つのコード・モジュールは f1、f2、f3、および f4 です。モジュール f1 および f2 は結合されて出力セクション S1 になり、f3 および f4 は結合されて出力セクション S2 になります。S1 および S2 用の PAGE 指定は、リンカにこれらのセクションを対応するページにリンクするように指示します。この結果、この 2 つのセクションはどちらもロード・アドレス A00h にリンクされますが、メモリ空間は異なります。プログラムがロードされるとき、ローダは、各セクションが適切なメモリ・バンクにロードされるようにハードウェアを構成できます。

出力セクション S1 および S2 は、オンチップ RAM 内に実行アドレスをもつ共用体の中に配置されます。アプリケーションは、実行時に、これらのセクションを転送してから実行する必要があります。セクション S1 の転送には、シンボル s1_load および s1_length を使用でき、セクション S2 の転送には s2_load および s2_length を使用できます。両方のセクションの実行アドレスは、UNION に RUN_START() 演算子を適用する union_start に割り当てられます。

ページ内では、通常の方法で、出力セクションをバインドしたり名前付きメモリ領域を使用したりできます。例 8–13 では、S1 次のように割り当てられています。

```
S1 : load = 01200h, page = 1  { . . . }
```

これは、S1 をページ 1 のアドレス 1200h にバインドします。ページをアドレスの修飾子として使用することもできます。次に例を示します。

```
S1 : load = (01200h PAGE 1)  { . . . }
```

セクションにバインディングや名前付きメモリ範囲を指定しない場合、リンカは（メモリ空間が 1 つだけの場合に通常行われるのと同様に）そのセクションを、割り当てることができる任意のページ内に割り当てます。たとえば、S2 は次のように指定することもできます。

```
S2 : PAGE 2  { . . . }
```

OVR_MEM はページ 2 の唯一のメモリなので、このセクションに = OVR_MEM を指定する必要はありません（ただし、指定しても構いません）。

8.12.3 ページ定義構文

例 8–12 と例 8–13 で示されたオーバーレイ・ページを指定するには、MEMORY 疑似命令で、次の構文を使用します。

MEMORY
{
    [**PAGE 0 :**]*name 1* [*(attr)*] **: origin =** *constant*, **length =** *constant;*
    [**PAGE** *n* **:**]*name n* [*(attr)*] **: origin =** *constant*, **length =** *constant;*
}

それぞれのページを定義するには、PAGE キーワードとページ番号を指定し、その後にコロンと、そのページに含まれるメモリ範囲リストを指定します。ボールド体の部分は、ここに示したとおりに入力する必要があります。メモリ範囲は、通常の方法で指定します。最大 255 ページまでのオーバーレイ・ページを定義できます。

それぞれのページは完全に独立したアドレス空間を表しているので、異なるページ上のメモリ範囲が同じ名前でも構いません。あるページの構成メモリが、他のページの構成メモリとオーバーラップしても構いません。ただし、1 つのページ内では、すべてのメモリ範囲は固有の名前を持っていなければならず、オーバーラップも許されません。

PAGE 指定の有効範囲外に一覧表示されたメモリ範囲は、デフォルトで PAGE 0 になります。次の例について考えてみます。

```
MEMORY
{           ROM   :org = 0h     len = 1000h
            EPROM :org = 1000h  len = 1000h
            RAM   :org = 2000h  len = 0E000h
   PAGE1:   XROM  :org = 0h     len = 1000h
            XRAM  :org = 2000h  len = 0E000h
}
```

ROM、EPROM、および RAM の各メモリ範囲は、すべて PAGE 0 上にあります（ページが指定されていないため）。XROM と XRAM は PAGE 1 上にあります。PAGE 1 上の XROM は PAGE 0 上の ROM と重なっており、PAGE 1 上の XRAM は PAGE 0 上の RAM に重なっていることに注意してください。

出力リンク・マップ（リンカ・オプション -m により取得）では、メモリ・モデルのリストにページ別にキーが付けられます。したがって、メモリ・モデルを正しく指定したかどうかを簡単に確認できます。また、出力セクションのリストには、各セクションがロードされるメモリ空間を示す PAGE の欄があります。

8.13 デフォルトの割り当てアルゴリズム

MEMORY および SECTIONS の疑似命令を使用すると、セクションを柔軟に作成し、結合し、割り当てることができます。ただし、指定されなかったメモリ・ロケーションやセクションは、リンカにより処理される必要があります。リンカは指定された範囲内で、デフォルトのアルゴリズムを使用してセクションを作成し、割り当てます。8.13.1 項「割り当てアルゴリズム」と 8.13.2 項「出力セクション作成の一般的な規則」で、デフォルトの割り当てについて説明します。

8.13.1 割り当てアルゴリズム

MEMORY および SECTIONS の疑似命令を使用しない場合、リンカは次の定義が指定されたものとして出力セクションを割り当てます。

例 8–14. TMS320C55x デバイス用のデフォルトの割り当て

```
MEMORY
{
    ROM (RIX)       :  origin = 0100h,     length = 0FEFFh
    VECTOR (RIX)    :  origin = 0FFFF00h, length = 0100h
    RAM (RWIX)      :  origin = 010100h,   length = 0FFFFh
}
SECTIONS
{
    .text       > ROM
    .switch     > ROM
    .const      > ROM
    .cinit      > ROM
    .vectors    > VECTOR
    .data       > RAM
    .bss        > RAM
    .sysmem     > RAM
    .stack      > RAM
    .sysstack   > RAM
    .cio        > RAM
}
```

すべての .text 入力セクションは連結され、実行可能出力ファイルの中で 1 つの .text 出力セクションを形成します。また、すべての .data 入力セクションは結合され、1 つの .data 出力セクションを形成します。.text および .data のセクションは、プログラム・メモリ空間の構成メモリの中に割り当てられます。すべての .bss セクションは結合され、.bss 出力セクションを形成します。この .bss セクションは、データ・メモリ空間上の構成メモリの中に割り当てられます。

SECTIONS 疑似命令を使用する場合、リンカは、デフォルト割り当てを一切行いません。割り当ては、8.13.2 項「出力セクション作成の一般的な規則」（8-65 ページ）に述べる、SECTIONS 疑似命令で指定した規則、および一般的なアルゴリズムに従って実行されます。

8.13.2 出力セクション作成の一般的な規則

出力セクションは、次の 2 つの方法のいずれかで作成できます。

規則 1 SECTIONS 疑似命令の定義の結果として作成できる。

規則 2 同じ名前の複数の入力セクションを結合して、SECTIONS 疑似命令の中で定義されていない 1 つの出力セクションにすることにより作成できる。

出力セクションが SECTIONS 疑似命令の結果として作成される場合（規則 1）、この定義は完全にセクションの内容を決定します（出力セクションの内容を定義する方法の例については、8.9 節「SECTIONS 疑似命令」（8-32 ページ）を参照してください）。

出力セクションは、入力セクションが SECTIONS 疑似命令により指定されていないときにも作成できます（規則 2）。この場合、リンカは同じ名前をもつすべての入力セクションを結合し、その名前をもつ出力セクションを作成します。たとえば、f1.obj と f2.obj の両方のファイルに Vectors という名前のセクションが入っており、SECTIONS 疑似命令はそれらのための出力セクションを定義していないとします。リンカは、入力ファイルに入っている 2 つの Vectors セクションを結合して Vectors という名前の 1 つの出力セクションを作成し、その出力セクションをメモリ内に割り当て、出力ファイルに組み込みます。

リンカは、すべての出力セクションの組成を決定した後、これらの出力セクションを構成メモリ内に割り当てる必要があります。MEMORY 疑似命令は、メモリのどの部分が構成済みであるかを指定します。MEMORY 疑似命令がない場合、リンカはデフォルトの構成を使用します。

リンカの割り当てアルゴリズムは、メモリの断片化を最小限に抑えようとします。これにより、メモリを効率的に使用でき、プログラムがメモリ内に収まる可能性も高くなります。このアルゴリズムは、次のとおりです。

1） 特定のバインディング・アドレスが指定された出力セクションは、メモリ内のそのアドレスに配置されます。

2） 特定の、名前付きメモリ範囲に含まれるか、またはメモリ属性の制約がある出力セクションは割り当てられます。それぞれの出力セクションは、名前が指定された領域内で最初に使用可能な空間に配置され、その際、必要であれば位置合わせが考慮されます。

3） 残りのセクションがある場合、これらのセクションは定義された順に割り当てられます。SECTIONS 疑似命令で定義されていないセクションは、検出された順に割り当てられます。それぞれの出力セクションは、最初に使用可能なメモリ空間に配置され、その際、必要であれば位置合わせが考慮されます。

リンカは、最後の .text（アプリケーションのオブジェクト・ファイルからのすべての .text セクションのグループ化）セクションの末尾に、パラレルではない NOP を埋め込むことに注意してください。

注 : PAGE オプション

PAGE オプションを使用して出力セクションのメモリ空間を明示的に指定しない場合、リンカはそのセクションを PAGE 0 に割り当てます。これは、PAGE 0 に空きがなく、他のページに空きがある場合でも同様です。PAGE 0 以外のページを使用するには、SECTIONS 疑似命令でページを指定する必要があります。

8.14 特別なセクションの型（DSECT、COPY、および NOLOAD）

出力セクションには、DSECT、COPY、および NOLOAD の特別な 3 つの型の指定を割り当てることができます。これらの型は、プログラムのリンク時およびロード時に、そのプログラムの取り扱い方に影響を及ぼします。セクションに型を割り当てるには、セクション定義の後に、型を（丸括弧で囲んで）指定します。次に例を示します。

```
SECTIONS
{
   sec1 2000h   (DSECT)    :{f1.obj}
   sec2 4000h   (COPY)     :{f2.obj}
   sec3 6000h   (NOLOAD)   :{f3.obj}
}
```

- DSECT 型は、次の性質を備えたダミー・セクションを作成します。
  - ダミー・セクションは、出力セクションのメモリ割り当てに含まれません。メモリを占有せず、メモリ・マップ・リストにも含まれません。
  - ダミー・セクションは、他の出力セクション、他の DSECT、および未構成メモリにオーバーレイできます。
  - ダミー・セクション内で定義したグローバル・シンボルは、通常どおり再配置されます。これらのシンボルは、DSECT が実際にロードされた場合にもつ値と同じ値を指定して、出力モジュールのシンボル・テーブル内に置かれます。これらのシンボルは、別の入力セクションから参照できます。
  - DSECT 内で未定義の外部シンボルが検出されると、指定したアーカイブ・ライブラリが検索されます。
  - ダミー・セクションの内容、再配置情報、および行番号情報は、出力モジュール内には配置されません。

  上記の例では、f1.obj に入っている各セクションは割り当てられませんが、すべてのシンボルは、これらのセクションがバイト・アドレス 2000h でリンクされているものとして再配置されます。他のセクションは、sec1 内のすべてのグローバル・シンボルを参照できます。

- COPY セクションは、その内容と関連情報が出力モジュールに書き込まれる点を除けば、DSECT セクションと同様のものです。TMS320C55x C コンパイラ用の初期化テーブルが入っている .cinit セクションは、RAM モデルでは、この属性を備えています。プラグマ IDENT によって生成された .comment セクションは、COPY セクションです。

- NOLOAD セクションは、通常の出力セクションと 1 つだけ異なる点があります。それは、セクションの内容、再配置情報、および行番号情報が出力モジュールに配置されない点です。リンカは、このセクション用に空間を割り当て、このセクションはメモリ・マップ・リストに含まれます。

8.15 リンク時のシンボルの割り当て方法

リンカ代入文を使用すると、外部（グローバル）シンボルを定義し、リンク時にこれらのシンボルに値を代入できます。この機能を使用して、変数やポインタを割り当てに依存した値に初期化できます。

8.15.1 代入文の構文

リンカでの代入文の構文は、C 言語の代入文の構文と同様のものです。

symbol	**=**	*expression;*	シンボルに式の値を代入します。
symbol	**+ =**	*expression;*	シンボルに式の値を加算します。
symbol	**- =**	*expression;*	シンボルから式の値を減算します。
symbol	*** =**	*expression;*	シンボルに式の値を乗算します。
symbol	**/ =**	*expression;*	式の値でシンボルを除算します。

シンボルは、外部で定義されなければなりません。外部が定義されない場合は、リンカは新しいシンボルを定義し、これをシンボル・テーブルに入れます。式は、8.15.3 項「代入式」（8-70 ページ）で定義されている規則に従っていなければなりません。代入文は、必ずセミコロンで終了しなければなりません。

リンカは、すべての出力セクションを割り当てた後に代入文を処理します。したがって、式にシンボルが含まれている場合、そのシンボルに使用されたアドレスは、そのシンボルの実行可能出力ファイル内のアドレスを反映します。

たとえば、プログラムが Table1 と Table2 という 2 つの外部シンボルにより表される 2 つのテーブルのいずれかからデータを読み取るとします。このプログラムは、現行のテーブルのアドレスとして cur_tab というシンボルを使用します。cur_tab は、必ず Table1 と Table2 のどちらかを指していなければなりません。これは、アセンブリ・コードの中でも実現できますが、テーブルを変更するには、プログラムを再度アセンブルする必要があります。これを避けるため、次のようにリンカの代入文を使用して、リンク時に cur_tab に値を割り当てることができます。

```
prog.obj            /* Input file                          */
cur_tab = Table1;   /* Assign cur_tab to one of the tables */
```

8.15.2 SPC をシンボルに割り当てる方法

1 つのドット（.）で表記する特殊記号は、割り当て時の SPC の現行値を表します。リンカの「.」記号は、アセンブラの $ 記号に似ています。「.」記号は、SECTIONS 疑似命令の中の代入文にのみ使用できます。この理由は、「.」は割り当てのときにだけ意味をもち、SECTIONS は割り当てプロセスを制御するからです（8.9 節「SECTIONS 疑似命令」（8-32 ページ）を参照してください）。「.」記号は、ひとつの出力セクションを定義する大括弧の外では使用できないことに注意してください。

「.」は、セクションの現行のロード・アドレスではなく、現行の実行アドレスを指します。

たとえば、プログラムで .data セクションの開始アドレスを知る必要があるとします。この場合、.global 疑似命令を使用することにより、プログラム内に Dstart という未定義の外部変数を作成できます。次に、値「.」を Dstart に代入します。

```
SECTIONS
{
   .text:    {}
   .data:    { Dstart = .; }
   .bss:     {}
}
```

これにより、Dstart は .data セクションの最初にリンクされるアドレスとして定義されます（Dstart は .data を割り当てる前に代入されます）。リンカは、Dstart へのすべての参照を再配置します。

「.」記号に値を代入する特別な種類の代入があります。この代入により、出力セクション内の SPC が調整され、2 つの入力セクションの間にホールが作成されます。ホールを作成するために「.」に代入される値は、実際に「.」で表されるアドレスではなく、セクションの先頭からの相対アドレスです。「.」への代入とホールについては、8.16 節「ホールの作成および埋め込みの方法」（8-73 ページ）に説明があります。

8.15.3 代入式

リンカ式には、次の規則が適用されます。

- ❑ 式には、グローバル・シンボル、定数、および表 8–1 に示す C 言語の演算子を使用できます。
- ❑ すべての数値は、ロング（32 ビット）整数として扱われます。
- ❑ 定数がリンカによって識別される方法は、アセンブラの場合と同じです。つまり、数値は接尾部（16 進数を表す H または h、8 進数を表す Q または q）がなければ、10 進数と見なされます。C 言語の接頭部（8 進数を表す 0、16 進数を表す 0x）も認識されます。16 進定数の最初の文字は、数字でなければなりません。2 進定数は使用できません。
- ❑ 式の中のシンボルは、自身のアドレスだけを値としてもつことができます。型のチェックは行われません。
- ❑ リンカ式は絶対的でも再配置可能でもかまいません。式に 1 つでも再配置可能シンボル（およびゼロ個以上の定数または絶対シンボル）が含まれていれば、この式は再配置可能です。それ以外の場合、式は絶対的です。シンボルに再配置可能な式の値が代入されていれば、このシンボルは再配置可能です。また、絶対的な式の値が代入されていれば、このシンボルは絶対的です。

リンカは、表 8–1 に示した C 言語の演算子を、この表の優先順位でサポートします。同じグループの演算子は、同じ優先順位を持ちます。表 8–1 に示した演算子のほか、リンカには *align* という演算子もあり、これを使用すると、シンボルを出力セクション内で *n* バイト境界に位置合わせできます（*n* は 2 の累乗です）。たとえば、

```
. = align(16);
```

という式は、現行のセクション内の SPC を、その次の 16 バイト境界に位置合わせします。align 演算子は現行の SPC の関数なので、「.」と同じ文脈、つまり、SECTIONS 疑似命令の中でのみ使用できます。

表 8–1. 式で使用できる演算子（優先順位）

シンボル	演算子	計算
+ - ~	単項プラス、単項マイナス、1 の補数	右から左へ
* / %	乗算、除算、剰余	左から右へ
+ -	加算、減算	左から右へ
<< >>	左シフト、右シフト	左から右へ
< <= > >=	小さい、小さいか等しい、大きい、大きいか等しい	左から右へ
!=, =[=]	等しくない、等しい	左から右へ
&	ビット単位 AND	左から右へ
^	ビット単位 OR	左から右へ
|	ビット単位 OR	左から右へ

注： 単項の +、-、および * は、2 項演算形式よりも優先順位が高くなります。

8.15.4 リンカで定義されるシンボル

リンカは、プログラムが実行時にセクションのリンク先を決めるために使用できるいくつかのシンボルを、自動的に定義します。これらのシンボルは外部シンボルなので、リンク・マップの中に示されます。これらのシンボルは、.global 疑似命令で宣言されていれば、どのアセンブリ言語モジュールからでもアクセスできます。これらのシンボルには、次のように値が代入されます。

.text .text 出力セクションの最初のアドレスが代入されます。
（実行可能コードの先頭を示します。）

etext .text 出力セクションの直後にある最初のアドレスが代入されます。
（実行可能コードの終わりを示します。）

.data .data 出力セクションの最初のアドレスが代入されます。
（初期化されたデータ・テーブルの先頭を示します。）

edata .data 出力セクションの直後にある最初のアドレスが代入されます。
（初期化されたデータ・テーブルの終わりを示します。）

.bss .bss 出力セクションの最初のアドレスが代入されます。
（初期化されないデータの先頭を示します。）

end .bss 出力セクションの直後にある最初のアドレスが代入されます。
（初期化されないデータの終わりを示します。）

8.15.5 C サポート用のみに定義されるシンボル（-c オプションまたは -cr オプション）

__STACK_SIZEは .stack セクションのサイズが代入されます。

__SYSSTACK_SIZEは .sysstack セクションのサイズが代入されます。

__SYSMEM_SIZEは .sysmem セクションのサイズが代入されます。

注 : .stack および .sysstack セクションの割り当て

.stack と .sysstack セクションは、同じ 64K ワードのデータ・ページに割り当てる必要があります。

8.16 ホールの作成および埋め込みの方法

リンカは、出力セクション内に、どこからもリンクされていない領域を作成する機能を備えています。このような領域を**ホール**と呼びます。特別なケースとして、初期化されないセクションをホールとして扱うこともできます。この節では、リンカがこのようなホールを処理する方法と、どのようにしてホール（および初期化されないセクション）に値を埋め込むことができるかについて説明します。

8.16.1 初期化されたセクションと初期化されないセクション

出力セクションは、次のどちらかの状態にあります。

- ❏ セクション全体に使用される生データを含んでいる
- ❏ 生データを含んでいない

生データをもつセクションを初期化されたセクションといいます。これは、オブジェクト・ファイルにそのセクションの実際のメモリ・イメージ内容が含まれることを意味します。このイメージは、セクションがロードされるとき、そのセクションの指定された開始アドレスのメモリ内にロードされます。.text および .data のセクションは、その中にアセンブルされたものがあれば、必ず生データを含んでいます。.sect アセンブラ疑似命令で定義された名前付きセクションも、生データを持っています。

デフォルトでは、.bss セクション、および .usect 疑似命令で定義されたセクション（初期化されないセクションです）には、生データは含まれません。これらのセクションはメモリ・マップ内の空間を占めますが、実際の内容はありません。初期化されないセクションは、通常、変数用の空間を RAM 内に予約します。オブジェクト・ファイル内では、初期化されないセクションは通常のセクション・ヘッダをもち、その中に定義されたシンボルをもつ場合があります。しかし、セクション内にメモリ・イメージは格納されません。

8.16.2 ホールの作成方法

初期化された出力セクションにホールを作成できます。ホールが作成されるのは、1 つの出力セクションの中で入力セクション相互の間に余分の空間を残すようにリンカに強制するときです。このようなホールが作成されるとき、リンカは上記の最初のガイドラインに従って、ホール用の生データを提供する必要があります。

ホールが作成されるのは、出力セクションの中だけです。出力セクション相互の間にも空間を残すことができますが、このような空間はホールではありません。SECTIONS 疑似命令を使用して、出力セクション相互間の空間を埋めたり初期化したりすることはできません。

出力セクション内にホールを作成するには、出力セクション定義の中で特別な種類のリンカ代入文を使用する必要があります。この代入文は、SPC（「.」で表される）に加算を行うか、より大きい値を代入するか、あるいは SPC をアドレス境界上に位置合わせするかによって SPC を修正します。代入文の演算子、式、および構文については、8.15 節「リンク時のシンボルの割り当て方法」（8-68 ページ）に説明があります。

次の例では、代入文を使用して出力セクション内にホールを作成します。

```
SECTIONS
{
   outsect:
   {
      file1.obj(.text)
      . += 100h;  /* Create a hole with size 100h bytes */
      file2.obj(.text)
      . = align(16);  /* Create a hole to align the SPC */
      file3.obj(.text)
   }
}
```

出力セクション outsect は、次のように作成されます。

- ❏ file1.obj に入っている .text セクションがリンクされます。
- ❏ リンカは 256 バイトのホールを作成します。
- ❏ file2.obj に入っている .text セクションがホールの後にリンクされます。
- ❏ リンカは SPC を 16 バイト境界上に位置合わせすることによって、別のホールを作成します。
- ❏ 最後に、file3.obj に入っている .text セクションがリンクされます。

セクション内で「.」記号に代入されたすべての値は、セクション内の相対アドレスを参照します。リンカは「.」記号への代入を、セクションが（バインディング・アドレスが指定された場合でも）アドレス 0 から始まっているものとして処理します。たとえば、この例の . = align(16) という文について考えてみます。この文は、file3.obj の .text を outsect 内の 16 バイト境界に位置合わせする効果があります。outsect が最終的に、位置合わせされていないアドレス上から始まるように割り当てられた場合、file3.obj の .text も位置合わせされません。

「.」記号は、セクションの現行のロード・アドレスではなく、現行の実行アドレスを指すことに注意してください。

「.」をデクリメントする式は、不正です。たとえば、「.」への代入に -= 演算子を使用しても無効です。「.」への代入に最もよく使用される演算子は、+= と align です。

ある出力セクションに含まれるすべての入力セクションが 1 種類（.text など）の場合、次の文を使用して、出力セクションの始めまたは終わりにホールを作成できます。

```
.text:   {  .+= 100h; }    /* Hole at the beginning */
.data:   {
            *(.data)
            . += 100h; }   /* Hole at the end       */
```

出力セクションにホールを作成するもう 1 つの方法は、初期化されないセクションを初期化されたセクションに結合して、1 つの出力セクションにする方法です。この場合、リンカは初期化されないセクションをホールとして扱い、そのセクションにデータを提供します。次の例は、この方法を示しています。

```
SECTIONS
{
     outsect:
     {
     file1.obj(.text)
     file1.obj(.bss)          /* This becomes a hole */
     }
}
```

.text セクションには生データがあるので、すべての outsect には生データが入っていなければなりません（規則 1)。したがって、初期化されない .bss セクションはホールになります。

初期化されないセクションがホールになるのは、それらのセクションを初期化されたセクションに結合した場合だけです。複数の初期化されないセクションをまとめてリンクしても、結果として生成される出力セクションは、やはり初期化されないセクションです。

8.16.3 ホールの埋め込み方法

初期化された出力セクションにホールが存在する場合、リンカはそのホールを埋める生データを提供しなければなりません。リンカは、16 ビットの埋め込み値を使用してホールを埋めます。この値はホール全体がいっぱいになるまで、メモリを介してコピーされます。リンカは、次のようにして埋め込み値を決定します。

1) 初期化されないセクションと初期化されたセクションを結合してホールが作成される場合、初期化されないセクションの埋め込み値を指定できます。次のように、セクション名の後に = 記号と 16 ビット定数を指定してください。

```
SECTIONS
{
   outsect:
   {
      file1.obj(.text)
      file2.obj(.bss) = 00FFh    /* Fill this hole */
   }                             /* with 0FFh      */
}
```

2) 次のように出力セクションの定義の後に埋め込み値を指定することでも、その出力セクション内のすべてのホールの埋め込み値を指定できます。

```
SECTIONS
{
   outsect:fill = 0FF00h /* fills holes with 0FF00h  */
   {
      . += 10h;          /* This creates a hole      */
      file1.obj(.text)
      file1.obj(.bss)    /* This creates another hole*/
   }
}
```

3) ホールの初期値を指定しない場合、リンカは -f オプションで指定された値でホールを埋めます。たとえば、コマンド・ファイル link.cmd に次のような SECTIONS 疑似命令が入っているとします。

```
SECTIONS
{
   .text:{ .= 100; }      /* Create a 100-byte hole */
}
```

ここで、次のように -f オプションを指定してリンカを起動します。

```
lnk55 -f 0FFFFh link.cmd
```

これにより、ホールは 0FFFFh で埋められます。

4) リンカを起動するときに -f オプションを使用しない場合、リンカは 0 でホールを埋めます。

ホールが作成され、初期化された出力セクション内でホールが埋められた場合、そのホールは必ず、リンカが埋め込みに使用した値とともにリンク・マップ内に示されます。

8.16.4 初期化されないセクションの明示的な初期化

初期化されないセクションがホールになるのは、初期化されたセクションに結合された場合だけです。初期化されないセクションを互いに結合しても、結果として生成される出力セクションは、初期化されないセクションのままです。

しかし、SECTIONS 疑似命令の中で明示的に埋め込み値を指定することにより、リンカに対して初期化されないセクションを初期化させるように強制できます。これにより、セクション全体が生データ（埋め込み値）をもつようになります。次に例を示します。

```
SECTIONS
{
   .bss:fill = 1234h      /* Fills .bss with 1234h */
}
```

注 : セクションの埋め込み

セクションに埋め込みを行うと、たとえ埋め込み値が 0 でも、出力ファイル内のセクション全体を埋める生データが生成されます。このため、大きなセクションやホールに埋め込み値を指定すると、出力ファイルは非常に大きくなります。

8.17 リンカが生成するコピー・テーブル

リンカは、次の機能を有効にするリンカ・コマンド・ファイル構文の拡張子をサポートします。

- ❑ ブート時に、ロード空間から実行空間へのオブジェクトのコピーを簡単にする
- ❑ 実行時に、メモリ・オーバーレイの管理を簡単にする
- ❑ 独立したロード・アドレスと実行アドレスをもつ GROUP と出力セクションを分割できるようにする

8.17.1 現行のブートロード・アプリケーション開発プロセス

一部の組み込み型アプリケーションでは、アプリケーションが実際にブート時にメイン実行スレッドを開始する前に、コードおよびデータ、またはそのいずれかを、ある場所から別の場所にコピーまたはダウンロードする必要があります。たとえば、アプリケーションが FLASH メモリにコードおよびデータ、またはそのいずれかを保持している場合、アプリケーションは実行を開始する前に、これらをオンチップ・メモリにコピーする必要があります。

このようなアプリケーションを開発する方法の 1 つは、ブート時に FLASH からオンチップ・メモリに移動する必要のある、次のコードまたはデータの各ブロックの 3 つの要素を含むアセンブリ・コードにコピー・テーブルを作成することです。

- ❑ ロードする位置（ロード・ページ ID とアドレス）
- ❑ 実行する位置（実行ページ ID とアドレス）
- ❑ サイズ

このようなアプリケーションを開発するプロセスは、次のようになります。

1) 別個のロードおよび実行位置をもつ各セクションのロードおよび実行アドレスを含む .map ファイルを生成するアプリケーションを作成する。
2) コピー・テーブル （ブート・ローダが使用）を編集して、ロードおよび実行アドレスとともに、ブート時に移動する必要のあるコードまたはデータの各ブロックのサイズを修正する。
3) 更新されたコピー・テーブルが組み込まれたアプリケーションをもう一度作成する。
4) アプリケーションを実行する。

このプロセスは、コピー・テーブルを保守するために大きな負担となります（手作業で実行する場合に劣らず）。1 つ 1 つのコードまたはデータをアプリケーションに追加し、削除するたびに、コピー・テーブルの内容を最新の状態に維持するために、このプロセスを繰り返す必要があります。

8.17.2 代替方法

リンカ・コマンド・ファイル構文の一部である LOAD_START()、RUN_START()、および SIZE() 演算子を使用すると、この保守負担の一部を軽減することができます。たとえば、.map ファイルを生成するアプリケーションを作成する代わりに、リンカ・コマンド・ファイルで次のように記述することができます。

```
SECTIONS
{
   .flashcode:{ app_tasks.obj(.text) }
      load = FLASH, run = PMEM,
      LOAD_START(_flash_code_ld_start),
      RUN_START(_flash_code_rn_start),
      SIZE(_flash_code_size)

   ...
}
```

この例では、LOAD_START()、RUN_START()、および SIZE() 演算子は、次の 3 つのシンボルを作成するように、リンカに指示します。

シンボル	説明
_flash_code_ld_start	.flashcode セクションのロード・アドレス
_flash_code_rn_start	.flashcode セクションの実行アドレス
_flash_code_size	.flashcode セクションのサイズ

これらのシンボルはコピー・テーブルから参照することができます。コピー・テーブルの実際のデータは、アプリケーションがリンクされるたびに自動的に更新されます。この方法により、8.17.1 項で説明したプロセスの手順 1 が不要になります。

コピー・テーブルの保守は著しく軽減されますが、それでもコピー・テーブルの内容とリンカ・コマンド・ファイルで定義されるシンボルとの同期を維持するための負荷が発生します。リンカが自動的にブート・コピー・テーブルを生成すれば理想的です。これにより、アプリケーションを 2 度作成しなくても済むようになり、同時にブート・コピー・テーブルの内容を明示的に管理する必要がなくなります。

LOAD_START()、RUN_START()、および SIZE() 演算子の詳細は、8.10.5 項「アドレス演算子とサイズ演算子」（8-48 ページ）を参照してください。

8.17.3 オーバーレイ管理の例

実行時に管理する必要のあるメモリ・オーバーレイを含むアプリケーションについて考えてみましょう。メモリ・オーバーレイは、例 8–15 で説明されているとおり、リンカ・コマンド・ファイルの UNION を使用して定義します。

例 8–15. メモリ・オーバーレイの UNION の使用

```
SECTIONS
{
   ...

   UNION
   {
      GROUP
      {
         .task1:{ task1.obj(.text) }
         .task2:{ task2.obj(.text) }

      } load = ROM, LOAD_START(_task12_load_start), SIZE(_task12_size)

      GROUP
      {
         .task3:{ task3.obj(.text) }
         .task4:{ task4.obj(.text) }

      } load = ROM, LOAD_START(_task34_load_start), SIZE(_task_34_size)

   } run = RAM, RUN_START(_task_run_start)

 ...
}
```

アプリケーションは、実行時にメモリ・オーバーレイの内容を管理する必要があります。つまり、.task1 または .task2 からのサービスが必要の場合、アプリケーションは最初に .task1 と .task2 がメモリ・オーバーレイに常駐していることを確認する必要があります。.task3 と .task4 についても同様です。

実行時に .task1 と .task2 の ROM から RAM へのコピーに影響を与えるために、アプリケーションは、まずタスクのロード・アドレス（_task12_load_start）、実行アドレス（_task_run_start）、およびサイズ（_task12_size）にアクセスする必要があります。この情報は、実際のコード・コピーの実行に使用されます。

8.17.4 リンカによるコピー・テーブルの自動生成

リンカは、次の機能を実行可能にするリンカ・コマンド・ファイル構文の拡張子をサポートします。

- ❑ アプリケーションの実行時のある時点でロード空間から実行空間にコピーする必要のあるオブジェクト・コンポーネントを特定する
- ❑ コピーする必要のあるコンポーネントの（少なくとも）ロード・アドレス、実行アドレス、およびサイズを含むコピー・テーブルを自動的に生成するようにリンカに指示する
- ❑ リンカが生成するコピー・テーブルのアドレスを提供するように指定したシンボルを生成するようにリンカに指示する。たとえば、例 8–15 は 例 8–16 に示すように記述することができます。

例 8–16. リンカが生成するコピー・テーブルのアドレスの作成

```
SECTIONS
{
   ...

   UNION
   {
      GROUP
      {
         .task1: { task1.obj(.text) }
         .task2: { task2.obj(.text) }

      } load = ROM, table(_task12_copy_table)

      GROUP
      {
         .task3: { task3.obj(.text) }
         .task4: { task4.obj(.text) }

      } load = ROM, table(_task34_copy_table)

   }  run = RAM

   ...
}
```

リンカ・コマンド・ファイルの例 8–16 で SECTIONS 疑似命令を使用することにより、リンカは _task12_copy_table と _task34_copy_table という 2 つのコピー・テーブルを生成します。各コピー・テーブルは、コピー・テーブルに関連付けられた GROUP のロード・ページ ID、実行ページ ID、ロード・アドレス、実行アドレス、およびサイズを提供します。この情報は、リンカが生成するシンボル _task12_copy_table と _task34_copy_table を使用して、アプリケーション・ソース・コードからアクセスできます。これらのシンボルは、それぞれ 2 つのコピー・テーブルのアドレスを提供します。

この方法を使用することにより、コピー・テーブルの作成と保守について考慮する必要がなくなります。この値を、コピー・テーブルを処理し、実際のコピーに影響を与える汎用コピー・ルーチンに渡すことにより、C/C++ またはアセンブリ・ソース・コードで、リンカが生成したコピー・テーブルのアドレスを参照することができます。

8.17.5 table() 演算子

table() 演算子を使用して、コピー・テーブルを作成するように、リンカに指示することができます。table() 演算子は、出力セクション、GROUP、または UNION のメンバに適用できます。特定の table() 指定用に生成されたコピー・テーブルは、table() 演算子の引数として提供されるように指定したシンボルを介してアクセスできます。リンカは、この名前をもつシンボルを作成し、シンボルの値としてコピー・テーブルのアドレスに割り当てます。これにより、コピー・テーブルは、リンカが生成したシンボルを使用するアプリケーションからアクセスできるようになります。

与えられた UNION のメンバに適用する各 table() 指定は、固有の名前が付いている必要があります。table() が GROUP に適用される場合、GROUP のメンバが table() 指定によりマーク付けされることはありません。リンカは、これらの規則の違反を検出し、違反している table() 指定の使用を無視するように、警告としてレポートします。リンカは、間違った table() 演算子の指定で、コピー・テーブルを生成することはありません。

8.17.6 ブート時のコピー・テーブル

リンカは、ブート時のコピー・テーブルを作成するために使用できる特殊なコピー・テーブル名 BINIT（または binit）をサポートします。たとえば、8.17.2 項で説明されているブートロード・アプリケーション用のリンカ・コマンド・ファイルを、次のように書き換えることができます。

```
SECTIONS
{
   .flashcode:{ app_tasks.obj(.text) }
   load = FLASH, run = PMEM,
      table(BINIT)
   ...
}
```

この例では、リンカは、ブート時にロードする位置から実行する位置にコピーする必要のあるすべてのオブジェクト・コンポーネントのリストを含み、特殊なリンカが生成したシンボル _ _ _binit_ _ を介してアクセスできるコピー・テーブルを作成します。リンカ・コマンド・ファイルが table(BINIT) を使用するように指定されていない場合、_ _ _binit_ _ シンボルには、特定のアプリケーションにブート時のコピー・テーブルが存在しないことを示す -1 の値が指定されます。

table(BINIT) 指定は、出力セクション、GROUP、または UNION のメンバに適用できます。UNION のコンテキストでは、UNION の 1 つのメンバだけを table(BINIT) で指定できます。GROUP に適用される場合、その GROUP のメンバが table(BINIT) でマーク付けされることはありません。リンカは、これらの規則の違反を検出し、違反している table(BINIT) 指定の使用を無視するように、警告としてレポートします。

8.17.7 table() 演算子を使用するオブジェクト・コンポーネントの管理

同時に管理する必要のある複数のコードがある場合、複数の異なるオブジェクト・コンポーネントに同じ table() 演算子を適用することができます。さらに、特定のオブジェクト・コンポーネントを複数の方法で管理したい場合は、複数の table() 演算子を適用することができます。例 8–17 のリンカ・コマンド・ファイルの抜粋について考えてみましょう。

例 8–17. オブジェクト・コンポーネントを管理するリンカ・コマンド・ファイル

```
SECTIONS
{
   UNION
   {
      .first:{ a1.obj(.text), b1.obj(.text), c1.obj(.text) }
            load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)

      .second:{ a2.obj(.text), b2.obj(.text) }
            load = EMEM, run = PMEM, table(_second_ctbl)
   }

   .extra:load = EMEM, run = PMEM, table(BINIT)

   ...
}
```

この例では、ブート時に BINIT コピー・テーブルを処理する間に、出力セクション .first と .extra が外部メモリ（EMEM）からプログラム メモリ（PMEM）にコピーされます。アプリケーションがメイン・スレッドの実行を開始すると、アプリケーションは _first_ctbl と _second_ctbl という名前の 2 つのオーバーレイ・コピー・テーブルを使用して、オーバーレイの内容を管理することができます。

8.17.8 コピー・テーブルの内容

リンカが生成するコピー・テーブルを使用するためには、コピー・テーブルの内容について理解する必要があります。この情報は、リンカにより自動的に生成されるコピー・テーブルのデータ構造の C ソース表現が含まれる、新しいランタイム・サポート・ライブラリのヘッダ・ファイル cpy_tbl.h にあります。

例 8–18 は C55x コピー・テーブルのヘッダ・ファイルを示します。

例 8–18. TMS320C55x cpy_tbl.h ファイル

```
/*****************************************************************************/
/* cpy_tbl.h  vxxxxx                                                         */
/* Copyright (c) 2003 Texas Instruments Incorporated                         */
/*                                                                           */
/* Specification of copy table data structures which can be automatically    */
/* generated by the linker (using the table() operator in the LCF).          */
/*****************************************************************************/
#ifndef _CPY_TBL
#define _CPY_TBL

#include <stdlib.h>

/*****************************************************************************/
/* Copy Record Data Structure                                                */
/*****************************************************************************/
typedef struct copy_record
{
   unsigned long load_loc;
   unsigned long run_loc;
   unsigned long size;
} COPY_RECORD;

/*****************************************************************************/
/* Copy Table Data Structure                                                 */
/*****************************************************************************/
typedef struct copy_table
{
   unsigned short rec_size;
   unsigned short num_recs;
   COPY_RECORD    recs[1];
} COPY_TABLE;

/*****************************************************************************/
/* Prototype for general purpose copy routine.                               */
/*****************************************************************************/
extern void copy_in(COPY_TABLE *tp);

/*****************************************************************************/
/* Prototypes for I/O aware copy routines used in copy_in().                 */
/*****************************************************************************/
extern void cpy_io_to_io(void *from, void *to, size_t n);
extern void cpy_io_to_mem(void *from, void *to, size_t n);
extern void cpy_mem_to_io(void *from, void *to, size_t n);

#endif /* !_CPY_TBL */
```

コピーのマーク付けされている各オブジェクト・コンポーネントに対して、リンカは COPY_RECORD オブジェクトを作成します。各 COPY_RECORD には、少なくとも次のオブジェクト・コンポーネントの情報が含まれています。

- ロード・ページ ID
- 実行ページ ID
- ロード・アドレス
- 実行アドレス
- サイズ

ロード・ページ ID とロード・アドレスは、COPY_RECORD の load_loc フィールドに結合されます。同様に、実行ページ ID と実行アドレスは、COPY_RECORD の run_loc フィールドに結合されます。どちらの場合も、ページ ID は load_loc と run_loc フィールドの最上位 8 ビットにエンコードされます。実際のロード・アドレスと実行アドレスは、それぞれ load_loc と run_loc フィールド最下位 3 バイトにエンコードされます。ページ ID が 0 の場合は、表示されるアドレスが標準の C55x メモリを指していることを示します。ページ ID が 0 以外の場合は、表示されるアドレスが I/O メモリの位置を指していることを示します。

リンカは、同じコピー・テーブルに関連付けられているすべての COPY_RECORD を COPY_TABLE オブジェクトに収集します。COPY_TABLE オブジェクトには、特定の COPY_RECORD のサイズ、テーブル内の COPY_RECORD の数、およびテーブル内の COPY_RECORD の配列が含まれています。たとえば、8.17.6 項の BINIT の例では、.first と .extra 出力セクションは BINIT コピー・テーブルにそれぞれ独自の COPY_RECORD エントリを持っています。BINIT コピー・テーブルは、次のようになります。

```
COPY_TABLE _ _binit_ _ = { 12, 2,
                           { <load page id and address of .first>,
                             <run page id and address of .first>,
                             <size of .first> },
                           { <load page id and address of .extra>,
                             <run page id and address of .extra>,
                             <size of .extra> } };
```

8.17.9 汎用コピー・ルーチン

また、例 8–18 の cpy_tbl.h ファイルにも、ランタイムサポート・ライブラリの一部として提供される汎用コピー・ルーチン copy_in() のプロトタイプが含まれています。copy_in() ルーチンは、単一の引数、すなわちリンカが生成するコピー・テーブルのアドレスをとります。次に、ルーチンはコピー・テーブルのデータ・オブジェクトを処理し、コピー・テーブルで指定された各オブジェクト・コンポーネントのコピーを実行します。

copy_in() 関数の定義は、例 8–19 で示す cpy_tbl.c ランタイムサポート・ソース・ファイルで提供されます。

例 8–19. ランタイムサポート cpy_tbl.c ファイル

```
/*****************************************************************************/
/* cpy_tbl.c                                                                 */
/*                                                                           */
/* Copyright (c) 2003 Texas Instruments Incorporated                         */
/*                                                                           */
/* General purpose copy routine.Given the address of a linker-generated    */
/* COPY_TABLE data structure, effect the copy of all object components      */
/* that are designated for copy via the corresponding LCF table() operator. */
/*                                                                           */
/*****************************************************************************/
#include <cpy_tbl.h>
#include <string.h>

/*****************************************************************************/
/* Static Function Prototypes for I/O aware copy routines.                   */
/*****************************************************************************/
static void cpy_io_to_io(void *from, void *to, size_t n);
static void cpy_io_to_mem(void *from, void *to, size_t n);
static void cpy_mem_to_io(void *from, void *to, size_t n);

/*****************************************************************************/
/* COPY_IN()                                                                 */
/*****************************************************************************/
void copy_in(COPY_TABLE *tp)
{
   unsigned short i;
   for (i = 0; i < tp->num_recs; i++)
   {
      COPY_RECORD   *crp       = &tp->recs[i];
      int            load_pgid = (int)(crp->load_loc >> 24);
      unsigned char *load_addr = (unsigned char *)(crp->load_loc & 0x7fffff);
      int            run_pgid  = (int)(crp->run_loc >> 24);
      unsigned char *run_addr  = (unsigned char *)(crp->run_loc & 0x7fffff);
      unsigned int   cpy_type  = 0;

      /*************************************************************************/
      /* If page ID != 0, location is assumed to be in I/O memory.             */
      /*************************************************************************/
      if (load_pgid) cpy_type += 2;
      if (run_pgid)  cpy_type += 1;
```

例 8–19. ランタイムサポート cpy_tbl.c ファイル（続き）

```
      /*************************************************************************/
      /* Dispatch to appropriate copy routine based on whether or not load   */
      /* and/or run location is in I/O memory.                                 */
      /*************************************************************************/
      switch (cpy_type)
      {
         case 3:cpy_io_to_io(load_addr, run_addr, crp->size);  break;
         case 2:cpy_io_to_mem(load_addr, run_addr, crp->size); break;
         case 1:cpy_mem_to_io(load_addr, run_addr, crp->size); break;
         case 0:memcpy(run_addr, load_addr, crp->size); break;
      }
   }
}

/*************************************************************************/
/* CPY_IO_TO_IO() - Move code/data from one location in I/O to another.      */
/*************************************************************************/
static void cpy_io_to_io(void *from, void *to, size_t n)
{
   ioport unsigned char *src = (unsigned char *)from;
   ioport unsigned char *dst = (unsigned char *)to;
   register size_t rn;

   for (rn = 0; rn < n; rn++) *dst++ = *src++;
}

/*************************************************************************/
/* CPY_IO_TO_MEM() - Move code/data from I/O to normal system memory.        */
/*************************************************************************/
static void cpy_io_to_mem(void *from, void *to, size_t n)
{
   ioport unsigned char *src = (unsigned char *)from;
   unsigned char        *dst = (unsigned char *)to;
   register size_t rn;

   for (rn = 0; rn < n; rn++) *dst++ = *src++;
}

/*************************************************************************/
/* CPY_MEM_TO_IO() - Move code/data from normal memory to I/O.               */
/*************************************************************************/
static void cpy_mem_to_io(void *from, void *to, size_t n)
{
   unsigned char        *src = (unsigned char *)from;
   ioport unsigned char *dst = (unsigned char *)to;
   register size_t rn;

   for (rn = 0; rn < n; rn++) *dst++ = *src++;
}
```

ロード・ページ ID と実行ページ ID は load_loc と run_loc フィールドからアンパックし、転送元メモリ・タイプから転送先メモリ・タイプにコピーする適切なサブルーチンを選択するために使用します。ページ ID が 0 の場合は、指定したアドレスが標準の C55x メモリにあることを示し、ページ ID が 0 以外の場合は、アドレスが I/O メモリにあることを示します。コード / データを I/O メモリに移動し、およびまたは I/O メモリから移動する必要がある場合、汎用コピー・ルーチンは特殊なコピー・ルーチンを使用します。

ポインタは、*ioport* キーワードを指定することができる。その場合、そのアドレスからのメモリの読み取りが readport() 命令修飾子または port() オペランド修飾子によって指定するということを意味します。同様に、このようなポインタへのメモリの書き込みは、writeport() 命令修飾子または port() オペランド修飾子によって修飾する必要があります。ioport キーワードを使用してポインタを修飾することにより、コンパイラはこれらの I/O 修飾子を自動的に生成します。

8.17.10リンカが生成するコピー・テーブルのセクションとシンボル

リンカは、生成する各コピー・テーブルの独立した入力セクションを作成し、割り当てます。各コピー・テーブルのシンボルは、対応するコピー・テーブルを含む入力セクションのアドレス値によって定義されます。

リンカは、各オーバーレイ・コピー・テーブルの入力セクション用に固有の名前を生成します。たとえば、table(_first_ctbl) は、.first セクションのコピー・テーブルを .ovly:_first_ctbl と呼ばれる入力セクションに配置します。リンカは、ブート時コピー・テーブル全体を含む単一の入力セクション .binit を作成します。

例 8–20 は、リンカ・コマンド・ファイルの入力セクション名を使用して、リンカが生成するコピー・テーブル・セクションの配置を制御する方法を示しています。

例 8–20. リンカが生成するコピー・テーブル・セクションの配置の制御方法

```
SECTIONS
{
   UNION
   {
      .first:{ a1.obj(.text), b1.obj(.text), c1.obj(.text) }
             load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)

      .second:{ a2.obj(.text), b2.obj(.text) }
             load = EMEM, run = PMEM, table(_second_ctbl)
   }

   .extra:load = EMEM, run = PMEM, table(BINIT)

   ...

   .ovly:{ } > BMEM
   .binit:{ } > BMEM
}
```

例 8–20 のリンカ・コマンド・ファイルでは、ブート時コピー・テーブルは .binit 入力セクションに生成され、.binit 出力セクションに収集され、BMEM メモリ領域のアドレスにマップされます。_first_ctbl は .ovly:_first_ctbl 入力セクションに生成され、_second_ctbl は .ovly:_second_ctbl 入力セクションに生成されます。これらの入力セクションの基本名が .ovly 出力セクションと一致しているので、入力セクションは .ovly 出力セクションに収集され、BMEM メモリ領域のアドレスにマップされます。

リンカが生成するコピー・テーブル・セクションに明示的な配置命令を指定しない場合は、リンカのデフォルトの配置アルゴリズムに基づいて配置されます。

リンカは、同じ出力セクション内のコピー・テーブルの入力セクションに、他の入力セクションの型を結合することはできません。リンカは、部分リンク・セッションで作成されたコピー・テーブル・セクションを後続のリンク・セッションの入力として使用することはできません。

8.17.11オブジェクト・コンポーネントの分割とオーバーレイ管理

リンカの従来のバージョンでは、別個のロードおよび実行配置命令をもつセクションの分割は使用が禁止されていました。この制限は、開発者にとって、分割されたオブジェクト・コンポーネントの各断片のロード・アドレスまたは実行アドレスにアクセスする効果的なメカニズムがなかったためです。このため、分割セクションをロードする位置から実行する位置に移動可能にするコピー・ルーチンを記述する効果的な方法がありませんでした。

しかし、リンカは、分割されたオブジェクト・コンポーネントのすべての断片のロードする位置と実行する位置の両方にアクセスすることができます。table() 演算子を使用して、この情報をコピー・テーブルに生成するようにリンカに指示することができます。リンカは、分割されたオブジェクト・コンポーネントの個々の断片に、コピー・テーブル内の COPY_RECORD エントリを指定します。

たとえば、7 つのタスクをもつアプリケーションについて考えてみましょう。タスク 1 ～ 3 は、タスク 4 ～ 7 によってオーバーレイされています（UNION 疑似命令を使用）。すべてのタスクのロードされる位置は 4 つの異なるメモリ領域（LMEM1、LMEM2、LMEM3、LMEM4）に分割されています。オーバーレイは、メモリ領域 PMEM の一部として定義されています。このセットのサービスを使用する前に、実行時にタスクの各セットをオーバーレイに移動する必要があります。

table() 演算子を分割演算子 >> と組み合わせて使用して、例 8–21 に示すように、いずれのタスクのグループでもメモリ・オーバーレイに移動するために必要なすべての情報をもつコピー・テーブルを作成します。例 8–22 には、このようなアプリケーションで使用可能なドライバを示します。

例 8–21. 分割されたオブジェクト・コンポーネントにアクセスするためのコピー・テーブルの作成方法

```
SECTIONS
{
   UNION
   {
      .task1to3:{ *(.task1), *(.task2), *(.task3) }
              load >> LMEM1 | LMEM2 | LMEM4, table(_task13_ctbl)

      GROUP
      {
         .task4:{ *(.task4) }
         .task5:{ *(.task5) }
         .task6:{ *(.task6) }
         .task7:{ *(.task7) }

      } load >> LMEM1 | LMEM3 | LMEM4, table(_task47_ctbl)

   } run = PMEM

   ...

   .ovly:                          > LMEM4
}
```

例 8–22. 分割されたオブジェクト・コンポーネントのドライバ

```
#include <cpy_tbl.h>

extern COPY_TABLE task13_ctbl;
extern COPY_TABLE task47_ctbl;

extern void task1(void);
...
extern void task7(void);

main()
{
   ...
   copy_in(&task13_ctbl);
   task1();
   task2();
   task3();
   ...

   copy_in(&task47_ctbl);
   task4();
   task5();
   task6();
   task7();
   ...
}
```

.task1to3 セクションの内容は、セクションのロード空間に分割され、実行空間で連続しています。リンカが生成したコピー・テーブル _task13_ctbl には、分割されたセクション .task1to3 の各断片の独立した COPY_RECORD が含まれています。_task13_ctbl のアドレスが copy_in() に渡されると、.task1to3 の各断片がロードする位置から実行する位置にコピーされます。

タスク 4 ～ 7 に含まれる GROUP の内容もロード空間に分割されます。リンカは、GROUP の各メンバに順番に分割演算子を適用して、GROUP の分割を行います。この結果、GROUP のコピー・テーブルに、GROUP のすべてのメンバのすべての断片の COPY_RECORD エントリが含まれます。_task47_ctbl が copy_in() により処理されると、これらの断片はメモリ・オーバーレイにコピーされます。

分割演算子は、出力セクション GROUP あるいは UNION または UNION メンバのロード配置に適用できます。リンカは、分割演算子の UNION または UNION メンバのいずれかの実行配置への適用を許可しておりません。リンカは、このような違反を検出し、警告を出力し、違反している分割演算子の使用を無視します。

8.18 部分（インクリメンタル）リンク

すでにリンクが済んだ出力ファイルは、追加モジュールとともに再度リンクできます。これは、部分リンクまたはインクリメンタル・リンクと呼ばれます。部分リンクを使用すると、大きなアプリケーションを分割し、個々の部分を別々にリンクしてからすべての部分をまとめてリンクし、最終的な実行可能プログラムを作成できます。。

再リンクするファイルを作成するためには、次のガイドラインに従ってください。

- ❑ 中間ファイルは、シンボル情報が必要です。デフォルトでは、リンカの出力にはシンボル情報が保持されます。ファイルを再リンクする予定がある場合は、-s オプションを使用してはなりません。-s は出力モジュールからシンボル情報を除去するからです。

- ❑ 中間のリンク段階は、出力セクションの編成だけに関係します。割り当てには関係しません。すべての割り当て、バインディング、および MEMORY 疑似命令は最終リンクの段階で実行してください。

- ❑ 中間ファイルに、別のファイル内のグローバル・シンボルと同じ名前のグローバル・シンボルが存在し、それらのシンボルを静的シンボルとして（中間ファイルの中にだけ見えるように）扱う場合は、-h オプションを使用してファイルをリンクする必要があります（8.4.9 項「すべてのグローバル・シンボルの静的化（-h オプション）」（8-11 ページ）を参照してください）。

- ❑ C コードをリンクする場合は、最終リンクの段階まで -c または -cr を使用しないでください。-c または -cr オプションを指定してリンカを起動するたびに、リンカはエントリ・ポイントを作成しようとします。

次の例は、部分リンクの使用方法を示しています。

ステップ 1: file1.com ファイルをリンクします。-r オプションを使用して、出力ファイル tempout1.out 内に再配置情報を保持します。

```
cl55 -z -r -o tempout1 file1.com
```

file1.com には、次の構文が含まれています。

```
SECTIONS
{
   ss1:   {
          f1.obj
          f2.obj
           .
           .
           .
          fn.obj
          }
}
```

ステップ 2: file2.com ファイルをリンクします。-r オプションを使用して、出力ファイル tempout2.out 内に再配置情報を保持します。

```
cl55 -z -r -o tempout2 file2.com
```

file2.com には、次の構文が含まれています。

```
SECTIONS
{
      ss2:   {
             g1.obj
             g2.obj
               .
               .
               .
             gn.obj
             }
}
```

ステップ 3: tempout1.out と tempout2.out をリンクします。

```
cl55 -z -m final.map -o final.out tempout1.out tempout2.out
```

8.19 C/C++ コードのリンク

TMS320C55x C/C++ コンパイラは、アセンブルとリンクができるアセンブリ言語ソース・コードを生成します。たとえば、次のように prog1、prog2 などのモジュールで構成される C/C++ プログラムをアセンブルしてからリンクし、prog.out という名前の実行可能ファイルを作成できます。

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

大規模のメモリ・モデルを使用するには、rts55x.lib ランタイム・ライブラリを指定する必要があります。

-c オプションは、C/C++ 環境によって定義された特別な規則を使用するようにリンカに指示します。ランタイム・ライブラリには、C/C++ ランタイムサポート関数が含まれています。

ランタイム環境およびランタイムサポート関数を含む C/C++ に関する詳細は、TMS320C55x オプティマイジング C/C++ コンパイラ ユーザーズ・ガイドを参照してください。

8.19.1 ランタイム初期化

すべての C/C++ プログラムは、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。プログラムは、実行を開始すると、最初に boot.obj を実行します。boot.obj には、ランタイム環境を初期化するためのコードとデータが入っています。このモジュールは、次の作業を実行します。

- ❏ システム・スタックをセットアップする。
- ❏ プライマリおよびセカンダリ・システム・スタックをセットアップする。
- ❏ ランタイム初期化テーブルを処理し、グローバル変数を自動初期化する（ROM モデルの場合）。
- ❏ 割り込みを禁止し、_main を呼び出す。

ランタイムサポート・オブジェクト・ライブラリには、boot.obj が含まれています。次のことができます。

- ❏ アーカイバを使用してライブラリから boot.obj を抽出し、このモジュールを直接リンクする。
- ❏ rts55.lib を入力ファイルとして組み込む（-c または -cr オプションを使用する場合、リンカは boot.obj を自動的に抽出します）。
- ❏ 適切なランタイム・ライブラリを入力ファイルとして組み込む（-c または -cr オプションを使用する場合、リンカは boot.obj を自動的に抽出します）。

8.19.2 オブジェクト・ライブラリとランタイム・サポート

TMS320C55x オプティマイジング C/C++ コンパイラ ユーザーズ・ガイドは、rts55.lib と rts55x.lib に組み込まれた追加ランタイムサポート関数について説明しています。プログラムがこれらの関数を使用している場合、適切なランタイム・ライブラリをオブジェクト・ファイルにリンクさせる必要があります。

ユーザー独自のオブジェクト・ライブラリを作成し、それをリンクすることもできます。リンカは、未定義の参照を解決するライブラリ・メンバだけを組み込んでリンクします。

8.19.3 スタック・セクションとヒープ・セクションのサイズ設定

C は、それぞれ malloc() 関数とランタイム・スタックが使用するメモリ・プールとして、.sysmem、.stack、および .sysstack という初期化されないセクションを使用します。-heap オプション、-stack オプション、または -sysstack オプションを使用し、オプションのすぐ後にセクションのサイズを定数として指定して、これらのサイズを設定することができます。.sysmem のデフォルト・サイズは 2000 バイトです。.stack と .sysstack のデフォルト・サイズは 1000 バイトです。

注 : .stack および .sysstack セクションの割り当て

.stack と .sysstack セクションは、同じ 64K ワードのデータ・ページに割り当てる必要があります。

詳細は、8.4.10 項「ヒープ・サイズの定義（-heap constant オプション）」（8-12 ページ）、8.4.16 項「スタック・サイズの定義（-stack constant オプション）」（8-16 ページ）、または 8.4.17 項「セカンダリ・スタック・サイズの定義（-sysstack constant オプション）」（8-17 ページ）を参照してください。

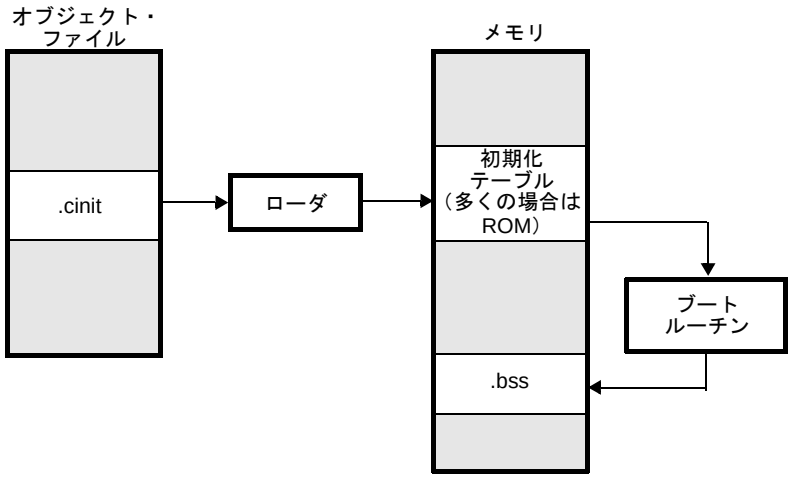
8.19.4 実行時の変数の自動初期化

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

この方式を使用すると、.cinit セクションが他のすべての初期化セクションとともにメモリにロードされます。リンカは、メモリ内の初期化テーブルの始めを指す cinit という特殊シンボルを定義します。プログラムが実行を開始すると、C/C++ ブート・ルーチンは、テーブル（.cinit で指定された）に入っているデータを .bss セクション内の指定された変数の中にコピーします。これにより、初期化データを低速外部メモリに格納し、プログラムを起動するたびに高速外部メモリにコピーできるようになります。

図 8-8 は、ROM 自動初期化モデルを示しています。

図 8-8. 実行時の自動初期化



8.19.5 ロード時の変数の初期化

ロード時の変数の初期化により、ブート時間が短縮され、初期化テーブルに使用するメモリも節約できるので、パフォーマンスが向上します。この方式を使用するには、-cr オプションを使用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場合、リンカは .cinit セクションのヘッダに STYP_COPY ビットを設定します。これは、ローダに .cinit セクションをメモリにロードしないように指示します。(.cinit セクションはメモリ・マップの空間を一切占有しません。) また、リンカは、cinit シンボルに -1 を設定します (通常、cinit は初期化テーブルの始まりを指します)。これは、初期化テーブルがメモリに存在しないことをブート・ルーチンに通知します。したがって、ブート時にランタイム初期化は行われません。

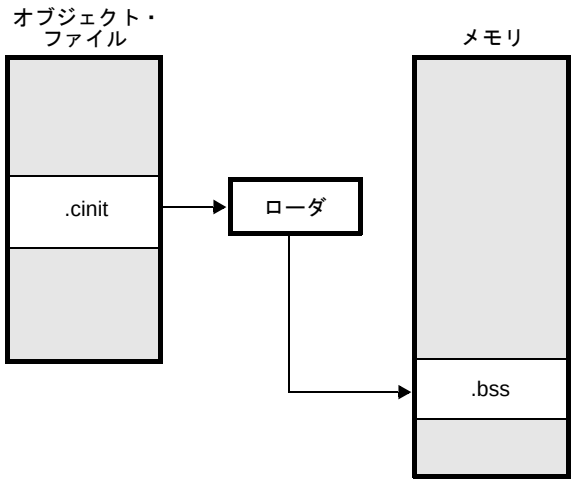
ロード時に初期化を使用するために、ローダは、次のタスクを実行できなければなりません。

- ❏ 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 セクションがメモリにコピーされないことを確認できるように、STYP_COPY が .cinit セクション・ヘッダに設定されているかどうかを判定する。
- ❏ 初期化テーブルのフォーマットを理解する (このフォーマットについては、TMS320C55x オプティマイジング C/C++ コンパイラ ユーザーズ・ガイドを参照してください)。

この後、ローダは初期化テーブルをオブジェクト・ファイルから直接使用して、.bss 内の変数を初期化します。

図 8-9 は、ロード時の変数の初期化を示しています。

図 8-9. ロード時の初期化



8.19.6 -c リンカ・オプションと -cr リンカ・オプション

次のリストに、-c または -cr のオプションを指定してリンカを起動したときに何が起きるかをまとめます。

- ❑ シンボル _c_int00 はプログラムのエントリ・ポイントとして定義されます。_c_int00 は boot.obj 内の C/C++ ブート・ルーチンの開始点です。_c_int00 を参照することにより、boot.obj はランタイムサポート・ライブラリ rts55.lib から自動的にリンクされます。

- ❑ .cinit 出力セクションに終端レコードが埋め込まれ、ブート・ルーチン（ROM モデルの場合）またはローダ（RAM モデルの場合）に初期化テーブルの読み取りを停止する時期を知らせます。

- ❑ 実行時に自動初期化を行う場合（-c オプション）、リンカは cinit を .cinit セクションの開始アドレスとして定義します。C/C++ ブート・ルーチンは、このシンボルを自動初期化の開始点として使用します。

- ❑ ロード時に初期化を行う場合（-cr オプション）は、次のように実行されます。
  - ■ リンカはシンボル cinit 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、したがって実行時に初期化は行われません。
  - ■ STYP_COPY フラグ（0010h）は .cinit セクションのヘッダ内に設定されます。STYP_COPY は特別な属性で、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内にロードしないように指示します。リンカは、メモリ内に .cinit セクション用の空間を割り当てません。

8.20 リンカの例

この例では、demo.obj、fft.obj、および tables.obj という 3 つのオブジェクト・ファイルをリンクし、demo.out というプログラムを作成します。シンボル SETUP は、プログラムのエントリ・ポイントです。

ターゲット・メモリは次のような構成であるとします。

プログラム・メモリ

アドレス範囲内容
0080 ～ 7000 オンチップ RAM_PG
C000 ～ FF80オンチップ ROM

データ・メモリ

アドレス範囲内容
0080 ～ 0FFF RAM ブロック ONCHIP
0060 ～ FFFF マップされた外部アドレス EXT

バイト・アドレス範囲内容
000100 ～ 007080 オンチップ RAM_PG
007081 ～ 008000 RAM ブロック ONCHIP
008001 ～ 00A000 マップされた外部アドレス EXT
00C000 ～ 00FF80オンチップ ROM

出力セクションは、次の入力セクションから構築されます。

- ❏ demo.obj、fft.obj、および tables.obj の .text セクションに入っている実行可能コードは、プログラム ROM にリンクされなければなりません。
- ❏ demo.obj の var_defs セクションに入っている変数は、ONCHIP ブロック内のデータ・メモリの中へリンクされなければなりません。
- ❏ demo.obj、tables.obj、および fft.obj の .data セクションに入っている係数テーブルは、データ・メモリ内の RAM ブロック ONCHIP の中へリンクされなければなりません。長さが 100 バイトで、埋め込み値が 07A1Ch のホールが作成されます。ONCHIP ブロックの残りの部分は、07A1Ch の値に初期化されなければなりません。
- ❏ demo.obj、tables.obj、および fft.obj の変数が入った .bss セクションは、プログラム RAM の RAM_PG ブロックの中へリンクされなければなりません。この RAM の未使用部分は、0FFFFh に初期化されなければなりません。
- ❏ demo.obj のバッファと変数が入った xy セクションは、明示的にリンクされなかったので、デフォルトとしてデータ RAM の ONCHIP ブロックの中へリンクされます。

例 8–23 は、この例のリンカ・コマンド・ファイルを示しています。例 8–24 は、マップ・ファイルを示しています。

例 8–23. リンカ・コマンド・ファイル demo.cmd

```
/************************************************************/
/***                  Specify Linker Options              ***/
/************************************************************/
-e coeff                      /* Define the program entry point */
-o demo.out                   /* Name the output file           */
-m demo.map                   /* Create an output map           */

/************************************************************/
/***                   Specify the Input Files            ***/
/************************************************************/

demo.obj
fft.obj
tables.obj

/************************************************************/
/***             Specify the Memory Configurations        ***/
/************************************************************/

MEMORY
{
   RAM_PG:origin=00100h    length=06F80h
   ONCHIP:origin=007081h   length=0F7Fh
     EXT:origin=08001h    length=01FFFh
     ROM:origin=0C000h    length=03F80h
}

/************************************************************/
/***               Specify the Output Sections            ***/
/************************************************************/

SECTIONS
{
        .text:load = ROM           /*  link .text into ROM */

        var_defs:load = ONCHIP     /*     defs in RAM     */

        .data:fill = 07A1Ch, load=ONCHIP
         {
                tables.obj(.data) /* .data input */
                fft.obj(.data)    /* .data input */
                . = 100h;     /* create hole, fill with 07A1Ch */
         }                    /* and link with ONCHIP          */

        .bss:load=RAM_PG,fill=0FFFFh
                              /* Remaining .bss; fill and link */
}

/************************************************************/
/***                   End of Command File                ***/
/************************************************************/
```

次のコマンドを使用してリンカを起動します。

```
cl55 -z demo.cmd
```

これにより、例 8–24 に示したマップ・ファイルと、TMS320C55x 上で実行できる demo.out という出力ファイルが作成されます。

例 8–24. 出力マップ・ファイル demo.map

```
OUTPUT FILE NAME:<demo.out>
ENTRY POINT SYMBOL: 0

MEMORY CONFIGURATION
  name    org(bytes) len(bytes) used(bytes) attributes  fill
 --------  --------  ---------  ---------  ----------  --------
 RAM_PG    00000100  000006f80   00000064    RWIX
 ONCHIP    00007081  000000f7f   00000104    RWIX
   EXT     00008000  000001fff   00000000    RWIX
   ROM     0000c000  000003f80   0000001f    RWIX

SECTION ALLOCATION MAP
 output                                                  attributes/
section  page  org(bytes) org(words) len(bytes) len(words)  input sections
-------- ---- ---------- ---------- ---------- ---------- ----------------
.text    0     0000c000              0000001f
               0000c000              0000000a              tables.obj(.text)
               0000c00a              00000008              fft.obj (.text)
               0000c012              0000000c              demo.obj (.text)
               0000c01e              00000001              --HOLE-- [fill = 2020]
var_defs  0               00003841              00000002
                          00003841              00000002    fft.obj (var_defs)
.data    0                00003843              00000080
                          00003843              00000001    tables.obj (.data)
                          00003844              00000004    fft.obj (.data)
                          00003848              0000007b    --HOLE-- [fill = 7a1c]
                          000038c3              00000000    demo.obj (.data)
.bss     0                00000080              00000002
                          00000080              00000002    demo.obj(.bss)[fill=ffff]
                          00000082              00000000    fft.obj (.bss)
                          00000082              00000000    tables.obj (.bss)
xy       0                00000082              00000030    UNINITIALIZED
                          00000082              00000030    demo.obj (xy)

GLOBAL SYMBOLS:
Sorted alphabetically by name               Sorted by symbol address
abs. value/                                 abs. value/
byte addr  word addr  name                  byte addr  word addr  name
---------  ---------  -----                 --------   --------  ----
          00000080   .bss                              00000080  .bss
          00003843   .data                             00000082   end
0000c000             .text                             00003843  .data
0000c016              ARRAY                            00003843   TEMP
          00003843    TEMP                             000038c3   edata
0000c012              _x42                  0000c012              _x42
          000038c3    edata                 0000c000              .text
          00000082    end                   0000c016 .            ARRAY
0000c01f              etext                 0000c01f              etext
```

第 9 章

アーカイバの説明

TMS320C55xx アーカイバを使用すると、いくつかのファイルを集めて 1 つのアーカイブ・ファイルにできます。たとえば、いくつかのマクロを集めて 1 つのマクロ・ライブラリにできます。アセンブラはこのライブラリを検索し、ソース・ファイルによりマクロとして呼び出されたメンバを使用します。アーカイバを使用して、いくつかのオブジェクト・ファイルを集めて 1 つのオブジェクト・ライブラリにすることもできます。リンカは、リンク中に、外部参照を解決するメンバをライブラリから取り込みます。

項目 **ページ**

9.1 アーカイバの概要

TMS320C55x アーカイバを使用すると、いくつかのファイルを結合して アーカイブまたはライブラリと呼ばれる 1 つのファイルを作成できます。アーカイブ内の個々のファイルはメンバと呼ばれます。アーカイブを作成すると、アーカイバを使用してメンバの追加、削除、または抽出を行うことができます。

ライブラリは、どのような種類のファイルからも作成できます。アセンブラとリンカは、どちらもアーカイブ・ライブラリを入力として受け入れます。アセンブラは個々のソース・ファイルが入ったライブラリを使用でき、リンカは個々のオブジェクト・ファイルが入ったライブラリを使用できます。

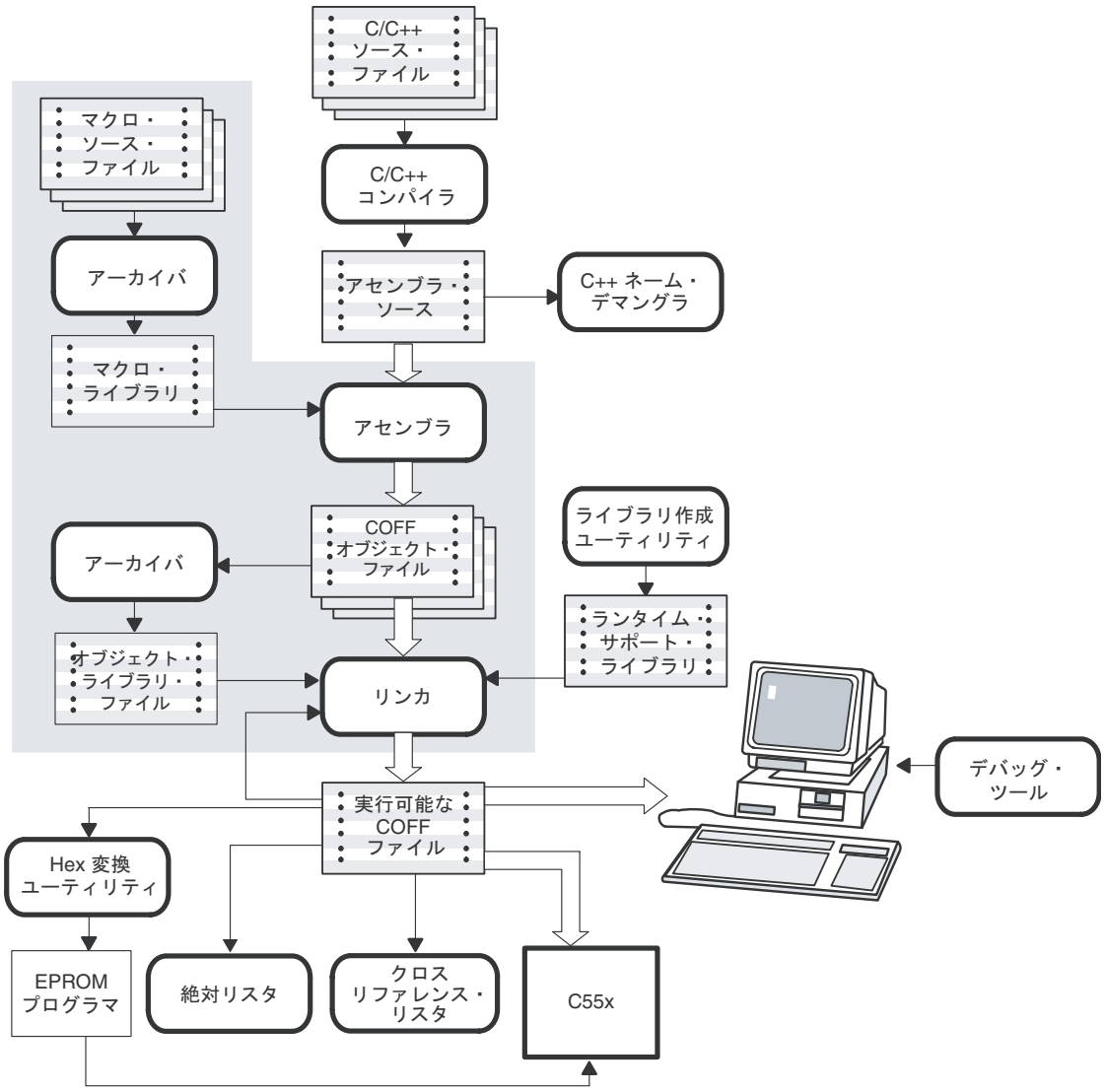
アーカイバの最も便利な使用方法の 1 つは、オブジェクト・モジュールのライブラリを作成することです。たとえば、いくつかの算術ルーチンを書いてアセンブルし、さらに、アーカイバを使用してオブジェクト・ファイルを集めて 1 つの論理グループを作成できます。次に、オブジェクト・ライブラリをリンカ入力として指定します。リンカは、ライブラリ全体を検索し、外部参照を解決するメンバがあれば取り込みます。

アーカイバを使用してマクロ・ライブラリを作成することもできます。それぞれがマクロを 1 つ含むソース・ファイルをいくつか作成し、アーカイバを使用してこれらのマクロを結合して 1 つの機能グループを作ることができます。.mlib アセンブラ疑似命令を使用してマクロ・ライブラリの名前を指定できます。アセンブリ処理時に、アセンブラは指定されたライブラリからユーザーに呼び出されたマクロを検索します。マクロおよびマクロ・ライブラリの詳細は、第 5 章「マクロ言語」を参照してください。

9.2 アーカイバ開発のフロー

図 9-1 は、アセンブリ言語開発プロセスにおけるアーカイバの役割を示したものです。アセンブラとリンカは、どちらもライブラリを入力として受け入れます。

図 9-1. アーカイバ開発のフロー



9.3 アーカイバの起動方法

アーカイバを起動するには、次のように入力します。

ar55 [-]*command*[*option*] *libname* [*filename*$_1$ ... *filename*$_n$]

ar55 アーカイバを起動するコマンドです。

command アーカイバに対して、ライブラリ・メンバの操作方法を指示します。コマンドの前には、オプションでハイフンを付けることができます。アーカイバを起動するときには、以下のコマンドのいずれかを必ず指定してください。ただし、1 回の起動で指定できるのは 1 つのコマンドだけです。有効なアーカイバ・コマンドは次のとおりです。

a 指定されたファイルをライブラリに追加します。このコマンドは、追加したファイルと同じ名前をもつメンバがすでにあっても、そのメンバを置換しません。単に新しいメンバをアーカイブの最後に追加するだけです。

d 指定されたメンバをライブラリから削除します。

r ライブラリ内の指定されたメンバを置換します。ファイル名を指定しない場合、アーカイバは、ライブラリ・メンバを現行のディレクトリ内にある同じ名前のファイルに置換します。ライブラリ内に指定されたファイルが存在しない場合、アーカイバはそのメンバを置換せずに追加します。

t ライブラリの内容の一覧を出力します。ファイル名を指定する場合、指定されたファイルのみのリストを出力します。ファイル名を指定しない場合、アーカイバは指定されたすべてのライブラリ・メンバのリストを作成します。

x 指定されたファイルを抽出します。メンバ名を指定しない場合、アーカイバはライブラリのすべてのメンバを抽出します。アーカイバは、メンバを 1 つ抽出すると単にそのメンバを現行のディレクトリにコピーするだけで、そのメンバをライブラリから除去することはありません。

option

アーカイバに対して動作方法を指示します。次のオプションは希望する数だけ指定できます。

-q

（静的な実行）見出しとステータス・メッセージを出力しません。

-s

ライブラリに定義されているグローバル・シンボルのリストを出力します（このオプションは、コマンド -a、-r、-d についてのみ有効です）。

-v

（verbose: 補足）古いライブラリとその構成メンバから新しいライブラリを作成するときに、ファイルごとにその説明を表示します。

libname

アーカイブ・ライブラリの名前を指定します。libname の拡張子を指定しない場合、アーカイバはデフォルトの拡張子 .lib を付けます。

filename

ライブラリに関連付けられる個々のメンバ・ファイルの名前を指定します。拡張子がある場合は拡張子付きの完全なファイル名を指定する必要があります。
1 つのライブラリに同じ名前の複数のメンバを組み込むことは可能です（望ましくはありませんが）。メンバを削除、置換、または抽出しようとし、かつライブラリ内に指定した名前のメンバが複数存在する場合、アーカイバはその名前をもった最初のメンバを削除、置換、または抽出します。

9.4 アーカイバの例

アーカイバの使用例を次に示します。

- sine.obj、cos.obj、および flt.obj のファイルを格納する function.lib という名前のライブラリを作成する場合は、次のように入力します。

```
ar55 -a function sine.obj cos.obj flt.obj
TMS320C55x Archiver              Version x.xx
Copyright (c) 2001    Texas Instruments Incorporated
    ==>    new archive 'function.lib'
    ==>    building archive 'function.lib'
```

- -t オプションを指定すると、function.lib の内容一覧を出力できます。

```
ar55 -t function
TMS320C55x Archiver              Version x.xx
Copyright (c) 2001    Texas Instruments Incorporated
      FILE NAME      SIZE  DATE
-----------------    ----- ----------------------
      sine.obj       248   Mon Nov 19 01:25:44 2001
      cos.obj        248   Mon Nov 19 01:25:44 2001
      flt.obj        248   Mon Nov 19 01:25:44 2001
```

- ライブラリに新しいメンバを追加する場合は、次のように入力します。

```
ar55  -as function atan.obj
TMS320C55x Archiver              Version x.xx
Copyright (c) 2001    Texas Instruments Incorporated
    ==>    symbol defined: 'symbol_name'
    ==>    symbol defined: 'symbol_name'
    ==>    building archive 'function.lib'
```

上記の例では libname の拡張子が指定されていないため、アーカイバは function.lib という名前のライブラリにファイルを追加します。function.lib が存在しない場合、アーカイバは function.lib を作成します（-s オプションはアーカイバに対して、ライブラリで定義されているグローバル・シンボルのリストを出力するように指示します）。

- ライブラリ・メンバを変更する場合は、変更するメンバを抽出し、編集して、置換します。次の例では、macros.lib という名前のライブラリに push.asm、pop.asm、および swap.asm というメンバが含まれていると仮定します。

```
ar55 -x macros push.asm
```

アーカイバは push.asm のコピーを作成し、そのコピーを現行のディレクトリに入れます。ただし、ライブラリから push.asm が除去されることはありません。これで抽出したファイルを編集できます。ライブラリの push.asm コピーを編集した push.asm コピーに置換するには、次のように入力します。

```
ar55 -r macros push.asm
```

第 10 章

絶対リスタの説明

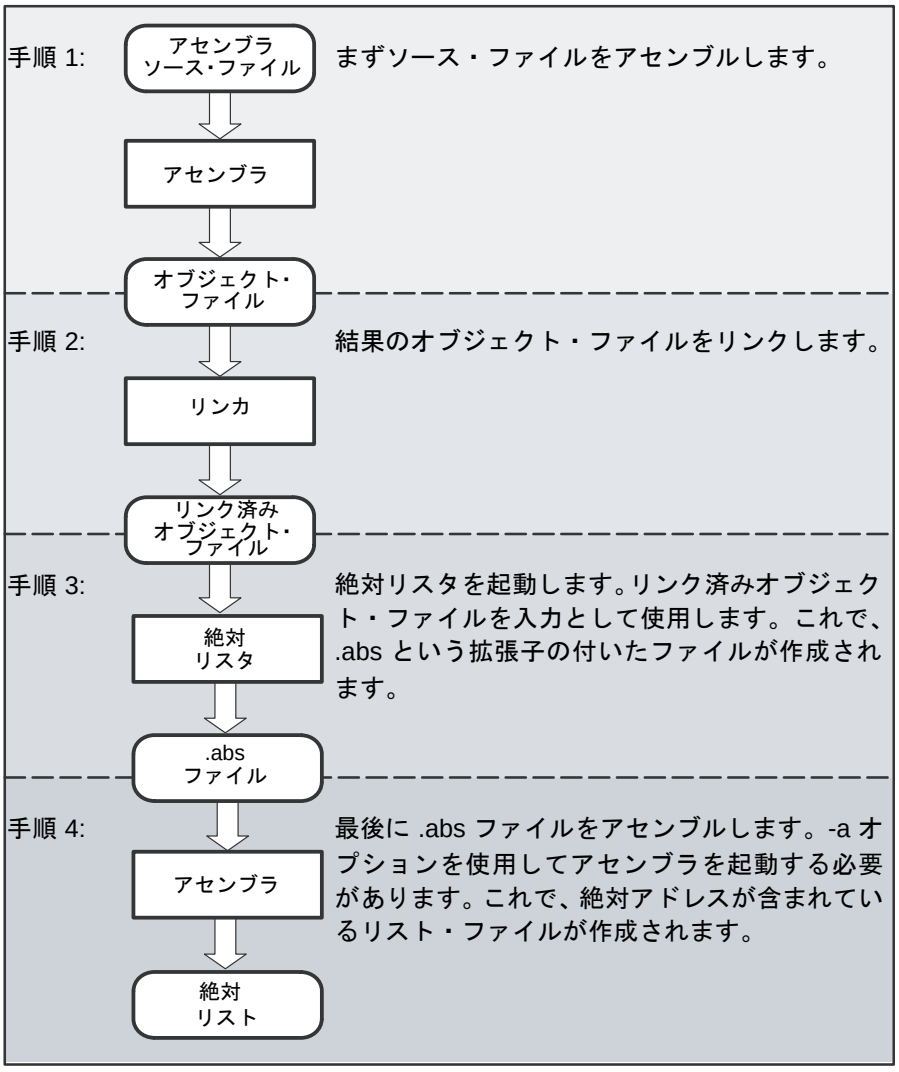
絶対リスタは、リンク済みオブジェクト・ファイルを入力として受け入れ、.abs ファイルを出力として作成するデバッグ・ツールです。.abs ファイルをアセンブルすることにより、オブジェクト・コードの絶対アドレスを示すリスト出力を作成することができます。これを手動で行うと、多数の操作を要する繁雑なプロセスが必要になります。しかし、絶対リスタ・ユーティリティを使用すると、このような操作は自動的に行われます。

項目 **ページ**

10.1 絶対リストの作成方法

図 10-1 は、絶対リストの作成に必要な手順を示しています。

図 10-1. 絶対リスタ開発のフロー



10.2 絶対リスタの起動方法

絶対リスタを起動するには、次のように入力します。

abs55 [*-options*] *input file*

abs55 絶対リスタを起動するコマンドです。

options 使用する絶対リスタ・オプションを識別します。オプションでは大文字と小文字は区別されません。オプションは、コマンドの後ならそのコマンド行のどこにでも指定できます。各オプションの前にはハイフン (-) を付けます。有効な絶対リスタ・オプションは次のとおりです。

-e アセンブリ・ファイル、C ソース・ファイル、および C ヘッダ・ファイルのファイル名拡張子についてのデフォルトの命名規則を変更できます。3 つのオプションを、以下に記載します。

- -ea [.]*asmext* アセンブリ・ファイル用（デフォルトは .asm）
- -ec [.]*cext* C ソース・ファイル用（デフォルトは .c）
- -eh [.]*hext* C ヘッダ・ファイル用（デフォルトは .h）

拡張子の「.」とオプションと拡張子の間の空白は、指定してもしなくてもかまいません。

-q （静的な実行）見出しとすべての進捗情報を出力しません。

input file リンク済みオブジェクト・ファイルの名前を指定します。拡張子を指定しなければ、絶対リスタは入力ファイルがデフォルトの拡張子 .out をもつものと解釈します。絶対リスタの起動時に入力ファイル名を指定しないと、絶対リストは入力を求めるプロンプトを表示します。

絶対リスタは、リンクされた各ファイルにつき 1 つの出力ファイルを作成します。出力ファイルの名前は、入力ファイルの名前に拡張子 .abs を付けたものとなります。ただし、ヘッダ・ファイルについては対応する .abs ファイルは作成されません。

絶対リストを作成するには、次のような -a アセンブラ・オプションを使用して、上記の出力ファイルをアセンブルします。

```
masm55 -a filename.abs
```

-e オプションは、コマンド行のファイル名の解釈、および出力ファイルの名前の両方に作用します。これらのオプションは必ず、コマンド行ですべてのファイル名の前に指定する必要があります。

-e オプションは、デバッグ・オプション（-g コンパイラ・オプション）を指定してコンパイルした C ファイルから作成された、リンク済みオブジェクト・ファイルに使用すると便利です。デバッグ・オプションを設定すると、結果のリンク済みオブジェクト・ファイルにはその作成に使用したソース・ファイルの名前が組み込まれます。この場合、絶対リスタは C ヘッダ・ファイルについては対応する .abs ファイルを作成しません。また、C ソース・ファイルに対応する .abs ファイルは、C ソース・ファイルそのものでなく C ソース・ファイルから作成されたアセンブリ・ファイルを使用します。

たとえば、C ソース・ファイル hello.csr をデバッグ・オプションを指定してコンパイルするとします。この場合、アセンブリ・ファイル hello.s が作成されます。hello.csr には hello.hsr も組み込まれています。作成された実行可能ファイルの名前が hello.out だとすれば、次のコマンドにより適正な .abs ファイルを作成できます。

```
abs55 -ea s -ec csr -eh hsr hello.out
```

hello.hsr（ヘッダ・ファイル）については .abs ファイルは作成されません。また、hello.abs には、C ソース・ファイル hello.csr ではなくアセンブリ・ファイル hello.s が組み込まれます。

10.3 絶対リスタの例

この例では、3 つのソース・ファイルを使用しています。module1.asm と module2.asm の両方に、ファイル globals.def が含まれています。

module1.asm

```
.bss   array,100
.bss   dflag, 2
.copy  globals.def
.text
MOV #offset,AC0
MOV dflag,AC0
```

module2.asm

```
.bss   offset, 2
.copy  globals.def
.text
MOV #offset,AC0
MOV #array,AC0
```

globals.def

```
.global dflag
.global array
.global offset
```

次の手順で、module1.asm と module2.asm の 2 つのファイルについての絶対リスト出力を作成します。

ステップ 1: まず、module1.asm と module2.asm をアセンブルします。

```
masm55 module1
masm55 module2
```

これで、module1.obj と module2.obj という名前の 2 つのオブジェクト・ファイルが作成されます。

ステップ 2: 次に、bttest.cmd という名前の次のようなリンカ・コマンド・ファイルを使用して、module1.obj と module2.obj をリンクします。

```
/************************************************/
/* File bttest.cmd -- COFF linker command file  */
/*       for linking TMS320C55x modules         */
/********************************** ************/
-o bttest.out           /* Name the output file */
-m bttest.map           /* Create an output map */

/************************************************/
/*            Specify the Input Files           */
/************************************************/
module1.obj
module2.obj
```

```
/***********************************************/
/*       Specify the Memory Configurations     */
/***********************************************/
MEMORY
{
   ROM:  origin=2000h   length=2000h
   RAM:  origin=8000h   length=8000h
}

/***********************************************/
/*         Specify the Output Sections         */
/***********************************************/
SECTIONS
{
   .data:  >RAM
   .text:  >ROM
   .bss:   >RAM
}
```

ステップ 3: リンカを起動します。

```
cl55 -z bttest.cmd
```

これで、bttest.out という名前の実行可能オブジェクト・ファイルが作成されます。この新しいファイルを、絶対リスタの入力として使用します。

ステップ 4: 絶対リスタを起動します。

```
abs55 bttest.out
```

これで、module1.abs と module2.abs という 2 つのファイルが作成されます。

module1.abs:

```
          .nolist
array     .setsym     0004000h
dflag     .setsym     0004064h
offset    .setsym     0004066h
.data     .setsym     0004000h
_ _data_  .setsym     0004000h
edata     .setsym     0004000h
_ _edata_ .setsym     0004000h
.text     .setsym     0002000h
_ _text_  .setsym     0002000h
etext     .setsym     000200fh
_ _etext_ .setsym     000200fh
.bss      .setsym     0004000h
_ _bss_   .setsym     0004000h
end       .setsym     0004068h
_ _end_   .setsym     0004068h
          .setsect    ".text",0002000h
          .setsect    ".data",0004000h
          .setsect    ".bss",0004000h
          .list
```

```
          .text
          .copy       "module1.asm"
```

module2.abs:

```
          .nolist
array     .setsym     0004000h
dflag     .setsym     0004064h
offset    .setsym     0004066h
.data     .setsym     0004000h
_ _data_  .setsym     0004000h
edata     .setsym     0004000h
_ _edata_ .setsym     0004000h
.text     .setsym     0002000h
_ _text_  .setsym     0002000h
etext     .setsym     000200fh
_ _etext_ .setsym     000200fh
.bss      .setsym     0004000h
_ _bss_   .setsym     0004000h
end       .setsym     0004068h
_ _end_   .setsym     0004068h
          .setsect    ".text",02006h
          .setsect    ".data",04000h
          .setsect    ".bss",04066h
          .list
          .text
          .copy       "module2.asm"
```

この 2 つのファイルには、ステップ 4 でアセンブラを起動したときにアセンブラが必要とする次の情報が入っています。

- .setsym 疑似命令。これは、値をグローバル・シンボルにします。どちらのファイルにも、シンボル dflag 用のグローバルな等価値が含まれています。シンボル dflag は、module1.asm と module2.asm. に組み込まれているファイル globals.def の中で定義されています。
- .setsect 疑似命令。これは、セクションの絶対アドレスを定義しています。
- .copy 疑似命令。これは、どのアセンブリ言語ソース・ファイルを組み込むかをアセンブラに指示します。

.setsym 疑似命令と .setsect 疑似命令は、通常のアセンブリで使用しても役に立ちません。この 2 つの疑似命令が役に立つのは、絶対リストを作成する場合だけです。

ステップ 5: 最後に、絶対リスタにより作成された .abs ファイルをアセンブルします（アセンブラを起動するときは -a オプションを使用する必要があることを忘れないでください）。

```
masm55  -a  module1.abs
masm55  -a  module2.abs
```

これで、module1.lst と module2.lst という 2 つのリスト・ファイルが作成されます。オブジェクト・コードは生成されません。これらのリスト・ファイルは通常のリスト・ファイルに似ていますが、表示されているアドレスは絶対アドレスです。

module1.1st（図 10-2 参照）と module2.1st（図 10-3 参照）という 2 つの絶対リスト・ファイルが作成されます。

図 10-2. module1.lst

```
TMS320C55x COFF Assembler       Version x.xx     Wed Oct 16 12:00:05 2001
 Copyright (c) 2001    Texas Instruments Incorporated

module1.abs                                                          PAGE    1

      21 002000               .text
      22                      .copy       "module1.asm"
 A     1 004000              .bss   array, 100
 A     2 004064              .bss   dflag, 2
 A     3                     .copy  globals.def
 B     1                     .global dflag
 B     2                     .global array
 B     3                     .global offset
 A     4 002000               .text

 A     5 002000 7640         MOV #offset,AC0
         002002 6608!
 A     6 002004 A000%        MOV dflag,AC0
 No Errors, No Warnings
```

図 10-3. module2.lst

```
TMS320C55x COFF Assembler       Version x.xx     Wed Oct 16 12:00:17 2001
 Copyright (c) 2001    Texas Instruments Incorporated

module2.abs                                                          PAGE    1

      21 002006              .text
      22                    .copy       "module2.asm"
 A     1 004066             .bss   offset, 2
 A     2                    .copy  globals.def
 B     1                    .global dflag
 B     2                    .global array
 B     3                    .global offset

 A     3 002006              .text
 A     4 002006 7640        MOV #offset,AC0
         002008 6680-
 A     5 00200a 7640        MOV #array,AC0
         00200c 0080!
 No Errors, No Warnings
```

第 11 章

クロスリファレンス・リスタの説明

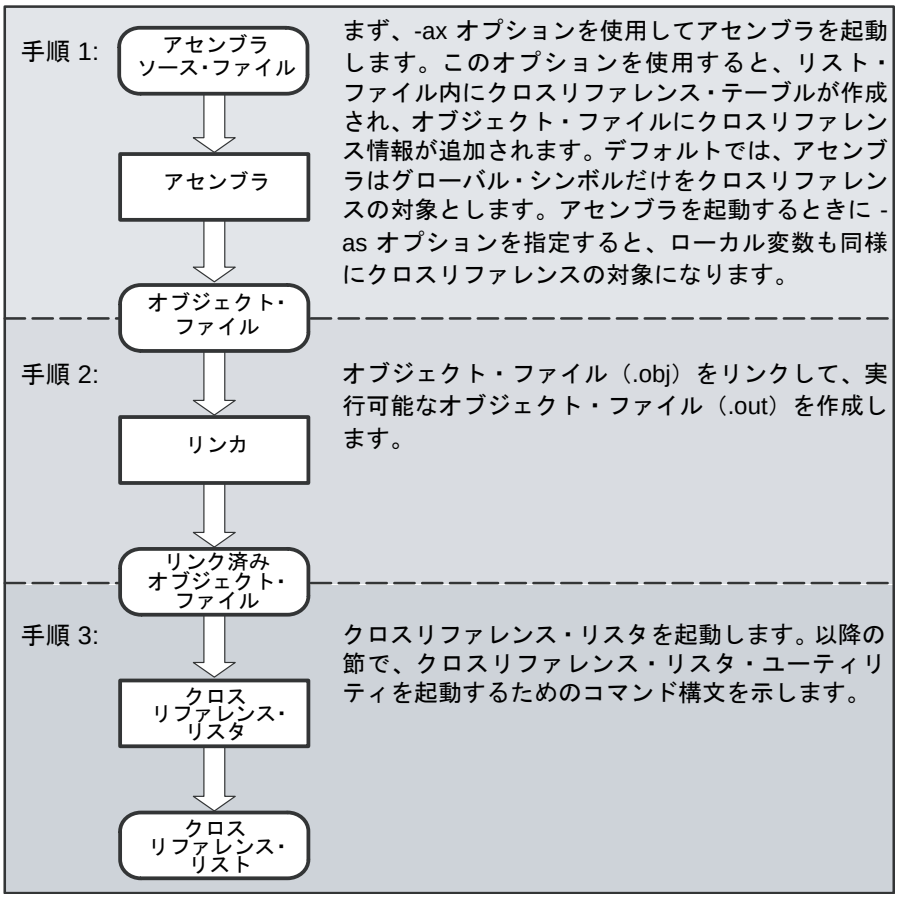
クロスリファレンス・リスタはデバッグ・ツールです。このユーティリティは、リンク済みオブジェクト・ファイルを入力として受け入れ、クロスリファレンス・リストを出力として作成します。このリストには、シンボル、シンボルの定義、リンクされたソース・ファイル内でのシンボルの参照が示されています。

項目 **ページ**

11.1　クロスリファレンス・リストの作成方法

図 11-1 は、クロスリファレンス・リスタ開発のフローを示しています。

図 11-1.　　クロス- リファレンス・リスタ開発のフロー



11.2 クロスリファレンス・リスタの起動方法

クロスリファレンス・ユーティリティを使用するには、正しいオプションを指定してファイルをアセンブルし、リンクして実行可能ファイルにする必要があります。-ax オプションを使用してアセンブリ言語ファイルをアセンブルします。このオプションを指定するとクロスリファレンス・リストが作成され、クロスリファレンス情報がオブジェクト・ファイルに追加されます。

デフォルトではアセンブラがクロスリファレンスの対象にするのはグローバル・シンボルだけですが、-as オプションを使ってアセンブラを起動した場合は、ローカル・シンボルも追加されます。実行可能ファイルを作成するには、オブジェクト・ファイルをリンクします。

クロスリファレンス・リスタを起動するには、次のように入力します。

xref55 [-*options*] [*input filename* [*output filename*]]

xref55 クロスリファレンス・ユーティリティを起動するコマンドです。

options 使用するクロスリファレンス・リスタ・オプションを識別します。オプションでは大文字と小文字は区別されません。オプションは、コマンドの後ならそのコマンド行のどこにでも指定できます。各オプションの前にはハイフン（-）を付けます。クロスリファレンス・リスタ・オプションは以下のとおりです。

- **-l** （L の小文字）出力ファイルの 1 ページ当りの行数を指定します。-l オプションのフォーマットは -l*num* で、num は 10 進定数です。たとえば -l30 を指定すると、出力ファイルの 1 ページ当りの行数は 30 行に設定されます。オプションと 10 進定数との間には、空白を入れても入れなくても構いません。デフォルトは 1 ページ当り 60 行です。
- **-q** （静的な実行）見出しとすべての進捗情報を出力しません。

input filename リンク済みオブジェクト・ファイルです。入力ファイル名を省略すると、ユーティリティはファイル名の入力を求めるプロンプトを表示します。

output filename クロスリファレンス・リスト・ファイルの名前です。出力ファイル名を省略すると、デフォルトのファイル名は入力ファイルの名前に拡張子 .xrf を付けたものになります。

11.3 クロスリファレンス・リストの例

例 11–1 は、クロスリファレンス・リストの例を示します。

例 11–1. クロスリファレンス・リストの例

```
================================================================================

Symbol: INIT

Filename          RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________          ____   ________   ________    ______  _______  _______  _______
file1.asm         EDEF   '000000    000080       3       1
file2.asm         EREF    000000    000080               2         11
================================================================================

Symbol: X

Filename          RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________          ____   ________   ________    ______  _______  _______  _______
file1.asm         EREF    000000     000001              2         5
file2.asm         EDEF    000001     000001      5       1
================================================================================

Symbol: Y

Filename          RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________          ____   ________   ________    ______  _______  _______  _______
file2.asm         EDEF    -000000     000080      7       1

================================================================================

Symbol: Z

Filename          RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________          ____   ________   ________    ______  _______  _______  _______
file2.asm         EDEF    000003     000003      9       1

================================================================================
```

次に定義した用語は、前ページのクロスリファレンス・リストに出力されている用語です。

Symbol Name リストされているシンボルの名前

Filename シンボルが使用されているファイルの名前

RTYP このファイル内でのシンボルの参照タイプ。参照タイプには次のものがあります。

- **STAT** シンボルはこのファイルで定義されていて、グローバルとして宣言されていません。
- **EDEF** シンボルはこのファイルで定義されていて、グローバルとして宣言されています。
- **EREF** シンボルはこのファイルで定義されていませんが、グローバルとして参照されます。
- **UNDF** シンボルはこのファイルで定義されておらず、グローバルとして宣言されていません。

AsmVal この 16 進数は、アセンブル時にシンボルに割り当てられた値です。値の前には、シンボルの属性を示す文字が付くこともあります。表 11–1 に、このような文字と名前を掲載しています。

LnkVal この 16 進数は、リンク後にシンボルに割り当てられた値です。

DefLn シンボルが定義されている文の番号です。

RefLn シンボルが参照されている行の番号です。行番号の後にアスタリスク（*）が付いている場合は、参照によりオブジェクトの内容が変更される可能性があります。行番号の後に文字（A、B、C など）が付いている場合は、アセンブリ・ソースの .include 疑似命令で指定されているファイルでこのシンボルが参照されています。「A」は .include 疑似命令で 1 番目に指定されているファイルに割り当てられており、「B」は 2 番目に指定されているファイルに割り当てられていて、以降も同様です。この欄が空白であれば、このシンボルが一度も使用されなかったことを示します。

表 11–1 は、クロスリファレンス・リストの例に表示されているシンボルの属性の一覧です。

表 11–1. シンボルの属性

文字	意味
'	.text セクションで定義されているシンボル
"	.data セクションで定義されているシンボル
+	.sect セクションで定義されているシンボル
-	.bss または .usect セクションで定義されているシンボル
=	.reg セクションで定義されているシンボル

第 12 章

逆アセンブラの説明

COFF 逆アセンブラは、オブジェクト・ファイルおよび実行可能ファイルを入力として受け入れ、アセンブリ・リストを出力として作成します。このリストは、アセンブリ命令、その命令コード、およびセクション・プログラム・カウンタ値を示しています。

逆アセンブリ・リストは、以下の表示に便利です。

- ❑ アセンブリ命令およびそのサイズ
- ❑ アセンブリ命令のエンコード
- ❑ リンク済み実行可能ファイルの出力

項目 **ページ**

12.1 逆アセンブラの起動方法

逆アセンブラを使用する前に、アセンブラの -s オプション（またはシェルの -as オプション）を使い、オブジェクト・ファイルの作成を検討します。このオプションを使ってファイルはアセンブルされ、ローカル・シンボルは逆アセンブリに含まれるため、より包括的なリストが作成できます。

逆アセンブラを起動するには、次のように入力します。

dis55 [*-options*] [*input filename* [*output filename*]]

dis55 逆アセンブラを起動するためのコマンドです。

input filename オブジェクト・ファイル（.obj）または実行可能なファイル（.out）です。入力ファイル名を省略すると、逆アセンブリはファイルの入力を求めるプロンプトを表示します。ファイル拡張子を指定しない場合、逆アセンブラは、filename、filename.out、filename.obj の順で検索します。

output filename 逆アセンブリ・リスト・ファイルの名前です。出力ファイル名を省略すると、標準出力にリストが送られます。

options 使用する逆アセンブラ・オプションを指定します。オプションでは大文字と小文字は区別されません。オプションは、起動の後ならそのコマンド行のどこにでも指定できます。各オプションの前にはハイフン(-)を付けます。逆アセンブラ・オプションは次のとおりです。

- **-a** 分岐先アドレスをラベルとともに表示します。
- **-b** データをバイト単位で表示します。デフォルトでは、データはワード単位で表示されます。
- **-c** リストの上部には COFF ファイルについて説明があります。この説明には、メモリ・モデル、再配置、行番号、およびローカル・シンボルの情報が含まれます。
- **-d** リスト内におけるデータ・セクションの表示を抑止します。
- **-g** （代数表記）ソース・デバッガにおけるアセンブラ・ソースのデバッグを可能にします。
- **-h** 使用可能な逆アセンブラ・オプションのリストを表示します。
- **-i** 逆アセンブラは、.data セクションを命令に逆アセンブルしようとします。

-q （静的な実行）見出しとすべての進捗情報を出力しません。

-qq 逆アセンブラの追加した見出し、すべての進捗情報、およびセクション・ヘッダの情報を抑止します。

-r 逆アセンブラに、ARMS および CPL ビットを有効化するためのコンパイラの規則を使用させます。デフォルトでは、逆アセンブラは ARMS および CPL を無効とします。C/C++ ソースから生成されたファイルを逆アセンブルする場合は必ず、-r を使います。

-s リスト内における命令コードおよびセクション・プログラム・カウンタの表示を抑止します。-qq とともにこのオプションを使う場合、逆アセンブリ・リストはオリジナルのアセンブリ・ソース・ファイルのように見えます。

-t リスト内におけるテキスト・セクションの表示を抑止します。

12.2 逆アセンブリの例

ここでは逆アセンブラの様々な機能について、例を示します。

test.asm と呼ばれる次のアセンブリ・ソース・ファイルを見てみましょう。

```
          .global GLOBAL
          .global FUNC
CONSTANT  .set   1
          .text
START     MOV    AR1,AR0
          ADD    #CONSTANT,AC0
last      ADD    #GLOBAL,AC0
          .data
          .word  4
foo       .word  1
          .word  FUNC
```

シンボル GLOBAL および FUNC は、test2.asm で定義されます。

```
          .global GLOBAL
          .global FUNC
GLOBAL    .set   100
FUNC:     RETURN
```

以下の例では、test.asm および test2.asm がアセンブルされ、以下のコマンドを使ってリンクされたと想定しています。

```
masm55 -qs test.asm
masm55 -qs test2.asm
cl55 -z -q test.obj test2.obj -o test.out
```

❑ オブジェクト・ファイルの標準逆アセンブリ・リストを作成するには、以下のように入力します。

```
dis55 test.obj
TMS320C55x COFF Disassembler             Version x.xx
Copyright (c) 1996-2001  Texas Instruments Incorporated
Disassembly of test.obj:

TEXT Section .text, 0x8 bytes at 0x0
000000:           START:
000000: 2298         MOV    AR1,AR0
000002: 4010         ADD    #1,AC0
000004:              last:
000004: 7b000000     ADD    #0,AC0,AC0

DATA Section .data, 0x3 words at 0x0
000000: 0004         .word  0x0004
000001:           foo:
000001: 0001         .word  0x0001
000002: 0000         .word  0x0000
```

値 1 は最初の ADD 命令にエンコードされ、16 ビット ADD 命令が使われたことに注意してください。2 番目の ADD 命令には、グローバル・シンボル GLOBAL が使用されることにより、アセンブラは 32 ビット ADD 命令を使いました。シンボル GLOBAL および FUNC は、リンカによって解決されます。

- -c オプションを使って COFF ファイル情報を表示できます。-q オプションは、見出しの出力を抑止します。

dis55 -qc test.obj

```
>> Target is C55x Phase 3, mem=small, call=c55_std
   Relocation information may exist in file
   File is not executable
   Line number information may be present in the file
   Local symbols may be present in the file

TEXT Section .text, 0x8 bytes at 0x0
000000:           START:
000000: 2298         MOV    AR1,AR0
000002: 4010         ADD    #1,AC0
000004:           last:
000004: 7b000000     ADD    #0,AC0,AC0

DATA Section .data, 0x3 words at 0x0
000000: 0004         .word  0x0004
000001:           foo:
000001: 0001         .word  0x0001
000002: 0000         .word  0x0000
```

- 実行可能なファイルの標準逆アセンブリ・リストを作成するには、以下のように入力します。

dis55 -q test.out

```
TEXT Section .text, 0xB bytes at 0x100
000100:           START:
000100: 2298         MOV AR1,AR0
000102: 4010         ADD       #1,AC0
000104:           last:
000104: 7b006400     ADD       #100,AC0,AC0
000108:           FUNC:
000108: 4804         RET
00010a: 20              NOP
00010b:           _ _ _etext_ _:
00010b:           etext:

DATA Section .data, 0x3 words at 0x8000
008000: 0004         .word  0x0004
008001:           foo:
008001: 0001         .word  0x0001
008002: 0108         .word  0x0108
```

逆アセンブリ・リストは、ADD 命令および最後の .word 疑似命令に、命令およびデータの使用するアドレスを表示します。これは、解決済みシンボル値と同様です。.word 疑似命令には関数の適切なアドレスが含まれています。.text セクションの NOP は、セクションを埋め込むために使われます。

第 13 章

オブジェクト ファイル ユーティリティの説明

本章では、以下に示すその他のユーティリティの起動方法について説明します。

- **オブジェクト・ファイル表示ユーティリティ**は、オブジェクト・ファイル、実行可能なファイル、およびアーカイブ・ライブラリを、人間の読めるフォーマットおよび XML フォーマットの両方に出力します。
- **名前ユーティリティ**は、COFF オブジェクトまたは実行可能なファイルで定義および参照される名前の一覧を出力します。
- **ストリップ・ユーティリティ**は、オブジェクトおよび実行可能なファイルからシンボル・テーブルおよびデバッグ情報を削除します。

項目 **ページ**

13.1 オブジェクト・ファイル表示ユーティリティの起動方法

オブジェクト・ファイル表示ユーティリティ *ofd55* は、オブジェクト・ファイル（.obj）、実行可能なファイル（.out）、およびアーカイブ・ライブラリ（.lib）を、人間の読めるフォーマットおよび XML フォーマットの両方に出力するために使われます。

オブジェクト・ファイル表示ユーティリティを起動するには、次のように入力します。

ofd55 [*-options*] *input filename* [*input filename*]

ofd55 オブジェクト・ファイル表示ユーティリティを起動するコマンドです。

input filenames アセンブリ言語ソース・ファイルの名前を指定します。ファイル名には .asm 拡張子を付ける必要があります。

options 使用するオブジェクト・ファイル表示ユーティリティ・オプションを指定します。オプションでは大文字と小文字は区別されません。オプションは、コマンドの後ならそのコマンド行のどこにでも指定できます。個々のオプションの前にはハイフンを付けます。

- **-g** プログラム出力に DWARF デバッグ情報を追加します。
- **-o***filename* 画面でなくファイル名にプログラム出力を送ります。
- **-x** 出力を XML フォーマットで表示します。

オプションを使わずにオブジェクト・ファイル表示ユーティリティを起動する場合、コンソール画面には入力ファイルの内容に関する情報が表示されます。

13.2 XML タグ・インデックス

表 13–1 に、オブジェクト・ファイル表示ユーティリティの生成する XML タグについて示します。

表 13–1. XML タグ・インデックス

タグ名	コンテキスト	説明
<addr>	<line_entry>	PC アドレス
	<row>	PC アドレス
	<value>	マシン・アドレス
<addr_class>	<value>	アドレス・クラス
<addr_size>	<compile_unit>	1 マシン・アドレスのサイズ（8 ビット）
	<section>	1 マシン・アドレスのサイズ（8 ビット）
<alignment>	<section>	境界位置合わせ係数
<archive>	<ofd>	アーカイブ・ファイル（.lib）
<attribute>	<die>	DWARF DIE の属性
<aux_count>	<symbol>	このシンボルの補足エントリの数
<banner>	<ofd>	ツール名とバージョン情報
<block>	<section>	境界位置合わせがブロック化係数として使われる場合に真
	<value>	データ・ブロック
<bss>	<section>	このセクションに初期化されないデータが含まれている場合に真
<bss_size>	<optional_file_header>	初期化されないデータのサイズ
<byte_swapped>	<file_header>	作成ホストのエンディアンが現行ホストの逆である
<clink>	<section>	このセクションが条件付きでリンクされている場合に真
<column>	<line_entry>	ソース・カラム番号
<compile_unit>	<section>	コンパイル・ユニット
<const>	<value>	定数
<copy>	<section>	このセクションがコピー・セクションである場合に真
<copyright>	<ofd>	著作権表示
<cpu_flags>	<file_header>	CPU ags
<data>	<section>	このセクションに初期化されたデータが含まれている場合に真

表 13–1. XML タグ・インデックス

タグ名	コンテキスト	説明
<data_size>	<optional_file_header>	初期化されたデータのサイズ
<data_start>	<optional_file_header>	初期化されたデータの開始アドレス
<destination>	<register>	転送先レジスタ
<die>	<compile_unit>	DWARF デバッグ情報エントリ（DIE）
<dim_bound>	<dimension>	サイズの上限
<dim_num>	<dimension>	サイズ数
<dimension>	<symbol>	配列のサイズ
<disp>	<reloc_entry>	余分のアドレス・エンコード情報
<dummy>	<section>	このセクションがダミー・セクションである場合に真
<dwarf>	<ti_coff>	DWARF 情報
<endian>	<file_header>	ターゲット・マシンのエンディアン
<entry_point>	<optional_file_header>	実行可能なプログラムのエントリー・ポイント
<exec>	<file_header>	このファイルが実行可能である場合に真
<fde>	<section>	DWARF フレーム概要エントリ（FDE）
<field_size>	<reloc_entry>	再配置するフィールドのサイズ
<file_header>	<ti_coff>	COFF ファイル・ヘッダ
<file_length>	<file_header>	このファイルのサイズ
<file_name>	<line_entry>	ソース・ファイルの名前
	<symbol>	ソース・ファイルの名前
<file_offsets>	<section>	このセクションに関連するファイル・オフセット
<flag>	<value>	フラグ
<form>	<attribute>	属性形式
<frame_size>	<symbol>	関数フレームのサイズ
<function>	<line_numbers>	1 つの関数に対する行番号エントリ
<icode>	<section>	このセクションに関連する I コードがある場合に真
<index>	<symbol>	シンボル・テーブルにおけるこのシンボルのインデックス

表 13–1. XML タグ・インデックス

タグ名	コンテキスト	説明
<indirect_register>	<memory>	転送先アドレスの演算に使用する間接レジスタ
<initial_location>	<fde>	FDE によって参照される関数の先頭
<internal>	<reloc_entry>	この再配置が内部にある場合に真
<kind>	<symbol>	シンボルの種類（定義済み、未定義、絶対、シンボリック・デバッグ）
<length>	<symbol>	セクションの長さ
<line>	<line_entry>	ソース行番号
	<symbol>	このシンボルに関連する最初のソース行
<line_count>	<section>	行番号エントリの数
	<symbol>	行番号エントリの数
<line_entry>	<compile_unit>	行番号エントリ
	<line_numbers>	行番号エントリ
<line_numbers>	<section>	行番号エントリ
<line_ptr>	<file_offsets>	行番号エントリのファイル・オフセット
	<symbol>	行番号エントリのファイル・オフセット
<lnno_strip>	<file_header>	行番号がこのファイルから除去された場合に真
<localsym_strip>	<file_header>	ローカル・シンボルがこのファイルから除去された場合に真
<magic>	<optional_file_header>	オプショナル・ファイル・ヘッダのマジック・ナンバ（0x0108）
<math_relative>	<reloc_entry>	この再配置が数学的に相対的である場合に真
<memory>	<row>	SOE レジスタがメモリに保存される
<name>	<fde>	FDE によって参照される関数の名前
	<function>	現行の関数の名前
	<ofd>	オブジェクトまたはアーカイブ・ファイルの名前
	<section>	このセクションの名前
	<symbol>	このシンボルの名前
<next_symbol>	<symbol>	マルチシンボル・エンティティの後の次のシンボルのインデックス

表 13–1. XML タグ・インデックス

タグ名	コンテキスト	説明
<noload>	<section>	このセクションが非ロード・セクションである場合に真
<object_file>	<ofd>	オブジェクト・ファイル（.obj、.out）
<ofd>		オブジェクト・ファイル表示（OFD）ドキュメント
<offset>	<memory>	間接レジスタから転送先アドレスのオフセット
	<reloc_entry>	再配置可能アドレスからフィールドのオフセット
<optional_file_header>	<ti_coff>	オプショナル・ファイル・ヘッダ
<padded>	<section>	このセクションが埋め込まれた場合に真（C55x のみ）
<page>	<section>	メモリ・ページ
<pass>	<section>	このセクションが変更なしで渡される場合に真
<physical_addr>	<section>	セクションの物理的（実行）アドレス
<raw_data_ptr>	<file_offsets>	生データのファイル・オフセット
<raw_data_size>	<section>	生データのサイズ（8 ビット）
<ref>	<value>	参照
<register>	<row>	SOE レジスタがレジスタに保存される
<register_mask>	<symbol>	保存された SOE レジスタのマスク
<regular>	<section>	このセクションが通常セクションである場合に真
<reloc_count>	<section>	再配置エントリの数
	<symbol>	再配置エントリの数
<reloc_entry>	<relocations>	再配置エントリ
<reloc_ptr>	<file_offsets>	再配置エントリのファイル・オフセット
<reloc_strip>	<file_header>	再配置情報がこのファイルから除去された場合に真
<relocations>	<section>	再配置エントリ
<return_address_register>	<fde>	この関数のリターン・アドレスを渡すために使われるレジスタ
<row>	<table>	テーブルの列
<section>	<dwarf>	DWARF セクション

表 13–1. XML タグ・インデックス

タグ名	コンテキスト	説明
	<symbol>	このシンボルの定義を含むセクション
	<ti_coff>	COFF セクション
<section_count>	<file_header>	セクション・ヘッダの数
<size_in_addrs>	<symbol>	関数におけるマシン・アドレス・サイズのユニットの数
<size_in_bits>	<symbol>	シンボルのサイズ（ビット）
<source>	<memory>	ソース・レジスタ
	<register>	ソース・レジスタ
<start_symbol>	<symbol>	マルチ・シンボル・エンティティにおける最初のシンボル
<storage_class>	<symbol>	このシンボルの記憶クラス
<storage_type>	<symbol>	このシンボルの記憶の種類
<string>	<string_table>	文字列テーブル・エントリ
	<value>	文字列
<string_table>	<ti_coff>	文字列テーブル
<string_table_size>	<string_table>	文字列テーブルのサイズ
<sym_merge>	<file_header>	デバッグ・タイプ・シンボルがマージされた場合に真
<symbol>	<symbol_table>	シンボル・テーブル・エントリ
<symbol_count>	<file_header>	シンボル・テーブル内のエントリの数
<symbol_relative>	<reloc_entry>	再配置が指定のシンボルに関連している
<symbol_table>	<ti_coff>	シンボル・テーブル
<table>	<fde>	FDE テーブル
<tag>	<die>	タグ名
<tag_index>	<symbol>	ユーザー定義の種類に関連
<target_id>	<file_header>	ターゲット ID。ターゲット・マシンを識別するマジック・ナンバ
<text>	<section>	このセクションにコードが含まれている場合に真
<text_size>	<optional_file_header>	実行可能なコードのサイズ

表 13–1. XML タグ・インデックス

タグ名	コンテキスト	説明
<text_start>	<optional_file_header>	実行可能なコードの開始アドレス
<ti_coff>	<object_file>	TI COFF ファイル
<tool_version>	<optional_file_header>	ツール・バージョン・スタンプ
<type>	<attribute>	属性タイプ
	<reloc_entry>	再配置のタイプ
<type_ref>	<value>	タイプの参照
<value>	<attribute>	属性値
	<reloc_entry>	値
	<symbol>	値
<vector>	<section>	このセクションにベクトル・テーブルが含まれている場合に真（C55x のみ）
<version>	<compile_unit>	DWARF バージョン
	<file_header>	バージョン ID。COFF ファイルの構造バージョン
<virtual_addr>	<reloc_entry>	再配置される仮想アドレス
	<section>	セクションの仮想（ロード）アドレス
<word_size>	<reloc_entry>	再配置可能なフィールドを含むアドレス・サイズ・ユニットの数
<xml_version>	<dwarf>	DWARF XML 言語のバージョン
	<ti_coff>	COFF XML 言語のバージョン

13.3 XML コンシューマの例

このセクションでは、ofd 55 の XML 出力を使ってオブジェクト・ファイルに含まれる実行可能なコードのサイズを計算する、小さなアプリケーション例を示します。

この例には、3 つのソース・ファイルが含まれます。codesize.cpp、xml.h、および xml.cpp です。実行可能な名前付きコードサイズにコンパイルされると、ofd55 を使ってコマンド行から以下のように使用できます。

```
% ofd55 -x a.out | codesize

Code Section Name: .text
Code Section Size: 44736

Code Section Name: .text2
Code Section Size: 64

Code Section Name: .text3
Code Section Size: 64

Total Code Size: 44864
```

13.3.1 主要なアプリケーション

codesize.cpp ファイルには、オブジェクト・ファイル表示ユーティリティ例のための主要なアプリケーションがあります。

```
//****************************************************************************
// CODESIZE.CPP - An example application that calculates the size of the      *
// executable code in an object file using the XML output                     *
// of the OFD utility.                                                        *
//****************************************************************************
#include "xml.h"
#include <iostream>

using namespace std;

static void parse_XML_prolog(istream &in);

//****************************************************************************
// main() - List the names and sizes of the code sections (in octets), and    *
//          output the total code size.                                       *
//****************************************************************************
int main()
{
   //-------------------------------------------------------------------------
   // Build our tree of XML Entities from standard input (See xml.{cpp,h} for -
   // the definition of the XMLEntity object).                                -
   //-------------------------------------------------------------------------
   parse_XML_prolog(cin);
   XMLEntity *root = new XMLEntity(cin);
```

```
//-------------------------------------------------------------------------
// Fetch the XML Entities of the section roots. In other words, get a      -
// list of all the XMLEntity sub-trees named "section" that are in the     -
// context of "ofd->object_file->ti_coff", where "ofd" is the root of our  -
// XML document.                                                           -
//-------------------------------------------------------------------------
CEntityList    query_result;
const char *section_query[] =
   { "ofd", "object_file", "ti_coff", "section", NULL };

query_result = root->query(section_query);

//-------------------------------------------------------------------------
// Iterate over the section Entities, looking for code sections.           -
//-------------------------------------------------------------------------
CEntityList_CIt pit;
unsigned long total_code_size = 0;

for (pit = query_result.begin(); pit != query_result.end(); ++pit)
{

   //----------------------------------------------------------------------
   // Query for the name, text, and raw_data_size sub-entities of each     -
   // section. XMLEntity::query() always returns a list, even if there     -
   // will only ever be a maximum of one result. If the tag is not         -
   // found, an empty list is returned.                                    -
   //----------------------------------------------------------------------
   const char *section_name_query[] = { "section", "name",          NULL };
   const char *section_text_query[] = { "section", "text",          NULL };
   const char *section_size_query[] = { "section", "raw_data_size", NULL };



   CEntityList sname_l;
   CEntityList stext_l;
   CEntityList ssize_l;

   sname_l = (*pit)->query(section_name_query);
   stext_l = (*pit)->query(section_text_query);
   ssize_l = (*pit)->query(section_size_query);
   //----------------------------------------------------------------------
   // If a "text" flag was found, this is a code section. Output           -
   // the section name and size, and add its size to our total code size   -
   // counter.                                                             -
   //----------------------------------------------------------------------
   if (stext_l.size() > 0)
   {
      unsigned long size;

      size = strtoul((*ssize_l.begin())->value().c_str(), NULL, 16);

      cout << "Code Section Name:  " << (*sname_l.begin())->value() << endl;
      cout << "Code Section Size:  " << size << endl;
      cout << endl;

      total_code_size += size;
   }
```

```
   }

   //-------------------------------------------------------------------------
   // Output the total code size, and clean up. -
   //-------------------------------------------------------------------------
   cout << "Total Code Size: " << total_code_size << endl;
   delete root;

   return 0;
}


//*************************************************************************
// parse_XML_prolog() - Parse the XML prolog, and throw it away. *
//*************************************************************************
static void parse_XML_prolog(istream &in)
{
   char c;

   while (true)
   {
      //-------------------------------------------------------------------
      // Look for the next tag; if it is not an XML directive, we're done.    -
      //-------------------------------------------------------------------
      for (in.get(c); c != '<' && !in.eof(); in.get(c))
      ; // empty body

      if (in.eof()) return;
      if (in.peek() != '?') { in.unget(); return; }

      //-------------------------------------------------------------------
      // Otherwise, read in the directive and continue. -
      //-------------------------------------------------------------------
      for (in.get(c); c != '>' && !in.eof(); in.get(c))
         ; // empty body
   }
}
```

13.3.2 XMLEntity オブジェクトの xml.h 宣言

xml.h ファイルには、codesize.cpp アプリケーションのための XMLEntity オブジェクトの宣言があります。

```
//*************************************************************************
// XML.H - Declaration of the XMLEntity object. *
//*************************************************************************
#ifndef XML_H
#define XML_H
#include <list>
#include <string>


//*************************************************************************
// Type Declarations. *
//*************************************************************************
class XMLEntity;
typedef list<XMLEntity*>            EntityList;
typedef list<const XMLEntity*>      CEntityList;
typedef CEntityList::const_iterator CEntityList_CIt;
typedef EntityList::const_iterator  EntityList_CIt;
//*************************************************************************
// CLASS XMLENTITY - A Simplified XML Entity Object.                      *
//*************************************************************************
class XMLEntity
{
   public:
      XMLEntity  (istream &in, XMLEntity *parent=NULL);
      ~XMLEntity ();
      const CEntityList  query  (const char **context) const;
      const string      &tag    () const { return tag_m;      }
      const string      &value  () const { return value_m;    }

   private:
      void  parse_raw_tag (const string &raw_tag);
      void  sub_query     (CEntityList &result, const char **context) const;

      string      tag_m;       // Tag Name
      string      value_m;     // Value
      XMLEntity  *parent_m;    // Pointer to parent in XML hierarchy
      EntityList  children_m;  // List of children in XML hierarchy
};
#endif
```

13.3.3 XMLEntity オブジェクトの xml.cpp 定義

xml.cpp ファイルには、codesize.cpp アプリケーションのための XMLEntity オブジェクトの定義があります。

```
//**************************************************************************
// XML.CPP - Definition of the XMLEntity object. *
//**************************************************************************
#include "xml.h"
#include <iostream>
#include <string>
#include <list>
#include <cstdlib>

//**************************************************************************
// XMLEntity::query() - Return the list of XMLEntities a list that reside *
// in the given XML context. *
//**************************************************************************
const CEntityList XMLEntity::query(const char **context) const
{
   CEntityList result;

   if (!*context) return result;

   sub_query(result, context);

   return result;
}

//**************************************************************************
// XMLEntity::sub_query() - Recurse through the XML tree looking for a match  *
//                          to the current query.                             *
//**************************************************************************
void XMLEntity::sub_query(CEntityList &result, const char **context) const
{
   if (!context[0] || tag() != context[0]) return;

   if (!context[1])
      result.push_front(this);
   else
   {
      EntityList_CIt pit;

      for (pit = children_m.begin(); pit != children_m.end(); ++pit)
         (*pit)->sub_query(result, context+1);
   }
   return;
}


//**************************************************************************
// XMLEntity::parse_raw_tag() - Cut out the tag name from the complete string *
// we found between the < > brackets. This throws out any attributes.         *
//**************************************************************************
void XMLEntity::parse_raw_tag(const string &raw_tag)
```

```
{
   string attribute;
   int    i;

   for (i = 0; i < raw_tag.size() && raw_tag[i] != ' '; ++i)
      tag_m += raw_tag[i];
}

//****************************************************************************
// XMLEntity::XMLEntity() - Recursively construct a tree of XMLEntities from *
//                          the given input stream.                           *
//****************************************************************************
XMLEntity::XMLEntity(istream &in, XMLEntity *parent) :
tag_m(""), value_m(""), parent_m(parent)
{
   string raw_tag;
   char   c;
   int    i;
   //-------------------------------------------------------------------------
   // Read in the leading '<'.                                               -
   //-------------------------------------------------------------------------
   in.get();

   //-------------------------------------------------------------------------
   // Store the tag name and attributes in "raw_tag", then call              -
   // process_raw_tag() to separate the tag name from the attributes and     -
   // store it in tag_m.                                                     -
   //-------------------------------------------------------------------------
   for (in.get(c); c != '>' && c != '/' && !in.eof(); in.get(c))
      raw_tag += c;

   parse_raw_tag(raw_tag);

   //-------------------------------------------------------------------------
   // If we're reading in an end-tag, read in the closing '>' and return.    -
   //-------------------------------------------------------------------------
   if (c == '/') { in.get(c); return; }

   //-------------------------------------------------------------------------
   // Otherwise, parse our value.                                            -
   //-------------------------------------------------------------------------
   while (true)
   {
      //----------------------------------------------------------------------
      // Read in the closing '>', then start reading in characters and add   -
      // them to value_m. Stop when we hit the beginning of a tag.           -
      //----------------------------------------------------------------------
      for (in.get(c); c != '<'; in.get(c)) value_m += c;

      //----------------------------------------------------------------------
      // If we're reading in a start tag, parse in the entire entity, and    -
      // add it to our child list (recursive constructor call).              -
      //----------------------------------------------------------------------
      if (in.peek() != '/')
      {


         //-------------------------------------------------------------------
         // Put back the opening '<', since XMLEntity() expects to read it.  -
```

```
         //---------------------------------------------------------------------
      in.unget();
      children_m.push_front(new XMLEntity(in, this));
      }
      //-----------------------------------------------------------------------
      // Otherwise, read in our end tag, and exit.                           -
      //-----------------------------------------------------------------------
      else
      {
         for (in.get(c); c != '>'; in.get(c))
            ; // empty body
         break;
      }
   }

   //-------------------------------------------------------------------------
   // Strip off leading and trailing white space from our value.           -
   //-------------------------------------------------------------------------
   for (i = 0; i < value_m.size(); i++)
      if (value_m[i] != ' ' && value_m[i] != '\n') break;
   value_m.erase(0, i);

   for (i = value_m.size()-1; i >= 0; i--)
      if (value_m[i] != ' ' && value_m[i] != '\n') break;
   value_m.erase(i+1, value_m.size()-i);
}

//*************************************************************************
// XMLEntity::~XMLEntity() - Delete a XMLEntity object.                   *
//*************************************************************************
XMLEntity::~XMLEntity()
{
   EntityList_CIt pit;

   for (pit = children_m.begin(); pit != children_m.end(); ++pit)
      delete (*pit);
}
```

13.4 名前ユーティリティの起動方法

名前ユーティリティ *nm55* は、COFF オブジェクト（.obj）または実行可能なファイル（.out）で定義および参照される名前の一覧を出力するために使われます。シンボルに関連する値およびシンボルの種類の表示も出力されます。

名前ユーティリティを起動するには、次のように入力します。

nm55 [*-options*] [*input filename*]

nm55 名前ユーティリティを起動するためのコマンドです。

input filename COFF オブジェクト・ファイル（.obj）、実行可能なファイル（.out）、またはアーカイブ・ファイルです。アーカイブ・ファイルについては、名前ユーティリティがアーカイブの各オブジェクト・ファイルを処理します。

options 使用する名前ユーティリティ・オプションを指定します。オプションでは大文字と小文字は区別されません。オプションは、起動の後ならそのコマンド行のどこにでも指定できます。各オプションの前にはハイフン（-）を付けます。名前ユーティリティ・オプションは次のとおりです。

- **-a** すべてのシンボルを出力します。
- **-c** C_NULL シンボルも出力します。
- **-d** デバッグ・シンボルも出力します。
- **-f** 各シンボルの前にファイル名を付けます。
- **-g** グローバル・シンボルだけを出力します。
- **-h** 現在のヘルプ画面を表示します。
- **-l** シンボル情報の詳細リストを作成します。
- **-n** シンボルを、アルファベット順でなく数字順にソートします。
- **-o***file* 特定のファイルに出力します。
- **-p** 名前ユーティリティにシンボルをソートさせません。
- **-q** （静的なモード）見出しとすべての進捗情報を出力しません。
- **-r** シンボルを逆順でソートします。
- **-t** タグ情報シンボルも出力します。
- **-u** 未定義シンボルだけを出力します。

13.5 ストリップ・ユーティリティの起動方法

ストリップ・ユーティリティ *strip55* は、オブジェクトおよび実行可能なファイルからシンボル・テーブルおよびデバッグ情報を削除するために使われます。

ストリップ・ユーティリティを起動するには、次のように入力します。

strip55 [-p] *input filename* [*input filename*]

strip55 ストリップ・ユーティリティを起動するためのコマンドです。

input filename COFF オブジェクト・ファイル（.obj）または実行可能なファイル（.out）です。

options 使用するストリップ・ユーティリティ・オプションを指定します。オプションでは大文字と小文字は区別されません。オプションは、起動の後ならそのコマンド行のどこにでも指定できます。各オプションの前にはハイフン（-）を付けます。名前ユーティリティ・オプションは次のとおりです。

-p 実行に必要ない情報をすべて削除します。このオプションにより、デフォルトの動作より多くの情報が削除されますが、オブジェクト・ファイルはリンクできない状態のままです。このオプションは、実行可能な（.out）ファイルでのみ使用する必要があります。

ストリップ・ユーティリティを起動すると、入力オブジェクト・ファイルが除去されたバージョンに置き換えられます。

第 14 章

Hex 変換ユーティリティの説明

TMS320C55x のアセンブラおよびリンカは、共通オブジェクト・ファイル・フォーマット（COFF）のオブジェクト・ファイルを作成します。COFF は、モジュラ・プログラミングを促進するとともに、コード・セグメントおよびターゲット・システム・メモリを管理するための強力で柔軟な方式を提供する 2 進オブジェクト・ファイル・フォーマットです。

EPROM プログラマの多くは、COFF オブジェクト・ファイルを入力として受け入れません。Hex 変換ユーティリティは、COFF オブジェクト・ファイルを、EPROM プログラマへのロードに適したいくつかの標準 ASCII 16 進フォーマットのいずれかに変換します。このユーティリティは、COFF オブジェクト・ファイルの 16 進変換を必要とするその他のアプリケーション（デバッガやローダなど）を使用する場合にも役立ちます。このユーティリティは、ターゲット・デバイスに組み込まれているオンチップ・ブート・ローダもサポートしているため、C55x のコード生成プロセスは自動化されます。

Hex 変換ユーティリティでは、次の出力ファイル・フォーマットを作成できます。

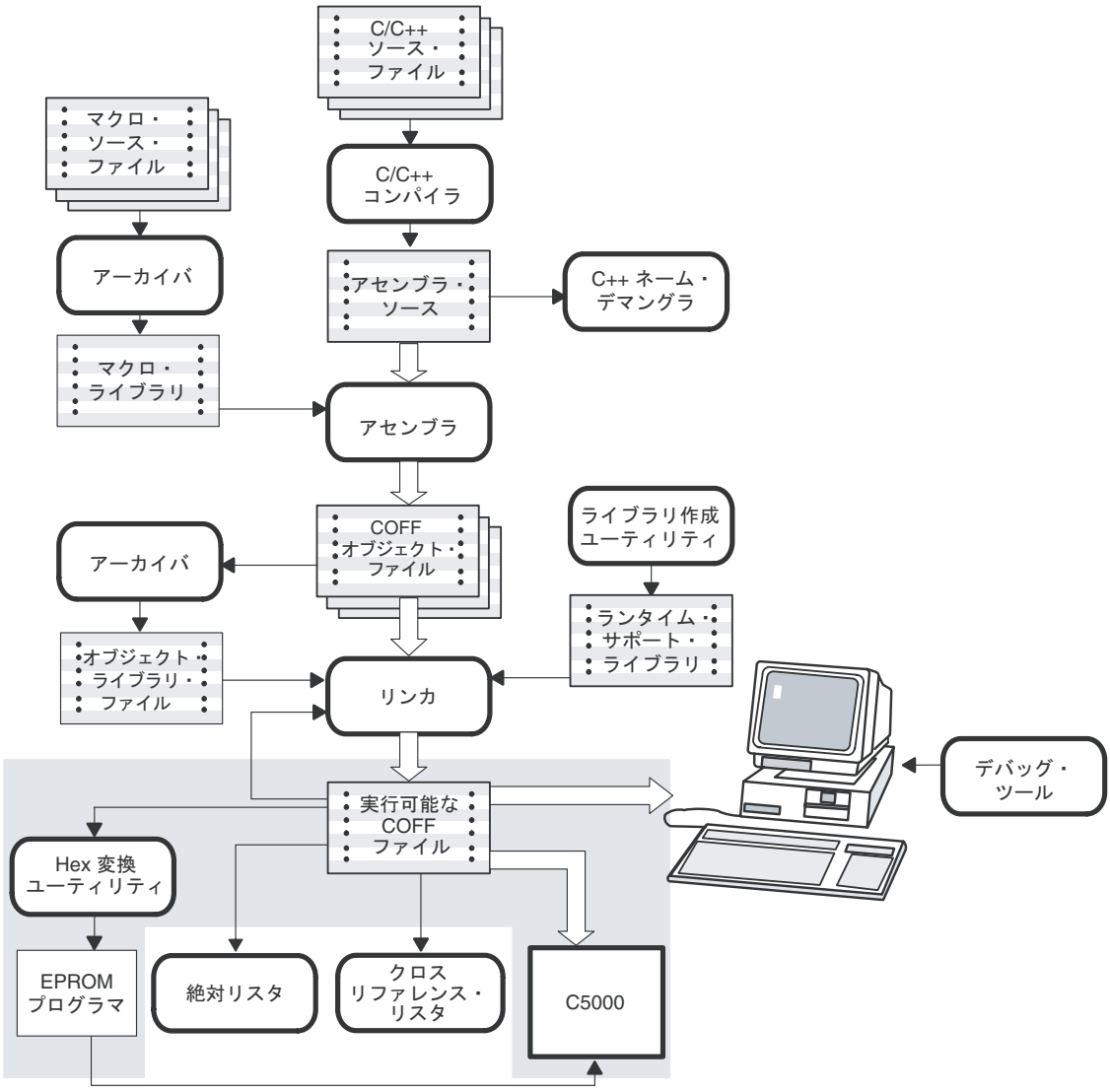
- ❏ ASCII-Hex、16 ビット・アドレスをサポート
- ❏ Extended Tektronix（Tektronix）
- ❏ Intel MCS-86（Intel）
- ❏ Motorola Exorciser（Motorola-S）、16 ビット、24 ビット、および 32 ビット・アドレスをサポート
- ❏ Texas Instruments SDSMAC（TI-Tagged）、16 ビット・アドレスをサポート

項目 **ページ**

14.1 Hex 変換ユーティリティ開発のフロー

図 14-1 は、アセンブリ言語開発プロセスにおける Hex 変換ユーティリティの役割を示したものです。

図 14-1. Hex 変換ユーティリティ開発のフロー



14.2 Hex 変換ユーティリティの起動方法

Hex 変換ユーティリティを起動するには、次の 2 つの基本的な方法があります。

- **コマンド行でオプションとファイル名を指定します。**次の例では、ファイル firmware.out を TI-Tagged フォーマットに変換し、firm.lsb と firm.msb という 2 つの出力ファイルを作成します。

  ```
  hex55 -t firmware -o firm.lsb -o firm.msb
  ```

- **コマンド・ファイルにオプションとファイル名を指定します。**Hex 変換ユーティリティを起動するためのコマンド行オプションとファイル名を格納したバッチ・ファイルを作成できます。次の例では、hexutil.cmd というコマンド・ファイルを使用してユーティリティを起動しています。

  ```
  hex55 hexutil.cmd
  ```

  コマンド・ファイルの中では、通常のコマンド行情報のほかに、Hex 変換ユーティリティの ROMS 疑似命令と SECTIONS 疑似命令を使用できます。

Hex 変換ユーティリティを起動するには、次のように入力します。

hex55 [*-options*] *filename*

hex55 Hex 変換ユーティリティを起動するコマンドです。

-options Hex 変換ユーティリティの処理を制御する追加情報を指定します。オプションは、コマンド行で使用できるほか、コマンド・ファイルの中でも使用できます。

- オプションの前には必ずダッシュを付けます。また、どのオプションでも大文字と小文字は区別されません。
- 一部のオプションには追加のパラメータがあります。これらのパラメータは、1 つ以上の空白でオプションから区切る必要があります。
- 複数文字のオプション名は、本書に記載されているとおり正確に指定する必要があります。省略形は使用できません。
- オプションの指定順序は各オプションに影響を与えることはありません。ただし -q オプションは例外です。-q オプションは、他のどのオプションよりも前に指定しなければなりません。

filename COFF オブジェクト・ファイルまたはコマンド・ファイルの名前を指定します（コマンド・ファイルの詳細は、14.3 節「コマンド・ファイル」（14-6 ページ）を参照してください）。

表 14–1. Hex 変換ユーティリティ・オプション

(a) 一般オプションは、Hex 変換ユーティリティの動作全体を制御します。

オプション	説明	ページ
-exclude *section_name*	指定セクションを無視します。	14-23
-map *filename*	マップ・ファイルを作成します。	14-20
-o *filename*	出力ファイル名を指定します。	14-24
-q	静的な実行（使用するときは他のオプションより前に指定する必要があります)。	14-6

(b) イメージ・オプションは、ある範囲のターゲット・メモリを連続したイメージとして作成します。

オプション	説明	ページ
-fill *value*	ホールに値を埋め込みます。	14-27
-image	イメージ・モードを指定します。	14-26
-zero	アドレス起点をゼロにリセットします。	14-35

(c) メモリ・オプションは、出力ファイルのメモリ幅を設定します。

オプション	説明	ページ
-memwidth *value*	システム・メモリのワード幅を定義します（デフォルト値 = 8 ビット)。	14-9
-order {LS | MS}	メモリ・ワードの順番を指定します。	14-13
-romwidth *value*	ROM のデバイス幅を指定します（デフォルト値は、使用するフォーマットにより異なります)。	14-10

例 14–1. Hex 変換ユーティリティ・オプション（続き）

(d) 出力フォーマットは、出力ファイルのフォーマットを指定します。

オプション	説明	ページ
-a	ASCII-Hex を選択します。	14-39
-b	バイナリを選択します。	
-i	Intel を選択します。	14-40
-m1	Motorola-S1 を選択します。	14-41
-m2 または -m	Motorola-S2 を選択します（デフォルト）。	14-41
-m3	Motorola—S3 を選択します。	14-41
-t	TI-Tagged を選択します。	14-42
-x	Tektronix を選択します。	14-43

(e) すべての C55x デバイス用のブートローダ・オプションは、Hex 変換ユーティリティを使用してブート・テーブルを作成する方法を制御します。

オプション	説明	ページ
-boot	全セクションをブート可能な形式に変換します（SECTIONS 疑似命令の代わりに使用します）。	14-31
-bootorg *value*	ブート・ローダ・テーブルのソース・アドレスを指定します。	14-31
-bootpage *value*	ブート・ローダ・テーブルのターゲット・ページ番号を指定します。	14-31
-e *value*	ブート・ロード後に実行を始めるエントリ・ポイントを指定します。*value* はアドレスまたはグローバル・シンボルになります。	14-32
-parallel16	16 ビットのパラレル・インタフェース・ブート・テーブルの指定（-memwidth 16 および -romwidth 16）	14-32
-parallel32	32 ビットのパラレル・インタフェース・ブート・テーブルの指定（-memwidth 16 および -romwidth 32）	14-32
-serial8	8 ビットのシリアル・インタフェース・ブート・テーブルの指定（-memwidth 8 および -romwidth 8）	14-32
-serial16	16 ビットのシリアル・インタフェース・ブート・テーブルの指定（-memwidth 16 および -romwidth 16）	14-32
-v*device*:*revision*	デバイスおよびシリコン改訂番号の指定	14-33

14.3 コマンド・ファイル

同じ入力ファイルとオプションを指定してこのユーティリティを繰り返し起動する場合は、コマンド・ファイルを使用すると便利です。また、ROMS および SECTIONS の Hex 変換ユーティリティ疑似命令を使用して変換処理をカスタマイズする場合も、コマンド・ファイルが役立ちます。

コマンド・ファイルは、次の情報が 1 つ以上含まれている ASCII ファイルです。

- **オプションおよびファイル名。**これらは、コマンド行で指定する場合と全く同じ方法でコマンド・ファイル内に指定します。

- **ROMS 疑似命令。**ROMS 疑似命令は、システムの物理メモリ構成をアドレス範囲パラメータのリストとして定義します（ROMS 疑似命令の詳細は、14.5 節「ROMS 疑似命令」（14-15 ページ）を参照してください）。

- **SECTIONS 疑似命令。**SECTIONS 疑似命令は、COFF オブジェクト・ファイルからどのセクションを選択するかを指定します（SECTIONS 疑似命令の詳細は、14.6 節「SECTIONS 疑似命令」（14-21 ページ）を参照してください）。

  この疑似命令を使用すると、オンチップ・ブート・ローダで初期化する特定のセクションを識別することもできます（オンチップ・ブート・ローダの詳細は、14.10.3 項「ブート・テーブルの作成方法」（14-29 ページ）を、参照してください）。

- **コメント。**コマンド・ファイルには、/* および */ の区切り記号を使用してコメントを付加できます。次に例を示します。

  ```
  /*   This is a comment   */
  ```

ユーティリティを起動してコマンド・ファイル内で定義したオプションを使用するには、次のように入力します。

```
hex55 command_filename
```

コマンド行には、その他のファイルおよびオプションも指定できます。コマンド・ファイルとコマンド行オプションの両方を使用して、次のようにユーティリティを起動することもできます。

```
hex55 firmware.cmd -map firmware.mxp
```

これらのオプションおよびファイル名を指定する順序に意味はありません。ユーティリティは、変換処理を開始する前にコマンド行からすべての入力を読み込むとともに、コマンド・ファイルからすべての情報を読み込みます。ただし -q オプションを使用する場合は、-q オプションをコマンド行またはコマンド・ファイルの最初のオプションとして指定する必要があります。

-q オプションは、このユーティリティの通常の見出しおよび進捗情報の表示を抑止します。

14.3.1 コマンド・ファイルの例

- firmware.cmd という名前のコマンド・ファイルに以下の行が含まれているとします。

```
firmware.out   /* input file  */
-t             /* TI-Tagged   */
-o   firm.lsb  /* output file 1, LSBs of ROM */
-o   firm.msb  /* output file 2, MSBs of ROM*/
```

次のように入力すれば、Hex 変換ユーティリティを起動できます。

```
hex55 firmware.cmd
```

- この例では、appl.out という名前のファイルを Intel フォーマットの 4 つの Hex ファイルに変換します。各出力ファイルの幅は 1 バイトで、長さは 16K バイトです。.text セクションはブート・ローダ・フォーマットに変換されます。

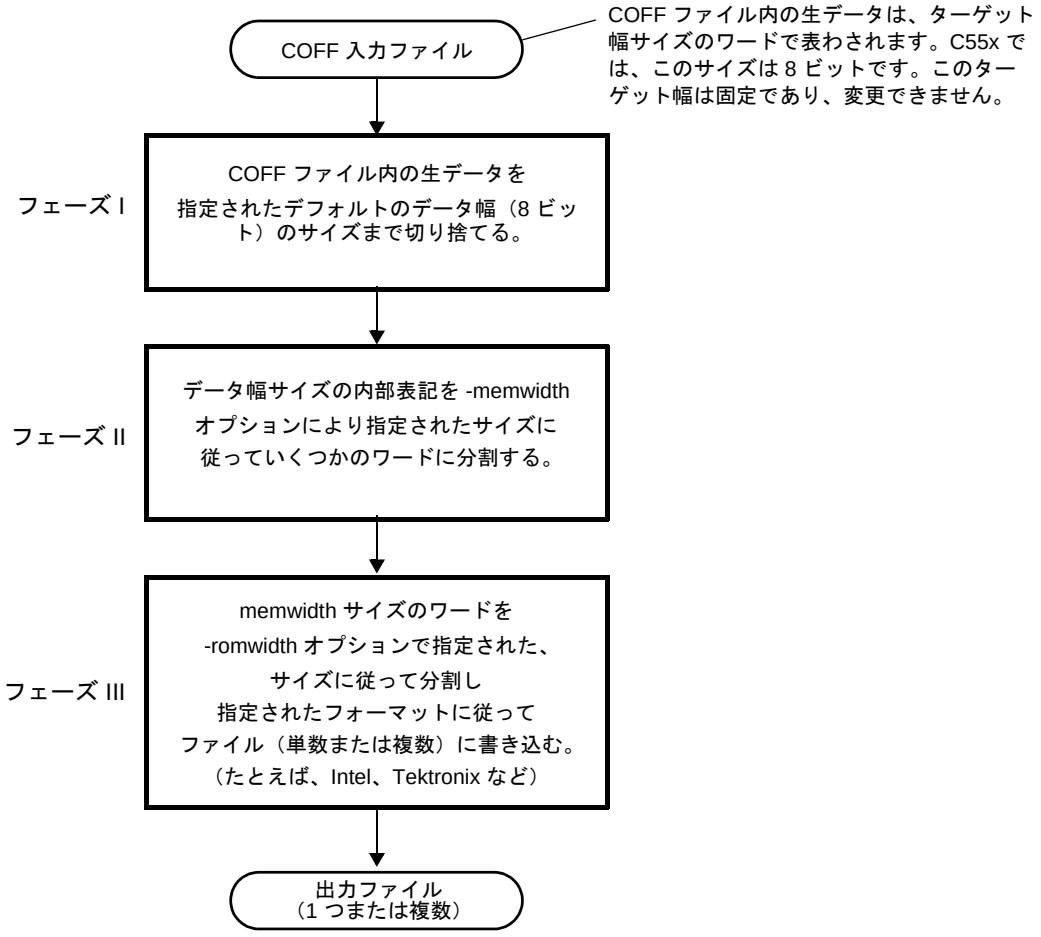
```
appl.out          /* input file   */
-i                /* Intel format */
-map appl.mxp     /* map file     */

ROMS
{
   ROW1:origin=01000h len=04000h romwidth=8
         files={ appl.u0 appl.u1 }
   ROW2:origin 05000h len=04000h romwidth=8
         files={ app1.u2 appl.u3 }
}

SECTIONS
{   .text:BOOT
    .data, .cinit, .sect1, .vectors, .const:
}
```

14.4 メモリ幅について

Hex 変換ユーティリティは、ユーザー自身がメモリ幅と ROM 幅を指定することにより、運用しているメモリ・アーキテクチャの柔軟性をさらに向上させます。Hex 変換ユーティリティを使用するには、ユーティリティがワード幅をどのように扱うかを理解しておく必要があります。変換処理では、4 つの幅が重要な意味をもちます。ターゲット幅、データ幅、メモリ幅、ROM 幅です。ターゲット・ワード、データ・ワード、メモリ・ワード、ROM ワードという用語は、このような幅をもつワードを意味します。

図 14-2 は、Hex 変換ユーティリティの処理フローにおける 3 つの独立した個別フェーズを示した物です。

図 14-2. Hex 変換ユーティリティ処理のフロー



14.4.1 ターゲット幅

ターゲット幅は、COFF ファイル内の生データ・フィールドの単位サイズ（ビット数）です。これは、ターゲット・プロセッサ上の命令コードのサイズに対応します。この幅はターゲットごとに固定であり、変更することはできません。C54x のターゲット幅は 16 ビットです。C55x ターゲットは 16 ビットの幅で表示されます。

14.4.2 データ幅

データ幅は、COFF ファイルの特定のセクションに格納されるデータ・ワードの論理幅（ビット数）です。通常、論理データ幅はターゲット幅と同じになります。TMS320C55x では、データ幅は 16 ビットに固定されていて、変更できません。

14.4.3 メモリ幅

メモリ幅は、メモリ・システムの物理的な幅（ビット数）です。通常、メモリ・システムはターゲット・プロセッサの幅と物理的に同じ幅になります。つまり、16 ビットのプロセッサであれば 16 ビットのメモリ・アーキテクチャをもちます。ただし一部のアプリケーションでは、ターゲット・ワードを分割して連続した複数のより狭い幅のメモリ・ワードを作成する必要があります。さらに、C55x のような特定のプロセッサでは、メモリ幅はターゲット幅より狭くなることがあります。

C55x Hex 変換ユーティリティのデフォルトのメモリ幅は、16 ビットです。

メモリ幅は次の方法で変更できます。

- **-memwidth** オプションの使用。この方法では、ファイル全体についてメモリ幅が変更されます。

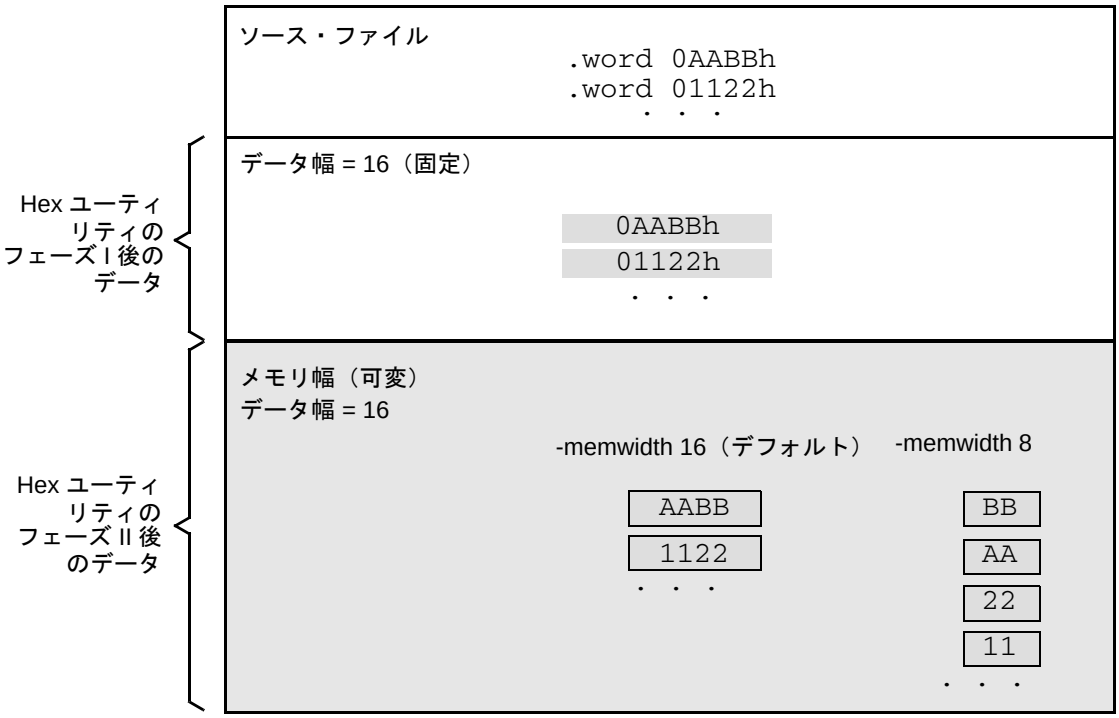
- ROMS 疑似命令の **memwidth** パラメータの設定。この方法では、ROMS 疑似命令で指定したアドレス範囲についてメモリ幅が変更されます。そのアドレス範囲に対する -memwidth オプションは無効になります。14.5 節「ROMS 疑似命令」（14-15 ページ）を参照してください。

どちらの場合も、使用する値は 8 以上の 2 の累乗値でなければなりません。

メモリ幅のデフォルト値である 16 は、たとえば 1 つのターゲット・ワードをそれより狭い幅の連続した複数のメモリ・ワードに分割する必要がある場合などの、例外的な状況がある場合以外は変更しないでください。メモリ・ワードがターゲット・ワードより狭いという状況がよく起きるのはオンチップ・ブート・ローダを使用する場合です。オンチップ・ブート・ローダの中には、ターゲット・ワードよりも狭い幅のメモリ・ワードからブートをサポートするものがいくつかあります。たとえば、16 ビットの TMS320C55x は、8 ビットのメモリまたは 8 ビットのシリアル・ポートからブートできます。この場合、それぞれの 16 ビット値は 2 つのメモリ・ロケーションを占めます（これは -memwidth 8 として指定されます）。

図 14-3 は、メモリ幅がデータ幅とどのように関連するかを示しています。

図 14-3. データ幅とメモリ幅



14.4.4 ROM 幅

ROM 幅は、各 ROM デバイスと対応する出力ファイルの物理的な幅（ビット数）を指定します（通常は 1 バイト、つまり 8 ビットです）。Hex 変換ユーティリティがデータをどのように分割して出力ファイルに入れるかは、ROM 幅により決まります。ターゲット・ワードをメモリ・ワードにマップした後で、メモリ・ワードは 1 つ以上の出力ファイルに分割されます。アドレス領域ごとの出力ファイルの数は、次の式で求めることができます。ただし、メモリ幅≧ ROM 幅です。

ファイル数 = メモリ幅 ÷ ROM 幅

たとえば、メモリ幅が 16 である場合に、ROM 幅の値として 16 を指定すると、16 ビットの各ワードが格納された出力ファイルが 1 つ作成されます。または、ROM 幅の値として 8 を使用して 2 つの出力ファイルを作成し、それぞれのファイルに 8 ビットの各ワードを格納することもできます。

アドレス領域ごとのファイルの計算については、14.5 節「ROMS 疑似命令」（14-15 ページ）を参照してください。

Hex 変換ユーティリティが使用するデフォルトの ROM 幅は、出力フォーマットによって次のように異なります。

- ❏ TI-Tagged フォーマットを除くすべての Hex フォーマットは、8 ビット・バイトのリストとして配置されます。このため、これらのフォーマットのデフォルトの ROM 幅は 8 ビットになります。

- ❏ TI-Tagged は 16 ビット・フォーマットです。TI-Tagged のデフォルトの ROM 幅は 16 ビットです。

注 : TI-Tagged フォーマットは 16 ビット幅

TI-Tagged フォーマットの ROM 幅は変更できません。TI-Tagged フォーマットは、16 ビット ROM 幅のみをサポートします。

ROM 幅（TI-Tagged フォーマットの場合を除き）は、次の方法で変更できます。

- ❏ **-romwidth** オプションの使用。この方法では、COFF ファイル全体について ROM 幅の値が変更されます。

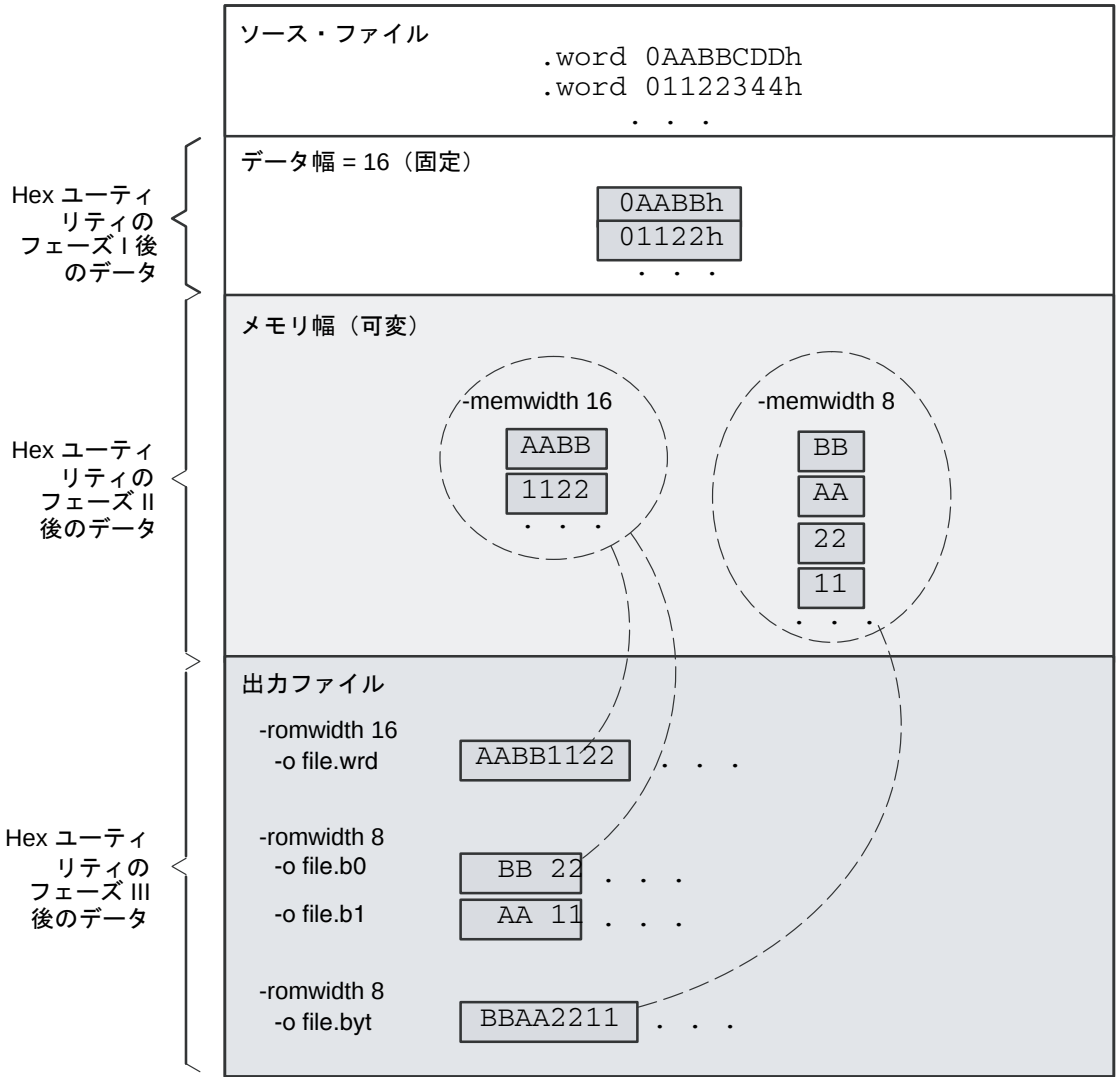
- ❏ ROMS 疑似命令の **romwidth** パラメータの設定。この方法では、特定の ROM アドレス範囲について ROM 幅の値が変更されます。そのアドレス範囲に対する -romwidth オプションは無効になります。14.5 節「ROMS 疑似命令」（14-15 ページ）を参照してください。

どちらの場合も、使用する値は 8 以上の 2 の累乗値でなければなりません。

出力フォーマットの本来のサイズ（TI-Tagged フォーマットでは 16 ビット、それ以外のフォーマットでは 8 ビット）より広い ROM 幅を選択する場合、ユーティリティは単に複数バイトのフィールドをファイルに書き込みます。

図 14-4 に、ターゲット幅、メモリ幅、および ROM 幅が互いにどのように関連し合っているかを示します。

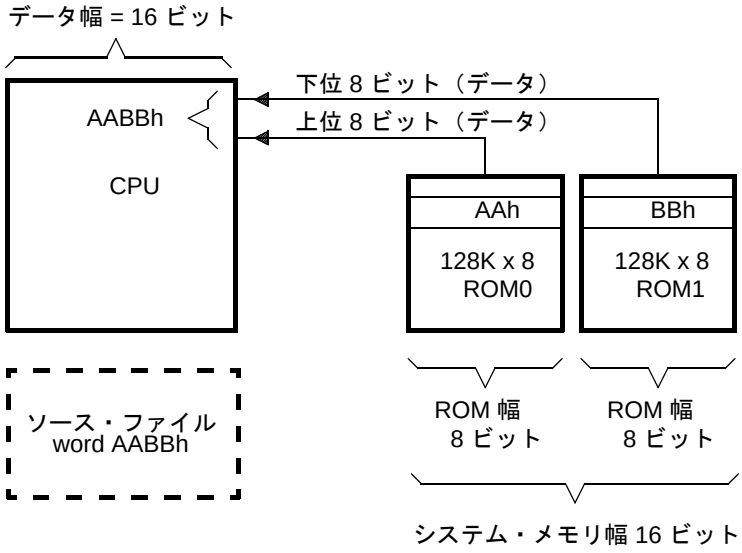
図 14-4.　　データ幅、メモリ幅、および ROM 幅



14.4.5 メモリ構成の例

図 14-5 は、一般的なメモリ構成の例を示しています。このメモリ・システムは、128K × 8 ビットの 2 つの ROM デバイスから構成されます。

図 14-5. C55x のメモリ構成の例



14.4.6 出力ワードのワード配列の指定方法

メモリ・ワードが ROM ワードよりも幅が広い場合（メモリ幅 > ROM 幅）、メモリ・ワードは連続した複数の ROM ワードに分割されます。幅の広いワードを同じ Hex 変換ユーティリティ出力ファイル内の連続した複数のメモリ・ロケーションに分割するには、次の 2 つの方法があります。

- **-order MS** により**ビッグエンディアン**配列を指定する方法。この場合、連続したロケーションの先頭に幅の広いワードの最上位部分が置かれます。

- **-order LS** により**リトルエンディアン**配列を指定する方法。この場合、連続したロケーションの先頭に幅の広いワードの最下位部分が置かれます。

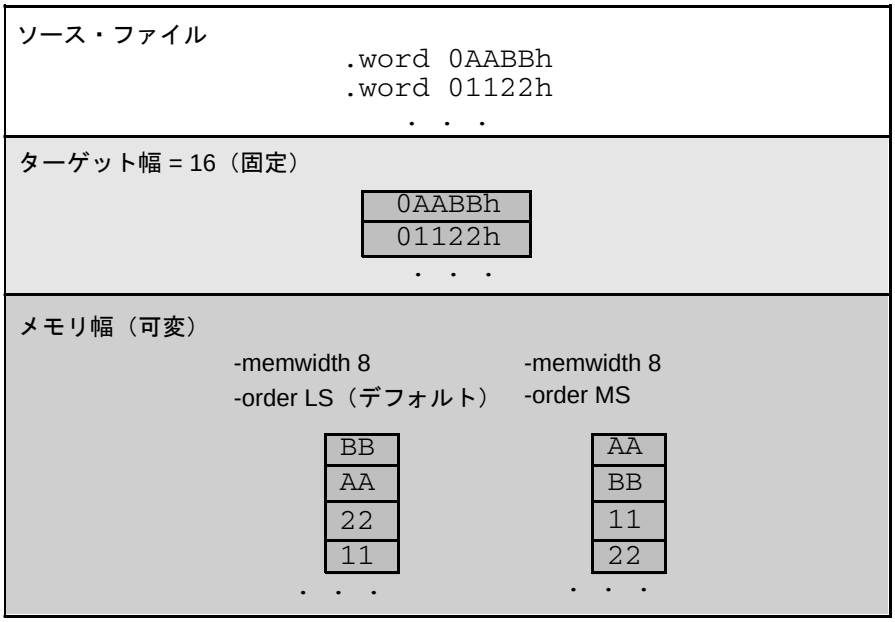
C55x のブート・ローダはリトルエンディアン方式を前提としているので、デフォルトでは、Hex 変換ユーティリティはリトルエンディアン方式を使用します。-order MS は、独自のブート・ローダ・プログラムを使用している場合以外は使用しないでください。

注 : -order オプションが適用される場合

- ❑ 16 より大きい値のメモリ幅を使用している場合にのみ -order オプションを使用します。それ以外の場合、-order は無視されます。
- ❑ -order オプションは、メモリ・ワードが出力ファイルにどのように分割されるかには影響を与えません。これらのファイルは 1 つの集合であると考えてください。この集合には最下位のファイルと最上位のファイルが含まれていますが、集合全体について配列が決まっているわけではありません。この集合に含まれるファイルの名前を指定するときには、-order オプションとは関係なく必ず最下位のファイルの名前を最初に指定します。

図 14-6 では、-order が変換処理にどのように影響するかを説明します。この図と前出の図 14-4 は、Hex 変換ユーティリティ出力ファイル内のデータの状態を示しています。

図 14-6. ワード配列の変化



14.5 ROMS 疑似命令

ROMS 疑似命令は、システムの物理的なメモリ構成をアドレス範囲パラメータのリストとして指定します。

それぞれのアドレス範囲ごとに、そのアドレス範囲に対応する Hex 変換ユーティリティ出力データを含むファイルの集合が 1 つ作成されます。各ファイルを使用して単一の ROM デバイスのプログラムを行います。

ROMS 疑似命令を使用しない場合、このユーティリティは 2 つのアドレス空間（PAGE 0 と PAGE 1）を組み込んだデフォルトのメモリ構成を定義します。それぞれのアドレス空間は 1 つのアドレス領域を含みます。PAGE 0 はプログラム・アドレス空間全体のデフォルト領域を含み、PAGE 1 はデータ・アドレス空間全体のデフォルト領域を含みます。

ROMS 疑似命令は、TMS320C55x リンカの MEMORY 疑似命令に類似しています。どちらの疑似命令も、ターゲット・アドレス空間のメモリ・マップを定義するものです。ROMS 疑似命令では、各行エントリで特定のアドレス範囲を定義します。一般的な構文は次のとおりです。

```
ROMS
{
   [PAGE n:]
      romname: [origin=value,] [length=value,] [romwidth=value,]
               [memwidth=value,] [fill=value,]
               [files={filename1, filename2, ...}]

      romname: [origin=value,] [length=value,] [romwidth=value,]
               [memwidth=value,] [fill=value,]
               [files={filename1, filename2, ...}]
   ...
}
```

ROMS 疑似命令定義を開始します。

PAGE プログラムアドレス空間とデータアドレス空間を使用するターゲット用のメモリ空間を識別します。プログラムが正常にリンクされている場合、PAGE 0 でプログラム・メモリを指定し、PAGE 1 でデータ・メモリを指定します。PAGE コマンドの後に定義した各メモリ範囲は、別の PAGE コマンドを指定するまでそのページに属します。PAGE を指定しない場合、すべての範囲は PAGE 0 に属します。

romname メモリ範囲を識別します。メモリ範囲の名前は 1 から 8 文字までの長さで指定します。この名前は、プログラムに対しては特に重要な意味をもちません。単に範囲を識別するだけです（重複したメモリ範囲名も許されます）。

origin

メモリ範囲の開始アドレスを指定します。これは、origin、org、あるいは o と入力できます。指定する値は、10 進、8 進、16 進の定数でなければなりません。origin の値を省略すると、デフォルト値として 0 が使用されます。
次の表は、10 進、8 進、または 16 進の定数を指定するときに使用できる表記法についてまとめたものです。

定数	表記法	例
16 進数	接頭部 0x または接尾部 h を付ける	0x77 または 077h
8 進数	接頭部 0 を付ける	077
10 進数	接頭部も接尾部も付けない	77

length

メモリ範囲の長さを ROM デバイスの物理的な長さとして指定します。これは、length、len、あるいは l と入力できます。値は、10 進、8 進、または 16 進の定数で指定しなければまりません。length の値を省略すると、値はデフォルトでアドレス空間全体の長さになります。

romwidth

該当するアドレス範囲の物理的な ROM 幅をビット単位で指定します（14.4.4 項「ROM 幅」（14-10 ページ）を参照)。ここで値を指定すると、-romwidth オプションはその値により変更されます。8 以上の 2 の累乗値を 10 進、8 進、または 16 進の定数として指定しなければなりません。

memwidth

該当するアドレス範囲のメモリ幅をビット単位で指定します（14.4.3 項「メモリ幅」（14-9 ページ）を参照)。ここで値を指定すると、-memwidth オプションはその値により変更されます。8 以上の 2 の累乗値を 10 進、8 進、または 16 進の定数として指定しなければなりません。memwidth パラメータを使用するときは、同時に SECTIONS 疑似命令で paddr パラメータをセクションごとに指定しなければなりません。

fill

アドレス範囲で使用する埋め込み値を指定します。イメージ・モードでは、Hex 変換ユーティリティは、あるアドレス範囲のセクション間にあるホール（穴）部分をこの値で埋め込みます。ターゲット幅と同じ幅をもつ値は 10 進、8 進、または 16 進の定数として指定しなければなりません。ここで値を指定すると、-fill オプションはその値により変更されます。fill を使用するときは -image コマンド行オプションも使用しなければなりません。14.9.2 項「埋め込み値を指定する方法」（14-27 ページ）を参照してください。

files アドレス範囲に対応する出力ファイルの名前を指定します。名前のリストは中括弧で囲み、最下位から最上位へと順に出力ファイルを指定してください。
ファイル名の数は、該当するアドレス範囲で生成される出力ファイルの数と同じでなければなりません。出力ファイルの数を計算するには、14.4.4 項「ROM 幅」（14-10 ページ）を参照してください。指定したファイル名が多すぎたり少なすぎたりする場合は、Hex 変換ユーティリティから警告が出されます。

-image オプションを使用している場合を除いて、アドレス範囲を定義するパラメータはすべて任意です。コンマと等号も任意です。origin または length なしでアドレス範囲を指定すると、アドレス空間全体が定義されます。イメージ・モードでは、すべてのアドレス範囲について origin と length が必要です。

同じページのアドレス範囲が重複しないようにしてください。また、アドレス範囲は小さい方から順に指定する必要があります。

14.5.1 ROMS 疑似命令を指定する場合

ROMS 疑似命令を使用しない場合、このユーティリティは 2 つのアドレス空間（PAGE 0 と PAGE 1）を組み込んだデフォルトのメモリ構成を定義します。それぞれのアドレス空間は 1 つのアドレス領域を含みます。PAGE 0 はプログラム・アドレス空間全体のデフォルト領域を含み、PAGE 1 はデータ・アドレス空間全体のデフォルト領域を含みます。特定のページに何もロードされない場合は、そのページに対する出力は作成されません。

ROMS 疑似命令は、次のような場合に使用します。

- ❏ **大量のデータを固定サイズの ROM に書き込む場合。**ROM の長さに対応したメモリ範囲を指定すると、出力は ROM に収まるいくつかのブロックに自動的に分割されます。

- ❏ **出力を特定のセグメントだけに制限する場合。**ROMS 疑似命令を使用してターゲット・アドレス空間の特定のセグメント（1 つ以上）だけに変換を制限することもできます。ROMS 疑似命令で定義した範囲にないデータは変換されません。セクションが範囲の境界にまたがることもありますが、そのような場合は境界で複数の範囲に分割されます。あるセクションがユーザー定義したどの範囲にもまったく入らない場合、そのセクションは変換されず、メッセージも警告も発行されません。この方法では、SECTIONS 疑似命令でセクションの名前を一覧で表示しなくてもセクションを除外できます。ただし、あるセクションの一部だけが範囲内にありその他の部分が未構成メモリ内にある場合は、警告が発行され、範囲内にある部分だけが変換されます。

- **image モードを使用する場合。**-image オプションを使用するときは、必ず ROMS 疑似命令を使用してください。各範囲はすべて埋め込まれ、ある範囲に対応するそれぞれの出力ファイルにその範囲全体のデータが入ります。セクションの前後またはセクション間にある空き部分には、ROMS 疑似命令で指定した埋め込み値、-fill オプションで指定した値、またはデフォルト値のゼロが埋め込まれます。

14.5.2 ROMS 疑似命令の例

例 14–1 の ROMS 疑似命令は、16 ビット・メモリの 16K のワードを 4 個の 8K × 8 ビット EPROM に分割する方法を示しています。

例 14–1. ROMS 疑似命令の例

```
infile.out
-image
-memwidth 16

ROMS
{
   EPROM1: org = 04000h, len = 02000h, romwidth = 8
           files = { rom4000.b0, rom4000.b1 }

   EPROM2: org = 06000h, len = 02000h, romwidth = 8,
           fill = 0FFh,
           files = { rom6000.b0, rom6000.b1 }
}
```

この例では、EPROM1 で 4000h から 5FFFh までのアドレス範囲を定義しています。この範囲には、次のセクションが含まれます。

セクション	対応する範囲
.text	4000h ～ 487Fh
.data	5B80H ～ 5FFFh

このアドレス範囲の残り部分には、0h（デフォルトの埋め込み値）が埋め込まれます。この範囲にあるデータは、次の 2 つの出力ファイルに変換されます。

- rom4000.b0（ビット 0 ～ 7 が入る）
- rom4000.b1（ビット 8 ～ 15 が入る）

EPROM2 は、6000h から 7FFFh までのアドレス範囲を定義しています。この範囲には、次のセクションが含まれます。

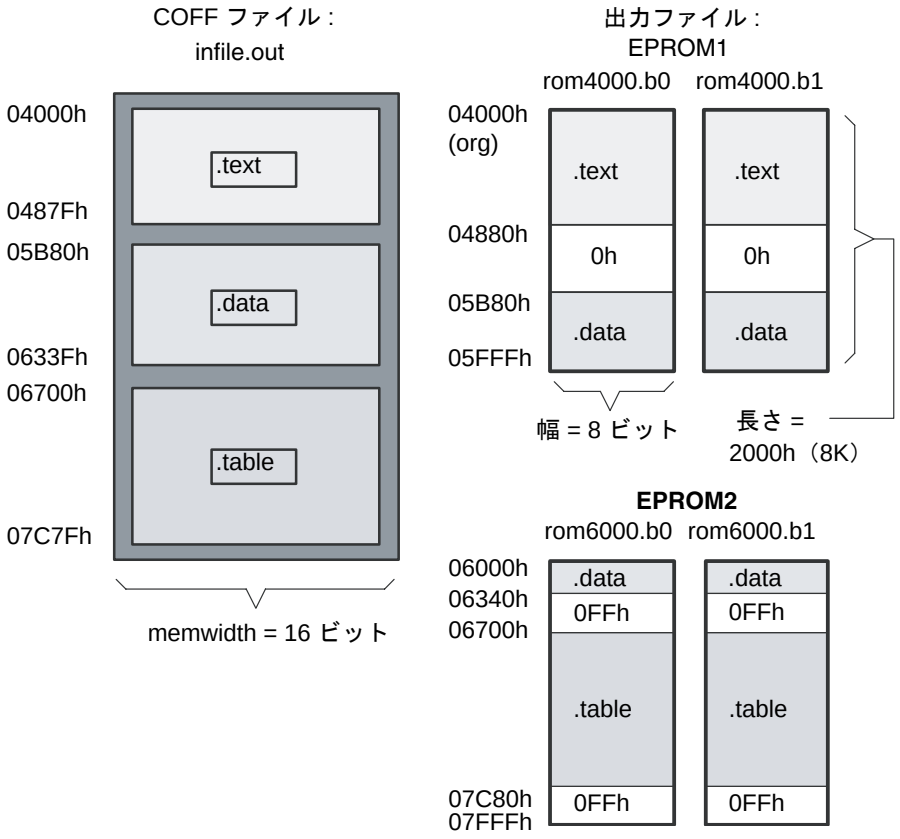
セクション	対応する範囲
.data	6000h ～ 633Fh
.table	6700h ～ 7C7Fh

このアドレス範囲の残りの部分には、0FFh（指定した埋め込み値）が埋め込まれます。この範囲にあるデータは、次の 2 つの出力ファイルに変換されます。

- ❏ rom6000.b0（ビット 0 ～ 7 が入る）
- ❏ rom6000.b1（ビット 8 ～ 15 が入る）

図 14-7 は、この ROMS 疑似命令により infile.out ファイルが 4 個の出力ファイルにどのように分割されるかを示しています。

図 14-7. 例 14–1 の infile.out ファイルを 4 個の出力ファイルに分割する



14.5.3 ROMS 疑似命令のマップ・ファイルを作成する方法

マップ・ファイル（-map オプションで指定）は、ROMS 疑似命令で複数の範囲を指定する場合に非常に有効です。マップ・ファイルには、各アドレス範囲、各範囲のパラメータ、関連付けられた出力ファイルの名前、およびアドレスごとに分割した内容のリスト（セクション名と埋め込み値）が示されます。例 14–1 で作成されるマップ・ファイルの一部を次に示します。

例 14–2. 例 14–1 のマップ・ファイル出力によるメモリ範囲

```
-------------------------------------------------------
00004000..00005fff  Page=0  Width=8  "EPROM1"
-------------------------------------------------------
   OUTPUT FILES:  rom4000.b0    [b0..b7]
                  rom4000.b1    [b8..b15]

   CONTENTS: 00004000..0000487f .text
             00004880..00005b7f FILL = 00000000
             00005b80..00005fff .data
-------------------------------------------------------
00006000..00007fff  Page=0  Width=8  "EPROM2"
-------------------------------------------------------
   OUTPUT FILES:  rom6000.b0    [b0..b7]
                  rom6000.b1    [b8..b15]

   CONTENTS: 00006000..0000633f .data
             00006340..000066ff FILL = 000000ff
             00006700..00007c7f .table
             00007c80..00007fff FILL = 000000ff
```

14.6 SECTIONS 疑似命令

SECTIONS 疑似命令で名前を指定することにより、COFF ファイルの特定のセクションを変換できます。また、オンチップ・ブート・ローダからロードするように構成するセクションや、リンカ・コマンド・ファイルで指定したロードアドレスとは異なるアドレスで ROM に配置するセクションを指定することもできます。

- ❑ SECTIONS 疑似命令を使用する場合は、この疑似命令でリストを作成するセクションのみがユーティリティによって変換されます。COFF ファイル内のその他のセクションは、すべて無視されます。

- ❑ SECTIONS 疑似命令を使用しない場合は、構成メモリの範囲内にある初期化されたセクションはすべて変換されます。TMS320C55x コンパイラが生成する初期化されたセクションには、.text、.const、.cinit、および .switch があります。

初期化されないセクションは、SECTIONS 疑似命令に指定してあるかどうかにかかわらず、決して変換されることはありません。

注 : C/C++ コンパイラによって生成されるセクション

TMS320C55x C/C++ コンパイラは、以下のセクションを自動的に生成します。

- ❑ **初期化されたセクション :** .text、.const、.cinit、および .switch。
- ❑ **初期化されないセクション :** .bss、.stack、および .sysmem。

SECTIONS 疑似命令はコマンド・ファイル内で使用します（コマンド・ファイルの使用方法の詳細は、14.3 節「コマンド・ファイル」（14-6 ページ）を参照してください）。SECTIONS 疑似命令の一般的な構文は次のとおりです。

```
SECTIONS
{
   sname:[paddr=value]
   sname:[paddr=boot]
   sname:[= boot ],
   ...
}
```

SECTIONS 疑似命令定義を開始します。

sname COFF 入力ファイル内のセクションを識別します。存在しないセクションを指定するとユーティリティから警告が発行され、その名前は無視されます。

paddr 該当するセクションを配置する物理的な ROM アドレスを指定します。リンカで与えられるセクションのロード・アドレスは、この値により変更されます（14.11 節「ROM デバイス・アドレスの制御方法」（14-34 ページ）を参照してください）。値は、10 進、8 進、または 16 進の定数で指定しなければなりません。また、**ブート**（オンチップ・ブート・ローダとともに使うためのブート・テーブル・セクションを示す）にすることもできます。ファイルに複数のセクションが含まれていて、あるセクションで *paddr* パラメータを使用する場合は、すべてのセクションで *paddr* パラメータを使用しなければなりません。

= boot オンチップ・ブート・ローダでロードするためにセクションを構成します。これは、**paddr=boot** を使うのと同等です。ブート・セクションの物理アドレスは、ターゲット・プロセッサのタイプと各種のブートローダ固有のコマンド行オプションの両方によって決まります。

セクション名を区切るコンマは任意です。リンカの SECTIONS 疑似命令との共通性をもたせるために、セクション名の後で（boot キーワードの等号の代わりに）コロンを使用できます。たとえば次の 2 つの文は同じです。

```
SECTIONS { .text: .data: boot }
SECTIONS { .text, .data = boot }
```

次の例では、COFF ファイル内に .text、.data、.const、.vectors、.coeff、.tables という 6 個の初期化されたセクションが含まれています。.text と .data のみを変換する場合は、SECTIONS 疑似命令を使用して次のように指定します。

```
SECTIONS { .text, .data }
```

この 2 つのセクションを両方ともブート・ローディング用に構成するには、boot キーワードを追加して次のように指定します。

```
SECTIONS { .text = boot, .data = boot }
```

注 : -boot オプションおよび　SECTIONS 疑似命令の使用

オンチップ・ブート・ローダに対して SECTIONS 疑似命令を使用すると、-boot オプションは無視されます。この場合は、SECTIONS 疑似命令の中でブート・セクションを明示的に指定する必要があります。-boot オプションと、オンチップ・ブート・ローダに関連するその他のコマンド行オプションの詳細は、表 14–2（14-29 ページ）を参照してください。

14.7 指定セクションの除外

-exclude section_name オプションを使用して Hex ユーティリティに指定のセクションを無視するよう通知できます。SECTIONS 疑似命令を使う場合、-exclude オプションは無効になります。

例えば、セクション名 mysect を含む SECTIONS 疑似命令が使われ、同時に -exclude mysect が指定されている場合、その SECTION 疑似命令が優先し、mysect は除外されません。

-exclude オプションには、限定的なワイルドカード機能があります。* を名前指定子の最初または最後に配置し、それぞれ接頭部または接尾部を指定できます。たとえば、-exclude sect* は、文字 sect で開始するセクションすべてを無視します。

* ワイルドカードの付いたコマンド行で -exclude オプションを指定する場合、セクション名とワイルドカードを引用符で囲みます。たとえば、-exclude"sect*" のようになります。引用符を使うと、* が Hex 変換ユーティリティによって解釈されるのを防ぎます。-exclude がコマンド・ファイルにある場合、引用符は指定してはなりません。

14.8 出力ファイル名

Hex 変換ユーティリティは、使用している COFF オブジェクト・ファイルをあるデータ・フォーマットに変換するときに、データを 1 つ以上の出力ファイルに分割します。データをバイト幅またはワード幅のファイルに分割することにより複数のファイルが作成されるときは、必ず最下位のファイルから最上位のファイルへの順序でファイル名が割り当てられます。これは、ターゲットまたは COFF のエンディアン配列や -order オプションとは無関係に行われます。

14.8.1 出力ファイル名を割り当てる方法

Hex 変換ユーティリティは、次の手順に従って出力ファイル名を割り当てます。

1) **ROMS 疑似命令を検索します。**ファイルが ROMS 疑似命令において範囲に関連付けられていて、その範囲についてファイルのリスト（files = {. . .}）が組み込まれている場合は、そのリストにあるファイル名が使用されます。

   たとえば、ターゲット・データは 16 ビットのワードで、8 ビット幅の 2 つのファイルに分割されるとします。ROMS 疑似命令を使用してこの出力ファイルに名前を付けるには、次のように指定します。

   ```
   ROMS
   {
    RANGE1:romwidth=8, files={ xyz.b0 xyz.b1 }
   }
   ```

   ユーティリティは、最下位ビット（LSBs）を xyz.b0 に書き込み、最上位ビット（MSBs）を xyz.b1 に書き込むことにより出力ファイルを作成します。

2) **-o オプションを検索します。**これらの出力ファイルの名前は、-o オプションを使用して指定できます。ROMS 疑似命令にファイル名のリストを指定せずに -o オプションを使用した場合、Hex 変換ユーティリティは -o オプションのリストからファイル名を取り出します。次のように指定すると、前述の ROMS 疑似命令の使用例と同じ結果が得られます。

   ```
   -o xyz.b0 -o xyz.b1
   ```

   ROMS 疑似命令と -o オプションを同時に指定した場合は、-o オプションよりも ROMS 疑似命令の方が優先されることに注意してください。

3) **デフォルトのファイル名を割り当てます。**ファイル名を指定しない場合や、指定したファイル名の数が出力ファイルの数より少ない場合は、ユーティリティはデフォルトのファイル名を割り当てます。デフォルトのファイル名は、COFF 入力ファイルからの基本名と、2 ～ 3 文字の拡張子（ファイル名 .abc など）で構成されます。拡張子には、次の 3 つの部分があります。

a) 出力フォーマットに基づいたフォーマット文字

a ASCII-Hex
i Intel
t TI-Tagged
m Motorola-S
x Tektronix

b) ROMS 疑似命令での範囲番号。範囲には 0 から順に番号が振られます。ROMS 疑似命令がない場合または範囲が 1 つだけの場合、この文字は省略されます。

c) その範囲に関するファイルの集合でのファイル番号。この番号は、0（最下位のファイルを示す）から始まります。

たとえば 16 ビットのターゲット・プロセッサ用の coff.out があり、Intel フォーマットの出力を作成するとします。出力ファイル名を指定しなければ、Hex 変換ユーティリティにより coff.i0 と coff.i1 という名前の 2 つの出力ファイルが作成されます。

Hex 変換ユーティリティの起動時に次の ROMS 疑似命令を指定した場合は、2 個の出力ファイルが作成されます。

```
ROMS
{
   range1:o = 1000h l = 1000h
   range2:o = 2000h l = 1000h
}
```

出力ファイル	格納されるデータ
coff.i00	1000h ～ 1FFFh
coff.i10	2000h ～ 2FFFh

14.9 イメージ・モードおよび -fill オプション

この節では、イメージ・モードでの操作の利点を示すとともに、ターゲット・メモリ範囲の正確な連続イメージをもつ出力ファイルを作成する方法について説明します。

14.9.1 -image オプション

-image オプションを指定すると、ROMS 疑似命令で指定したマップされた範囲のすべてを完全に埋め込むことによりメモリ・イメージが作成されます。

COFF ファイルは、メモリ・ロケーションが割り当てられている複数のメモリのブロック（セクション）から構成されます。通常は、すべてのセクションが隣接しているということはありません。つまり、アドレス空間内のセクション間にデータのない空き部分が生じます。イメージ・モードを使用せずにこのようなファイルを変換すると、Hex 変換ユーティリティは、出力ファイル内のアドレス・レコードを使用して次のセクションの先頭までスキップすることでこれらの空き部分を処理します。つまり、出力ファイルのアドレスに不連続が生じることが考えられます。一部の EPROM プログラマはアドレスの不連続をサポートしていません。

イメージ・モードでは、このような不連続が生じることはありません。各出力ファイルには、ターゲット・メモリ内のアドレス範囲に厳密に対応する連続したデータ・ストリームが格納されます。セクションの前後またはセクション間にある空き部分には、指定した埋め込み値が埋め込まれます。

大部分の 16 進フォーマットは各行にアドレスを必要とするので、イメージ・モードを使用して変換した出力ファイルもアドレス・レコードをもちます。ただし、イメージ・モードではこれらのアドレスは必ず連続したものになります。

注 : ターゲット・メモリの範囲を定義する方法

イメージ・モードを使用する場合は、ROMS 疑似命令も指定しなければなりません。イメージ・モードでは、各出力ファイルが、ある範囲のターゲット・メモリに直接対応します。ユーザーはこの範囲を定義する必要があります。ターゲット・メモリの範囲を定義しない場合、ユーティリティはターゲット・プロセッサのアドレス空間全体のメモリ・イメージを作成しようとします。場合によっては、非常に大量の出力データが作成されます。このような状態を避けるために、ROMS 疑似命令を使用してユーザー側でアドレス空間を明示的に制限しなければなりません。

14.9.2 埋め込み値を指定する方法

-fill オプションは、セクション間のホールに埋め込む値を指定します。この埋め込み値は、-fill オプションの後に整数定数として指定する必要があります。定数の幅は、ターゲット・プロセッサ上のワード幅であるものと見なされます。C55x で -fill 0FFh と指定すると、埋め込みパターンは 00FFh となります。この定数値では、符号拡張は行われません。

fill オプションを使用して値を指定しないと、Hex 変換ユーティリティはデフォルトの埋め込み値である 0 を使用します。*-image* を使う場合のみ、*-fill* オプションが有効です。それ以外では -fill オプションは無視されます。

14.9.3 イメージ・モードを使用する手順

ステップ 1: ROMS 疑似命令を使用して、ターゲット・メモリの範囲を定義します。14.5 節「ROMS 疑似命令」（14-15 ページ）を参照してください。

ステップ 2: -image オプションを指定して、Hex 変換ユーティリティを起動します。バイトを順にカウントするために -byte オプションを使用します。各出力ファイルについてアドレス起点を 0 にリセットするために -zero オプションを使用します。-byte オプションの詳細については 14.11.3 項「バイト・カウントの指定」（14-36 ページ）、-zero オプションの詳細については 14-35 ページを参照してください。ROMS 疑似命令で埋め込み値を指定せず、デフォルト（0）以外の値を使用する場合は、-fill オプションを使用してください。

14.10 オンチップ・ブート・ローダ用のテーブルの作成方法

C55x などの一部の DSP デバイスには、組み込みのブート・ローダが搭載されています。このブート・ローダは、1 つ以上のコードまたはデータ・ブロックによってメモリを初期化します。ブート・ローダは、メモリ（EPROM など）に格納されるか、またはデバイス・ペリフェラル（シリアル・ポートまたは通信ポートなど）からロードされる特別なテーブル（ブート・テーブル）を使用して、コードまたはデータを初期化します。Hex 変換ユーティリティは、ブート・テーブルを自動的に作成することによりブート・ローダをサポートします。

14.10.1 ブート・テーブルについて

ブート・ローダへの入力は「ブート・テーブル」と呼ばれます。ブート・テーブルには、テーブル内に含まれているデータ・ブロックを指定された転送先アドレスにコピーするためのオンチップ・ローダに指示するレコードが含まれています。また、各種のプロセッサ制御レジスタを初期化するための値がブート・テーブルに含まれることもあります。ブート・テーブルは、メモリ内に格納することもデバイス・ペリフェラルを通じて読み取ることもできます。

Hex 変換ユーティリティは、ブート・ローダ用のブート・テーブルを自動的に作成します。このユーティリティを使用すると、ブート・ローダで初期化する COFF セクション、テーブルのロケーション、および任意の制御レジスタの値を指定できます。Hex 変換ユーティリティでは、COFF ファイルを基にターゲット・デバイスの型を識別してデバイスに必要なフォーマットに従って完全なイメージ・テーブルを作成し、そのテーブルを出力ファイル内で 16 進フォーマットに変換します。その後、テーブルを ROM に焼き込んだり、その他の手段でロードしたりできます。

ブート・ローダは、通常のメモリ幅より幅が狭いメモリからのロードをサポートします。例えば、-serial8-memwidth オプションを使ってブート・テーブルの幅を設定することにより、16 ビット TMS320C55x を 1 つの 8 ビット EPROM からブートできます。Hex 変換ユーティリティは、テーブルのフォーマットおよび長さを自動的に調整します。ブート・テーブルの例については、TMS320C55x DSP CPU リファレンス・ガイドのブート・ローダの例を参照してください。

14.10.2 ブート・テーブル・フォーマット

ブート・テーブルのフォーマットは単純です。一般に、各種の制御レジスタ用の値が入っているヘッダ・レコードがあります。また、後続の各ブロックにはそのブロックのサイズおよび転送先アドレスが含まれたヘッダがあり、ヘッダの後にそのブロックのデータがあります。ブロックは複数入力できます。最後のブロックの後には、終了ブロックが続きます。さらに、テーブルはその他の制御レジスタ値が格納されるフッタをもつことができます。詳細は、TMS320C55x DSP CPU リファレンス・ガイドのブート・ローダの節を参照してください。

14.10.3 ブート・テーブルの作成方法

表 14–2 は、ブート・ローダで使用可能な Hex 変換ユーティリティ・オプションについてまとめたものです。

表 14–2. ブートローダ・オプション

(a) すべての C55x デバイス用のオプション

オプション	説明
-boot	全セクションをブート可能な形式に変換します（SECTIONS 疑似命令の代わりに使用します）。
-bootorg *value*	ブート・ローダ・テーブルのソース・アドレスを指定します。
-bootpage *value*	ブート・ローダ・テーブルのターゲット・ページ番号を指定します。
-e *value*	ブート・ロード後に実行を始めるエントリ・ポイントを指定します。*value* はアドレスまたはグローバル・シンボルになります。
-parallel16	16 ビットのパラレル・インタフェース・ブート・テーブルの指定（-memwidth 16 および -romwidth 16）
-parallel32	32 ビットのパラレル・インタフェース・ブート・テーブルの指定（-memwidth 16 および -romwidth 32）
-serial8	8 ビットのシリアル・インタフェース・ブート・テーブルの指定（-memwidth 8 および -romwidth 8）
-serial16	16 ビットのシリアル・インタフェース・ブート・テーブルの指定（-memwidth 16 および -romwidth 16）
-v*device*:*revision*	デバイスおよびシリコン改訂番号の指定

表 14–2. ブートローダ・オプション（続き）

（b）C55x LP デバイス専用のオプション

オプション	説明
-arr *value*	ABU 受信アドレス・レジスタ値を設定します。
-bkr *value*	ABU 送信バッファ・サイズ・レジスタの値を設定します。
-bootorg COMM	ブート・ローダ・テーブルのソースに通信ポートを指定します。
-bootorg WARM または -warm	ブート・ローダ・テーブルのソースに、現在メモリ内にあるテーブルを指定します。
-bscr *value*	PARALLEL/WARM ブート・モードのバンク・スイッチ制御レジスタの値を設定します。
-spc *value*	シリアル・ポート制御レジスタの値を設定します。
-spce *value*	シリアル・ポート制御拡張レジスタの値を設定します。
-swwsr *value*	PARALLEL/WARM ブート・モードのソフトウェア・ウェイト・ステート・レジスタの値を設定します。
-tcsr *value*	TDM シリアル・ポート・チャネル選択レジスタの値を設定します。
-trta *value*	TDM シリアル・ポート送受信アドレス・レジスタの値を設定します。

14.10.3.1 ブート・テーブルの作成方法

ブート・テーブルを作成するには、次の手順に従います。

ステップ 1: ファイルをリンクします。ブート・テーブル・データの各ブロックは、COFF ファイル内の 1 つの初期化されたセクションに対応します。初期化されないセクションは、Hex 変換ユーティリティでは変換されません（14.6 節「SECTIONS 疑似命令」（14-21 ページ）を参照してください）。

ブートローダ・テーブル内に入れるセクションを選択すると、Hex 変換ユーティリティは、そのセクションのロード・アドレスをブート・テーブル内の該当するブロックの転送先アドレス・フィールドに配置します。セクションの内容は、該当するブロックの生データとして扱われます。

Hex 変換ユーティリティは、セクション実行アドレスを使用しません。リンク時には、ROM アドレスやブート・テーブルの構成について考慮する必要はありません。この構成は Hex 変換ユーティリティが取り扱います。

ステップ 2: **ブート可能なセクションを識別します。**-boot オプションを使用して、すべてのセクションをブート・ロード用に設定するように Hex 変換ユーティリティに指示できます。または、SECTIONS 疑似命令を使って設定する指定のセクションを選択できます（14.6 節「SECTIONS 疑似命令」（14-21 ページ）を参照してください）。SECTIONS 疑似命令を使用した場合は -boot オプションは無視されることに注意してください。

ステップ 3: **ブート・テーブルの ROM アドレスを設定します。**-bootorg オプションを使用して、完成したテーブルのソース・アドレスを設定します。たとえば、C55x を使用していて、メモリ位置 8000h からブートする場合は、-bootorg 8000h と指定します。このように指定すると、Hex 変換ユーティリティ出力ファイルのアドレス・フィールドは 8000h から開始します。

-bootorg オプションを全く使用しなかった場合は、ROMS 疑似命令で指定した最初のメモリ範囲の起点にテーブルが配置されます。ROMS 疑似命令を使用しないと、最初のセクションのロード・アドレスからテーブルが開始されます。テーブルを PAGE 0 以外から開始できるようにするために、-bootpage オプションも用意されています。

ステップ 4: **ブートローダ固有のオプションを設定します。**エントリ・ポイント、パラレル・インタフェース、またはシリアル・インタフェース・オプションを必要に応じて設定します。改訂 1.0 のシリコンを使う場合、デバイスおよびシリコン改訂番号の指定には、改訂 1.0 ブートローダと異なるため -v5510:1 オプションを使わなければなりません。

ステップ 5: **システム・メモリ構成を示します。**詳細は、14.4 節「メモリ幅について」（14-8 ページ）、および 14.5 節「ROMS 疑似命令」（14-15 ページ）を参照してください。

14.10.3.2 ブート・テーブル用の空きを確保する方法

完成したブート・テーブルは、ブート・ローダに関するヘッダ・レコードおよびデータがすべて含まれている単一のセクションに類似しています。このセクションのアドレスはブート・テーブルの起点となります。Hex 変換ユーティリティは、通常の変換処理の一部としてブート・テーブルを 16 進フォーマットに変換し、それを他のセクションと同じように出力ファイルにマップします。

システムのメモリには、必ずブート・テーブル用の空きを残しておいてください。特に、ROMS 疑似命令を使用しているときは注意が必要です。ブート・テーブルは、他の非ブート・セクションや未設定のメモリと重なり合うことはできません。通常は、これが問題になることはありません。普通は、システムのメモリの一部がブート・テーブルのために確保されています。このメモリを ROMS 疑似命令で 1 つまたは複数の範囲として設定し、-bootorg オプションを使用して開始アドレスを指定してください。

14.10.4 ペリフェラルからのブート方法

-parallel16、-parallel32、-serial8、または -serial16 オプションを使用すると、シリアル・ポートまたはパラレル・ポートからブートするように指定できます。オプションは、ターゲット・デバイスと使用するチャネルに応じて選択します。たとえば、C55x を 16 ビット McBSP ポートからブートするには、コマンド行またはコマンド・ファイルで -serial16 と指定します。また、C55x を EMIF ポートのいずれか 1 つからブートするには、-parallel16 または -parallel32 と指定します。

注 : オンチップ・ブート・ローダについて

- **メモリ・コンフリクトの可能性。**ペリェラルからブートする場合、ブート・テーブルは実際にはメモリ内にありません。ブート・テーブルはデバイス・ペリフェラルを介して受け取ります。ただし、14-31 ページのステップ 3 で説明したように、メモリ・アドレスが割り当てられます。

  テーブルが非ブート・セクションとコンフリクトを発生する場合、ブート・テーブルを異なるページに置いてください。ROMS 疑似命令を使用して未使用のページの範囲を定義し、-bootpage オプションを使用してブート・テーブルをこのページに入れます。これにより、ブート・テーブルはダミー・ページの 0 の位置に表示されます。

- **ペリフェラルのブート・ローダ・アドレスに対する EPROM フォーマットが必要な理由。**通常のシステムでは、親プロセッサは子プロセッサのペリフェラルを介して、その子プロセッサをブートします。ブート・ローダ・テーブル自体が親プロセッサのメモリ・マップ内の空間を占めることがあります。EPROM フォーマットと ROMS 疑似命令のアドレスは、子プロセッサが使用するものではなく、親プロセッサが使用するフォーマットとアドレスに対応するものです。

14.10.5 ブート・テーブルのエントリ・ポイントの設定方法

ブート・ロード処理が完了すると、リンカにより指定され COFF ファイルに含まれたデフォルトのエントリ・ポイントで実行が開始されます。Hex 変換ユーティリティで -e オプションを使用することにより、このエントリ・ポイントを別のアドレスに設定できます。

たとえば、ロード後にプログラム実行をアドレス 0123h から開始したい場合は、コマンド行またはコマンド・ファイルで -e 0123h と指定します。-e のアドレスは、リンカが生成するマップ・ファイルを見て決めることができます。

注 : 有効なエントリ・ポイント

値は、定数またはアセンブリ・ソース内で対外的に定義された（たとえば .global を使って）シンボルでなければなりません。

14.10.6 C55x ブート・ローダの使用方法

この項では、C55x デバイスのブート・ローダによる Hex 変換ユーティリティの使用方法について説明します。シリコン改訂 1.0 を使っている場合は、-v5510:1 オプションを使わなければなりません。C55x ブート・ローダには、いくつかの異なるブート・テーブル・フォーマットがあります。

フォーマット	オプション
EMIF 16 ビット	-parallel16
EMIF 32 ビット	-parallel32
McBSP 8 ビット	-serial8
McBSP 16 ビット	-serial16

モード	-bootorg の設定	-memwidth の設定
8 ビット・パラレル I/O	-bootorg PARALLEL	-memwidth 8
16 ビット・パラレル I/O	-bootorg PARALLEL	-memwidth 16
8 ビット・シリアル RS232	-bootorg SERIAL	-memwidth 8
16 ビット・シリアル RS232	-bootorg SERIAL	-memwidth 16
8 ビット・パラレル EPROM	-bootorg 0x8000	-memwidth 8
16 ビット・パラレル EPROM	-bootorg 0x8000	-memwidth 16

C55x は、メモリ内のブート・テーブルからブートもできます。外部メモリ（EPROM）からブートするには、-bootorg オプションを使用してブート・メモリのソース・アドレスを指定します。-memwidth 8 または -memwidth 16 のどちらかを使用してください。

たとえば、図 14-8 のコマンド・ファイルを使用すると、0x8000 の位置にあるバイト幅の EPROM から abc.out の .text セクションをブートできます。

図 14-8. C55x EPROM からブートするためのコマンド・ファイルの例

```
abc.out              /* input file                          */
-o abc.i             /* output file                         */
-i                   /* Intel format                        */
-memwidth 8          /* 8-bit memory                        */
-romwidth 8          /* outfile is bytes, not words         */
-bootorg 0x8000      /* external memory boot                */

SECTIONS { .text: BOOT }
```

14.11 ROM デバイス・アドレスの制御方法

Hex 変換ユーティリティの出力アドレス・フィールドは、ROM デバイス・アドレスに対応します。EPROM プログラマは、Hex 変換ユーティリティ出力ファイルのアドレス・フィールドで指定されている位置にデータを焼き付けます。Hex 変換ユーティリティには、各セクションの ROM での開始アドレスを制御したり、アドレス・フィールドをインクリメントするのに使用するアドレス・インデックスを制御したりする、いくつかのメカニズムが用意されています。ただし、多くの EPROM プログラマには、データを焼き込む ROM の位置を直接制御する機能があります。

14.11.1 開始アドレスの制御方法

Hex 変換ユーティリティ出力ファイルの制御メカニズムは、ブート・ローダを使用しているかどうかにより異なります。

非ブート・ローダ・モード。Hex 変換ユーティリティ出力ファイルのアドレス・フィールドは、以下のそれぞれのメカニズムにより制御されます。ここでは、各メカニズムを優先順位の低いものから高いものへ順に示します。

1) **リンカ・コマンド・ファイル。**Hex 変換ユーティリティ出力ファイルのアドレス・フィールドは、デフォルトでは、ロード・アドレス（リンカ・コマンド・ファイルで指定される）と Hex 変換ユーティリティのパラメータ値の関数になります。この関係を次にまとめます。

$$\text{out\_file\_addr}^{\dagger} = \text{load\_addr} \times (\text{data\_width} \div \text{mem\_width})$$

out_file_addr	出力ファイルのアドレスです。
load_addr	リンカが割り当てたロード・アドレスです。
data_width	TMS320C55x デバイスの場合は 16 ビットとして指定されます。14.4.2 項「データ幅」（14-9 ページ）を参照してください。
mem_width	メモリ・システムのメモリ幅です。メモリ幅は -memwidth オプションで指定するか、または ROMS 疑似命令内部の memwidth パラメータで指定できます。14.4.3 項「メモリ幅」（14-9 ページ）を参照してください。

† paddr が指定されていない場合

データ幅をメモリ幅で割った値は、アドレス生成のための補正係数です。データ幅がメモリ幅よりも大きい場合は、この補正係数によりアドレス空間が拡張されます。たとえば、ロード・アドレスが 0×1 で、データ幅をメモリ幅で割った値が 2 の場合、出力ファイル・アドレス・フィールドは 0×2 となります。データはメモリ幅のサイズに相当する 2 つの連続する位置に分割されます。

2） **SECTIONS 疑似命令の paddr パラメータ。**あるセクションに対して paddr パラメータが指定されている場合、Hex 変換ユーティリティは、セクションのロード・アドレスをバイパスして、paddr で指定されたアドレスにそのセクションを配置します。Hex 変換ユーティリティ出力ファイルのアドレス・フィールドと paddr パラメータとの関係は、次のようにまとめることができます。

out_file_addr† = paddr_val + (load_addr - sect_beg_load_addr) × (data_width ÷ mem_width)

out_file_addr	出力ファイルのアドレスです。
paddr_val	SECTIONS 疑似命令内の paddr パラメータで指定された値です。
sec_beg_load_addr	リンカにより割り当てられたセクションのロード・アドレスです。

† paddr が指定されていない場合

データ幅をメモリ幅で割った値は、アドレス生成のための補正係数です。ロード・アドレスからセクション開始ロードアドレス係数を引いたものは、セクションの開始点からのオフセットです。

3） **-zero オプション。**-zero オプションを使用すると、すべての出力ファイルについてアドレス起点はゼロにリセットされます。各ファイルはゼロから開始され上位方向にカウントされるため、アドレス・レコードは、データの実際のターゲット・アドレスではなくファイルの先頭からのオフセット（ROM 内でのアドレス）を表します。

各出力ファイル内の開始アドレスを強制的にゼロにするには、-zero オプションと -image オプションを組み合わせて使用しなければなりません。-image オプションを指定せずに -zero オプションを指定した場合は、ユーティリティから警告が発行され、-zero オプションは無視されます。

ブート・ローダ・モード。ブート・ローダを使用している場合、Hex 変換ユーティリティはブート・テーブル内の異なる COFF セクションを連続するメモリ位置に入れます。各 COFF セクションは 1 つのブート・テーブル・ブロックになり、それぞれのブロックの転送先アドレスは、リンカが割り当てたセクション・ロード・アドレスと同じになります。

ブート・テーブルでは、Hex 変換ユーティリティ出力ファイルのアドレス・フィールドは、リンカが割り当てたセクションのロード・アドレスとは無関係です。ブート・テーブルのアドレス・フィールドは、テーブルの開始に対するオフセットに、補正係数（メモリ幅で割ったデータ幅）を掛けた値を示しているだけです。リンカにより割り当てられたセクションのロード・アドレスは、そのセクションのサイズ、およびそのセクション内に含まれたデータと一緒にブート・テーブルにエンコードされます。これらのアドレスは、ブート・ロードの処理中にデータをメモリに保存するために使用されます。

次のメカニズムのいずれかを使用しない限り、COFF 入力ファイル内にある最初のブート可能なセクションのリンク後のロード・アドレスは、デフォルトによりブート・テーブルの開始アドレスになります。それぞれのメカニズムは、優先順位の低い方から高い方へ順に示してあります。重複している範囲では、優先順位の低いオプションで設定した値よりも、優先順位の高いメカニズムで設定した値が使用されます。

1） **ROMS 疑似命令で指定した ROM 起点。**Hex 変換ユーティリティは、ROMS 疑似命令内の最初のメモリ範囲の起点にブート・テーブルを配置します。

2） **-bootorg オプション。**メモリからのブートを選択している場合、Hex 変換ユーティリティは、-bootorg オプションで指定されたアドレスにブート・テーブルを配置します。

14.11.2 アドレス・インクリメント・インデックスの制御方法

デフォルトでは、Hex 変換ユーティリティは、出力ファイルのアドレス・フィールドをメモリ幅の値に基づいてインクリメントします。メモリ幅が 16 ビットの場合は、出力ファイルの各行にある 16 ビット・ワードの個数に基づいて、アドレスがインクリメントされます。

14.11.3 バイト・カウントの指定

一部の EPROM プログラマによっては、出力ファイル・アドレス・フィールドに、ワード・カウントでなくバイト・カウントを入れることを必須とする場合があります。-byte オプションを使用すると、出力ファイル・アドレスは各バイトごとに 1 回インクリメントします。たとえば、開始アドレスが 0h で 1 行目に 8 ワード含まれている場合、-byte オプションを使用しないと、2 行目はアドレス 8（8h）から始まります。それに対して、開始アドレスが 0h で 1 行目に 8 ワード含まれている場合、-byte オプションを使用すると、2 行目はアドレス 16（010h）から始まります。両方の例のデータは同じです。-byte は、出力ファイル・アドレス・フィールドの計算に影響を及ぼすだけで、変換後のデータの実際のターゲット・プロセッサ・アドレスには影響しません。

-byte オプションを使用すると、ターゲット・プロセッサでバイト単位のアドレス指定が可能であるかどうかにかかわらず、出力ファイル内のアドレス・レコードはファイル内のバイト位置を参照します。

14.11.4 アドレス・ホールの処理方法

メモリ幅がデータ幅と異なる場合は、自動的にロード・アドレスに補正係数がかけられるので、あるセクションの開始、または 2 つのセクションの間にホールが作成されることがあります。

たとえば、COFF セクション（.sec1）を 8 ビット EPROM のアドレス 0x0100 にロードするとします。リンカ・コマンド・ファイルでこのロード・アドレスを 0x0100 と指定すると、Hex 変換ユーティリティは、そのアドレスに 2（データ幅をメモリ幅で割った値 = 16/8 = 2）を掛けて、出力ファイルの開始アドレスを 0x0200 とします。EPROM プログラマを使用して EPROM の開始アドレスを制御しない限り、EPROM 内にホールが作成されます。EPROM には、0x0100 ではなく、0x0200 からデータが焼き込まれます。これは、次のような方法で解決できます。

- **SECTIONS 疑似命令の paddr パラメータを使用する。**これにより、指定した値から強制的にセクションが開始されます。図 14-9 は、.sec1 の開始部分にホールができないようにした Hex コマンド・ファイルを示しています。

図 14-9. セクションの開始部分にホールができないようにするための Hex コマンド・ファイル

```
-i
a.out
-map a.map

ROMS
{
 ROM :org = 0x0100, length = 0x200, romwidth = 8,
       memwidth = 8
}

SECTIONS
 {
   sec1:paddr = 0x100
}
```

ファイルに複数のセクションがあり、1 つのセクションで paddr パラメータを使用する場合は、すべてのセクションで paddr パラメータを使用しなければなりません。

- **-bootorg オプションまたは ROMS origin パラメータを使用する（ブート・ロード時のみ）。**14-35 ページで説明したように、ブート・ロードを実行する場合は、-bootorg オプションまたは ROMS 疑似命令の origin により、ブートローダ・テーブル全体の EPROM アドレスを制御できます。

14.12 オブジェクト・フォーマットについて

Hex 変換ユーティリティは、COFF オブジェクト・ファイルを大部分の EPROM プログラマが入力として受け入れる、5 種類のオブジェクト・フォーマット（ASCII-Hex、Intel MCS-86、Motorola-S、Extended Tektronix、および TI-Tagged）のいずれかに変換します。

表 14–3 は、フォーマットを指定するオプションを示しています。

- ❏ これらのオプションを 2 つ以上使用する場合は、最後に指定したオプションが有効になります。
- ❏ デフォルトの出力フォーマットは、Tektronix（-x オプション）です。

表 14–3. Hex 変換フォーマットを指定するオプション

オプション	フォーマット	アドレスのビット数	デフォルト幅
-a	ASCII-Hex	16	8
-i	Intel	32	8
-m1	Motorola-S1	16	8
-m2 または -m	Motorola-S2	24	8
-m3	Motorola-S3	32	8
-t	TI-Tagged	16	16
-x	Tektronix	32	8

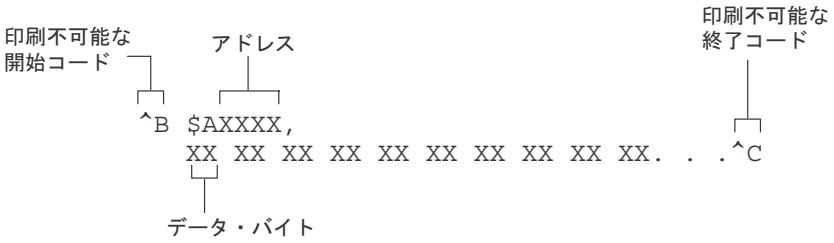
アドレスのビット数は、該当するフォーマットでサポートされるアドレス情報のビット数を示します。16 ビット・アドレスのフォーマットでは、64K までのアドレスのみがサポートされます。Hex 変換ユーティリティは、ターゲット・アドレスを有効なビット数に合わせて切り捨てます。

デフォルト幅は、該当するフォーマットのデフォルト出力幅を示します。このデフォルト幅は、-romwidth オプションを使用するか、ROMS 疑似命令で romwidth パラメータを使用して変更できます。ただし、TI-Tagged フォーマットのデフォルト幅は変更できません。TI-Tagged フォーマットでは、16 ビット幅のみがサポートされます。

14.12.1 ASCII-Hex オブジェクト・フォーマット（-a オプション）

ASCII-Hex オブジェクト・フォーマットは、16 ビット・アドレスをサポートします。このフォーマットは、各バイトが空白で区切られているバイト・ストリームから構成されます。図 14-10 は、ASCII-Hex フォーマットを示しています。

図 14-10. ASCII-Hex オブジェクト・フォーマット



このファイルは、ASCII STX 文字（ctrl-B、02h）で開始し、ASCII ETX 文字（ctrl-C、03h）で終了します。アドレス・レコードは、$AXXXX（XXXX は 4 桁（16 ビット）の 16 進アドレス）で示されます。アドレス・レコードが存在するのは、次の場合に限られます。

- ❏ 不連続が生じた場合
- ❏ バイト・ストリームがアドレス 0 から開始されていない場合

-image オプションと -zero オプションを使用すると、不連続およびアドレス・レコードをすべて回避できます。これらが回避される場合は、バイト値のリストとして出力されます。

14.12.2 Intel MCS-86 オブジェクト・フォーマット（-i オプション）

Intel オブジェクト・フォーマットは、16 ビット・アドレスおよび 32 ビット拡張アドレスをサポートします。Intel フォーマットは、レコードの開始、バイト・カウント、ロード・アドレス、およびレコード・タイプを定義する 9 文字（4 つのフィールド）の接頭部と、データと、2 文字のチェックサム接尾部から構成されます。

9 文字の接頭部は、以下の 3 つのレコード・タイプを表します。

レコード・タイプ	説明
00	データ・レコード
01	ファイルの終わりレコード
04	拡張リニア・アドレス・レコード

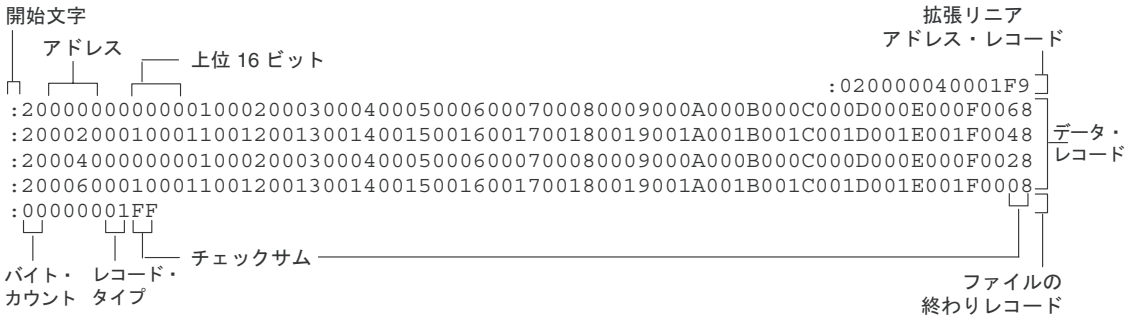
レコード・タイプ 00（データ・レコード）は、コロン（:）で始まり、その後にバイト・カウント、最初のデータ・バイトのアドレス、レコード・タイプ（00）およびチェックサムが順に続きます。このアドレスは、32 ビット・アドレスの最下位 16 ビットです。この値が最新の 04（拡張リニア・アドレス）レコード内の値と連結されて、完全な 32 ビット・アドレスが作成されます。チェックサムは、バイト・カウント、アドレス、データ・バイトなどのレコード内で先行する各バイトの和の 2 の補数（2 進数）です。

レコード・タイプ 01（ファイルの終わりレコード）も、コロン（:）で始まり、その後にバイト・カウント、アドレス、レコード・タイプ (01) およびチェックサムが順に続きます。

レコード・タイプ 04（拡張リニア・アドレス・レコード）は、上位 16 個のアドレス・ビットを指定します。このレコードは、コロン（:）で始まり、その後にバイト・カウント、ダミーアドレス（0h）、レコード・タイプ（04）、アドレスの上位 16 ビット、およびチェックサムが順に続きます。データ・レコード内の後続のアドレス・フィールドには、アドレスの下位ビットが入ります。

図 14-11 は、Intel Hex オブジェクト・フォーマットを示しています。

図 14-11. Intel - Hex オブジェクト・フォーマット



14.12.3 Motorola Exorciser オブジェクト・フォーマット（-m1 オプション、-m2 オプション、および -m3 オプション）

Motorola S1、S2、および S3 のフォーマットは、それぞれ 16 ビット、24 ビット、および 32 ビットのアドレスをサポートします。これらのフォーマットは、ファイルの始め（ヘッダ）レコード、データ・レコード、およびファイルの終わり（終了）レコードから構成されます。各レコードには、レコード・タイプ、バイト・カウント、アドレス、データ、およびチェックサムの 5 つのフィールドがあります。レコード・タイプには、次のものがあります。

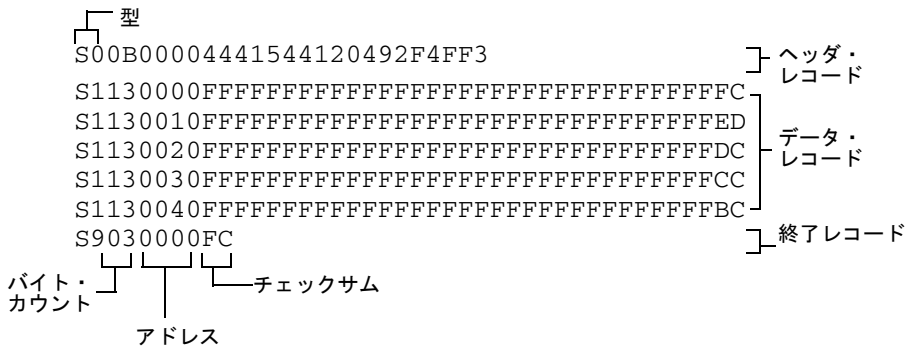
レコード・タイプ	説明
S0	ヘッダ・レコード
S1	16 ビット・アドレス用のコード／データ・レコード（S1 フォーマット）
S2	24 ビット・アドレス用のコード／データ・レコード（S2 フォーマット）
S3	32 ビット・アドレス用のコード／データ・レコード（S3 フォーマット）
S7	32 ビット・アドレス用の終了レコード（S3 フォーマット）
S8	24 ビット・アドレス用の終了レコード（S2 フォーマット）
S9	16 ビット・アドレス用の終了レコード（S1 フォーマット）

バイト・カウントは、レコード・タイプとバイト・カウントを除いた、レコード内の文字のペアのカウントです。

チェックサムは、バイト・カウント、アドレス、およびコード / データの各フィールドを構成している文字のペアで表される値の合計の 1 の補数の最下位バイトです。

図 14-12 は、Motorola-S オブジェクト・フォーマットを示しています。

図 14-12. Motorola-S フォーマット



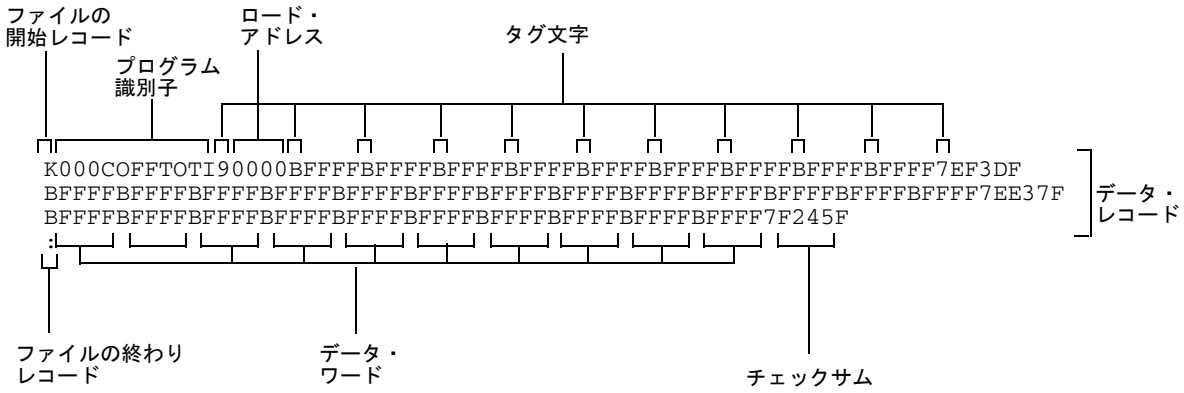
14.12.4 TI-SDSMAC オブジェクト・フォーマット（-t オプション）

TI-Tagged オブジェクト・フォーマットは、16 ビット・アドレスをサポートします。このフォーマットは、ファイル開始レコード、データ・レコード、およびファイルの終わりレコードから構成されます。各データ・レコードは、一連の小フィールドから構成され、タグ文字により意味付けされています。有効なタグ文字は次のとおりです。

タグ文字	説明
K	この後にプログラム識別子が続く
7	この後にチェックサムが続く
8	この後にダミーのチェックサム（無視される）が続く
9	この後に 16 ビットのロード・アドレスが続く
B	この後にデータ・ワード（4 文字）が続く
F	データ・レコードの終了を識別する
*	この後にデータ・バイト（2 文字）が続く

図 14-13 は、TI-Tagged オブジェクト・フォーマットのタグ文字とフィールドを示しています。

図 14-13. TI-Tagged オブジェクト・フォーマット



最初のアドレスより前にデータ・フィールドがある場合は、その最初のフィールドにアドレス 0000h が割り当てられます。どのデータ・バイトについてもアドレス・フィールドを作成できますが、必須のアドレス・フィールドはありません。タグ文字 7 の後にあるチェックサム・フィールドは、最初のタグ文字からチェックサムのタグ文字（7 または 8）までにある文字の、8 ビット ASCII 値の合計の 2 の補数です。ファイルの終わりレコードは、コロン（:）です。

14.12.5 Extended Tektronix オブジェクト・フォーマット（-x オプション）

Tektronix オブジェクト・フォーマットは、32 ビット・アドレスをサポートし、次の 2 つのレコード・タイプがあります。

データ・レコード　ヘッダ・フィールド、ロード・アドレス、およびオブジェクト・コードが格納されます。

終了レコード　モジュールの終了を示します。

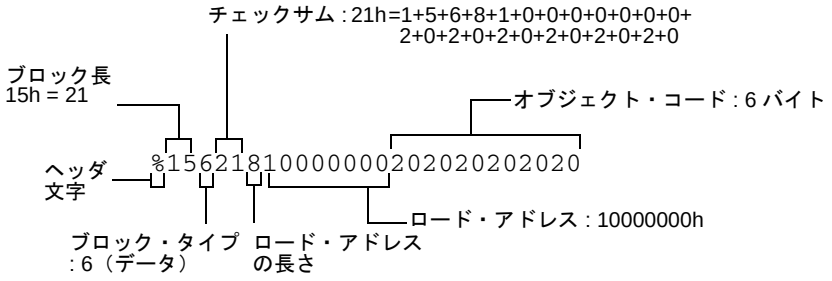
データ・レコード内のヘッダ・フィールドには、次の情報が入ります。

項目	ASCII 文字の数	説明
%	1	Tektronix フォーマットのデータ・タイプ
ブロック長	2	% を除いたレコード内の文字数
ブロック・タイプ	1	6 = データ・レコード 8 = 終了レコード
チェックサム	2	% とチェックサムを除いたレコード内のすべての値の合計の 256 に対するモジュロ（2 桁の 16 進数）

データ・レコード内のロード・アドレスは、オブジェクト・コードが指定されたアドレスを示します。最初の桁はアドレス長を示していて、常に 8 です。データ・レコードの残りの文字には、オブジェクト・コード（1 バイトについて 2 文字）が含まれています。

図 14-14 は、Tektronix オブジェクト・フォーマットを示しています。

図 14-14.　Extended Tektronix オブジェクト・フォーマット



14.13 Hex 変換ユーティリティのエラー・メッセージ

section mapped to reserved memory message

説明 セクションまたはブートローダ・テーブルが、プロセッサのメモリ・マップに示されている予約メモリ領域にマップされています。

動作 セクションまたはブートローダ・テーブルのアドレスを訂正してください。有効なメモリ位置については、TMS320C55x DSP CPU リファレンス・ガイドを参照してください。

sections overlapping

説明 複数の COFF セクションのロード・アドレスが重複しているか、ブート・テーブルのアドレスが他のセクションと重複しています。

動作 この問題の原因には、メモリ幅がデータ幅より狭い場合に、Hex 変換ユーティリティで実行したロード・アドレスから 16 進出力ファイル・アドレスへの変換が正しく行われなかったことが考えられます。詳細は、14.4 節「メモリ幅について」（14-8 ページ）、および 14.11 節「ROM デバイス・アドレスの制御方法」（14-34 ページ）を参照してください。

unconfigured memory error

説明 このエラーは、以下のどちらかが原因で起きたと考えられます。

- ロード・アドレスが ROMS 疑似命令で定義したメモリ範囲内にないセクションが、COFF ファイルに含まれている。
- ブートローダ・テーブルのアドレスが、ROMS 疑似命令で定義したメモリ範囲内にない。

動作 ROMS 疑似命令で定義された ROM 範囲を訂正して、各アドレスがメモリ範囲内に収まるようにするか、またはセクションのロード・アドレスかブートローダ・テーブルのアドレスを修正してください。ROMS 疑似命令を使用していない場合は、デフォルトによりプロセッサのアドレス空間全体がメモリ範囲になることを忘れないでください。この理由により、ROMS 疑似命令を除去する方法も次の解決策として使用できます。

付録 A

共通オブジェクト・ファイル・フォーマット

コンパイラ、アセンブラ、およびリンカは、共通オブジェクト・ファイル・フォーマット（COFF）のオブジェクトを作成します。COFF は、AT&T 社が UNIX ベースのシステム向けに開発したフォーマットと同名のオブジェクト・ファイル・フォーマットを実現したものです。このオブジェクト・ファイル・フォーマットが使用された理由は、モジュラ・プログラミングが促進される点と、コード・セグメントおよびターゲット・システムのメモリを管理する強力かつ柔軟な方式が提供される点にあります。

COFF の基本概念はセクションです。第 2 章「共通オブジェクト・ファイル・フォーマットの概要」で、COFF セクションについて詳しく説明します。セクションの操作を理解しておくと、アセンブリ言語ツールをより効果的に使用できます。

この付録には、COFF オブジェクト・ファイルの構造に関する技術的な詳細情報が収録されています。この情報の大部分は、C/C++ コンパイラで生成されるシンボリック・デバッグ情報に関するものです。この付録の目的は、COFF オブジェクト・ファイルの内部フォーマットに関する補足情報を提供することです。

項目 **ページ**

A.1 COFF ファイルの構造

COFF オブジェクト・ファイルの各要素は、ファイルのセクションとシンボリック・デバッグ情報を記述しています。次の内容があります。

- ❑ ファイル・ヘッダ
- ❑ オプション・ヘッダ情報
- ❑ セクション・ヘッダのテーブル
- ❑ 初期化されたセクションごとの生データ
- ❑ 初期化されたセクションごとの再配置情報
- ❑ シンボル・テーブル
- ❑ 文字列テーブル

アセンブラとリンカによって作成されるオブジェクト・ファイルは、同一の COFF 構造をもちます。ただし、最終的にリンクされるプログラムには、通常、再配置エントリが記述されていません。図 A-1 は、オブジェクト・ファイル全体の構造を示しています。

図 A-1. COFF ファイルの構造

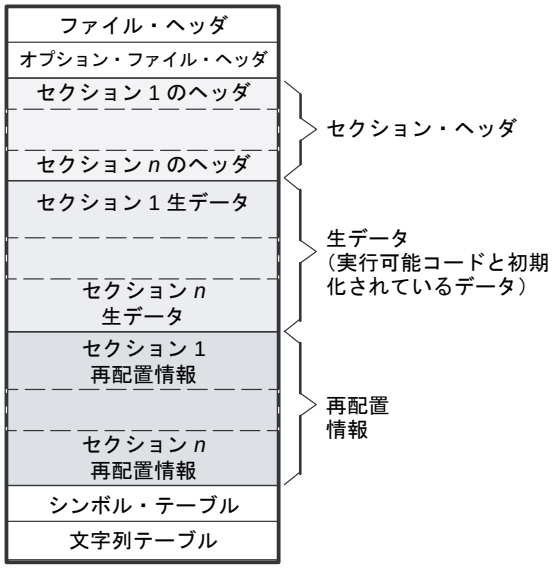
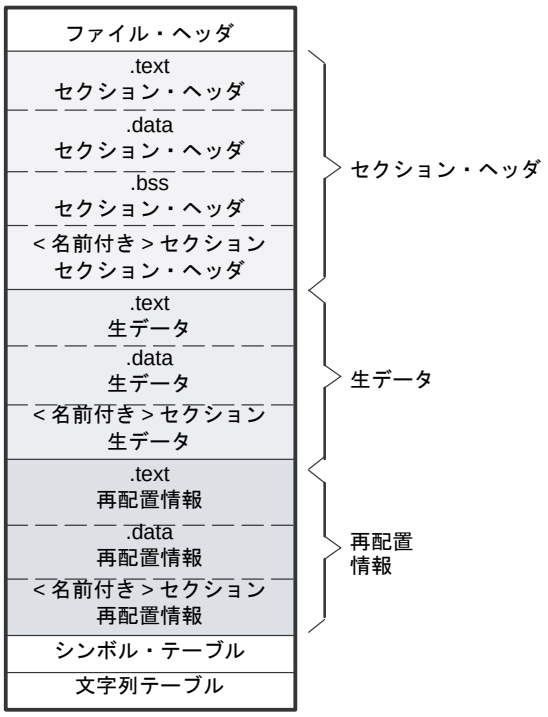


図 A-2 は、COFF オブジェクト・ファイルの一般的な例を示しています。このファイルには、.text、.data、および .bss の 3 つのデフォルト・セクションと、名前付きセクション（図中では < 名前付き > と示されています）が含まれています。デフォルトでは、.text セクション、.data セクション、初期化された名前付きセクション、.bss セクション、初期化されない名前付きセクションという順序で、オブジェクト・ファイル内に各セクションが配置されます。初期化されないセクションにもセクション・ヘッダはあります。ただし、生データ、再配置情報はありません。これは、.bss および .usect の疑似命令が初期化されないデータ用に単に空間を確保するだけだからです。初期化されないセクションには実際のコードは含まれません。

図 A-2. COFF オブジェクト・ファイル



A.2 ファイル・ヘッダの構造

ファイル・ヘッダには、オブジェクト・ファイルの一般的なフォーマットを記述した情報（22 バイト）が含まれています。表 A–1 は、COFF ファイル・ヘッダの構造を示しています。

表 A–1. ファイル・ヘッダの内容

バイト数	型	説明
0-1	Unsigned short integer	バージョン ID。COFF ファイル構造のバージョンを示します。
2-3	Unsigned short integer	セクション・ヘッダの数
4-7	Long integer	時刻および日付スタンプ。ファイルの作成日時を示します。
8-11	Long integer	ファイル・ポインタ。シンボル・テーブルの開始アドレスが入ります。
12-15	Long integer	シンボル・テーブル内のエントリの数
16-17	Unsigned short integer	オプション・ヘッダのバイト数。このフィールドには 0 か 28 のいずれかが入ります。0 の場合、オプション・ファイル・ヘッダは存在しません。
18-19	Unsigned short integer	フラグ（表 A–2 を参照してください）。
20-21	Unsigned short integer	ターゲット ID。マジック・ナンバは、ファイルが TMS320C55x™ システムで実行可能なことを示します。

表 A–2 は、ファイル・ヘッダのバイト 18 ～ 19 に格納されるフラグのリストです。これらのフラグは任意の数および組み合わせで同時に設定される場合があります（たとえば、バイト 18 ～ 19 に 0003h が設定された場合は、F_RELFLG と F_EXEC が両方とも設定されたことを示します）。

表 A–2. ファイル・ヘッダのフラグ（バイト 18 ～ 19）

ニーモニック	フラグ	説明
F_RELFLG	0001h	再配置情報はファイルから除去されました。
F_EXEC	0002h	ファイルは再配置可能です（未解決の外部参照は含まれていません）。
	0004h	予約済み。
F_LSYMS	0008h	ローカル・シンボルはファイルから除去されました。
F_LITTLE	0100h	このファイルのバイト順序は、C55x デバイスが使用できる形式です（1 ワード当たり 16 ビットで最下位のバイトが最初に置かれています）。
F_SYMMERGE	1000h	重複するシンボルは除去されました。

A.3 オプション・ファイル・ヘッダのフォーマット

リンカは、オプション・ファイル・ヘッダを作成して、ダウンロード時にそのヘッダを使用して再配置を実行します。部分的にリンクされるファイルには、オプション・ファイル・ヘッダは含まれません。表 A–3 は、オプション・ファイル・ヘッダのフォーマットを示しています。

表 A–3. オプション・ファイル・ヘッダの内容

バイト数	型	説明
0-1	Short integer	マジック・ナンバ（SunOS または HP-UX の場合は 108h、DOS の場合は 801h）
2-3	Short integer	バージョン・スタンプ
4-7	Long integer	実行可能なコードのサイズ（バイト数）
8-11	Long integer	初期化された .data セクションのサイズ（バイト数）
12-15	Long integer	初期化されていない .bss セクションのサイズ（バイト数）
16-19	Long integer	エントリ・ポイント
20-23	Long integer	実行可能なコードの開始アドレス
24-27	Long integer	初期化されたデータの開始アドレス

A.4 セクション・ヘッダの構造

COFF オブジェクト・ファイルには、オブジェクト・ファイル内の各セクションの開始位置を定義するセクション・ヘッダのテーブルが含まれています。それぞれのセクションには、独自のセクション・ヘッダがあります。表 A–4 は、COFF ファイルのセクション・ヘッダの内容を示しています。

8 文字より長いセクション名は文字列テーブルに格納されます。通常は、シンボル名が格納されるシンボル・テーブル・エントリ内のフィールドには、シンボル名の代わりに文字列テーブル内にあるシンボル名へのポインタが格納されます。

表 A–4. セクション・ヘッダの内容

バイト	型	説明
0-7	Character	このフィールドには次のいずれかが格納されます。 1）8 文字のセクション名（未使用部分にはヌルが埋め込まれます） 2）文字列テーブルへのポインタ（セクション名が 8 文字よりも長い場合）
8-11	Long integer	セクションの物理アドレス
12-15	Long integer	セクションの仮想アドレス
16-19	Long integer	セクション・サイズ（バイト数）
20-23	Long integer	生データへのファイル・ポインタ
24-27	Long integer	再配置エントリへのファイル・ポインタ
28-31	Long integer	予約済み
32-35	Unsigned long	再配置エントリの数
36-39	Unsigned long	予約済み
40-43	Unsigned long	フラグ（表 A–5 を参照してください）。
44-45	Short	予約済み
46-47	Unsigned short	メモリ・ページ番号

表 A–5 は、セクション・ヘッダに格納されるフラグのリストを示しています。複数のフラグを組み合わせて格納できます。たとえば、フラグ・バイトが 024h に設定されている場合は、STYP_GROUP と STYP_TEXT の両方が格納されます。

表 A–5. セクション・ヘッダのフラグ

ニーモニック	フラグ	説明
STYP_REG	0000h	通常セクション（割り当て、再配置、およびロードが行われます）
STYP_DSECT	0001h	ダミー・セクション（再配置は行われますが、割り当てとロードは行われません）
STYP_NOLOAD	0002h	非ロード・セクション（割り当てと再配置は行われますが、ロードは行われません）
STYP_GROUP	0004h	グループ・セクション（いくつかの入力セクションから構成されます）
STYP_PAD	0008h	埋め込みセクション（ロードは行われますが、割り当てと再配置は行われません）
STYP_COPY	0010h	コピー・セクション（再配置とロードは行われますが、割り当ては行われません。再配置エントリは通常どおり処理されます）
STYP_TEXT	0020h	実行可能なコードが書き込まれているセクション
STYP_DATA	0040h	初期化されたデータが含まれているセクション
STYP_BSS	0080h	初期化されないデータが含まれているセクション
STYP_CLINK	4000h	条件付きでリンクされるセクション

注： ロードという用語は、セクションの生データがオブジェクト・ファイル内に現れることを意味します。

図 A-3 は、セクション・ヘッダ内のポインタが、オブジェクト・ファイル内の .text セクションに関連する要素をどのように指すかを示しています。

図 A-3. .text セクションのセクション・ヘッダのポインタ

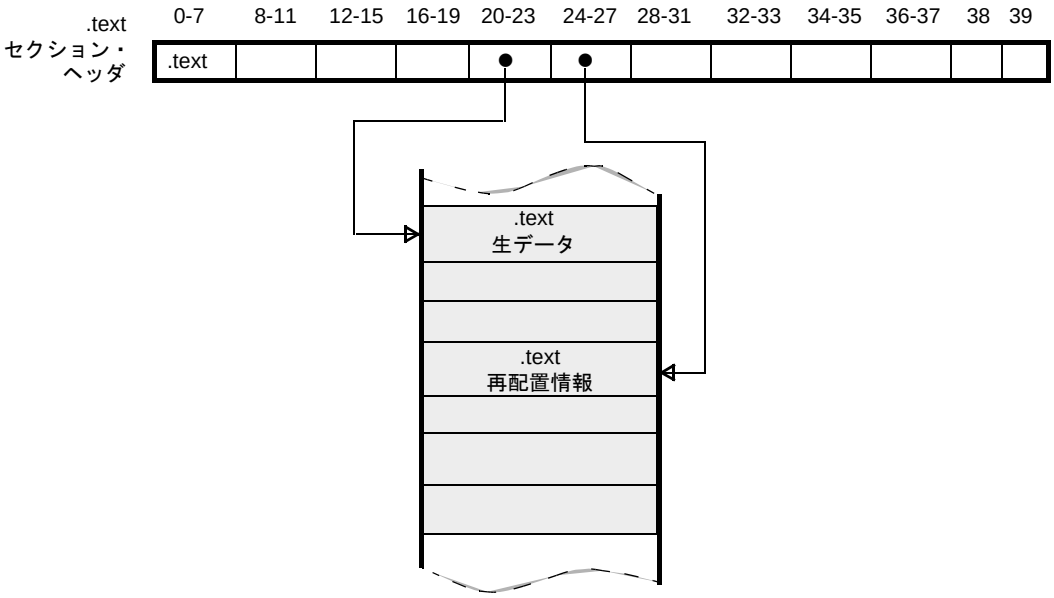


図 A-2（A-3 ページ）に示すように、（.bss および .usect の疑似命令によって作成される）初期化されないセクションは、このフォーマットとは異なります。初期化されないセクションにはセクション・ヘッダはありますが、生データや再配置情報はありません。初期化されないセクションは、オブジェクト・ファイル内で実際に空間を占有することはありません。このため、初期化されないセクションについては、再配置エントリの数およびファイル・ポインタは 0 に設定されます。初期化されないセクションのヘッダは、変数のための空間をメモリ・マップ内にどれだけ確保するかをリンカに指示するだけです。

A.5 再配置情報の構造化

COFF オブジェクト・ファイルには、再配置可能な参照ごとに 1 つの再配置エントリがあります。アセンブラは、この再配置エントリを自動的に生成します。リンカは、各入力セクションを読み取るときに再配置エントリを読み取り、再配置を実行します。再配置エントリにより、各入力セクション内の参照がどのように取り扱われるかが決まります。

COFF ファイルの再配置情報エントリでは、表 A–6 に示す 12 バイトのフォーマットが使用されます。

表 A–6. 再配置エントリの内容

バイト数	型	説明
0-3	Long integer	参照の仮想アドレス
4-7	Unsigned long integer	シンボル・テーブルのインデックス
8-9	Unsigned short integer	拡張アドレス計算のための追加バイト
10-11	Unsigned short integer	再配置タイプ（表 A–7 を参照してください）。

仮想アドレスは、再配置前の現行のセクション内でのシンボルのアドレスです。仮想アドレスは、再配置が必要な場所を指定します（これは、オブジェクト・コード内でのパッチが必要なフィールドのアドレスです）。

再配置エントリを生成するコードの例を次に示します。

```
2                       .global   X
3   000000 6A00         B         X
    000001 0000!
```

この例では、再配置可能フィールドの仮想アドレスは 0001 となります。

シンボル・テーブルのインデックスは、参照するシンボルのインデックスです。上記の例では、このフィールドにはシンボル・テーブル内の X のインデックスが格納されます。セクション内のシンボルの現行のアドレスと、そのシンボルのアセンブル時のアドレスとの差が、再配置の大きさです。再配置可能フィールドは、参照シンボルと同じ大きさだけ再配置する必要があります。例では、X は再配置前に値 0 をもちます。X がアドレス 2000h に再配置される場合を考えてみましょう。これは再配置の大きさ（2000h - 0 = 2000h）なので、アドレス 1 の再配置フィールドはパッチされ、2000h が加算されます。

あるシンボルがどのセクションで定義されているかが分かれば、そのシンボルの再配置後のアドレスを求めることができます。たとえば、X は .data セクションで定義されており .data は 2000h に再配置されるとすれば、X は 2000h に再配置されます。

再配置エントリ内でシンボル・テーブルのインデックスが -1（0FFFFh）であれば、この再配置は内部再配置と呼ばれます。この場合、再配置の大きさは単に現行のセクションが再配置される大きさになります。

再配置タイプは、パッチされるフィールドのサイズを指定するとともに、パッチする値の計算方法を記述するものです。このタイプ・フィールドは、再配置可能な参照を生成するのに使用されたアドレッシング・モードにより異なります。上記の例では、参照されるシンボル（X）の実アドレスは、オブジェクト・コード内の 16 ビット・フィールドに置かれます。この実アドレスは 16 ビットの直接再配置なので、再配置タイプは R_RELWORD となります。表 A–7 は再配置タイプを示しています。指定されたフラグ・エントリは 8 進数値です。

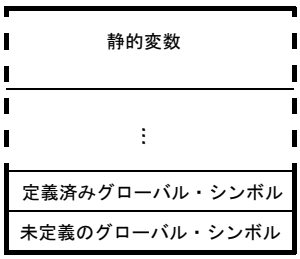
表 A–7. 再配置タイプ（バイト 10 ～ 11）

ニーモニック	フラグ	再配置タイプ
R_ABS	0000	再配置なし
R_REL24	0005	シンボルのアドレスへの 24 ビット直接参照
R_RELBYTE	0017	シンボルのアドレスへの 8 ビット直接参照
R_RELWORD	0020	シンボルのアドレスへの 16 ビット直接参照
R_RELLONG	0021	シンボルのアドレスへの 32 ビット直接参照
R_LD3_DMA	0170	バイトの 7 MSB、符号なし、DMA アドレスにて使用
R_LD3_MDP	0172	2 バイトにまたがる 7 ビット、符号なし、MDP レジスタ値として使用
R_LD3_PDP	0173	2 バイトにまたがる 9 ビット、符号なし、PDP レジスタ値として使用
R_LD3_REL23	0174	24 ビット・フィールド内の 23 ビット符号なし値
R_LD3_k8	0210	8 ビット、符号なし直接参照
R_LD3_k16	0211	16 ビット、符号なし直接参照
R_LD3_K8	0212	8 ビット、符号あり直接参照
R_LD3_K16	0213	16 ビット、符号あり直接参照
R_LD3_l8	0214	8 ビット、符号なし、PC に相対する参照
R_LD3_l16	0215	16 ビット、符号なし、PC に相対する参照
R_LD3_L8	0216	8 ビット、符号あり、PC に相対する参照
R_LD3_L16	0217	16 ビット、符号あり、PC に相対する参照
R_LD3_k4	0220	符号なし 4 ビット・シフト即値
R_LD3_k5	0221	符号なし 5 ビット・シフト即値
R_LD3_K5	0222	符号あり 5 ビット・シフト即値
R_LD3_k6	0223	符号なし 6 ビット即値
R_LD3_k12	0224	符号なし 12 ビット即値

A.6 シンボル・テーブルの構造と内容

シンボル・テーブル内のシンボルの順序は非常に重要です。シンボルは、図 A-4 に示すような順序でシンボル・テーブルに格納されます。

図 A-4. シンボル・テーブルの内容



静的変数とは、C/C++ で定義されるシンボルのうち、関数外部の記憶クラス static をもつ変数を指します。同じ名前のシンボルを複数のモジュールで使用する必要がある場合は、それらのシンボルを静的にして、それぞれのシンボルのスコープを該当するシンボルを定義しているモジュールだけに限定します（こうすると、複数の定義が矛盾しなくなります)。

シンボル・テーブル内のそれぞれのシンボル・エントリには、次の情報が格納されます。

- ❏ 名前（または文字列テーブルへのポインタ）
- ❏ 型
- ❏ 値
- ❏ シンボルが定義されたセクション
- ❏ 記憶クラス

シンボル・テーブルでは、セクション名も定義されます。

シンボル・テーブル内のシンボル・エントリは、クラスおよび型とは無関係に同一のフォーマットです。シンボル・テーブルの各エントリには、表 A–8 に示す 18 バイトの情報が格納されています。それぞれのシンボルは 18 バイトの補足エントリをもつことがあります。表 A–9（A-12 ページ）に示す特殊シンボルは、必ず補足エントリをもちます。シンボルの中には、前述の特性を全部はもたないシンボルもあります。フィールドが設定されない場合、そのフィールドはヌルになります。

表 A–8. シンボル・テーブル・エントリの内容

バイト数	型	説明
0-7	Character	このフィールドには次のいずれかが格納されます。 1）8 文字のシンボル名（未使用部分にはヌルが埋め込まれます） 2）文字列テーブルへのポインタ（シンボル名が 8 文字よりも長い場合）
8-11	Long integer	シンボルの値（記憶クラスにより異なります）
12-13	Short integer	シンボルのセクション番号
14-15	Unsigned short integer	予約済み
16	Character	シンボルの記憶クラス
17	Character	補足エントリの数（常に 0 または 1）

A.6.1 特殊シンボル

シンボル・テーブルには、コンパイラ、アセンブラ、およびリンカで生成されるいくつかの特殊シンボルが格納されます。それぞれの特殊シンボルには、通常のシンボル・テーブル情報と補足エントリが含まれています。表 A–9 はこれらの特殊シンボルを示しています。

表 A–9. シンボル・テーブル内の特殊シンボル

シンボル	説明
.file	ファイル名
.text	.text セクションのアドレス
.data	.data セクションのアドレス
.bss	.bss セクションのアドレス
etext	.text 出力セクションの後にある、次に使用可能なアドレス
edata	.data 出力セクションの後にある、次に使用可能なアドレス
end	.bss 出力セクションの後にある、次に使用可能なアドレス

A.6.2 シンボル名のフォーマット

シンボル・テーブル・エントリの最初の 8 バイト（バイト 0 ～ 7）は、シンボルの名前を示します。

- シンボル名が 8 文字以内の場合、このフィールドは *Character* 型になります。このシンボル名は必要に応じてヌルが埋め込まれ、バイト 0 ～ 7 に格納されます。
- シンボル名が 8 文字を超えている場合、このフィールドは 2 つの倍長整数（long）型として扱われます。この場合は、シンボル名全体が文字列テーブルに格納されます。バイト 0 ～ 3 には 0 が入り、バイト 4 ～ 7 は文字列テーブルへのオフセットとなります。

A.6.3 文字列テーブルの構造

8 文字より長いシンボル名は文字列テーブルに格納されます。通常は、シンボル名が格納されるシンボル・テーブル・エントリ内のフィールドには、シンボル名の代わりに文字列テーブル内にあるシンボル名へのポインタが格納されます。文字列テーブルではそれぞれの名前がヌル・バイトで区切られ、連続して格納されます。文字列テーブルの最初の 4 バイトには、文字列テーブルのサイズがバイト数単位で入ります。このため、文字列テーブルへのオフセットは 4 以上になります。

文字列テーブルのアドレスは、シンボル・テーブルのアドレスおよびシンボル・テーブルのエントリ数から計算されます。

図 A-5 は、Adaptive-Filter と Fourier-Transform という 2 つのシンボル名が格納されている文字列テーブルを示しています。この文字列テーブル内のインデックスは、Adaptive-Filter については 4、Fourier-Transform については 20 となります。

図 A-5. 文字列テーブル

38			
'A'	'd'	'a'	'p'
't'	'i'	'v'	'e'
'-'	'F'	'i'	'l'
't'	'e'	'r'	'\0'
'F'	'o'	'u'	'r'
'i'	'e'	'r'	'-'
'T'	'r'	'a'	'n'
's'	'f'	'o'	'r'
'm'	'\0'		

A.6.4 記憶クラス

シンボル・テーブル・エントリのバイト 16 は、シンボルの記憶クラスを示します。記憶クラスとは、C/C++ コンパイラがシンボルにアクセスする方式を指します。表 A–10 は、有効な記憶クラスを示しています。

表 A–10. シンボルの記憶クラス

ニーモニック	値	記憶クラス	ニーモニック	値	記憶クラス
C_NULL	0	記憶クラスなし	C_UNTAG	12	予約済み
C_AUTO	1	予約済み	C_TPDEF	13	予約済み
C_EXT	2	外部シンボル	C_USTATIC	14	初期化されない静的
C_STAT	3	静的	C_ENTAG	15	予約済み
C_REG	4	予約済み	C_MOE	16	予約済み
C_EXTREF	5	外部定義	C_REGPARM	17	予約済み
C_LABEL	6	ラベル	C_FIELD	18	予約済み
C_ULABEL	7	未定義ラベル	C_BLOCK	100	予約済み
C_MOS	8	予約済み	C_FCN	101	予約済み
C_ARG	9	予約済み	C_EOS	102	予約済み
C_STRTAG	10	予約済み	C_FILE	103	予約済み
C_MOU	11	予約済み	C_LINE	104	ユーティリティ・プログラムでのみ使用

.text、.dat、および .bss シンボルは、C_STAT 記憶クラスに制限されています。

A.6.5 シンボルの値

シンボル・テーブル・エントリのバイト 8 ～ 11 は、シンボルの値を示します。C_EXT、C_STAT、および C_LABEL 記憶クラスは、再配置可能なアドレスをもっています。

あるシンボルの記憶クラスが C_FILE である場合、そのシンボルの値は次の .file シンボルへのポインタです。このため、シンボル・テーブル内に .file シンボルによる片方向のリンク・リストが作成されます。.file シンボルがそれ以上ない場合、最後の .file シンボルはシンボル・テーブル内の最初の .file シンボルを指します。

再配置可能なシンボルの値は、そのシンボルの仮想アドレスです。リンカがあるセクションを再配置すると、再配置可能なシンボルの値はそれに応じて変更されます。

A.6.6 セクション番号

シンボル・テーブル・エントリのバイト 12 ～ 13 には、シンボルが定義されているセクションを示す番号が格納されます。表 A–11 は、これらの番号とそれぞれの番号が示すセクションを示しています。

表 A–11. セクション番号

ニーモニック	セクション番号	説明
なし	-2	予約済み
N_ABS	-1	絶対シンボル
N_UNDEF	0	未定義の外部シンボル
N_SCNUM	1	.text セクション（通常）
N_SCNUM	2	.data セクション（通常）
N_SCNUM	3	.bss セクション（通常）
N_SCNUM	4 ～ 32,767	名前付きセクションのセクション番号（名前付きセクションの出現順）

.text、.data、または .bss セクションが存在しなかった場合は、名前付きセクションの番号は 1 から始まります。

セクション番号が 0、-1、または -2 のシンボルは、セクション内では定義されていません。セクション番号 -1 は、値をもっているが、再配置可能でないシンボルであることを示します。セクション番号 0 は、現行のファイルでは定義されていない再配置可能な外部シンボルであることを示します。

A.6.7 補足エントリ

それぞれのシンボル・テーブル・エントリは、補足エントリを **1** つ持つ場合と、持たない場合があります。このシンボル・テーブル補足エントリはシンボル・テーブル・エントリと同じバイト数（18）をもちます。表 A–12 は、テーブル補足エントリのフォーマットを示しています。

表 A–12. テーブル補足エントリ用のセクション・フォーマット

バイト数	型	説明
0-3	Long integer	セクションの長さ
4-6	Unsigned short integer	再配置エントリの数
7-8	Unsigned short integer	行番号エントリの数
9-17	—	未使用（ゼロが埋め込まれます）

付録 B

シンボリック・デバッグ疑似命令

アセンブラは、TMS320C55x C/C++ コンパイラがシンボリック・デバッグ用に使用する疑似命令をサポートします。これらの疑似命令は、2 つのデバッグ・フォーマット、DWARF および COFF について異なっています。

これらの疑似命令は、アセンブラ言語プログラマによる使用を意図しません。これらの疑似命令に必要な引数は、手動で計算するのが難しい場合があります。使用する際には、コンパイラ、アセンブラ、およびデバッガ間の所定の取り決めに従わなければなりません。この付録では、これらの疑似命令について情報を提供します。

項目 **ページ**

B.1 DWARF デバッグ・フォーマット

DWARF シンボリック・デバッグ疑似命令のサブセットは、プログラム分析のためにコンパイラが作成するアセンブリ言語ファイルに必ず出力されます。シンボリック・デバッグ全体に使う完全な設定を出力するには、次のように -g オプションを指定してコンパイラを起動します。

```
cl6x -g -k input_file
```

-k オプションはコンパイラに指示を与え、生成されたアセンブリ・ファイルを保持します。

すべてのシンボリック・デバッグ疑似命令の生成を抑制するには、-symdebug:none オプションを使ってコンパイラを起動します。

```
cl6x --symdebug:none -k input_file
```

DWARF デバッグ・フォーマットは、以下の疑似命令で構成されています。

- **.dwtag** および **.dwendtag** 疑似命令は、.debug_info セクションにおいてデバッグ情報エントリ（DIE）を定義します。
- **.dwattr** 疑似命令は、既存の DIE に属性を追加します。
- **.dwpsn** 疑似命令は、C/C++ 文のソース位置を識別します。
- **.dwcie** および **.dwendentry** 疑似命令は、.debug_frame セクションにおいて共通情報エントリ（CIE）を定義します。
- **.dwfde** および **.dwendentry** 疑似命令は、.debug_frame セクションにおいてフレーム概要エントリ（FDE）を定義します。
- **.dwcfa** 疑似命令は、CIE または FDE のために呼び出しフレーム命令を定義します。

B.2 COFF デバッグ・フォーマット

現在 COFF シンボリック・デバッグは使われていません。これらの疑似命令は、下位互換のためだけにサポートされています。C++ のサポートがない等、COFF シンボリック・デバッグにある多くの制限を克服するために、シンボリック・デバッグ・フォーマットとして DWARF に切り替える決定が成されました。

COFF デバッグ・フォーマットは、以下の疑似命令で構成されています。

- **.sym** 疑似命令は、グローバル変数、ローカル変数、または関数を定義します。いくつかのパラメータにより、各種のデバッグ情報を変数または関数に関連付けることができます。
- **.stag**、**.etag**、および **.utag** 疑似命令は、それぞれ、構造体、列挙型、および共用体を定義します。**.member** 疑似命令は、構造体、列挙型、または共用体のメンバを指定します。**.eos** 疑似命令は、構造体、列挙型、または共用体の定義を終了させます。
- **.func** および **.endfunc** の疑似命令は、C/C++ 関数の最初と最後の行を指定します。
- **.block** および **.endblock** の疑似命令は、C/C++ ブロックの境界を指定します。
- **.file** 疑似命令は、現行のソース・ファイル名を識別するシンボルを、シンボル・テーブル内に定義します。
- **.line** 疑似命令は、C/C++ ソース文の行番号を識別します。

B.3 デバッグ疑似命令の構文

表 B–1 では、シンボリック・デバッグ疑似命令の一覧をアルファベット順に記載しています。C/C++ コンパイラについての詳細は、TMS320C55x オプティマイジング（最適化）C/C++ コンパイラ ユーザーズ・ガイドを参照してください。

表 B–1. シンボリック・デバッグ疑似命令

ラベル	疑似命令	引数
	.block	[開始行番号]
	.dwattr	DIE ラベル , DIE 属性名（DIE 属性値）[,DIE 属性名（属性値）[, ...]
	.dwcfa	呼び出しフレーム命令コード[, オペランド[, オペランド]]
CIE ラベル	**.dwcie**	バージョン , リターン・アドレス・レジスタ
	.dwendentry	
	.dwendtag	
	.dwfde	CIE ラベル
	.dwpsn	“ ファイル名 ”, 行番号 , カラム番号
DIE ラベル	**.dwtag**	DIE タグ名 , DIE 属性名（DIE 属性値）[,DIE 属性名（属性値）[, ...]
	.endblock	[終了行番号]
	.endfunc	[終了行番号[, レジスタ・マスク [, フレーム・サイズ]]]
	.eos	
	.etag	名前[, サイズ]
	.file	“ ファイル名 ”
	.func	[開始行番号]
	.line	行番号[, アドレス]
	.member	名前 , 値[, 型 , 記憶クラス , サイズ , タグ , dims]
	.stag	名前[, サイズ]
	.sym	名前 , 値[, 型 , 記憶クラス , サイズ , タグ , dims]
	.utag	名前[, サイズ]

付録 C

XML リンク情報ファイルの説明

リンカは、--xml_link_info *file* オプションを介して、XML リンク情報ファイルの生成をサポートします。このオプションは、リンカがリンク結果の詳細情報を含む見やすい XML ファイルを生成するようにします。このファイルに入れられる情報には、リンカが生成するマップ・ファイルに現在生成されているすべての情報が含まれます。

リンカの進歩に応じて XML リンク情報ファイルは拡張され、リンカの結果の静的分析に有効な追加情報も含まれる場合があります。

この付録では、リンカが XML リンク情報ファイルに生成する全要素を列挙します。

項目 **ページ**

C.1 XML 情報ファイル要素タイプ

これらの要素タイプは、リンカによって生成されます。

- **コンテナ要素**は、オブジェクトを記述する他の要素を含むオブジェクトを示します。コンテナ要素には id 属性があり、そのため他の要素からアクセス可能です。
- **文字列要素**には、値の文字列表記があります。
- **定数要素**には、値の 32 ビットの符号なし表記があります（0x 接頭部付き）。
- **参照要素**は、別のコンテナ要素へのリンクを指定する idref 属性を持つ、空の要素です。

C.2 節で、要素タイプは、要素に関する記述に続く括弧内の各要素について指定されます。たとえば、<link_time> 要素は、リンク実行時間の一覧です（文字列）。

C.2 文書要素

ルート要素、または文書要素は **<link_info>** です。XML リンク情報ファイルにある他のすべての要素は、<link_info> 要素の子要素です。次の節では、XML 情報ファイルが持つ可能性のある要素について説明します。

C.2.1 ヘッダ要素

XML リンク情報ファイルの最初の要素は、リンカおよびリンク・セッションについての一般的な情報を提供します。

- **<banner>** 要素は、実行可能なリンク名およびバージョン情報の一覧です（文字列）。
- **<copyright>** 要素は、TI の著作権情報の一覧です（文字列）。
- **<link_time>** 要素は、リンク実行時間の一覧です（文字列）。
- **<link_timestamp>** 要素は、リンク時間のタイムスタンプ表示です（符号なし 32 ビット整数）。
- **<output_file>** 要素は、生成されたリンク出力ファイル名の一覧です（文字列）。
- **<entry_point>** 要素は、リンカ（コンテナ）の決定に従い、2 つのエントリを使ってプログラム・エントリ・ポイントを指定します。
  - **<name>** は、エントリ・ポイント・シンボル名（エントリ・ポイント・シンボルがある場合）です（文字列）。
  - **<address>** は、エントリ・ポイント・アドレスです（定数）。

例 C–1. hi.out 出力ファイルのためのヘッダ要素

```
<banner>TMS320Cxx COFF Linker       Version x.xx (Jan  6 2003)</banner>
<copyright>Copyright (c) 1996-2003 Texas Instruments Incorporated</copyright>
<link_time>Mon Jan  6 15:38:18 2003</link_time>
<output_file>hi.out</output_file>
<entry_point>
   <name>_c_int00</name>
   <address>0xaf80</address>
</entry_point>
```

C.2.2 入力ファイル・リスト

XML リンク情報ファイルの次のセクションは、入力ファイル・リストです。これは **<input_file_list>** コンテナ要素で区切られます。<input_file_list> は、<input_file> 要素をいくつでも含むことができます。

それぞれの **<input_file>** インスタンスは、リンクに含まれる入力ファイルを指定します。それぞれの <input_file> には、<object_component> などの他の要素によって参照できる id 属性があります。<input_file> は、以下の要素を囲むコンテナ要素です。

- **<path>** 要素は、ディレクトリ・パスがある場合、これの名前を付けます（文字列）。
- **<kind>** 要素は、ファイル・タイプをアーカイブまたはオブジェクトのいずれかに指定します（文字列）。
- **<file>** 要素は、アーカイブ名またはファイル名を指定します（文字列）。
- **<name>** 要素は、オブジェクト・ファイル名またはアーカイブ・メンバ名を指定します（文字列）。

例 C–2. hi.out 出力ファイルのための入力ファイル・リスト

```
<input_file_list>
   <input_file id="fl-1">
      <kind>object</kind>
      <file>hi.obj</file>
      <name>hi.obj</name>
   </input_file>
   <input_file id="fl-2">
      <path>/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>boot.obj</name>
   </input_file>
   <input_file id="fl-3">
      <path>/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>exit.obj</name>
   </input_file>
   <input_file id="fl-4">
      <path>/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>printf.obj</name>
   </input_file>
    ...
</input_file_list>
```

C.2.3 オブジェクト・コンポーネント・リスト

XML リンク情報ファイルの次のセクションは、リンクに含まれるすべてのオブジェクト・コンポーネントの指定についてです。オブジェクト・コンポーネントの例は、入力セクションです。一般的にオブジェクト・コンポーネントは、リンカが操作できる最小単位のオブジェクトです。

<object_component_list> は、<object_component> 要素をいくつでも囲むコンテナ要素です。

それぞれの **<object_component>** は、1 つのオブジェクト・コンポーネントを指定します。それぞれの <object_component> には id 属性があるため、<logical_group> などの他の要素によって直接参照できます。<object_component> は、以下の要素を囲むコンテナ要素です。

- **<name>** 要素は、オブジェクト・コンポーネントの名前を付けます（文字列）。
- **<load_address>** 要素は、オブジェクト・コンポーネントのロード時アドレスを指定します（定数）。
- **<run_address>** 要素は、オブジェクト・コンポーネントの実行時アドレスを指定します（定数）。
- **<size>** 要素は、オブジェクト・コンポーネントのサイズを指定します（定数）。
- **<input_file_ref>** 要素は、オブジェクト・コンポーネントが開始されるソース・ファイルを指定します（参照）。

例 C–3. fl-4 入力ファイルのためのオブジェクト・コンポーネント・リスト

```
<object_component id="oc-20">
   <name>.text</name>
   <load_address>0xac00</load_address>
   <run_address>0xac00</run_address>
   <size>0xc0</size>
   <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-21">
   <name>.data</name>
   <load_address>0x80000000</load_address>
   <run_address>0x80000000</run_address>
   <size>0x0</size>
   <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-22">
   <name>.bss</name>
   <load_address>0x80000000</load_address>
   <run_address>0x80000000</run_address>
   <size>0x0</size>
   <input_file_ref idref="fl-4"/>
</object_component>
```

C.2.4 ロジカル・グループ・リスト

XML リンク情報ファイルの **<logical_group_list>** セクションは、リンカが生成するマップ・ファイルの出力セクション・リストに類似しています。しかし、XML リンク情報ファイルには、GROUP および UNION 出力セクションの指定が含まれています。これらはマップ・ファイルには表記されません。<logical_group_list> で発生する可能性のあるリスト・アイテムは 3 種類あります。

- **<logical_group>** は、セクションの指定、またはオブジェクト・コンポーネントまたはロジカル・グループ・メンバのリストを持つ GROUP の指定です。それぞれの <logical_group> 要素には id があるため、他の要素から参照できます。それぞれの <logical_group> は、以下の要素を囲むコンテナ要素です。
  - **<name>** 要素は、ロジカル・グループの名前を付けます（文字列）。
  - **<load_address>** 要素は、ロジカル・グループのロード時アドレスを指定します（定数）。
  - **<run_address>** 要素は、ロジカル・グループの実行時アドレスを指定します（定数）。
  - **<size>** 要素は、ロジカル・グループのサイズを指定します（定数）。
  - **<contents>** 要素は、このロジカル・グループの要素の一覧です（コンテナ）。これらの要素は、このロジカル・グループに含まれる各メンバ・オブジェクトを示しています。
    - **<object_component_ref>** は、このロジカル・グループのオブジェクト・コンポーネントです（参照）。
    - **<logical_group_ref>** は、このロジカル・グループに含まれるロジカル・グループです（参照）。
- **<overlay>** は、特別な種類のロジカル・グループです。UNION または同じメモリ空間を共有する一連のオブジェクトを示します（コンテナ）。それぞれの <overlay> 要素には id があるため、配置マップの <allocated_space> 要素など他の要素から参照できます。各 <overlay> には以下の要素があります。
  - **<name>** 要素は、オーバーレイの名前を付けます（文字列）。
  - **<run_address>** 要素は、オーバーレイの実行時アドレスを指定します（定数）。
  - **<size>** 要素は、ロジカル・グループのサイズを指定します（定数）。

- **<contents>** コンテナ要素は、このオーバーレイに含まれる要素の一覧です。これらの要素は、このロジカル・グループに含まれる各メンバ・オブジェクトを示しています。
  - **<object_component_ref>** は、このオーバーレイに含まれるオブジェクト・コンポーネントです（参照）。
  - **<logical_group_ref>** は、このオーバーレイに含まれるロジカル・グループです（参照）。

❑ **<split_section>** は、もう一つの特別な種類のロジカル・グループです。複数のメモリ領域を分割するロジカル・グループの集合を示しています。それぞれの <split_section> 要素には id があるため、他の要素から参照できます。id は以下の要素で構成されています。

- **<name>** 要素は、分割セクションに名前を付けます（文字列）。
- **<contents>** 要素は、この分割セクションに含まれる要素の一覧です（コンテナ）。<logical_group_ref> 要素は、この分割セクションに含まれる各メンバ・オブジェクトを参照しています。参照される各要素は、この分割セクションに含まれるロジカル・グループです（参照）。

例 C–4. fl-4 入力ファイルのためのロジカル・グループ・リスト

```
<logical_group_list>
   ...
<logical_group id="lg-7">
      <name>.text</name>
      <load_address>0x20</load_address>
      <run_address>0x20</run_address>
      <size>0xb240</size>
      <contents>
         <object_component_ref idref="oc-34"/>
         <object_component_ref idref="oc-108"/>
         <object_component_ref idref="oc-e2"/>
      ...
      </contents>
   </logical_group>
   ...
   <overlay id="lg-b">
      <name>UNION_1</name>
      <run_address>0xb600</run_address>
      <size>0xc0</size>
      <contents>
         <object_component_ref idref="oc-45"/>
         <logical_group_ref idref="lg-8"/>
      </contents>
   </overlay>
    ...
   <split_section id="lg-12">
      <name>.task_scn</name>
      <size>0x120</size>
      <contents>
         <logical_group_ref idref="lg-10"/>
         <logical_group_ref idref="lg-11"/>
      </contents>
   ...
</logical_group_list>
```

C.2.5 配置マップ

<placement_map> 要素は、アプリケーションにおけるすべての名前付きメモリ領域の詳細なメモリ配置を記述しています。その中には、特定のメモリ領域に配置されたロジカル・グループ間の未使用空間も含まれています。

- **<memory_area>** は、名前付きメモリ領域内の詳細な配置の記述です（コンテナ）。記述はこれらの項目で構成されています。
  - **<name>** は、メモリ領域の名前を付けます（文字列）。
  - **<page_id>** は、このメモリ領域が定義されるメモリ・ページに id を与えます（定数）。
  - **<origin>** は、メモリ領域の開始アドレスを指定します（定数）。
  - **<length>** は、メモリ領域の長さを指定します（定数）。
  - **<used_space>** は、この領域に割り当てられた空間の大きさを指定します（定数）。
  - **<unused_space>** は、この領域で使用可能な空間の大きさを指定します（定数）。
  - **<attributes>** は、この領域に関連する RWXI 属性の一覧です（RWXI 属性がある場合）（文字列）。
  - **<fill_value>** は、メモリ領域で埋め込み疑似命令が指定される場合、未使用空間に配置される埋め込み値を指定します（定数）。
  - **<usage_details>** は、このメモリ領域に割り当てられたフラグメントまたは使用可能なフラグメントの詳細についての一覧です。フラグメントがロジカル・グループに割り当てられると、<logical_group_ref> 要素が提供され、該当するロジカル・グループの詳細へのアクセスが容易になります。全フラグメントの指定には、<start_address> 要素および <size> 要素があります。
    - **<allocated_space>** 要素は、このメモリ領域内に割り当てられたフラグメントの詳細を提供します（コンテナ）。

      <start_address> は、フラグメントのアドレスを指定します（定数）。

      <size> は、フラグメントのサイズを指定します（定数）。

      <logical_group_ref> は、このフラグメントに割り当てられるロジカル・グループへの参照を提供します（参照）。
    - **<available_space>** 要素は、このメモリ領域内で使用できるフラグメントの詳細を提供します（コンテナ）。

      <start_address> は、フラグメントのアドレスを指定します（定数）。

      <size> は、フラグメントのサイズを指定します（定数）。

例 C–5. fl-4 入力ファイルのための配置マップ

```
<placement_map>
   <memory_area>
      <name>PMEM</name>
      <page_id>0x0</page_id>
      <origin>0x20</origin>
      <length>0x100000</length>
      <used_space>0xb240</used_space>
      <unused_space>0xf4dc0</unused_space>
      <attributes>RWXI</attributes>
      <usage_details>
         <allocated_space>
            <start_address>0x20</start_address>
            <size>0xb240</size>
            <logical_group_ref idref="lg-7"/>
         </allocated_space>
         <available_space>
            <start_address>0xb260</start_address>
            <size>0xf4dc0</size>
         </available_space>
      </usage_details>
   </memory_area>
   ...
</placement_map>
```

C.2.6 シンボル・テーブル

<symbol_table> には、リンクに含まれている全グローバル・シンボルの一覧があります。この一覧には、シンボルの名前と値についての情報があります。symbol_table 一覧は、将来的に、タイプ情報、シンボルが定義されるオブジェクト・コンポーネント、記憶クラスなどを提供する予定です。

<symbol> は、次の要素を使ってシンボルの名前と値を指定するコンテナ要素です。

- **<name>** 要素は、シンボル名を指定します（文字列）。
- **<value>** 要素は、シンボル値を指定します（定数）。

例 C–6. fl-4 入力ファイルのためのシンボル・テーブル

```
<symbol_table>
   <symbol>
      <name>_c_int00</name>
      <value>0xaf80</value>
   </symbol>
   <symbol>
      <name>_main</name>
      <value>0xb1e0</value>
   </symbol>
   <symbol>
      <name>_printf</name>
      <value>0xac00</value>
   </symbol>
   ...
</symbol_table>
```

付録 D

用語集

A

ASCII： 情報交換用米国標準コード（American Standard Code for Information Exchange）。英数字の情報を表現、交換するための標準コンピュータ・コード

B

.bss： デフォルトの COFF セクションの 1 つ。.bss 疑似命令を使用すると、メモリ・マップに一定量の空間を確保でき、その空間に後でデータを格納することができます。.bss セクションは初期化されません。

C

C コンパイラ： C のソース文をアセンブリ言語のソース文に変換するためのプログラム

COFF： 共通オブジェクト・ファイル・フォーマット。セクションの概念をサポートすることにより、モジュラ・プログラミングを促進する、バイナリ・オブジェクト・ファイルのフォーマット。

D

.data： デフォルトの COFF セクションの 1 つ。.data セクションは、初期化されたデータを含む初期化されたセクションです。.data 疑似命令を使用すると、コードを .data セクションにアセンブルできます。

G

GROUP： SECTIONS 疑似命令のオプションの 1 つ。指定された出力セクションを（グループとして）連続して強制的に割り当てます。

H

Hex 変換ユーティリティ： COFF ファイルを受け取り、そのファイルを EPROM プログラマにロードできる標準 ASCII 16 進フォーマットの 1 つに変換するプログラム

R

RAM モデル： リンカが C コードをリンクするときに使用する自動初期化モデル。-cr オプションを指定してリンカを起動するとき、リンカは RAM モデルを使用します。RAM モデルを使用すると、実行時ではなくロード時に変数を初期化できます。

ROM 幅： 各出力ファイルの幅（ビット単位）。厳密には、ファイル内の 1 つのデータ値の幅を意味します。ユーティリティがデータをどのように分割して出力ファイルに入れるかは、ROM 幅により決まります。ターゲット・ワードがメモリ・ワードにマップされた後、メモリ・ワードは 1 つまたは複数の出力ファイルに分割されます。出力ファイル数は ROM 幅により決まります。

ROM モデル： リンカが C コードをリンクするときに使用する自動初期化モデル。-c オプションを指定してリンカを起動するときに、リンカは ROM モデルを使用します。ROM モデルでは、リンカはデータ・テーブルの .cinit セクションをメモリにロードし、変数は実行時に初期化されます。

S

SPC（Section Program Counter）： セクション内の現在位置を追跡するためのアセンブラの一要素。各セクションには専用の SPC があります。

static： 関数またはプログラムに限定されているスコープをもつ変数の一種。静的変数の値は、それが使用されている関数またはプログラムが終了しても破棄されません。再度その関数またはプログラムを開始すると以前の値が使用されます。

T

.text： デフォルトの COFF セクションの 1 つ。.text セクションは、実行可能コード書き込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。

U

UNION： SECTIONS 疑似命令のオプションの 1 つ。このオプションを使用すると、リンカは複数のセクションに同じ実行アドレスを割り当てます。

unsigned： 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

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収集するためのソフトウェア・プログラム。アーカイバを使用すると、新しいメンバを追加するだけでなく、アーカイブ・ライブラリのメンバのファイルを削除、抽出、または置換できます。

アーカイブ・ライブラリ： 単一のファイル内で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組み込んだソース・ファイルから、機械語のプログラムを作成するソフトウェア・プログラム。アセンブラはシンボリックな命令コードを絶対的な命令コードに置換し、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アドレスに置換します。

アセンブル時定数： .set 疑似命令を指定して定数値を割り当てたシンボル

い

位置合わせ： リンカが出力セクションを *n* ビット境界上のアドレスに置くこと。ここで、*n* は 2 の累乗。位置合わせの指定には、SECTIONS リンカ疑似命令を使用します。

インクリメンタル・リンク： 複数のパスでリンクされるファイルをリンクすること。多くの場合、セクションをリンクしてから、これらのリンク済みセクションをまとめてリンクするかなり大きなファイルを意味します。

え

エミュレータ： TMS320C55x の機能をエミュレートするためのハードウェア開発システム

エントリ・ポイント： ターゲット・メモリでの実行開始点

お

オーバーレイ・ページ： 別のメモリのセクションと同じアドレス範囲にマップされた、物理メモリのセクション。どの範囲をアクティブにするかは、ハードウェア・スイッチにより決定されます。

オブジェクト・ファイル： 機械語オブジェクト・コードを書き込むための、アセンブルまたはリンクされたファイル

オブジェクト・フォーマット変換プログラム： COFF オブジェクト・ファイルを Intel フォーマットまたは Tektronix フォーマットのオブジェクト・ファイルに変換するプログラム

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

オプション・ヘッダ： COFF オブジェクト・ファイルの一部分。リンカが、ダウンロード時に再配置を行うために使用します。

オペランド： 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またはパラメータ

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

き

記憶クラス： シンボル・テーブルのエントリ。シンボルへのアクセス方法を示します。

疑似命令： 特別な目的をもつ複数のコマンド。ソフトウェア・ツールの動作や機能を制御します（デバイスの動作を制御する「アセンブリ言語命令」と対比）。

共通オブジェクト・ファイル・フォーマット： COFF を参照してください。

行番号エントリ： COFF 出力モジュールにあるエントリの 1 つ。アセンブリ・コードの各行を、これを作成した元の C ソース・ファイルに逆にマップします。

共用体： 型とサイズが異なるオブジェクトを保持できる変数

く

グローバル： 次のいずれかの条件を満たすシンボルの一種です。1）カレント・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）カレント・モジュールでアクセスされているが、他のモジュールで定義されている。

クロスリファレンス・リスト： アセンブラにより作成される出力ファイル。定義されたシンボル、シンボルを定義した行、シンボルを参照する行、およびその最終値が表示されます。

こ

高水準言語デバッグ： シンボル情報および高水準言語情報（型や関数の定義など）を保持して、デバッグ・ツールがこれらの情報を使用できるようにするコンパイラの機能

構成メモリ： リンカが割り当てを行うために指定したメモリ

構造体： グループ化されて 1 つの名前を付けられた単数または複数の変数の集まり

コマンド・ファイル： リンカまたは Hex 変換ユーティリティのオプション、ファイル名、疑似命令、またはコメントが記述されているファイル

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及ぼしません。

さ

再配置： シンボルのアドレスが変更されるときに、リンカがそのシンボルに対するすべての参照を調整すること。

サブセクション： セクション内のさらに小さなセクション。メモリ・マップをセクションよりも細かく制御します。セクションも参照してください。

し

式： 定数、シンボル、または算術演算子により区切られた定数とシンボル

実行アドレス： セクションが実行されるアドレス

実行可能モジュール： TMS320C55x 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

自動初期化： プログラムの実行の前に、C のグローバル変数（.cinit セクションに保持されている）を初期化すること

シミュレータ： TMS320C55x の機能をシミュレートするためのソフトウェア開発システム

出力セクション： リンクされた実行可能モジュールの中の、最終的な割り当て済みセクション

出力モジュール： ターゲット・システム上にダウンロードして実行できる、リンクされた実行可能オブジェクト・ファイル

条件付き処理： ソース・コードの 1 ブロック、またはソース・コードの代替ブロックを、指定された式の計算に基づいて処理する方法

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 疑似命令から構成されます。

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 疑似命令と .usect 疑似命令から構成されます。

シンボリック・デバッグ： シ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

せ

整合定義式： シンボルまたはアセンブル時定数が式に表示される前に、すでに定義されているシンボルまたはアセンブル時定数のみを指定した式

静的な実行： 通常の見出しと進捗情報の出力を抑止すること

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 TMS320C55x メモリ・マップ内に連続した空間を占めます。

セクション・プログラム・カウンタ： *SPC* を参照してください

セクション・ヘッダ： ファイルのセクションに関する情報を含んだ COFF オブジェクト・ファイルの位置部分。各セクションには専用のヘッダがあります。ヘッダは、そのセクションの開始アドレスを示し、サイズなどの情報を含んでいます。

絶対アドレス： TMS320C55x のメモリ位置に永続的に割り当てられているアドレス

絶対リスタ： リンク済みファイルを入力として受け入れ、出力として .abs ファイルを作成するデバッグ・ツール。.abs ファイルをアセンブルすることにより、オブジェクト・コードの絶対アドレスを示すリスト出力を作成することができます。絶対リスタを使用しないと、絶対リストの作成作業は手動操作が繁雑になります。

そ

ソース・ファイル： C コードまたはアセンブリ言語コードを書き込んだファイル。コードをコンパイルまたはアセンブルすることによってオブジェクト・ファイルを作成します。

た

ターゲット・メモリ： 実行可能なオブジェクト・コードをロードできる、TMS320C55x システムの物理メモリ

代入文： 変数に値を代入する文

タグ： 構造体、共用体、または列挙型に割り当てることのできる任意の型名

て

定数： オペランドとして使用できる数値

な

名前付きセクション： .sect 疑似命令を使って定義される初期化されたセクション

生データ： 出力セクションにおける実行可能なコードまたは初期化されたデータ

に

ニーモニック： アセンブラによって機械コードに変換される命令の名前

入力セクション： オブジェクト・ファイルのセクションの 1 つ。このセクションは、実行可能モジュールにリンクされます。

は

バインディング： 出力セクションまたはシンボルに異なるアドレスを指定する際の処理

ふ

ファイル・ヘッダ： COFF オブジェクト・ファイルの一部分。オブジェクト・ファイルに関する一般情報（セクション・ヘッダの数、オブジェクト・ファイルをダウンロードできるシステムの種類、シンボル・テーブルにあるシンボルの数、テーブルの開始アドレスなど）が含まれます。

フィールド： TMS320C55x においてソフトウェアにより設定可能なデータ型。1 ～ 16 ビットの任意の範囲で長さをプログラミングできます。

符号拡張： 値を構成する未使用の MSB を、その値の符号ビットで埋めること

部分リンク： 再度リンクされるファイルのリンク

ブロック： 中括弧でまとめられた一連の宣言または文

ほ

ホール: 出力セクションを構成する入力セクション間にある領域。実際のコードやデータは含みません。

補足エントリ： シンボル・テーブルにあるシンボルの追加エントリ。そのシンボルに関する追加情報（それがファイル名か、セクション名、関数名か、など）が含まれます。

ま

マクロ： 命令として使用できるユーザー定義ルーチン

マクロ・コール： マクロを起動すること

マクロ定義： マクロを構成する名前とコードを定義するソース文のブロック

マクロ展開： マクロ・コールに置換され、続いてアセンブルされるソース文

マクロ・ライブラリ： マクロで構成されたアーカイブ・ライブラリ。ライブラリの各ファイルには、最低 1 つのマクロが含まれていなければなりません。また、ファイル名はファイルに定義されているマクロ名と同じで、拡張子は .asm でなければなりません。

マジック・ナンバ： オブジェクト・ファイルを TMS320C55x によって実行できるモジュールとして識別する COFF ファイル・ヘッダ・エントリ

マップ・ファイル： リンカを使用して作成される出力ファイル。メモリ構成、セクション構成、セクションの割り当て、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。

み

未構成メモリ： メモリ・マップの一部として定義されていないメモリ。コードまたはデータとともにロードすることはできません。

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。

メンバ： 構造体、共用体、アーカイブ、または列挙型の要素あるいは変数

も

文字列テーブル： 8 文字以上のシンボル名を格納するテーブル（8 文字以上のシンボル名はシンボル・テーブルに格納できないので、文字列テーブルが使用されます)。シンボルのエントリ・ポイントの名前の部分は、文字列テーブルの文字列の位置を指します。

モデル文： マクロ定義の中の命令またはアセンブラ疑似命令。マクロが起動されるたびにアセンブルされます。

ら

ラベル： ソース文の 1 カラム目で始まるシンボル。その文のアドレスに対応します。

り

リスト・ファイル： アセンブラの作成する出力ファイル。ソース文、その行番号、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・モジュールを形成するソフトウェア・ツール。生成されたモジュールは TMS320C55x 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 TMS320C55x システム・メモリにロードするデバイス

わ

ワード： 、ターゲット・メモリ内の 16 ビットのアドレッシングが可能な位置

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こと

索引

記号

数

A

B

C

D

E

F

G

H

N

O

P

Q

R

S

T

U

V

W

X

Z

あ

い

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

て

と

な

に

ね

は

ひ

ふ

へ

ほ

ま

み

め

も

ゆ

よ

ら

り

れ

ろ

わ

日本テキサス・インスツルメンツ株式会社

本　　社　〒160-8366　東京都新宿区西新宿６丁目２４番１号　西新宿三井ビルディング３階　☎03（4331）2000（番号案内）

西日本ビジネスセンター　〒530-6026　大阪市北区天満橋１丁目８番３０号　OAPオフィスタワー26階　☎06（6356）4500（代　　表）
工　　場　大分県・日出町／茨城県・美浦村／静岡県・小山町（センサーズ＆コントロールズ事業部）
研究開発センター　茨城県・つくば市（筑波テクノロジー・センター）／神奈川県・厚木市（厚木テクノロジー・センター）

■お問い合わせ先
プロダクト・インフォメーション・センター（PIC）　FAX 0120-81-0036
URL: http: //www.tij.co.jp/pic/

TMS320C55x
アセンブリ言語ツール
ユーザーズ・マニュアル

第 1 版 2005 年 9 月

発行所 **日本テキサス・インスツルメンツ株式会社**
〒160-8366
東京都新宿区西新宿 6-24-1（西新宿三井ビルディング）

Made in Japan
’05.09

日本テキサス・インスツルメンツ株式会社

SPRUE46