TEXAS INSTRUMENTS

TMS320C1x/C2x/C2xx/C5x
アセンブリ言語ツール

ユーザーズ・マニュアル

1996

Microprocessor Development Systems

SPRU018D 翻訳版

JAJU002 WAS SCJ2223

TMS320C1x/C2x/C2xx/C5x
アセンブリ言語ツール
ユーザーズ・マニュアル

1996年11月

TEXAS INSTRUMENTS

ご 注 意

日本テキサス・インスツルメンツ株式会社（以下TIといいます）は、通知をすることなくその製品を変更し、もしくは半導体集積回路製品またはサービスの製造または提供を中止することがありますので、お客様は、発注される前に、これから参照しようとする資料が最新のものであることを確実にするため、最新版の資料を取得するようお勧めします。

TIは、その半導体集積回路製品および関連するソフトウェアが、TIの標準保証条件に従い販売の際の現行の仕様書に対応した性能を有していることを保証します。検査およびその他の品質管理技法は、TIが当該保証を支援するのに必要とみなす範囲で行なわれております。各デバイスの全てのパラメーターに関する特定の検査は、政府がそれ等の実行を義務づけている場合を除き、必ずしも行なわれておりません。

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

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

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

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

TIは製品の使用用途に関する支援、お客様の製品の設計、ソフトウェアの性能、または特許侵害もしくはサービスに対する責任を負うものではありません。またTIは、その半導体集積回路製品もしくはサービスが使用されうる、もしくは使用されている組み合せ、機械装置、もしくは方法をカバーしている、またはそれ等に関連している特許権、著作権、回路配置利用権、その他の知的財産権に基づいて何らかのライセンスを許諾するということは明示的にも黙示的にも保証も表示もしておりません。

弊社半導体製品の取り扱い・保管について

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

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

以上

まえがき

最初にお読み下さい

本書について

本書は、以下のアセンブリ言語ツールの使用方法について説明したものです。

- ❑ 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

本書を読む前に、TMS320C1x/C2x/C2xx/C5x コード生成ツール Getting Started で上記のアセンブリ言語ツールをインストールしておいて下さい。

本書のご利用方法

本書の目的は、特に TMS320 固定小数点 DSP 用に作られた TI のアセンブリ言語ツールの使用方法の習得に役立つ情報を提供することです。本書の構成は次のようになっています。

- ❑ **第１部「概要」**では、アセンブリ言語開発ツールの概要を説明します。また、TMS320C1x/C2x/C2xx/C5x のツールを一層効果的に使用するための共通オブジェクト・ファイル・フォーマット（COFF）についても説明します。アセンブラおよびリンカの使用にあたっては第 2 章をお読み下さい。
- ❑ **第２部「アセンブラの説明」**では、アセンブラの使用方法に関して詳しく説明します。具体的には、アセンブラの呼び出し方法、ソース文フォーマット、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。また、TMS320C1x、TMS320C2x、TMS320C2xx、および TMS320C5x 命令セットをアルファベット順にその概要を示し、マクロ要素についても説明します。
- ❑ **第３部「その他のアセンブリ言語ツール」**では、アセンブリ言語ソース・ファイルを作成するための上記のツールのうち、アセンブラ以外の各ツールについて詳しく説明します。たとえば、第 8 章では、リンカの呼び出し方法、リンカの機能、リンカ疑似命令の使用方法について説明します。第11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。
- ❑ **第４部「参考資料」**では、補足的な説明をします。ここでは、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ついての技術的な説明をします。TMS320C2x/C2xx/C5x C コンパイラで使われるシンボリック・デバッグ疑似命令についても説明します。最後に、リンカ・コマンド・ファイルの例、アセンブラおよびリンカのエラー・メッセージ、ならびに用語集を記載します。

表記規則

本書では、以下の表記規則を使用しています。

- プログラム・リスト、プログラム例、および対話表示は、特殊なフォントで示してあります。例は、強調のため、特殊なフォントにボールドをかけて示してあります。プログラム・リストの例を以下に示します。

```
11 0005   0001          .field    1, 2
12 0005   0003          .field    3, 4
13 0005   0006          .field    6, 3
14 0006                 .even
```

- 構文の記述、命令、コマンド、疑似命令はボールド (**bold face** font)、パラメータはイタリック体 (*italics*) で示します。構文でボールドの部分は、その表記通り入力する箇所です。構文でイタリック体の部分は、入力する情報の種類を示しています。コマンド行で入力する構文は、縁取りのある枠囲みの中心に示します。テキスト・ファイルで使用する構文は、縁取りのない枠囲みに左詰めで示します。以下に、コマンド行での構文例を示します。

dspabs *filename*

dspabs はコマンドです。このコマンドは、絶対リスタを呼び出し、*filename* により示されるパラメーターを 1 つとります。絶対リスタを呼び出すときには、絶対リスタが入力として使うファイルの名前を指定します。

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

dsphex [-*options*] *filename*

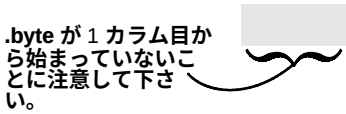
dsphex コマンドには 2 つのパタメータが使われています。最初のパラメータ -*options* は任意です。*option* が複数形で書かれていますから、複数のオプションを指定できます。2 つ目のパラメータ *filename* は必須です。

- アセンブラ構文の説明文では、1 カラム目はラベルまたはシンボルの先頭文字のために予約されています。ラベルまたはシンボルが**任意**の場合には、通常は表記されません。**必須の**パラメータである場合は、以下の例に示すように陰影の付いた枠囲みの中に左詰めで記載されます。シンボルまたはラベル以外の命令、コマンド、疑似命令、およびパラメータは 1 カラム目から始めることはできません。

symbol **.usect** **"***section name***",** *size in bytes*

この *symbol* は、.usect **疑似命令では必須**なので、**１カラム目**から始めなければなりません。*section name* は必ず二重引用符で囲み、セクションのサイズ (バイト) を示す *size in bytes* と *section name* の間はコンマを使って区切らなければなりません。

❑ 疑似命令が使えるパラメータの数は、疑似命令によって異なります。たとえば、.byte 疑似命令は 100 個までパラメータを使うことができます。この疑似命令の構文は次のように示されます。



.byte *value*$_1$ [, ... , *value*$_n$]

この構文は、.byte には少なくとも 1 つの値パラメータが必要であることを示していますが、コンマで区切ってさらに多くの値パラメータを指定することができます。

❑ 中括弧（{および}）は複数の項目を列挙する場合に使用します。|（「または」と読みます）は、列挙された項目を区切るために使われます。以下に例を示します。

```
{ * | *+ | *- }
```

これは *、*+、または *- の 3 つの選択項目があることを示しています。

リストが大括弧で囲まれていなければ、リストの中の 1 項目を選択しなければなりません。

当社発行の関連文献

次に示すのは、TMS320C1x、'C2x、'C2xx、および'C5xデバイスと関連サポート・ツールについて解説している文献です。当社発行のこれらの資料を希望される方は、タイトルと文献番号を添えて、プロダクト・インフォメーション・センター (PIC) までご連絡下さい。

TMS320C1x User's Guide（文献番号 SPRU013）は、CMOS 固定小数点デジタル・シグナル・プロセッサのハードウェアについての解説書で、ピン配置、アーキテクチャ、命令セット、ソフトウェア・アプリケーション、ハードウェア・アプリケーションについて解説しています。また、'C1x DSP 用のアナログ・インターフェイス・ペリフェラルおよびアプリケーションについて説明した項と、すべての 'C1x デバイスについての電気的仕様とパッケージ情報を示す総合データ・シートも含まれています。

TMS320C2x User's Guide（文献番号 SPRU014）は、'C2x 固定小数点デジタル・シグナル・プロセッサのハードウェアについての解説書で、ピン配置、アーキテクチャ、命令セット、ソフトウェア・アプリケーション、ハードウェア・アプリケーションについて解説しています。また、すべての'C2x デバイスについての電気的仕様とパッケージのメカニカル・データも含まれています。'C1x から 'C2X への DSP システムの移行についても解説しています。

TMS320C2xx User's Guide（文献番号 SPRU127）は、'C2xx 固定小数点デジタル・シグナル・プロセッサのハードウェアについての解説書で、ピン配置、アーキテクチャ、命令セット、ソフトウェア・アプリケーション、ハードウェア・アプリケーションについて解説しています。また、すべての 'C2xx デバイスについての電気的仕様とパッケージのメカニカル・データも含まれています。'C2x と 'C2xx との命令の比較についても解説しています。

TMS320C5x ユーザーズ・マニュアル（文献番号 SCJ1864）は、TMS320C5x 16 ビット固定小数点汎用デジタル・シグナル・プロセッサについての解説書で、このプロセッサのアーキテクチャ、内部レジスタ構造、命令セット、パイプライン、仕様、DMA、I/O ポートについて解説しています。ソフトウェア・アプリケーションについては独立した章で取り扱っています。

TMS320C2x/C2xx/C5xオプティマイジングCコンパイラ・ユーザーズ・マニュアル（文献番号 SCJ2222）は、'C2x/'C2xx/'C5x C コンパイラについて解説しています。本 C コンパイラは、ANSI 標準の C ソース・コードを受け入れ、'C2x、'C2xx、および 'C5x 世代のデバイス用の TMS320 アセンブリ言語ソース・コードを作成します。

TMS320 Family Development Support Reference Guide（文献番号 SPRU011）は、'320 ファミリーのデジタル・シグナル・プロセッサについての解説書で、この製品ラインをサポートする各種の製品について解説しています。対象には、コード作成ツール（コンパイラ、アセンブラ、リンカなど）、システム統合およびデバッグ・ツール（シミュレータ、エミュレータ、評価モジュールなど）が含まれています。また、皆様にご利用いただける文献、セミナ、大学の講義プログラム、工場の修理や交換についての情報も含まれています。

TMS320 Third - Party Support Reference Guide（文献番号 SPRU052）は、'320 デジタル・シグナル・プロセッサ・ファミリー向けに各種製品を発売している 100 社を越えるサード・パーティをアルファベット順に掲載しています。ソフトウェアおよびハードウェア開発ツール、音声認識、イメージ処理、ノイズ・キャンセル、モデムなど、無数の製品およびアプリケーションが提供されています。

Digital Signal Processing Applications with the TMS320 Family（文献番号 SPRA012（第 1 巻）、SPRA016（第 2 巻）、SPRA017（第 3 巻））。第 1 巻と第 2 巻は、'C10 および 'C20 ファミリーの固定小数点プロセッサを使用するアプリケーションについて解説しています。第 3 巻は、固定小数点プロセッサと同様に 'C30 浮動小数点プロセッサも使用するアプリケーションについて解説しています。

TMS320C2x C Source Debugger User's Guide（文献番号 SPRU070）は、TMS320C2x シミュレータの C ソース・デバッガ・インターフェイスを呼び出す方法について解説しています。デバッガ・インターフェイスについて、ウィンドウ管理、コマンド入力、コード実行、データ管理、ブレークポイントなど、さまざまな観点から説明していて、デバッガの基本機能を紹介するチュートリアルも含まれています。

TMS320C5x C Source Debugger User's Guide（文献番号 SPRU055）は、'C5x エミュレータ、EVM、シミュレータ、それぞれの C ソース・デバッガ・インターフェイスを呼び出す方法について解説しています。デバッガ・インターフェイスについて、ウィンドウ管理、コマンド入力、コード実行、データ管理、ブレークポイントなど、さまざまな観点から説明していて、デバッガの基本機能を紹介するチュートリアルも含まれています。

商標

IBM、PC、PC - DOS、および OS/2 は、International Business Machines Corp. の商標です。

MS、MS - DOS、および MS - Windows は、Microsoft Corp. の登録商標です。

SPARC は、SPARC International, Inc. の商標です。

SunView、Sun Windows、および Sun Workstation は、Sun Microsystems, Inc. の商標です。

UNIX は、Unix System Laboratories, Inc. の登録商標です。

XDS は、Texas Instruments Incorporated の商標です。

サポートが必要な場合・・・

必要なサポート	連絡先
TI製品についてのお問い合わせおよび資料の請求	プロダクト・インフォメーション・センター (PIC) へご連絡ください。 **TEL:** 0120 - 81 - 0026 **FAX :** 0120 - 81 - 0036 **E - mail:** pic - japan@ti.com
TI 社製DSP製品の最新情報を入手したい	お手持ちのホームページ閲覧ソフトにて以下のアドレスからアクセスできます。 http://www - a.tij.co.jp/DSPForum/Default.asp
サンプル・プログラムおよび関連ソフト・ウェアのダウンロード	anonymous ftp を使用して、**ftp.ti.com** (192.94.94.33) の /pub/mirrors にアクセスしてください。

目次

図

表

例

注意

第 1 章

はじめに

TMS320C1x/C2x/C2xx/C5x 固定小数点 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・リスタ
- ❑ Hex 変換ユーティリティ

この章では、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、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が、このアセンブリ言語ツールにはコンパイラとデバッガは付属していません。コンパイラとデバッガ、および TMS320C1x/C2x/C2xx/C5x デバイスについての詳しい説明は、VI ページに掲載されている「当社発行の関連文献」を参照して下さい。

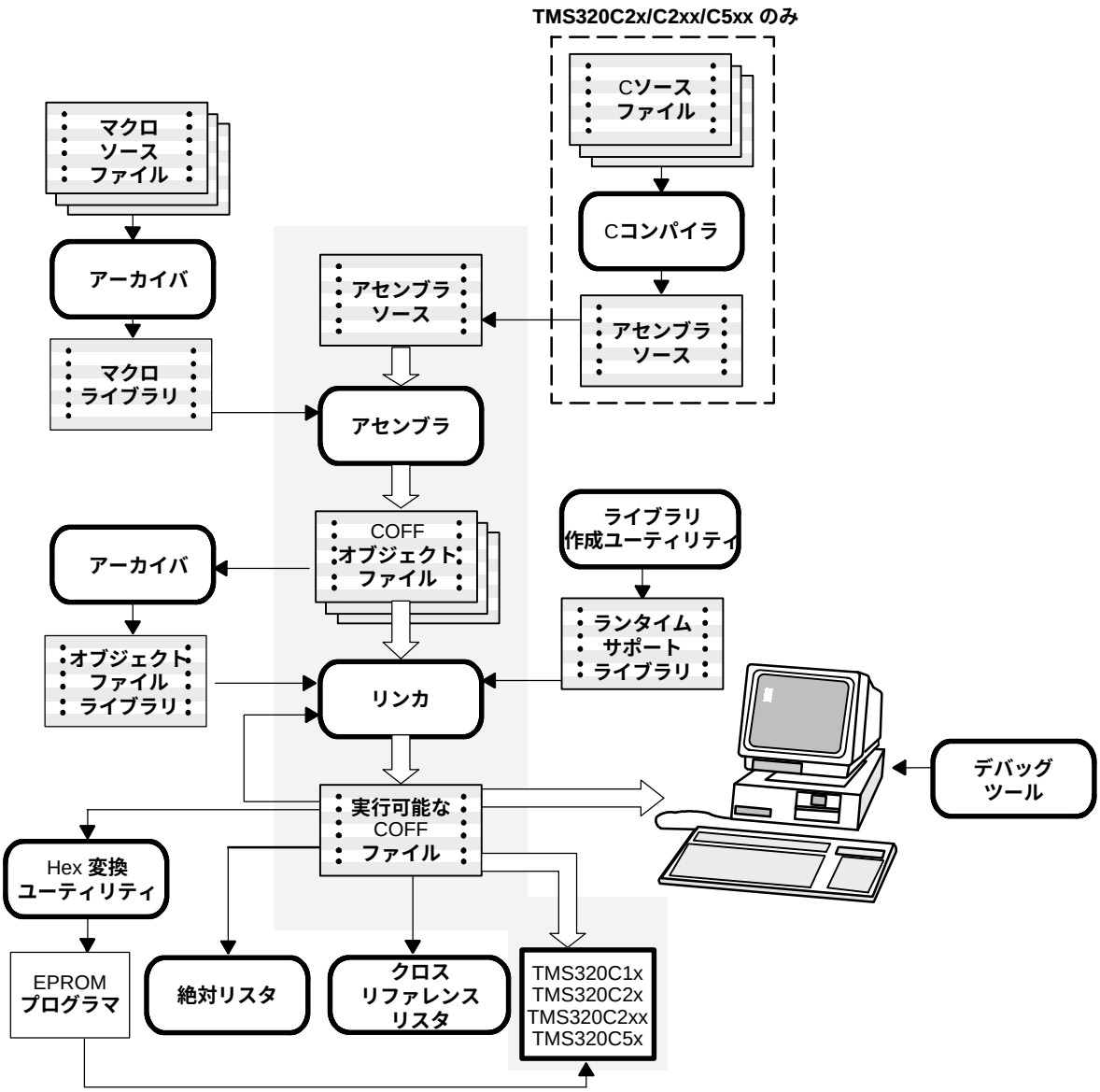
アセンブリ言語ツールでは、モジュラ・プログラミングに便利なように共通オブジェクト・ファイル・フォーマット (COFF) を使ったオブジェクト・ファイルを作成し、使用します。オブジェクト・ファイルには、それぞれが独立したコードおよびデータのブロック（セクションと呼ばれる）が含まれており、これを TMS320 メモリ空間にロードすることができます。COFF に関する基礎的な知識があると、TMS320 デバイスのプログラミングの能率が上がります。第 2 章「共通オブジェクト・ファイル・フォーマットの概要」においてオブジェクト・フォーマットについて詳しく説明します。

項目 **ページ**

1.1 ツールの概要と開発フロー

図 1 - 1 にアセンブリ言語でのソフトウェア開発の流れを示します。陰影個所は、最も基本的な開発手順を示しています。それ以外の部分は、必要に応じて行われます。

図 1 - 1. TMS320C1x/C2x/C2xx/C5x アセンブリ言語でのソフトウェア開発フロー



1.2 ツールの説明

- **TMS320C2x/C2xx/C5x C コンパイラ**は、C ソース・コードをTMS320C2x/C2xx/C5x のアセンブリ言語ソース・コードに変換します。C コンパイラは、アセンブリ言語ツール・パッケージに含まれていません。

- **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COFFオブジェクト・ファイルに変換します。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うと、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することができます。

- **アーカイバ**を使うと、複数のファイルを集めて 1 つのアーカイブ・ファイルを作成することができます。たとえば、いくつかのマクロを集めてマクロ・ライブラリを作ることができます。アセンブラは、ライブラリの中を検索して、ソース・ファイル内でマクロとして呼び出されているメンバを使います。同様にアーカイバを使って、複数のオブジェクト・ファイルを 1 つのオブジェクト・ライブラリにまとめることもできます。リンカは、リンク中に外部参照を解決するライブラリ内のメンバを取り込みます。

- **リンカ**は、複数のオブジェクト・ファイルを結合して 1 つの実行可能なオブジェクト・モジュールを作成します。実行可能なモジュールを作成する際に、再配置を実施し、外部参照を解決します。リンカへは、再配置可能な COFF オブジェクト・ファイル（アセンブラで作成されたもの）を入力することができます。リンカは、アーカイバ・ライブラリ・メンバや前のリンカ実行時に作成された出力モジュールも受け付けます。リンカ疑似命令を使うと、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いろいろな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することができます。

- 絶対リスタは、再アセンブルしてオブジェクト・ファイルの絶対アドレスのリストを作成するために使用できるファイルを生成します。

- クロス・リファレンス・リスタは、オブジェクト・ファイルを使用して、リンクされたソース・ファイルにおけるシンボル、その定義、その参照を示すクロス・リファレンス・リストを作成します。

- TMS320C1x/C2x/C2xx/C5x のデバッグ・ツールは、COFF ファイルを入力として受け入れますが、ほとんどの EPROM プログラマはこのファイルを受け入れません。**Hex** 変換ユーティリティを使うと、COFF オブジェクト・ファイルを TI-Tagged、Intel、Motorola、または Tektronix のオブジェクト・フォーマットに変換できます。

- 表 1-1 に、コードをダウンロードする前にコードを改善、修正するために各種システムで使用できるデバッグ・ツール示します。

表1 - 1. 各種システムの固定小数点デバッグ・サポート

デバッグ・ツール	'C1x	'C2x	'C2xx	'C5x
シミュレータ	PC	PC SPARC	PC SPARC	PC SPARC
評価モジュール	PC	PC SPARC	PC SPARC	PC SPARC
エミュレータ	PC	PC	PC SPARC	PC SPARC

注: PC は、MS - DOS または OS/2 のもとで実行中であることを示します。
SPARC は、UNIX のもとで実行中であることを示します。

第 2 章

共通オブジェクト・ファイル・フォーマットの概要

アセンブラとリンカを使って、TMS320C1x/C2x/C2xx/C5x 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、つまり COFF と呼びます。

COFF を使う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・プログラミングをより容易に行うことができるようになります。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。

この章では、COFF のセクションの概要を示し、次の項目についても解説します。

項目 ページ

COFF オブジェクト・ファイルの構造の詳しい情報については付録 A を参照してください。

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、独立していて全く別になっています。COFF オブジェクト・ファイルには、必ずデフォルトの 3 つ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、.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。

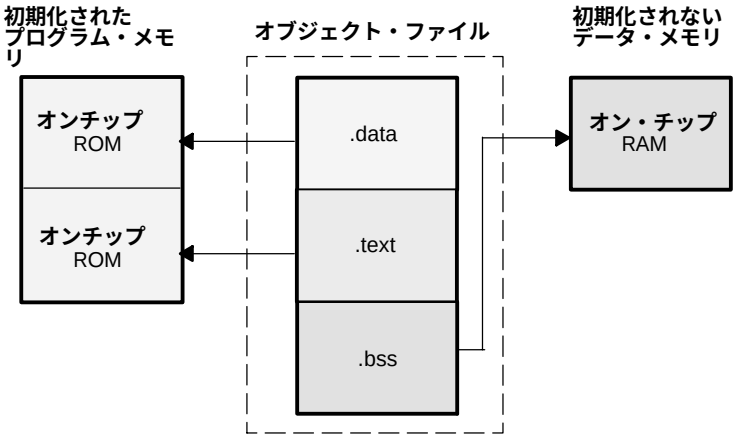
初期化されたセクション	データまたはコードを含みます。.text セクションと.data セクションは初期化されます。.sect アセンブラ疑似命令によって作成された名前付きセクションも初期化されます。
初期化されないセクション	メモリ・マップ内に初期化されないデータのための空間を確保します。.bss セクションは初期化されません。.usect アセンブラ疑似命令によって作成された名前付きセクションも初期化されません。

アセンブラでは、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ための疑似命令がいくつか用意されています。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構築し、図 2 - 1 に示されているオブジェクト・ファイルと同様の構造を持ったオブジェクト・ファイルを作成します。

リンカには、セクションをターゲット・メモリ・マップへ再配置する機能があります。これを**割り振り**と呼びます。ほとんどのシステムではいくつかの型のメモリが使われているので、セクションを使うとターゲット・メモリをより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、独立しているので再配置が可能です。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 EPROM を含む部分に割り振ることができます。

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取り扱い

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見きわめます。アセンブラには、この機能をサポートする 6 つの疑似命令があります。

- ❑ **.bss**
- ❑ **.usect**
- ❑ **.text**
- ❑ **.data**
- ❑ **.sect**
- ❑ **.asect**

.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。.text、.data、.sect および .asect の各疑似命令は**初期化されたセクション**を作成します。

> **注: デフォルトのセクション疑似命令**
>
> セクション疑似命令を全く使わないと、アセンブラはすべてを .text セクション内にアセンブルします。

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、TMS320 メモリに空間を確保します。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・ファイルに実際の中味を持っていません。プログラムは、実行時にこの空間を変数の作成と格納に使うことができます。

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って構築されます。.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。ユーザが .bss 疑似命令を呼び出すたびに、アセンブラはさらに多くの空間を.bssセクションに確保します。ユーザが .usect 疑似命令を呼び出すたびに、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。

各疑似命令は次の構文を使います。

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

symbol	.bss または .usect 疑似命令の実行によって確保される空間の最初のバイトを指します。*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って）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	これは絶対式です。.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 これは、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 ワードを連続して関連付けます。size が 1 ページより大きくない限り、割り振られる空間がページの境界にまたがることはありません。size が 1 ページより大きい場合は、そのオブジェクトはページの境界から始まるように割り振られます。

section name アセンブラに対し、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しい説明は、2.2.3 の項を参照してください。

.text、.data、.sect、.asect 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ル作業を中止し、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「逃れ」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うことができ、しかもその初期化されたセクションには何の影響も及ぼしません。

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、実行可能なコードか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、プログラムがロードされる時に TMS320 メモリに置かれます。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することができます。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。

4 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
.data
.sect     ”section name”
.asect    “section name” , address
```

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、.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。

セクションは、反復作業によって構築されます。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、.sect、.asect 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。

2.2.3 名前付きセクション

名前付きセクションは、ユーザが作成します。ユーザは名前付きセクションをデフォルトの .text、.data、.bss セクションと同様に使うことができますが、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振ることができます。ここに .text に割り振り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て、それを別々にメモリに割り振ることができます。.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。

3 種類の疑似命令を使っ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** 疑似命令および **.asect** 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 疑似命令を使うと、再配置可能なアドレスを持った名前付きセクションを作成できます。.asect 疑似命令を使うと、絶対アドレスを持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]
         .sect    "section name"
         .asect   "section name" , address
```

section name パラメータは、セクションの名前です。セクション名は 8 文字まで有効です。32,767 種類の名前付きセクションを作成することができます。

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既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 疑似命令を使ってセクションを作成し、同じ名前のセクションを .sect で使うことはできません。

.asect の address は、ターゲット・メモリ内でのセクションの絶対開始アドレスを識別します。このアドレスは、特定の絶対セクションに初めてアセンブルするときには、必須です。すでにコードを含んでいる絶対セクションへのアセンブルを続けるために .asect を使う場合は、address パラメータを使うことはできません。

2.2.4 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、適切な SPC をインクリメントします。ユーザがあるセクションへのアセンブルを再開するとアセンブラは該当する SPC の以前の値を取り出し、またそのSPCを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扱います。リンカは、メモリ・マップ内の最後のロケーションに従って各セクションを再配置します。

2.2.5 絶対セクション

.asect 疑似命令は、絶対アドレスによりアドレスを指定する名前付きセクションを定義します。絶対セクションは、オフチップ・メモリから高速のオンチップ・メモリにコードをロードする場合に便利です。

注： .asect 疑似命令はすでに使われていません。

現在では、リンカの SECTIONS 疑似命令を使ってすべてのセクションのロード・アドレスと実行アドレスを分けることができるので、.asect 疑似命令は使う必要がなくなりました。しかし、.asect 疑似命令は、アセンブラの以前のバージョンとの互換性を維持する目的で、現在でも以前と同様にその機能を果たします。詳しい説明については、8 - 24 ページの 8.7 節を参照して下さい。

2.2.6 セクション疑似命令の使用例

図 2 - 1 は、セクション疑似命令を使っ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。セクション疑似命令は、あるセクションへのアセンブルを始めて開始するため、またはすでにコードが含まれているセクションへのアセンブルを続行するために使用できます。続行の場合には、アセンブラは、そのセクション内の既存のコードに新しいコードをただ単に追加します。

例 2 - 1 例のフォーマットは、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. セクション疑似命令の使用法

```
     1
     2            ************************************************
     3            **  assemble an initialized table into .data  **
     4            ************************************************
     5 0000               .data
     6 0000 0011  coeff   .word   011h, 022h, 033h
       0001 0022
       0002 0033
     7            ************************************************
     8            ** reserve space in .bss for two variables    **
     9            ************************************************
    10 0000               .bss    var1, 1
    11 0001               .bss    buffer, 10
    12           ************************************************
    13           **   still in data                            **
    14           ************************************************
    15 0003 0123  ptr     .word   0123h
    16           ************************************************
    17           ** assemble code into .text section           **
    18           ************************************************
    19 0000               .text
    20 0000 200f  add:    LAC     0Fh
    21 0001 d003  aloop:  SBLK    1
       0002 0001
    22 0003 f280          BLEZ    aloop
       0004 0001'
    23 0005 6000-         SACL    var1, 0
    24           ************************************************
    25           **  assemble another initialized table into   **
    26           **  the .data section                         **
    27           ************************************************
    28 0004               .data
    29 0004 00aa  ivals   .word   0AAh, 0BBh
       0005 00bb
    30           ************************************************
    31           **  define the named section ‘‘newvars’’      **
    32           ************************************************
    33 0000       var2    .usect  ”newvars”, 1
    34 0001       inbuf   .usect  ”newvars”, 7
    35           ************************************************
    36           **  assemble more code into .text             **
    37           ************************************************
    38 0006               .text
    39 0006 ccff          ADD #0FFh
```

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 4 つのセクションが作成されます。

.text 7 ワードのオブジェクト・コードが入ります。

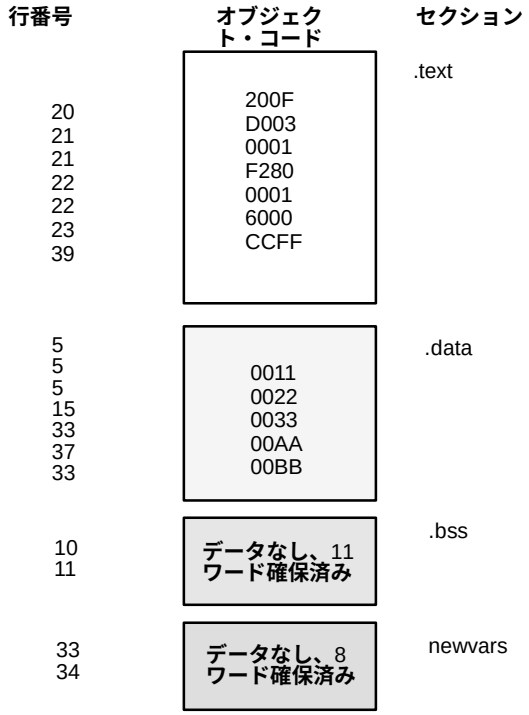
.data 5 ワードのオブジェクト・コードが入ります。

.bss 11 ワードをメモリに確保します。

newvars .usect 疑似命令を使って作成さ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 のリストで生成されたオブジェクト・コード



2.3 リンカによるセクションの取り扱い

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**を使うとターゲット・システムのメモリ・マップを定義することができます。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、リンカに対して入力セクションの結合方法と出力セクションのメモリ内の置き場所を指定します。

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と、リンカは 2.3.1 項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割り振り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についての詳しい説明は、以下の各節を参照して下さい。

リンカはプログラム・メモリを PAGE 0 として参照し、データ・メモリを PAGE 1 として参照する点に注意して下さい。これらのページは、TMS320C1x/C2x/C2xx/C5x デバイスのデータ・ページ・メモリ・フォーマットとは別のもので、これと混同してはなりません。

注： この節では TMS320C25 の場合の例を示します

セクションの割り振りは、ターゲット・メモリの構成に基づいて行われます。この節に示す例では、TMS320C25 をターゲット・プロセッサとして使用して、この概念を説明します。各プロセッサには、独自のプログラム可能なメモリ構成があります。

2.3.1 デフォルト・メモリ割り振り

MEMORY 疑似命令や SECTIONS 疑似命令を指定しなくてもファイルをリンクすることができます。リンカはデフォルト・モデルを使ってセクションを結合し（その必要があれば）、それをメモリに割り振ります。デフォルト・モデルを使う場合には、リンカは次のように動作します。

1) オブジェクト・ファイルを読み取って、正しいターゲット・プロセッサを判別します。

2) ターゲット・プロセッサが TMS320C25 だとすれば、次のメモリ・ロケーションが使用可能です。

- ❑ プログラム・メモリ

外部ロケーション	1000h-0FEFFh

- ❑ データ・メモリ

オンチップ・ロケーション	300h-3FFh（オンチップ・ブロック B1）
外部ロケーション	400h-0FFFFh

3) .text をアドレス 1000h からプログラム・メモリに割り振ります。

4) .text に続けてすぐに .data をプログラム・メモリに割り振ります。

5) .data に続けてすぐに、すべての初期化された名前付きセクションをプログラム・メモリに割り振ります。名前付きセクションの割り振りは、リンカが入力ファイル内で検出した順序に従って行われます。

6) .bss をアドレス 300h からデータ・メモリに割り振ります。

7) .bss に続けてすぐに、すべての初期化されない名前付きセクションをデータ・メモリに割り振ります。名前付きセクションの割り振りは、リンカが入力ファイル内で検出した順序に従って行われます。

図 2 - 3 は、1 つのファイルが、TMS320C25 用のデフォルト割り振りを使ってどのようにメモリに割り振られるかを示しています。リンカは実際にはオブジェクト・コードをメモリに入れない点に注意して下さい。リンカはセクションへアドレスを割り振るだけで、実際にコードをメモリに入れるのはローダです。

TMS320C1x、TMS320C2x、TMS320C2xx、および TMS320C5x のデフォルト割り振りについての詳しい説明は、8 - 45 ページの 8.11 節を参照して下さい。

図 2 - 3.　図 2 - 2 におけるオブジェクト・コードのデフォルト割り振り

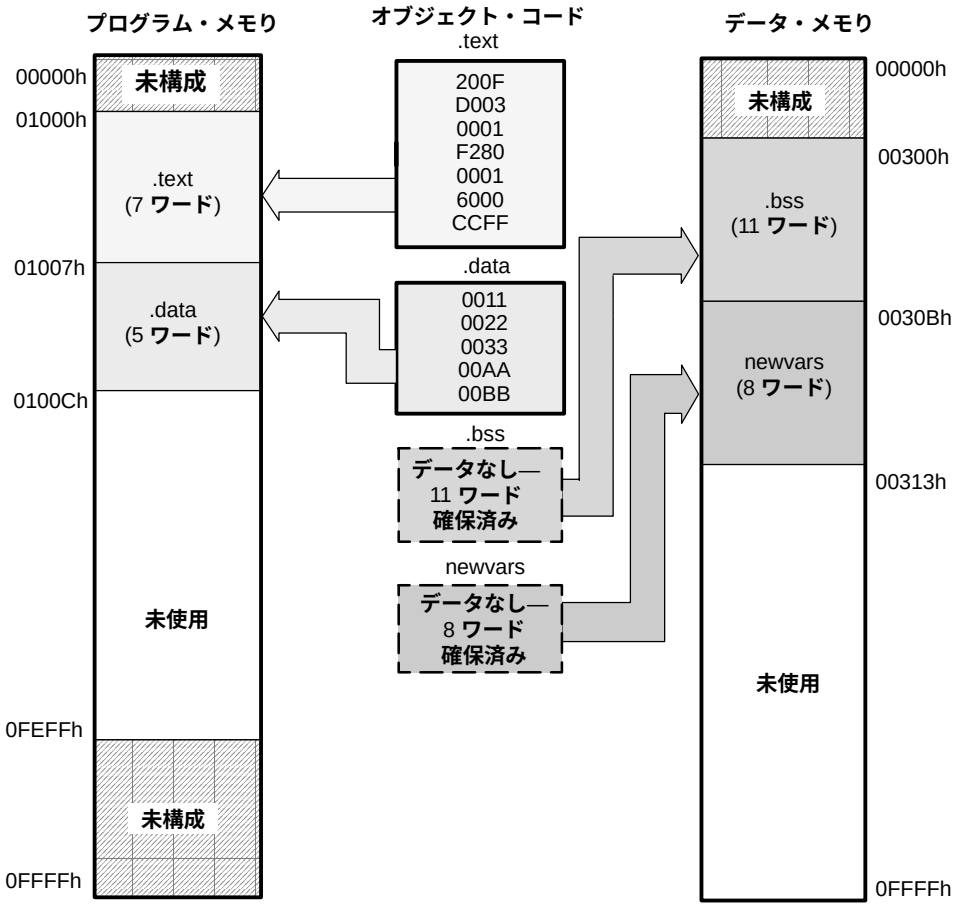


図 2 - 3 で示されているように、リンカは次のように動作します

1) 最初に .text セクションをプログラム・メモリのアドレス 1000h から始まる位置に割り振ります。.text セクションには 7 ワードのオブジェクト・コードが入ります。

2) 次に .data セクションを プログラム・メモリのアドレス 1007h から始まる位置に割り振ります。.data セクションには 5 ワードのオブジェクト・コードが入ります。

3) .bss セクションを データ・メモリのアドレス 300h から始まる位置に割り振ります。.bss セクションは 11 ワードをメモリに確保します。

4) データ・メモリのアドレス 30Bh に初期化されていない名前付きセクション newvars を割り振ります。newvars セクションは 8 ワードをメモリに確保します。

プログラム・メモリ内のアドレス範囲 0100Ch～0FFFFh、およびデータ・メモリ内の 0313h～0FFFFh の空間は使用されないので注意してください。

図 2 - 4 は、2 つのファイルをリンクする方法の簡単な例を示しています。 デフォルトのアルゴリズムを使っていくつかのファイルをリンクすると、リンカは同じ名前を持つ入力セクションすべてを結合して、その名前を持つ 1 つの出力セクションを作成します。たとえば、リンカは 2 つの入力ファイルからの .text セクションを結合して、1 つの .text 出力セクションを作成します。

図 2 - 4.　2 つのファイルからの入力セクションの結合（デフォルト割り振り）

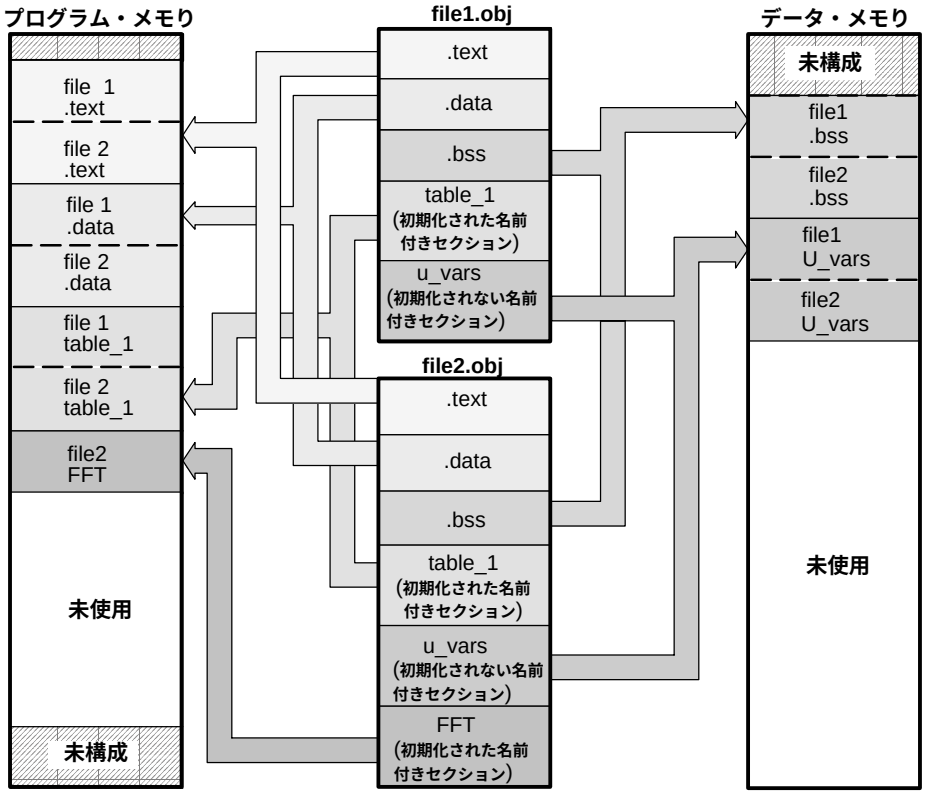


図 2 - 4 では、file1.obj と file2.obj のそれぞれに .text、.data、および .bss のデフォルト・セクション、Table_1 と呼ばれる初期化された名前付きセクション、および U_vars と呼ばれる初期化されない名前付きセクションがあります。file2.obj には、FFT と呼ばれる初期化された名前付きセクションも含まれています。図 2 - 4 で示されているように、リンカは次のように動作します。

1) file1 の .text と file2 の .text を結合して 1 つの .text 出力セクションを作成します。.text 出力セクションは、プログラム・メモリのアドレス 1000h に割り振られます。

2) file1 の .data と file2 の .data を結合して .data 出力セクションを作成します。.data 出力セクションは、.text 出力セクションに続いてプログラム・メモリに割り振られます。

3) file1 の table_1 を file2 の table_1 と結合して、table_1 出力セクションを作成します。table_1 セクションは、リンカが最初に検出する初期化された名前付きセクションであり、したがって、2 番目の初期化された名前付きセクションである FFT より前に割り振られます。table_1 出力セクションは、プログラム・メモリ内部の .data 出力セクションに続く位置に割り振られます。

4) file2 から取り出した FFT セクションを table_1 セクションの次に割り振ります。

5) file1 の .bss を file2 の .bss と結合して、.bss 出力セクションを作成します。.bss 出力セクションは、データ・メモリのアドレス 300h に割り振られます。

6) file1 の U_vars を file2 の U_vars と結合して、U_vars 出力セクションを作成します。U_vars 出力セクションは、データ・メモリの .bss 出力セクションに続く位置に割り振られます。

デフォルト割り振りのアルゴリズムについてのさらに詳しい説明は 、8-44 ページの 8.11 節を参照してください。

2.3.2 セクションのメモリ・マップへの配置

図 2 - 3 と図 2 - 4 は、リンカが複数のセクションを結合してメモリに割り振るデフォルトの方法を示しています。デフォルトの設定を使いたくない場合もあります。たとえば、すべての .data セクションを 1 つの .data 出力セクションには結合したくない場合です。また、.text セクションの代わりに名前付きセクションをプログラム・メモリのアドレス 1000h に配置したい場合もあります。多くのメモリ・マップは、様々な量の様々な型のメモリ（DRAM、ROM、EPROM、など）でできています。あるセクションを特定の型のメモリに置きたい場合もあります。

次の 2 つの図は、図 2 - 3 で使ったセクションの別の結合の可能性を示したものです。CNFD 命令で定義されたブロック B0 はデータ・メモリであり、マイクロコンピュータ・モード (MP/$\overline{\text{MC}}$=0) が選択されているものとします。

例 2 - 2 には、TMS320C25 の MEMORY と SECTIONS が定義されています。

図 2 - 5 は、例 2 - 2 で定義された範囲が TMS320C25 のメモリ・マップにどのように収まるのかを示しています。

図 2 - 6 は、図 2 - 3 からのセクションがどのようにメモリ・マップに割り振られるかを示しています。

他のモードの場合の TMS320C10、TMS320C25、および TMS320C50 のリンカ・コマンド・ファイルの例については、付録 C を参照してください。

例 2 - 2.　TMS320C25 用の MEMORY 疑似命令と SECTIONS 疑似命令

```
MEMORY
{                                          /* Program Memory */
   PAGE 0 : VECS:  origin =   0h,  length = 020h
            CODE:  origin = 020h,  length = 0F90h
                                           /* Data Memory    */
   PAGE 1 : RAMB2: origin = 060h,  length = 020h
            RAMB0: origin = 200h,  length = 100h
            RAMB1: origin = 300h,  length = 100h
}

SECTIONS
{
   vectors:              >  000000h
   .text:                >  CODE

   .data:                >  RAMB2
   .bss:                 >  RAMB0
   newvars:              >  RAMB1
}
```

❑ 例 2 - 2 の MEMORY 疑似命令は 5 つのメモリ範囲を定義しています。

VECS	CODE
RAMB2	RAMB0
RAMB1	

PAGE オプションは、範囲が占めるメモリの型を識別します。PAGE 0 はプログラム・メモリを識別し、PAGE 1 はデータ・メモリを識別します。それぞれの範囲の origin は、メモリ内でのその範囲の開始アドレスを識別します。length は、範囲の長さを指定します。

たとえば、PAGE 1 の RAMB0 というメモリ範囲は、開始アドレスが 200h で長さが 100h です。これは、データ・メモリ内のアドレス 0200h から 02FFh までを定義しています。

このMEMORY定義では、次の範囲は定義されていないことに注意してください。

- プログラム・メモリの 0FB0h ～ 0FFFFh
- データ・メモリの 0h ～ 05Fh
- データ・メモリの 080h ～ 01FFh
- データ・メモリの 0400h ～ 0FFFFh

これらの未定義の範囲は**未構成**です。リンカに関する限り、これらの領域は存在せず、これらの領域にはコードもデータもロードできません。MEMORY 疑似命令を使うときはいつも、コードまたはデータを入れることができるのは、この疑似命令で定義されているメモリ範囲だけです。

❑ 例 2 - 2 の SECTIONS 疑似命令は、セクションがメモリに割り振られる順序を定義します。vectors セクションは、必ずプログラム・メモリのアドレス 0 で始まらなければなりません。.text は、プログラム・メモリの CODE 範囲に割り振られます。.data セクションはデータ・メモリの RAMB2 範囲に割り振られ、.bss はデータ・メモリの RAMB0 に割り当てられ、newvars はデータ・メモリの RAMB1 に割り振られます。

図 2 - 5.　図 2 - 2 のメモリ・マップの定義

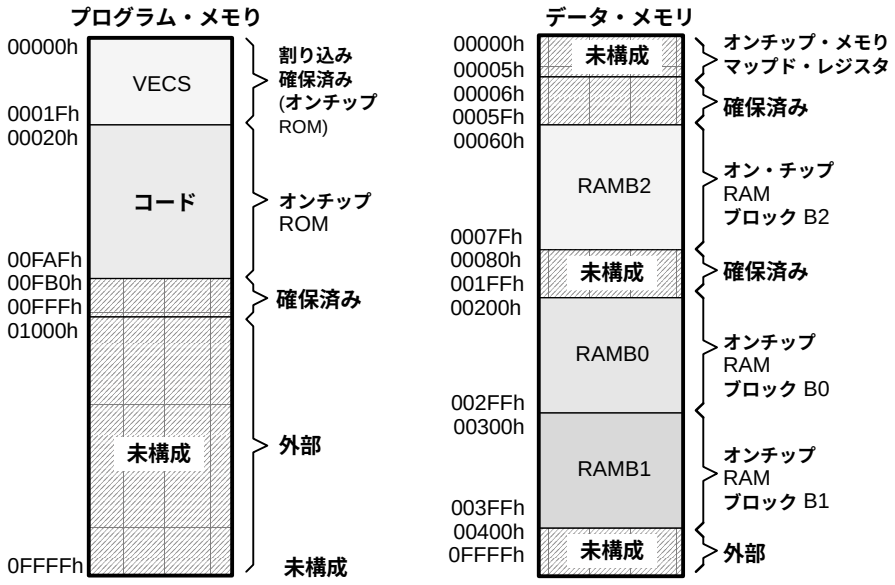


図 2 - 5 は、例 2 - 2 の MEMORY 疑似命令により定義される範囲がどのようにメモリ・マップに収まるのかを示しています。

- プログラム・メモリでは、VECS 範囲は、通常は割り込み用に使用されるアドレスを占めます。CODE 範囲は、オンチップ ROM 領域を占めます。確保済みおよび外部のメモリ・ロケーションは未構成です。
- データ・メモリでは、RAMB2 範囲は、オンチップ RAM のブロック B2 が使用するアドレスを占め、RAMB0 はブロック B0 を占め、RAMB1 はブロック B1 を占めます。オンチップ・メモリマップド・レジスタと、確保済みおよび外部のメモリ・ロケーションは未構成です。

図 2 - 6. リンカによるセクションの割り振り

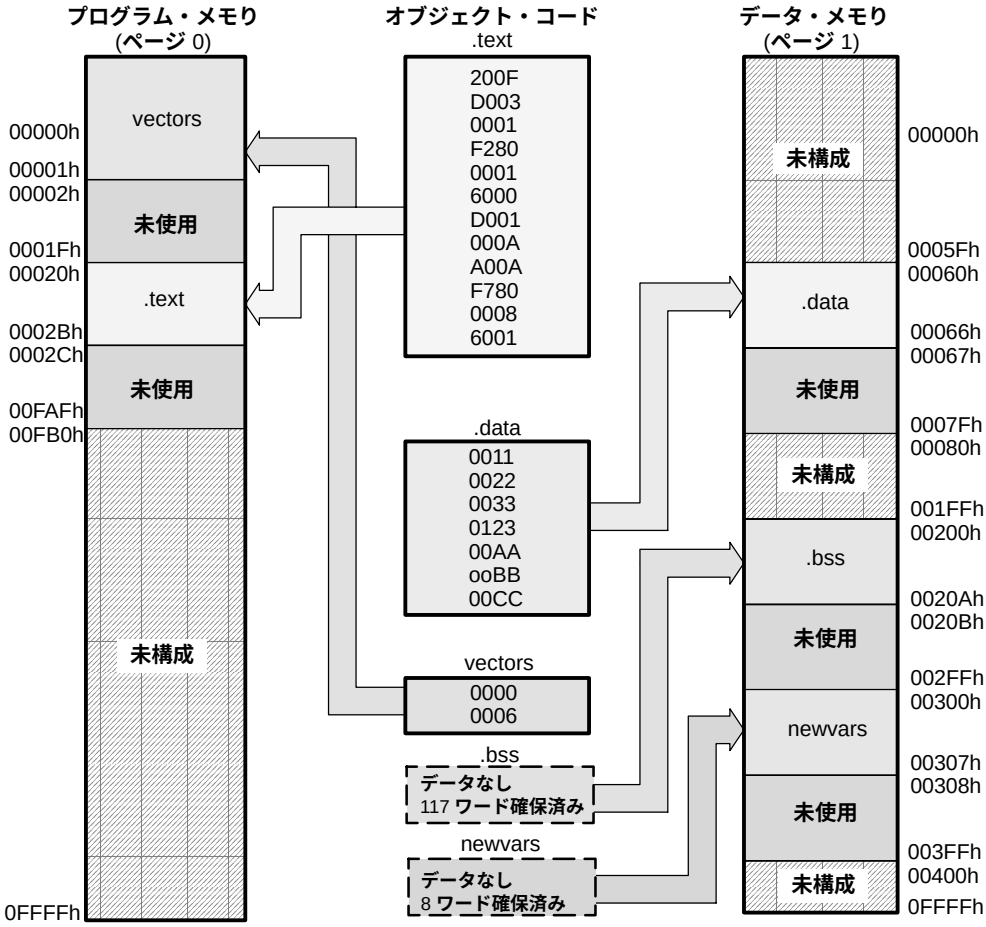


図 2 - 6 は、例 2 - 2 で定義されたメモリ範囲にセクションがどのように割り振られるかを示しています。メモリの一部は構成されているが使用されていないことに注意して下さい。たとえば、.text セクションは CODE 領域に割り振られています。CODE 範囲の長さは 0F90h ワードです。しかし、.text セクションには 12 ワードしか含まれていません。したがって、ロケーション 02Ch ～ 0FAFh は未使用です。

2.4 再配置

アセンブラは各セクションがアドレス 0 から始まるかのように扱います。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・マップに割り振って、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っ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調整します。例 2 - 3には、TMS320C25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 - 3. 再配置エントリを生成するコード

```
1                   .ref   X
2   0000            .text
3   0000  FF80       B      X  ; Generates a relocation entry
    0001  0000!
4   0002  D001       LALK   Y  ; Generates a relocation entry
    0003  0004'
5   0004  CE1F  Y:   IDLE
```

例 2 - 3では、X と Y の 2 つのシンボルはともに再配置可能です。Y は、このモジュールの .text セクションで定義されています。X は、どれか別のモジュールで定義されています。コードのアセンブルを行うと、X は値 0 を取り（アセンブラはすべての未定義の外部シンボルは値が 0 と見なします）、Y の値は 4 となります（.text セクションでアドレス 0 に相対的）。アセンブラは、X と Y それぞれのために 2 つの再配置エントリを生成します。X 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）。Y に対する参照は内部で定義された再配置可能なシンボル（リスト内の ' 文字によって示されます）に対するものです。

コードのリンク後は、X はアドレス 0x0100 に再配置されると考えて下さい。また、.text セクションは、アドレス 0x0200 で始まる位置に再配置されたものと考えて下さい。Y の再配置後の値は 0x0204 となります。 リンカは、2 つの再配置エントリを使ってオブジェクト・コード内の 2 つの参照にパッチを当てます。

```
FF80   B    X   は        FF80  となります
0000                      0100
D001   LALK Y   は        D001  となります
0004                      0204
```

COFF オブジェクト・ファイルにある各セクションには、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、通常は再配置エントリを使うと、それを除去します。そうすることにより、出力ファイルが再度再配置されることを防ぎます（再リンクされたり、ロードされたりした場合に）。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だから）。リンカに再配置エントリを引き続き保持させたい場合は、リンカを起動する時に -r オプションを使います。

2.5 実行時の再配置

時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性能がクリティカルな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 を使っ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 回割り振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・アドレスにはキーワード load を使用し、実行アドレスにはキーワード run 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判別します。そのセクションに対するすべての参照（ラベルなど）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、8-34 ページの例 8 - 6 を参照してください。

1 つのセクションに対して 1 回の割り振りしか行わない場合（load または run）は、そのセクションは 1 回だけ割り振られ、ロードも実行も同じアドレスで行われます。2 つの割り振りを指定した場合は、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かのように割り振られます。

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、意味があるアドレスは実行アドレスです。リンカは、初期化されないセクションを 1 回だけ割り振ります。ユーザが実行アドレスとロード・アドレスの両方を指定すると、リンカが警告を発行し、ロード・アドレスは無視されます。

実行時の再配置の詳しい説明は、8-32 ページの 8.8 節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、リンカ入力に使われるオブジェクト・ファイルと同じ COFFフォーマットが使われます。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。

プログラムを実行するためには、実行可能なオブジェクト・モジュール内のデータはターゲット・システム・メモリに転送 (**ロード**) する必要があります。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。以下によく使われる 3 つの方法を示します。

- ソフトウェア・シミュレータ、XDS エミュレータ、ソフトウェア開発システムなどの TMS320C1x/C2x/C2xx/C5x 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。ローダー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 dsphex）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。この後で、EPROM プログラマを使って変換済みファイルを EPROM に焼き付けることができます。

- TMS320C1x/C2x/C2xx/C5x プログラムの中には、ターゲット・システムで直接実行されているオペレーティング・システムまたはモニタ・ソフトウェアの制御によりロードされるものがあります。このような型のアプリケーションでは、通常、ターゲットシステムは、実行可能なモジュールを格納するファイルシステムへのインターフェイスを持つことになります。このようなシステムには、ユーザがカスタムのローダを作成する必要があります。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的な説明が付録Aにあります。ローダは、ターゲット・システムのメモリ構造（プログラムをメモリにロードするために）ばかりでなく、ファイル・システム、(ファイルをアクセスするために）も理解する必要があります。

2.7 COFF ファイルで使われるシンボル

COFF ファイルには、プログラムで使わ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、このテーブルを使っ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・テーブルを使います。

2.7.1 外部シンボル

1 つの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。

定義された (.def)	現行のモジュールで定義され、別のモジュールで使われる。
参照された (.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。
グローバル (.global)	上記のどちらかに該当する。

以下のコード・セグメントがこれらの定義を示しています。

```
x:      ADD       056h      ;  Define x
        MPY       y         ;  Reference y
        .global   x         ;  DEF of x
        .global   y         ;  REF of y
```

x の .global 定義には、それ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記述されています。y の .global 定義には、それ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記述されています。

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・テーブルに置きます。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 に対する未解釈の参照を解決します。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なければなりません。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、このシンボルを使ってセクション内の他のシンボルに対する参照を再配置します。

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。それは、リンカがそれを必要としないためです。たとえば、ラベルは .global で宣言されていない限り、シンボル・テーブルには含まれません。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、アセンブラを呼び出すときにオプション -s を使います。

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) で作成されますが、このフォーマットについては第 2 章および付録 A に説明があります。ソース・ファイルには、次のアセンブリ言語要素を含めることができます。

アセンブラ疑似命令	第 4 章で説明
アセンブリ言語命令	第 5 章で説明
マクロ疑似命令	第 6 章で説明

項目 ページ

3.1 アセンブラの概要

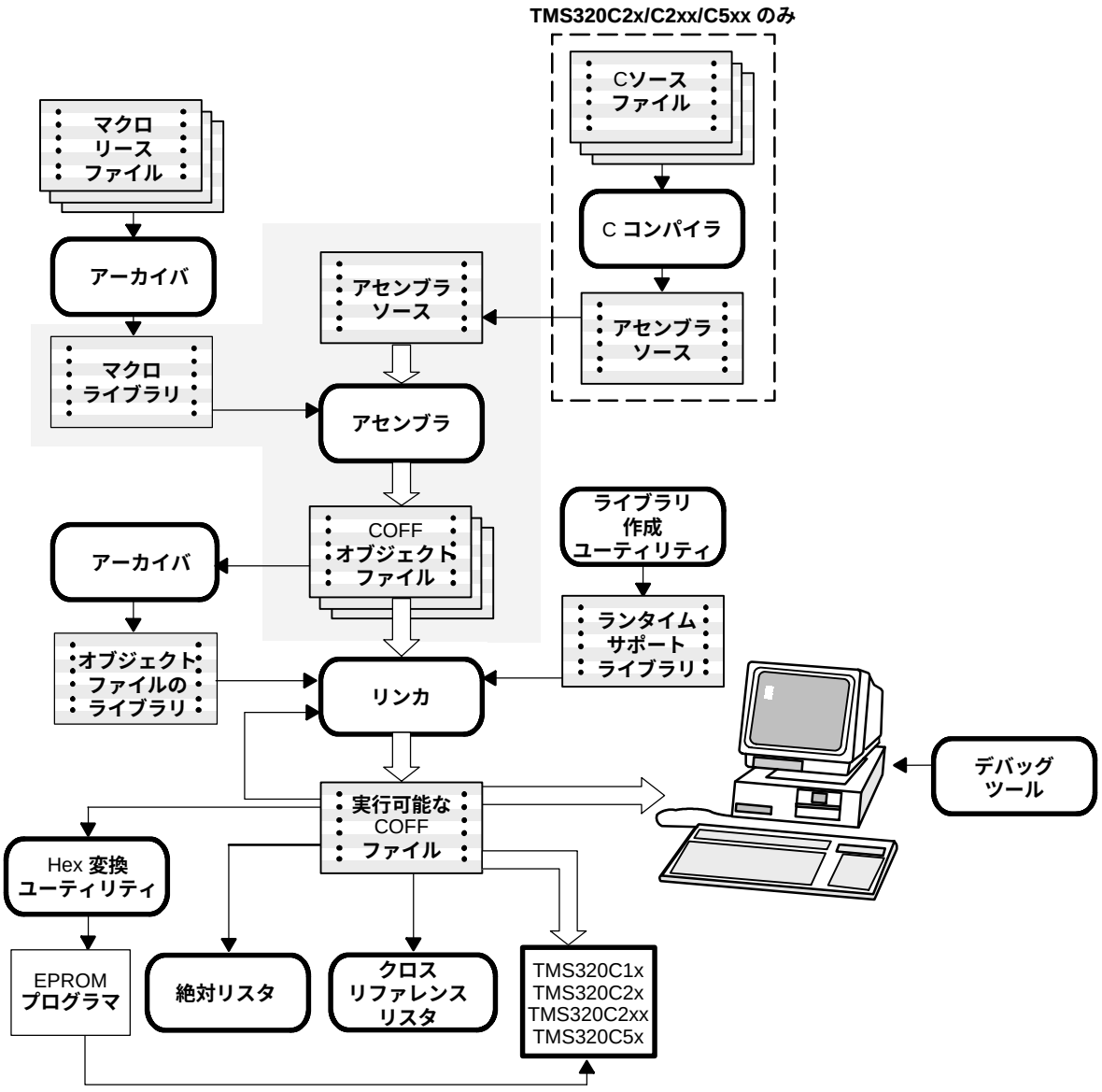
2 パス・アセンブラには次の機能があります。

- 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オブジェクト・ファイルを作成する。
- ソース・リストを作成し（必要に応じて）、ユーザ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）を維持する。
- グローバル・シンボルを定義し、参照し、（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。
- 条件付きブロックをアセンブルする。
- マクロをサポートし、ユーザがインラインで、あるいはライブラリでマクロを定義できるようにする。
- TMS320C1x、TMS320C2x、TMS320C2xx および TMS320C5x のコードのアセンブルを可能にする。

3.2 アセンブラ開発フロー

図 3 - 1 は、アセンブリ言語開発フローにおいてアセンブラの果たす役割を示したものです。アセンブラは、アセンブリ言語ソース・ファイルを入力として受け付けます。TMS320C1x/C2x/C2xx/C5x アセンブラは、TMS320C2x/C2xx/C5x C コンパイラの作成したアセンブリ言語ファイルも受け付けます。

図 3 - 1. アセンブラ開発フロー



3.3 アセンブラの呼び出し方法

アセンブラを呼び出すには、以下のように入力します。

dspa [*input file* [*object file* [*listing file*]]] [*-options*]

dspa アセンブラを呼び出すためのコマンドです。

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。何の拡張子も付けないと、アセンブラはデフォルトの拡張子である *.asm* を使います。入力ファイル名を入力しないと、アセンブラは入力ファイル名の入力を催促します。

object file アセンブラが作成するオブジェクト・ファイルの名前を指定します。拡張子を付けないと、アセンブラがデフォルトの *.obj* を付けます。オブジェクト・ファイルを指定しないと、アセンブラが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。リスト・ファイル用の名前を指定しないと、 -l (英字 L の小文字) オプションを使わない限り、アセンブラはリスト・ファイルを作成しません。この場合には、アセンブラは入力ファイル名を使います。拡張子を付けないと、アセンブラは *.lst* をデフォルトとして使います。

options 使用するアセンブラ・オプションを指定します。オプションは大文字と小文字の区別はありません。コマンドに続くコマンド行のどこに置いても構いません。個々のオプションの前にはハイフンを付けます。パラメータのない 1 文字のオプションは結合できます。たとえば、 *-lc* は *-l -c* と同じです。パラメータのあるオプション (**-i** など) は、個別に独立させて指定する必要があります。

-a 絶対リストを作成します。 -a を使うと、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せて使います。

-c アセンブリ言語ファイル内で大文字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。このオプションを使わないと、大文字小文字が区別されます (デフォルト) 。

-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 を省略すると、シンボルは 1 に設定されます。詳しい説明については、3-19 ページの 3.9.3 項を参照して下さい。

-i .copy、.include、または .mlib 疑似命令により指定されたファイルをアセンブラが検索するためのディレクトリを指定します。 -i オプションのフォーマットは -i*pathname* です。この方法で、最大 10 個のディレクトリを指定できます。個々のパス名の前に -i オプションを付ける必要があります。詳しい説明については、3-11 ページの 3.5.1 項を参照して下さい。

-l （「L」の小文字）リスト・ファイルを作成します。

-p TMS320C2x のコードを TMS320C5x または TMS320C2xx へ移植できるようにします（3-6 ページの 3.4.1 項を参照して下さい）。

- pp TMS320C2x のコードを TMS320C2xx へ移植できるようにし、.TMS32025 および .TMS3202xx シンボルを定義します。- pp オプションは、.TMS32025 および .TMS32050 シンボルを使って書かれたコードで使用できるので、コードをどちらのプロセッサ用にもコンパイルできます (3-8 ページの 3.4.2 項を参照して下さい)。

- q (quiet: 静的な実行) 見出しとすべての進捗情報を抑制します。

- s すべての定義済み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。通常は、アセンブラがシンボル・テーブルに入れるのはグローバル・シンボルだけです。- s を使うと、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。

- v バージョンを指定します。バージョンは、次の TMS320 デバイスのうち、どのデバイス用のコードを作成するのかを、アセンブラに指示します（デフォルトは - v25 です)。

-v10 は TMS320C1x を選択します。

-v16 は TMS320C16 を選択します。

-v20 は TMS320C20 を選択します。

-v25 は TMS320C2x を選択します。

-v2xx は TMS320C2xx を選択します。

-v50 は TMS320C5x を選択します。

- w TMS320C5x コードのパイプライン・コンフリクトを検出します。これは基本的には警告スイッチです。このオプションは TMS320C5x の場合にのみ有効です。詳しい説明については、3.4.3 項を参照して下さい。

- x クロスリファレンス・テーブルを作成し、それをリスト・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。ユーザがリスト・ファイルを要求しなかった場合でも、アセンブラがリスト・ファイルを自動的に作成します。

3.4 TMS320C1x/C2x/C2xx/C5x プロセッサの上位互換性

TMS320C1x/C2x/C2xx/C5x 固定小数点プロセッサは、上位のソース・コードと互換性があります。たとえば、TMS320C10 用に書かれたコードをアセンブラ・オプションの -v25 を使って TMS320C25 用にアセンブルすることが可能です。 -v オプションについては、3 - 5 ページに詳しい説明があります。

すべてのプロセッサは、ターゲット・プロセッサの番号がオリジナルのターゲット・プロセッサの番号と同じかそれ以上であれば上位移植されたコードを取り扱うことができます。ただし、コードの下位移植を行うと、予期しない結果を招くことがあります。多くの場合、コードのアセンブルに失敗します。

この節では、 -p オプションと -w オプションを使って移植を簡単にする方法を説明します。

3.4.1 移植時の不整合（ -p オプション）

TMS320C5x と TMS320C2xx のハードウェアの性能が強化された結果、コードの移植時にいくつかの不整合が生じることになりました。次に示す 3 つのケースは、TMS320C5x または TMS320C2xx ではサポートされないか、またはオリジナルのアルゴリズムを間接的に変更してしまう可能性があります。

- TMS320C5x および TMS320C2xx は、ARP 更新または AR 更新を伴う条件付き分岐をサポートしません。次の例を見て下さい。

  ```
  bgz label, *+, ar6
  ```

- NORM 命令は、命令パイプの 4 番目のフェーズで AR を変更します。したがって、パイプライン・コンフリクトが起こることがあります。

- TMS320C2x シリアル・ポート・コマンドは、TMS320C5x または TMS320C2xx については無効です。該当するコマンドは、RFSM、SFSM、RTXM、STXM、および FORT です。

このような移植にともなう不整合点に注意して、検出し修正する必要があります。しかし、コードを迅速に移植できるようにするために、 -p アセンブラ・オプションを使うことができます。 -p オプションは、上記のうち 2 つのケースを処理でき、3 番目のケースについてはフラグを作成します。 -p オプションは、適切なバージョン・オプション（ -v50 または -v2xx）と合わせて使って下さい。具体的には、 -p オプションは次の処理を実行します。

- ARP 更新を伴う条件付き分岐を、条件付き遅延分岐 -MAR -NOP 命令シーケンスに変換します。

  TMS320C5x

  ```
  bgz label, *+, ar6 は、変換後は  bcndd label, GT
                                   mar *+, ar6
                                   nop
  ```

  TMS320C2xx

  ```
  bgz label, *+, ar6 は、変換後は  mar *+, ar6
                                   BCND gt
  ```

- パイプラインを保護するために、NORM 命令の後に 2 つの NOP 命令を挿入します。

- シリアル・ポート制御レジスタ命令に不正なオペコードのフラグを付けます（この命令をアセンブラは有効な TMS320C5x 命令に直接的には変換できません）。

例 3 - 1 は、 - p と - v50 をいっしょに使ったときに TMS320C25 コードにどのような影響が出るかを示しています。

例 3 - 1. 移植されたコードにおける - p オプションと - v50 オプションの効果

```
 1 *-p option branch with ARP update
 2
 3 0000   b90a      LACK   10
 4 0001   7c01   L1 SBRK   1
 5 0002   90aa      SACL   *+,AR2
 6 0003   90a0      SACL   *+
 7 0004   f304      BGZ L1, AR1   ; arp update not supported on 'C5x
   0005   001'   ; *** DELAYED BRANCH REPLACES PREVIOUS INSTRUCTION ***
   0006   8b09      MAR ; *** UPDATE ARx AS IN PREVIOUS BRANCH ***
   0007   8b00      NOP ; *** INSERTED FOR PIPELINE ALIGNMENT ***
 8
 9 0008   2080   L2 ADD *
10 0009   f100      BBNZ   L2,AR1
   000a   0008   ; *** DELAYED BRANCH REPLACES PREVIOUS INSTRUCTION ***
   000b   8b09      MAR ; *** UPDATE ARx AS IN PREVIOUS BRANCH ***
   000c   8b00      NOP ; *** INSERTED FOR PIPELINE ALIGNMENT ***
11
12        * -p option indirect addressing after NORM instruction
13
14 000d   8b8a      LARP   AR2
15 000e   a080      NORM   *
   000f   8b00      NOP ; *** INSERTED FOR PIPELINE PROTECTION ***
   0010   8b00      NOP ; *** INSERTED FOR PIPELINE PROTECTION ***
16 0011   8213'     SAR 2,x       ; pipeline conflict
17 0012   8280      SAR AR2,*     ; pipeline conflict
18
19 0013   0000   x   .word  00h

No Errors, No Warnings
```

TMS320C2x コードの TMS320C5x への移植に関する詳しい説明については、TMS320C5x ユーザーズ・マニュアルを参照して下さい。TMS320C2x コードの TMS320C2xx への移植に関する詳しい説明については、TMS320C2xx User's Guide を参照してください。

3.4.2 'C2x 用または 'C5x 用のコードから 'C2xx への移植（ - pp オプション）

.TMS32025 および .TMS32050 シンボルを使って書かれたコードのボディがすでにあってどちらのプロセッサでもコードをコンパイルできる状態になっている場合は、 - v2xx オプションと - pp オプションを組み合わせて使って、このコードを TMS320C2xx に簡単に移植することができます。 - pp オプションは、.TMS32025 と .TMS32050 の両方のシンボルを定義し、3.4.1 項で説明した移植時の不整合性を解決します。これによって、例 3 - 2 に示すような作成済みのコードを、修正せずに直接 TMS320C2xx に移植することができます。

例 3 - 2. TMS320C25 と TMS320C50 の基本アセンブリ構造

```
      .if .tms32050
          .
          .
      code specific to 'C50
          .
          .
      .endif

      .if .tms32025
          .
          .
      code specific to 'C25
          .
          .
      .endif
```

- pp スイッチを使わずに例 3 - 3 を TMS320C2xx 用にアセンブルする場合には、行 .if .tms32025 は、アセンブルを正しく実行するために、.if .tms32025 | | .tms3202xx に変更する必要があります。 - pp スイッチを使うと、.tms32025 シンボルがアセンブラで定義されるので、コード・セグメントの変更は不要になります。

例 3 - 3. TMS320C25 および TMS320C50 から TMS320C2xx への移植

```
          .
          .
      .if .tms32050
      BD  label       ; delayed branch, specific to 'C50
      .endif

      INSTR1
      INSTR2

      .if .tms32025
      B   label       ; C25 does not have delayed branch
      .endif
          .
          .
```

このオプションは、既存のコードを TMS320C2xx デバイスに移植するときの便宜を計るために提供されています。 - pp オプションは、アセンブル時に - v2xx オプションを使用している場合にのみ、正常に働きます。

3.4.3 TMS320 パイプラインのプログラミング（ - w オプション）

TMS320C1xおよびTMS320C2xプロセッサの命令実行はパイプライン化されています。このパイプラインは、分岐命令などによって中断される場合を除き、基本的には透過的です。プロセッサの個々のパイプラインの詳しい説明については、該当の TMS320 ユーザーズ・マニュアルを参照して下さい。

4 段階のパイプラインやメモリマップド・レジスタなど、TMS320C5x のハードウェアの性能が強化された結果、パイプライン・コンフリクトが起こることがあります。 - w アセンブラ・オプションを指定すると、次のようなパイプライン・コンフリクトについて警告が表示されます。

- WARNING - CONTEXT SWITCH IN DELAYED SEQUENCE

  この警告は、遅延シーケンス・コンテキスト変更中のコンテキスト・スイッチを示します。プロセッサ・コンテキストを変更する命令は次のとおりです。

  TRAP[D] . . CALA[D] . . BACC[D] RETI
  BCND[D] CC[D] . . RETC[D] RETE
  B[D] . . CALL[D] . . BANZ[D] INTR

- WARNING - POSSIBLE PIPE HIT AT BRANCH ADDRESS

  この警告は、遅延シーケンス・コンテキスト変更中に NORM 命令が検出されたことを示します。NORM 命令はパイプライン・コンフリクトの原因になることがあります。

- WARNING - DELAYED JUMP MISSES SECOND WORD

  この警告は、遅延シーケンス・コンテキスト・スイッチの 2 番目のワードに 2 ワード命令が入っていることを示します。2 番目のワードは、分岐アドレスから取り出されることになります。

- WARNING - POSSIBLE PIPE CONFLICT WITH NORM

  この警告は、NORM 命令の次に続く 2 ワードの中で間接アドレッシングが使用されていることを示します。これはパイプライン・コンフリクトの原因になることがあります。

- WARNING - POSSIBLE PIPE HIT WITH ARx ACCESS

  この警告は、メモリ・アドレスとして AR レジスタにアクセスしていることを示します。これはパイプライン・コンフリクトの原因になることがあります。

注: - w オプションがすべてのパイプライン・コンフリクトについて警告するわけではありません

TMS320C5x をプログラムするときは、パイプライン化された実行の効果に十分注意してください。 - w スイッチは、すべてのパイプライン・コンフリクトについて警告を出すわけではありません。アセンブラが検出できない微妙な状況もあります。

例 3 - 4 は、TMS320C25 コードに対する - w オプションの影響を示しています。

例 3 - 4. 移植されたコードにおける - w オプションの効果

```
 1        * -w switch context switch
 2
 3 0000   8b89      LARP   AR1
 4 0000   3080   x  SUB    *
 5
 6 0002   f388      BCNDD  x,eq
   0003   0001'
 7 0004   2080      ADD    *
 8 0005   7980      B      Y   ; context switch during delayed sequence
   0006   0009'
********** WARNING - CONTEXT SWITCH IN DELAYED SEQUENCE
 9
10 0007   2080      ADD    *
11 0008   2080      ADD    *
12
13 0009   8B00   y  NOP
14
15
16        * -w switch norm pipeline conflict
17
18 000a   8b8a      LARP   AR2
19 000B   a080      NORM   *   ; norm hits ar2 in execute phase
20 000c   820e      SAR    AR2,exp
********** WARNING - PROBABLE PIPE CONFLICT WITH NORM
LAST ERROR AT 8
0021   000d0200     LAR    AR2,0
********** WARNING - PROBABLE PIPE CONFLICT WITH NORM
LAST ERROR AT 20
22
23 000e   0000   exp.WORD  OOh
24
25        * -w switch arx as an address pipeline conflict
26
27
28 000f   101b      LAC    AR3,AR3      ; modifies ar3 in fourth phase
********** WARNING - POSSIBLE PIPE HIT WITH ARx ACCESS
LAST ERROR AT 21
29 0010   2080      ADD    *         ; uses ar3 in second phase

No Errors, 4 Warnings
```

コンフリクトの潜在的原因になる命令のリストとこれらの命令の使用法も含め、パイプライン・コンフリクトの詳しい説明については、TMS320C5x ユーザーズ・マニュアルを参照して下さい。

3.5 アセンブラ入力のための代替ディレクトリの命名

.copy、.include、.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うように指示します。.copy と .include 疑似命令は、アセンブラに対して別の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 .copy、.include、.mlib 疑似命令の例が挙げてあります。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先のコピー/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。ファイル名は、絶対パス名、部分パス名、パス情報の指定のないファイル名のどれでもかまいません。アセンブラは、以下のディレクトリ内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、.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していたファイルを指します。
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、-i アセンブラ・オプションや環境変数を使って強化することができます。

3.5.1 -i アセンブラ・オプション

-i アセンブラ・オプションは、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の名前を指定します。-i オプションのフォーマットは次のとおりです。

```
dspa  -ipathname  source filename
```

各呼び出しにつき最高 10 回 -i オプションを使うことができます。各 -i オプションが 1 つの *pathname* を命名します。アセンブリ・ソースでは、パス情報を指定せずに疑似命令.copy、.include、または .mlib を使うことができます。アセンブリは、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と、-i オプションにより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 のパス名	呼び出しコマンド
DOS	`c:\dsp\files\copy.asm`	`dspa -ic:\dsp\files source.asm`
UNIX	`/dsp/files/copy.asm`	`dspa -i/dsp/files source.asm`

アセンブラは、source.asm が現行のディレクトリにあるので、まず現行のディレクトリで 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を検索します。

3.5.2 A_DIR環境変数

環境変数とは、ユーザが定義して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 **A_DIR** 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。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。

DOS `set A_DIR =` *pathname;another pathname ...*

UNIX `setenv A_DIR` **”***pathname;another pathname ...* **”**

pathnames にはコピー/インクルード・ファイル、またはマクロ・ライブラリが入った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、.mlib 疑似命令を使うことができます。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 を使って指定したディレクトリにファイルが見つからない場合は、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″
```

	パス名	呼ぶだしコマンド
DOS	`c:\320\files\copy1.asm` `c:\dsys\copy2.asm`	`set  A_DIR=c:\dsys` `asm30  -ic:\320\files source.asm`
UNIX	`/320/files/copy1.asm` `/dsys/copy2.asm`	`setenv  A_DIR ″/dsys″` `asm30  -i/320/files source.asm`

アセンブラは、source.asm が現行のディレクトリにあるので、まず現行のディレクトリで 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 を使って指定したディレクトリを検索して copy2.asm を検出します。

設定した環境変数は、次にシステムをリブートするか以下のコマンドを入力して変数を再設定するまで変わりません。

DOS `set A_DIR=`

UNIX `unsetenv A_DIR`

3.6 ソース文のフォーマット

TMS320 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、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いくら長くてもかまいませんが、アセンブラは 1 行につき 200 字までしか読みません。1 つの文の文字数が 200 を超えると、アセンブラはその行を切り捨てて警告を発します。

以下にソース文の例を示します。

```
SYM1      .set      2      ; Symbol SYM1 = 2
Begin:    LDPK      SYM1   ; Load DP with 2
          .word     016h   ; Initialize a word with 016h
```

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。

[*label*] [:] *mnemonic* [*operand list*] [;*comment*]

以下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、セミコロンのいずれかで始めなければなりません。
- 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ことが必要です。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。

3.6.1 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。使用する場合には、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、_、$）を使うことができます。ラベルでは大文字と小文字が区別され、先頭に数字を使うことはできません。ラベルの後ろにコロン (:) を付けることができますが、コロンはラベル名の一部としては取り扱われません。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、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には、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 疑似命令を使っていくつかのワードを初期化する場合には、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。

```
.      .           .          .
.      .           .          .
.      .           .          .
9      003F        * Assume some other code was assembled
10     0040  000A  Start:  .word 0Ah,3,7
       0041  0003
       0042  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。これは次の疑似命令文と同じ働きをします。

```
label .set   $  ;  $ provides the current value of the SPC
```

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を指します (SPC はインクリメントされません)。

```
3   0050            Here:
4   0050   0003        .word 3
```

3.6.2 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。ニーモック・フィールドは、1 カラム目から始めてはなりません。1 カラム目から始めると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を含めることができます。

- 機械命令 (たとえば、ABS、MPYU、SPH)
- 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)
- マクロ呼び出し

3.6.3 オペランド・フィールド

オペランド・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（3-15 ページの 3.7 節を参照）、シンボル（3-18 ページの 3.9 節 を参照）または式の中の定数とシンボルの組合せ(3-23 ページの 3.10 節を参照)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。

3.6.4 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ます。そして、ソース行の最後まで続きます。コメントには、ASCII 文字と空白を使うことができます。コメントは、アセンブリ・ソース・リストに出力することはできますが、アセンブリに何の影響も与えません。

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 (*) を使うことができます。 行の中のそれ以外の位置からコメントを始める場合は、最初の文字は必ず ; でなければなりません。記号 * は、1 カラム目で使われた場合のみ、それがコメントであることを示します。

3.7 定数

アセンブラは 6 つの型の定数をサポートしています。

- ❑ 2 進整数
- ❑ 8 進整数
- ❑ 10 進整数
- ❑ 16 進整数
- ❑ 文字定数
- ❑ アセンブリ時定数

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。定数は符号拡張されないので注意が必要です。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。

3.7.1 2 進整数

2 進整数定数とは、16 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。指定された定数が 16 桁以下の数値を指定した場合には、アセンブラはその数値を右側揃えとし、未指定の桁をゼロで埋めます。有効な 2 進定数の例を以下に示します。

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

3.7.2 8 進整数

8 進整数定数は、16 桁までの 8 進数（0 から 7 まで）に接尾部 **Q**（または **q**）を付けた文字列です。有効な 8 進定数の例を以下に示します。

10Q 8_{10} または 8_{16} に等しい定数

100000Q $32{,}768_{10}$ または 8000_{16} に等しい定数

226q 150_{10} または 96_{16} に等しい整数

3.7.3 10 進整数

10 進整数定数とは、 - 32,768 から 65,535 までの 10 進数の文字列です。 有効な 10 進定数の例を以下に示します。

1000 1000_{10} または $3E8_{16}$ に等しい定数

- 32768 $-32{,}768_{10}$ 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

3.7.4 16 進整数

16 進整数定数とは、4 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 から 9 までの 10 進数と A から F まで、または a から f までのアルファベットが使われます。 16 進定数の先頭の文字は 10 進数（0 から 9 まで）でなければなりません。4 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 進定数の例を以下に示します。

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 $14{,}252_{10}$ または $37AC_{16}$ に等しい定数

3.7.5 文字定数

文字定数とは、単一引用符で囲まれた 1 文字または 2 文字の文字列です。文字は、内部的に 8 ビットの ASCII 文字で表されます。文字定数の中で個々の単一引用符を示すには単一引用符を 2 つ続けて使います。2 つの単一引用符だけでできている（文字を含まない）文字定数も有効で、値 0 が割り当てられます。1 文字だけを指定した場合は、ビットの右側揃えが行なわれます。有効な文字定数の例を以下に示します。

'a' 文字定数 a を定義し、内部的には 61_{16} として表されます。

'C' 文字定数 C を定義し、内部的には 43_{16} として表されます。

'''' 文字定数 ' を定義し、内部的には 27_{16} として表されます。

'' ヌル文字を定義し、内部的には 00_{16} として表されます。

文字定数と文字列は同じではないことに注意して下さい（文字列については 3-17 ページの 3.8 節で説明します）。文字定数は、1 つの整数値を表わします。一方、文字列は文字のリストです。

3.7.6 アセンブリ時定数

.set 疑似命令を使って値をシンボルに割り当てると、そのシンボルは定数となります。この定数を式の中で使うためには、割り当てる値は絶対数でなければなりません。例を示します。

```
shift3    .set   3
          LAC    *,shift3,AR1
```

.set 疑似命令を使って、レジスタ名用のシンボル定数を割り当てることもできます。この場合には、そのシンボルはレジスタと同義語になります。

```
AuxR1     .set   AR1
          LAC    *,0,AuxR1
```

3.8 文字列

文字列とは、二重引用符に囲まれた文字の列を指します。文字列の一部として二重引用符を使う場合には、連続した二重引用符を使います。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。

以下に有効な文字列の例を示します。

”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。

文字列は以下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” のようなデータ初期化疑似命令
- .string 疑似命令のオペランド

3.9 シンボル

シンボルは、ラベル、定数、置換シンボルに使われます。シンボル名は、32 文字までの英数字（A～Z、a～z、0～9、$、および_）の列です。 シンボルの最初の文字として数字を使うことはできません。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・オプションを使う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 疑似命令を使っ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リ時にのみ有効です。

3.9.1 ラベル

シンボルをラベルとして使うと、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われるラベルは、ユニークなものでなければなりません。ニーモニック命令コードとアセンブラ疑似命令の名前（接頭部「.」のないもの）は有効なラベル名です。

ラベルを .global、.ref、.def、.bss 疑似命令のオペランドとして使うこともできます。

```
          .global   label1

label2    nop
          add       label1
          b         label2
```

3.9.2 シンボル定数

シンボルを定数値に設定することができます。定数を使うと、意味のある名前を定数値にすることができます。.set および .struct/.tag/.endstruct 疑似命令を使う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。次にこれらの疑似命令の使い方を示します。

```
K      .set   1024            ; constant definitions
maxbuf .set   2*K

item   .struct                ; item structure definition
       .int   value           ; constant offsets value = 0
       .int   delta           ; constant offsets delta = 1
i_len  .endstruct

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

       .ADD   array.delta     ; array+1
```

アセンブラにも事前定義されたシンボル定数がいくつかあります。それについては次の節で説明します。

3.9.3 シンボル定数の定義（ - d オプション）

- d オプションは、定数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うことができます。 - d オプションのフォーマットは次のとおりです。

dspa - d*name* = [*value*]

name は定義するシンボルの名前で、value はシンボルに割り当てる値です。value が指定されていない場合、シンボルは 1 に設定されます。

アセンブラ・ソース内では、次の疑似命令を使っ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組み込み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下さい。引用符で囲むと、引数は、置換シンボルとしてでなくそのままの文字として解釈されます。

3.9.4 事前定義されたシンボル定数

アセンブラには、以下のものを含め、事前定義されたシンボルがいくつかあります。

- $（ドル記号文字）は、セクション・プログラム・カウンタ (SPC) の現行の値を表します。
- **レジスタ・シンボル**
  - AR0 - AR1 (TMS320C1x 用)
  - AR0 - AR7 (TMS320C2x/C2xx/5x 用)
- **ポート・アドレス・シンボル**
  - PA0 - PA7 (TMS320C1x 用)
  - PA0 - PA15 (TMS320C2x/C2xx/5x 用)
- **バージョン・シンボル**は、事前定義されたアセンブラ定数で、各種のターゲット・プロセッサ用のコードを作成することをアセンブラに指示するために使うことができます。バージョン・シンボルは、ターゲット・バージョン・スイッチまたは .version 疑似命令とともに使って下さい。

バージョン・シンボル	- Versionオプション	Version疑似得命令
.TMS32010	-v10	10
.TMS32016	-v16	16
.TMS32020	-v20	20
.TMS32025	-v25	25
.TMS3202XX	-v2xx	29
.TMS32050	-v50	50
.TMS320XX	プロセッサ・バージョン	

.TMS320XX 定数は、バージョン・スイッチの値に設定されます。たとえば、アセンブラを次のコマンドを使って呼び出したとします。

```
dspa -v20 filename
```

すると、.TMS320XX は 20 に等しく、.TMS32020 は 1 または真に等しく、.TMS32010、.TMS32016、.TMS32025、.TMS32050、.TMS3202XX はすべてゼロまたは偽に等しくなります。

バージョン・シンボルを使うと、アセンブラに、特定のターゲット・デバイス用のコードを作成するよう指示できます。たとえば、次の例では、アセンブラを呼び出した方法に応じて異なるターゲット・デバイス用のコードを作成できます。

```
if     .TMS320xx = 50
       .
       .
.endif
.if    .TMS32020
       .
       .
.endif
```

❑ .mmregs 疑似命令を使って定義されるシンボル。

注: レジスタ・シンボル

レジスタ・シンボルの値は、TMS320C1x/C2x/C2xx デバイスの場合は AR0 = 0、AR1 = 1、AR2 = 2 のようになります。しかし、TMS320C5x デバイスの場合は、デバイスのメモリマップド・アドレスと同じになります。つまり、アセンブラはシンボルのコンテキストを次のように解釈します。

```
ADD *+, AR7
ADD AR7
```

上記の最初の例では、AR7 の値は 7 です。2 番目の例では、AR7 は AR7 のアドレスになり、アドレス値が 17 なら 17 になります。

3.9.5 置換シンボル

シンボルを文字列値（変数）に割り当てることができます。これにより文字列に別名を付けることによって、シンボル名を文字列と同様に扱うことができるようになります。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、そのシンボル名を文字列値に変えます。シンボル定数の場合と違って、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に置換シンボルを割り当てることができます。例を示します。

```
   .asg  ”ar0”,    SP     ;  stack pointer
   .asg  ”*+”,     INC    ;  indirect auto-increment
   .asg  ”*-”,     DEC    ;  indirect auto-decrement

   ADD   label, INC, SP
```

マクロを使うときに置換シンボルが重要な役割を果たします。なぜならば、マクロ・パラメータと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。以下のコードは、置換シンボルがマクロでどのように使われるかを示しています。

```
add2   .macro    a,b,dest  ;  add2 macro definition

       lac    a
       add    b
       sacl   dest

       .endm

*add2  invocation
       add2   loc1,loc2,loc3  ;  assign ”loc1” argument to a
```

マクロについての詳しい説明は第6章を参照してください。

3.9.6 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とを持つ特殊ラベルです。ローカル・ラベルの形式は $n で、n は 0 から 9 までの範囲の 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。

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、定義を解除したり再び定義したりできます。

ローカル・ラベルは、次の 4 つの方法のいずれかで、定義を解除（つまりリセット）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）を開始します。
- 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）を終了します。

次に示すのは、ローカル・ラベルの宣言と使い方が正しいコードの例です。

```
Label1:   lac a
          sub b
          blz $1
          lac b
          b  $2
$1        lac a
$2        add c
          .newblock ; Undefine $1 so it can be used again
          blez $1
          sacl c
$1        nop
```

次に、ローカル・ラベルの使い方が誤っているコード例を示します。

```
Label1:   lac a
          sub b
          blz $1
          lac b
          b  $2
$1        lac a
$2        add c
          blez $1 ;
          sacl c
$1        nop    ; WRONG $1 is multiply defined
```

ローカル・ラベルはマクロの中で使うと特に便利です。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ラベルの数は最大 10 個です。あるローカル・ラベルの定義を解除した後で、同じものを再び定義して使うことができます。ローカル・ラベルはオブジェクト・コードのシンボル・テーブルには組み込まれません。

3.10 式

式とは、定数、シンボル、または算術演算子によって区切られた一連の定数とシンボルです。有効な式の値の範囲は、 - 32,768 から 32,767 です。式の評価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評価します。

8/(4/2) = 4、ただし 8/4/2 = 1

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

演算子（表3 - 1 を参照）は優先順位を基準に 4 つのグループに分かれます。式の評価の順序が括弧で決められない場合は、優先順位が高い順に評価が行われます。

8 + 4/2 = 10 （4/2 が最初に評価されます。）

左から右へ向かって 評価する

括弧によっても優先順位グループによっても式の評価順序が決められない場合は、式はグループ 1 を除いて左から右に向かって評価されます。グループ 1 は右から左に評価されます。

8/4*2 = 4、ただし 8/(4*2) = 1

3.10.1 演算子

表3 - 1 に式で使える演算子を示します。優先順位の高いグループから順に列挙します。

表3 - 1. 式で使える演算子 (優先順位)

グループ	演算子	説明
1	+	単項プラス
	-	単項マイナス
	~	1の補数
2	*	乗算
	/	除算
	%	剰余 (モジュロ)
	< <	左シフト
	> >	右シフト
3	+	加算
	-	減算
	^	ビット単位 XOR
	|	ビット単位 OR
	&	ビット単位 AND
4	<	より小さい
	>	より大きい
	< =	小さいか等しい
	> =	大きいか等しい
	= または = =	等しい
	!=	等しくない

注: グループ 1 の演算子は右から左へ評価されます。その他のすべての演算子は左から右へ評価されます。

3.10.2 式のオーバーフローとアンダーフロー

アセンブラは、アセンブリ時に算術演算が行なわれるときに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を見つけると 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。

3.10.3 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リ時定義のみが含まれます。整合定義式の評価は絶対でなければなりません。

整合定義式の例を示します。

```
1000h+X
```

ここで X は絶対シンボルとして前に定義されているものです。

3.10.4 条件式

アセンブラでは、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なうときに特に便利です。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		
<	より小さい	< =	小さいか等しい
>	より大きい	> =	大きいか等しい

条件式の評価は、真の場合は 1 に、偽の場合は 0 になります。条件式が使えるのは、オペランドのタイプが等価の場合だけです。つまり、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。

3.10.5 再配置可能なシンボルと有効な式

表3 - 2 は、絶対シンボル、再配置可能シンボル、外部シンボルの有効な演算についてまとめたものです。式は再配置可能シンボルや外部シンボルで乗算や除算をすることはできません。式には、別のセクションの再配置可能な未解決のシンボルを使うことはできません。

.global 疑似命令を使ってグローバルとして定義されたシンボルまたはレジスタを式で使うこともできます。表 3 - 2 では、この種のシンボルやレジスタは外部として示されています。

再配置可能レジスタは、式の中で使えます。これらのレジスタのアドレスは、外部レジスタとして宣言されているものを除き、レジスタが定義されたレジスタ・セクションに関して再配置可能です。

表3 - 2. 絶対シンボルおよび再配置可能シンボルを持った式

A が...	そして B が...	A+B は...	A - B は...
絶対	絶対	絶対	絶対
絶対	外部	外部	不正
絶対	再配置可能	再配置可能	不正
再配置可能	絶対	再配置可能	再配置可能
再配置可能	再配置可能	不正	絶対†
再配置可能	外部	不正	不正
外部	絶対	外部	外部
外部	再配置可能	不正	不正
外部	外部	不正	不正

† A と B は同じセクションになければなりません。それ以外は不正となります。

再配置可能シンボルと絶対シンボルを使った式の例をいくつか示します。以下の例では、同じセクション内で定義された 4 つのシンボルを使います。

```
.global   extern_1  ;  Defined in an external module
intern_1: .word '"D';  Relocatable, defined in current module
LAB1:     .set 2    ;  LAB1 = 2
intern_2            ;  Relocatable, defined in current module
```

❑ 例1

この例の文では、絶対シンボル LAB1 (上記の例で値 2 を持つように定義されています) が使われています。最初の文は、値 51 をアキュムレータに入れ、2 つ目の文は値 27 をアキュムレータに入れます。

```
LACK  LAB1  +  ((4+3) *  7)    ;  ACC = 51
LACK  LAB1  +  4  +  (3*7)     ;  ACC = 27
```

❑ 例2

すべての有効な式は、以下のいずれかの形式になります。

再配置可能シンボル ± 絶対シンボル

または

絶対値

単項演算子は絶対値にのみ使うことができます。再配置可能シンボルには適用できません。1 つのみの再配置可能シンボルを含む形に整理できない式は不正です。次の例の最初の文は有効です。それに続くすべての式は不正です。

```
LACK extern_1 - 10       ; Legal
LACK 10-extern_1         ; Can't negate reloc. symbol
LACK -(intern_1)         ; Can't negate reloc. symbol
LACK extern_1/10         ; / isn't an additive operator
LACK intern_1 + extern_1 ; Multiple relocatables
```

❑ **例 3**

以下の最初の式は有効です。intern_1 と intern_2 は再配置可能ですが、両方とも同じセクションに属すのでその差は絶対的です。1 つの再配置可能シンボルを別の再配置可能なシンボルから引くと、式は、**再配置可能シンボル** + **絶対値** の形の再配置可能な式に整理することができます。2 番目の文は、2 つの再配置可能シンボルの合計が絶対値にならないので不正です。

```
LACK  intern_1 - intern_2 + extern_1    ; Legal
LACK  intern_1 + intern_2 + extern_1    ; Illegal
```

❑ **例 4**

外部シンボルの位置は、式を評価する上で重要です。以下の文は前の例の最初の文に似ていますが、不正です。なぜならば、左から右への演算子優先順位によりアセンブラが intern_1 を intern1 に加えようとするからです。

```
LACK  intern_1 + extern_1 - intern_2    ; Illegal
```

3.11 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ためには、オプション -l（小文字の「L」）を付けてアセンブラを呼び出します。

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、タイトル行があります。.title 疑似命令で指定するすべてのタイトルはこの行に出力されます。ページ番号はタイトルの右側に出力されます。.title 疑似命令を使わない場合は、タイトル領域は空白となります。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 (例 3 - 5 参照)。 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、ソース文の行の直ぐ下に置かれます。

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・ファイル中のリストには出力されない介在文の数を表します。

組み込みファイル文字

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、その行が組み込みファイルからアセンブルされたことを示します。

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、マクロ展開とループ・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。すべてのセクション（.text、.data、.bss、名前付きセクション）には別々の 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以下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
^	.asect 再配置可能

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。アセンブラでは、1 行に最大 200 字までを使うことができます。このフィールドの空白指定はソース文の空白指定によって決まります。

例 3 - 5 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 - 5.　アセンブラ・リスト

```
                2            .global   RESET, INT0 INT1, INT2
                3            .global   TINT, RINT, XINT, USER
                4            .global   ISR0, ISR1, ISR2
                5            .global   time, rcv, xmt, proc
                6
                7            .copy  ”init.inc”
          A     1         initmac   .macro
          A     2         ! initialize macro
          A     3            rovm   ; disable oflow
          A     4            ldpk 0 ; dp = 0
          A     5            larp 7 ; arp = ar7
          A     6            lack 03fh ; acc = 03fh
          A     7            sacl 4 ; enable ints
          A     8            .endm
                8         *
                9         ****************************************
               10         * Reset and interrupt vectors    *
               11         ****************************************
               12 0000           .sect  ”reset”
               13 0000   ff80    RESET: B   init
                  0001   0008+
               14 0002   ff80    INT0:  B   ISR0
                  0003   0000!
               15 0004   ff809   INT1:  B   ISR1
                  0005   0000!
               16 0006   ff80    INT2   B   ISR2
                  0007   0000!
               17         *
               18 0000           .sect  ”ints”
               19 0000   ff80    TINT   B   time
                  0001   0000!
               20 0002   ff80    RINT   B   rcv
                  0003   0000!
               21 0004   ff80    XINT   B   xmt
                  0005   0000!
               22 0006   ff80    USER   B   proc
                  0007   0000!
               23         ****************************************
               24         * Initialize processor        *
               25         ****************************************
               26 0008       init:  initmac
                1 0008   ce02       rovm   ; disable oflow
                1 0009   c800       ldpk 0 ; dp = 0
                1 000a   558f       larp 7 ; arp = ar7
                1 000b   ca3f       lack 03fh ; acc = 03fh
                1 000c   6004       sac14  ; enable ints
               27 000d   ca00       zac ; zero acc
               28 000e   c760       lark ar7, 060h    ; ar7 -> B2
               29 000f   cb1f       rptk 31
               30 0010   60a0       sacl *+   ; zero out
```

フィールド 1　フィールド 2　フィールド 3　フィールド 4

3.12 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・リストを取得するためには、アセンブラを起動するときに -x オプションを使うか、.option 疑似命令を使います。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。

例 3 - 6. アセンブラのクロス・リファレンス・リスト

```
LABEL                           VALUE     DEFN      REF

INT0                            0002+      14       2
INT1                            0004+      15       2
INT2                            0006+      16       2
ISR0                            REF                 4    14
ISR1                            REF                 4    15
ISR2                            REF                 4    16
RESET                           0000+      13       2
RINT                            0002+      24       3
TINT                            0000+      23       3
VECS                            0006+      26       3
XINT                            0004+      27
init                            0000+      34      13
```

label このカラムには、アセンブリ時に定義、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を表わす 4 桁の 16 進数またはシンボルの属性を表す名前が入ります。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示しましてあります。

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列挙されます。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. シンボル属性

文字または名前	意味
REF	外部参照（グローバル・シンボル）
UND	未定義
’	.text セクションで定義されたシンボル
”	.data セクションで定義されたシンボル
+	.sect セクションで定義されたシンボル
-	.bss または .usect セクションで定義されたシンボル

3.13 拡張命令の形式

拡張命令は、複数の類似した命令の機能を実行する単一のニーモニックです。このような拡張命令は、TMS320C2x、TMS320C2xx、および TMS320C5x の各デバイスに有効です。

アセンブラは、次に示すように、拡張命令のオペランドを解釈して適切なオペコードを判別します。

拡張命令		基本命令	
ADD	x	ADD	x
ADD	*+	ADD	*+
ADD	#100	ADDK	100
ADD	#3000	ADLK	3000
ADD	*+,16	ADDH	*+

拡張命令の詳しい説明については、5-5 ページの 5.2 節を参照してください。

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うと、以下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。
- ❑ リストの出現を制御する。
- ❑ メモリの初期化を行う。
- ❑ 条件ブロックをアセンブルする。
- ❑ グローバル変数を定義する。
- ❑ 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・デバッグ情報を調べる。

第 1 部では（4.1 節から 4.9節まで）機能別に疑似命令を説明し、第 2 部には（4.10 節）アルファベット順のリファレンスを掲載します。この章の内容は以下のとおりです。

項目 **ページ**

4.1 疑似命令のまとめ

表4 - 1 は、アセンブラ疑似命令をまとめたものです。疑似命令を含むすべてのソース文には、ラベルとコメントを指定することができます。読みにくくなるので、ラベルとコメントは疑似命令の構文には示してありません。

表4 - 1. アセンブラ疑似命令のまとめ

セクションを定義する疑似命令	
ニーモニックと構文	**説明**
.asect "*section name*", *address*	名前付き（初期化された）絶対的なセクションにアセンブルします（この疑似命令はもう使われません）。
.bss *symbol*, *size in words* [,*blocking flag*]	size ワードを .bss（初期化されないデータ）セクションに確保します。
.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。
.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。
.text	.text（実行可能コード）セクションにアセンブルします。
symbol **.usect** "*section name*", *size in words*, [*blocking flag*]	size ワードを名前付き（初期化されない）セクションに確保します。

定数（データとメモリ）を初期化する疑似命令	
ニーモニックと構文	**説明**
.bes *size in bits*	現行のセクションに size ビットを確保します。ラベルは、確保した空間の終りを指します。
.bfloat *value*	32 ビット、IEEE 単精度、浮動小数点定数を初期化します。ページ境界をまたいでオブジェクトを配置することはできません。
.blong $value_1$ [, ... , $value_n$]	1 つ以上の 32 ビット整数を初期化します。ページ境界をまたいでオブジェクトを配置することはできません。
.byte $value_1$ [, ... , $value_n$]	現行のセクションの 1つ以上の連続したバイトを初期化します。
.field *value* [, *size in bits*]	可変長フィールドを初期化します。
.float *value*	32 ビット、IEEE 単精度、浮動小数点定数を初期化します。
.int $value_1$ [, ... , $value_n$]	1 つ以上の 16 ビット整数を初期化します。
.long $value_1$ [, ... , $value_n$]	1 つ以上の 32 ビット整数を初期化します。
.space *size in bits*	現行のセクションに size ビットを確保します。ラベルは、確保した空間の始まりを指します。
.string "$string_1$"[, ... , "$string_n$"]	1 つ以上のテキスト文字列を初期化します。
.word $value_1$ [, ... , $value_n$]	1 つ以上の 16 ビット整数を初期化します。

表4 - 1. アセンブラ疑似命令のまとめ（続き）

セクション・プログラム・カウンタ (SPC) の位置合わせを行う疑似命令	
ニーモニックと構文	**説明**
.align	SPC をページ境界で位置合わせします。
.even	SPC を偶数ワード境界で位置合わせします。

出力リストをフォーマットする疑似命令	
ニーモニックと構文	**説明**
.drlist	すべての疑似命令行のリスト作成を可能にします（デフォルト）。
.drnolist	一部の疑似命令行のリスト作成を禁止します。
.fclist	偽の条件付きコード・ブロックのリスト作成を認めます（デフォルト）。
.fcnolist	偽の条件付きコード・ブロックのリスト作成を禁止します。
.length *page length*	ソース・リストのページの長さを設定します。
.list	ソース・リストの作成を再開します。
.mlist	マクロ・リストとループ・ブロックの作成を認めます（デフォルト）。
.mnolist	マクロ・リストとループ・ブロックの作成を禁止します。
.nolist	ソース・リストの作成を中止します。
.option {*B* | *D* | *F* | *L* | *M* | *T* | *X*}	出力リスト作成オプションを選択します。
.page	ソース・リストのページ替えを行います。
.sslist	展開された置換シンボルのリスト作成を認めます。
.ssnolist	展開された置換シンボルのリスト作成を禁止します（デフォルト）。
.tab *size*	タブ・サイズを設定します。
.title ”*string*”	リストのページ見出しにタイトルを出力します。
.width *page width*	ソース・リストのページの幅を設定します。

表4 - 1.　アセンブラ疑似命令のまとめ（続き）

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**
.copy ["]*filename*["]	**他のファイルからのソース文を含めます。**
.def $symbol_1$[, ... , $symbol_n$]	**現行のモジュールで定義され、他のモジュールで使われる** 1 **つ以上のシンボルを特定します。**
.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。**
.include ["]*filename*["]	**他のファイルからのソース文を含めます。**
.mlib ["]*filename*["]	**マクロ・ライブラリを定義します。**
.ref $symbol_1$[, ... , $symbol_n$]	**他のモジュールで定義され、現行のモジュールで使われる** 1 **つ以上のシンボルを特定します。**

条件付きアセンブル疑似命令	
ニーモニックと構文	**説明**
.break [*well-defined expression*]	**条件が真であれば** .loop **のアセンブリを終了します。**.break **は必須ではありません。**
.else	.if **条件が偽であればコード・ブロックをアセンブルします。** .else **は必須ではありません。**
.elseif *well-defined expression*	.if **条件が偽で、かつ** .elseif **条件が真の場合に、コード・ブロックをアセンブルします。**.elseif **は必須ではありません。**
.endif	.if **コード・ブロックを終了します。**
.endloop	.loop **コード・ブロックを終了します。**
.if *well-defined expression*	**条件が真であれば、コード・ブロックをアセンブルします。**
.loop [*well-defined expression*]	**コード・ブロックの繰り返しアセンブリを開始します。**

表4 - 1. アセンブラ疑似命令のまとめ（続き）

アセンブリ時間記号	
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。
.endstruct	構造体定義を終了します。
.equ	値をシンボルと等価にします。
.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。
.newblock	ローカル・ラベルの定義を解除します。
.set	値をシンボルと等価にします。
.struct	構造体定義を開始します。
.tag	構造体属性をラベルに割り当てます。

その他の疑似命令	
ニーモニックと構文	**説明**
.emsg *string*	ユーザ定義のエラー・メッセージを出力デバイスに送ります。
.end	プログラムを終了します。
.label *symbol*	セクション内にロード時に再配置可能なラベルを定義します。
.**mmregs**	メモリマップド・レジスタをシンボル・テーブルに入れます。
.mmsg *string*	ユーザ定義メッセージを出力デバイスに送ります。
.port	アセンブラの移植スイッチをオンにします。
.sblock ”*section name*” [,”*section name*”, . . .]	ブロッキング用のセクションを指定します。
.version *generation #number*	プロセッサのバージョンを設定します。
.wmsg *string*	ユーザ定義の警告メッセージを出力デバイスに送ります。

4.2 セクションを定義する疑似命令

次の 6 つの疑似命令は、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す。

- **.asect** は、絶対アドレスを持つ初期化された名前付きセクションを作成します。.asect を使って定義したセクションには、コードまたはデータを入れることができます（絶対セクション内では、.label 疑似命令を使って再配置可能ラベルを定義できます）。
- **.bss** は、.bss セクションに初期化されていない変数用の空間を確保します。
- **.data** は、.data セクションのコード部分を識別します。.data セクションには、通常は初期化されたデータが含まれています。
- **.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、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 セクションには、通常は実行可能コードが含まれています。
- **.usect** は、初期化されない名前付きセクションに空間を確保します。.usect 疑似命令は .bss 疑似命令と機能が似ていますが、.bss セクションとは別に空間を確保することを可能にします。

第 2 章では、COFF セクションについて詳しく説明しています。

例 4 - 1 は、セクション疑似命令を使っ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。これは出力リストです。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、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かのように、カウントを再開します。

以下に示す各セクションで、例 4 - 1 の疑似命令が指示されている作業を実行します。

.text	値 1、2、3、4、5、6、7、および 8 を持つワードを初期化します。
.data	値 9、10、11、12、13、14、15、および 16 を持つワードを初期化します。
var_defs	値 17 および 18 を持つワードを初期化します。
.bss	19 ワードを確保します。
xy	20 ワードを確保します。

.bss 疑似命令と .usect 疑似命令は、現行のセクションを終了したり新規セクションを開始したりはしません。この2つの疑似命令は指定量の空間を確保するもので、その後で、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 0000                    .text
 5 0000 0001               .word     1, 2
   0001 0002
 6 0002 0003               .word     3, 4
   0003 0004
 7
 8                ******************************************
 9                *  Start assembling into the .data section  *
10                ******************************************
11 0000                    .data
12 0000 0009               .word     9, 10
   0001 000A
13 0002 000B               .word     11, 12
   0003 000C
14
15                ******************************************
16                *  Start assembling into a named, initial-  *
17                *  ized section, var_defs                   *
18                ******************************************
19 0000                    .sect     ”var_defs”
20 0000 0011               .word     17, 18
   0001 0012
21
22                ******************************************
23                *  Resume assembling into the .data section *
24                ******************************************
25 0004                    .data
26 0004 000D               .word     13, 14
   0005 000E
27 0000                    .bss      sym, 19    ; Reserve space
28                                              ; in .bss
29 0006 000F               .word     15, 16     ; Still in .data
   0007 0010
30
31                ******************************************
32                **  Resume assembling into the .text section *
33                ******************************************
34 0004                    .text
35 0004 0005               .word     5, 6
   0005 0006
36 0000           usym     .usect    ”xy”,20    ; Reserve space
37                                              ; in xy
38 0006 0007               .word     7, 8       ; Still in .text
   0007 0008
```

4.3 定数を初期化する疑似命令

以下の疑似命令は、値を現行のセクションにアセンブルします。

- **.bes** 疑似命令と **.space** 疑似命令は、指定数のビット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 で割ることによって、指定数のワードを確保することができます。

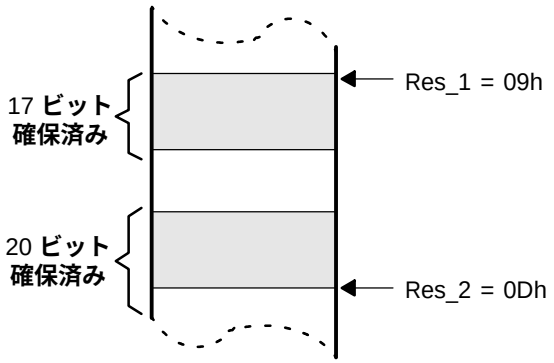
  - .space でラベルを使うと、そのラベルは確保済みのビットを含む最初のワードを指します。
  - .bes でラベルを使うと、そのラベルは確保済みのビットを含む最後のワードを指します。

  図 4 - 1 に .space 疑似命令と .bes 疑似命令を示します。この例では、以下のコードをアセンブルしています。

```
45 0007  0100           .word 100h,200h
   0008  0200
46 0009        Res_1:   .space 17
47 000B  000F           .word 15
48 000C        Res_2:   .bes  20
49 000E  00BA           .byte 0BAh
```

  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。Res_2 は、.bes により確保された空間の最後のワードを指します。

図 4 - 1. .space **疑似命令と** .bes **疑似命令**



- **.byte** 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入れます。この疑似命令は .word 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 つの値を現行のワードの指定数のビットに入れます。.field を使えば、複数のフィールドを単一のワードにパックすることができます。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。

図 4 - 2 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。SPC は変わらない（つまりフィールドが同じワードにパックされる）ことに注意して下さい。

```
1  0000  0003        .field   3,3
2  0000  0008        .field   8,6
3  0000  0010        .field   16,5
```

図 4 - 2.　.field **疑似命令**

- **.float** と **.bfloat** は、1 つの浮動小数点値の単精度 32 ビット IEEE 浮動小数点表記を計算し、その結果を現行セクションの連続する 2 ワードに格納します。最下位ワードが最初に格納されます。.bfloat 疑似命令を使うと、オブジェクトがページ境界にまたがることはありません。

- **.int** と **.word** は、1 つ以上の 16 ビット値を現行のセクションの連続するワードに入れます。

- **.long** と **.blong** は、32 ビット値を現行のセクションの連続する 2 ワード・ブロックに入れます。最下位ワードが最初に格納されます。.blong 疑似命令を使うと、オブジェクトがページ境界にまたがることはありません。

- **.string** は、1 つ以上の文字列から取り出した 8 ビット文字を現行のセクションに入れます。この疑似命令は .byte に似ていますが、各ワードに 2 文字がパックされる点が異なります。必要があれば、文字列内の最後のワードにヌル文字 (0) を埋め込みます。

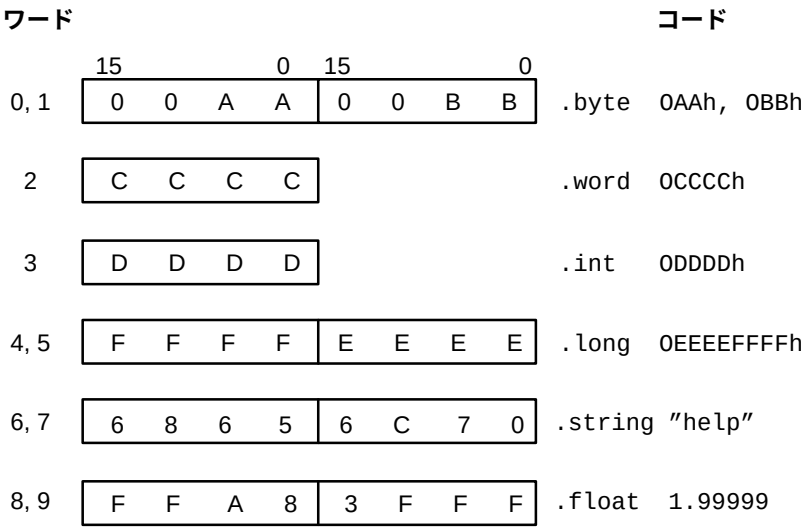
注： .struct/.endstruct シーケンスにおける .byte, .word, .int, .long, .string, .float, .field 疑似命令の機能

.byte、.word、.int、.long、.string、.float、.field 疑似命令は、.struct/.endstruct シーケンスの一部として使われているときにはメモリを初期化しません。単にメンバのサイズを定義するだけです。.struct/.endstruct 疑似命令に関する詳しい説明は 4.8 節を参照して下さい。

図 4 - 3 では、.byte、.int、.long、.float、.word、.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   0000     aa             .byte     0AAh, 0BBh
    0001     bb
2   0002   cccc             .word     0CCCCh
3   0003   dddd             .int      0DDDDh
4   0004   ffff             .long     0EEEEFFFFh
    0005   eeee
5   0006   6865             .string   ”help”
    0007   6c70
6   0008   ffa8             .float    1.99999
    0009   3fff
```

図 4 - 3. 初期化疑似命令

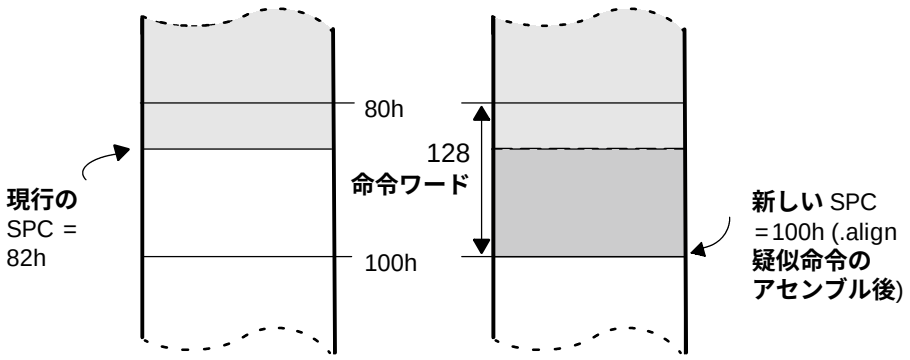


4.4 セクション・プログラム・カウンタの位置合わせを行う疑似命令

❑ **.align** 疑似命令は、128ワード境界上に SPC の位置合わせを行います。その結果、.align 疑似命令の後に続くコードは必ずページ境界から始まります。SPC がすでにページ境界上に位置合わせされている場合は、SPC 値はインクリメントされません。図 4 - 4は、.align 疑似命令を示しています。この例では、以下のコードをアセンブルしています。

```
1  0000  0004     .byte    4
2  0080            .align
3  0080  4572     .string  "Error"
   0081  726F
   0082  7200
4  0100            .align
5  0100  0004     .byte    4
```

図 4 - 4. .align **疑似命令**

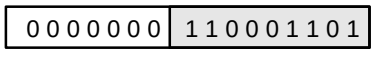


❑ **.even** 疑似命令は、SPC が次のフルワードを指すように位置合わせを行います。.field 疑似命令を使った後では .even を使う必要があります。.field 疑似命令がワードを完全に埋めなかった場合、.even 疑似命令は、アセンブラがフル・ワードを書き出し、未使用ビットに 0 を埋め込むようにします。

図 4 - 5 は、.field 疑似命令の後で .even 疑似命令をアセンブルした場合を示しています。この例では、以下のコードをアセンブルしています。

```
11 0005  0001        .field   1,2
12 0005  0003        .field   3,4
13 0005  0006        .field   6,3
14 0006              .even
```

図 4 - 5. .even **疑似命令**



.even 疑似命令をアセンブルするとこの範囲のビットには 0 が埋め込まれます。

この範囲のビットは .field 疑似命令により埋められています。

4.5 出力リストをフォーマットする疑似命令

リスト・ファイルをフォーマットする疑似命令には次のようなものがあります。

- **.drlist** 疑似命令と **.drnolist** 疑似命令は、リスト・ファイルへの疑似命令行の出力を認めたり、禁止したりします。.drnolist 疑似命令を使うと、以下の疑似命令の出力を禁止することができます。
  .asg、.eval、.var、.sslist、.mlist、.fclist、.ssnolist、.mnolist、.fcnolist、.emsg、.wmsg、.mmsg、.length、.width、.break。
  リスト作成を再開させるには、.drlist 疑似命令を使います。

- ソース・コードには、コードを生成しない、偽の条件付きブロックのリストが含まれています。**.fclist** と **.fcnolist** 疑似命令は、これのリスト出力を禁止したり、認めたりします。.fclist 疑似命令を使うと、ソース・コードどおりに偽の条件付きブロックのリストを作成できます。.fcnolist 疑似命令を使うと、実際にアセンブル対象となる条件付きブロックのみのリストを作成することができます。

- **.length** 疑似命令は、リスト・ファイルのページの長さを制御します。この疑似命令を使ってさまざまな出力デバイス用にリストを調整することができます。

- **.list** と **.nolist** 疑似命令は、リスト・ファイルへの出力を認めたり、禁止したりします。.nolist 疑似命令を使うと、アセンブラに選択したソース文のリスト・ファイルへの出力を中止させることができます。.list 疑似命令を使うと、リスト作成を再開させることができます。

- **.mlist** と **.mnolist** 疑似命令は、マクロ展開リストとループ・ブロック・リストの作成を認めたり、禁止したりします。.mlist 疑似命令を使うと、すべてのマクロ展開とループ・ブロックをリストに出力させることができます。

- **.option** 疑似命令は、リスト・ファイル作成のためのいくつかの機能を制御します。この疑似命令には次のオペランドを指定することができます。

  B .byte 疑似命令によるリスト作成を 1 行に制限します。

  D .int と .word 疑似命令によるリスト作成を 1 行に制限します。

  F B、D、L、M、T オプションの設定をリセットします。

  L .long 疑似命令によるリスト作成を 1 行に制限します。

  M リスト作成のマクロ展開を禁止します。

  T .string 疑似命令によるリスト作成を 1 行に制限します。

  X シンボルのクロスリファレンス・リストを作成します（アセンブラを呼び出すときに -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** 疑似命令を使うと、出力リストのページ替えを行うことができます。

- **.sslist と .ssnolist 疑似命令を使うと、展開された置換シンボルのリスト作成を認めたり、禁止したりできます。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なうときに便利です。**
- **.tab 疑似命令を使うと、タブ・サイズを定義することができます。**
- **.title 疑似命令を使うと、アセンブラが各ページの 1 行目に出力するタイトルを設定できます。**
- **.width 疑似命令は、リスト・ファイルのページの幅を制御します。この疑似命令を使ってさまざまな出力デバイス用にリストを調整することができます。**

4.6 他のファイルを参照する疑似命令

以下の疑似命令は、他のファイルのための、または他のファイルについての情報を提供します。

- **.copy** と **.include** 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、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。

- **.def** 疑似命令は、現行のモジュールで定義されていて、他のモジュールで使うことが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て、リンク時に他のモジュールが使えるようにします (外部シンボルの詳しい説明については、 2-21 ページの 2.7.1 項を参照して下さい) 。 .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、定義されないシンボルに対しては .ref として機能するという2 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のみ、そのグローバル・シンボルを解決します。

- **.mlib** 疑似命令は、アセンブラに、マクロ定義を含むアーカイブ・ライブラリ名を提供します。アセンブラは、現行のモジュールで定義されていないマクロを検出すると、.mlib に指定されたマクロ・ライブラリにそのマクロがあるかどうかを検索します。

- **.ref** 疑似命令は、他のモジュールで定義されていて、現行のモジュールで使われてい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。

4.7 条件アセンブリ疑似命令

条件アセンブリ疑似命令を使うと、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 組の疑似命令を使って条件付きのコード・ブロックをアセンブルすることができます。

- **.if/.elseif/.else/.endif** 疑似命令は、アセンブラに対して式の評価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。

.if expression	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 expression	.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評価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。

.loop expression	繰返し使用可能なコード・ブロックの始まりにマークを付けます。
.break expression	.break 式が偽の場合に、アセンブルの繰り返しを継続します。式が真の場合には、.endloop 直後のコードに行きます。
.endloop	繰返し使用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、条件式に対して有用な関係演算子をいくつかサポートしています。関係演算子についての詳しい説明は、3-24 ページの 3.10.4 項を参照して下さい。

4.8 アセンブリ時シンボル疑似命令

アセンブリ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、テキスト文字列を置換シンボルに割り当てます。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
```

- **.set** 疑似命令と **.equ** 疑似命令は、定数値をシンボルに設定します。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。例を示します。

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

.set 疑似命令と .equ 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、互いに取り替えて使うことができます。

- **.struct/.endstruct** 疑似命令は、C に似た構造体を定義し、**.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。

C に似た構造体定義を使うと、類似の要素をまとめてグループ化することができます。そうすれば、要素オフセットの計算をアセンブラに任せることができます。.struct/.endstruct 疑似命令は、メモリの割り振りは行いません。単に、繰り返し使用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 疑似命令によりラベルに構造体特性を割り当てることで、シンボル表記が簡略化され、他の構造体を包含する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振りは行いません。構造体タグ（stag）を使うには、まず最初にそれを定義しておく必要があります。

```
type   .struct             ; structure tag definition
x      .int
y      .int
t_len  .endstruct

coord  .tag   type         ; declare coord (coordinate)

add    coord.y

       .bss   coord, t_len ; actual memory allocation
```

- **.eval 疑似命令は、式を評価し、その結果をテキスト文字列に変換し、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、カウンタの操作を行うときに非常に便利です。例を示します。**

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

- **.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、形式 $n を持つシンボル（n は 10 進数）で、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、ジャンプ命令用のオペランドとして使用できるテンポラリ・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、それによってラベルの有効範囲を制限します（ローカル・ラベルの詳しい説明については、**3-21 **ページの** 3.9.6 **項を参照してください）。**

4.9 その他の疑似命令

以下の疑似命令はその他の雑多な機能を実行します。

- **.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、EOF（ファイルの終わり）と同じ働きをします。

- **.label** 疑似命令は、現行のセクション内で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。これは、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便利です。たとえば、パフォーマンスが重要なブロックをメモリ空間節約のために低速のオフチップ・メモリにロードしておき、実行時にはそれを高速メモリに移すことができます。

- **.mmregs** 疑似命令は、メモリマップド・レジスタ用のシンボル名を定義します。.mmregs を使うのは、すべてのメモリマップド・レジスタに対して .set（たとえば greg .set 5）を実行するのと同じで、個々のシンボルを定義する必要がなくなります。メモリマップド・レジスタのリストについては、4 - 58 ページの表 4 - 2 を参照してください。

- **.port** 疑似命令は、アセンブラの移植スイッチをオンにします。これは、 - p コマンド行オプション（3-6 ページの 3.4 節を参照）を使うのと同じです。
  TMS320C5x 以外のターゲットに対するアセンブルに .port 疑似命令を使っても、効力はありません。

- **.sbloc**疑似命令は、ブロッキング用のセクションを指定します。ブロッキングは、ページ位置合せに似た（ただしそれより弱い）機能を持つアドレス位置合せメカニズムです。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。.sblock疑似命令では、初期化されたセクションのみについてブロッキングを指定できます。.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については指定できません。セクション名は引用符で囲んでも囲まなくてもかまいません。

- **.version** 疑似命令は、 - v アセンブラ・オプションと同じ機能を果たします。つまり、アセンブラに対してコードがどの世代のプロセッサ用であるかを指定します。有効な世代は、10、16、20、25、29、50 です。デフォルトは 25 です。
  .version 疑似命令は、他のすべての命令や疑似命令よりも前に指定しなければなりません。そうしないとエラーが発生します。

以下の 3 つの疑似命令を使うと、ユーザは独自のエラー・メッセージや警告メッセージを作成できます。

- **.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、アセンブラが行うのと同じようにエラーを発生させます。つまり、エラー・カウントを** 1 **つずつ増加させ、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 疑似命令は、アセンブリ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 **および** .wmsg **疑似命令と同じ機能を果たしますが、エラー・カウントや警告カウントを設定せず、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、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、警告カウントを増加させる点と、アセンブラによるオブジェクト・ファイル作成を禁止しない点が異なります。**

4.10 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 ページにまとめることもあります。以下にアルファベット順に並べた疑似命令のリファレンスの目次を掲載します。

構文　　**.align**

説明　　**.align 疑似命令は、セクション・プログラム・カウンタ (SPC) を次の 128 ワード境界に位置合わせします。これにより、後続のコードは必ずページ境界から開始されます。アセンブラは、NOP を次の 128 ワード境界に達するまで埋め込んでアセンブルします。この疑似命令は、ページ境界上にコードを位置合わせするときに便利です。**

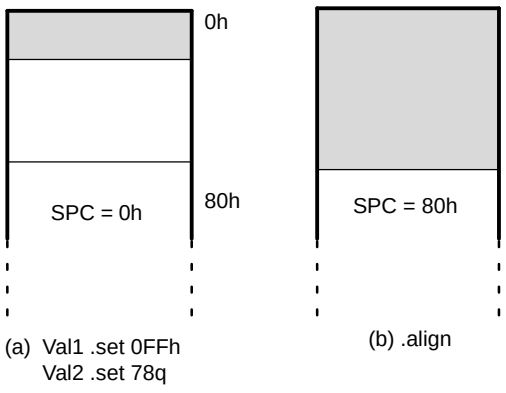
.align **疑似命令を使う目的は 2 つあります。**

- **アセンブラは、SPCを現行のセクションの中の 128 ワード境界に位置合わせします。**
- **アセンブラはリンカにセクションの位置合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るときに個々の位置合せに影響が及ばないようになります。**

例　　**ここに示した例では、SPC を次の 128 ワード境界に位置合せして、SPC の後に続くコードがメモリの新しいページから始まるようにします。図 4 - 6 は、このコードが SPC をどのように位置合せするかを示しています。**

```
 1   0000                  .data
 2         00FF  Val1      .set    0FFh
 3         0076  Val2      .set    76q
 4         0001  PP1       .set    1
 5         0002  PP2       .set    2
 6         0003  PP3       .set    3
 7               ;          .
 8               ;          .
 9               ;          .
10   0080                  .align
11   0080  5202  Page_2:   LDP     PP2
12   0081  207F            LAC     Val1
13   0082  0076            ADD     Val2
```

図 4 - 6.　.align **疑似命令**



構文

```
.asect　"section name", address
```

説明

.asect 疑似命令は、絶対的なアドレス *address* をもつ名前付きセクションを定義します。この疑似命令は、すでに使われていません。現在では、どのセクションでも、リンカを使って 2 つの別々のアドレスにロードして実行することができます。この疑似命令は、互換性を維持する目的で提供されています。

- *section name* は、絶対セクションの名前を識別します。*section name* は、二重引用符で囲む必要があります。
- *address* は、ターゲット・メモリ内でのセクションの絶対開始アドレスを識別します。特定の絶対セクションに初めてアセンブルするときには、このアドレスを必ず指定しなければなりません。.asect を使って、すでにコードが含まれている絶対セクションへのアセンブルを続ける場合には、address パラメータを使ってはいけません。

絶対セクションは、オフチップ・メモリからより高速なオンチップ・メモリにコードのセクションをロードするときに使うと便利です。絶対セクションを使うためには、そのセクションをどの位置から実行するのかが分かっていて、その位置を address パラメータに指定することが必要です。

ほとんどのセクション疑似命令は、再配置可能アドレスを持つセクションを作成します。これらのセクションの開始 SPC 値は必ずゼロです。リンカがこれらのセクションを適切な位置に再配置します。ただし、絶対セクションの開始 SPC 値は、*address* に指定された値となります。絶対セクションにアセンブルされたすべてのコードのアドレスは、指定されたアドレスからのオフセットとなります。リンカは、.asect を使って定義したセクションを再配置します。しかし、絶対セクションの中で定義されたラベルはすべて絶対（実行時）アドレスを保持します。したがって、これらのラベルに対する参照は、そのセクションが初めからその実行アドレスにロードされていない場合でも実行アドレスへの参照となります。

絶対セクションにあるすべてのラベルは、絶対アドレスを持っています。.label 疑似命令は再配置可能アドレスを持ったラベルを作成します。したがって、そのセクションのオフチップ・メモリにおけるロード時の位置を指すシンボルを定義することができます。

.asect を使ってセクションを定義すると、そのプログラム内で .sect 疑似命令を使って絶対セクションへのコードのアセンブルを続けることができます

注：　.asect 疑似命令はすでに使われていません。

現在ではリンカの SECTIONS 疑似命令を使ってすべてのセクションのロード・アドレスと実行アドレスを分けることができるので、.asect 疑似命令は使う必要がなくなりました。.asect 疑似命令は、アセンブラの以前のバージョンとの互換性を維持する目的で現在でも以前と同様に提供されています。詳しい説明については、8-24 ページの8.7 節を参照してください。

構文

.asg [”]*character string*[”]**,** *substitution symbol*

.eval *well-defined expression*, *substitution symbol*

説明

.asg 疑似命令は、文字列を置換シンボルに割り当てます。使い方はほとんど .set 疑似命令と同じですが、.set がシンボルに定数値（再定義不可能）を割り当てるのに対して、.asg は置換シンボルに文字列（再定義可能）を割り当てます。

- 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。引用符がない場合には、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去ります。どちらの場合も、文字列が読み取られ、置換シンボルに割り当てられます。
- *substitution symbol* は必須パラメータで、有効なシンボル名を指定する必要があります。置換シンボルの長さは 32 文字まで指定でき、先頭は文字にしなければなりません。シンボルの残り部分には、英数字、下線、ドル記号を組み合わせて指定できます。
- *well - defined expression* (整合定義式) は、.eval 疑似命令の必須パラメータです。整合定義式は、式に使う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だけを含んでいる式です。

.eval 疑似命令は、置換シンボルに対して算術演算を実行します。この疑似命令は、式を評価し、その結果得られた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 ブロックでカウンタとして使うと特に便利です。

例　　以下に .asg と .eval の使用例を示します。

```
  1                 .sslist ; show expanded sub. symbols
  2
  3              *
  4              *  .asg/.eval example
  5              *
  6                 .asg      *+,   INC
  7                 .asg      ar0,  FP
  8
  9   0000  cc64    add       #100
 10   0001  55a8    mar       INC,  FP
#                   mar       *+,   ar0
 11
 12                 .asg      0,    x
 13                 .loop     5
 14                 .eval     x+1,  x
 15                 .word     x
 16                 .endloop
1                   .eval     x+1,  x
#                   .eval     0+1,  x
1     0002  0001    .word     x
#                   .word     1
1                   .eval     x+1,  x
#                   .eval     1+1,  x
1     0003  0002    .word     x
#                   .word     2
1                   .eval     x+1,  x
#                   .eval     2+1,  x
1     0004  0003    .word     x
#                   .word     3
1                   .eval     x+1,  x
#                   .eval     3+1,  x
1     0005  0004    .word     x
#                   .word     4
1                   .eval     x+1,  x
#                   .eval     4+1,  x
1     0006  0005    .word     x
#                   .word     5
```

構文

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

説明

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

- ❑ *symbol* **は、必須パラメータです。疑似命令が確保する最初の位置を指すシンボルを定義します。このシンボル名は、空間を予約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* **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 size に指定されたワード数を割り当てます。デフォルトのサイズは決まっていません。**
- ❑ *blocking flag* **は、任意のパラメータです。このパラメータに 0 より大きい値を指定すると、アセンブラは連続した size ワードを割り振ります。つまり、size が 1 ページを越えていない限り、割り振られた空間がページ境界にまたがることはないということです。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。**

.bss セクションに空間を割り振るとき、アセンブラは次の 2 つの規則に従います。

規則 1　**メモリに（図 4 - 7 に示すように）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振ろうとします（これは、blocking flag 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、要求された空間を入れられるだけの大きさをもつホールを見つけることができなかった場合は、blocking flag 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。**

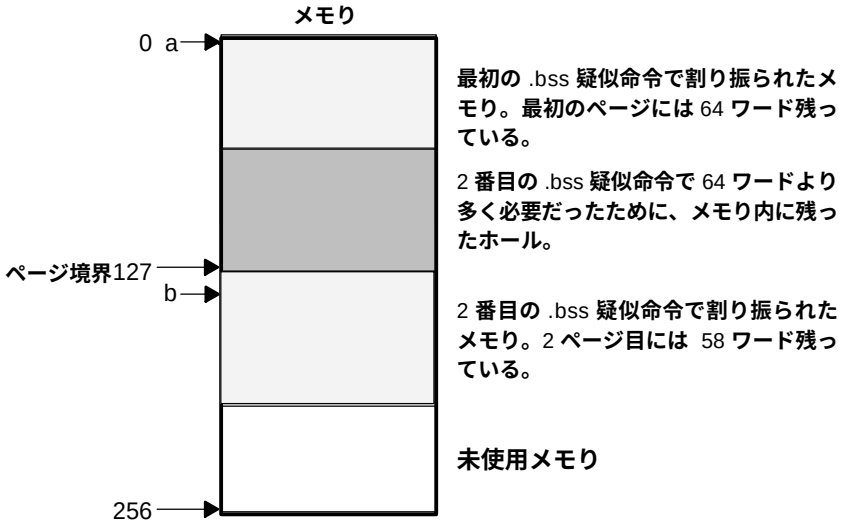
- ❑ **ブロック化が指定されていなければ、メモリは現行の SPC に割り振られます。**
- ❑ **ブロック化が指定されていれば、アセンブラは、現行の SPC からページ境界までに十分な空間があるかどうかを調べます。ここに十分な空間がなければ、アセンブラはそこに別のホールを作成し、次のページに空間を割り振ります。**

ブロック化オプションを使うと、.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することができます（もちろん、一度に 128 ワードより多く確保することもできますが、その場合は 1 つのページには収まりません）。以下のコード例では、.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 つの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振った後では、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 - 7 に示すように、2 番目のブロックは次のページに割り振られます。

図 4 - 7.　**ページ内部への** .bss **ブロックの割り振り**



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、および .asect）は、現行のセクションを終了させ、アセンブラに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。しかし、初期化されないセクション用のセクション疑似命令（.bss および .usect）は、現行のセクションには影響を及ぼしません。アセンブラは、.bss疑似命令または.usect疑似命令をアセンブルし、その後で現行のセクションへのコードのアセンブルを再開します。COFF セクションについての詳しい説明は、第 2 章を参照して下さい。

例

この例では、.bss 疑似命令を使って、2 つの変数 temp と array のための空間を割り振ります。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。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、ページ内部に連続して割り振る必要があります。.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、外部で宣言することもできます。

```
 1                ************************************
 2                *    Begin assembling into .text     *
 3                ************************************
 4  0000                 .text
 5  0000   CA00          LACK      0
 6
 7                ************************************
 8                *  Allocate 4 words in .bss for the  *
 9                *  symbol named temp                 *
10                ************************************
11  0000          Var_1: .bss     temp, 4
12
13                ************************************
14                *           Still in .text           *
15                ************************************
16  0001   0056          ADD       56h
17  0002   3873          MPY       73h
18
19                ************************************
20                * Allocate 100 words in .bss for the *
21                * symbol named array; this part of   *
22                * bss must fit on a single page      *
23               ************************************
24  0004                 .bss      array, 100 , 1
25
26                ************************************
27                *           Still in .text           *
28                ************************************
29  0003   6000          SACL      Var_1
30
31                ************************************
32                *   Declare external .bss symbols    *
33                ************************************
34                       .global   array, temp
```

.byte　バイトの初期化

構文

.byte *value*$_1$ [, ... , *value*$_n$]

説明

.byte 疑似命令は、現行のセクションにある連続するワードに 1 つ以上の 8 ビット値を入れます。各 *value* 値は、以下のいずれかです。

- ❏ **アセンブラが評価して、8 ビットの符号付き数として取り扱う式、または**
- ❏ **二重引用符で囲まれた文字列。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。各バイトは、完全な 16 ビットワードの最下位の 8 ビットを使います。アセンブラは、8ビットよりも大きい値は切り捨てます。1 つの .byte 命令に対して値を 100 個まで定義できますが、ラベル（使用されている場合）、この疑似命令、およびすべての値を含めた行の長さは 200 文字までです。文字列の中の各文字は、別々のオペランドとしてカウントされます。

ラベルを使う場合、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。

.struct/.endstruct シーケンスの中で .byte を使うと、.byte はメンバのサイズを定義します。メモリの初期化は行いません。.struct/.endstruct の詳しい説明については、4.8 節を参照してください。

例

以下の例では、8 ビット値の 10、 - 1、abc、および a をメモリ内の 連続するワードに入れます。ラベル strx の値は 100 h で、これは初期化されたワードの先頭位置を表します。

```
0   0000                .space   100h * 16
1   0100   0A   strx     .byte    10,-1,"abc",'a'
       0101   FF
       0102   61
       0103   62
       0104   63
       0105   61
```

構文

```
.copy     ["]filename["]
.include  ["]filename["]
```

説明

.copy および **.include** 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、アセンブルされた .list/.nolist 疑似命令の数にかかわらずアセンブリ・リストには出力されません。アセンブラは以下の機能を果たします。

1) 現行のソース・ファイルの文のアセンブルを中止する。

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する。

3) メイン・ソース・ファイルの文のアセンブルを、.copy または .include 疑似命令に続く文から再開する。

filename は、ソース・ファイルの名前を指定するための必須パラメータです。*filename* は、二重引用符で囲むこともできます。*filename* は、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。完全なパス名を指定することもできます（c:\320\file1.asm など）。完全なパス名を指定しない場合、アセンブラは以下の場所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 オプションと環境変数についての詳しい説明は、3-11 ページの 3.5 節を参照して下さい。

.copy および .include 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することができます。アセンブラは、このようなネストを 10 段階まで許容します。さらに、ホスト・オペレーティング・システムが別の制限を設定している場合もあります。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 疑似命令を使っ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、現行のファイルへのアセンブルを再開します。

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)
.space 29 **.copy "byte.asm"** **Back in original file .string "done"	** In byte.asm .byte 32,1+ 'A' **.copy "word.asm"** ** Back in byte.asm .byte 67h + 3q	** In word.asm .word 0ABCDh, 56q

リスト・ファイル：

```
    1  0000                .space  29
    2                      .copy   "byte.asm"
A   1                   ** In byte.asm
A   2  0002     20         .byte   32, 1 + 'A'
       0003     42
A   3                      .copy   "word.asm"
B   1                   ** In word.asm
B   2  0004   ABCD         .word   0ABCDh, 56q
       0005   002E
A   4                   ** Back in byte.asm
A   5  0006     6A         .byte   67h+3
    3
    4                   ** Back in original file
    5  0007   446F   .string "Done"
       0008   6E65
```

例 2　　この例では、.include 疑似命令を使っ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、現行のファイルへのアセンブルを再開します。

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)
.space 29 **.include "byte2.asm"** **Back in original file .string "done"	** In byte2.asm .byte 32,1+ 'A' **.include "word2.asm"** ** Back in byte.asm .byte 67h + 3q	** In word2.asm .word 0ABCDh, 56q

リスト・ファイル：

```
1 0000             .space 29
2                  .include "byte2.asm"
3
4                  ** Back in original file
5 0007   446F      .string "Done"
  0008   6E65
```

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となります。**.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われます。**

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、アセンブルの開始時にセクション制御疑似命令を使わない限り、アセンブラはコードを** .text **セクションにアセンブルします。**

COFF **セクションについては、第** 2 **章で詳しく説明しています。**

例

この例では、コードを .data **セクションと** .text **セクションへアセンブルします。**

```
 1                    ****************************************
 2                    **      Reserve space in .data        **
 3                    ****************************************
 4 0000                 .data
 5 0000                 .space    0CCh
 6
 7                    ****************************************
 8                    **        Assemble into .text         **
 9                    ****************************************
10 0000                 .text
11        0000   Index  .set      0
12 0000   2000         LAC       Index
13
14                    ****************************************
15                    **        Assemble into .data         **
16                    ****************************************
17 000D           Table: .data
18 000D   FFFF         .word     -1     ; Assemble 16-bit
19                                      ; constant into .data
20 000E   FF           .byte     0FFh   ; Assemble 8-bit
21                                      ; constant into .data
22
23                    ****************************************
24                    **        Assemble into .text         **
25                    ****************************************
26 0001                 .text
27 0001   000D         ADD       Table
28
29                    ****************************************
30                    **  Resume assembling into .data at   **
31                    **  address 0Fh                       **
32                    ****************************************
33 000F                 .data
```

構文

```
.drlist
.drnolist
```

説明

この 2 つの疑似命令を使うと、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することができます。

.drist 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、以下の疑似命令のリスト・ファイル出力を抑制します。

- ❏ .asg
- ❏ .break
- ❏ .emsg
- ❏ .eval
- ❏ .fclist
- ❏ .fcnolist
- ❏ .length
- ❏ .mlist
- ❏ .mmsg
- ❏ .mnolist
- ❏ .sslist
- ❏ .ssnolist
- ❏ .var
- ❏ .width
- ❏ .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場合と同様に、すべての疑似命令をリスト・ファイルに出力します。

例　この例は、前ページに示した疑似命令の リスト作成を .drnolist で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：

```
*
*   .drlist/.drnolist example
*
        .length 65
        .width  85
        .asg    0, x
        .loop   2
        .eval   x+1, x
        .endloop

        .drnolist

        .length 55
        .width  95
        .asg    1, x
        .loop   3
        .eval   x+1, x
        .endloop
```

リスト・ファイル：

```
        1               *
        2               *      .drlist/.drnolist example
        3               *
        4                          .length 65
        5                          .width  85
        6                          .asg    0, x
        7                          .loop   2
        8                          .eval   x+1, x
        9                          .endloop
1                                  .eval   0+1, x
1                                  .eval   1+1, x
       10
       12
       16                          .loop   3
       17                          .eval   x+1, x
       18                          .endloop
```

構文

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

説明

ユーザ独自のエラー・メッセージと警告メッセージを作成するときには、この 3 つの疑似命令を使います。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、アセンブラが行うのと同様にエラー・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・カウントをインクリメントし、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。
- **.wmsg** 疑似命令は、.emsg 疑似命令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・カウントでなく警告カウント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・ファイルの作成を禁止しません。
- **.mmsg** 疑似命令は、アセンブリ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 および .wmsg 疑似命令と同じ機能を果たしますが、エラー・カウントや警告カウントを設定せず、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、マクロ MSG_EX に最低 1 つのパラメータが必要です。マクロが初めて呼び出されたときにはパラメータ PARAM があるので、正常にアセンブルされます。2 回目は、パラメータがないのでエラーが発行されます。

ソース・ファイル：

```
MSG_EX   .macro    a
         .if       $symlen(a) = 0
         .emsg     "ERROR -- MISSING PARAMETER"
         .else
         ADD       #a
         .endif

         .endm

         MSG_EX    10

         MSG_EX
```

リスト・ファイル：

```
    1 MSG_EX    .macro a
    2           .if   $symlen(a) = 0
    3           .emsg  ”ERROR -- MISSING PARAMETER”
    4           .else
    5           ADD    #a
    6           .endif
    7
    8           .endm
    9
   10 0000      MSG_EX 10
1               .if    $symlen(a) = 0
1               .emsg  ”ERROR -- MISSING PARAMETER”
1               .else
1     0000 cc0a ADD    #10
1               .endif
1
   11
   12 0001      MSG_EX
1               .if    $symlen(a) = 0
1               .emsg  ”ERROR -- MISSING PARAMETER”
 ********* USER ERROR  -  ERROR -- MISSING PARAMETER
1               .else
1               ADD    #a
1               .endif
1

    3           .emsg  ”ERROR -- MISSING PARAMETER”
      1 Error,  No Warnings
```

ファイルのアセンブル時に、次のメッセージが標準出力デバイスに出力されます。

```
”emsg.asm” line 12: ** USER ERROR **
  ERROR -- MISSING PARAMETER


 1 Error,  No Warnings

Errors in source - Assembler Aborted
```

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。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。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、EOF (ファイルの終り) と同じ働きをします。.end を使って、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。

> **注：　マクロを終了するためには .endm を使います**
>
> **マクロを終了するときには .end 疑似命令ではなく、.endm マクロ疑似命令を使って下さい。**

例

以下の例では、アセンブルを終了するときの .end 疑似命令の使用方法を示します。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。

```
1  0000         Start: .space    300
2         000F  temp   .set      15
3  0000                .bss      loc1,48h
4  0013  CE1B          ABS
5  0014  000F          ADD       temp
6  0015  6000          SACL      loc1
7                      .end
```

構文　　**.even**

説明　　**.even** 疑似命令は、セクション・プログラム・カウンタ (SPC) を次のフルワード上に位置合わせします。.field 疑似命令を使うとき、それに続けて .even 疑似命令を使うことができます。これにより、アセンブラは、次のワードのフィールドの初期化を行う前に、一部のみが埋められたワードを書き出すようになります。アセンブラは、未使用のビットを 0 で埋めます。すでに SPC がワード境界にある（部分的に埋められたワードがない）場合には、.even は作用しません。

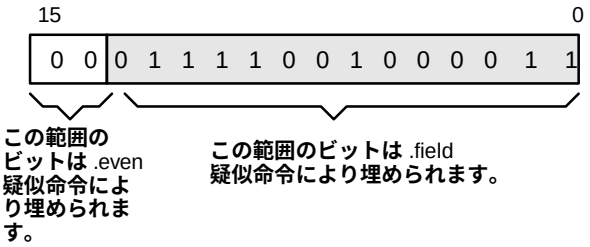
.struct/.endstruct シーケンスで .even を使う場合には、.even は構造体メンバの位置合わせのみを行い、メモリの初期化は行いません。.struct/.endstruct についての詳しい説明は、4-16 ページの 4.8 節を参照して下さい。

例　　この例では、ワード 0 がいくつかのフィールドにより初期化されます。.even 疑似命令により次のフィールドはワード 1 に置かれます。

```
 1                    ********************************
 2                    *   Initialize a 3-bit field   *
 3                    ********************************
 4    0000   0003              .field   03h,3
 5
 6                    ********************************
 7                    *   Initialize a 6-bit field   *
 8                    ********************************
 9    0000   0008              .field   08h,6
10
11                    ********************************
12                    *   Initialize a 5-bit field   *
13                    ********************************
14    0000   000F              .field   0Fh,5
15
16                    ********************************
17                    *      Write out the word      *
18                    ********************************
19    0001                     .even
20
21                    ********************************
22                    *  This field is in the next   *
23                    *            word              *
24                    ********************************
25    0001   0002              .field   02h,2
```

図 4 - 8 は、この例でどのようにワード 0 が初期化されるのかを示しています。最初の 14 ビットは .field 疑似命令により初期化します。残りのビットは .even 疑似命令により 0 に設定されます。

図 4 - 8.　.even 疑似命令



構文

```
.fclist
.fcnolist
```

説明

以下の 2 つの疑似命令を使うと、偽の条件付きブロックのリスト作成を制御することができます。

.fclist 疑似命令を使うと、偽の条件付きブロック (コードを生成しない条件付きブロック) のリストを作成することができます。

.fcnolist 疑似命令は、.fclist 疑似命令を検出するまで、偽の条件付きブロックのリスト作成を禁止します。.fcnolist を使うと、実際にアセンブルを行う条件付きブロック内のコードのみがリストに入ります。.if、.elseif、.else、.endif 疑似命令はリストに入りません。

デフォルトでは、すべての条件付きブロックがリストに出力されます。アセンブラは .fclist 疑似命令が使われている場合と同様の動作をします。

例

以下の例では、アセンブリ言語ファイルと、条件付きブロックのリストを作成したり、作成しなかったりした場合のリスト・ファイルを示します。

ソース・ファイル：

```
a   .set   1
b   .set   0

      .fclist

      .if       a
      addk      #1024
      .else
      adlk      #1024*10
      .endif

      .fcnolist

      .if       b
      addk      #1024
      .else
      adlk      #1024*10
      .endif
```

リスト・ファイル：

```
  1         0001 a     .set    1
  2         0000 b     .set   0
  3                      .fclist
  4
  5                      .if         a
  6   0000  d002         addk        #1024
      0001  0400
  7                      .else
  8                      adlk        #1024*10
  9                      .endif
  10
  11                     .fcnolist
  12
  16  0002  d002         adlk        #1024*10
      0003  2800
```

構文

.field　*value*　[**,** *size in bits*]

説明

.field 疑似命令は、メモリの単一ワード内の複数ビットからなるフィールドを初期化します。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 **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** (1〜16) **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 **ビットであるものとみなします。実際の値が入りきらないビット幅のフィールドを指定すると、アセンブラはその値の切り捨てを行います。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、アセンブラは以下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、最下位ビットから始めて、値を現行のワードの指定したビット数にパックします。アセンブラは現行のワードに入りきらないフィールド・サイズを検出すると、そのワードを書き出し、**SPC **値を増加させ、次のワードへのフィールドのパックを開始します。**.even **疑似命令を使うと、次の** .field **疑似命令に新しいワードへのパックを開始させることができます。**

ラベルを使う場合、ラベルはそのフィールドを含んだワードを指します。

.struct/.endstruct **シーケンスで** .field **を使うと、**.field **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 **についての詳しい説明は、**4-16 **ページの** 4.8 **節を参照してください。**

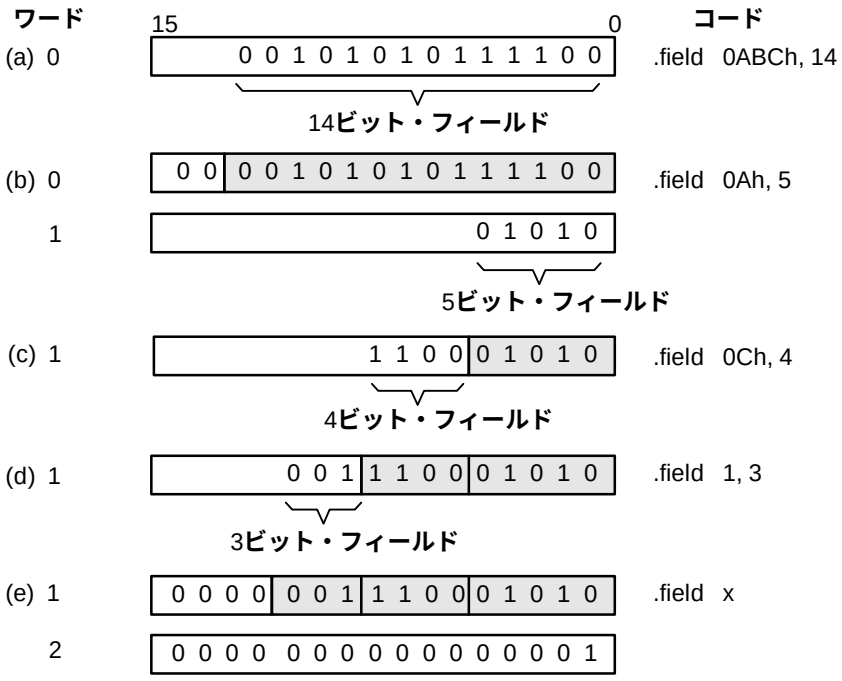
例

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                *********************************
 2                *   Initialize a 14-bit field   *
 3                *********************************
 4   0000   0ABC           .field   0ABCh, 14
 5
 6                *********************************
 7                *   Initialize a 5-bit field    *
 8                *   in a new word               *
 9                *********************************
10   0001   000A   L_F:     .field   0Ah, 5
11
12                *********************************
13                *   Initialize a 4-bit field    *
14                *   in the same word            *
15                *********************************
16   0001   000C           .field   0Ch,4
17
18                *********************************
19                *   Initialize a 3-bit field    *
20                *   in the same word            *
21                *********************************
22   0001   0001   x:       .field   01h,3
23
24                *********************************
25                *  16-bit relocatable field     *
26                *  in the next word             *
27                *********************************
28   0002   0001'           .field   x
```

図 4 - 9 では、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ます。

図 4 - 9.　.field **疑似命令**



構文

.float *value*

.bfloat *value*

説明

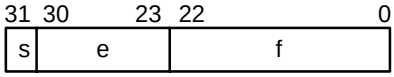
.float 疑似命令と .bfloat 疑似命令は、浮動小数点表記された 1 つの浮動小数点定数を現行のセクションの 2 ワードに入れます。.bfloat **疑似命令を使うと、オブジェクトがページ境界にまたがることはありません。**

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ければなりません。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。**

32 **ビット値は以下の** 3 **つのフィールドから構成されます。**

- 1 **ビットの符号フィールド（**s**）**
- 8 **ビットのバイアスのかかった指数** (e)
- 23 **ビットの仮数部** (f)

値は以下の形式で LSB **を含むワードが最初に格納され、次に** MSB **を含むワードが格納されます。**



.struct/.endstruct **シーケンスで** .float **を使うと、**.float **はメンバーのサイズを定義し、メモリの初期化は行いません。**.struct/.endstruct **についての詳しい説明は、**4-16 **ページの** 4.8 **節を参照して下さい。**

例

.float **および** .bfloat **疑似命令の例を以下に示します。**

```
1   0000   594f        .float     -1.0e25
    0001   e904
2   0002   0000        .bfloat     3
    0003   4040
3   0004   0000        .float     123
    0005   42f6
```

構文

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

説明

.global、.def、.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・シンボルを特定します。

.def 疑似命令は、現行のモジュールで定義され、他のファイ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、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、他のモジュールで定義され、現行のモジュールで使われてい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・シンボルの定義方法は他のシンボルと変わりません。つまり、ラベルとして使われるか、.set、.bss、.usect **疑似命令のいずれかによって定義されます。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うとき以外はシンボル・テーブル・エントリを作成しません。**

シンボルは以下の 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、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なくて済みます。リンク時に、リンカは他のモジュールでそのシンボルが定義されていないか検索します。**
- **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 **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うことが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、他のモジュールから使えるようにしています。どちらのファイルも外部シンボル** x、y、z **を使います。**file1.lst **では、**.global **疑似命令を使ってこれらのグローバル・シンボルを特定します。**file3.lst **では** .ref **と** .def **を使ってこれらのシンボルを特定します。**

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、z **を定義し、他のモジュールから使えるようにしています。どちらのファイルも外部シンボル** Init **を使います。**file2.lst **では、**.global **疑似命令を使ってこれらのグローバル・シンボルを特定します。**file4.lst **では** .ref **と** .def **を使ってこれらのシンボルを特定します。**

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   0000           Init:
 6   0000 0056            ADD       56h
 7   0001 0000            .word     x
 8                  ;       .
 9                  ;       .
10                  ;       .
11                        .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          0001    x:    .set      1
 6          0002    y:    .set      2
 7          0003    z:    .set      3
 8   0000 0000            .word     Init
 9                  ;       .
10                  ;       .
11                  ;       .
12                        .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   0000           Init:
 6   0000 0056            ADD       56h
 7   0001 0000            .word     x
 8                  ;       .
 9                  ;       .
10                  ;       .
11                        .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          0001    x:    .set      1
 6          0002    y:    .set      2
 7          0003    z:    .set      3
 8   0000 0000            .word     Init
 9                  ;       .
10                  ;       .
11                  ;       .
12                        .end
```

構文

```
.if       well-defined expression
.elseif   well-defined expression
.else
.endif
```

説明

この 4 つの疑似命令を使って条件アセンブリを行えます。

.if 疑似命令は、条件ブロックの始まりにマークを付けます。*well - defined expression* は必須パラメータです。

- ❑ 式の評価結果が真（非ゼロ）の場合は、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 のいずれかが検出されるまで）。
- ❑ 式の評価結果が偽 (0) の場合は、アセンブラは .elseif か .else があればそれに続くコードを、どちらもなければ .endif 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（非ゼロ）の場合にアセンブルされるコード・ブロックを特定します。.elseif 式が偽の場合は、アセンブラは（.else があれば）.else まで、または .endif へ処理を進めます。.elseif 疑似命令は、条件ブロックでは任意で、2 つ以上使用できます。式が偽で、.elseif 文がない場合は、アセンブラは（.else があれば）.else、または .endif に続くコードへと処理を続けます。

.else 疑似命令は、.if 式とすべての .elseif 式が偽 (0) の場合にアセンブラがアセンブルするコードのブロックを特定します。この疑似命令は、条件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、条件ブロックを終了します。

.elseif と .else 疑似命令は、同じ条件付きアセンブリ・ブロックで指定することができ、また .elseif 疑似命令は 1 つの条件付きアセンブリ・ブロックの中で 2 回以上指定することができます。

関係演算子についての説明は、3-24 ページの 3.10.4 項を参照して下さい。

例　　条件アセンブリの例を以下に示します。

```
 1       0001   sym1   .set   1
 2       0002   sym2   .set   2
 3       0003   sym3   .set   3
 4       0004   sym4   .set   4
 5              If_4:  .if    sym4 = sym2 * sym2
 6  0000 0004          .byte  sym4   ;  Equal values
 7                     .else
 8                     .byte  sym2 * sym2; Unequal values
 9                     .endif
10              If_5:  .if sym1 <= 10
11  0001 000A          .byte  10     ; Less than/equal
12                     .else
13                     .byte  sym1   ; Greater than
14                     .endif
15              If_6:   .if sym3 * sym2 != sym4 + sym2
16                     .byte  sym3 * sym2 ;Unequal values
17                     .else
18  0002 0006          .byte  sym4 + sym2  ;Equal values
19                     .endif
20              If_7   .if   sym1 = 2
21                     .byte  sym1
22                     .elseif sym2 + sym3 = 5
23  0003 0005          .byte  sym2 + sym3
24                     .endif
```

構文

.int　*value*$_1$ [, ..., *value*$_n$]
.**word**　*value*$_1$ [,... , *value*$_n$]

説明

.int、.word 疑似命令は同等の機能を果たします。どれも 1 つ以上の値を現行のセクションの連続する 16 ビット・フィールドに入れます。

value **には、絶対式か再配置可能な式のいずれかを指定します。式が再配置可能な場合は、アセンブラは該当するシンボルを参照する再配置エントリを生成します。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。ユーザは変数またはラベルを指すポインタを使ってメモリを初期化することができます。**

value **は、1 行に収まる限りいくつでも指定することができます。ラベルを使う場合、ラベルは初期化される最初のワードを指します。**

.struct/.endstruct **シーケンスの中で** .int **または** .word **を使うと、これらの疑似命令はメンバのサイズを定義します。メモリの初期化は行いません。**.struct/.endstruct **の詳しい説明については、**4-16 **ページの** 4.8 **節を参照してください。**

例 1　**この例では、**.int **疑似命令を使ってワードを初期化します。**

```
1  0000                .space 73h
2  0000                .bss   page,128
3  0080                .bss   symptr,3
4  0008 2056   Inst:    LAC   056h
5  0009 000A            .int  10, symptr, -1, 35+'a', Inst
   000A 0080
   000B FFFF
   000C 0084
   000D 0008
```

例 2　**この例では、**.word　**疑似命令を使ってワードを初期化しています。シンボル**　WordX **は、確保される最初のワードを指します。**

```
1  0000 0C80   WordX: .word 3200, 1+'AB', -0AFh, 'X'
   0001 4143
   0002 FF51
   0003 0058
```

.label　再配置可能ラベルの作成

構文

.label　*symbol*

説明

.label 疑似命令は、現行のセクション内の実行アドレスではなくロード・アドレスを参照する特殊な *symbol* を定義します。アセンブラが作成す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、実行されるアドレスに再配置します。

アプリケーションの中には、セクションがロードされるアドレスとは別のアドレスで実行した方が望ましいのもあります。たとえば、パフォーマンスがクリティカルなコードのブロックを速度の遅いオフチップ・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速度の速いオンチップ・メモリに移動するような場合があげられます。

このようなセクションには、リンク時に 2 つのアドレスが割り当てられます。1 つはロード・アドレスで、もう 1 つは実行アドレスです。そのセクションで定義されるすべてのラベルは、コード実行時にそのセクションに対する参照（ブランチなど）が正しくなるように、実行アドレスを参照するように再配置されます。

.label **疑似命令は、ロード・アドレスを参照する特別なラベルを作成します。これは主として、セクションを再配置するコードがそのセクションがどこにロードされたかを知ることができるので便利です。**

例

以下の例は、ロード・アドレス・ラベルの使用法を示しています。

```
;-----------------------------------------
; .label Example
;-----------------------------------------
       .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
```

リンカを使って実行アドレスとロード・アドレスを割り当てる方法についての詳しい説明は、8-32 **ページの** 8.8 **節を参照してください。**

構文

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

説明

.length 疑似命令を使うと、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 **疑似命令を使います。**

- **デフォルト長：** 60 **行**
- **最小長：** 1 **行**
- **最大長：** 32,767 **行**

.width 疑似命令を使うと、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 **疑似命令を使います。**

- **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およ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 **と** .length **疑似命令をリストに出力しません。**

例

以下の例では、ページ長とページ幅をさまざまに変えています。

```
*******************************************
**        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/中止

構文

```
.list
.nolist
```

説明

2 つの疑似命令を使って、ソース・リストの出力を制御することができます。

.list 疑似命令は、ソース・リストが出力できるようにします。

.nolist 疑似命令は、.list 疑似命令を検出するまではソース・リスト出力を抑制します。.nolist 疑似命令を使うと、アセンブル時間を短縮し、ソース・リストのサイズを小さくすることができます。マクロ定義で使うと、マクロ展開のリスト作成を禁止することができます。

アセンブラは、.list や .nolist 疑似命令、また .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、行カウンタは増加していきます。.list および .nolist 疑似命令はネストすることができます。リストを回復するためには各 .nolist につきそれに対応する .list が必要です。

デフォルトでは、ソース・リストがリスト・ファイルに出力されます。アセンブラは、.list 疑似命令が使われている場合と同様の動作をします。

> **注：　リスト・ファイルの作成（-l オプション）**
>
> アセンブラを呼び出すときにリスト・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

以下の例では、.copy 疑似命令を使っ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したときには、コピーされたソース行をリスト・ファイルに入れます。2 回目に .copy を検出したときには、.nolist 疑似命令が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入れません。.nolist、2 回目の .copy、および .list 疑似命令はリスト・ファイルに含まれないことに注意して下さい。また、ソース文がリストされなくても行カウンタは増加することにも注意して下さい。

ソース・ファイル：

```
             .copy      "copy2.asm"
* Back in original file
             .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 0000   0020           .word   32, 1+'A'
       0001   0042
     2                * Back in original file
     6                * Back in original file
     7 0004   446F           .string "Done"
       0005   6E65
```

構文

.long　*value*$_1$ [, ... , *value*$_n$]
.blong　*value*$_1$ [, ... , *value*$_n$]

説明

.long 疑似命令と .blong 疑似命令は、1 つ以上の 32 ビット値を現行のセクションの連続するワードに入れます。最下位ワードが最初に格納されます。.blong 疑似命令を使うと、オブジェクトがページ境界にまたがることはありません。

value **オペランドは、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は、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。ユーザは、変数またはラベルを指すポインタを使ってメモリを初期化することができます。**

値は 100 個まで指定できますが、1 行のソース文の中に収める必要があります。ラベルを使うと、ラベルは初期化された最初のワードを指します。

.struct/.endstruct **シーケンスの中で** .long **を使うと、**.long **はメンバのサイズを定義します。メモリの初期化は行いません。**.struct/.endstruct **の詳しい説明については、**4-16 **ページの** 4.8 **節を参照して下さい。**

例

以下の例は、.long **疑似命令と** .blong **疑似命令がダブル・ワードをどのように初期化するかを示しています。**

```
2  0008   ABCD   Dat1:  .long     0ABCDh,'A'+100h,'g','o'
   0009   0000
   000A   0141
   000B   0000
   000C   0067
   000D   0000
   000E   006F
   000F   0000
3  0010   0008'          .blong     Dat1,0AABBCCDDh
   0011   0000
   0012   CCDD
   0013   AABB
```

.loop/.break/.endloop　コード・ブロックの繰り返しアセンブル

構文

.loop [*well-defined expression*]
.break [*well-defined expression*]
.endloop

説明

上記の 3 つの疑似命令を使うと、コードのブロックを繰り返してアセンブルすることができます。

.loop 疑似命令は、繰り返し使用可能なコード・ブロックを開始します。指定が任意の式には、ループ回数（実行されるループ回数）を指定します。式がない場合は、アセンブラが真（非ゼロ）の式を持った .break 疑似命令か、式を持たない.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 疑似命令とその式の指定は任意です。式が偽 (0) の場合は、ループが継続します。式が真（非ゼロ）、または省略された場合は、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をアセンブルします。

.endloop 疑似命令は、繰返し使用可能なコード・ブロックの終りを示します。この疑似命令は、.break 疑似命令が真（非 ゼロ）のとき、または実行されたループの数が .loop で指定されたループ・カウントに等しいときに実行されます。

例

以下の例は、.loop、.break、および .endloop を .eval 疑似命令とともに使う方法を示しています。最初の 6 行のコードは、そのすぐ後に示されているコードに展開されます。

```
    1              .eval 0, x
    2              coef  .loop
    3                    .word       x*100
    4                    .eval       x+1, x
    5                    .break      x = 6
    6                    .endloop
1       0000   0000       .word       0*100
1                         .eval       0+1, x
1                         .break      1 = 6
1       0001   0064       .word       1*100
1                         .eval       1+1, x
1                         .break      2 = 6
1       0002   00c8       .word       2*100
1                         .eval       2+1, x
1                         .break      3 = 6
1       0003   012c       .word       3*100
1                         .eval       3+1, x
1                         .break      4 = 6
1       0004   0190       .word       4*100
1                         .eval       4+1, x
1                         .break      5 = 6
1       0005   01f4       .word       5*100
1                         .eval       5+1, x
1                         .break      6 = 6
```

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・ライブラリとは、マクロ定義を含んだファイルの集合です。これらのファイルは、アーカイバによって（アーカイブまたはライブラリと呼ばれる）1 つのファイルに結合されます。マクロ・ライブラリの各メンバには、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・ライブラリのメンバは、(オブジェクト・ファイルではなく) ソース・ファイルでなければなりません。

マクロ・ライブラリ・メンバのファイル名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。完全なパス名を指定することもできます (C:\320\macs.lib など) 。完全なパス名を指定しない場合、アセンブラは以下の場所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 A_DIR を使って指定したすべてのディレクトリ**

-i オプションと環境変数についての詳しい説明は、3-11 ページの 3.5 節を参照して下さい。

アセンブラは .mlib 疑似命令を検出すると、ライブラリをオープンし、そ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・エントリとして命令コード・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・コードをリストには出力しません。ライブラリから実際に呼び出されたマクロのみが一度だけ抽出されます。

例

この例では、inc1 および dec1 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 inc1.asm には inc1 の定義が、ファイル dec1.asm には dec1 の定義がそれぞれ入ります。

inc1.asm	dec1.asm
* Macro for incrementing inc1 .MACRO reg LARP reg ADRK 1 .ENDM	* Macro for decrementing dec1 .MACRO reg LARP reg SBRK 1 .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。

```
dspar -a mac inc1.asm dec1.asm
```

これで、.mlib 疑似命令を使ってマクロ・ライブラリを参照し、マクロ inc1 と dec1 を呼び出すことができます。

```
    1   0000          .mlib  "mac.lib"
    2   0000          inc1   AR0       ; Macro call
1       0000   5588   LARP   AR0
1       0001   7E01   ADRK   1
    3   0002          dec1   AR1       ; Macro call
1       0002   5589   LARP   AR1
1       0003   7F01   SBRK   1
```

構文

```
.mlist
.mnolist
```

説明

この 2 つの疑似命令を使ってリスト・ファイルにあるマクロ展開および繰り返して使用可能なブロックの展開リスト作成を制御することができます。

.mlist 疑似命令を使うと、マクロと .loop/.endloop ブロックの展開をリスト・ファイルに出力することができます。

.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、マクロと .loop/.endloop ブロックで検出されたすべてのコードがリストに出力されます。アセンブラは、.mlist 疑似命令が使われている場合と同様の動作をします。

例

以下の例では、str_3 という名前のマクロを定義します。マクロが最初に呼び出されたときは、マクロ展開はリストに出力されます（デフォルト）。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再びリストに出力されます。

```
       1                str_3   .MACRO    p1,p2,p3
       2                        .string   ":p1:",":p2:",":p3:"
       3                        .ENDM
       4
       5 0000                   str_3     "red","green","blue"
1        0000 7265               .string   "red","green","blue"
         0001 6467
         0002 7265
         0003 656e
         0004 626c
         0005 7565
       6
       7                        .mnolist
       8 0006                   str_3     "red","green","blue"
       9
      10                        .mlist
      11 000c                   str_3     "red","green","blue"
1        000c 7265               .string   "red","green","blue"
         000d 6467
         000e 7265
         000f 656e
         0010 626c
         0011 7565
```

構文

```
.mmregs
```

説明

.mmregs 疑似命令は、TMS320 レジスタ用のグローバル・シンボル名を定義し、それをグローバル・シンボル・テーブルに入れます。これは、greg .set 5、imr .set 4 などを実行した場合と同じです。シンボルはローカルの絶対シンボルです。.mmregs 疑似命令を使うと、これらのシンボルを定義する必要がなくなります。

TMS320C50、TMS320C25、または TMS320C2xx を使うと、シンボルの配置は表 4 - 2 に示すようになります。

表4 - 2.　**メモリマップド・レジスタ**

名前	C2x のみ	C2xx のみ	C5x アドレス 10 進	C5x アドレス 16 進	説明
			0-3	0-3	予約領域
IMR	4	4	4	4	割り込みマスク・レジスタ
GREG	5	5	5	5	グローバル・メモリ領域設定レジスタ
IFR		6	6	6	割り込みフラグ・レジスタ
PMST			7	7	プロセッサ・モード・ステータス・レジスタ
RPTC			8	8	リピート・カウンタ・レジスタ
BRCR			9	9	ブロック・リピート・カウンタ・レジスタ
PASR			10	A	ブロック・リピート開始アドレス用レジスタ
PAER			11	B	ブロック・リピート終了アドレス用レジスタ
TREG0			12	C	被乗数格納レジスタ
TREG1			13	D	シフト・カウント・レジスタ
TREG2			14	E	ビット・ポインタ・レジスタ
DBMR			15	F	動的ビット操作レジスタ
AR0			16	10	補助レジスタ 0
AR1			17	11	補助レジスタ 1
AR2			18	12	補助レジスタ 2
AR3			19	13	補助レジスタ 3

表4 - 2.　メモリマップド・レジスタ（続き）

名前	C2x のみ	C2xx のみ	C5x アドレス 10 進	C5x アドレス 16 進	説明
AR4			20	14	補助レジスタ 4
AR5			21	15	補助レジスタ 5
AR6			22	16	補助レジスタ 6
AR7			23	17	補助レジスタ 7
INDX			24	18	インデックス・レジスタ
ARCR			25	19	補助レジスタ比較用レジスタ
CBSR1			26	1A	サーキュラ・バッファ 1 開始レジスタ
CBER1			27	1B	サーキュラ・バッファ 1 終了レジスタ
CBSR2			28	1C	サーキュラ・バッファ 2 開始レジスタ
CBER2			29	1D	サーキュラ・バッファ 2 終了レジスタ
CBCR			30	1E	サーキュラ・バッファ制御レジスタ
BMAR			31	1F	ブロック転送用アドレス・レジスタ
DRR	0	0	32	20	データ受信レジスタ
DXR	1	1	33	21	データ送信レジスタ
SPC			34	22	シリアル・ポート制御レジスタ
			35	23	予約領域
TIM	2	2	36	24	タイマ・レジスタ
PRD	3	3	37	25	期間レジスタ
TCR			38	26	タイマ制御レジスタ
			39	27	予約領域
PDWSR			40	28	プログラム・スイッチ・ウェイト・ステート・レジスタ
IOWSR			41	29	I/O　スイッチ・ウェイト・ステート・レジスタ
CWSR			42	2A	スイッチ・ウェイト・ステート制御レジスタ
			43 - 47	2B - 2F	予約領域

構文

```
.newblock
```

説明

.newblock 疑似命令を使うと、現在定義されているローカル・ラベルの定義を解除することができます。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て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、式の中で使うことはできません。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・ラベルを定義して使用したら、.newblock **疑似命令を使ってリセットします。**.text**、**.data**、**.sect **疑似命令もローカル・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・ラベルはインクルード・ファイルの外では無効です。**

例

この例では、ローカル・ラベル $1 **を宣言し、リセットして、再び宣言する方法を示します。**

```
 1 0000   2000- Label1:lac    a
 2 0001   1001-         sub    b
 3 0002   f380          blz    $1
   0003   0007
 4 0004   2001-         lac    b
 5 0005   ff80           b     $2
   0006   0008
 6 0007   2000-  $1     lac    a
 7 0008   0002-  $2     add    c
 8                     .newblock ; Undefine $1 to use it again
 9 0009   f280          blez   $1
   000a   000c
10 000b   6002-         sacl   c
11 000c   5500   $1     nop
```

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、コンマで区切った一連のオプションを指定します。各オプションはリスト作成機能を選択するために使います。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作成を** 1 **行に制限します。**

D .int **および** .word **疑似命令によるリスト作成を** 1 **行に制限します。**

F B、D、L、M、T **オプションをリセットします。**

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行わないようにします。**

T .string **疑似命令によるリスト作成を** 1 **行に制限します。**

X **シンボルのクロス・リファレンス・リストを作成します。**

オプションは大文字小文字の区別をしません。

例

この例では、.byte、.word、.long、.string **疑似命令によるリスト作成をそれぞれ** 1 **行に制限します。**

```
 1                  ****************************************
 2                  *  Limit the listing of .byte, .int, .word,*
 3                  *  .long, and .string directives to 1 line *
 4                  *  each                                    *
 5                  ****************************************
 6 0000                       .option   B,D,L,T
 7 0000 00BD                  .byte     -'C',0B0h,5
 8 0003 CCDD                  .long     0AABBCCDDh,536+'A'
 9 0007 15AA                  .word     5546,78h
10 0009 0015                  .int      010101b,356q,85
11 000C 4578                  .string   "Extended Registers"
12
13                  ****************************************
14                  *        Reset the listing options       *
15                  ****************************************
16 0015                       .option   f
17 0015   BD                  .byte     -'C',0B0h,5
   0016   B0
   0017   05
18 0018 CCDD                  .long     0AABBCCDDh,536+'A'
   0018 AABB
   001A 0259
   001A 0000
19 001C 15AA                  .word     5546,78h
   001D 0078
20 001E 0015                  .int      010101b,356q,85
   001F 00EE
   0020 0055
21 0021 4578                  .string   "Extended Registers"
   0022 7465
   0023 6E64
   0024 6564
   0025 2052
   0026 6567
   0027 6973
   0028 7465
   0029 7273
```

構文　　**.page**

説明　　**.page** 疑似命令は、リスト・ファイルのページ替えを行います。.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 疑似命令を使ってソース・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　　この例では、.page 疑似命令がアセンブラに対してソース・リストのページ替えをどのように行うかを示します。

ソース・ファイル：

```
        .title   "**** Page Directive Example ****"
;         .
;         .
;         .
        .page
```

リスト・ファイル：

```
DSP COFF Assembler, Version X.XX       Tue Feb 7 1:29:32 1995
Copyright (c)  1987-1995  Texas Instruments Incorporated

**** Page Directive Example ****                                    PAGE    1

   2            ;         .
   3            ;         .
   4            ;         .
DSP COFF Assembler, Version X.XX       Tue Feb 7 1:29:32 1995
Copyright (c)  1987-1995  Texas Instruments Incorporated

**** Page Directive Example ****                                    PAGE    2
```

構文　　**.port**

説明　　**.port** 疑似命令は、アセンブラの移植スイッチをオンにします。これは、 -p コマンド行スイッチ（3-6 ページの 3.4.1 項を参照）を使うのと同じです。TMS320C2xx または TMS320C5x 以外のターゲットについてのアセンブル中に .port 疑似命令を使っても、効力はありません。コマンド行で対応する -v オプションを指定して、TMS320C2xx 用または TMS320C5x 用のどちらのコードを作成するのかをアセンブラに指示して下さい。

例　　以下の例では、'C2x から 'C5x への移植スイッチがオンになります。

```
 1 ****************************************************
 2 **  Turn on C2x to C5x port switch                 **
 3 ****************************************************
 4        .port
 5 0000   b90a      LACK   10
 6 0001   7c01   L1 SBRK   1
 7 0002   90aa      SACL   *+,AR2
 8 0003   90a0      SACL   *+
 9 0004   f304      BGZ L1, AR1
   0005   001'   ; *** DELAYED BRANCH REPLACES PREVIOUS INSTRUCTION ***
   0006   8b09      MAR ; *** UPDATE ARx AS IN PREVIOUS BRANCH ***
   0007   8b00      NOP ; *** INSERTED FOR PIPELINE ALIGNMENT ***
10
11 0008   2080   L2 ADD *
12 0009   f100      BBNZ   L2,AR1
   000a   0008   ; *** DELAYED BRANCH REPLACES PREVIOUS INSTRUCTION ***
   000b   8b09      MAR ; *** UPDATE ARx AS IN PREVIOUS BRANCH ***
   000c   8b00      NOP ; *** INSERTED FOR PIPELINE ALIGNMENT ***
13
14 000d   8b8a      LARP   AR2
15 000e   a080      NORM   *
   000f   8b00      NOP ; *** INSERTED FOR PIPELINE PROTECTION ***
   0010   8b00      NOP ; *** INSERTED FOR PIPELINE PROTECTION ***
16 0011   8213'     SAR 2,x        ; pipeline conflict
17 0012   8280      SAR AR2,*      ; pipeline conflict
18
19 0013   0000   x   .word  00h
```

.sblock　初期化セクションのブロッキング指定

構文

.sblock [”]*section name*[”] [, ”*section name*”, . . .]

説明

.sblock 疑似命令は、ブロック化用のセクションを指定します。ブロック化は、ページ位置合せに似た（ただしそれより弱い）機能を持つアドレス位置合せメカニズムです。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。この疑似命令では、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については指定できません。*section name* は引用符で囲んでも囲まなくてもかまいません。

例

以下の例では、.text 疑似命令と .data 疑似命令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
```

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わ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・コードのアセンブリを開始します。**

sectionname **は、アセンブラがアセンブルするコードを入れるセクションを特定します。指定するセクション名は** 8 **文字まで有効で、必ず引用符で囲みます。**

.asect **疑似命令の機能は** .sect **疑似命令と似ていますが、**.asect **は絶対アドレスを持った名前付きセクションを作成します。**.asect **疑似命令を使って名前付きの絶対セクションを定義した場合は、後でプログラム内で** .sect **疑似命令を使って絶対セクションへのコードのアセンブルを続けることができます。**

COFF **セクションについての詳しい説明は、第** 2 **章を参照して下さい。**

例

以下の例では、2 **つの専用セクション** Sym - Defs **と** 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ます。**

```
 1                ********************************************
 2                **   Begin assembling into .text section    **
 3 0000                     .text
 4 0000 2078                LAC     78h ; Assembled into   .text
 5 0001 0036                ADD     36h ; Assembled into   .text
 6
 7                ********************************************
 8                **   Begin assembling into Sym_Defs section **
 9      0000                .sect   ”Sym_Defs”
10 0000 CCCC                .float  0.05; Assembled into   Sym_Defs
   0001 3D4C
11 0002 00AA X:             .word   0AAh; Assembled into   Sym_Defs
12 0003 0002                ADD     X   ; Assembled into   Sym_Defs
13
14               ********************************************
15               **   Begin assembling into Vars section     **
16 0000                     .sect   ”Vars”
17 0010          Word_Len   .set    16
18 0020          DWord_Len  .set    Word_Len * 2
19 0008          Byte_Len   .set    Word_Len / 2
20 0053          Str        .set    53h
21
22               ********************************************
23               **   Resume assembling into .text section   **
24 0002                     .text
25 0002 0042                ADD     42h ; Assembled into  .text
26 0003   03                .byte   3, 4; Assembled into  .text
   0004   04
27
28               ********************************************
29               **   Resume assembling into Vars section    **
30 0000                     .sect   ”Vars”
31 0000 000D                .field  13, Word_Len
32 0001 000A                .field  0Ah, Byte_Len
33 0001 0008                .field  10q, DWord_Len / 8
```

構文

symbol **.set** *value*
symbol **.equ** *value*

説明

.set および .equ 疑似命令は、定数値をシンボルと等価にします。これにより、そのシンボルをアセンブリ・ソースの中で値の代わりに使うことができます。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。.set **および** .equ **疑似命令は同じ機能を持ち、片方の代わりに他方を使うことができます。**

- *symbol* **は、ラベル・フィールドで指定しなければなりません。**
- *value* **式の中のシンボルはすでに定義されているものでなければなりません。**

未定義の外部シンボル、あるいはモジュールで後で定義されるシンボルを式で使うことはできません。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 **を使って定義されたシンボルは、**.def **または** .global **疑似命令を使って外部から参照することができます。この方法で、グローバルな絶対定数を定義することができます。**

例

以下の例では、.set **と** .equ **を使ったシンボルの割り当て方法を示します。**

```
 1                      ************************************
 2                      * Equate symbol FP to register AR1  *
 3                      *  and use it instead of AR1        *
 4                      ************************************
 5        0001          FP      .set    AR1
 6 0000   3156                  LAR     FP, 56h
 7
 8                      ************************************
 9                      * Set symbol count to an integer    *
10                      * expression and use it as an       *
11                      * immediate operand                 *
12                      ************************************
13        0035          count   .equ    100/2 + 3
14 0001   CC35                  ADDK    count
15
16                      ************************************
17                      * Set symbol symtab to relocatable  *
18                      * expression                        *
19                      ************************************
20 0002   000a          label   .word   10
21        0003'         symtab  .set    label+1
22 0003   0003'                 .word   symtab
23
24                      ************************************
25                      * Set symbol nsyms to another       *
26                      * symbol (count) and use it instead *
27                      * of count                          *
28                      ************************************
29        0035          nsyms   .equ    count
30 0004   0035                  .word   nsyms
```

構文

.space *size in bits*
.bes *size in bits*

説明

.space 疑似命令と .bes 疑似命令は、*size* に指定された数のビットを現行のセクションに確保し、そのワードを 0 で埋めます。セクション・プログラム・カウンタ（SPC）は、確保された空間に続くワードを指すよう増加します。

.space **疑似命令でラベルを使うと、そのラベルは確保された最初のワードを指します。**
.bes **疑似命令でラベルを使うと、そのラベルは確保された最後のワードを指します。**

例

以下の例では、.space **と** .bes **を使ってメモリを確保する方法を示します。**

```
 1                ************************************
 2                *  Begin assembling into .text       *
 3                ************************************
 4     0000               .text
 5
 6                ************************************
 7                *  Reserve 0F0h bits (15 words) in   *
 8                *  the .text section                 *
 9                ************************************
10  0000                  .space    0F0h
11  000F 0100             .word     100h,200h
    0010 0200
12
13                ************************************
14                *  Begin assembling into .data       *
15                ************************************
16  0000                  .data
17  0000 496E             .string   "In .data"
    0001 202E
    0002 6461
    0003 7461
18
19                ************************************
20                *  Reserve 100 bits in the .data     *
21                *  section; Res_1 points to the      *
22                *  first word that contains reserved *
23                *  bits                              *
24                ************************************
25  0004          Res_1:    .space    100
26  000B 000F             .word     15
27  000C 0004             .word     Res_1
28
29              ************************************
30              *  Reserve 20 bits in the .data      *
31              *  section; Res_2 points to the      *
32              *  last word that contains reserved  *
33              *  bits                              *
34              ************************************
35  000E        Res_2:    .bes      20
36  000F 003              .word     36h
37  0010 000E             .word     Res_2
```

構文

```
.sslist
.ssnolist
```

説明

この 2 つの疑似命令を使うと、リスト・ファイルでの置換シンボルの展開を制御することができます。

.sslist 疑似命令は、置換シンボルの展開のリスト・ファイルへの出力を認めます。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、置換シンボルの展開のリスト・ファイルへの出力を禁止します。

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禁止されます。アセンブラは、.ssnolist 疑似命令が使われている場合と同様の動作をします。(#) の付いた行は、置換シンボルが展開されたことを示しています。

例

この例では、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ます。続いて .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・コードの展開をリストに出力するように指示します。

```
   1   0000                .bss       x, 1
   2   0001                .bss       y, 1
   3                ADD2   .macro     a,b
   4                       LAC        a
   5                       ADD        b
   6                       SACL       b
   7                       .endm
   8
   9                       .asg       ”*+”, INC
  10                       .asg       ar0 , FP
  11
  12   0000   5588         LARP       FP
  13   0001   55a0         MAR        INC
  14
  15   0002                ADD2       x,y
1      0002   2000-        LAC        x
1      0003   0001-        ADD        y
1      0004   6001-        SACL       y
  16
  17                       .sslist
  18
  19   0005   5588         LARP       FP
#                          LARP       ar0
  20   0006   55a0         MAR        INC
#                          MAR        *+
  21
  22   0007                ADD2       x,y
1       0007   2000-       LAC        a
#                          LAC        x
1       0008   0001-       ADD        b
#                          ADD        y
1       0009   6001        SACL       b
#                          SACL       y
```

構文

.string　”*string*$_1$”［，... ，”*string*$_n$”］

説明

.string 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入れます。各ワードが 2 つの 8 ビット値を持つようにデータはパックされます。各 *string* には以下のいずれかを指定します。

- **アセンブラが評価し、16 ビットの符号付き数値として取り扱う式、または**
- **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。**

値はワードにパックされます。値はワードの最上位バイトからパックされ、未使用の空間はヌル・バイト (0) で埋められます。.string と .byte 疑似命令の違いは、.byte は値をワードにパックしない点です。

アセンブラは、8 ビットよりも大きい値は切り捨てます。オペランドは 100 個まで指定できますが、1 行のソース文に収めることが必要です。

ラベルを使う場合、ラベルは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 がメンバーのサイズを定義し、メモリは初期化されません。.struct/.endstruct に関する詳しい情報については 4-16 ページの 4.8 節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 ビット値を入れます。

```
1  0000 4142  Str_Ptr:.string ”ABCD”
   0001 4344
2  0002 4142          .string 41h,42h,43h,44h
   0003 4344
3  0004 4175          .string ”Austin”,”SanAntonio”,”Houston”
   0005 7374
   0006 696E
   0007 5361
   0008 6E20
   0009 416E
   000A 746F
   000B 6E69
   000C 6F48
   000D 6F75
   000E 7374
   000F 6F6E
4  0010 3000          .string 36 + 12
```

構文

[*stag*]	**.struct**	[*expr*]
[mem_0]	*element*	[$expr_0$]
[mem_1]	*element*	[$expr_1$]
.	.	.
.	.	.
[mem_n]	**.tag** *stag*	[$expr_n$]
.	.	.
[mem_N]	*element*	[$expr_N$]
[*size*]	**.endstruct**	
label	**.tag**	*stag*

説明

.struct 疑似命令は、シンボリックなオフセットをデータ構造定義の要素に割り振ります。これにより、類似するデータ要素をグループ化し、アセンブラに要素のオフセットを計算するように指示することができます。この方式は、C 構造体または Pascal レコードに似ています。.struct **疑似命令は、メモリの割り当ては行いません。単に繰り返し使用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を終了させます。

.tag 疑似命令は構造体の特性をラベルに付与し、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 **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。この値は、構造体の始まりを示します。**stag **が未指定の場合、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に持つ、としてグローバル・シンボル・テーブルに入れます。**

expr **構造体の始まりのオフセットを示す任意選択の式です。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。**

mem_n **構造体のメンバを示す任意選択のラベルです。このラベルは、絶対的で、構造体の始まりからの現在のオフセットと等価です。メンバ構造体のためのラベルはグローバル宣言ができません。**

element .string、.byte、.word、.float、.tag、.field **疑似命令のうちのいずれかです。**.tag **以外の疑似命令はすべてメモリの初期化によく使われます。** .struct **疑似命令に続いてこれらの疑似命令を指定し、構造体の要素サイズを記述します。どれもメモリの割り当ては行いません。**.tag **疑似命令は、**stag **を指定する必要があるので（定義参照）特別です。**

$expr_n$ **記述される要素数を指定する指定が任意の式です。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 **構造体全体のサイズを示す指定が任意のラベルです。**

注： struct/.endstruct シーケンスで使える疑似命令の種類

.struct/.endstruct **シーケンスで使用可能な疑似命令は、要素記述子、条件付きアセンブリ疑似命令、**.align **疑似命令（メンバ・オフセットをバイト境界に位置合せする）、および** .even **疑似命令（メンバ・オフセットをワード境界に位置合せする）だけです。空の構造体は不正です。**

以下の例では、.struct、.tag、.endstruct 疑似命令のさまざまの使用法をします。

例 1

```
 1          0000   real_rec  .struct                  ; stag
 2          0000   nom       .int                     ; member1  = 0
 3          0001   den       .int                     ; member2  = 1
 4          0002   real_len  .endstruct               ; real_len = 2
 5
 6   0000   2001-             add   real+real_rec.den  ; access structure elem
 7
 8   0000                     .bss  real, real_len     ; allocate
```

例 2

```
 1
 2   0000   0000   cplx_rec  .struct                  ; stag
 3          0000   reali     .tag    real_rec         ; member1  = 0
 4          0002   imagi     .tag    real_rec         ; member2  = 2
 5          0004   cplx_len  .endstruct               ; cplx_len = 4
 6
 7                 complex   .tag    cplx_rec         ; assign structure attrib
 8
 9   0001   2002-             add     complex.reali    ; access structure
10   0002   9002-             sac1    complex.reali
11
12   0003   2005-             add     complex.imagi
13
14   0002                     .bss    complex,cplx_len ; allocate space
```

例 3

```
 1   0000                     .struct                  ; no stag puts memNs into
 2                                                     ; global symbol table
 3   0000     x               .int                     ; create 3 dim templates
 4   0001     y               .int
 5   0002     x               .int
 6   0003                     .endstruct
```

例 4

```
 1          0000   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 = 67
 7          0042   bit_len   .endstruct                       ; length = 68
 8
 9                 bits      .tag     bit_rec
10   0004   2046-             add      bits.bit7       ; move into acc
11   0005   bfb0              add   #007Fh             ; mask off garbage bits
            007f
12
13   0006                     .bss  bits, bit_rec
```

構文

.tab *size*

説明

.tab 疑似命令は、タブの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・サイズは空白文字 8 個分です。

例

以下のそれぞれの行は、タブ文字 1 つとそれに続く NOP 命令から成っています。

```
 1                      ;default tab size
 2 0000 5500            nop
 3 0001 5500            nop
 4 0002 5500            nop
 5                  .tab 4
 6 0003 5500        nop
 7 0004 5500        nop
 8 0005 5500        nop
 9                                 .tab 16
10 0006 5500                       nop
11 0007 5500                       nop
12 0008 5500                       nop
13
```

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。.text セクションには通常は実行可能なコードが含まれます。.text セクションへのアセンブルが行われていない場合は、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は、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 はデフォルトのセクションです。したがって、他のセクション疑似命令 (.data、.sect) を指定しなければ、アセンブリ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しい説明は第 2 章を参照して下さい。

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含まれ、.text セクションには文字列が含まれます。

```
 1                    ****************************************
 2                    *  Begin assembling into .data section  *
 3                    ****************************************
 4  0000                      .data
 5  0000    0A                .byte     0Ah,0Bh
    0001    0B
 6
 7                    ****************************************
 8                    *  Begin assembling into .text section  *
 9                    ****************************************
10  0000                      .text
11  0000  4142        Start:  .string   "A","B","C"
    0001  4300
12  0002  5859        End:    .string   "X","Y","Z"
    0003  5A00
13  0004  0000'               ADD       Start
14  0005  0002'               ADD       End
15
16                    ****************************************
17                    * Resume assembling into .data section  *
18                    ****************************************
19  0002                      .data
20  0002    0C                .byte     0Ch,0Dh
    0003    0D
21
22                    ****************************************
23                    * Resume assembling into .text section  *
24                    ****************************************
25  0006                      .text
26  0006  5175                .string   "Quit"
    0007  6974
```

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 ソース文は出力されませんが、行カウンタは増加します。

***string* には引用符でかこまれたタイトルを 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、最初のソース文を .title 疑似命令にしなければなりません。

例

この例では、1 ページ目に出力されるタイトルとは別のタイトルを後続ページに出力します。

ソース・ファイル：

```
        .title   ”**** Fast Fourier Transforms ****”
;         .
;         .
;         .
        .title   ”**** Floating Point Routines ****”
        .page
```

リスト・ファイル：

```
DSP COFF Assembler, Version X.XX        Tue Feb 7 13:47:261995
Copyright (c)  1987-1995 Texas Instruments Incorporated

 **** Fast Fourier Transforms ****                                    PAGE    1

   2            ;         .
   3            ;         .
   4            ;         .
DSP COFF Assembler, Version X.XX       Tue Feb 7 13:47:261995
Copyright (c)  1987-1995 Texas Instruments Incorporated

 **** Floating Point Routines ****                                    PAGE    2
```

構文

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

説明

.usect 疑似命令は、初期化され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。この疑似命令の機能は、.bss 疑似命令と似ています。どちらもデータのための空間を確保するだけで、内容はありません。ただし、.usect は、.bss セクションとは無関係にメモリのどこにでも置ける追加のセクションを定義します。

- *symbol* は、.usect 疑似命令の呼出しによって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を示します。
- *section name* 値は最初の 8 文字だけが意味を持ちます。*section name* 値は引用符で囲まなければなりません。このパラメータにより、初期化されないセクションの名前を指定します。
- *size* は、*section name* が示すセクションに確保されるワード数を定義する式です。
- *blocking flag* は指定が任意です。0 以外の値を使ってこのフラグを指定した場合は、フラグは、このセクションをブロック化することを表します。ブロック化は、ページ位置合せに似た（ただしそれより弱い）機能を持つアドレス・メカニズムです。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もので、.usect 疑似命令のこのインスタンスで宣言されているオブジェクトには適用されません。

他のセクション（.text、.data、.sect、および .asect）疑似命令は現行のセクションを終了し、アセンブラに対して別の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、.usect と .bss 疑似命令は、現行のセクションには影響を及ぼしません。アセンブラは .usect と .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。

メモリ内の連続した位置に置くことのできる変数を、同じ指定セクションに定義することができます。そのためには、同じセクション名を指定して、.usect 疑似命令を繰り返し使います。

COFF セクションについての詳しい説明は第 2 章を参照して下さい。

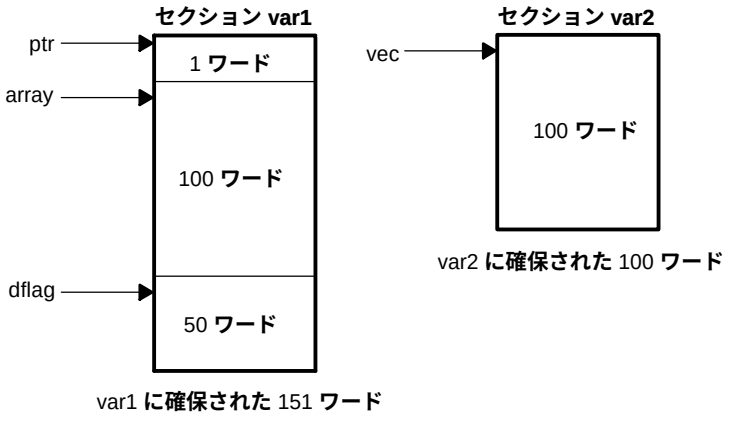
例

この例では、.usect 疑似命令を使って 2 つの初期化されない、名前付きセクション var1 と var2 を定義しています。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 array は、var1 に確保された 100 ワードのブロックにある最初のワードを指し、dflag は var1 にある 50 ワードのブロックにある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。

4-74 ページの 図 4 - 10 は、この例でどのように空間が 2 つの初期化されないセクション var1 と var2 に確保されるかを示しています。

```
 1            ************************************
 2            *      Assemble into .text         *
 3            ************************************
 4 0000              .text
 5 0000 2003        LAC       03h
 6
 7            ************************************
 8            *      Reserve 1 word in var1      *
 9            ************************************
10 0000       ptr    .usect    "var1", 1
11
12            ************************************
13            *  Reserve 100 more words in var1  *
14            ************************************
15 0001       array  .usect    "var1", 100
16
17 0001 0037        ADD       037h    ; Still in .text
18
19            ************************************
20            *   Reserve 50 more words in var1  *
21            ************************************
22 0065       dflag  .usect    "var1", 50
23
24 0002 0065        ADD       dflag   ; Still in .text
25
26            ************************************
27            *     Reserve 100 words in var2    *
28            ************************************
29 0000       vec    .usect    "var2", 100
30
31 0003 0000        ADD       vec     ; Still in .text
32
33            ************************************
34            * Declare an external .usect symbol *
35            ************************************
36                  .global   array
```

図 4 - 10.　.usect **疑似命令**



構文

.version *generation#number*

説明

.version 疑似命令は、アセンブラに対して対象コードがどの世代のプロセッサ用であるかを指示します。有効な世代は 10、16、20、25、29、50 です。.version 疑似命令は、命令や疑似命令よりも前に指定することが必要です。そうしないとエラーが発生します。.version 疑似命令は、コマンド行オプション -v の代わりとして使うことができます。

10 は TMS320C1x デバイスです。

16 は TMS320C16 デバイスです。

20 は TMS320C20 デバイスです。

25 は TMS320C2x デバイスです。

29 は TMS320C2xx デバイスです。

50 は TMS320C5x デバイスです。

第 5 章

命令セット

TMS320C1x、TMS320C2x、TMS320C2xx、および TMS320C5x デバイスは、基本的な汎用命令と同時に、DSP その他の数値集約的なアプリケーションに特に適した算術集約型の命令もサポートしています。

この章には、TMS320C1x、TMS320C2x、TMS320C2xx、および TMS320C5x の命令をアルファベット順にまとめた表を掲載してあります。表の各項目ごとに、命令の構文、命令をサポートするデバイス、命令の動作についての説明を示してあります。5.1節では、表で使われている略語について説明し、表項目の例を示します。

TMS320C2x、TMS320C2xx および TMS320C5x デバイスには拡張命令があります。拡張命令は、1 つのニーモニックで複数の類似した命令の機能を実行するものです。拡張命令については 5.2 節にまとめてあります。

この章では、演算コード、命令のタイミング、アドレッシング・モードなどの項目は扱っていません。これらの項目については、以下のマニュアルに詳しい説明があります。

TMS320C1x User's Guide （文献番号 SPRU013）

TMS320C2x User's Guide （文献番号 SPRU014）

TMS320C2xx User's Guide （文献番号 SPRU127）

TMS320C5x ユーザーズ・マニュアル （文献番号 SCJ1864）

これらのユーザーズ・マニュアルには、本書の4 - 20ページから始まっている疑似命令のリファレンスに似た形式で、命令セットがアルファベット順に示されています。

項目 ページ

5.1 まとめの使い方

この節では、まとめの表を読むときに分かりやすいように、表項目の例と略語のリストを示します。

表項目の例

複数の構文が使われる場合、最初に示す構文が一般に直接アドレッシング用で、2 番目の構文が一般に間接アドレッシング用です。3 つ以上の構文が使われる場合、その構文は通常はデバイスに特有の構文です。

AND 命令を例にとって説明します。

構文		1x	2x	2xx	5x	説明
AND	*dma*	√	√	√	√	**アキュムレータとの AND**
AND	{*ind*} *[,next ARP]*	√	√	√	√	
AND	*# lk [,shift]*			√	√	TMS320C1x、TMS320C2x **デバイス**: **アドレス指定されたデータ・メモリ・ロケーションの内容とアキュムレータの** 16 LSB **との** AND **演算を実行します。アキュムレータの** 16 MSB **は** 0 **と** AND **演算されます。** TMS320C2xx、TMS320C5x **デバイス**: **アドレス指定されたデータの内容または** 16 **ビットのイミディエイト値とアキュムレータのメモリ・ロケーションとの** AND **演算を実行します。アキュムレータの** 16 MSB **は** 0 **と** AND **演算されます。シフトが指定されていれば、定数は** AND **演算の前に左にシフトされます。シフトされた値の上位および下位のビットは、**0 **として取り扱われます。**

最初の構文欄には、AND **命令のニーモニックと構文が示されます。**

2 **番目から** 5 **番目までの欄（**1x、2x、2xx、5x**）にあるチェックマークは、どのデバイスでどの構文が使用できるかを示しています。**

1x **は** TMS320C1x **デバイスです。**

2x **は** TMS320C2x **デバイスで、**TMS320C25 **もこれに含まれます。**

2xx **は** TMS320C2xx **デバイスです。**

5x **は** TMS320C5x **デバイスです。**

この例では、最初の 2 **つの構文は**TMS320C1x、TMS320C2x、TMS320C2xx、**および** TMS320C5x **で使えますが、最後の構文は** TMS320C2xx **と** TMS320C5x **の両デバイスでしか使えません。**

6 **番目の説明欄には、命令の機能を簡単に説明してあります。デバイスが異なると、命令の機能もやや異なることがよくあります。命令を使う前に、説明全体をよく読んで下さい。**

記号と略語

以下の表に、この章で使う命令セットの記号と略語を示します。

表5 - 1. 命令セットのまとめで使われる記号と略語

記号	説明	記号	説明
# lk	16 ビットのイミディエイト値	**INTM**	割り込みマスク・ビット
# k	8 ビットのイミディエイト値	**INTR**	割り込みモード・ビット
{ind}	間接アドレス	**OV**	オーバーフロー・ビット
ACC	アキュムレータ	**P**	プログラム・バス
ACCB	アキュムレータ・バッファ	**PA**	ポート・アドレス
AR	補助レジスタ	**PC**	プログラム・カウンタ
ARCR	補助レジスタ比較	**PM**	プロダクト・シフタ・モード
ARP	補助レジスタ・ポインタ	**pma**	プログラム・メモリ・アドレス
BMAR	ブロック転送アドレス・レジスタ	**RPTC**	リピート・カウンタ
BRCR	ブロック・リピート・カウント・レジスタ	**shift*n***	シフト・データ
C	キャリー・ビット	**src**	ソース・アドレス
DBMR	動的ビット操作レジスタ	**ST**	ステータス・レジスタ
dma	データ・メモリ・アドレス	**SXM**	符号拡張モード・ビット
DP	データ・メモリ・ページ・ポインタ	**TC**	テスト/制御ビット
dst	デスティネーション・アドレス	**T**	テンポラリ・レジスタ
FO	フォーマット・ステータス・リスト	**TREGn**	TMS320C5x テンポラリ・レジスタ（0 ～ 2）
FSX	外部フレーミング・パルス	**TXM**	送信モード・ステータス・レジスタ
IMR	割り込みマスクレジスタ	**XF**	XF ピン・ステータス・ビット

間接アドレス {ind} がどのように解釈されるかを、以下にデバイス別に示します。

{ind}	**'C1x**:	{ * | * + | *- }
	'C2x:	{ * | * + | *- | *0+ | *0- | *BR0+ | *BR0- }
	'C2xx:	{ * | * + | *- | *0+ | *0- | *BR0+ | *BR0- }
	'C5x:	{ * | * + | *- | *0+ | *0- | *BR0+ | *BR0- }

たとえば、

ADD {*ind*}

は次のように解釈されます。

'C1x デバイス	ADD { * | * + | *- }
'C2x デバイス	ADD { * | * + | *- | *0+ | *0- | *BR0+ | *BR0- }
'C2xx デバイス	ADD { * | * + | *- | *0+ | *0- | *BR0+ | *BR0- }
'C5x デバイス	ADD { * | * + | *- | *0+ | *0- | *BR0+ | *BR0- }

デバイス別の、shift*n* の有効なエントリは次のとおりです。

shift	**'C1x**:	0-15 ビット
	'C2x:	0-15 ビット
	'C2xx:	0-16 ビット
	'C5x:	0-16 ビット
shift1	**'C1x**:	適用外
	'C2x:	0-15 ビット
	'C2xx:	0-16 ビット
	'C5x:	0-16 ビット
shift2	**'C1x**:	適用外
	'C2x:	適用外
	'C2xx:	0-15 ビット
	'C5x:	0-15 ビット

5.2 拡張命令

拡張命令は、複数の類似した命令の機能を実行する単一のニーモニックです。たとえば、ADD というニーモニック 1 つで、ADD、ADDH、ADDK、ADLK の機能を実行します。このような拡張命令は、TMS320C2x、TMS320C2xx、TMS320C5x の各デバイスに有効です (TMS320C1x については無効です) 。

表 5 - 2 は、拡張命令と拡張命令が実行する機能を (TMS320C1x/2x ニーモニックに基づいて) まとめたものです。

表5 - 2. 拡張命令のまとめ

拡張命令	組み込まれている演算
ADD	ADD, ADDH, ADDK, ADLK
AND	AND, ANDK
BCND	BBNZ, BBZ, BC, BCND, BGEZ, BGZ, BIOZ, BLEZ, BLZ, BNC, BNV, BNZ, BV, BZ
BLDD	BLDD, BLKD
BLDP	BLDP, BLKP
CLRC	CLRC, CNFD, EINT, RC, RHM, ROVM, RSXM, RTC, RXF
LACC	LAC, LACC, LALK, ZALH
LACL	LACK, LACL, ZAC, ZALS
LAR	LAR, LARK, LRLK
LDP	LDP, LDPK
LST	LST, LST1
MAR	LARP, MAR
MPY	MPY, MPYK
OR	OR, ORK
RPT	RPT, RPTK
SETC	CNFP, DINT, SC, SETC, SHM, SOVM, SSXM, STC, SXF
SUB	SUB, SUBH, SUBK

拡張命令の詳しい説明については、3-31 ページの 3.13 節を参照してください。

5.3 命令セットのまとめ

構文	1x	2x	2xx	5x	説明
ABS	√	√	√	√	**アキュムレータの絶対値** **アキュムレータの内容が** 0 **より小さければ、内容をその内容の** 2 **の補数で置き換えます。内容が** 0 **以上なら、アキュムレータは影響を受けません。**
ADCB				√	**ACCB をキャリー付きでアキュムレータに加算** ACCB **の内容とキャリー・ビットの値をアキュムレータに加算します。加算の結果、アキュムレータの** MSB **からキャリーが生じると、キャリー・ビットが** 1 **に設定されます。**
ADD *dma* [,*shift*]	√	√	√	√	**シフト付きでアキュムレータに加算**
ADD {*ind*} [,*shift* [,*next ARP*]]	√	√	√	√	TMS320C1x、TMS320C2x**：アドレス指定されたデータ・メモリ・ロケーションの内容をアキュムレータに加算します。シフトが指定されていれば、加算の前にロケーションの内容が左にシフトされます。シフト時に、下位ビットに** 0 **が埋め込まれ、上位ビットは符号拡張されます。**
ADD # *k*			√	√	
ADD # *lk* [,*shift2*]			√	√	TMS320C2xx、TMS320C5x**：アドレス指定されたデータ・メモリ・ロケーションの内容またはイミディエイト値をアキュムレータに加算します。シフトが指定されていれば、加算の前にデータが左にシフトされます。シフト時に、下位ビットに** 0 **が埋め込まれ、**SXM = 1 **であれば上位ビットは符号拡張されます。**
ADDB				√	**ACCB をアキュムレータに加算** ACCB **の内容をアキュムレータに加算します。**
ADDC *dma*		√	√	√	**キャリー付きでアキュムレータに加算**
ADDC {*ind*} [,*next ARP*]		√	√	√	**アドレス指定されたデータ・メモリ・ロケーションの内容とキャリー・ビットをアキュムレータに加算します。**
ADDH *dma*	√	√	√	√	**上位をアキュムレータに加算**
ADDH {*ind*} [,*next ARP*]	√	√	√	√	**アドレス指定されたデータ・メモリ・ロケーションの内容をアキュムレータの** 16 MSB **に加算します。** LSB **は影響を受けません。加算の結果、キャリーが作成されると、キャリー・ビットが** 1 **に設定されます。** TMS320C2x、TMS320C2xx、TMS320C5x **加算の結果、アキュムレータの** MSB **からキャリーが生じると、キャリー・ビットが** 1 **に設定されます。**

構文	1x	2x	2xx	5x	説明
ADDK *# k*		√	√	√	**ショート・イミディエイトをアキュムレータに加算** TMS320C1x： 8 **ビットのイミディエイト値をアキュムレータに加算します。** TMS320C2x、TMS320C2xx、TMS320C5x： 8 **ビット・イミディエイト値を右寄せでアキュムレータに加算し、結果をアキュムレータの内容に置き換えます。イミディエイト値は** 8 **ビットの正の数値として取り扱われます。符号拡張は抑制されます。**
ADDS *dma* **ADDS** {*ind*} [,*next ARP*]	√ √	√ √	√ √	√ √	**符号拡張を抑制してアキュムレータに加算** **アドレス指定されたデータ・メモリ・ロケーションの内容をアキュムレータに加算します。値は** 16 **ビットの符号なしの数値として取り扱われます。符号拡張は抑制されます。**
ADDT *dma* **ADDT** {*ind*} [,*next ARP*]		√ √	√ √	√ √	**T レジスタ指定によるシフト付きでアキュムレータに加算** **アドレス指定されたデータ・メモリ・ロケーションの内容を** T **レジスタの** 4 LSB **の値だけ左にシフトし、結果をアキュムレータに加算します。シフトが指定されていれば、加算の前にデータを左にシフトします。シフト時に、下位ビットに** 0 **が埋め込まれ、**SXM = 1 **であれば上位ビットは符号拡張されます。** TMS320C2xx、TMS320C5x**：加算の結果、アキュムレータの** MSB **からキャリーが生じると、キャリー・ビットが** 1 **に設定されます。**
ADLK *# lk* [,*shift*]		√	√	√	**シフト付きロング・イミディエイトをアキュムレータに加算** 16 **ビットのイミディエイト値をアキュムレータに加算します。シフトが指定されていれば、加算の前にデータを左にシフトします。シフト時に、下位ビットに** 0 **が埋め込まれ、**SXM = 1 **であれば上位ビットは符号拡張されます。**
ADRK *# k*		√	√	√	**ショート・イミディエイトを補助レジスタに加算** 8 **ビットのイミディエイト値を現行の補助レジスタに加算します。**

構文		1x	2x	2xx	5x	説明
AND	*dma*	√	√	√	√	**アキュムレータとの AND**
AND	{*ind*} [,*next ARP*]	√	√	√	√	TMS320C1x、TMS320C2x：**アドレス指定されたデータ・メモリ・ロケーションの内容とアキュムレータの** 16 LSB **との** AND **演算を実行します。アキュムレータの** 16 MSB **は** 0 **と** AND **演算されます。**
AND	# *lk* [,*shift*]			√	√	TMS320C2xx、TMS320C5x：**アドレス指定されたデータ・メモリ・ロケーションの内容または** 16 **ビットのイミディエイト値とアキュムレータの内容との** AND **演算を実行します。アキュムレータの** 16 MSB **は** 0 **と** AND **演算されます。シフトが指定されていれば、**AND **の前に値を左にシフトします。シフトされた値の上位および下位のビットは** 0 **として取り扱われます。**
ANDB					√	**ACCB とアキュムレータとの AND** ACCB **の内容とアキュムレータとの** AND **演算を実行します。**
ANDK	# *lk* [,*shift*]		√	√	√	**シフト付きでのイミディエイトとアキュムレータとの AND** 16 **ビットのイミディエイト値とアキュムレータの内容との** AND **演算を実行します。シフトが指定されていれば、**AND **演算の前にイミディエイト値が左にシフトされます。**
APAC		√	√	√	√	**レジスタをアキュムレータに加算** P **レジスタの内容をアキュムレータに加算します。** TMS320C2x、TMS320C2xx、TMS320C5x：**加算の前に、**P **レジスタの内容を** PM **ステータス・ビットの定義に従って左にシフトします。**
APL	[# *lk*] ,*dma*				√	**データ・メモリ値と DBMR またはロング定数との AND**
APL	[# *lk*,] {*ind*} [,*next ARP*]				√	**データ・メモリ値と** DBMR **の内容またはロング定数との** AND **演算を実行します。ロング定数が指定されていれば、それとデータ・メモリ・ロケーションの内容との** AND **演算を実行します。結果は、前に最初のオペランドが入っていたデータ・メモリ・ロケーションに書き戻されます。結果が** 0 **なら** TC **ビットが** 1 **に設定され、その他の場合は**TC **ビットがクリアされます。**
B	*pma*	√				**無条件分岐**
B	*pma* [, {*ind*} [,*next ARP*]]		√	√		**指定のプログラム・メモリ・アドレスに分岐します。** TMS320C2x、TMS320C2xx：**現行のの** AR **と** ARP **を指定に従い修正します。**
B[*D*]	*pma* [, {*ind*} [,*next ARP*]]				√	**無条件分岐（遅延オプションあり）** **現行の補助レジスタと** ARP **を指定に従い修正し、指定されたプログラム・メモリ・アドレスに制御を渡します。遅延分岐（**BD**）を指定すると、分岐の前に、次の** 2 **つの命令ワード（**1 **ワード命令** 2 **個または** 2 **ワード命令** 1 **個）がフェッチされ実行されます。**

構文	1x	2x	2xx	5x	説明
BACC		√	√		**アキュムレータで指定されたアドレスへの分岐** **アキュムレータの** 16 LSB **により指定されているロケーションに分岐します。**
BACC[*D*]				√	**アキュムレータで指定されたアドレスへの分岐** (**遅延オプションあり**) **アキュムレータの** 16 LSB **により指定されているロケーションに分岐します。** **遅延分岐**（BACCD）**を指定すると、分岐の前に、次の** 2 **つの命令ワード**（1 **ワード命令** 2 **個または** 2 **ワード命令** 1 **個**）**がフェッチされ実行されます。**
BANZ *pma* **BANZ** *pma* [, {*ind*} [,*next ARP*]]	√	√	√		**補助レジスタがゼロでなければ分岐** **現行の補助レジスタの** 9 LSB（TMS320C1x **の場合**）**または現行の補助レジスタ全体の内容**(TMS320C2x **の場合**）**が** 0 **でなければ、指定のプログラム・メモリ・アドレスに分岐します。** TMS320C1x**：カレント** AR **をデクリメントします。** TMS320C2x**、**TMS320C2xx**：現行のカレントの** AR **および** ARP **を修正するか**（**指定されている場合**) **または現行の** AR **をデクリメントします**（**デフォルト**）**。**
BANZ[*D*] *pma* [, {*ind*} [,*next ARP*]]				√	**補助レジスタがゼロでなければ分岐**（**遅延オプションあり**） **現行の補助レジスタが** 0 **でなければ、指定のプログラム・メモリ・アドレスに分岐します。現行の** AR **と** ARP **を指定に従い修正するか、または現行の** AR **をデクリメントします。** **遅延分岐**（BANZD）**を指定すると、分岐の前に、次の** 2 **つの命令ワード**（1 **ワード命令** 2 **個または** 2 **ワード命令** 1 **個**）**がフェッチされ実行されます。**
BBNZ *pma* [, {*ind*} [,*next ARP*]]		√	√	√	**TC**ビットがゼロでなければ分岐 TC **ビット** = 1 **であれば、指定のプログラム・メモリ・アドレスに分岐します。** TMS320C2x**：現行の** AR **と** ARP **を指定に従い修正します。** TMS320C2xx**、**TMS320C5x**：** - p **移植スイッチが使用されていれば、現行の** AR **と** ARP **を指定に従い修正します。**

構文		1x	2x	2xx	5x	説明
BBZ	*pma* [, {*ind*} [,*next ARP*]]		√	√	√	**TCビットがゼロならば分岐**
BBZ	*pma*				√	TC **ビット** = 0 **であれば、指定のプログラム・メモリ・アドレスに分岐します。** TMS320C2x**：現行の** AR **と** ARP **を指定に従い修正します。** TMS320C2xx**、**TMS320C5x**：** - p **移植スイッチが使用されていれば、現行の** AR **と** ARP **を指定に従い修正します。**
BC	*pma* [, {*ind*} [, *next ARP*]]		√		√	**キャリーがあれば分岐**
BC	*pma*			√	√	C **ビット** = 1 **であれば、指定のプログラム・メモリ・アドレスに分岐します。** TMS320C2x**：現行の** AR **と** ARP **を指定に従い修正します。** TMS320C2xx**、**TMS320C5x**：現行** - p **移植スイッチが使用されていれば、現行の** AR **と** ARP **を指定に従い修正します。**
BCND	*pma* [*cond*$_1$] [, *cond*$_2$] [...]			√		**条件付き分岐** **指定の条件を満たしていれば、プログラム・メモリ・アドレスに分岐します。条件のすべての組み合わせが有効なわけではありません。**
BCND[*D*]	*pma* [*cond*$_1$][, *cond*$_2$] [...]				√	**条件付き分岐（遅延オプションあり）** **指定の条件を満たしていれば、プログラム・メモリ・アドレスに分岐します。条件のすべての組み合わせが有効なわけではありません。** **遅延分岐（**BCNDD**）を指定すると、分岐の前に、次の** 2 **つの命令ワード（**1 **ワード命令** 2 **個または** 2 **ワード命令** 1 **個）がフェッチされ実行されます。**
BGEZ	*pma*	√		√	√	**アキュムレータがゼロ以上ならば分岐**
BGEZ	*pma* [,{*ind*} [,*next ARP*]]		√		√	**アキュムレータの内容が** 0 **以上なら、指定のプログラム・メモリ・アドレスに分岐します。** TMS320C2x**：現行の** AR **と** ARP **を指定に従い修正します。** TMS320C2xx**、**TMS320C5x**：** - p **移植スイッチが使用されていれば、現行の** AR **と** ARP **を指定に従い修正します。**
BGZ	*pma*	√		√	√	**アキュムレータがゼロより大ならば分岐**
BGZ	*pma* [, {*ind*} [,*next ARP*]]		√		√	**アキュムレータの内容が** 0 **より大きければ、指定のプログラム・メモリ・アドレスに分岐します。** TMS320C2x**：現行の** AR **と** ARP **を指定に従い修正します。** TMS320C2xx**、**TMS320C5x**：** - p **移植スイッチが使用されていれば、現行の** AR **と** ARP **を指定に従い修正します。**

構文		1x	2x	2xx	5x	説明
BIOZ	*pma*	√		√	√	**I/O ステータスがゼロならば分岐**
BIOZ	*pma* [,{*ind*} [,*next ARP*]]		√		√	$\overline{\text{BIO}}$ **ピンがローなら、指定のプログラム・メモリ・アドレスに分岐します。** TMS320C2x**：現行の** AR **と** ARP **を指定に従い修正します。** TMS320C2xx**、**TMS320C5x**：** - p **移植スイッチが使用されていれば、現行の** AR **と** ARP **を指定に従い修正します。**
BIT	*dma, bit code*		√	√	√	**ビットをテスト**
BIT	{*ind*}*, bit code* [,*next ARP*]		√	√	√	**データ・メモリ値の指定のビットを** ST1 **の** TC **ビットにコピーします。**
BITT	*dma*		√	√	√	**T レジスタで指定されたビットをテスト**
BITT	{*ind*} [,*next ARP*]		√	√	√	TMS320C2x**、**TMS320C2xx**：データ・メモリ値の指定のビットを** ST1 **の** TC **ビットにコピーします。**T **レジスタの** 4 LSB **は、コピーするビットを指定します。** TMS320C5x**：データ・メモリ値の指定のビットを** ST1 **の** TC **ビットにコピーします。**TREG2 **の** 4 LSB **は、コピーするビットを指定します。**
BLDD	*src, dst*			√	√	**データ・メモリ間のブロック転送**
BLDD	*# lk, dma*			√	√	**データ・メモリからデータ・メモリへ** 1 **ブロックをコピーします。ソース・データ・メモリ・ブロックは** *src* **により、デスティネーション・データ・メモリ・ブロックは** *dst* **により指し示されます。**
BLDD	*# lk,* {*ind*} [,*next ARP*]			√	√	
BLDD	*dma, # lk*			√	√	
BLDD	{*ind*}*, # lk* [,*next ARP*]			√	√	TMS320C2xx**：ソースのワードとデスティネーション空間は、ロング・イミディエイト値またはデータ・メモリ・アドレスを使って指すことができます。**RPT **命令を** BLDD **とともに使うと、間接的に指し示されたデータ・メモリ内の連続するワードを、連続するプログラム・メモリ空間に転送することができます。転送されるワード数は、**RPTC**の初期値** +1 **ワードです。**
BLDD	BMAR, *dma*				√	
BLDD	BMAR, {*ind*} [,*next ARP*]				√	
BLDD	*dma,* BMAR				√	
BLDD	{*ind*}*,* BMAR [,*next ARP*]				√	
						TMS320C5x**：ソースのワードとデスティネーション空間は、ロング・イミディエイト値、**BMAR **の内容、またはデータ・メモリ・アドレスを使って指すことができます。**RPT **命令を** BLDD **とともに使うと、間接的に指し示されたデータ・メモリ内の連続するワードを、連続するプログラム・メモリ空間に転送することができます。転送されるワード数は、**RPTC**の初期値** +1 **ワードです。**

構文		1x	2x	2xx	5x	説明
BLDP	*dma*				√	**データ・メモリからプログラム・メモリへのブロック転送**
BLDP	{*ind*} [,*next ARP*]				√	**データ・メモリの** 1 **ブロックを、**BMAR **が指すプログラム・メモリにコピーします。**RPT **命令を** BLDP **とともに使うと、間接的に指し示されたデータ・メモリ内の連続するワードを、**BMAR **で指し示された連続するプログラム・メモリ空間に転送することができます。**
BLEZ	*pma*	√		√	√	**アキュムレータがゼロ以下ならば分岐**
BLEZ	*pma* [, {*ind*} [,*next ARP*]]		√	√	√	**アキュムレータの内容が** 0 **以下なら、指定のプログラム・メモリ・アドレスに分岐します。** TMS320C2x**：現行の** AR **と** ARP **を指定に従い修正します。** TMS320C2xx**、**TMS320C5x**：** - p **移植スイッチが使用されていれば、現行の** AR **と** ARP **を指定に従い修正します。**
BLKD	*dma1, dma2*		√	√	√	**データ・メモリ間のブロック転送**
BLKD	*dma1,* {*ind*} [,*next ARP*]		√	√	√	**データ・メモリ内のロケーションからデータ・メモリ内の別のロケーションへ、**1 **ブロックのワードを転送します。現行の** AR **と** ARP **を指定に従い修正します。複数のワードを転送するときは、**RPT **または** RPTK **を間接アドレッシング・モードで** BLKD **とともに使う必要があります。転送されるワード数は、** RPTC **初期値＋１ワードです。**
BLKP	*pma, dma*		√	√	√	**プログラム・メモリからデータ・メモリへのブロック転送**
BLKP	*pma,* {*ind*} [,*next ARP*]		√	√	√	**プログラム・メモリ内のロケーションからデータ・メモリ内のロケーションへ、**1 **ブロックのワードを移動します。現行の** AR **と** ARP **を指定に従い修正します。複数のワードを転送するときは、**RPT **または** RPTK **を間接アドレッシング・モードで** BLKD **とともに使う必要があります。転送されるワード数は、** RPTC **初期値＋１ワードです。**

構文		1x	2x	2xx	5x	説明
BLPD	*src, dst*			√	√	**プログラム・メモリからデータ・メモリへのブロック転送**
BLPD **BLPD** **BLPD** **BLPD**	*#pma, dma* # *pma,* {*ind*} [,*next ARP*] BMAR, *dma* BMAR, {*ind*} [,*next ARP*]			√ √	√ √ √ √	**プログラム・メモリからデータ・メモリへ、**1 **ブロックをコピーします。プログラム・メモリのブロックは** *src* **により、データ・メモリのブロックは** *dst* **により指示されます。** TMS320C2xx**：ソース空間のワードはロング・イミディエイト値を使って指すことができます。**RPT **命令を** BLPD **とともに使うと、間接的に指し示されたデータ・メモリ内の連続するワードを、連続するプログラム・メモリ空間に転送することができます。** TMS320C5x**：ソース空間のワードはロング・イミディエイト値または** BMAR **の内容を使って指すことができます。**RPT **命令を** BLPD **とともに使うと、間接的に指し示されたデータ・メモリ内の連続するワードを、連続するプログラム・メモリ空間に転送することができます**
BLZ **BLZ**	*pma* *pma* [, {*ind*} [,*next ARP*]]	√	√	√ √	√	**アキュムレータがゼロより小ならば分岐** **アキュムレータの内容が** 0 **より小さければ、指定のプログラム・メモリ・アドレスに分岐します。** TMS320C2x**：現行の** AR **と** ARP **を指定に従い修正します。** TMS320C2xx**、**TMS320C5x**：** - p **移植スイッチが使用されていれば、現行の** AR **と** ARP **を指定に従い修正します。**
BNC	*pma* [, {*ind*} [,*next ARP*]]		√	√	√	**キャリーがなければ分岐** C **ビット** = 0 **なら、指定のプログラム・メモリ・アドレスに分岐します。** TMS320C2x**：現行の** AR **と** ARP **を指定に従い修正します。** TMS320C2xx**、**TMS320C5x**：** - p **移植スイッチが使用されていれば、現行の** AR **と** ARP **を指定に従い修正します。**
BNV	*pma* [, {*ind*} [,*next ARP*]]		√	√	√	**オーバーフローでなければ分岐** OV **フラグがクリアされていれば、指定のプログラム・メモリ・アドレスに分岐します。** TMS320C2x**：現行の** AR **と** ARP **を指定に従い修正します。** TMS320C2xx**、**TMS320C5x**：** - p **移植スイッチが使用されていれば、現行の** AR **と** ARP **を指定に従い修正します。**

構文		1x	2x	2xx	5x	説明
BNZ	*pma*	√				**アキュムレータがゼロでなければ分岐**
BNZ	*pma* [, {*ind*} [,*next ARP*]]		√	√	√	**アキュムレータの内容が** 0 **でなければ、指定のプログラム・メモリ・アドレスに分岐します。** TMS320C2x**：現行の** AR **と** ARP **を指定に従い修正します。** TMS320C2xx**、**TMS320C5x**：** - p **移植スイッチが使用されていれば、現行の** AR **と** ARP **を指定に従い修正します。**
BSAR	[*shift*]				√	**バレル・シフト** 1 **サイクルで、アキュムレータの** 1 **〜** 16 **ビットの右へのバレル算術シフトを実行します。符号拡張は、**ST1 **の符号拡張モード・ビットにより決まります。**
BV	*pma*	√				**オーバーフローならば分岐**
BV	*pma* [, {*ind*} [,*next ARP*]]		√	√	√	OV **フラグが設定されていれば、指定のプログラム・メモリ・アドレスに分岐して** OV **フラグをクリアします。** TMS320C2x**、**TMS320C2xx**、**TMS320C5x **：現行の** AR **と** ARP **を指定に従い修正します。** TMS320C2xx**、**TMS320C5x**：**AR **と** ARP **を修正するには、** - p **移植スイッチを使います。**
BZ	*pma*	√		√	√	**アキュムレータがゼロならば分岐**
BZ	*pma* [, {*ind*} [,*next ARP*]]		√			**アキュムレータの内容が** 0 **なら、指定のプログラム・メモリ・アドレスに分岐します。** TMS320C2x**、**TMS320C2xx**、**TMS320C5x **：現行の** AR **と** ARP **を指定に従い修正します。** TMS320C2xx**、**TMS320C5x**：**AR **と** ARP **を修正するには、** - p **移植スイッチを使います。**
CALA		√	√	√		**サブルーチンの間接呼び出し** **アキュムレータの内容は、サブルーチンのアドレスを指定します。**PC **をインクリメントし、** PC **をスタックにプッシュします。** TMS320C1x**：アキュムレータの** 12 LSB **を** PC **にロードします。** TMS320C2x/C2xx**：アキュムレータの** 16 LSB **を** PC **にロードします。**
CALA[*D*]					√	**サブルーチンの間接呼び出し（遅延オプションあり）** **アキュムレータの内容は、サブルーチンのアドレスを指定します。**PC **をインクリメントし、** PC **をスタックにプッシュし、アキュムレータの** 16 LSB **を** PC **にロードします。** **遅延分岐（**CALAD**）を指定すると、呼び出しの前に、次の** 2 **つの命令ワード（**1 **ワード命令** 2 **個または** 2 **ワード命令** 1 **個）がフェッチされ実行されます。**

構文	1x	2x	2xx	5x	説明
CALL *pma* **CALL** *pma* [,{*ind*} [,*next ARP*]]	√	√	√		**サブルーチンの呼び出し** **アドレス指定されたプログラム・メモリ・ロケーションの内容は、サブルーチンのアドレスを指定します。**PC **を** 2 **だけインクリメントし、**PC **をスタックにプッシュし、指定のプログラム・メモリ・アドレスを** PC **にロードします。** TMS320C2x**、**TMS320C2xx**：現行の** AR **と** ARP **を指定に従い修正します。**
CALL[*D*] *pma* [,{*ind*} [*, next ARP*]]				√	**無条件呼び出し（遅延オプションあり）** **アドレス指定されたプログラム・メモリ・ロケーションの内容は、サブルーチンのアドレスを指定します。**PC **をインクリメントし、**PC **をスタックにプッシュし、指定のプログラム・メモリ・アドレス（シンボルまたは数値）を** PC **にロードします。現行の** AR **と** ARP **を指定に従い修正します。** **遅延分岐（**CALLD**）を指定すると、呼び出しの前に、次の** 2 **つの命令ワード（**1 **ワード命令** 2 **個または** 2 **ワード命令** 1 **個）がフェッチされ実行されます。**
CC *pma* [$cond_1$] [,$cond_2$] [,...]			√		**条件付き呼び出し** **指定の条件を満たしていれば、**pma **に制御が渡されます。条件のすべての組み合わせが有効なわけではありません。**
CC[*D*] *pma* [,*cond1*] [,*cond2*] [,...]				√	**条件付き呼び出し（遅延オプションあり）** **指定の条件を満たしていれば、**pma **に制御が渡されます。条件のすべての組み合わせが有効なわけではありません。** **遅延分岐（**CCD**）を指定すると、呼び出しの前に、次の** 2 **つの命令ワード（**1 **ワード命令** 2 **個または** 2 **ワード命令** 1 **個）がフェッチされ実行されます。**
CLRC *control bit*			√	√	**制御ビットのクリア** **指定の制御ビットを論理** 0 **に設定します。マスク可能割り込みは、**CLRC **命令の実行後ただちに有効化されます。**
CMPL		√	√	√	**アキュムレータの補数** **アキュムレータの内容を論理反転します（**1 **の補数）。**

構文	1x	2x	2xx	5x	説明
CMPR *CM*		√	√	√	**補助レジスタを AR0 と比較** **下記に基づき、現行の補助レジスタの内容を** AR0 **と比較します。** CM = 00_2 **なら、**AR(ARP) = AR0 **かどうかをテスト** CM = 01_2 **なら、**AR(ARP) < AR0 **かどうかをテスト** CM = 10_2 **なら、**AR(ARP) > AR0 **かどうかをテスト** CM = 11_2 **なら、**AR(ARP) ≠ AR0 **かどうかをテスト** **結果が真の場合は、**TC **ステータス・ビットに** 1 **をロードします。それ以外の場合は、**TC **ビットに** 0 **をロードします。比較は、テスト済みのレジスタには影響を及ぼしません。** TMS320C5x**：補助レジスタ** ARCR **と比較します。**
CNFD		√	√	√	**ブロックをデータ・メモリとして構成** **オンチップ** RAM **ブロック** B0 **をデータ・メモリとして構成します。** TMS320C2x,TMS320C2xx**：ブロック** B0 **はデータ・メモリ・ロケーション** 512 ～ 767 **にマップされます。** TMS320C5x**：ブロック** B0 **はデータ・メモリ・ロケーション** 256 ～ 767 **にマップされます。**
CNFP		√	√	√	**ブロックをプログラム・メモリとして構成** **オンチップ** RAM **ブロック** B0 **をプログラム・メモリとして構成します。** TMS320C2x**、**TMS320C2xx**：ブロック** B0 **はプログラム・メモリ・ロケーション** 65280 ～ 65535 **にマップされます。** TMS320C5x**：ブロック** B0 **はプログラム・メモリ・ロケーション** 65024 ～ 65535 **にマップされます。**
CONF *2-bit constant*		√			**ブロックをプログラム・メモリとして構成** **オンチップ** RAM **ブロック** B0/B1/B2/B3 **をプログラム・メモリとして構成します。**B0/B1/B2/B3 **のメモリ・マッピングの説明については** *TMS320C2x User's Guide* **を参照してください。**
CPL [*, # lk*] *dma* **CPL** [*, # lk*] {*ind*} [*,next ARP*]				√ √	**DBMR またはイミディエイト値をデータ値と比較** 2 **つの数量を比較します。**2 **つの数量が等しい場合は、**TC **ビットを** 1 **に設定します。それ以外の場合は、**TC **ビットをクリアします。**
CRGT				√	**ACC > ACCB のテスト** ACC **の内容を** ACCB **の内容と比較します。そして、大きい方の符号付きの値を両方のレジスタにロードし、比較の結果に従ってキャリー・ビットを修正します。**ACC **の内容が** ACCB **の内容以上であれば、キャリー・ビットを** 1 **に設定します。**

構文	1x	2x	2xx	5x	説明
CRLT				√	**ACC<ACCB のテスト** ACC の内容を ACCB の内容と比較します。そして、小さい方の符号付きの値を両方のレジスタにロードし、比較の結果に従ってキャリー・ビットを修正します。ACC の内容が ACCB の内容より小さければ、キャリー・ビットを 1 に設定します。
DINT	√	√	√	√	**割り込みの禁止** すべての割り込みを禁止し、INTM を 1 に設定します。マスク可能割り込みは、DINT 命令の実行後ただちに禁止されます。DINT は、マスク不能割り込み $\overline{RS}$ は禁止しません。DINT は、IMR には影響を及ぼしません。
DMOV *dma* **DMOV** {*ind*} [,*next ARP*]	√ √	√ √	√ √	√ √	**データ・メモリ内でのデータ・ムーブ** アドレス指定されたデータ・メモリ・ロケーションの内容を 1 つ上位のアドレスにコピーします。DMOV は、オンチップ RAM ブロックの内部だけでデータを転送します。 TMS320C2x、TMS320C2xx、TMS320 C 5x：オンチップ RAM ブロックは、B0（データ・メモリとして構成されている場合）、B1、および B2 です。
EINT	√	√	√	√	**割り込みの許可** すべての割り込みを許可し、INTM を 0 にクリアします。マスク可能割り込みは、EINT 命令の実行後ただちに許可されます。
EXAR				√	**ACCB と ACC の交換** ACC の内容を ACCB の内容と交換します。
FORT *1-bit constant*		√			**シリアル・ポート・レジスタのフォーマット** FO に 0 または 1 をロードします。FO=0 の場合は、レジスタは 16 ビット・ワードの送受信用に構成されます。FO=1 の場合は、8 ビット・バイトの送受信用に構成されます。
IDLE		√	√	√	**割り込みまでアイドル** 実行中のプログラムに、実行を停止してリセットまたは割り込みを受信するまで待つように強制します。デバイスは割り込みを受けるまでアイドル状態になります。
IDLE2				√	**割り込みまでアイドル − 省電力モード** デバイス内部からの動作クロックの供給を停止します。これにより、省電力モードにすることができます。IDLE2 命令は、実行中のプログラムに、実行を停止してリセットまたはマスクされていない割り込みを受け取るまで待つように強制します。

構文		1x	2x	2xx	5x	説明
IN	*dma, PA*	√	√	√	√	**ポートからのデータを入力**
IN	{*ind*}, *PA* [,*next ARP*]	√	√	√	√	**外部** I/O **ポートの** 1 **つからアドレス指定されたデータ・メモリ・ロケーションに** 16 **ビット値を読み取ります。** TMS320C1x**：これは** 2 **サイクル命令です。最初のサイクルで、ポート・アドレスがアドレス・ライン** A2/PA2 **〜** A0/PA0 **に送られます。** $\overline{\text{DEN}}$ **がローになり、アドレス指定されたペリフェラルがデータ・バス** D15 **〜** D0 **に入れるデータをストローブします。** TMS320C2x**：** $\overline{\text{IS}}$ **ラインが、ローになって** I/O **アクセスを示します。** $\overline{\text{STRB}}$**、**R/$\overline{\text{W}}$**、**READY **の各タイミングは外部データ・メモリ読み取りの場合と同じです。** TMS320C2xx**、**TMS320C5x**：** $\overline{\text{IS}}$ **ラインがローになって** I/O **アクセスを示します。**$\overline{\text{STRB}}$**、**$\overline{\text{RD}}$**、** READY **の各タイミングは外部データ・メモリ読み取りの場合と同じです。**
INTR	*k*			√	√	**ソフト割り込み** **プログラム制御を** *k* **により指定されたプログラム・メモリ・アドレスに移します。この命令を使うと、ソフトウェアを使って任意の割り込みサービス・ルーチンを実行することができます。** TMS320C5x**：** ***k* **割り込み ロケーション** 0 $\overline{\text{RS}}$ 0h 1 $\overline{\text{INT1}}$ 2h 2 $\overline{\text{INT2}}$ 4h 3 $\overline{\text{INT3}}$ 6h 4 TINT 8h 5 RINT Ah 6 XINT Ch 7 TRNT Eh 8 TXNT 10h 9 $\overline{\text{INT4}}$ 12h 10 **予約割り込み** 14h 11 **予約割り込み** 16h 12 **予約割り込み** 18h 13 **予約割り込み** 20h 17 TRAP 22h 18 $\overline{\text{NMI}}$ 24h TMS320C2xx**：***TMS320C2xx User's Guide* **を参照して下さい。**
LAC	*dma* [,*shift*]	√	√	√	√	**シフト付きでアキュムレータにロード**
LAC	{*ind*} [,*shift* [,*next ARP*]]	√	√	√	√	**アドレス指定されたデータ・メモリ・ロケーションの内容をアキュムレータにロードします。シフトが指定されていれば、アキュムレータにロードする前に値を左にシフトします。シフト時に、下位ビットに** 0 **が埋め込まれ、**SXM = 1 **であれば上位ビットは符号拡張されます。**
LACB					√	**アキュムレータに ACCB をロード** **アキュムレータ・バッファの内容をアキュムレータにロードします。**

構文		1x	2x	2xx	5x	説明
LACC	*dma* [,$shift_1$]		√	√	√	**シフト付きでアキュムレータにロード**
LACC	{*ind*} [,$shift_1$ [,*next ARP*]]		√	√	√	**アドレス指定されたデータ・メモリ・ロケーションの内容または** 16 **ビット定数をアキュムレータにロードします。シフトが指定されていれば、アキュムレータにロードする前に値を左にシフトします。シフト時に、下位ビットに** 0 **が埋め込まれ、**SXM = 1 **であれば上位ビットは符号拡張されます。**
LACC	# *lk* [,$shift_2$]		√	√	√	
LACK	*8-bit constant*	√	√	√	√	**ショート・イミディエイト値をアキュムレータにロード** 8 **ビット定数をアキュムレータにロードします。アキュムレータの** 24 MSB **はゼロに設定されます。**
LACL	*dma*			√	√	**アキュムレータの下位へのロードと上位のクリア**
LACL	{*ind*} [,*next ARP*]			√	√	**アドレス指定されたデータ・メモリ・ロケーションの内容またはゼロ拡張** 8 **ビット定数をアキュムレータの** 16 LSB **にロードします。アキュムレータの** MSB **はゼロに設定されます。データは** 16 **ビットの符号なしの数値として取り扱われます。** TMS320C2xx: **定数** 0 **は、アキュムレータの内容を符号拡張なしで** 0 **にクリアします。**
LACL	# k			√	√	
LACT	*dma*		√	√	√	**T レジスタ指定のシフト付きでアキュムレータにロード** **アドレス指定されたデータ・メモリ・ロケーションの内容を** T **レジスタの** 4 LSB **に指定されている値だけ左にシフトし、結果をアキュムレータにロードします。シフトが指定されていれば、アキュムレータにロードする前に値を左にシフトします。シフト時に、下位ビットに** 0 **が埋め込まれ、**SXM = 1 **であれば上位ビットは符号拡張されます。**
LACT	{*ind*} [,*next ARP*]		√	√	√	
LALK	# *lk* [,*shift*]		√	√	√	**シフト付きでロング・イミディエイト値をアキュムレータにロード** 16 **ビット・イミディエイト値をアキュムレータにロードします。シフトが指定されていれば、アキュムレータにロードする前に値を左にシフトします。シフト時に、下位ビットに** 0 **が埋め込まれ、**SXM = 1 **であれば上位ビットは符号拡張されます。**
LAMM	*dma*				√	**メモリマップド・レジスタからアキュムレータにロード** **アドレス指定されたメモリマップド・レジスタの内容をアキュムレータの下位ワードにロードします。**DP **の現行の値または** AR（ARP）**の** 9 MSB **に関係なく、データ・メモリ・アドレスの** 9 MSB **はクリアされて使われます。**
LAMM	{*ind*} [,*next ARP*]				√	

構文		1x	2x	2xx	5x	説明
LAR	*AR, dma*	√	√	√	√	**補助レジスタにロード**
LAR	*AR,* {*ind*} [,*next ARP*]	√	√	√	√	TMS320C1x、TMS320C2x：**アドレス指定されたデータ・メモリ・ロケーションの内容を指定の補助レジスタにロードします。**
LAR	*AR, # k*			√	√	
LAR	*AR, # lk*			√	√	TMS320C25、TMS320C2xx、TMS320C5x：**アドレス指定されたデータ・メモリ・ロケーションの内容、または** 8 **ビットか** 16 **ビットのイミディエイト値を、指定の補助レジスタにロードします。**
LARK	*AR, 8-bit constant*	√	√	√	√	**補助レジスタにショート・イミディエイト値をロード** 8 **ビットの正の定数を指定の補助レジスタにロードします。**
LARP	*1-bit constant*	√				**補助レジスタ・ポインタにロード**
LARP	*3-bit constant*		√	√	√	TMS320C1x： 1 **ビットの定数を補助レジスタ・ポインタにロードします（**AR0 **または** AR1 **を指定）。** TMS320C2x、TMS320C2xx、TMS320C5x： 3 **ビットの定数を補助レジスタ・ポインタにロードします（**AR0 ～ AR7 **を指定）。**
LDP	*dma*	√	√	√	√	**データ・メモリ・ページ・ポインタにロード**
LDP	{*ind*} [,*next ARP*]	√	√	√	√	TMS320C1x：**アドレス指定されたデータ・メモリ・ロケーションの内容の** LSB **を** DP **レジスタにロードします。すべての上位ビットは無視されます。** DP＝0 **はページ** 0 (**ワード** 0 ～ 127)**を定義し、**DP＝1 **はページ** 1 **（ワード** 128 ～143/255**）を定義します。**
LDP	*# k*			√	√	TMS320C2x、TMS320C2xx、TMS320C5x：**アドレス指定されたデータ・メモリ・ロケーションの** 9 LSB **または** 9 **ビットのイミディエイト値を** DP **レジスタにロードします。**DP **と** 7 **ビットのデータ・メモリ・アドレスが結合されて、**16 **ビットのデータ・メモリ・アドレスになります。**
LDPK	*1-bit constant*	√				**データ・メモリ・ページ・ポインタにイミディエイト値をロード**
LDPK	*9-bit constant*		√	√	√	TMS320C1x： 1 **ビット・イミディエイト値を** DP **レジスタにロードします。**DP＝0 **はページ** 0 (**ワード** 0 ～ 127) **を定義し、**DP＝1 **はページ** 1 (**ワード** 128 ～143/255) **を定義します。** TMS320C2x、TMS320C2xx、TMS320C5x：9 **ビット・イミディエイトを** DP **レジスタにロードします。**DP **と** 7 **ビットのデータ・メモリ・アドレスが結合されて、**16 **ビットのデータ・メモリ・アドレスになります。** DP≧8 **は外部データ・メモリを指定します。** DP＝4 ～ 7 **は、オンチップ** RAM **ブロック** B0 **または** B1 **を指定します。ブロック** B2 **は、ページ** 0 **の上位** 32 **ワードに位置しています。**

構文		1x	2x	2xx	5x	説明
LMMR	*dma, # lk*				√	**メモリマップド・レジスタにロード**
LMMR	{*ind*}, *# lk* [,*next ARP*]				√	**直接または間接アドレッシングされたデータ・メモリ値の** 7 LSB **が指しているメモリマップド・レジスタに、ロング・イミディエイトのアドレス指定されたデータ・メモリの内容をロードします。**DP **の現行値または** AR（ARP）**の** 9 MSB **に関係なく、データ・メモリ・アドレスの** 9 MSB **はクリアされて使われます。**
LPH	*dma*		√	√	√	**P レジスタの上位**
LPH	{*ind*} [,*next ARP*]		√	√	√	**アドレス指定されたデータ・メモリ・ロケーションの内容を** P **レジスタの** 16 MSB **にロードします。** LSB **は影響を受けません。**
LRLK	*AR, lk*		√	√	√	**補助レジスタにロング・イミディエイト値をロード** 16 **ビット・イミディエイト値を指定の補助レジスタにロードします。**
LST	*dma*	√	√	√	√	**ステータス・レジスタにロード**
LST	{*ind*} [,*next ARP*]	√	√	√	√	**アドレス指定されたデータ・メモリ・ロケーションの内容を** ST（TMS320C1x **の場合）または** ST0（TMS320C2x/2xx/5x **の場合）にロードします。**
LST	*# n, dma*		√	√	√	**ステータス・レジスタ n にロード**
LST	*# n,* {*ind*} [,*next ARP*]		√	√	√	**アドレス指定されたデータ・メモリ・ロケーションの内容を** STn **にロードします。**
LST1	*dma*		√	√	√	**ST1 にロード**
LST1	{*ind*} [,*next ARP*]		√	√	√	**アドレス指定されたデータ・メモリ・ロケーションの内容を** ST1 **にロードします。**
LT	*dma*	√	√	√	√	**T レジスタにロード**
LT	{*ind*} [,*next ARP*]	√	√	√	√	**アドレス指定されたデータ・メモリ・ロケーションの内容を** T **レジスタ** (TMS320C1x/2x/2xx **の場合**) **または** TREG0(TMS320C5x **の場合)にロードします。**
LTA	*dma*	√	√	√	√	**T レジスタにロードし直前の積を累算**
LTA	{*ind*} [,*next ARP*]	√	√	√	√	**アドレス指定されたデータ・メモリ・ロケーションの内容を** T **レジスタ** (TMS320C1x/2x/2xx **の場合）または** TREG0(TMS320C5x **の場合) にロードし、**P **レジスタの内容をアキュムレータに加算します。** TMS320C2x、TMS320C2xx、TMS320C5x**：加算の前に、**PM **ステータス・ビットの指定に従い** P **レジスタの内容をシフトします。**

構文		1x	2x	2xx	5x	説明
LTD	*dma*	√	√	√	√	**T レジスタにロードし、直前の積を累算し、データを転送** **アドレス指定されたデータ・メモリ・ロケーションの内容を** T **レジスタ（**TMS320C1x/2x/2xx **の場合）または**TREG0**（**TMS320C5x **の場合）にロードし、**P **レジスタの内容をアキュムレータに加算し、指定ロケーションの内容を** 1 **つ上位のアドレスにコピーします（両方のデータ・メモリ・ロケーションはオンチップ・データ** RAM **に入っていなければなりません）。** TMS320C2x**、**TMS320C2xx**、**TMS320C5x**：加算の前に、**PM **ステータス・ビットの指定に従い** P **レジスタの内容をシフトします。**
LTD	{*ind*} [,*next ARP*]	√	√	√	√	
LTP	*dma*		√	√	√	**T レジスタにロードし、P レジスタをアキュムレータにロード** **アドレス指定されたデータ・メモリ・ロケーションの内容を** T **レジスタ（**TMS320C2x/2xx **の場合）または**TREG0**（**TMS320C5x **の場合）にロードします。プロダクト・レジスタの内容をアキュムレータにロードします。**
LTP	{*ind*} [,*next ARP*]		√	√	√	
LTS	*dma*		√	√	√	**T レジスタにロードし、直前の積を減算** **アドレス指定されたデータ・メモリ・ロケーションの内容を** T **レジスタ（**TMS320C2x/2xx **の場合）または**TREG0**（**TMS320C5x **の場合）にロードします。**PM **ステータス・ビットの指定に従いプロダクト・レジスタの内容をシフトし、結果をアキュムレータから減算します。**
LTS	{*ind*} [,*next ARP*]		√	√	√	
MAC	*pma, dma*		√	√	√	**積和** **データ・メモリ値とプログラム・メモリ値を乗算し、直前の積を（**PM **ステータス・ビットの指定に従いシフトした後に）アキュムレータに加算します。**
MAC	*pma,* {*ind*} [,*next ARP*]		√	√	√	
MACD	*dma, pma*		√	√	√	**データ・ムーブ付きの積和** **データ・メモリ値とプログラム・メモリ値を乗算し、直前の積を（**PM **ステータス・ビットの指定に従いシフトした後に）アキュムレータに加算します。データ・メモリ・アドレスがオンチップ**RAM **ブロック** B0**、**B1**、または** B2 **にある場合は、アドレスの内容を** 1 **つ位のアドレスにコピーします。**
MACD	*pma,* {*ind*} [,*next ARP*]		√	√	√	
MADD	*dma*				√	**データ・ムーブと動的アドレッシング付きの積和** **データ・メモリ値とプログラム・メモリ値を乗算し、直前の積を（**PM **ステータス・ビットの指定に従いシフトした後に）アキュムレータに加算します。プログラム・メモリ・アドレスは** BMAR **に含まれています。したがって、係数テーブルの動的アドレッシングが可能です。** MADD **は** MADS **と同じ働きをするほか、オンチップ** RAM **ブロックの場合にはデータ・ムーブも行います。**
MADD	{*ind*} [,*next ARP*]				√	

構文		1x	2x	2xx	5x	説明
MADS	*dma*		√	√	√	**動的アドレッシング付きの積和**
MADS	{*ind*} [,*next ARP*]		√	√	√	**データ・メモリ値とプログラム・メモリ値を乗算し、直前の積を（**PM **ステータス・ビットの指定に従いシフトした後に）アキュムレータに加算します。プログラム・メモリ・アドレスは** BMAR **に含まれています。したがって、係数テーブルの動的アドレッシングが可能です。**
MAR	*dma*	√	√	√	√	**補助レジスタの修正**
MAR	{*ind*} [,*next ARP*]	√	√	√	√	**現行の** AR **または** ARP **を指定に従い修正します。** MAR **は、間接アドレッシング・モードでは** NOP **として動作します。**
MPY	*dma*	√	√	√	√	**乗算**
MPY	{*ind*} [,*next ARP*]	√	√	√	√	TMS320C1x**、**TMS320C2x**：**T **レジスタの内とアドレス指定されたデータ・メモリ・ロケーションの内容を乗算し、結果を** P **レジスタに入れます。**
MPY	*# k*			√	√	
MPY	*# lk*			√	√	TMS320C2xx**、**TMS320C5x**：**T **レジスタ**(TMS320 C2xx **の場合) **または** TREG0**（**TMS320C5x **の場合）の内容とアドレス指定されたデータ・メモリ・ロケーションの内容または** 13 **ビットか** 16 **ビットのイミディエイト値を乗算し、結果を** P **レジスタに入れます。**
MPYA	*dma*		√	√	√	**乗算し直前の積を累算**
MPYA	{*ind*} [,*next ARP*]		√	√	√	T **レジスタ**（TMS320C2x/2xx **の場合）または** TREG0**（**TMS320C5x **の場合）の内容とアドレス指定されたデータ・メモリ・ロケーションの内容を乗算し、結果を** P **レジスタに入れます。直前の積を（**PM **ステータス・ビットに従いシフトした後で）アキュムレータに加算します。**
MPYK	*13-bit constant*	√	√	√	√	**イミディエイト値の乗算**
						T **レジスタ**（TMS320C1x/2x/2xx **の場合）または** TREG0**（**TMS320C5x **の場合）の内容と符号付きの** 13 **ビット定数を乗算し、結果を** P **レジスタに入れます。**
MPYS	*dma*		√	√	√	**乗算し直前の積を減算**
MPYS	{*ind*} [,*next ARP*]		√	√	√	T **レジスタ**（TMS320C2x/2xx **の場合）または** TREG0**（**TMS320C5x **の場合）の内容とアドレス指定されたデータ・メモリ・ロケーションの内容を乗算し、結果を** P **レジスタに入れます。直前の積を（**PM **ステータス・ビットに従いシフトした後で）アキュムレータから減算します。**
MPYU	*dma*		√	√	√	**符号なしの乗算**
MPYU	{*ind*} [,*next ARP*]		√	√	√	T **レジスタ**（TMS320C2x/2xx **の場合）または** TREG0**（**TMS320C5x **の場合）の符号なしの内容とアドレス指定されたデータ・メモリ・ロケーションの内容を乗算し、結果を** P **レジスタに入れます。**

構文	1x	2x	2xx	5x	説明
NEG		√	√	√	**アキュムレータの符号反転** **アキュムレータの内容の符号を反転させます** (2 **の補数**)**。**
NMI			√	√	**マスク不能割り込み** **プログラム・カウンタをマスク不能割り込みベクトル・ロケーション** 24h **に強制設定します。**NMI **は、ハードウェアのマスク不能割り込みと同じ効果を持っています。**
NOP	√	√	√	√	**ノー・オペレーション** **オペレーションを何も実行しません。**
NORM **NORM** {*ind*}		√ √	√ √	√ √	**アキュムレータの内容を正規化** **アキュムレータの符号付き数値を正規化します。**
OPL [# *lk*,] *dma* **OPL** [# *lk*,] {*ind*} [,*next ARP*]				√ √	**DBMR またはロング・イミディエイト値との OR** **ロング・イミディエイト値が指定されている場合は、それと指定のデータ・メモリ・ロケーションの値との** OR **演算を実行します。それ以外の場合は、**DBMR **の内容が** OR **演算の** 2 **番目のオペランドになります。結果は、指定されたデータ・メモリ・ロケーションに書き戻されます。**
OR *dma* **OR** {*ind*} [,*next ARP*] **OR** # *lk* [,*shift*]	√ √	√ √	√ √ √	√ √ √	**アキュムレータとの OR** TMS320C1x、TMS320C2x**：アキュムレータの** 16 LSB **とアドレス指定されたデータ・メモリ・ロケーションの内容との** OR **演算を実行します。アキュムレータの** 16 MSB **は** 0 **と** OR **演算されます。** TMS320C2xx、TMS320C5x**：アキュムレータの** 16 LSB **とアドレス指定されたデータ・メモリ・ロケーションの内容との** OR **演算を実行します。シフトが指定されていれば、**OR **演算の前に左にシフトします。シフトされた値の上位および下位のビットは** 0 **として取り扱われます。**
ORB				√	**ACCB とアキュムレータを OR** ACCB **の内容とアキュムレータの内容との** OR **演算を実行します。**ORB **は、結果をアキュムレータに入れます。**
ORK # *lk* [,*shift*]		√	√	√	**シフト付きでイミディエイト値とアキュムレータを OR** 16 **ビット・イミディエイト値とアキュムレータの内容との** OR **演算を実行します。シフトが指定されていれば、**OR **演算の前に値を左にシフトします。シフトされた値の上位および下位のビットは** 0 **として取り扱われます。**

構文	1x	2x	2xx	5x	説明
OUT *dma, PA* **OUT** {*ind*}, *PA* [,*next ARP*]	√ √	√ √	√ √	√ √	**ポートへのデータの出力** **データ・メモリ・ロケーションから指定の** I/O **ポートへ** 16 **ビット値を書き出します。** TMS320C1x**：この命令は、最初のサイクルでポート・アドレスをアドレス・ライン** A2/PA2 ～ A0/PA0 **に入れます。同じサイクルの中で** $\overline{\text{WE}}$ **がローになります。その後で、データ・ワードをデータ・バス** D15 ～ D0 **に入れます。** TMS320C2x**、**TMS320C2xx**、**TMS320C5x**：**$\overline{\text{IS}}$ **ラインがローになって** I/O **アクセスを示します。**$\overline{\text{STRB}}$**、**R/$\overline{\text{W}}$**、**READY **の各タイミングは、外部データ・メモリ書き込みの場合と同じです。**
PAC	√	√	√	√	**アキュムレータに P レジスタをロード** P **レジスタの内容をアキュムレータにロードします。** TMS320C2x**、**TMS320C2xx**、**TMS320C5x**：ロードの前に、**PM **ステータス・ビットの指定に従って** P **レジスタをシフトします。**
POP	√	√	√	√	**スタックのトップをアキュムレータの下位にポップ** **スタックのトップの内容をアキュムレータの** 12（TMS320C1x **の場合）または** 16（TMS320C2x/2xx/5x **の場合）** LSB **にコピーし、スタックを** 1 **レベルだけポップします。アキュムレータの** MSB **はゼロに設定されます。**
POPD *dma* **POPD** {*ind*} [,*next ARP*]		√ √	√ √	√ √	**スタックのトップをデータ・メモリにポップ** **スタックのトップの値をアドレス指定されたデータ・メモリ・ロケーションに転送し、スタックを** 1 **レベルだけポップします。**
PSHD *dma* **PSHD** {*ind*} [,*next ARP*]		√ √	√ √	√ √	**データ・メモリ値をスタックにプッシュ** **アドレス指定されたデータ・メモリ・ロケーションをスタックのトップにコピーします。値をコピーする前に、スタックは** 1 **レベル下にプッシュされます。**
PUSH	√	√	√	√	**アキュムレータの下位をスタックにプッシュ** **アキュムレータの** 12（TMS320C1x **の場合）または** 16（TMS320C2x/2xx/5x **の場合）** LSB **の内容をハードウェア・スタックのトップにコピーします。値をコピーする前に、スタックは** 1 **レベル下にプッシュされます。**
RC		√	√	√	**キャリー・ビットのリセット** C **ステータス・ビットを** 0 **にリセットします。**
RET	√	√	√		**サブルーチンからの復帰** **スタックのトップの内容を** PC **にコピーし、スタックを** 1 **レベルだけポップします。**

構文	1x	2x	2xx	5x	説明
RET[*D*]				√	**サブルーチンからの復帰（遅延オプションあり）** **スタックのトップの内容を** PC **にコピーし、スタックを** 1 **レベルだけポップします。** **遅延分岐（**RETD**）を指定すると、復帰の前に、次の** 2 **つの命令ワード（**1 **ワード命令** 2 **個または** 2 **ワード命令** 1 **個）がフェッチされ実行されます。**
RETC [*cond*$_1$] [,*cond*$_2$] [,...]			√		**条件付き復帰** **指定の条件を満たしていれば、**RETC **は標準の復帰を実行します。条件のすべての組み合わせが有効なわけではありません。**
RETC[*D*] [*cond*$_1$] [,*cond*$_2$] [,...]				√	**条件付き復帰（遅延オプションあり）** **指定の条件を満たしていれば、**RETC **は標準の復帰を実行します。条件のすべての組み合わせが有効なわけではありません。** **遅延分岐（**RETCD**）を指定すると、復帰の前に、次の** 2 **つの命令ワード（**1 **ワード命令** 2 **個または** 2 **ワード命令** 1 **個）がフェッチされ実行されます。**
RETE				√	**割り込みを許可し、割り込みから復帰** **スタックのトップの内容を** PC **にコピーし、スタックを** 1 **レベルだけポップします。**RETE **は、グローバル割り込み許可ビットを自動的にクリアし、（割り込みの実行時にストアされた）シャドウ・レジスタをポップし戻します。シャドウ化されるのは以下のレジスタです。** ACC、ACCB、PREG、ST0、ST1、PMST、ARCR、INDX、TREG0、TREG1、TREG2。
RETI				√	**割り込みからの復帰** **スタックのトップの内容を** PC **にコピーし、スタックを** 1 **レベルだけポップします。**RETI **も、シャドウ・レジスタ（割り込みの実行時に格納された）をポップし 戻します。シャドウ化されるのは以下のレジスタです。** ACC、ACCB、PREG、ST0、ST1、PMST、ARCR、INDX、TREG0、TREG1、TREG2。
RFSM		√			**シリアル・ポート・フレーム同期モードのリセット** FSM **ステータス・ビットを** 0 **にリセットします。**
RHM		√	√	√	**ホールド・モードのリセット** HM **ステータス・ビットを** 0 **にリセットします。**
ROL		√	√	√	**アキュムレータの左ローテーション** **アキュムレータを左に** 1 **ビットだけローテーションします。**

構文	1x	2x	2xx	5x	説明
ROLB				√	**ACCB とアキュムレータの左ローテーション** ACCB **とアキュムレータを左に** 1 **ビットだけローテーションします。結果は、**65 **ビットのローテーションになります。**
ROR		√	√	√	**アキュムレータの右ローテーション** **アキュムレータを右に** 1 **ビットだけローテーションします。**
RORB				√	**ACCB とアキュムレータの右ローテーション** ACCB **とアキュムレータを右に** 1 **ビットだけローテーションします。結果は、**65 **ビットのローテーションになります。**
ROVM	√	√	√	√	**オーバーフロー・モードのリセット** OVM **ステータス・ビットを** 0 **にリセットします。これは、オーバーフロー・モードを禁止します。**
RPT *dma* **RPT** {*ind*} [,*next ARP*] **RPT** *# k* **RPT** *# lk*		√ √	√ √ √ √	√ √ √ √	**次の命令のリピート** TMS320C2x**：アドレス指定された値の** 8 LSB **を** RPTC **にロードします。**RPT **の後に続く命令は、**RPTC+1 **の回数だけ実行されます。** TMS320C2xx**、**TMS320C5x**：アドレス指定された値の** 8 LSB **または** 8 **ビットか** 16 **ビットのイミディエイト値を** RPTC **にロードします。**RPT **の後に続く命令は** *n* **回リピートされます（**n **は** RPTC+1**）。**
RPTB *pma*				√	**ブロック・リピート** RPTB **は、ループに対する制約なしに、**1 **ブロックの命令をメモリマップド・レジスタ** BRCR +1 **の回数だけリピートします。**BRCR **は、**RPTB **の実行前にロードされていなければなりません。**
RPTK *# k*		√	√	√	**イミディエイト値の指定による命令のリピート** 8 **ビット・イミディエイト値を** RPTC **にロードします。**RPTK **の後に続く命令は、**RPTC+1 **の回数だけ実行されます。**
RPTZ *# lk*				√	**アキュムレータと** P **レジスタのクリア後にリピート** **アキュムレータとプロダクト・レジスタをクリアし、**RPTZ **の後に続く命令を** n **回リピートします（**n = lk+1**）。**
RSXM		√	√	√	**符号拡張モードのリセット** SXM **ステータス・ビットを** 0 **にリセットします。これにより、以下の算術命令の場合に、シフトされたデータ値の符号拡張が抑制されます。** ADD、ADDT、ADLK、LAC、LACT、LALK、SBLK、SUB、SUBT。
RTC		√	√	√	**テスト/制御フラグのリセット** TC **ステータス・ビットを** 0 **にリセットします。**

構文	1x	2x	2xx	5x	説明
RTXM		√			**シリアル・ポート送信モードのリセット** TXM **ステータス・ビットを** 0 **にリセットします。これにより、シリアル・ポート送信セクションの構成は** FSX **が制御するモードになります。**
RXF		√	√	√	**外部フラグのリセット** XF **ピンと** XF **ステータス・ビットを** 0 **にリセットします。**
SACB				√	**アキュムレータを ACCB に格納** **アキュムレータの内容を** ACCB **にコピーします。**
SACH *dma* [,*shift*]	√	√	√	√	**シフト付きでアキュムレータの上位を格納**
SACH {*ind*} [,*shift* [,*next ARP*]]	√	√	√	√	**アキュムレータの内容をシフタにコピーします。内容全体を** 0**、**1**、または** 4 **ビット（**TMS320C1x **の場合）か、**0 **～** 7 **ビット（**TMS320C2x/2xx/5x **の場合）シフトし、シフトされた値の** 16 MSB **をアドレス指定されたデータ・メモリ・ロケーションにコピーします。アキュムレータは影響を受けません。**
SACL *dma*	√				**シフト付きでアキュムレータの下位を格納**
SACL *dma* [,*shift*]	√	√	√	√	TMS320C1x**：アキュムレータの** 16 LSB **をアドレス指定されたデータ・メモリ・ロケーションに格納します。**ARP **を変更する場合は、シフト値** 0 **を指定する必要があります。**
SACL {*ind*} [,*shift* [,*next ARP*]]		√	√	√	TMS320C2x**、**TMS320C2xx**、**TMS320C5x**：アキュムレータの** 16 LSB **をアドレス指定されたデータ・メモリ・ロケーションに格納します。シフトが指定されていれば、格納の前にアキュムレータの内容をシフトします。シフト値は、**0 **～** 7 **ビットです。**
SAMM *dma*				√	**アキュムレータをメモリマップド・レジスタに格納**
SAMM {*ind*} [,*next ARP*]				√	**アキュムレータの下位ワードをアドレス指定されたメモリマップド・レジスタに格納します。**DP **の現行の値または** AR（ARP）**の** 9 MSB **に関係なく、データ・アドレスの** 9MSB **はクリアされて使われます。**
SAR *AR*, *dma*	√	√	√	√	**補助レジスタを格納**
SAR *AR*, {*ind*} [,*next ARP*]	√	√	√	√	**指定の補助レジスタの内容をアドレス指定されたデータ・メモリ・ロケーションに格納します。**
SATH				√	**T レジスタ 1 の指定によりアキュムレータをバレルシフト** TREG1 **のビット** 4 **が** 1 **の場合は、アキュムレータを右に** 16 **ビットだけバレルシフトします。それ以外の場合は、アキュムレータは影響を受けません。**

構文	1x	2x	2xx	5x	説明
SATL				√	**T レジスタ 1 の指定によりアキュムレータをバレルシフト** TREG1 **の** 4 LSB **に指定された値だけ、アキュムレータを右にバレルシフトします。**
SBB				√	**ACCB をアキュムレータから減算** ACCB **の内容をアキュムレータから減算します。結果はアキュムレータに格納されます。アキュムレータ・バッファは影響を受けません。**
SBBB				√	**ボロー付きで ACCB をアキュムレータから減算** ACCB **の内容とキャリー・ビットの論理反転をアキュムレータから減算します。結果はアキュムレータに格納されます。アキュムレータ・バッファは影響を受けません。結果がボローを作成する場合は、キャリー・ビットをクリアします。**
SBLK *# lk* [,*shift*]		√	√	√	**シフト付きでロング・イミディエイト値をアキュムレータから減算** **アキュムレータからイミディエイト値を減算します。シフトが指定されていれば、減算の前に値が左にシフトされます。シフト時に、下位ビットに** 0 **が埋め込まれ、上位ビットは** SXM=1 **であれば符号拡張されます。**
SBRK *# k*		√	√	√	**ショート・イミディエイト値を補助レジスタから減算** **8 ビット・イミディエイト値を指定の補助レジスタから減算します。**
SC		√	√	√	**キャリー・ビットの設定** C **ステータス・ビットを** 1 **に設定します。**
SETC *control bit*			√	√	**制御ビットの設定** **指定の制御ビットを論理** 1 **に設定します。マスク可能割り込みは、**SETC **命令の実行後ただちに禁止されます。**
SFL		√	√	√	**アキュムレータを左にシフト** **アキュムレータの内容を左に** 1 **ビットだけシフトします。**
SFLB				√	**ACCB とアキュムレータを左にシフト** **アキュムレータと** ACCB **の結合を左に** 1 **ビットだけシフトします。**ACCB **の** LSB **は** 0 **にクリアされ、アキュムレータの** MSB **はキャリー・ビットにシフトされます。**
SFR		√	√	√	**アキュムレータを右にシフト** **アキュムレータの内容を右に** 1 **ビットだけシフトします。**SXM=1 **なら、**SFR **は算術右シフトを実施します。**SXM=0 **なら、**SFR **は論理右シフトを実施します。**

構文	1x	2x	2xx	5x	説明
SFRB				√	**ACCB とアキュムレータを右にシフト** **アキュムレータと** ACCB **の結合を右に** 1 **ビットだけシフトします。**ACCB **の** LSB **はキャリー・ビットにシフトされます。**SXM = 1 **なら、**SFRB **は算術右シフトを実施します。**SXM = 0 **なら、**SFRB **は論理右シフトを実施します。**
SFSM		√			**シリアル・ポート・フレーム同期モードの設定** FSM **ステータス・ビットを** 1 **に設定します。**
SHM		√	√	√	**ホールド・モードの設定** HM **ステータス・ビットを** 1 **に設定します。**
SMMR *dma*, # *lk* **SMMR** {*ind*}, # *lk* [,*next ARP*]				√ √	**メモリマップド・レジスタを格納** **データ・メモリ・アドレスの** 7 LSB **が指すメモリマップド・レジスタ値を、ロング・イミディエイト値でアドレス指定されたデータ・メモリ・ロケーションに格納します。**DP **の現行の値または** AR **（**ARP**）の上位** 9 **ビットに関係なく、メモリマップド・レジスタのデータ・メモリ・アドレスの** 9 MSB **はクリアされて使われます。**
SOVM	√	√	√	√	**オーバーフロー・モードの設定** OVM **ステータス・ビットを** 1 **に設定します。これで、オーバーフロー・モードが許可されます**(ROVM **命令は** OVM **をクリアします**)**。**
SPAC	√	√	√	√	**P レジスタをアキュムレータから減算** P **レジスタの内容をアキュムレータの内容から減算します。** TMS320C2x**、**TMS320C2xx**、**TMS320C5x**：減算の前に、**PM **ステータス・ビットの指定に従って** P **レジスタの内容をシフトします。**
SPH *dma* **SPH** {*ind*} [,*next ARP*]		√ √	√ √	√ √	**P レジスタの上位を格納** P **レジスタの上位ビットを（**PM **ステータス・ビットの指定に従いシフトした後で）アドレス指定されたデータ・メモリ・ロケーションに格納します。**
SPL *dma* **SPL** {*ind*} [,*next ARP*]		√ √	√ √	√ √	**P レジスタの下位を格納** P **レジスタの下位ビットを（**PM **ステータス・ビットの指定に従いシフトした後で）アドレス指定されたデータ・メモリ・ロケーションに格納します。**
SPLK # *lk*, *dma* **SPLK** # *lk*, {*ind*} [,*next ARP*]			√	√ √	**ロング・イミディエイト値の格納** **完全な** 16 **ビット・パターンをメモリ・ロケーションに書き込みます。パラレル論理装置** (PLU) **は**ALU **とは独立してこのビット操作をサポートしているので、アキュムレータは影響を受けません。**

構文	1x	2x	2xx	5x	説明
SPM *2-bit constant*		√	√	√	**P レジスタ出力シフト・モードの設定** 2 **ビット・イミディエイト値を** ST1 **の** PM **フィールドにコピーします。これにより、**P **レジスタのシフト処理が以下のように制御されます。** PM = 00_2 **乗算器出力はシフトされません。** PM = 01_2 **乗算器出力は左に** 1 **桁シフトされ、ゼロで埋められます。** PM = 10_2 **乗算器出力は左に** 4 **桁シフトされ、ゼロで埋められます。** PM = 11_2 **乗算器出力は右に** 6 **桁シフトされ、符号拡張されます。** LSB **は失われます。**
SQRA *dma* **SQRA** {*ind*} [,*next ARP*]		√ √	√ √	√ √	**二乗と直前の積の累算** P **レジスタの内容を（**PM **ステータス・ビットの指定に従いシフトした後で）アキュムレータに加算します。そして、アドレス指定されたデータ・メモリ・ロケーションの内容を** T **レジスタ（**TMS320C2x/2xx **の場合）または** TREG0**（**TMS320C5x **の場合）にロードし、値を二乗し、結果を** P **レジスタに格納します。**
SQRS *dma* **SQRS** {*ind*} [,*next ARP*]		√ √	√ √	√ √	**二乗と直前の積の減算** P **レジスタの内容を（**PM **ステータス・ビットの指定に従いシフトした後で）アキュムレータから減算します。そして、アドレス指定されたデータ・メモリ・ロケーションの内容を** T **レジスタ（**TMS320C2x/2xx **の場合）または** TREG0**（**TMS320C5x **の場合）にロードし、値を二乗し、結果を** P **レジスタに格納します。**
SST *dma* **SST** {*ind*} [,*next ARP*]	√ √	√ √	√ √	√ √	**ステータス・レジスタを格納** ST**（**TMS320C1x **の場合）または** ST0**（**TMS320C2x/2xx/5x**）の内容をアドレス指定されたデータ・メモリ・ロケーションに格納します。**
SST *# n dma* **SST** *# n* {*ind*} [,*next ARP*]			√ √	√ √	**ステータス・レジスタ n を格納** ST*n* **をデータ・メモリに格納します。**
SST1 *dma* **SST1** {*ind*} [,*next ARP*]		√ √	√ √	√ √	**ステータス・レジスタ ST1 を格納** ST1 **の内容をアドレス指定されたデータ・メモリ・ロケーションに格納します。**
SSXM		√	√	√	**符号拡張モードの設定** SXM **ステータス・ビットを** 1 **に設定します。これにより符号拡張が許可されます。**
STC		√	√	√	**テスト/制御フラグの設定** TC **フラグを** 1 **に設定します。**

構文	1x	2x	2xx	5x	説明
STXM		√			シリアル・ポート送信モードの設定 TXM ステータス・ビットを 1 に設定します。
SUB *dma* [,*shift*]	√	√	√	√	シフト付きでアキュムレータから減算
SUB {*ind*} [,*shift* [,*next ARP*]]	√	√	√	√	TMS320C1x、TMS320C2x：アドレス指定されたデータ・メモリ・ロケーションの内容をアキュムレータから減算します。シフトが指定されていれば、減算の前に値を左にシフトします。シフト時に、下位ビットに 0 が埋め込まれ、SXM＝1 であれば上位ビットは符号拡張されます。 TMS320C2xx、TMS320C5x：アドレス指定されたデータ・メモリ・ロケーションの内容または 8 ビットか 16 ビットの定数をアキュムレータから減算します。シフトが指定されていれば、減算の前にデータを左にシフトします。シフト時に、下位ビットに 0 が埋め込まれ、SXM＝1 であれば上位ビットは符号拡張されます。
SUB *# k*			√	√	
SUB *# lk* [,*shift2*]			√	√	
SUBB *dma*		√	√	√	キャリー・ビット付きでアキュムレータから減算
SUBB {*ind*} [,*next ARP*]		√	√	√	アドレス指定されたデータ・メモリ・ロケーションの内容とキャリー・ビットの値をアキュムレータから減算します。キャリー・ビットは通常の影響を受けます。
SUBC *dma*	√	√	√	√	条件付き減算
SUBC {*ind*} [,*next ARP*]	√	√	√	√	条件付き減算を実行します。SUBC は除算に使用できます。
SUBH *dma*	√	√	√	√	アキュムレータの上位から減算
SUBH {*ind*} [,*next ARP*]	√	√		√	アドレス指定されたデータ・メモリ・ロケーションの内容をアキュムレータの 16 MSB から減算します。アキュムレータの 16 LSB は影響を受けません。
SUBK *# k*		√	√	√	アキュムレータからショート・イミディエイト値を減算 8 ビット・イミディエイト値をアキュムレータから減算します。データは 8 ビットの正の数として取り扱われます。符号拡張は抑制されます。
SUBS *dma*	√	√	√	√	アキュムレータの下位からの符号なし減算
SUBS {*ind*} [,*next ARP*]	√	√	√	√	アドレス指定されたデータ・メモリ・ロケーションの内容をアキュムレータから減算します。データは 16 ビットの符号なしの数として取り扱われます。符号拡張は抑制されます。

構文	1x	2x	2xx	5x	説明
SUBT *dma*		√	√	√	**T レジスタ指定のシフト付きでアキュムレータから減算**
SUBT {*ind*} [,*next ARP*]		√	√	√	T **レジスタの** 4 LSB（TMS320C2x/2xx **の場合）または** TREG1（TMS320C5x **の場合）の指定に従ってデータ・メモリ値を左にシフトし、結果をアキュムレータから減算します。シフト時に、下位ビットに** 0 **が埋め込まれ、**SXM＝1 **であれば上位ビットは符号拡張されます。**
SXF		√	√	√	**外部フラグを設定** XF **ピンと** XF **ステータス・ビットを** 1 **に設定します。**
TBLR dma	√	√	√	√	**テーブル読み取り**
TBLR {*ind*} [,*next ARP*]	√	√	√	√	**プログラム・メモリからデータ・メモリ・ロケーションにワードを転送します。プログラム・メモリ・アドレスは、アキュムレータの** 12（TMS320C1x **の場合）または** 16（TMS320C2x/2xx/5x **の場合）** LSB **です。**
TBLW *dma*	√	√	√	√	**テーブル書き込み**
TBLW {*ind*} [,*next ARP*]	√	√	√	√	**データ・メモリからプログラム・メモリ・ロケーションにワードを転送します。プログラム・メモリ・アドレスは、アキュムレータの** 12（TMS320C1x **の場合）または** 16（TMS320C2x/2xx/5x **の場合）** LSB **です。**
TRAP		√	√	√	**ソフトウェア割り込み** TRAP **命令は、プログラム制御をプログラム・メモリ・アドレス** 30h（TMS3 20C2x **の場合）または** 22h（TMS320C 2xx/5x **の場合）に渡し、**PC＋1 **をハードウェア・スタックにプッシュするソフトウェア割り込みです。アドレス** 30h **または** 22h **の命令には、制御を** TRAP **ルーチンに転送するための分岐命令を入れることができます。**PC＋1 **をスタックに入れることで、**RET **命令で復帰** PC **をポップし、**TRAP **ルーチンから復帰できるようになります。**
XC *n* [,$cond_1$] [,$cond_2$] [,...]				√	**条件付き実行** 1≦*n*≦2 **の場合に、次の** *n* **個の命令ワードを条件付きで実行します。条件のすべての組み合わせが有効とは限りません。**
XOR *dma*	√	√	√	√	**アキュムレータと XOR**
XOR {*ind*} [,*next ARP*]	√	√	√	√	TMS320C1x**、**TMS320C2x**：アドレス指定されたデータ・メモリ・ロケーションの内容とアキュムレータの** 16 LSB **で** XOR **演算を実行します。**MSB **は影響を受けません。**
XOR # *lk* [,*shift*]			√	√	TMS320C2xx**、**TMS320C5x**：アドレス指定されたデータ・メモリ・ロケーションの内容または** 16 **ビット・イミディエイト値とアキュムレータとの** XOR **演算を実行します。シフトが指定されていれば、**XOR **演算の前に値が左にシフトされます。シフトされた値の上位および下位のビットは** 0 **として取り扱われます。**

構文	1x	2x	2xx	5x	説明
XORB				√	**ACCB とアキュムレータと XOR** **アキュムレータの内容と** ACCB **の内容との** XOR **演算を実行します。結果はアキュムレータに入れられます。**
XORK *# lk* [,*shift*]		√	√	√	**シフト付きでイミディエイト値とアキュムレータとを XOR** 16 **ビット・イミディエイト値とアキュムレータとの** XOR **演算を実行します。シフトが指定されていれば、**XOR **演算の前に値が左にシフトされます。シフトされた値の上位および下位のビットは** 0 **として取り扱われます。**
XPL [*# lk*],*dma* **XPL** [*# lk*,] {*ind*} [,*next ARP*]				√ √	**ロング・イミディエイト値または DBMR とアドレス指定されたデータ・メモリ値との XOR** **ロング・イミディエイト値が指定されていれば、それとアドレス指定されたデータ・メモリ値との**XOR **演算を実行します。それ以外の場合は、**DBMR **とアドレス指定されたデータ・メモリ値との**XOR **演算を実行します。そして、結果をデータ・メモリ・ロケーションに書き戻します。アキュムレータの内容は影響を受けません。**
ZAC	√	√	√	√	**アキュムレータの 0 クリア** **アキュムレータの内容を** 0 **にクリアします。**
ZALH *dma* **ZALH** {*ind*} [,*next ARP*]	√ √	√ √	√ √	√ √	**アキュムレータの下位を 0 にし、上位にロード** **アキュムレータの** 16 LSB **を** 0 **にクリアし、アドレス指定されたデータ・メモリ・ロケーションの内容をアキュムレータの** 16 MSB **にロードします。**
ZALR *dma* **ZALR** {*ind*} [,*next ARP*]		√ √	√ √	√ √	**アキュムレータの下位を 0 にし、上位に丸め付きでロード** **アドレス指定されたデータ・メモリ・ロケーションの内容をアキュムレータの** 16 MSB **にロードします。値は** 1/2 LSB **により丸められます。つまり、アキュムレータの** 15 LSB（0 ～ 14）**はクリアされ、ビット** 15 **が** 1 **に設定されます。**
ZALS *dma* **ZALS** {*ind*} [,*next ARP*]	√ √	√ √	√ √	√ √	**アキュムレータを 0 クリアし、下位に符号拡張なしロード** **アドレス指定されたデータ・メモリ・ロケーションの内容をアキュムレータの** 16 LSB **にロードします。**16 MSB **がゼロにされます。データは** 16 **ビットの符号なしの数として取り扱われます。**
ZAP				√	**アキュムレータとプロダクト・レジスタの 0 クリア** **アキュムレータとプロダクト・レジスタがゼロにされます。**ZAP **命令は、積和リピートの準備をスピード・アップします。**
ZPR				√	**プロダクト・レジスタの 0 クリア** **プロダクト・レジスタがゼロにクリアされます。**

第 6 章

マクロ言語

アセンブラでは、ユーザが自分自身の「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、この機能を使うと便利です。マクロ言語を使うと以下のようなことができます。

- ユーザ自身のマクロを定義し、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。
- マクロ内に条件及び、繰り返しブロックを定義する。
- マクロ内で文字列を操作する。
- 展開リストを制御する。

この章では次の事項について説明します。

項目 ページ

6.1 マクロの使用法

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、というように変えることができます。この方法で短くて簡潔なソース・プログラムを書くことができます。

マクロの使用は 3 つの手順に分けて考えることができます。

手順 1: **マクロの定義。**プログラムでマクロを使うためには、まずマクロを定義することが必要です。マクロを定義するには 2 つの方法があります。

- ❏ マクロは、**ソース・ファイル**の先頭で定義することも、**.include/.copy** ファイルで定義することも可能です。詳しい説明は 6.2 節を参照してください。
- ❏ マクロは**マクロ・ライブラリ**で定義することもできます。マクロ・ライブラリとは、アーカイバを使って作成したアーカイブ・フォーマットのファイルの集合を指します。アーカイブ・ファイル（マクロ・ライブラリ）の個々のメンバーには、そのメンバー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 疑似命令を使います。マクロ・ライブラリについては 6-12 ページの 6.4 節に説明があります。

手順 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・プログラム中で、オペコードとして使用することができます。これをマクロ呼び出しと言います。

手順 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、アセンブラがそれを展開します。展開の際に、アセンブラはマクロ・パラメータに引数を渡し、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、マクロ展開はリスト・ファイルに出力されます。.mnolist 疑似命令を使うと、展開リストの作成を止めることができます。詳しい説明については 6-17 ページの 6.8 節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、オペコード・テーブルにそのマクロ名を入れます。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、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。この機能により、新しい命令を追加すると同様に疑似命令や命令の機能を拡張することができます。

6.2 マクロの定義

マクロはプログラム内のどこでも定義することができますが、必ず使用する前に定義することが必要です。マクロはソース・ファイルの先頭で定義することも、.include/.copy ファイルで定義することもできます。また、マクロ・ライブラリで定義することもできます。マクロ・ライブラリについての詳しい説明は、6-12 ページの 6.4 節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については、6-18 ページの 6.9 節で詳しく説明します。

マクロの定義は次のような形式の一連のソース文です。

```
macname   .macro   [parameter1] [,parameter2] ... [,parametern]
          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オペコード・テーブルに入れ、すでに同じ名前を持った命令やマクロ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する疑似命令です。.macro は、必ずオペコード・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わ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 6-5 ページの 6.3 節で説明します。
model statements	マクロが呼び出される度に実行される命令、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われます。
[.mexit]	「goto .endm」として機能します。 エラーならばマクロ展開を確実に失敗させる、といった場合に .mexit 疑似命令を使うと便利です。
.endm	マクロ定義を終了します。

例 6 - 1 に、マクロの定義、呼び出し、展開の例を示します。

例 6 - 1. マクロの定義、呼び出し、展開

マクロ定義： 次のコードは 3 つのパラメータを持つマクロ add3 を定義しています。

```
 3 *   add3   p1, p2, p3
 4 *          p3 = p1 + p2 + p3
 5
 6     add3   .macro p1, p2, p3
 7
 8            lac    p1
 9            add    p2
10            add    p3
11            sacl   p3
12            .endm
```

マクロ呼び出し：次のコードは、3 つの引数を持つ add3 マクロを呼び出しています。

```
13
14 0000       add3   a, b, c
```

マクロ展開：次のコードは、マクロ呼び出しのマクロ定義による置換を示しています。アセンブラは、(マクロ呼び出しで指定された) 引数をパラメータ (置換シンボル) に渡します。

```
   0000 2000 lac    a
   0001 0001 add    b
   0002 0002 add    c
   0003 6002 sacl   c

15
16
17 *   Reserve space for vars
18 0000       .bss   a,1
19 0001       .bss   b,1
20 0002       .bss   c,1
```

マクロ定義にコメントを含めて、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したい場合には、アスタリスクかセミコロンを使います。マクロ・コメントに関する詳しい説明は、6-16 ページの 6.7 節を参照してください。

6.3 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けれども、その度にデータが変わるような場合には、そのデータにパラメータを割り当てることができます。この方法で、マクロを呼び出すたびにそのマクロに情報を渡すことができます。マクロ言語では、**置換シンボル**と呼ばれる特別なシンボルがサポートされており、このシンボルをマクロ・パラメータに使うことができます。この章では、**マクロ・パラメータ**という用語と**置換シンボル**という用語は同じ意味で使用されています。

6.3.1 置換シンボル

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ために、マクロの外部で使うこともできます。

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、ドル記号を使うことができます。

マクロ・パラメータとして使わ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を持ちません。1 つのマクロにつき 32 までのローカルな置換シンボルを定義することができます（.var 疑似命令を使って定義する置換シンボルを含めて）。.var 疑似命令についての詳しい説明は、6-11 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。各引数の文字列が対応するパラメータに割り振られます。対応する引数を持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やコンマやセミコロンをパラメータに渡す場合は、その引数を引用符で囲まなければなりません。

アセンブリ時に、アセンブラはまず置換シンボル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 6 - 2 は、引数の数が異なったマクロ展開の例を示したものです。

例 6 - 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 Parms 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
```

6.3.2　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** および **.eval** 疑似命令を使って操作することができます。

❑ **.asg** 疑似命令は、文字列を置換シンボルに割り当てます。

.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。引用符がない場合には、アセンブラは最初のコンマまでの文字を読み取ってその前後の空白を取り去ります。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 6 - 3 に置換シンボルに割り当てられた文字列を示します。

例 6 - 3.　.asg 疑似命令の使用法

```
      .asg   ar0 ,FP            ; frame pointer
      .asg   *+  ,Ind           ; indirect addressing
      .asg   *bro+,Rc_Prop      ; reverse carry propagation
      .asg   ”””string”””,strng ; string
      .asg   ”a,b,c” ,parms     ; parameters
      mar    Ind,FP             ; mar *+, AR0
```

- **.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、式を評価し、その結果得られた文字列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は、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 6 - 4 **は、置換シンボルへの算術の実行を示したものです。**

例 6 - 4. .eval **疑似命令の使用法**

```
      .asg      1,counter
      .loop     100
      .word     counter
      .eval     counter + 1,counter
      .endloop
```

例 6 - 4 **では、**.asg **疑似命令を** .eval **疑似命令(**.eval 1, counter**) 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 **はどちらを使ってもかまいません。 ただし、式から値を計算する場合には、**.eval **疑似命令を使う必要があります。**.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 **疑似命令は式を評価し、置換シンボルに等しい文字列を割り当てます。**

6.3.3 組み込み置換シンボル関数

以下の組み込み置換シンボル関数を使うと、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、必ず値を返し、式の中で使うこともできます。組み込み置換シンボル関数はアセンブリの条件式に使うと非常に便利です。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6 - 1 の関数定義では、 *a* と *b* がパラメータで、置換シンボルまたは文字列定数を表します。以下で使われている**文字列**という用語は、パラメータの文字列値を表します。

表6 - 1. 置換シンボル関数定義

関数	戻り値
$symlen(*a*)	文字列 *a* の長さ
$symcmp(*a*,*b*)	*a* < *b* の場合は < 0 *a* = *b* の場合は 0 *a* > *b* の場合は > 0
$firstch(*a*,*ch*)	文字列 *a* の中で文字定数 *ch* が最初に現われる位置
$lastch(*a*,*ch*)	文字列 *a* の中で文字定数 *ch* が最後に現われる位置
$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* の先頭のメンバーが文字列 *a* に割り当てられます。 0: *b* が空ストリングの場合
$iscons(*a*)	1: 文字列 *a* が 2 進数定数の場合 2: 文字列 *a* が 8 進定数の場合 3: 文字列 *a* が 16 進定数の場合 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

† 定義済みレジスタ名についての詳しい情報は、3-18 ページの 3.9 節を参照してください。

例 6 - 5 は、組み込み置換シンボル関数を示したものです。

例 6 - 5. 組み込み置換シンボル関数の使用

```
        .asg   label, a                    ; a = label
        .if    ($symcmp(a,"label") = 0)    ; evaluates to true
        sub    a
        .endif
        .asg   "x,y,z"    ,   list         ; list = x,y,z
        .if    ($ismember(a, list))        ; a = x  list = y,z
        sub    a                           ; sub      x
        .endif
```

6.3.4 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に出会うまで、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 6 - 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 6 - 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


*      add x         ; recursive expansion
```

6.3.5 強制置換

場合によっては、アセンブラが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うと、シンボルの文字列を強制的に置換することができます。単純にシンボルをコロンではさめば、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を示します。

: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。

強制置換シンボルはマクロの中でのみ使用することができ、また強制置換演算子を別の強制置換演算子でネストすることはできません。

例 6 - 7 は、強制置換演算子がどのように使用されるかを示しています。

例 6 - 7. 強制置換演算子の使用例

```
force  .macro x
       .asg      0,x
       .loop  8  ;.loop/.endloop are discussed on page 6-15
AUX:x: .set   x
       .eval  x+1,x
       .endloop
       .endm
```

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
```

6.3.6 添字付き置換シンボルの個々の文字へのアクセス法

マクロでは、添字付き置換シンボルを使って置換シンボルの個々の文字（部分文字列）にアクセスすることができます。明確に指示するために強制置換演算子を使う必要があります。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、評価結果は 1 文字の文字列に表わされます。

- :*symbol* (*well - defined expression*$_1$, *well - defined expression*$_2$):

  この方法では、expression$_1$ が部分文字列の開始位置を示し、expression$_2$ が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することができます。**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 6 - 8 と例 6 - 9 に添字付き置換シンボルとともに使われる組み込み置換シンボル関数の例を示します。

例 6 - 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    a
          .var      tmp
          .asg      :a(1):,tmp
          .if       $symcmp(tmp,"#") = 0
          addk      a
          .elseif   $symcmp(tmp,"*") = 0
          add       a
          .else
          .emsg     "Bad Macro Parameter"
          .endif
          .endm

          addx      #100      ;macro call
          addx      *+        ;macro call
```

例 6 - 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・イミディエイトを取り扱うことができるようにしています。

例 6 - 9.　**添字付き置換シンボルを使っ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     start,1
          .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
          .word     pos
```

例 6 - 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 **を見つけ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。**

6.3.7　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う場合には、.var 疑似命令を使います。マクロ 1 **つにつき（パラメータを含めて）**32 **までのローカル・マクロ置換シンボルを定義することができます。**.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、展開後は消えてしまいます。**

.var　*sym*$_1$*[,sym*$_2$*] ... [,sym*$_n$*]*

.var **疑似命令は例** 6 - 8 **と例** 6 - 9 **で使われています。**

6.4 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う必要があります。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。例を示します。

マクロ名	マクロ・ライブラリでのファイル名
`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います。.mlib の構文は次のとおりです。

.mlib *macro library filename*

アセンブラは .mlib 疑似命令を検出すると、ライブラリを開き、その内容のテーブルを作成します。アセンブラは、ライブラリ内の個々のメンバ名をライブラリ・エントリとしてオペコード・テーブルに入れます。もし同じ名前のオペコードやマクロがすでにあれば再定義されてしまいます。これらのマクロの内の 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方法で展開します。ライブラリ・エントリ展開のリスト作成は、.mlist 疑似命令を使って制御できます。.mlist 疑似命令についての詳しい説明は、6-17 ページの 6.8 節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るマクロだけです。1 つのマクロに対する抽出は 1 回しか行われません。.mlib 疑似命令についての詳しい説明は、4-53 ページの説明を参照してください。

アーカイバを使って必要なファイルをアーカイブに入れるだけで、マクロ・ライブラリを作成することができます。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期待する点を除けば、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望ましくない結果が生じることがあります。

6.5 マクロでの条件アセンブリの使用法

条件アセンブリ疑似命令には、**.if/.elseif/.else/.endif** および **.loop/.break/.endloop** があります。条件アセンブリ疑似命令は 32 レベルまでネストすることができます。条件ブロックの書式は次のとおりです。

.if *well-defined expression*

[.elseif *well-defined expression***]**

[.else]

.endif

.elseif と .else 疑似命令の指定は任意で、条件アセンブリ・コード・ブロック内で 2 回以上使うこともできます。この 2 つが省略されて、かつ .if 式が偽 (ゼロ) 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 疑似命令の詳しい説明については、4-45 ページを参照してください。

.loop/.break/.endloop 疑似命令を使うとコード・ブロックを繰り返してアセンブルすることができます。繰り返し可能ブロックの書式は次のとおりです。

.loop [*well-defined expression*]

[.break [*well-defined expression*]**]**

.endloop

.loop 疑似命令の式が評価されてループ・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、アセンブラが式が真 (非ゼロ) になる .break 疑似命令を検出しない限り、ループ・カウントはデフォルトの 1024 となります。.loop/.break/.endloop 疑似命令の詳しい説明については、4-52 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。式が評価されて偽と判断された場合は、ループが継続します。アセンブラは、.break 式が真と評価された場合、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 疑似命令の後ろのコードのアセンブルへ移ります。

例 6 - 10、例 6 - 11、例 6 - 12 は、.loop/.break/.endloop 疑似命令、適切なネストが行われている条件アセンブリ疑似命令、組み込み置換シンボル関数が条件アセンブリ・コード・ブロックでどのように使われているかを示したものです。

例 6 - 10. .loop/.break/.endloop 疑似命令

```
       .asg   1,x
       .loop            ;  ”infinite”

       .break (x == 10) ;  if x == 10, quit loop/break with
*                                              expression

       .eval  x+1,x
       .endloop
```

例 6 - 11. ネストされた条件アセンブリ疑似命令

```
       .asg   1,x
       .loop         ;  ”infinite”

       .if (x == 10);   if x == 10 quit loop
       .break        ;  force break
       .endif

       .eval   x+1,x
       .endloop
```

例 6 - 12. 条件アセンブリ・コード・ブロックで使われている組み込み置換シンボル関数

```
       .fcnolist
*
*Subtract Double
*
subx   .macro a            ; subtract double

       .if $symcmp(a,”*”)
       subh *+             ; subtract high
       subs *-             ; subtract low
       .elseif $symcmp(a,”*+”)
       subh *+             ; subtract high
       subs *+             ; subtract low
       .elseif $symcmp(a,”*-”)
       subs *-             ; subtract low
       subh *-             ; subtract high
       .else
       subh a              ; subtract high
       subs a + 1             ; subtract low
       .endif

       .endm

*Macro Call
       subx *+
```

条件アセンブリ疑似命令についての詳しい説明は、4-15 ページの 4.7 節を参照してください。

6.6 マクロでのラベルの使用法

アセンブリ言語プログラムで使われるラベルは、どれも一意である必要があります。マクロで使われるラベルについても同様です。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。しかし 2 回以上定義したラベルは不正です。 マクロ言語は、ラベルが必ず一意になるように、マクロでラベルを定義する方法を提供しています。単にラベルの最後に疑問符を付ければ、アセンブラがその疑問符を一意の数字に変えてくれます。マクロを展開しても、リスト・ファイル上でこの一意の数字を見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ついて現れます。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。一意のラベルの構文は次のとおりです。

*label***?**

例 6 - 13 次の例はマクロでの一意のラベルの作成法を示したものです。

例 6 - 13. マクロでの一意のラベル

```
;  define macro

MIN       .macro    a,b

          lac       a
          sub       b
          blz       m1?
          lac       b
          b         m2?
m1?       lac       a
m2?
          .endm

;  call macro

          MIN       x,#100

;  expand macro

          lac       x
          sub       #100
          blz       m1?
          lac       #100
          b         m2?
m1?       lac       x
m2?
```

一意の添字が使用できるようにするは、最大ラベル長を短くします。たとえば、マクロが展開される回数が 10 回より少ない場合は、最大ラベル長は 126 文字です。マクロが 10 〜 99 回の範囲内で展開される場合は、最大ラベル長は 125 文字です。この一意の添字を持つラベルは、クロス・リスティング・ファイルに示されています。

6.7 マクロでのメッセージの作成

マクロ言語では、ユーザが自分のアセンブリ時エラー・メッセージと警告メッセージを定義できるように 3 つの疑似命令がサポートされています。この 3 つの疑似命令は、自分に必要な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。このカウントによって自分のコードの問題点を知ることができるので、特にデバッグのときに役に立ちます。

.emsg は、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、アセンブラが行うのと同じようにエラーを発生させます。つまり、エラー・カウントを 1 つずつ増加させ、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。

.wmsg は、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、警告カウントをインクリメントす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。

.mmsg は、警告メッセージまたはアセンブリ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・カウントの設定をせず、オブジェクト・ファイルの作成も禁止しない点が異なります。

マクロ・コメントは、マクロの定義で使われるコメントですが、**マクロ展開の中には現われません。**1 カラム目の感嘆符(!)が、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 6 - 14 は、マクロでのユーザ・メッセージとマクロ展開には現れないマクロ・コメントを示しています。

例 6 - 14. マクロでのメッセージの作成

```
TEST   .MACRO    x,y
!
!  This macro checks for the correct number of parameters.
!  The macro generates an error message
!  if x and y are not present.
!
       .if    ($symlen(x) == 0|$symlen(y) == 0)  ; Test for
                                                 ; proper input
       .emsg  "ERROR - missing parameter in call to TEST"
       .mexit
       .else
         .
         .
       .endif
       .if
           .
         .
       .endif
       .endm

   1 error, no warnings
```

6.8 出力リストのフォーマット

マクロ、置換シンボル、条件アセンブリ疑似命令は、情報を隠してしまうことがあります。この隠された情報を見る必要がある場合もあるので、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、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 組の疑似命令を使って、この情報のリスト出力を制御することができます。

- **マクロ展開とループ展開のリスト**

  .mlist は、マクロと .loop/.endloop ブロックの展開リスト作成を有効にします。.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 は、マクロ展開と .loop/.endloop ブロックのリスト作成を抑制します。

  マクロ展開とループ展開のリストの場合、.mlist がデフォルトです。

- **偽の条件ブロックのリスト**

  .fclist は、コードを作成しないすべての条件ブロック（偽の条件ブロック）をアセンブラがリスト・ファイルに含めるようにします。条件ブロックは、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 は、偽の条件ブロックのリスト作成を抑制します。条件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、.elsif、.else、.endif 疑似命令はリストには出力されません。

  偽の条件ブロックのリストの場合、.fclist がデフォルトです。

- **置換シンボル展開リスト**

  .sslist は、リスト内で置換シンボルの展開リスト作成を有効にします。置換シンボルの展開のデバッグに便利です。展開された行は実際のソース行の下に示されます。

  .ssnolist は、リスト内の置換シンボルの展開リスト作成を抑制します。

  置換シンボル展開リストの場合、.ssnolist がデフォルトです。

- **疑似命令のリスト**

  .drlist は、アセンブラに、すべての疑似命令をリスト・ファイルに出力させます。

  .drnolist は、リスト・ファイルへの以下の疑似命令の出力を抑制します。.asg、.eval、.var、.sslist、.mlist、.fclist、.ssnolist、.mnolist、.fcnolist、.emsg、.wmsg、.mmsg、.length、.width、.break

  疑似命令のリストの場合、.drlist がデフォルトです。

6.9 再帰的なマクロとネストされたマクロの使用法

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がサポートされています。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 レベルまでネストすることができます。再帰的なマクロを使うと、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場合は、アセンブラがパラメータの動的な範囲設定を使うので、マクロ・パラメータに渡す引数に十分な注意が必要です。これは、呼び出されたマクロが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 6 - 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下さい。ただし、out_block マクロ内の x と z には、in_block マクロからアクセスすることができます。

例 6 - 15. ネストされたマクロの使用法

```
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 6 - 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 はイミディエイト値です。結果はデータ・メモリ・アドレス loc に入れられます。 fact マクロは fact1 を呼び出すことによって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 6 - 16. 再帰的なマクロの使用例

```
fact   .macro n,loc     ; n is an integer constant
                        ; loc memory address = n!
       .if    n < 2     ; 0! = 1! = 1
       lack   1
       sacl   loc
       .else
       lack   n         ; n >= 2 so, store n at loc
       sacl   loc       ; decrement n, and do the
       .eval  n - 1,n   ;      factorial of n - 1
       fact1            ; call fact with current
                        ;      environment
       .endif

       .endm

fact1  .macro

       .if    n > 1
       lt     loc       ; multiply present factorial
       mpyk   n         ;      by present position
       pac
       sacl   loc       ; save result
       .eval  n - 1,n   ; decrement position
       fact1            ; recursive call
       .endif

       .endm
```

6.10 マクロ疑似命令のまとめ

表6 - 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 に行きます。
.endm	マクロ定義を終了します。

表6 - 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6 - 4. 条件アセンブリ

ニーモニックと構文	説明
.if *well-defined expression*	条件が真であれば、コード・ブロックをアセンブルします。
.elseif *well-defined expression*	.if 条件が偽で、.elseif 条件が真の場合に、コード・ブロックをアセンブルします。.elseif 構造は必須ではありません。
.else	.if 条件が偽であればコード・ブロックをアセンブルします。.else 構造は必須ではありません。
.endif	.if コード・ブロックを終了します。
.loop [*well-defined expression*]	コード・ブロックの繰り返し可能なアセンブリを開始します。
.break [*well-defined expression*]	条件が真であれば .loop のアセンブリを終了します。.break構造は必須ではありません。
.endloop	.loop コード・ブロックを終了します。

表6 - 5.　アセンブリ時メッセージの作成

ニーモニックと構文		説明
.emsg	string	**エラー・メッセージを標準出力に送ります。**
.wmsg	string	**警告メッセージを標準出力に送ります。**
.mmsg	string	**アセンブリ時メッセージを標準出力に送ります。**

表6 - 6.　リストのフォーマット

ニーモニックと構文	説明
.**drlist**	**すべての疑似命令行のリストを出力します（デフォルト）。**
.drnolist	**以下の疑似命令行のリスト出力を禁止します。 .asg、.eval、.var、.sslist、.mlist、.fclist、.ssnolist、 .mnolist、.fcnolist、.emsg、.mmsg、.wmsg、.length、 .width、.break**
.fclist	**偽の条件コード・ブロックのリスト作成を認めます。(デフォルト)。**
.fcnolist	**偽の条件コード・ブロックのリスト作成を禁止します。**
.mlist	**マクロ・リストの作成を認めます (デフォルト)**
.mnolist	**マクロ・リストの作成を禁止します。**
.sslist	**置換シンボルの展開リスト作成を認めます。**
.ssnolist	**置換シンボルの展開リスト作成を禁止します (デフォルト)。**

第 7 章

アーカイバの説明

アーカイバを使うと、いくつかのファイルを集めて 1 つのアーカイブ・ファイルにすることができます。たとえば、いくつかのマクロを集めて 1 つのマクロ・ライブラリにすることができます。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います。アーカイバを使って、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。

この章では以下の内容について説明します。

項目 ページ

7.1 アーカイバの概要

TMS320C1x/C2x/C2xx/C5x アーカイバを使うと、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することができます。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、アーカイバを使ってさらにファイルをアーカイブに追加したり、既存のメンバを削除、置換したり、メンバの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ででも作成できます。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・ファイルを含むライブラリを使うことができます。リンカは個々のオブジェクト・ファイルを含むライブラリを使うことができます。

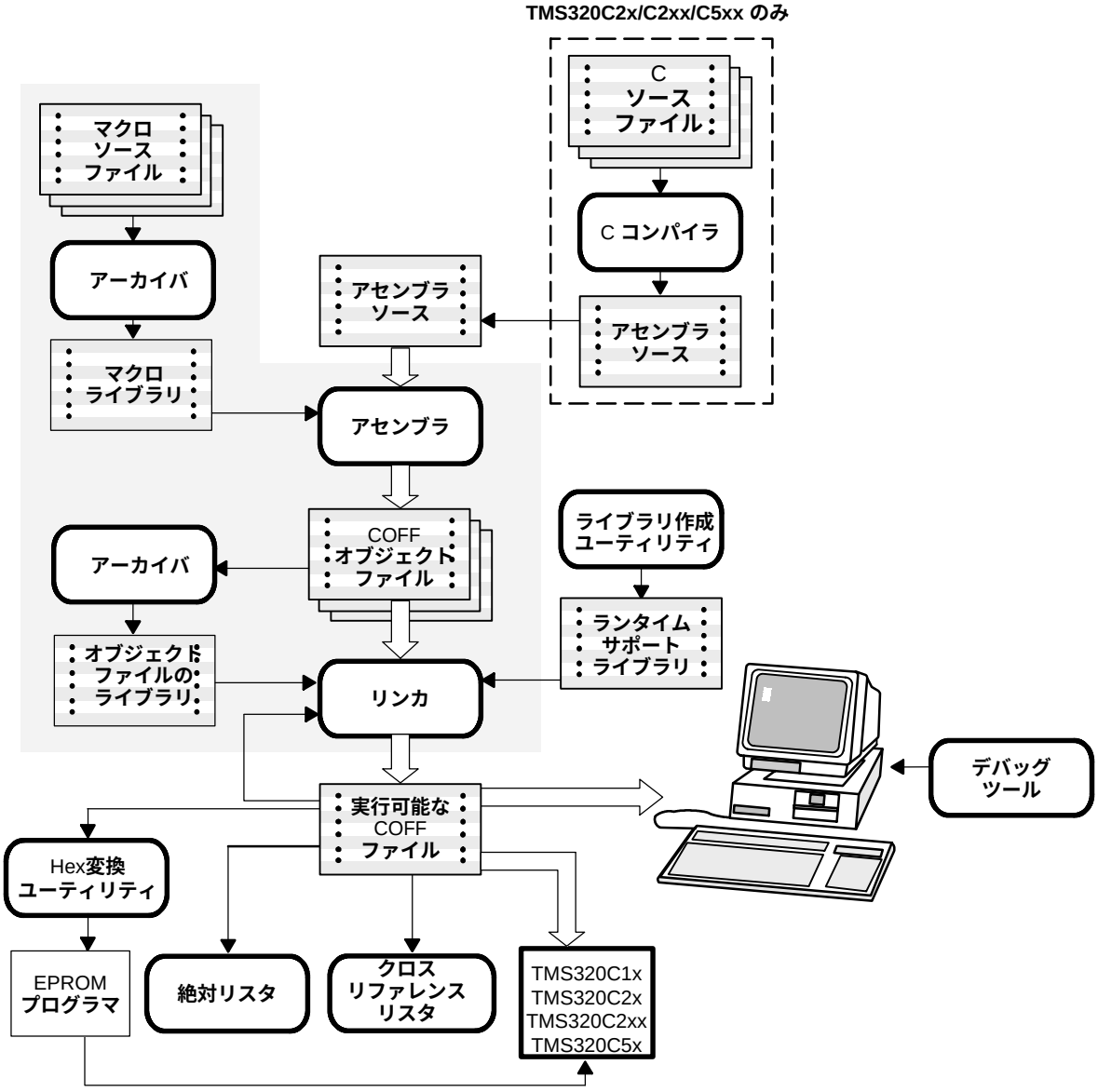
アーカイバの最も便利な使い方の 1 つとして、オブジェクト・モジュールのライブラリを作成することが挙げられます。たとえば、いくつかの算術ルーチンを書いて、アセンブルし、さらにアーカイバを使ってオブジェクト・ファイルを集めて 1 つの論理グループを作成します。次にオブジェクト・ライブラリをリンカ入力として指定します。リンカは、ライブラリ全体を検索し、外部参照を解決するメンバがあれば取り込みます。

アーカイバを使ってマクロ・ライブラリを作成することもできます。それぞれが 1 つのマクロを含むいくつかの独立したソース・ファイルを作成して、次にアーカイバを使ってこれらのマクロを結合して 1 つの機能グループを作ります。.mlib アセンブラ疑似命令を使うと、マクロ・ライブラリの名前をアセンブラに対して指定することができます。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が呼び出すマクロを検索します。第 6 章でマクロおよびマクロ・ライブラリについて詳しく説明しています。

7.2 アーカイバ開発フロー

図 7 - 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。アセンブラもリンカもライブラリを入力として受け入れます。

図 7 - 1. アーカイバ開発フロー



7.3 アーカイバの呼び出し

アーカイバを呼び出すときには、次のように入力します。

dspar [-]*command* [*option*] *libname* [*filename*$_1$ ... *filename*$_n$]

dspar アーカイバを呼び出すためのコマンドです。

libname アーカイブ・ライブラリの名前を指定します。拡張子を指定しないと、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。拡張子を指定しないと、アーカイバはデフォルトの拡張子 *.obj* を付けます。

注： ライブラリ・メンバの命名

1 つのライブラリに同じ名前の複数のメンバを含めることは可能です（望ましくはありませんが）。メンバの削除、置換、または抽出時に、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には、アーカイバはその名前を持った最初のメンバを削除、置換、または抽出します。

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もできます。アーカイバを呼び出す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回の呼び出しで指定できるのは 1 つのコマンドだけです。有効なアーカイバ・コマンドは以下のとおりです。

- **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持つメンバがすでにあっても、そのメンバを置換しないことに注意してください。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と、アーカイバはライブラリ・メンバをカレント・ディレクトリ内にある同じ名前のファイルで置換します。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と、指定されたファイルのみのリストを作成します。ファイル名を指定しないと、アーカイバは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と、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 1 つのメンバを抽出するときに、単にそのメンバをカレント・ディレクトリにコピーするだけです。そのメンバをライブラリから除去しません。

これらのコマンドの 1 つに対して以下のオプションを指定することができます。

- **-e** アーカイバに対して、メンバ名にデフォルトの拡張子 .obj を使用しないように指示します。したがって、拡張子の付かないファイル名を使用できるようになります。
- **-q** (quiet：静的な実行) 見出しと状況メッセージを抑制します。
- **-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状況を表示します。。

7.4 アーカイバの例

以下にアーカイバの使用例を示します。

- **例 1**

  この例では、sine.obj、cos.obj、flt.obj という 3 つのファイルを含んだ function.lib というライブラリが作成されます。

```
dspar -a function sine cos flt
DSP Archiver              Version x.xx
Copyright (c) 1987-1995 Texas Instruments Incorporated
   ==>    new archive 'function.lib'
   ==>    building archive 'function.lib'
```

  例 1 と 2　ではデフォルトの拡張子が使われているので（ライブラリは .lib、メンバは .obj）、拡張子を指定する必要はありません。

- **例 2**

  -t オプションを使うと function.lib の内容の一覧を出力することができます。

```
dspar -t function
DSP Archiver              Version x.xx
Copyright (c) 1987-1995 Texas Instruments Incorporated
      FILE NAME    SIZE   DATE
----------------  -----  ----------------------
      sine.obj     248    Mon Nov 19 01:25:44 1984
      cos.obj      248    Mon Nov 19 01:25:44 1984
      flt.obj      248    Mon Nov 19 01:25:44 1984
```

- **例 3**

  アーカイバにデフォルトの拡張子を使ってほしくない場合は明示的に拡張子を指定することができます。以下に例を示します。

```
dspar -av function.fn sine.asm cos.asm flt.asm
DSP Archiver              Version x.xx
Copyright (c) 1987-1995 Texas Instruments Incorporated
   ==>    add 'sine.asm'
   ==>    add 'cos.asm'
   ==>    add 'flt.asm'
   ==>    building archive 'function.fn'
```

  この例では、sine.asm、cos.asm、flt.asm というファイルを含む function.fn というライブラリが作成されます（-v は補足オプションです）。

❑ **例 4**

ライブラリに新しいメンバを追加する場合は、次のように指定します。

```
dspar -as function tan.obj arctan.obj area.obj
DSP Archiver              Version x.xx
Copyright (c) 1987-1995 Texas Instruments Incorporated
   ==>   symbol defined: 'K2'
   ==>   symbol defined: 'Rossignol'
   ==>   building archive 'function.lib'
```

この例では、ライブラリ名に拡張子を指定していないので、アーカイバは対象となるファイルを function.lib というライブラリに追加します。function.lib ライブラリがない場合、アーカイバはそのライブラリを作成します(-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・シンボルを表示するように指示します)。

❑ **例 5**

ライブラリ内のメンバを変更する場合には、そのメンバを抽出し、編集して、置換します。この例では、macros.lib というライブラリがあって、そこに push.asm、pop.asm、および swap.asm というメンバが含まれているものと仮定しています。

```
dspar -x macros push.asm
```

アーカイバは push.asm のコピーを作成して、カレント・ディレクトリに入れます。ただし、ライブラリから push.asm を除去しません。これで、抽出したファイルを編集することができます。ライブラリにある push.asm のコピーを、編集したコピーと置換するためには、次のように入力します。

```
dspar -r macros push.asm
```

第 8 章

リンカの説明

TMS320C1x/C2x/C2xx/C5x 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 セクションの概念がリンカの動作の基本になっています。第 2 章に、COFF フォーマットについての詳しい説明があります。

リンカは、オブジェクト・ファイルを結合するときに以下の作業を行います。

- ❑ セクションをターゲット・システムの構成メモリに割り振る。
- ❑ 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振る。
- ❑ 入力ファイル間の未定義の外部参照を解決する。

リンカ・コマンド言語は、メモリ構成、出力セクション定義、アドレスのバインディングを制御します。コマンド言語は、式の割り当てと評価をサポートし、2 つの協力な疑似命令 MEMORY と SECTIONS を提供しています。これらの疑似命令を使って以下の操作を行うことができます。

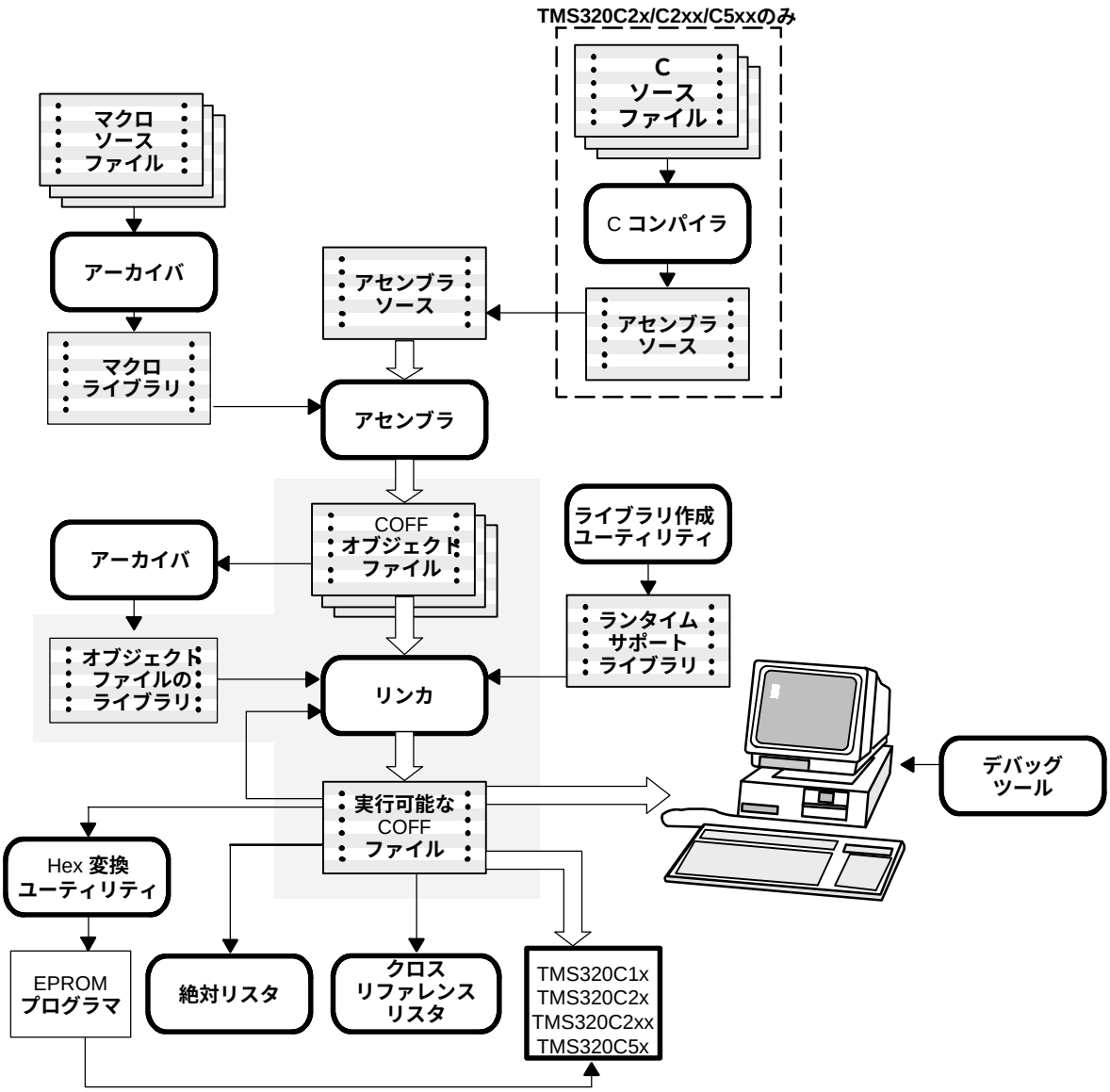
- ❑ ターゲット・システム・メモリに適したメモリ・モデルを定義する。
- ❑ オブジェクト・ファイル・セクションを結合する。
- ❑ セクションをメモリの特定の領域に割り振る。
- ❑ リンク時にグローバル・シンボルを定義、または再定義する。

項目 **ページ**

8.1 リンカ開発フロー

図 8 - 1 は、アセンブリ言語開発プロセスにおいてリンカの果たす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、開発ツールの 1 つにダウンロードすることができ、また TMS320 デバイスを使って実行することができます。

図 8 - 1. リンカ開発フロー



8.2 リンカの呼び出し

リンカを呼び出す一般的な構文は次のとおりです。

dsplnk [*-options*] *filename*$_1$ *... filename*$_n$

dsplnk	リンカを呼び出すためのコマンドです。
options	コマンド行、またはリンカ・コマンド・ファイル内のどこででも指定できます（オプションについては 8.3 節で説明しています）。
filenames	オブジェクト・ファイル、リンカ・コマンド・ファイル、またはアーカイブ・ライブラリが入ります。すべての入力ファイルのデフォルト拡張子としては *.obj* が使われ、それ以外の拡張子を使う場合は明示的に指定しなければなりません。リンカは、入力ファイルがオブジェクト・ファイルか、リンカ・コマンドを含む ASCII ファイルかを見分けることができます。デフォルトの出力ファイル名は *a.out* です。

リンカを呼び出すには、以下の 3 つの方法のいずれかを使います。

- 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、 link.out という名前の出力モジュールを作成します。

```
dsplnk file1.obj file2.obj -o link.out
```

- **dsplnk** コマンドをファイル名とオプションを付けずに入力すると、リンカはそれらに対するプロンプトを表示します。

```
Object files [.obj] :
Command files :
Output files [a.out] :
Options :
```

  - *object files* には、１つ以上のオブジェクト・ファィル名を入力します。デフォルトの拡張子は .obj です。ファイル名とファイル名の間は空白かコンマを使って区切ります。最後の文字がコンマの場合は、リンカは追加のオブジェクト・ファイル名の入力を求めるプロンプトを表示します。
  - *command files* には、１つ以上のコマンド・ファィル名を入力します。
  - *output file* には、リンカ出力モジュール名が入ります。この入力は、他のプロンプトに -o オプションが使われていてもそれを無効にします。 -o オプションが使われていず、ユーザがこのプロンプトに応答しない場合には、リンカはデフォルトのファイル名 a.out をもつオブジェクト・ファイルを作成します。
  - *options* は追加のオプションの入力を求めるプロンプトを表示します。ただし、コマンド・ファイルにオプションを入れることもできます。入力する場合には、コマンド行に入力する場合と同様にハイフンを使って下さい。

- 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。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呼び出すことができます。コマンド・ファイル名を入力ファイルとして指定して下さい。

  ```
  dsplnk linker.cmd
  ```

  コマンド・ファイルを使うときには、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、次のように入力することができます。

  ```
  dsplnk -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。

8.3 リンカ・オプション

リンカ・オプションは、リンク操作を制御するためのものです。リンカ・オプションは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 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。オプションを引数とともに指定するときには、空白で区切ります。表8 - 1 にリンカ・オプションをまとめます。

表8 - 1. リンカ・オプションのまとめ

オプション	説明
-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は、リンカは - a が指定されているように動作します。
-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。
-b	シンボリック・デバッグ情報のマージを抑制します。
-c	TMS320C2x/C2xx/C5x C コンパイラの ROM 自動初期化モデルによって定義されるリンク規則を使用します。
-cr	TMS320C2x/C2xx/C5x C コンパイラの RAM 自動初期化モデルによって定義されるリンク規則を使用します。
-e *global symbol*	出力モジュールのプライマリ・エントリ・ポイントを指定する *global-symbol* を定義します。
-f *fill value*	出力セクション内のホールのデフォルトの埋込み値を設定します。*fill_value* 値は 16ビットの定数です。
-h	すべてのグローバル・シンボルを静的にします。
-heap *size*	（Cの動的メモり割り振り用の）ヒープ・サイズを size ワードに設定し、ヒープ・サイズを指定するグローバル・シンボルを定義します。デフォルトは、1K ワードです。
-i *dir* †	デフォルト・ロケーションの検索場所を検索する前に dir に指定されたディレクトリ内を検索するようにライブラリ検索アルゴリズムを変更します。このオプションは、 - l オプションよりも前に指定しなければなりません。
-l *filename* †	リンカの入力としてアーカイブ・ライブラリの名前を指定します。*filename* にはアーカイブ・ライブラリ名が入ります。
-m *filename* †	ホールを含む入出力セクションのマップまたはリストを作成し、そのリストを *filename* に入れます。
-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。
-q	静的な実行 (見出しの抑制) を要求します。
-r	再配置出力モジュールを作成します。
-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*	C システム・スタック・サイズを size ワードに設定し、スタック・サイズを指定するグローバル・シンボルを定義します。デフォルトは、1K ワードです。
-u *symbol*	未解決の外部シンボル *symbol* を出力モジュールのシンボル・テーブルに入れます。
-v0	バージョン 0 の COFF フォーマットを作成します。
-w	SECTIONS 疑似命令に指定されていない出力セクションが作成されると、警告を作成します。
-x	ライブラリの再読み取りを強制実行します。後方参照を解決します。

† *directory* または *filename* は、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。

8.3.1 再配置機能（ -a オプションおよび -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）をサポートしていて、それを使って絶対出力モジュールまたは再配置出力モジュールを作成することができます。-a オプションも -r オプションも指定しなかった場合、リンカはデフォルトにより、-a オプションが指定された場合と同様の動作をします。

❑ **絶対的な出力モジュールの作成（-a オプション）**

-r オプションを指定せずに、**-a** オプションを指定すると、リンカは実行可能な絶対出力モジュールを作成します。絶対ファイルには、再配置情報は何も含まれていません。実行可能なファイルには以下のものが含まれます。

- リンカによって定義された特別なシンボル（これらのシンボルについては 8-52 ページの 8.13.4 項に説明があります）。
- プログラムへのエントリ・ポイントなどの情報を記述したオプションのヘッダ。
- 未解決の参照は含まれません。

次の例では、file1.obj と file2.obj がリンクされ、a.out という絶対的な出力モジュールが作成されます。

```
dsplnk -a file1.obj file2.obj
```

❑ **再配置可能な出力モジュールの作成（-r オプション）**

-a オプションを指定せずに **-r** オプションを指定すると、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って）再リンクされる場合は、-r オプションを使って再配置エントリを保持します。

-a オプションを指定せずに -r オプションを指定すると、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、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。

```
dsplnk -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、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といいます。 詳しい説明は、8-63 ページの 8.17 節を参照して下さい。)

❑ **実行可能で再配置可能な出力モジュールの作成 (- ar)**

- 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呼び出すと、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・シンボルとオプションのヘッダが含まれます。すべてのシンボル参照は解決されています。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、xr.out という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。

```
dsplnk -ar file1.obj file2.obj -o xr.out
```

これらのオプションは連続して入力する（- ar）ことも、区切って入力する（- a - r）こともできます。

❑ **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）。絶対的なファイルの再リンクは、各入力ファイルに再配置が必要な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）。

8.3.2 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、デフォルトにより、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、C プログラムをデバッグ用にコンパイルした場合です。次の例を見てください。

```
-[ 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中の 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、リンカがこのような重複エントリを保存しないようにしたい場合に使います。- b オプションを使うと、リンカの実行速度が上がり、使用するマシン・メモリも少なくてすみます。

8.3.3 C言語オプション（-c および -cr オプション）

-c および -cr オプションを使用すると、リンカは TMS320C2x/C2xx/C5x C コンパイラが必要とするリンク規則を使用します。

- **-c** オプションはリンカに対して ROM 自動初期化モデルを使うように指示します。
- **-cr** オプションはリンカに対して RAM 自動初期化モデルを使うように指示します。

C コードのリンクについての詳しい説明は、8-59 ページの 8.16 節を参照して下さい。

8.3.4 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・カウンタがプログラムの始まりを指します。

リンカがエントリ・ポイントに割り振ることのできる値は 4 種類あります。以下に示した 4 つの値を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 **-e** *global symbol* です。*global symbol* はエントリ・ポイントを定義し、リンクされる入力ファイル内で外部シンボルとして定義されてなければなりません。
- シンボル _c_int0 の値（使われている場合）。C コンパイラが作成したコードをリンクする場合には、_c_int0 がエントリ・ポイントでなければなりません。
- シンボル _main の値（使われている場合）。
- 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 がエントリ・ポイントです。begin は、file1 か file2 で外部シンボルとして定義されている必要があります。

```
dsplnk -e begin file1.obj file2.obj
```

8.3.5 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できたホールを埋めるか、初期化されていないセクションが初期化されたセクションと結合されるときにその初期化されていないセクションを初期化します。この機能を使えば、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することができます。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 を使います。

この例では、ホールは 16 進値 ABCD で埋められます。

```
dsplnk -f 0ABCDh file1.obj file2.obj
```

8.3.6 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、グローバル・シンボルを静的にします。この機能は、部分リンクを使って関連するオブジェクト・ファイルを自己完結型モジュールにリンクし、さらにそのモジュールを最終的なシステムに再リンクするときに便利です。あるモジュールの中に他のモジュールにあるグローバル・シンボルと同じ名前を持ったグローバル・シンボルがあって、その 2 つを別々のシンボルとして取り扱いたい場合に、- h オプションを使ってモジュールを構築します。モジュールの中のグローバル・シンボルは、通常は他のモジュールから見え、最終リンク時に再定義の問題が生じる可能性がありますが、静的にすると他のモジュールから見えなくなります。

たとえば、b1.obj、b2.obj、b3.obj がそれぞれ関連性があり、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、d3.obj もそれぞれ関連性があり、別のグローバル変数 GLOB を参照しているとします。この場合は、次のコマンドを使って関連するファイルをリンクすることができます。

```
dsplnk -h -r b1.obj b2.obj b3.obj -o bpart.out
dsplnk -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 がグローバル・シンボルを持たないこと、つまり、GLOB が 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 オプションを使うと、bpart.out と dpart.out はそれぞれの再配置エントリを保持することができます。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ってリンクしても安全です。

```
dsplnk bpart.out dpart.out -o system.out
```

8.3.7 ヒープ・サイズの定義（- heap *constant* オプション）

TMS320C2x/C2xx/C5x 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使って、C の動的なメモリ割り振りを行います。ユーザは、リンク時に、- heap オプションを使ってこのメモリ・プールのサイズを設定することができます。サイズは、オプションの直後に定数として指定します。

```
dsplnk -heap 0x8000 /* defines a 2K byte heap (.sysmem section)*/
```

リンカは、入力ファイルに .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、ヒープ・サイズを割り当てます。デフォルトのサイズは 1K ワードです。

C コードのリンクについての詳しい説明は、8-59 ページの 8.16 節を参照して下さい。

8.3.8 ライブラリ検索アルゴリズムの変更 (-i *dir* オプション/C_DIR)

通常、ライブラリをリンカの入力として指定するときには、他の入力ファイル名を指定するときと同じように単にライブラリ名を入力します。リンカは現行のディレクトリ内でそのライブラリを検索します。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。このライブラリにファイル 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示します。

```
dsplnk file1.obj object.lib
```

現行のディレクトリにないライブラリを使う場合には、リンカ・オプション -l（L の小文字）を使います。このオプションの構文は、**-l** *filename*です。 *filename* は、アーカイブ・ライブラリ名を示します。 -l と filename の間に空白を入れても入れなくてもかまいません。

-i リンカ・オプションや環境変数を用いてリンカのディレクトリ検索アルゴリズムを増加させることができます。リンカは次の順序でオブジェクト・ライブラリを検索します。

1) -i リンカ・オプションを使って指定されたディレクトリを検索します。
2) 環境変数 C_DIR を使って指定されたディレクトリを検索します。
3) C_DIR が設定されていない場合は、アセンブラの環境変数 A_DIR を使っ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。

-i リンカ・オプション

-i オプションは、オブジェクト・ライブラリを含む代替ディレクトリを指定します。このオプションの構文は、**-i** *dir* です。*dir* は、オブジェクト・ライブラリを含むディレクトリを指定します。 -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 オプションによって指定されたオブジェクト・ライブラリを検索するときには、 -i によって指定されたディレクトリ内をまず検索します。どの -i オプションも 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 オプションを指定することができます。 -i オプションを使っ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ことが必要です。

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。以下の表は、r.lib と lib2.lib があるディレクトリ、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。使っているオペレーティング・システムの行を選択して下さい。

	パス名	呼び出しコマンド
DOS	`\ld と \ld2`	`dsplnk f1.obj f2.obj -i\ld -i\ld2 -lr.lib -llib2.lib`
UNIX	`/ld と /ld2`	`dsplnk f1.obj f2.obj -i/ld -i/ld2 -lr.lib -llib2.lib`

環境変数（*C_DIR*）

環境変数とは、ユーザが定義する文字列を割り当てるシステム・シンボルです。リンカは **C_DIR** という環境変数を使ってオブジェクト・ライブラリが入った代替ディレクトリを指定します。次のコマンドを使って環境変数を割り当てます。

DOS **の場合**	`set`	`C_DIR=`pathname;another pathname ...
UNIX **の場合**	`setenv`	`C_DIR "`pathname;another pathname ... `"`

pathname は、オブジェクト・ライブラリが入っているディレクトリを示します。コマンド行またはコマンド・ファイルで -l オプションを使って、リンカにどのライブラリを検索するか指示します。

たとえば、r.lib と lib2.lib という 2 つのアーカイブ・ライブラリが別々のディレクトリに入っているとします。以下の表は、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つのライブラリの指定方法を示しています。使っているオペレーティング・システムの行を選択して下さい。

	パス名	呼び出しコマンド
DOS	`\ld と \ld2`	`set C_DIR=\ld;\ld2` `dsplnk f1.obj f2.obj -l r.lib -l lib2.lib`
UNIX	`/ld と /ld2`	`setenv C_DIR="/ld;/ld2"` `dsplnk 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ることに注意してください。

DOS の場合	`set`	`C_DIR=`
UNIX の場合	`unsetenv`	`C_DIR`

アセンブラは **A_DIR** という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・ライブラリを含んだ代替ディレクトリを指定します。C_DIR が設定されていない場合は、リンカは A_DIR を使って指定されたディレクトリ内でオブジェクト・ライブラリを検索します。オブジェクト・ライブラリに関する詳しい説明が 8-19 ページの 8.5 節にあります。

8.3.9 マップ・ファイルの作成（-m *filename* オプション）

-m オプションはリンク・マップ・リストを作成し、それを *filename* に入れます。この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振り
- 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。また以下の 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。このテーブルには 2 つのカラムがあり、左のカラムには名前順にソートされたシンボルが入り、右のカラム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、map.out という名前のマップ・ファイルが作成されます。

```
dsplnk file1.obj file2.obj -m map.out
```

8-63 ページの 8.17 節にマップ・ファイルの例が示してあります。

8.3.10 出力モジュールの命名（-o *filename* オプション）

リンカはエラーが検出されなければ、出力モジュールを作成します。ユーザが出力モジュールのファイル名を指定しないと、リンカはデフォルト名の a.out を使います。出力モジュールを別のファイルに書き込むには、-o オプションを使います。*filename* に、新しい出力モジュール名を指定します。

この例では、file1.obj と file2.obj がリンクされ、run.out という名前の出力モジュールが作成されます。

```
dsplnk -o run.out file1.obj file2.obj
```

8.3.11 静的な実行の指定（-q オプション）

-q オプションをコマンド行またはコマンド・ファイルで最初に指定すると、リンカの見出し表示が抑制されます。このオプションは、バッチ操作の際に使うと便利です。

8.3.12 シンボル情報の除去（-s オプション）

-s オプションを使うと、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することができます。アプリケーションの作成時に出力モジュールをできるだけ小さくしたいときに -s オプションを使うと便利です。

この例では、file1.obj と file2.obj をリンクし、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。

```
dsplnk -o nosym.out -s file1.obj file2.obj
```

-s オプションを使うことによって後でシンボリック・デバッガが使用できなくなったり、ファイルの再リンクができなくなることもあります。

8.3.13 スタック・サイズの定義（-stack *constant* オプション）

TMS320C2x/C2xx/C5x C コンパイラは初期化されないセクション .stack を使ってランタイム・スタックの空間を割り振ります。リンク時に -stack オプションを使って.stack セクションのサイズを設定できます。このオプションの直後に定数でサイズを指定します。

```
dsplnk -stack 0x1000 /* defines a 4K stack (.stack section)*/
```

入力セクションに別のスタック・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、セクションのサイズを割り当てます。デフォルトのスタック・サイズは 1K ワードです。

8.3.14 未解決のシンボルの導入（-u *symbol* オプション）

-u オプションを使うと、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、リンカはライブラリを検索し、そのシンボルを定義しているメンバを組み込みます。リンカがシンボルを定義するメンバをリンクするためには、リンカが -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、ユーザが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、このリンク時に symtab を定義したライブラリ・メンバを含めようとしているとします。以下に示すように -u オプションを使ってリンカに 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指示することができます。

```
dsplnk -u symtab file1.obj file2.obj rts.lib
```

-u オプションを指定しないと、file1.obj と 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、このメンバは含まれません。

8.3.15 バージョン 0 の COFF フォーマットの作成（ - v0 オプション）

デフォルトでは、リンカは、旧バージョンより多数の再配置エントリをサポートする改善バージョンの COFF（バージョン 1）を作成します。しかし、すべての固定小数点デバッガがこの新しい COFF フォーマットをサポートするように更新されているわけではありません。次のようなバージョン（あるいは旧バージョン）のデバッガを使用している場合は、 - v0 オプションを使って古い COFF フォーマットを生成します。

デバッガ・ツール	デバッガのバージョン
TMS320C2x **エミュレータ**	**バージョン** 6.40
TMS320C2x **シミュレータ**	**バージョン** 3.00
TMS320C5x (PG 1.x) **エミュレータ**	**バージョン** 6.50
TMS320C5x (PG 2.x) **エミュレータ**	**バージョン** 7.07
TMS320C5x **シミュレータ**	**バージョン** 1.20

8.3.16 警告スイッチ（ - w オプション）

- w オプションを使うと、SECTIONS 疑似命令に明示的に指定されていない出力セクションが作成された場合に、警告メッセージが作成されます。たとえば次のとおりです。

```
-[ fl.asm ]-

.sect ″xsect″
.word 0

-[ link.cmd ]-

SECTIONS
{
   <no output section specifications reference xsect>
}
```

リンカは、他のオブジェクト・ファイルから入力セクション xsect のデータを集めて、xsect という名前の出力セクションを作成します。次に、リンカはデフォルトの割り振り規則を使って、この出力セクションをメモリに割り振ります。リンク時に - w スイッチを使うと、前の例では以下のメッセージが作成されます。

```
>> warning: creating output section xsect without SECTIONS specification
```

8.3.17 ライブラリの徹底的読み取り（ - x オプション）

通常、リンカは入力ファイル（アーカイブ・ライブラリなど）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メンバがリンクに入れられます。その後で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（これを後方参照といいます）、その参照は解決されません。

- x オプションを使うと、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指示することができます。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読み取りします。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、以下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することができます。

```
dsplnk -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。

```
dsplnk -x -la.lib -lb.lib
```

- x オプションを使うとリンクの速度が低下します。したがって、使うのは本当に必要なときだけにして下さい。

8.4 リンカ・コマンド・ファイル

リンカ・コマンド・ファイルを使うと、リンク情報をファイルに入れることができます。リンカを同じ情報で何度も呼び出すときにこの機能を使うと便利です。また、リンカ・コマンド・ファイルを使うと、MEMORY と SECTIONS 疑似命令を使ってアプリケーションをカスタマイズすることができるので便利です。この 2 つの疑似命令はコマンド・ファイルでのみ使うことができます。コマンド行では使えません。コマンド・ファイルとは以下の要素を 1 つ以上含む ASCII ファイルです。

- ❑ 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・ファイルを指定する入力ファイル名（コマンド・ファイルで他のコマンド・ファイルを入力として呼び出す場合、この文は呼び出しコマンド・ファイル内の最後の文でなければなりません。リンカは呼び出されたコマンド・ファイルからは戻りません）。
- ❑ リンカ・オプション。このオプションは、コマンド行で使うときと同じようにコマンド・ファイルで使うことができます。
- ❑ MEMORY と SECTIONS リンカ疑似命令。MEMORY 疑似命令を使うと、ターゲット・メモリの構成を定義することができます。SECTIONS 疑似命令は、セクションの構築と割り振りの方法を制御します。
- ❑ グローバル・シンボルを定義し、それに値を割り当てる代入文。

コマンド・ファイルでリンカを呼び出すためには、コマンド **dsplnk** を入力し、その後にコマンド・ファイル名を入力します。

dsplnk *command_filename*

リンカは、検出した順に入力ファイルを処理します。リンカは、検出したファイルがオブジェクト・ファイルであれば、そのファイルをリンクします。そうでない場合、リンカは、検出した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・ファイル名は使っているシステムに関係なく大文字、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。

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 にはファイル名とオプションのみが示されています（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呼び出すためには、次のように入力します。

```
dsplnk link.cmd
```

コマンド・ファイルを使うときには、コマンド行に別のパラメータを置くこともできます。

```
dsplnk -r link.cmd c.obj d.obj
```

リンカは検出したコマンド・ファイルを直ちに処理するので、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・ファイルを指定することもできます。たとえば、ファイル名を含んでいる names.lst という名前のファイルと、リンカ疑似命令を含む 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。

```
dsplnk 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。コマンド・ファイルで他のコマンド・ファイルを入力として指定する場合は、その文は呼び出しコマンド・ファイル内の最後の文でなければなりません。

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以外は意味がありません。コマンド・ファイルでのリンカ疑似命令のフォーマットについても同様です。例 8 - 2　は、リンカ疑似命令を含むサンプルのコマンド・ファイルを示したものです。（リンカ疑似命令のフォーマットについては、後で説明します。）

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:  > ROM
  .bss:   > RAM
}
```

以下の名前は、リンカ疑似命令のキーワードとして予約されています。これらの名前をコマンド・ファイルでシンボル名やセクション名として使うことはできません。

リンカ・コマンド・ファイルのキーワード（予約語）

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

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って指定することができます。1 つはアセンブラで使用される 10 進定数、8 進定数、または 16 進定数を指定するために使う形式（3 - 16 ページの 3.7 節を参照）で、もう 1 つは C の構文で整数定数の場合に使う形式です。

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

8.5 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含みます。通常は、関連性のあるモジュールが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ものをすべて含めます。TMS320 アーカイバを使って、アーカイブ・ライブラリを構築し、維持することができます。アーカイバに関する詳しい説明は、第7 章を参照して下さい。

オブジェクト・ライブラリを使うと、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することができます。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。しかしながら、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その関数が組み込まれます。

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。なぜならば、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含めるからです。同じライブラリを必要に応じて何度でも指定することができます。ライブラリは指定されるたびに検索されます。この代わりに、 -x オプションを使うことができます（8-15 ページの 8.3.17 項を参照してください）。ライブラリには、そ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。リンカはそ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、そ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。ただし、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 が clrscr を定義している。

次のように入力すると、

```
dsplnk f1.obj libc.lib f2.obj liba.lib
```

以下の動作が行われます。

- ❑ libc.lib のメンバ 1 は、clrscr に対する両方の参照を満足させます。なぜならば、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 に対する参照を満足させます。
- ❑ liba.lib のメンバ 3 は、fillclr に対する参照を満足させます。

ただし、次のように入力した場合は、

```
dsplnk f1.obj f2.obj liba.lib libc.lib
```

clrscr に対する参照は liba.lib のメンバ 1 によって満足されます。

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 オプションを使ってリンカに強制的にライブラリ・メンバを含めさせることができます。次の例では、リンカのグローバル・シンボル・テーブルに未定義のシンボルrout1が作成されます。以下のように入力したとします。

```
dsplnk -u rout1 libc.lib
```

libc.lib に rout1 を定義しているメンバがある場合は、リンカはそのメンバを含めます。ただし、個々のライブラリ・メンバの割り振りを制御することはできません。メンバの割り振りは、SECTIONS 疑似命令のデフォルトの割り振りアルゴリズムに従って行われます。

オブジェクト・ライブラリを含むディレクトリの指定方法については 8.3.8 項（8-10 ページ）で説明しています。

8.6 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振るかを決定します。この作業を実行するためにリンカは、ターゲット・メモリのモデルを必要とします。MEMORY 疑似命令を使うと、ターゲット・メモリのモデルを指定することができ、ユーザは自分の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することができます。リンカは、出力セクションを割り振るときにモデルを保持し、そのモデルを使ってどのメモリ・ロケーションにオブジェクト・コードを置くべきかを決定します。

TMS320 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 疑似命令を使うと、さまざまな構成を指定することができます。MEMORY 疑似命令を使ってメモリ・モデルを指定した後で、SECTIONS 疑似命令を使って、定義されたメモリに出力セクションを割り振ることができます。

リンカによるセクションの取り扱い方法の詳しい説明については、2 - 10 ページの 2.3 節を参照してください。セクションの再配置の説明については、2 - 18 ページの 2.4 節を参照してください。

8.6.1 デフォルトのメモリ・モデル

アセンブラを使うと、TMS320C1x、TMS320C2x、TMS320C2xx、または TMS320C5x デバイス用のコードをアセンブルすることができます（デバイスを識別するには -v アセンブラ・オプションを使います。3 - 5 ページを参照してください）。アセンブラは、ユーザが選択した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ーに挿入します。リンカは、この情報をオブジェクト・ファイルのヘッダーから読み取ります。MEMORY 疑似命令を指定しない場合、リンカは指定されたデバイスに固有のデフォルトのメモリ・モデルを使います。デフォルトのメモリ・モデルの詳しい説明については、8-44 ページの 8.11.1 項を参照してください。

8.6.2 MEMORY 疑似命令の構文

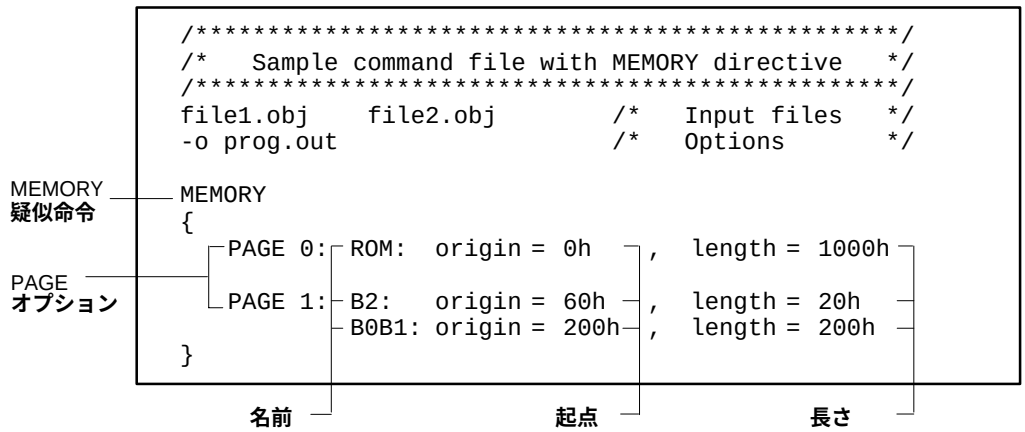
MEMORY 疑似命令は、ターゲット・システムに物理的に存在し、プログラムが使用することのできるメモリの範囲を特定します。それぞれのメモリ範囲には、名前、開始アドレス、および長さがあります。

TMS320 デバイスには、プログラム・メモリ、データ・メモリ、I/O メモリ用にそれぞれ別のメモリ空間があります。これらの空間は同じアドレス範囲を占めています。リンカでは、MEMORY 疑似命令の PAGE オプションを使って、これらのアドレス空間を別個に構成できるようになっています。PAGE 0 はプログラム・メモリを参照し、PAGE 1 はデータ・メモリを参照します。リンカは、この 2 つのページを完全に別個のメモリ空間として取り扱います。

MEMORY 疑似命令を使用する場合は、オブジェクト・コードをロードすることができるすべてのメモリ範囲を必ず特定するようにして下さい。MEMORY 疑似命令によって定義されたメモリは構成メモリとなります。MEMORY 疑似命令によって明示的に定義されなかったメモリは未構成メモリとなります。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、単にその空間を MEMORY 疑似命令文で指定するアドレス範囲に含めないようにします。

MEMORY 疑似命令をコマンド・ファイル内で指定するときには、MEMORY (大文字) に続けてメモリ範囲を中括弧に入れて指定します。例 8 - 3 の MEMORY 疑似命令は、プログラム・メモリのアドレス 0h に 4K ワードの ROM、データ・メモリのアドレス 60h に 32 ワードの RAM を持ち、データ・メモリのアドレス 200h に 512 ワードを持つシステムを定義しています（TMS320C25 をマイクロコンピュータ・モードで実行しているとします）。

例 8 - 3. MEMORY 疑似命令



これで、SECTIONS 疑似命令を使ってリンカに対しセクションをリンクする位置を指示することができます。たとえば、.text と .data セクションを ROM と名付けられたメモリ領域に割り振り、.bss セクションを B2 または B0B1 に割り振ることができます。

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。最大 255 ページまで指定できます。通常は、PAGE 0 がプログラム・メモリを、PAGE 1 がデータ・メモリを指定します。ユーザが PAGE オプションを指定しないと、リンカは PAGE 0 が指定されているように動作します。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 1 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 1～8 文字まで指定できます。使うことのできる文字は、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何の意味もありません。単にメモリ範囲を特定するための記号に過ぎません。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てもかまいません。ただし、同じページではどのメモリ範囲にも一意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もできません。

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、使用する場合は括弧に入れて使用します。属性を使うと、出力セクションの割り当てを特定のメモリ範囲に制限することができます。属性を全く使用しない場合は、どの出力セクションも自由にどのメモリ範囲にでも割り振ることができます。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 つの属性すべてを持ちます。有効な属性には以下のものがあります。

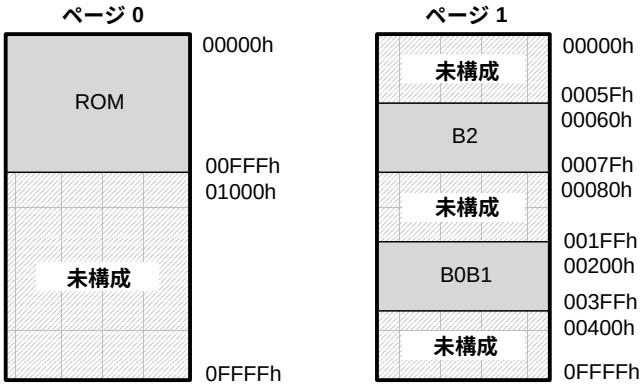
R メモリを読み取れることを指定します。
W メモリに書き込めることを指定します。
X メモリに実行可能なコードを含められることを指定します。
I メモリを初期化できることを指定します。

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。値はバイト単位で指定する 16 ビットの定数です。10進数、8進数、16進数のいづれも使えます。

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。値はバイト単位で指定する 16 ビットの定数です。10進数、8進数、16進数のいづれも使えます。

図 8 - 2 は、例 8 - 3 で定義されたメモリ・マップを示しています。

図 8 - 2. 例 8 - 3 で定義されたメモリ・マップ



8.7 SECTIONS 疑似命令

SECTIONS 疑似命令は、次のような働きをします。

- 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置く位置を指定する（相互に相対的に、そして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。

リンカによるセクションの取り扱い方法の詳しい説明については、2-10 ページの 2.3 節を参照してください。セクションの再配置の説明については、2-18 ページの 2.4 節を参照してください。

8.7.1 デフォルトのセクション構成

ユーザが SECTIONS 疑似命令を指定しないと、リンカはデフォルトのアルゴリズムを使ってセクションを結合し、割り振ります。8.11 節（8-44 ページ）でこのアルゴリズムについて詳しく説明します。

8.7.2 SECTIONS 疑似命令の構文

> **注： 前のバージョンとの互換性**
>
> 前のバージョンのリンカでは、これらの構造の多くについて異なった指定方法が取られていました。現在のバージョンのリンカでは、以前の形式も使用可能です。

SECTIONS 疑似命令をコマンド・ファイル内で指定する場合には、SECTIONS（大文字）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 *name* の後ろには、セクションの内容とその割り振り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。

- **ロード割り振り** 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*

- **実行割り振り** メモリ内のセクションを実行する位置を定義します。

  構文： **run =** *allocation* または
  run > *allocation*

- **入力セクション** 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** 特別なセクションの型を示すフラグを定義します。

  構文： **type = COPY** または
  type = DSECT または
  type = NOLOAD

  セクションの型についての詳しい説明は、8-48 ページの 8.12 節を参照してください。

- **埋め込み値** 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に関する詳しい説明は、8-53 ページの8.14 節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・ファイルにおける 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メモリでどのように割り振られるかを示しています。

例 8 - 4. SECTIONS 疑似命令

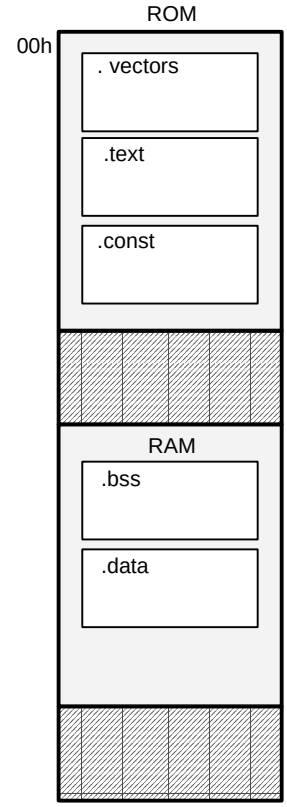
SETIONS 疑似命令

セクションの指定

```
/************************************************/
/* Sample command file with SECTIONS directive */
/************************************************/
file1.obj file2.obj           /* Input files   */
-o prog.out                   /* Options       */

SECTIONS
{
  .text:    load = ROM
  .const:   load = ROM run = 0800h
  .bss:     load = RAM,
  vectors:  load = 0h
      {
         t1.obj(.intvec1)
         t2.obj(.intvec2)
         endvec = .;
      }
  .data:    align = 16
}
```

図 8 - 3 は、例 8 - 4 で SECTIONS 疑似命令によって定義された .vectors、.text、.const、.bss、.data の 5 つの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振り



- 0 にバインド

.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 **セクションとで構成されます。**

- ROM に割り振り

.text セクションは、file1.obj **と** file2.obj **からの** .text **セクションを結合したものです。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。**

- ROM に割り振り

.const セクションは、file1.obj **と** file2.obj **からの**.const **セクションを結合したものです。**

- RAM に割り振り

.bss セクションは、file1.obj **と** file2.obj **からの** .bss **セクションを結合したものです。**

- 16 ワード境界に位置合わせ

.data セクションは、file1.obj **と** file2.obj **からの** .data **セクションを結合したものです。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 **ワード境界に位置合わせします。**

8.7.3　出力セクションのアドレス指定（割り振り）

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ように見えます。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割り振りと呼びます。ロードと実行に別の割り振りを行う方法についての詳しい説明は、8-32 ページの 8.8 節を参照して下さい。

リンカに対してセクションの割り振り方法を指定しないと、リンカはデフォルトのアルゴリズムを使ってセクションの割り振りを行います。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にセクションを置きます。このセクションのデフォルトの割り振りは、SECTIONS 疑似命令内でセクションを定義してその割り振り方法を指示することで無効にできます。

1 つ以上の割り振りパラメータを指定して、割り振りを制御します。各パラメータは、キーワード、(=) や (>) 記号、および括弧で囲んだ値から成ります。ロードと実行に別々の位置が割り振られる場合、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振りに適用され、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振りに適用されます。使用可能な割り振り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割り振ります。

```
.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）または属性を持った範囲に割り振ります。

```
.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 キーワードを使います。

```
.text: align = 0x80
```

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 キーワードを使います。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（8-39 ページの 8.10 節を参照して下さい）。

```
.text: PAGE 0
```

ロード割り振りの場合は（通常はこの割り振りのみ）、単に（>）記号を使い、LOAD キーワードを省略することもできます。

```
.text: > ROM              .text: {...} > ROM
.text: > 0x1000
```

パラメータを 2 つ以上使う場合は、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うこともできます。

```
.text: load = (ROM align(16) PAGE (2))
```

Binding (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することができます。

```
 .text: 0x1000
```

この例では、.text セクションが 1000h の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、16 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のどこにでもバインドすることができますが（十分な空間があれば）、重複させることはできません。セクションを指定したアドレスに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うことはできません。

位置合わせ、または名前付きメモリを使う場合は、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。この場合、リンカはエラー・メッセージを発行します。

Named Memory (名前付きメモリ)

セクションを MEMORY 疑似命令で定義したメモリ範囲に割り振ることができます。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 = 0h,    length = 1000h
   RAM (RWIX):  origin = 3000h, 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 に割り振られています。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 ワード境界に位置合わせされます。

同様に、セクションを特定の属性を持つメモリ領域にリンクすることができます。そのためには、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 疑似命令宣言を使って、次のように指定することができます。

```
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 領域も RAM 領域も X 属性を持っているので、.text 出力セクションをどちらの領域にリンクすることもできます。また、ROM にも RAM にも R 属性と I 属性があるので、.data セクションもどちらの領域にも置くことができます。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振るかは制御することはできません。ただし、リンカは、まず下位のメモリ・アドレスを使い、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、衝突する割り振り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。

Alignment (位置合わせ) と Blocking (ブロック化)

リンカに対して、出力セクションを n ワード境界上のアドレスに置くように指示することができます。ここで、n は 2 のベキ乗です。例を示します。

```
.text: load = align(128)
```

この例では、.text はページ境界上にくるように割り振られます。

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振られます。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 のべき乗でなければなりません。例を示します。

```
bss: load = block(0x80)
```

この例では、.bss は、セクション全体が単一の 128 ワードのページに含まれるか、またはページ境界で始まるように割り振られます。

位置合わせとブロック化は、単独で使用することもメモリ領域と組み合わせて使用することもできますが、位置合わせとブロック化を一緒に使うことはできません。

8.7.4 入力セクションの指定方法

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することができます。リンカは、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。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となります。

例 8 - 5 に示したのは、最も一般的なセクション指定方法です。入力セクションは 1 つも指定されていないことに注意して下さい。

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 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。この指定方法は、すべての出力セクションに使うことができます。

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することもできます。各入力セクションは、ファイル名とセクション名を使って特定されます。

```
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とともに示されていない場合は、ファイルの**すべての**セクションが出力セクションに含まれます。追加の入力セクションの名前が出力セクションの名前と同じで、しかも SECTIONS 疑似命令によって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追加されます。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振られていない .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。

- ❏ 特定の名前を持ったすべての入力セクションを出力セクションに含めたいが、出力セクションの名前が入力セクションの名前と違うとき。
- ❏ リンカに、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振らせたいとき。

上記で説明した２つの場合の例

```
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 が含まれ、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、すべての割り振られていないセクションが含まれます。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含まれている場合は、リンカはその.text 入力セクションをこの .text 出力セクションに含めることはできません。

8.8 セクションのランタイム (実行) アドレスの指定

時にはコードをメモリのある領域にロードして、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低下させ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 を使った方が速くなります。

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、リンカに同じセクションを 2 回割り振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。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については、2 - 20 ページの 2.5 節を参照して下さい。

8.8.1 ロード・アドレスと実行アドレスの指定

ロード・アドレスにより、ローダがセクションの生データを置く位置を決定します。そのセクションに対するすべての参照（ラベルなど）は実行アドレスを示します。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、ユーザが別の実行アドレスを指定するだけでは自動的には行われません。

1 つのセクションに対して 1 回の割り振り（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 回しか割り振られず、ロードも実行も同じアドレスで行われます。2 つの割り振りを指定した場合は、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かのように割り振られます。これは、両方の割り振りがそれぞれメモリ・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 疑似命令を使うと、セクションをオーバーレイさせることができます。8-36 ページの 8.9.1 項を参照して下さい。）

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列挙して下さい。キーワード *load* の後ろにあってキーワード run がみつかるまでの割り振り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。キーワード *run* 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。ロード割り振りと実行割り振りは全く独立しているため、一方の条件（位置合わせなど）は他方の条件に全く影響を与えません。run を先に指定して、後から load を指定することもできます。括弧を使うと読みやすくなります。以下に示す例では、ロード・アドレスと実行アドレスを指定しています。

```
.data: load = ROM, align = 32, run = RAM
```

（位置合わせはロードにのみ適用されます）

```
.data: load = (ROM align 32), run = RAM
```

（前の例と同じです）

```
.data:   run    =  RAM, align 32,
         load   =  align 16
```

(実行用では RAM 内で 32 ワードに位置合わせを行ない、ロード用ではすべての場所で 16 ワードに位置合わせを行ないます。)

8.8.2 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、意味がある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 回だけ割り振ります。ユーザが実行アドレスとロード・アドレスの両方を指定すると、リンカは警告を発行し、ロード・アドレスは無視されます。それ以外の場合は、ユーザが 1 つのアドレスのみを指定すると、リンカはユーザの意向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。以下の例では、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せられ、ロードは無視され、空間は RAM に割り振られます。以下のすべての例では、同じ結果が得られます。.bss セクションは、RAM に割り振られます。

```
.bss: load = RAM
.bss: run = RAM
.bss: > RAM
```

8.8.3 .label 疑似命令によるロード・アドレスの参照

セクション内の通常のシンボルへの参照は、その実行アドレスを示します。ただし、実行時にはロード・アドレスを参照しなければならない場合もあります。特に、セクションをそのロード・アドレスから実行アドレスにコピーするコードの場合は、ロード・アドレスにアクセスできることが必要です。アセンブラの .label 疑似命令は、セクションのロード・アドレスを示す特別な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 疑似命令に関する詳しい説明は、4-48 ページを参照して下さい。

注： .asect 疑似命令はすでに使われていません。

リンカでロード・アドレスと実行アドレスを別に割り振ることができるようになったので、.asect 疑似命令は不要になりました。すべての .asect セクションを (.sect で)、普通のセクションとして書き、リンク時に絶対的な実行アドレスを指定することができます。ただし、.asect 疑似命令はこれまでと全く同じように機能し、現在のバージョンでも使うことができます。また、asect セクション内の .label 疑似命令も以前と同様の機能を果たし、再配置可能なシンボルを定義します。ただし、現在のバージョンでは、すべてのセクションで .label を使ってロード・アドレスを示す再配置可能なシンボルを定義することもできます。

例 8 - 6. セクションの ROM から RAM へのコピー

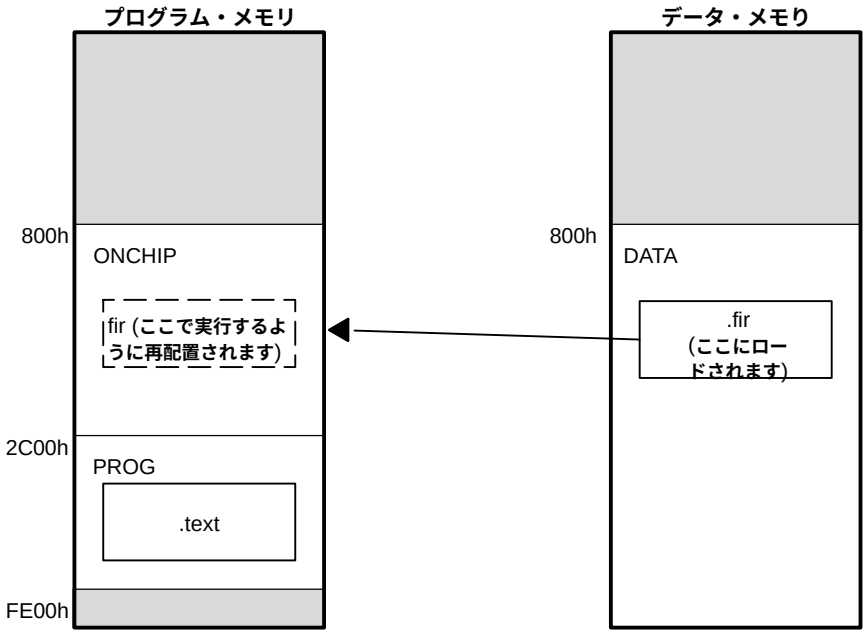
```
;------------------------------------------------------
;  define a section to be copied from ROM to RAM
;------------------------------------------------------
          .sect  ".fir"
          .label fir_src      ; load address of section
fir:                          ; run address of section
          <code here>         ; code for the section

       .label fir_end         ; load address of section end

;------------------------------------------------------
;  copy .fir section from ROM into RAM
;------------------------------------------------------
       .text

       LARK   AR3,fir_src     ; get load address
       LACK   fir             ; get run address
       MAR    *,AR3
       RPTK   fir_end - fir_src - 1
       TBLW   *+              ; block copy

;------------------------------------------------------
;  jump to section, now in RAM
;------------------------------------------------------
       CALL   fir             ; call runtime address
```

リンカ・コマンド・ファイル

```
/*****************************************************/
/* PARTIAL LINKER COMMAND FILE FOR FIR EXAMPLE       */
/*****************************************************/
MEMORY
{
   PAGE 0 :  ONCHIP :  origin = 0800h,  length = 02400h
   PAGE 0 :  PROG   :  origin = 02C00h, length = 0D200h
   PAGE 1 :  DATA   :  origin = 0800h,  length = 0F800h
}

SECTIONS
{
   .text: load = PROG PAGE 0
   .fir:  load = DATA PAGE 1, run = ONCHIP PAGE 0
}
```

図 8 - 4 は、この例の実行時の様子を示したものです。

図 8 - 4.　**例** 8 - 6 **の実行時の様子**



8.9 UNION 文と GROUP 文の使用法

２つの SECTIONS 文 GROUP と UNION を使って、メモリを節約することができます。セクションを共用体としてまとめると、リンカは共用体としてまとめられた複数のセクションに同じ実行アドレスを割り振ることができます。セクションをグループ化すると、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振ることができます。

8.9.1 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振りたい場合もあります。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な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 つの実行アドレスにいくつかのセクションを割り振ることができます。

例 8 - 7.　UNION **文**

```
SECTIONS
{
   .text: load = ROM
   UNION: run = RAM
   {
      .bss1: { file1.obj(.bss) }
      .bss2: { file2.obj(.bss) }
   }
      .bss3: run = RAM { globals.obj(.bss) }
```

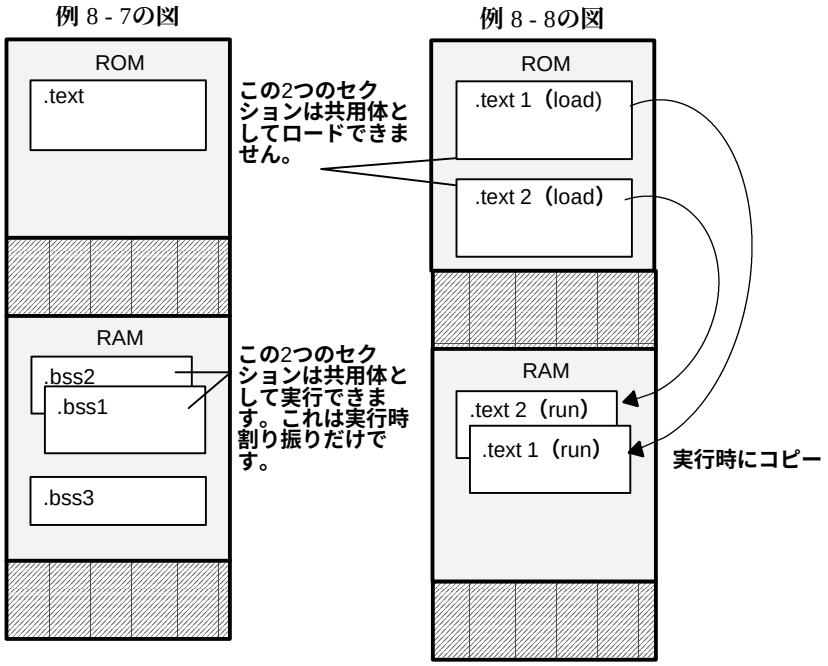
例 8 - 7 では、file1.obj と file2.obj の .bss セクションが RAM 内の同じアドレスに割り振られています。共用体は、その最大の構成要素と同じ大きさの空間をメモリ・マップ内で占めます。共用体の中の構成要素は、それぞれ独立したセクションのままです。これらのセクションは、単に 1 つのまとまりとして一緒に割り振られるだけです。

1 つのセクションを共用体の一部として割り振ることは、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）、そのロード割り振りは必ず別々に指定しなければなりません。例を示します。

例 8 - 8.　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 - 7 と例 8 - 8 で定義されたメモリ割り振り



.text セクションにはデータが入っているので、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。UNION の中の初期化されたセクションのロード割り振りに失敗した場合、リンカは警告を発し、構成メモリ内の可能な場所にロード空間を割り振ります。

初期化されないセクションはロードされず、ロード・アドレスは必要ありません。

UNION 文は実行アドレスの割り振りにのみ使用するので、共用体のロード・アドレスを指定する必要はありません。割り振りに対して、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。指定された割り振りは実行アドレスとみなされます。両方の割り振りが指定された場合、リンカは警告を発し、ロード・アドレスを無視します。

注：　共用体とオーバーレイ・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（8.10 節）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が、実際は、この 2 つの機能は全く異なるものです。UNION を使用すると、同一のメモリ空間内で複数のセクションをオーバーレイできます。これに対しオーバーレイ・ページは、複数のメモリ空間を定義するための機能です。ページ機能を使って 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。

8.9.2 出力セクションのグループ化

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振るための 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。この場合、リンカに .data と term_rec を一緒に割り振らせることができます。

例 8 - 9. セクション・グループの割り振り

```
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、位置合せ、または名前付きメモリを使って、単一の出力セクションの場合と同じ方法で GROUP を割り振ることができます。上の例では、GROUP はアドレス 1000h にバインドされています。これは、メモリ内で、.data が 1000h に割り振られ、term_rec がその後に続いて割り振られるということです。

GROUP オプションを使って、複数の出力セクションを、実行アドレスは 1 つにしロード・アドレスは別々にして割り振ることもできます。GROUP に 1 つだけ実行アドレスを指定した場合は、初期化されたセクションであるグループ内の各メンバに、個別にロード・アドレスを指定する必要があります。

例 8 - 10. 1 つの実行アドレスと別々のロード・アドレスの指定

```
GROUP run = ONCHIP      /* All 3 sections grouped to run ONCHIP   */
{
  .text : load = ROM1   /* but .text loads in ROM1                */
  .data : load = ROM2   /* .data loads in ROM2                    */
  .bss                  /* .bss is uninitialized ==> doesn't load */
}
```

この例では、3 つの出力セクション .text、.data、.bss がグループ化され、メモリ範囲 ONCHIP の 1 つの実行アドレスに割り振られます。.text セクションはメモリ範囲 ROM1 のロード・アドレスに割り振られます。.data セクションはメモリ範囲 ROM2 のロード・アドレスに割り振られます。.bss セクションは初期化されないセクションで、ロードされないので、ロード・アドレスも必要ありません。

8.10 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・メモリによって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、またはこの範囲外にマップさせることができます。言い替えれば、物理メモリの複数のバンクが 1 つのアドレス範囲で互いにオーバーレイします。リンカを使って、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することができます。

リンカはこの機能をオーバーレイ・ページを提供することによりサポートしています。各ページは 1 つのアドレス範囲を表し、MEMORY 疑似命令を使って別々に構成する必要があります。SECTIONS 疑似命令を使って、さまざまなページにマップされるセクションを指定することができます。

8.10.1 MEMORY 疑似命令によるオーバーレイ・ページの定義法

リンカにとっては、各オーバーレイ・ページは、完全な 16 ビットのアドレス指定可能なロケーションを持つ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）アドレスにリンクすることができます。

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 オプションが使われていない場合は、リンカは初期化されたセクションを PAGE 0（プログラム・メモリ）に割り振り、初期化されないセクションを PAGE 1（データ・メモリ）に割り振ります。

たとえば、データ・メモリ空間を 2 つの物理メモリバンク、PAGE1 と PAGE2 から選択できるシステムがあるとします。アドレス範囲は PAGE 1 用が 0A00h ～ FFFFh、PAGE 2 用が 0A00h ～ 2BFF であるとします。一度に選択できるバンクは 1 つだけですが、各バンクを別々のデータで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。

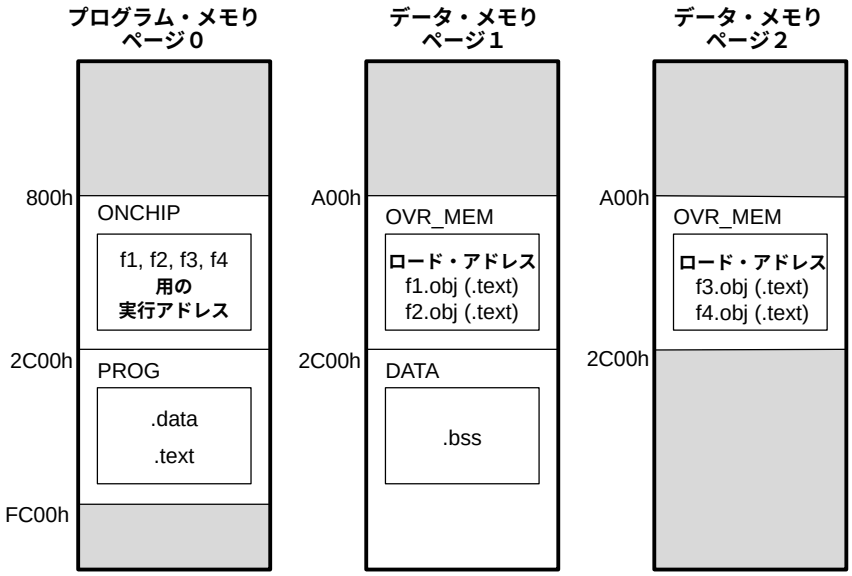
例 8 - 11.　オーバーレイ・ページを指定したMEMORY疑似命令

```
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 - 11 では、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 1 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 - 6 は、例 8 - 11 に示す MEMORY 疑似命令と例 8 - 12 に示す 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 - 6.　例 8 - 11 で定義されているオーバーレイ・ページ



8.10.2 SECTIONS 疑似命令によるオーバーレイ・ページの使用法

MEMORY 疑似命令を例 8 - 11 に示すように使っているとします。さらに、コードには通常のセクションの他に、データ・メモリ空間もロードしてプログラム・メモリ空間のオンチップ RAM で実行する予定の 4 つのコード・モジュールがあるとします。
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 - 12. 図 8 - 6 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
      {
         s1_load = 0A00h;
         s1_start = .;
         f1.obj (.text)
         f2.obj (.text)
         s1_length = . - s1_start;
      }
      S2 :  load = OVR_MEM PAGE 2
      {
         s2_load = 0A00h;
         s2_start = .;
         f3.obj (.text)
         f4.obj (.text)
         s2_length = . - s2_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、f4 です。モジュール f1 と f2 が結合されて出力セクション S1 に、f3 と f4 が結合されて出力セクション S2 になります。S1 と S2 用の PAGE 指定は、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リンカに指示します。その結果、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、ただしメモリ空間は同じではありません。プログラムをロードするときに、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します。

出力セクション S1 と S2 は、オンチップ RAM 内に実行アドレスを持つ共用体に配置されます。アプリケーションは、実行時に、実行する前にこれらのセクションを転送する必要があります。セクション S1 を転送するにはシンボル s1_load と s1_length を使い、セクション S2 を転送するには s2_load と s2_length を使うことができます。特殊記号「.」は、現行の実行アドレスを参照します。現行のロード・アドレスではないので注意してください。

ページ内部では、通常通り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ったりすることができます。例 8 - 12 では、S1 は次のように割り振りできます。

```
S1 : load = 01200h, PAGE = 1  { . . . }
```

これは、S1 を PAGE 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うこともできます。例を示します。

```
S1 : load = (01200h PAGE 1)  { . . . }
```

セクションについてバインディングも名前付きメモリ範囲も指定しなかった場合は、リンカは（1 つのメモリ空間の場合に通常行うのと同様に）可能な場合はいつもセクションをページの中に割り振ります。たとえば、S2 は次のように指定することができます。

```
S2 : PAGE 2  { . . . }
```

OVR_MEM は PAGE 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（指定してもかまいません）。

8.10.3 ページ定義構文

例 8 - 11 と 例 8 - 12 で示したように、以下の構文を使ってオーバーレイ・ページを MEMORY 疑似命令で指定することができます。

```
MEMORY
{
  PAGE 0 :   memory range
             memory range

  PAGE n :   memory range
             memory range
}
```

各ページは、キーワード PAGE とページ番号、それに続くコロンとページに含まれるメモリ範囲のリストによって指定されます。太字で示した部分は、示してある通りに入力する必要があります。メモリ範囲の指定は通常通りの方法で行われます。オーバーレイ・ページは、255 ページまで定義できます。

各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、個々のページのメモリ範囲に同じ名前を付けることができます。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かまいません。ただし、1 つのページの中では、すべてのメモリ範囲は一意の名前を持ち、オーバーラップすることはできません。

PAGE 指定の有効範囲外にリストされているメモリ範囲は、デフォルトでPAGE 0 に設定されます。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
   PAGE 1:   XROM   :  org =     0h   len =  1000h
             XRAM   :  org =  2000h   len = 0E000h
}
```

メモリ範囲の ROM、EPROM、RAM はすべてPAGE 0 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 にあります。PAGE 1 の XROM は PAGE 0 の ROM と重なっており、PAGE 1 の XRAM は PAGE 0 の RAM と重なっていることに注意して下さい。

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ごとの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容易に確認できます。また、出力セクションのリストには PAGE というカラムがあって、各セクションがロードされるメモリ空間を識別できます。

8.11 デフォルトの割り振りアルゴリズム

MEMORY と SECTIONS 疑似命令は、セクションを作成し、結合して、割り振るための柔軟な方法を提供しています。しかし、ユーザが指定しなかったメモリ・ロケーションやセクションも、リンカは取り扱う必要があります。ユーザが提供する仕様の範囲内において、リンカはデフォルトのアルゴリズムを使ってセクションを作成し、割り振ります。8.11.1 項および 8.11.2 項にデフォルトの割り振りアルゴリズムに関する説明があります。

8.11.1 割り振りアルゴリズム

MEMORY 疑似命令と SECTIONS 疑似命令を使用しないと、リンカは以下の定義が指定されているかのように出力セクションを割り振ります。

❑ TMS320C1x **デバイスのデフォルトの割り振り**

```
MEMORY
{
   PAGE 0: PROG: origin = 0x0600  length = 0x0A00
   PAGE 1: DATA: origin = 0x0000  length = 0x0080
}
SECTIONS
{
   .text:     PAGE = 0
   .data:     PAGE = 0
   .cinit:    PAGE = 0    /* For -c and -cr */
   .bss:      PAGE = 1
}
```

❑ TMS320C2x **デバイスのデフォルトの割り振り**

```
MEMORY
{
   PAGE 0: PROG: origin = 0x0020  length = 0xFEE0
   PAGE 1: DATA: origin = 0x0300  length = 0xFD00
}
SECTIONS
{
   .text:     PAGE = 0
   .data:     PAGE = 0
   .cinit:    PAGE = 0    /* For -c and -cr */
   .bss:      PAGE = 1
}
```

❑ TMS320C25 デバイスのデフォルトの割り振り

```
MEMORY
{
   PAGE 0: PROG: origin = 0x1000   length = 0xEF00
   PAGE 1: DATA: origin = 0x0300   length = 0xFD00
}
SECTIONS
{
   .text:     PAGE = 0
   .data:     PAGE = 0
   .cinit:    PAGE = 0    /* For -c and -cr */
   .bss:      PAGE = 1
}
```

❑ TMS320C2xx デバイスのデフォルトの割り振り

```
MEMORY
{
   PAGE 0: PROG: origin = 0x1000   length = 0xEFFF
   PAGE 1: DATA: origin = 0x0300   length = 0xFCFF
}
SECTIONS
{
   .text:     PAGE = 0
   .data:     PAGE = 0
   .cinit:    PAGE = 0    /* For -c and -cr */
   .bss:      PAGE = 1
}
```

❑ TMS320C5x デバイスのデフォルトの割り振り

```
MEMORY
{
   PAGE 0: PROG: origin = 800h   length = 0F800h
   PAGE 1: DATA: origin = 800h   length = 0F800h
}
SECTIONS
{
   .text:     PAGE = 0
   .data:     PAGE = 0
   .cinit:    PAGE = 0    /* For -c and -cr */
   .bss:      PAGE = 1
}
```

すべての .text 入力セクションは連結されて、実行可能な出力ファイル内の .text 出力セクションとなり、すべての .data 入力セクションは組み合わされて .data 出力セクションとなります。.text セクションと .data セクションは、PAGE 0、つまりプログラム・メモリ空間の構成メモリに割り振られます。すべての .bss セクションは組み合わされて .bss 出力セクションになります。.bss セクションは、PAGE 1、つまりデータ・メモリ空間の構成メモリに割り振られます。

入力ファイルに初期化された名前付きセクションがある場合は、リンカはこれらを .data セクションの後に続けてプログラム・メモリに割り振ります。

入力ファイルに初期化されない名前付きセクションがある場合は、リンカはこれらを .bss セクションの後に続けてデータ・メモリに割り振ります。これは、SECTIONS 疑似命令に PAGE を明示的に指定することによって無効にすることができます。

SECTIONS 疑似命令を使うと、リンカはデフォルトの割り振りを全く実行しません。割り振りは、SECTIONS 疑似命令で指定された規則と、8.11.2 項で説明する一般的なアルゴリズムに従って行われます。

8.11.2 出力セクション作成の一般的な規則

出力セクションは以下のいずれかの方法で作成されます。

規則 1 SECTIONS 疑似命令の定義に従って

規則 2 SECTIONS 疑似命令で定義されていない同じ名前を持った入力セクションをまとめて、出力セクションを作成することによって

出力セクションが SECTIONS 疑似命令によって作成される場合（規則 1）は、出力セクションの内容は完全にその定義によって決まります（出力セクションの内容を指定する方法の例については、8-24 ページの 8.7 節を参照して下さい）。

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場合にも作成されます（規則 2）。この場合は、リンカが同じ名前を持ったそのような入力セクションを、その名前の出力セクションにまとめます。たとえば、f1.obj と f2.obj というファイルに Vectors という名前付きセクションがあり、そのセクション用の出力セクションが SECTIONS 疑似命令に定義されていないとします。この場合、リンカは入力ファイルから 2 つの Vectors セクションをまとめて Vectors という名前の 1 つの出力セクションを作成し、それを出力ファイルに含めます。

リンカは、すべての出力セクションの構成を決定した後、それを構成メモリに割り振る作業を行わなければなりません。MEMORY 疑似命令は、メモリのどの部分を構成するかを指定しますが、MEMORY 疑似命令がない場合は、リンカはデフォルトの構成を使用します。

リンカの割り振りアルゴリズムは、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有効に使用することができ、作成するプログラムがメモリに収まる可能性が高くなります。アルゴリズムは次のとおりです。

1) ユーザが特定のバインディング・アドレスを指定した出力セクションは、そのアドレスのメモリに置かれます。

2) 特定の名前付きのメモリ範囲に含まれる出力セクションや、メモリ属性の制約を持つ出力セクションが割り振られます。各出力セクションは、名前付きの領域にある最初に見つかった使用可能な空間に置かれ、その際に位置合わせが必要であればそれも考慮されます。

3) 残りのセクションは、定義された順序で割り振りが行われます。SECTIONS 疑似命令で定義されていないセクションは、出会った順に割り振られます。各出力セクションは、最初に見つかった使用可能なメモリ空間に置かれ、必要な場合は位置合わせが考慮されます。

注: PAGE オプション

PAGE オプションを使って出力セクション用のメモリ空間を指定しなかった場合は、リンカはセクションを PAGE 0 に割り振ります。PAGE 0 に余地がなく別のページには余裕がある場合にも、PAGE 0 への割り振りが行われます。PAGE 0 以外のページを使うには、SECTIONS 疑似命令でページを指定する必要があります。

8.12 特別なセクションの型 (DSECT、COPY、NOLOAD)

出力セクションには、DSECT、COPY、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。これらの型によって、プログラムがリンク、ロードされるときにその取り扱われ方が異なります。セクション定義の後に（括弧で囲んで）型を指定することによってセクションに型を割り当てることができます。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、次のような性質を持ったダミー・セクションを作成します。
  - 出力セクションのメモリ割り振りには含まれません。メモリは使わず、メモリ・マップ・リストにも含まれません。
  - 他の出力セクション、DSECT、未構成メモリとオーバーレイすることができます。
  - ダミー・セクションで定義されるグローバル・シンボルは、普通に再配置されます。これらのグローバル・シンボルは、DSECT が実際にロードされた場合に、持つことになる値を持って出力モジュールのシンボル・テーブルに置かれます。これらのシンボルは、他の入力セクションから参照することができます。
  - DSECT で未定義の外部シンボルが見つかると、指定されたアーカイブ・ライブラリの検索が行われます。
  - セクションの内容、再配置情報、行番号情報は、その出力モジュールには入れられません。

  上記の例では、f1.obj からのセクションは全く割り振られていませんが、シンボルはすべてセクションがアドレス 2000h にリンクされている場合と同じように再配置されます。他のセクションは、sec1 にあるグローバル・シンボルをどれでも参照することができます。

- COPY セクションの機能は DSECT セクションと同様ですが、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が違います。TMS320C2x/C2xx/C5x C コンパイラ用の初期化テーブルを含んだ .cinit セクションには、RAM モデルではこの属性があります。

- NOLOAD セクションは、通常の出力セクションと 1 つだけ違う点があります。それは、セクションの内容、再配置情報、行番号情報が出力モジュールに含まれない点です。リンカはセクション用の空間を割り当て、セクションはメモリ・マップ・リストに含められます。

8.13 リンク時のシンボルの割り当て

リンカ代入文を使うと、外部（グローバル）シンボルを定義し、リンク時にそのシンボルに値を割り当てることができます。この機能を使うと、変数やポインタを割り振りに依存する値に初期化することができます。

8.13.1 代入文の構文

リンカで使われる代入文の構文は、C の代入文と似ています。

構文

symbol **=** *expression*; **シンボルに式の値を代入します。**
symbol **+ =** *expression*; **シンボルの式の値を加えます。**
symbol **- =** *expression*; **シンボルから式の値を引きます。**
symbol *** =** *expression*; **シンボルに式の値を掛けます。**
symbol **/ =** *expression*; **シンボルを式の値で割ります。**

シンボルはプログラムで外部的に定義しなければなりません。そうしないと、リンカは新しいシンボルを定義し、それをシンボル・テーブルに入れます。式は、8.13.3 項で定義する条件に従って作成することが必要です。代入文の最後にはセミコロンを置か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振った後で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は、そのシンボルのアドレスは、実行可能な出力ファイルにあるそのシンボルのアドレスを反映します。

たとえば、プログラムが 2 つの外部シンボル Table1 と Table2 によって識別される2つのテーブルのうちの 1 つからデータを読み取る場合を考えます。プログラムはシンボル cur_tab を現行テーブルのアドレスとして使います。cur_tab は、Table1 か Table2 を指しているはずです。この操作はアセンブリ・コードで行うことができますが、テーブルを変更するためにはプログラムを再度アセンブルする必要があります。その代わりに、リンク時にリンカ代入文を使って 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  */
```

8.13.2 SPC をシンボルに割り当てる方法

ドット (.) で示される特別な記号は、割り振り時のSPCの現行の値を示します。リンカのドット「.」記号はアセンブラのドル「$」記号と働きが似ています。ドット「.」記号は、SECTIONS 疑似命令内の代入文でのみ使用することができます。その理由は、ドット「.」は割り振り中にしか役割を果たさず、SECTIONS 疑似命令が割り振りのプロセスを制御するからです。

ドット「.」記号は、セクションの現行の実行アドレスを参照します。現行のロード・アドレスではないので注意してください。

たとえば、プログラムが .data セクションの始まりのアドレスを知る必要があるとします。.global 疑似命令を使ってプログラム中に Dstart という未定義の外部変数を作成することができます。次に、「.」の値を Dstart に割り当てます。

```
SECTIONS
{
   .text:   {}
   .data:   { Dstart = .; }
   .bss:    {}
}
```

これで、Dstart が .data セクションの最終的にリンクされたアドレスとして定義されました。Dstart の割り当ては、 .data の割り振りの前に行ないます。リンカは、Dstart に対するすべての参照を再配置します。

特別な代入に、「.」記号への代入があります。これにより出力セクションにある 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「.」に代入する値は、「.」が実際に表すアドレスではなく、セクションの先頭に対して相対的です。「.」に対する代入とホールに関する説明は、8.14 節を参照して下さい。

8.13.3 代入式

以下の規則がリンカ式に適用されます。

- 式には、グローバル・シンボル、定数、および表8 - 2 に示す C 言語の演算子を使用することができます。
- すべての数字は、倍長（32 ビットの）整数として取り扱われます。
- 定数は、リンカでも、アセンブラの場合と同じように識別されます。つまり、数字は接尾部（16 進数の場合は H または h、8 進数の場合は Q または q）がなければ 10 進数として認識されます。C 言語の接頭部も認識されます（8 進数は 0、16 進数は 0x）。16 進定数の先頭の文字は、数字でなければなりません。2 進定数は使えません。
- 式の中のシンボルは、そのシンボルのアドレスの値のみを持つことが 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。式に再配置可能なシンボル（およびゼロ、または別の定数、または絶対シンボル）が**1 つでも**含まれている場合は、その式は再配置可能です。その他の場合、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る場合は、そのシンボルは 再配置可能です。絶対式の値が代入されている場合は、そのシンボルは絶対的です。

リンカでは、表8 - 2 に優先順位順に示されている C 言語の演算子がサポートされています。同じグループに属する演算子は、同じ優先度を持ちます。リンカには、表8 - 2 に示されている演算子の他に、出力セクション内の n ワード境界(n は 2 のベキ乗）にシンボルを位置合わせすることのできる align 演算子があります。例を示します。

```
. = align(16);
```

この式は、現行のセクションにある SPC を次の 16 ワード境界に位置合わせます。aline 演算子は、現行の SPC の 1 つの機能なので、「.」と同じ文脈でのみ、つまり、SECTIONS 疑似命令の中でのみ使えます。

表8 - 2. 代入式で使われる演算子

グループ 1（最高優先順位）		グループ 6	
!	論理否定	&	ビット単位 AND
~	ビット単位否定		
-	負		
グループ 2		**グループ 7**	
*	乗算	|	ビット単位 OR
/	除算		
%	剰余		
グループ 3		**グループ 8**	
+	加算	&&	論理 AND
-	減算		
グループ 4		**グループ 9**	
>>	算術右シフト	||	論理 OR
<<	算術左シフト		
グループ 5		**グループ 10（最低優先順位）**	
==	等しい	=	代入
! =	等しくない	+ =	A + = B → A = A + B
>	より大なり	- =	A - = B → A = A - B
<	より小なり	*=	A * = B → A = A * B
< =	以下	/ =	A / = B → A = A / B
> =	以上		

8.13.4 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をどこにリンクすべきかを決めるために使うことのできる 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的で、リンク・マップに示されます。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でもアクセスすることができます。これらのシンボルへの値の割り当ては次のように行われます。

.text .text 出力セクションの最初のアドレスが割り当てられます（実行可能なコードの先頭を示します）。

etext .text 出力セクションに続く最初のアドレスが割り当てられます（実行可能なコードの終りを示します）。

.data .data 出力セクションの最初のアドレスが割り当てられます（初期化されたデータ・テーブルの先頭を示します）。

edata .data 出力セクションに続く最初のアドレスが割り当てられます（初期化されたデータ・テーブルの終りを示します）。

.bss .bss 出力セクションの最初のアドレスが割り当てられます（初期化されないデータの先頭を示します）。

end .bss 出力セクションに続く最初のアドレスが割り当てられます（初期化されないデータの終りを示します）。

C サポートのためだけ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 .stack セクションのサイズが割り当てられます。

__MEMORY_SIZE .sysmem セクションのサイズが割り当てられます。

8.14 ホールの作成と埋め込み

リンカは、出力セクション内に何もリンクされない領域を作成する機能を提供します。このような領域を**ホール**と呼びます。特別な場合として、初期化されないセクションをホールとして取り扱うこともあります。この節では、このようなホールがリンカでどのように取り扱われるか、またホール（および初期化されないセクション）を値で埋める方法を説明します。

8.14.1 初期化されたセクションと初期化されないセクション

出力セクションの内容については、2 つのガイドラインを知っておく必要があります。つまり出力セクションは、以下のいずれかの状態にあります。

- セクション全体の生データが含まれる場合
- 生データが含まれない場合

生データを含むセクションを**初期化された**セクションといいます。これは、そのオブジェクト・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ていることを意味します。セクションがロードされるとき、このイメージがメモリのセクションが指定した開始アドレスにロードされます。.text セクションと .data 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**生データを含みます。.sect または .asect アセンブラ疑似命令を使って定義された名前付きセクションも生データを持ちます。

デフォルトでは、.bss セクションと. usect 疑似命令で定義されたセクションには生データは含まれません（どちらも初期化されません）。どちらのセクションもメモリ・マップ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、通常は RAM に変数用の空間を確保します。オブジェクト・ファイルでは、初期化されないセクションは、通常のセクション・ヘッダーを持ち、その中に定義されたシンボルを持ちます。ただし、そのセクションにはメモリ・イメージは格納されません。

8.14.2 ホールの作成

初期化された出力セクションにホールを作成することができます。1 つの出力セクション内部で入力セクションと入力セクションの間に余分の空間を残すようにリンカに指示すると、ホールができます。このようなホールが作成されるとき、リンカは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。出力セクションと出力セクションの間にも空間を残すことができますが、この空間はホールとは呼びません。出力セクションの間の空間を埋める、つまり初期化する方法はありません。

出力セクションにホールを作成するためには、出力セクション定義の中で特別なリンカ代入文を使う必要があります。代入文で SPC（「.」で示される）を変える場合には、加算をするか、より大きな値を割り当てるか、アドレス境界に位置合わせする方法をとります。代入文の演算子、式、および構文については 8.13 節（8-49 ページ）に説明があります。

次の例では、代入文を使って出力セクションにホールを作成する方法を示しています。

```
SECTIONS
{
   outsect:
   {
      file1.obj(.text)
      . += 100h;       /* Create a hole with size 100h      */
      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 からの .text セクションがリンクされます。
- リンカが 256 ワードのホールを作成します。
- file2.obj からの .text セクションがホールの後ろにリンクされます。
- リンカが、16 ワード境界上に SPC を位置合わせして別のホールを作成します。
- 最後に、file3.obj からの .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的な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アドレス 0 から開始されているものとして取り扱います（ユーザがバインディング・アドレスを指定している場合でも）。上記の例で、. = align (16) という文について考えてみます。この文は、file3.obj の .text が outsect 内の 16 ワード境界から始まるように効率的に位置合わせします。outsect が最終的に位置合わせされないアドレスから始まるように割り振られる場合は、file3.text も位置合わせされません。

「.」記号はセクションの現行の実行アドレスを参照します。現行のロード・アドレスではないので注意してください。

「.」をデクリメントするような式は不正です。たとえば、「.」への代入で - = 演算子を使っても不正です。「.」への代入に使われる最も一般的な演算子は、＋＝ と align です。

出力セクションに含まれる入力セクションがすべて 1 つの種類である場合（たとえば .text）は、その出力セクションの始めか終わりに次の文を使ってホールを作ることができます。

```
.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と結合して、1 つの出力セクションとする方法です。この場合は、リンカは初期化されないセクションをホールとして取り扱い、データを提供します。この方法でホールを作成する例を示します。

```
SECTIONS
{
   outsect:
   {
      file1.obj(.text)
      file1.obj(.bss)    /* This becomes a hole */
   }
}
```

.text セクションに生データがあるので、すべての outsect に生データを含まなければならないことになります（最初のガイドライン）。したがって、初期化されない .bss セクションがホールとなります。

初期化されないセクションは、初期化されたセクションと結合された場合のみホールとなります。複数の初期化されないセクションをリンクした場合には、その結果できる出力セクションも初期化されないセクションとなります。

8.14.3 ホールの埋め込み

初期化された出力セクションにホールがあるときには、リンカは必ずそれを埋めるための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ってホールを埋めます。この値はホール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、埋め込み値を次のように決定します。

1) ホールが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作られた場合は、その初期化されないセクションの埋め込み値を指定することができます。セクション名の後に = 記号と 16 ビット の定数を続けます。

```
SECTIONS
{
   outsect:
   {
      file1.obj(.text)
      file2.obj(.bss) = 0FFh    /* Fill this hole */
   }                            /* with 00FFh     */
}
```

2) セクション定義の後に埋め込み値を指定することによって、同じ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することもできます。

```
SECTIONS
{
   outsect:fill = 0FF00h  /* fills holes with 0FF00h   */
   {
      . += 10h;           /* This creates a hole       */
      file1.obj(.text)
      file1.obj(.bss)     /* This creates another hole */
   }
}
```

3) ユーザがホールの初期化値を指定しないと、リンカは -f リンカ・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、コマンドファイルの link.cmd に次の SECTIONS 疑似命令が含まれているとします。

```
SECTIONS
{
   .text: { .= 100; }      /* Create a 100-word hole */
}
```

ここで -f オプションを使ってリンカを呼び出します。

```
lnk30 -f 0FFFFh link.cmd
```

そうするとホールは 0FFFFh で埋められます。

4) リンカを呼び出すときに -f を使わないと、リンカは 0 でホールを埋めます。

初期化された出力セクションにホールを作成して、それを埋めるときには、必ずそのホールはリンク・マップの中に、リンカがホールを埋めるのに使う値とともに出力されます。

8.14.4 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は、初期化されたセクションと結合されたときだけホールとなります。初期化されない複数のセクションが結合されたときには、その結果できる出力セクションも初期化されないセクションです。

ただし、SECTIONS 疑似命令の中で明示的な埋め込み値を指定するだけで、リンカに初期化されないセクションを初期化させることができます。これにより、セクション全体が生データ（埋め込み値）を持つようになります。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

セクションを埋めると（たとえ 0 を使った場合でも）、その出力ファイルのセクション全体を埋めるための生のデータが生成されるので、大きなセクションまたはホールの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。

8.15 部分 (インクリメンタル) リンク

リンクされている出力ファイルをさらに追加のモジュールとリンクすることができます。これを**部分リンク**またはインクリメンタル・リンクと呼びます。部分リンクを行うと、大きなアプリケーションを部分に分割し、各部分を別々にリンクし、それからすべての部分をリンクして結合して、最終的な実行可能プログラムを作成することができます。

再リンクを行うファイルを作成するときには、次のガイドラインに従ってください。

- ❏ 中間ファイルには必ず再配置情報が必要です。初めてファイルをリンクするときには -r オプションを使います。
- ❏ 中間ファイルには必ずシンボル情報が必要です。デフォルトで、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の場合には -s オプションを使うことはできません。-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。
- ❏ 中間リンク段階は、出力セクションの構成に関係するだけで、割り振りには関係しないようにするべきです。すべての割り振り、バインディング、および MEMORY 疑似命令は、最後のリンクで実行します。
- ❏ 中間ファイルに別のファイルにあるグローバル・シンボルと同じ名前のグローバル・シンボルがあり、それを静的（中間ファイル内でのみ見える）として取り扱いたい場合は、そのファイルをリンクするときに -h オプションを使うことが必要です（8-9 ページの 8.3.6 項を参照して下さい）。
- ❏ C コードをリンクする場合は、最後のリンク段階までは -c または -cr を使わないようにしてください。リンカを呼び出すときにオプション -c または -cr を使うと、リンカはその度にエントリ・ポイントを作成しようとします。

以下の例で、部分リンクの使い方を示します。

手順 1: ファイル file1.com を使ってリンクします。出力ファイル tempout1.out に再配置情報を保持するために、-r オプションを使います。

```
dsplnk -r -o tempout1 file1.com
```

file1.comには以下が含まれています。

```
SECTIONS
{
   ss1:   {
          f1.obj
          f2.obj
           .
           .
           .
          fn.obj
          }
}
```

手順 2: ファイル file2.com を使ってリンクします。出力ファイル tempout2.out に再配置情報を保持するために、 -r オプションを使います。

```
dsplnk -r -o tempout2 file2.com
```

file2.comには以下が含まれています。

```
SECTIONS
{
   ss2:  {
         g1.obj
         g2.obj
          .
          .
          .
         gn.obj
         }
}
```

手順 3: tempout1.out と tempout2.out をリンクします。

```
dsplnk -m final.map -o final.out tempout1.out tempout2.out
```

8.16 C コードのリンク

TMS320C2x/C2xx/C5x 最適化Cコンパイラを使うと、アセンブルし、リンクすることができるアセンブリ言語ソース・コードを作成することができます。たとえば、prog1 や prog2 などのモジュールから成る C プログラムをアセンブルしてからリンクして prog.out という名前の実行可能なファイルを作成することができます。

```
dsplnk -c -o prog.out prog1.obj prog2.obj ... rtsxx.lib
```

-c オプションは、リンカに対して、C 環境で定義された特別な規則を使用するように指示します。アーカイブ・ライブラリ rtsxx.lib（xx は 25、2xx、または 50）には、ランタイムサポート関数が含まれています。

C に関する詳しい説明、特にランタイム (実行時) 環境とランタイムサポート関数に関する説明は、TMS320C2x/C2xx/C5x オプティマイジング C コンパイラ・ユーザーズ・マニュアルを参照して下さい。

8.16.1 実行時初期化

すべての C プログラムは boot.obj というオブジェクト・モジュールにリンクする必要があります。プログラムは、実行を開始すると、boot.obj を最初に実行します。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含まれています。このモジュールは次の作業を実行します。

- ❑ システム・スタックのセットアップ。
- ❑ 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 rtsxx.lib には boot.obj が含まれています。以下のことが可能です。

- ❑ アーカイバを使ってライブラリから boot.obj を抽出し、それを直接リンクする。
- ❑ rtsxx.lib を入力ファイルとして組み込む（-c オプションまたは -cr オプションを使うと、リンカは自動的に boot.obj を抽出します）。

8.16.2 オブジェクト・ライブラリとランタイム・サポート

TMS320C2x/C2xx/C5x オプティマイジング C コンパイラ・ユーザーズ・マニュアルに rtsxx.lib に含まれている追加のランタイム・サポート関数についての説明があります。プログラムでこれらの関数のいずれかを使う場合は、rtsxx.libをユーザのオブジェクト・ファイルにリンクする必要があります。

また、ユーザ独自のオブジェクト・ライブラリを作成して、それをリンクすることができます。リンカは､未定義の参照を解決するライブラリ・メンバのみをリンクします。

8.16.3 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 には .sysmem と .stack という名の 2 つの初期化されないセクションがあり、それぞれを malloc() とランタイム・スタックがメモリ・プールとして使います。この 2 つのセクションのサイズは、 - heap オプション（8 - 9 ページの 8.3.7 項を参照）または - stack オプション（8 - 13ページの 8.3.13項を参照）を使用し、その直後にセクションのサイズを 定数として指定することにより設定できます。オプションを使わない場合の両方のセクションのデフォルトのサイズは、1K ワードです。

8.16.4 自動初期化（ROM モデルと RAM モデル）

C コンパイラは、グローバル変数を自動初期化するために使用するデータのテーブルを作成します。これらのテーブルは、.cinit と呼ばれる特別なセクションに入っています。初期化テーブルは、2 つのうちのいずれかの方法で自動初期化に使用することができます。

- **RAM モデル**（ - cr オプション）

  変数はロード時に初期化されます。これにより、ブート時間を短縮してパフォーマンスを向上させ、また初期化テーブルが使うメモリを節約することができます。自動初期化の RAM モデルを使って効果をあげるには、スマート・ローダ（つまり、変数を初期化できるローダ）を使わなければなりません。

  - cr オプションを使用すると、リンカは .cinit セクションに特別な属性で印を付けます。この属性は、リンカに対して .cnit セクションをメモリにロードしないように指示します。また、リンカは cinit シンボルを - 1 に設定します。これで、C ブート・ルーチンに初期化テーブルがメモリにないことを知らせます。したがって、ブート時には実行時初期化は行われません。

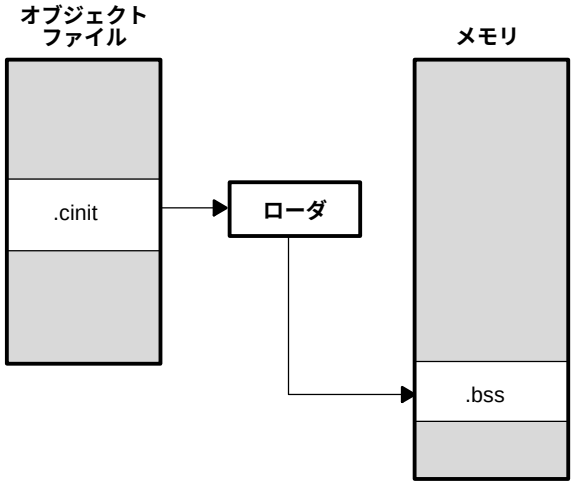
  プログラムがロードされると、ローダは次のことができるはずです。

  - オブジェクト・ファイルに .cinit セクションがあることを検知する。
  - .cinit セクションをコピーしないように指示する属性があることを検知する。
  - 初期化テーブルのフォーマットを理解する（この点については、TMS320C2x/C2xx/C5x オプティマイジング C コンパイラ・ユーザーズ・マニュアルに説明があります）。

  以後はローダはオブジェクト・ファイルから初期化テーブルを直接に使って .bss にある変数を初期化します。

  図 8 - 7 は、RAM 自動初期化モデルを示したものです。

図 8 - 7. **自動初期化の** RAM **モデル**

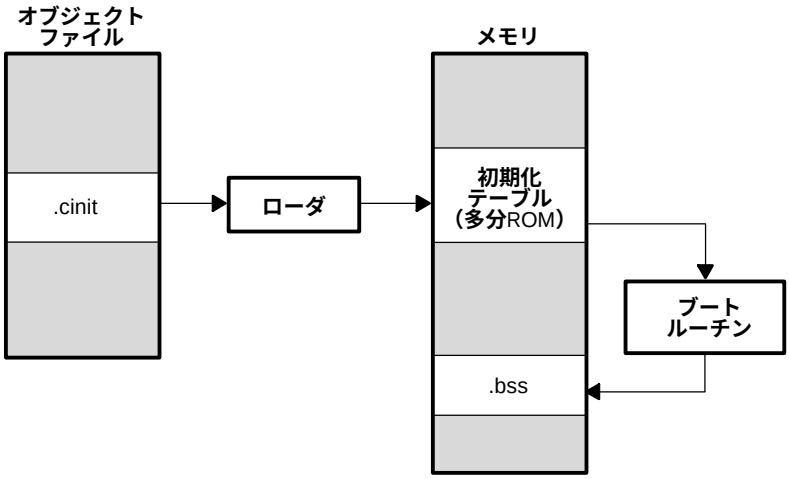


❑ **ROM モデル**（- c **オプション**）

変数は、実行時に初期化されます。.cinit **セクションは、他のすべてのセクションとともにメモリにロードされます。リンカは、メモリにあるテーブルの始まりを指す** cinit **と呼ばれる特別のシンボルを定義します。プログラムを実行すると、**C **ブート・ルーチンは、テーブルのデータを** .bss **セクションにある変数にコピーします。これにより、初期化データが** ROM **に格納され、それからプログラムが起動される度に** RAM **にコピーされることが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たものです。**

図 8 - 8. **自動初期化の** ROM **モデル**



8.16.5 - c および - cr リンカ・オプション

以下に - c オプションまたは - cr オプションを使ってリンカを呼び出すと何が起こるかをまとめます。

- シンボル _c_int0 はプログラム・エントリ・ポイントとして定義されます。_c_int0 は、boot.obj にある C ブート・ルーチンの始まりです。_c_int0 を参照すると、boot.obj がランタイム・サポート・ライブラリ rtsxx.lib から自動的にリンクされます。
- .cinit 出力セクションには、ブート・ルーチン（ROM モデル）またはローダ（RAM モデル）がいつ初期化テーブルの読み込みを停止するかを知ることができるように、終端レコードが入っています。
- ROM モデルでは（ - c オプション）、リンカはシンボル cinit を .cinit セクションの開始アドレスとして定義します。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います。
- RAM モデルの場合（ - cr オプション）
  - 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にないことを示しており、初期化は実行時には行われません。
  - STYP_COPY フラグ (010h) が .cinit セクションのヘッダーに設定されます。STYP_COPY は、特別な属性で、ローダに対して直接自動初期化を実行し、.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。リンカは、メモリ内に .cinit セクション用の空間を割り振りません。

8.17 リンカの例

ここで示すのは TMS320C2x/C2xx/C5x C コンパイラについての例です。この例では、demo.obj、fft.obj、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 という名前の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
0000 ～ 0FFF	オンチップ ROM
FF00 ～ FFFF	オンチップ RAM ブロック B0

データ・メモリ

アドレス範囲	内容
0000 ～ 0005	I/O レジスタ
0060 ～ 007F	オンチップ RAM ブロック B2
0300 ～ 03FF	オンチップ RAM ブロック B1

出力セクションは、次のように入力セクションから構成されます。

- ファイル tables.obj にあるセクション int_vecs からの一連の割り込みベクタは、プログラム ROM のアドレス 0 でリンクされなければなりません。
- demo.obj と fft.obj の .text セクションに含まれる実行可能な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。
- tables.obj と fft.obj というファイルの .data セクションにある 2 つの係数テーブルは、プログラム・メモリの RAM のブロック B0 にリンクされなければならない。ブロック B0 の残りの部分は、値 07A1Ch に初期化されなければなりません。
- 変数を含む fft.obj の .bss セクションは、データ RAM のブロック B1 にリンクされなければならない。
- バッファと変数を含む demo.obj の .bss セクションは、データ RAM のブロック B1 にリンクされなければならない。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。

例 8 - 13 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 - 14 は、マップ・ファイルを示します。

例 8 - 13. リンカ・コマンド・ファイル demo.cmd

```
/************************************************************************/
/****                     Specify Linker Options                     ****/
/************************************************************************/

-e SETUP                 /*   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 Configuration                ****/
/************************************************************************/

MEMORY
{
   PAGE 0: ROM:        origin = 00000h,      length = 01000h
           RAM_B0:     origin = 0FF00h,      length = 0100h
   PAGE 1: IO:         origin = 00000h,      length = 06h
           RAM_B2:     origin = 00060h,      length = 020h
           RAM_B1:     origin = 00300h,      length = 0100h
           RAM:        origin = 00400h,      length = 0FC00h
}
/************************************************************************/
/****                  Specify the Output Sections                   ****/
/************************************************************************/
SECTIONS
{
   .text:load = ROM, page = 0        /* Link .text sections into ROM */

    int_vecs: load = 0, page = 0     /* Link interrupts at 0         */

    .data: fill = 07A1Ch, load = RAM_B0, page = 0
                                     /* Build .data section          */
   {
        tables.obj(.data)       /* .data input section              */
        fft.obj(.data)          /* .data input section              */
        . = 100h;               /* Create a hole to the end of block */
   }                            /* Fill with 7A1Ch and link into B0  */

   fftvars: load = RAM_B2, page = 1  /* Create new fftvars section   */
   {
        fft.obj(.bss)
   }                            /* Link into B2                     */

   .bss: load = RAM_B1, page = 1, fill = 0FFFFh
                                /* Remaining .bss; fill and link     */

}
/************************************************************************/
/****                     End of Command File                        ****/
/************************************************************************/
```

次のコマンドを使ってリンカを呼び出します。

```
dsplnk demo.cmd
```

これにより例 8 - 14 に示されたマップ・ファイルと、TMS320 で実行することのできる demo.out という名の出力ファイルが作成されます。

例 8 - 14.　**出力マップ・ファイル** demo.map

```
OUTPUT FILE NAME:   <demo.out>
ENTRY POINT SYMBOL: ″SETUP″  address:  00000020

MEMORY CONFIGURATION
           name               origin         length         attributes
           -------            --------       --------       ----------
PAGE 0:    ROM                00000000       00001000       RWIX
            RAM_B0             0000FF00       000000100     RWIX

PAGE 1:    IO                 00000000       000000006      RWIX
            RAM_B2             00000060       000000020     RWIX
            RAM_B1             00000300       000000100     RWIX
            RAM                00000400       00000FC00     RWIX

SECTION ALLOCATION MAP

output                                          attributes/
section    page   origin         length         input sections
-------    ----   ----------     -----------    ----------------
int_vecs   0      00000000        00000020
                    00000000        00000020    demo.obj (int_vecs)

.text       0     00000020       000001B2
                    00000020       0000014E     demo.obj (.text)
                    0000016E       00000064     fft.obj (.text)

fftvars     1     00000060       0000001A       UNINITIALIZED
                    00000060      0000001A      fft.obj (.bss)

.bss        1     00000300       0000009A
                    00000300      0000009A      demo.obj (.bss)[fill = ffff]
.data       0       0000FF00      00000100
                    0000FF00      000000A5      tables.obj (.data)
                    0000FFA5      00000014      fft.obj (.data)
                    0000FFB9      00000047      --HOLE-- [fill = 7A1C]

GLOBAL SYMBOLS
address        name                            address    name
---------      ----                            ---------  ----
00000020       SETUP                              00000020   SETUP
00010000       edata                              0000002A   start
0000039A       end                                0000006A   fft
000001D2       etext                           00000100   sub
00000300       extvar                             00000146   list
0000006A       fft                                00000150   p1asm
00000146       list                               0000015A   p2asm
00000164       main                            00000164   main
00000150       p1asm                              000001D2   etext
0000015A       p2asm                              00000300   extvar
0000002A       start                              0000039A   end
00000100       sub                                00010000   edata

[12 symbols]
```

第 9 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して、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。これを手動で行おうとすると、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・ユーティリティはこのような操作を自動的に行います。

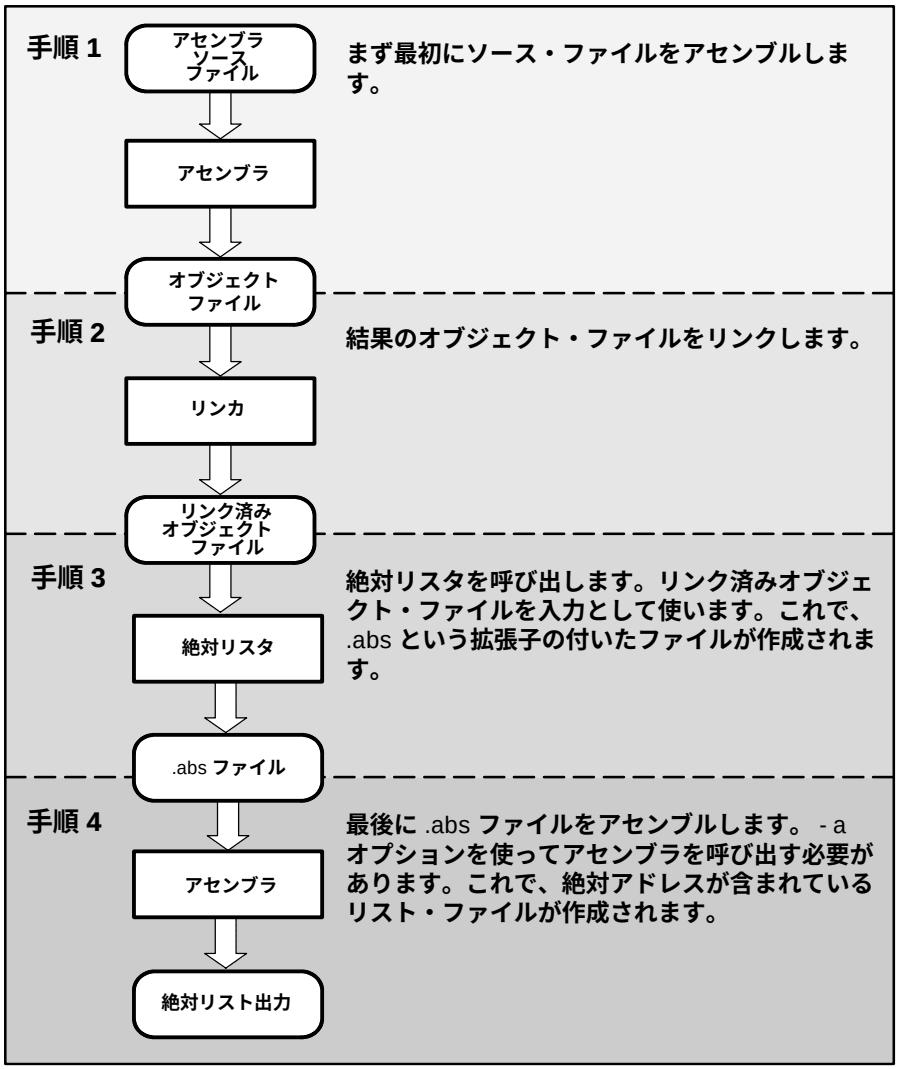
この章では以下の内容について説明します。

項目 ページ

9.1 絶対リスト出力の作成

図 9 - 1 に、絶対リスト出力を作成するために必要な手順を示します。

図 9 - 1. 絶対リスタの開発フロー



9.2 絶対リスタの呼び出し方法

絶対リスタを呼び出すには、以下のように入力します。

dspabs [*-options*] *input file*

dspabs 絶対リスタを呼び出すコマンドです。

options 使いたい絶対リスタ・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することができます。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・オプションは以下のとおりです。

-e これを使うと、アセンブリ・ファイル、Cソース・ファイル、およびCヘッダー・ファイルのファイル名拡張子についての dspabs の命名規則を変更できます。3 つのオプションを以下に示します。

- □ - 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ヘッダー・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 （静止）見出しとすべての進捗情報を抑制します。

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 を持つものと解釈します。絶対リスタを呼び出すときに入力ファイル名を指定しなかった場合は、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いて1つずつの出力ファイルを作成します。出力ファイルには、入力ファイルと同じ名前に拡張子 .abs を付けた名前が付きます。ただし、ヘッダー・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。

絶対リスト出力を作成するには、 - a アセンブラ・オプションを使って上記の出力ファイルをアセンブルします。

dspa -a *filename*.abs

- e オプションは、コマンド行のファイル名の解釈と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常に、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・オプションを使ってコンパイルした C ファイルから作成したリンク済みオブジェクト・ファイルの場合に使うと便利です。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 ヘッダー・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・オプションを使ってコンパイルするとします。すると、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 ファイルを作成することができます。

```
dspabs -ea s -c csr -eh hsr hello.out
```

hello.hsr（ヘッダー・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 でなくアセンブリ・ファイル hello.s が組み込まれます。

9.3 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

```
V_TBL    .sect     ".vectors"
RESET    B         INIT
         .copy     "globals.def"
         .data
int_a    .int      10
int_b    .int      20
         .bss      aaa,1
         .bss      bbb,1
         .bss      ccc,1
         .text
INIT     LDP       #aaa
         BLPD      #int_a,aaa
         BLPD      #int_b,bbb
         LACL      aaa
         ADD       bbb
         SACL      ccc
```

module2.asm

```
         .copy     "globals.def"
         .bss      ddd,1
         .text
         LDP       #aaa
         LACL      bbb
         SUB       aaa
         SACL      ddd
```

globals.def

```
         .global   aaa
         .global   bbb
```

次の手順では、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。

手順 1: まず、module1.asm と module2.asm をアセンブルします。

```
dspa -v50 module1
dspa -v50 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。

手順 2: 次に、calc.lnk という名前の以下のようなリンカ・コマンド・ファイルを使って、module1.obj と module2.obj をリンクします。

```
/************************************************/
/* File calc.lnk  -- COFF linker control file    */
/* for linking TMS320 modules                    */
/************************************************/
-o CALC.OUT        /* executable output file     */
-m CALC.MAP        /* output map file            */

/* input files                                   */
MODULE1.OBJ
MODULE2.OBJ

/* define memory map                             */
MEMORY
{
    PAGE 0: TRAP:      org=0h      len=040h
            ROM1:      org=040h    len=0100h
            ROM2:      org=0ff00h  len=0100h
    PAGE 1: RAM:       org=0100h   len=0100h
}

/* define the output sections                    */
SECTIONS
{
    .vectors:      >TRAP
    .text:         >ROM1
    .data:         >ROM2
    .bss:          >RAM
}
```

そして、リンカを呼び出します。

dsplnk calc.lnk

これで、calc.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います。

手順 3: ここで、絶対リスタを呼び出します。

```
dspabs calc.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
aaa       .setsym     00100h
bbb       .setsym     00101h
.text     .setsym     00040h
etext     .setsym     0004ch
.data     .setsym     0ff00h
edata     .setsym     0ff02h
.bss      .setsym     00100h
end       .setsym     00104h
          .setsect    ".text",00040h
          .setsect    ".data",0ff00h
          .setsect    ".bss",00100h
          .setsect    ".vectors",00000h
          .list
          .text
          .copy       "module1.asm"
```

module2.abs:

```
          .nolist
aaa       .setsym     00100h
bbb       .setsym     00101h
.text     .setsym     00040h
etext     .setsym     0004ch
.data     .setsym     0ff00h
edata     .setsym     0ff02h
.bss      .setsym     00100h
end       .setsym     00104h
          .setsect    ".text",00048h
          .setsect    ".data",0ff02h
          .setsect    ".bss",00103h
          .list
          .text
          .copy       "module2.asm"
```

この 2 つのファイルには、手順 4 でアセンブラを呼び出したときにアセンブラに必要となる以下のような情報が入っています。

- .setsym 疑似命令。これは、値をグローバル・シンボルと等価にします。どちらのファイルにも、シンボル *aaa* と *bbb* 用のグローバルな等価値が含まれています。シンボル *aaa* と *bbb* 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ってもあまり役にたちません。この 2 つの疑似命令が役に立つのは、絶対リスト出力を作成する場合だけです。

手順 4: 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（アセンブラを呼び出すときは -a オプションを使う必要があることを忘れないで下さい）。

```
dspa  -a  -v50  module1.abs
dspa  -a  -v50  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。

作成される絶対リスト・ファイルは以下のようになります。

module1.lst:

```
TMS320C1x/C2x/C2xx/C5x COFF Assembler  Version 6.60    Wed Sep 25 00:00:00 1996
Copyright (c) 1987-1995  Texas Instruments Incorporated

module1.abs                                                              PAGE    1

      15 0040                .text
      16                     .copy     ”module1.asm”
 A     1 0000       V_TBL    .sect     ”.vectors”
 A     2 0000 7980  RESET    B         INIT
         0001 0040’
 A     3                     .copy     ”globals.def”
 B     1                     .global   aaa
 B     2                     .global   bbb
 A     4 ff00                .data
 A     5 ff00 000a  int_a    .int      10
 A     6 ff01 0014  int_b    .int      20
 A     7 0100                .bss      aaa,1
 A     8 0101                .bss      bbb,1
 A     9 0102                .bss      ccc,1
 A    10 0040                .text
 A    11 0040 bc02- INIT     LDP       #aaa
 A    12 0041 a500-          BLPD      #int_a,aaa
         0042 ff00”
 A    13 0043 a501-          BLPD      #int_b,bbb
         0044 ff01”
 A    14 0045 6900-          LACL      aaa
 A    15 0046 2001-          ADD       bbb
 A    16 0047 9002-          SACL      ccc

 No Errors,  No Warnings
```

module2.lst:

```
TMS320C1x/C2x/C2xx/C5x COFF Assembler  Version 6.60   Wed Sep 25 00:00:00 1996
Copyright (c) 1987-1995  Texas Instruments Incorporated

module2.abs                                                          PAGE    1

      14 0048               .text
      15                    .copy     ”module2.asm”
 A     1                    .copy     ”globals.def”
 B     1                    .global   aaa
 B     2                    .global   bbb
 A     2 0103               .bss      ddd,1
 A     3 0048               .text
 A     4 0048 bc02!         LDP       #aaa
 A     5 0049 6901!         LACL      bbb
 A     6 004a 3000!         SUB       aaa
 A     7 004b 9003-         SACL      ddd

 No Errors,  No Warnings
```

第 10 章

クロス・リファレンス・リスタ

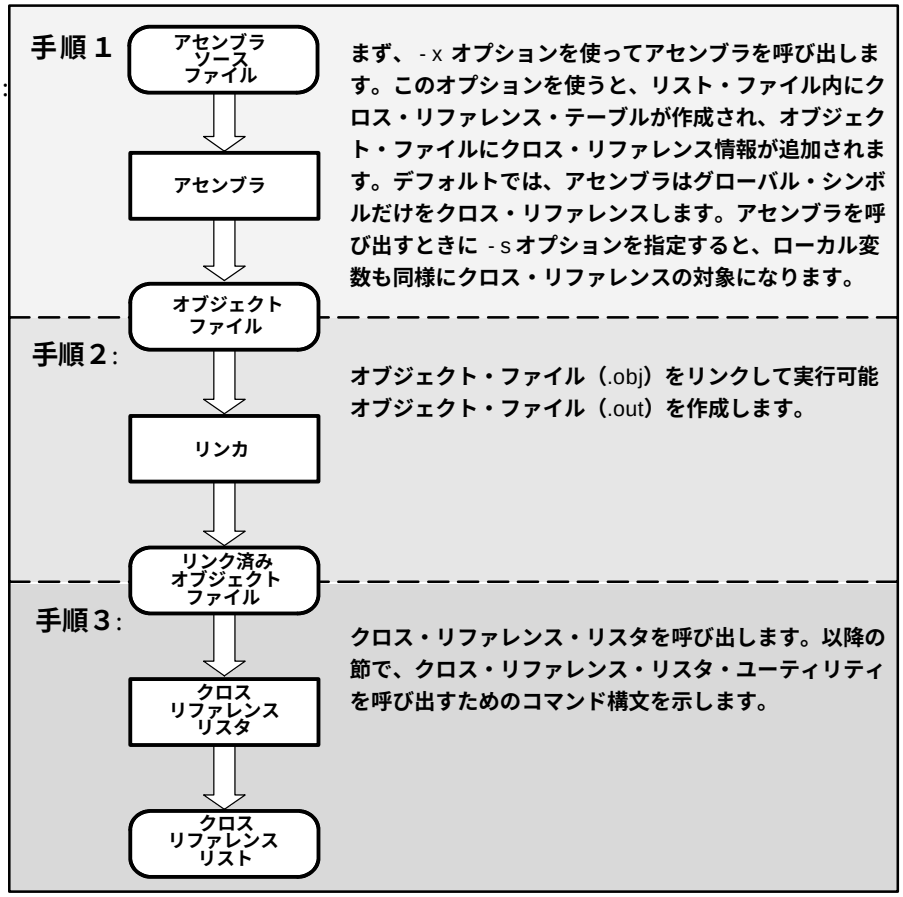
クロス・リファレンス・リスタはデバッグ・ツールです。このユーティリティは、リンク済みオブジェクト・ファイルを入力として受け入れ、クロス・リファレンス・リストを出力として作成します。このリストには、シンボル、シンボルの定義、リンクされたソース・ファイル内でのシンボルの参照が示されています。

この章では以下の内容について説明します。

項目 ページ

10.1 クロス・リファレンス・リストの作成

図 10 - 1. **クロス・リファレンス・リスト開発フロー**



10.2 クロス・リファレンス・リスタの呼び出し方法

クロス・リファレンス・ユーティリティを使うためには、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ことが必要です。アセンブリ言語ファイルは -x オプションを使ってアセンブルします。このオプションを使うと、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、 -s オプションを使ってアセンブラを呼び出すと、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・リスタを呼び出すには、以下のように入力します。

dspxref [-*options*] [*input filename* [*output filename*]]

dspxref クロス・リファレンス・ユーティリティを呼び出すコマンドです。

options 使いたいクロス・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することができます。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 （小文字の L）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 -lnum 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かまいません。デフォルトは 1 ページ当り 60 行です。

-q （静止）見出しとすべての進捗情報を抑制します。

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。

10.3 クロス・リファレンス・リストの例

```
================================================================================

Symbol: done

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____  _________  _________    _____  _______  _______  _______
x.asm            STAT    '000c       380c        18      14

================================================================================

Symbol: f1

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____  _________  _________    _____  _______  _______  _______
x.asm            STAT   3.45e+00   3.45e+00       4

================================================================================

Symbol: g3

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____  _________  _________    _____  _______  _______  _______
x.asm            EDEF     ffff       ffff        3       9
y.asm            EREF     0000       ffff                3        6

================================================================================

Symbol: start

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____  _________  _________    _____  _______  _______  _______
x.asm            STAT    '0000       3800        12      17

================================================================================

Symbol: table

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____  _________  _________    _____  _______  _______  _______
x.asm            STAT    -1000       3000        21      13

================================================================================

Symbol: y

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____  _________  _________    _____  _______  _______  _______
x.asm            EREF     0000       380d                7       10
y.asm            EDEF    '0000       380d         5      1

================================================================================
```

以下に定義した用語は、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 シンボルが使わ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。参照タイプには以下のようなものがあります。

- **STAT** シンボルはこのファイルで定義されていますが、グローバルとして宣言されてはいません。
- **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はいませんが、グローバルとして参照されています。
- **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もいません。

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。値の前にシンボルの属性を示す文字が付くこともあります。このような文字と名前を表 10 - 1 に掲載してあります。

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修正される可能性があります。この欄が空白の場合は、そのシンボルが使用されないことを示します。

表10 - 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 **セクションまたは** .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 11 章

Hex 変換ユーティリティの説明

TMS320C1x/C2x/C2xx/C5x 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・メモリを管理する柔軟な方式を提供するバイナリ・オブジェクト・ファイル・フォーマットです。

大部分の EPROM プログラマは、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て、'C26 および 'C5x のコード生成プロセスを自動化します。

Hex 変換ユーティリティでは、次のような出力ファイル・フォーマットを生成できます。

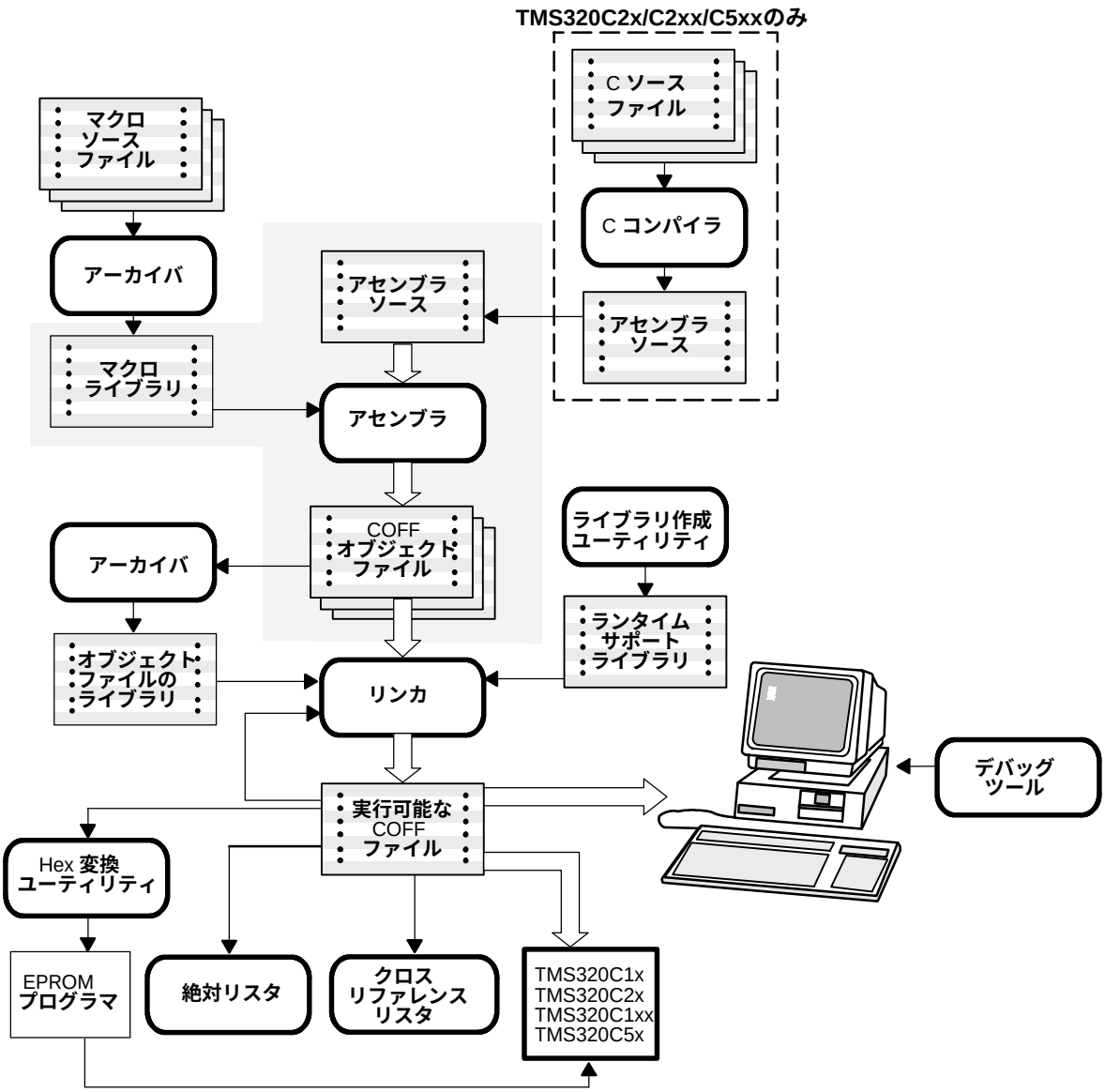
- ❑ 16 ビット・アドレスをサポートする ASCII - Hex
- ❑ Extended Tektronix (Tektronix)
- ❑ Intel MCS - 86 (Intel)
- ❑ 16 ビット・アドレスをサポートする Motorola Exorciser (Motorola - S)
- ❑ 16 ビット・アドレスをサポートする Texas Instruments SDSMAC (TI - Tagged)

項目 **ページ**

11.1 Hex 変換ユーティリティ開発フロー

図 11 - 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 11 - 1. Hex 変換ユーティリティ開発フロー



11.2 Hex 変換ユーティリティの呼び出し方法

Hex 変換ユーティリティを呼び出すには、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する。**以下の例では、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。

  ```
  dsphex -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する。**Hex 変換ユーティリティを呼び出すためのコマンド行オプションとファイル名を格納したコマンド・ファイルを作成することができます。以下の例では、hexutil.cmdというコマンド・ファイルを使ってユーティリティを呼び出しています。

  ```
  dsphex 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 疑似命令を使うことができます。

Hex 変換ユーティリティをコマンド行から呼び出す方法

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。

dsphex [*-options*] *filename*

dsphex Hex 変換ユーティリティを呼び出すコマンドです。

-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の綴りで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が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。

表11 - 1 は、基本オプションの一覧です。11 - 27 ページの表 11 - 2 は、'C26 および 'C5x のオンチップ・ブート・ローダのみに適用されるオプションの一覧です。このブート・ローダについては、11-26 ページの 11.9 節で詳しく説明しています。

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・ファイルの詳細については、11-5 ページの 11.3 節を参照してください)。

表11 - 1. 基本オプション

一般オプション	オプション	説明	参照ページ
Hex 変換ユーティリティの動作全体を制御します。	-byte	バイトを順にカウントします。	11-36
	-map *filename*	マップ・ファイルを作成します。	11-19
	-o *filename*	出力ファイル名を指定します。	11-22
	-q	静かに実行します（使用する場合は他のオプションより前に指定する必要があります）	11-5
イメージ・オプション	**オプション**	**説明**	**参照ページ**
ある範囲のターゲット・メモリを連続したイメージで作成します。	-fill *value*	ホールに値を埋め込みます。	11-24
	-image	イメージ・モードを指定します。	11-24
	-zero	アドレス起点をゼロにリセットします。	11-35
メモリ・オプション	**オプション**	**説明**	**参照ページ**
出力ファイルのメモリ幅を構成します。	-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 =16 ビット)。	11-8
	-order {LS | MS}	メモリ・ワードの順番を指定します。	11-12
	-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って異なります)。	11-9
出力フォーマット	**オプション**	**説明**	**参照ページ**
出力フォーマットを指定します。	- a	ASCII - Hex を選択します。	11-39
	- i	Intel を選択します。	11-40
	- m	Motorola - S を選択します。	11-41
	- t	TI - Tagged を選択します。	11-42
	- x	Tektronix を選択します。	11-43

11.3 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には、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 Hex 変換ユーティリティ疑似命令を使って変換処理をカスタマイズする場合にも、コマンド・ファイルが役立ちます。

コマンド・ファイルは、以下の情報が 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まった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 疑似命令の詳細については 11-14 ページの 11.5 節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 疑似命令の詳細については、それぞれ 11-20 ページの 11.6 節を参照して下さい) 。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・ローダの詳細については 11-26 ページの 11.9 節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。たとえば、次のように指定します。

```
/* 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。

dsphex *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。次のように、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ユーティリティを起動することもできます。

```
dsphex 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は、重要ではありません。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制します。

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 */
-o   firm.msb  /* output file */
```

以下のように入力すれば、Hex 変換ユーティリティを呼び出すことができます。

```
dsphex firmware.cmd
```

- この例では、appl.out というファイルを Intel フォーマットの 4 つの Hex ファイルに変換します。各出力ファイルは幅が 1 バイトで長さが 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。

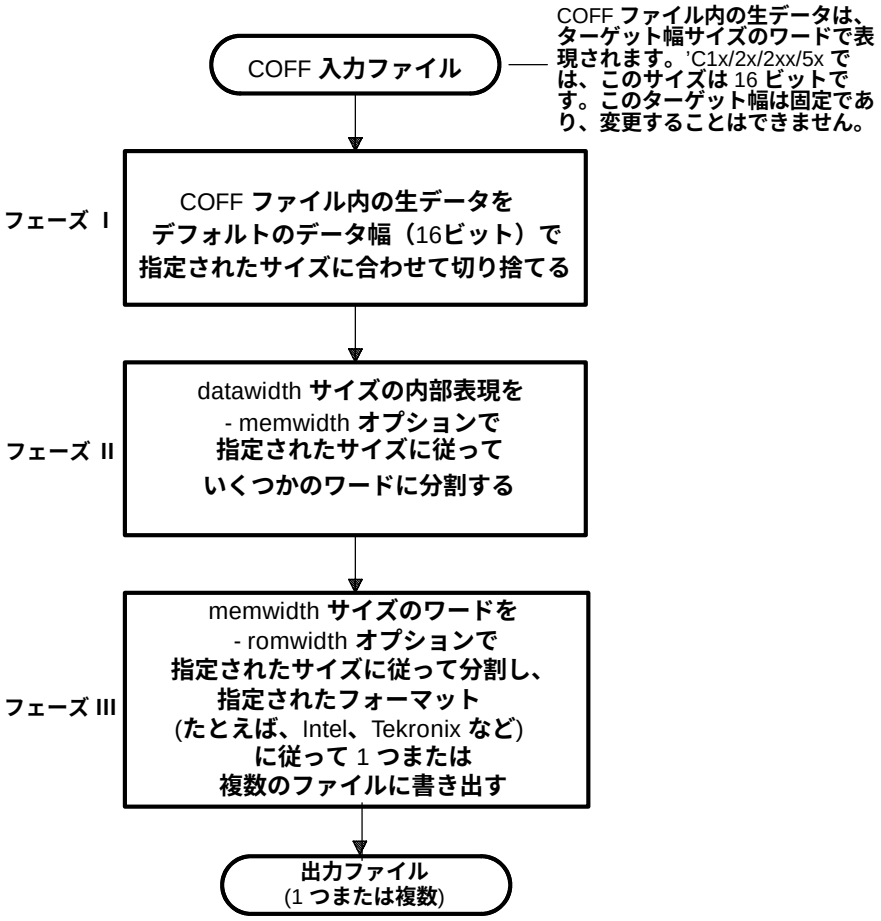
```
appl.out         /* input file   */
-i               /* Intel format */
-map appl.mxp    /* map file     */

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:
}
```

11.4 メモリ幅について

Hex 変換ユーティリティは、ユーザがメモリ幅と ROM 幅を指定できるようにすることによって、メモリ・アーキテクチャの柔軟性をさらに高めます。He変換ユーティリティを使うためには、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。変換プロセスでは、ターゲット幅、データ幅、メモリ幅、ROM 幅の 4 つの幅が重要な意味を持ちます。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、このような幅を持つワードを意味します。

図 11 - 2 は、Hex 変換ユーテイリティの処理の流れにおける 3 つの独立した個別フェーズを示したものです。

図 11 - 2. Hex 変換ユーティリティの処理の流れ



11.4.1 ターゲット幅

ターゲット幅とは、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。TMS320C1x、TMS320C2x、TMS320C2xx、TMS320C5x のターゲット幅は 16 ビットです。

11.4.2 データ幅

データ幅とは、COFF ファイルの特定のセクションに格納されるデータ・ワードの論理的な幅 (ビット数) です。通常、論理的なデータ幅はターゲット幅と同じになります。TMS320C1x/2x/2xx/5x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。

11.4.3 メモリ幅

メモリ幅とは、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・アーキテクチャを持ちます。ただし、ブート・ローダなどの一部のアプリケーションでは、ターゲット・ワードをより短いワードに分割して、連続した複数のメモリ・ワードを作成する必要があります。さらに、'C5xのような特定のプロセッサでは、メモリ幅がターゲット幅より小さくなることがあります。

Hex 変換ユーティリティは、メモリ幅のデフォルト値としてターゲット幅（この場合は、16 ビット）を使用します。

メモリ幅は、次のような方法で変更できます。

- ❑ **- memwidth** オプションを使用する。この方法をとると、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** パラメータを設定する。この方法をとると、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、そのアドレス範囲に対する - memwidth オプションが無視されます。11 - 14 ページの 11.5 節を参照してください。

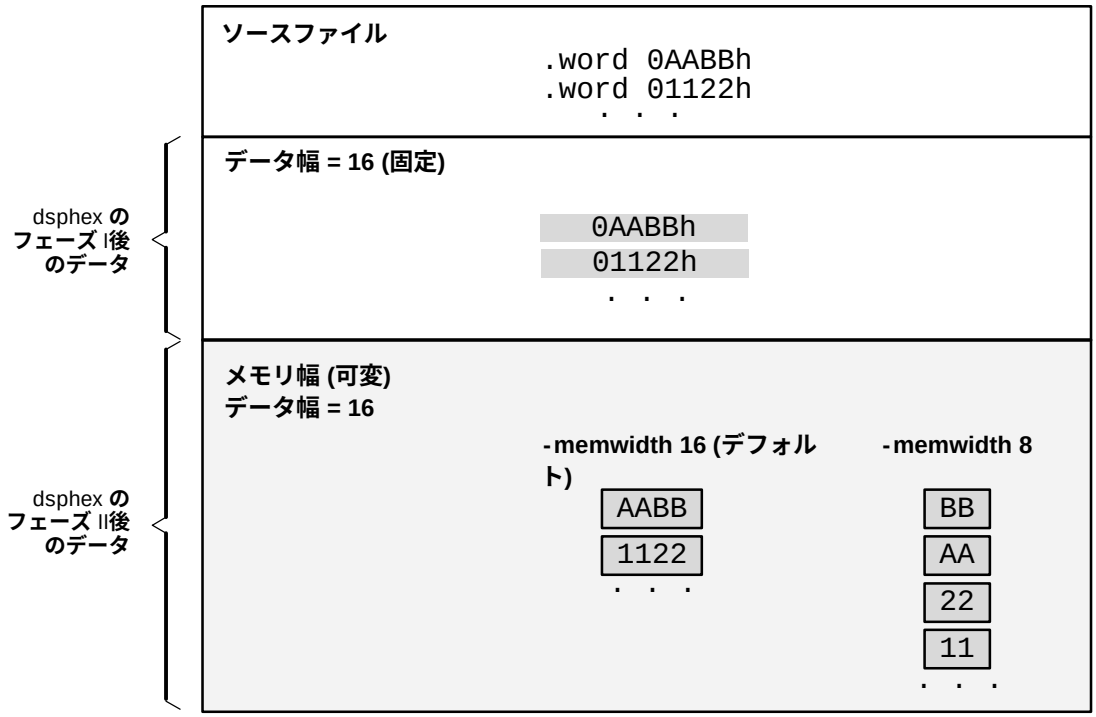
どちらの場合も、使用する値は、８以上の２のベキ乗値でなければなりません。

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・ワードをそれより短い幅の連続した複数のメモリ・ワードに分解する必要がある場合などのように、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・ワードより短いという状況がよく起こるのは、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・ワードよりも短いメモリ・ワードからの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TMS320C5xは８ビットのメモリまたは８ビットのシリアル・ポートからブートすることができます。この場合は、それぞれの 16 ビット値が２つのメモリ・ロケーションを占めます（これは、 - memwidth 8 として指定します）。

メモリ幅がデータ幅とどのように関連するかを図 11 - 3 に示します。

図 11 - 3. **データ幅とメモリ幅**



11.4.4 ROM 幅

ROM 幅は、それぞれの 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 幅によって決まります。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 つまたは複数の出力ファイルに分割されます。この出力ファイルの数は、次のような式で求めることができます。

ファイルの数 = メモリ幅÷ ROM 幅

ただし　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、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。

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 ビットの各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 フォーマットは 16 ビットです。TI - Tagged タグ付きフォーマットのデフォルトの 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 フォーマットでは、16 ビットの ROM 幅のみがサポートされます。

ROM 幅は（TI - Tagged フォーマットの場合を除き）次のような方法で変更することができます。

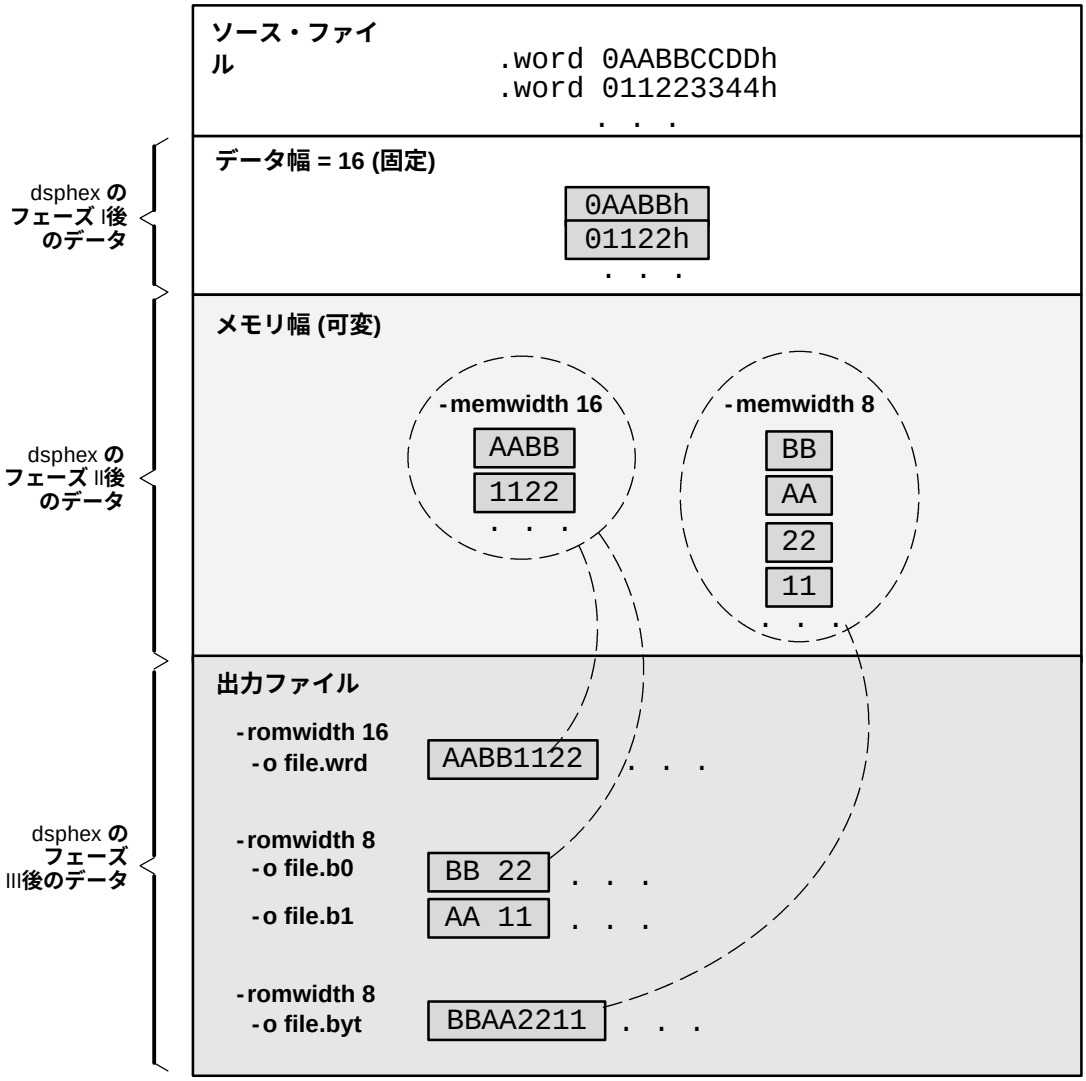
- **- romwidth オプションを使用する。この方法をとると、**COFF **ファイル全体について** ROM **幅の値が変更されます。**
- ROMS **疑似命令の** romwidth **パラメータを設定する。この方法をとると、特定の** ROM **アドレス範囲について** ROM **幅の値が変更され、そのアドレス範囲に対する** - romwidth **オプションが無視されます。**11-14 **ページの** 11.5 **節を参照してください。**

上記のどちらの場合も、８以上の２のベキ乗値を使用します。

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)より大きい** ROM **幅を選択した場合、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むだけです。**

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するかを 図** 11 - 4 **に示します。**

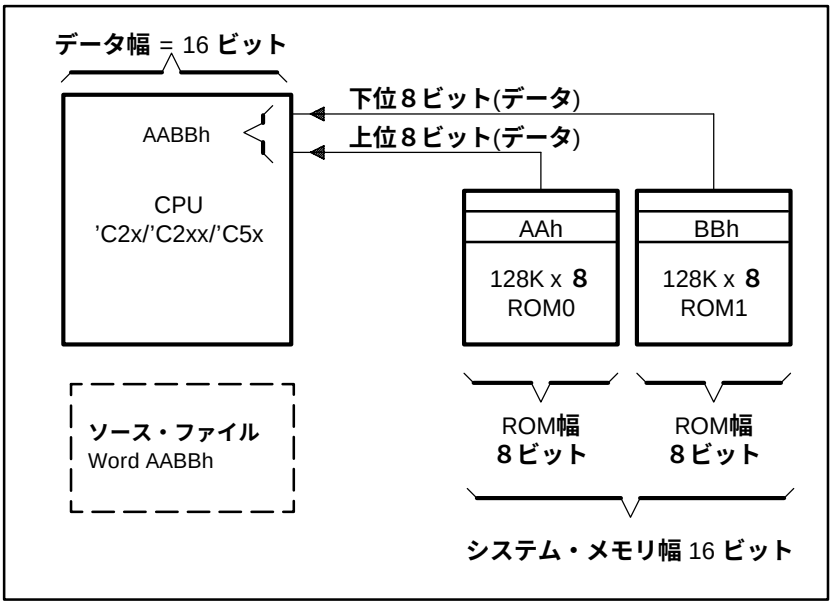
図 11 - 4. **データ幅、メモリ幅、および** ROM **幅**



11.4.5 メモリ構成の例

図 11 - 5 は、一般的なメモリ構成の例を示したものです。このメモリ・システムは、128K×8 ビットの ROM デバイスが 2 つで構成されます。

図 11 - 5. ’C2x/C2xx/C5xx **のメモリ構成の例**



11.4.6 出力ワードのワード配列の指定

メモリ・ワードがターゲット・ワードよりも短い (メモリ幅 <16) 場合、ターゲット・ワードは連続した複数のメモリ・ワードに分割されます。幅の広いワードを 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、以下の 2 つの方法があります。

- ❏ **- order MS** によって**ビッグ・エンディアン配列**を指定する。この場合は、連続したロケーションの先頭にワードの最上位部分が置かれます。
- ❏ **- order LS** によって**リトル・エンディアン配列**を指定する。この場合は、連続したロケーションの先頭にワードの最下位部分が置かれます。

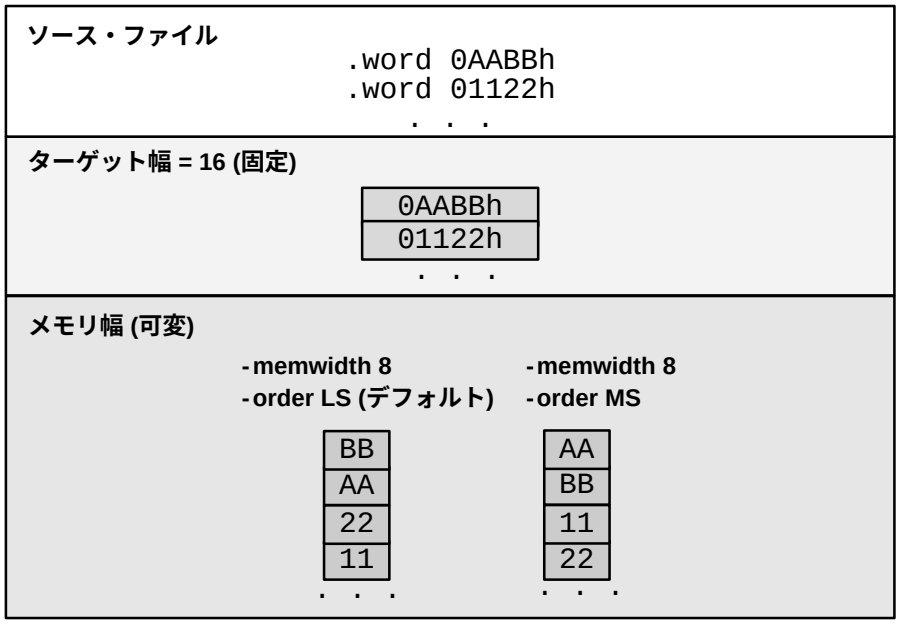
’C26 と ’C5x のブート・ローダはリトル・エンディアンを前提としているので、デフォルトではリトル・エンディアン・フォーマットが使用されます。 - order MS は、独自のブート・ローダ・プログラムを使用している場合以外は使用しないでください。

注: - order オプションが適用される場合

- このオプションは、16 より小さい値のメモリ幅を使用している場合にだけ適用されます。それ以外の場合、 - order は無視されます。
- このオプションは、メモリ・ワードがそれぞれの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 オプションとは無関係に、必ず最下位のファイルの名前を最初に指定します。

図 11 - 6 は、 - order が変換処理にどのような影響を与えるかを示したものです。この図と、前出の図 11 - 4 では、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 11 - 6. ワード配列の変化



11.5 ROMS 疑似命令

ROMS 疑似命令では、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。その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ないます。

ROMS 疑似命令は、TMS320C1x/C2x/C2xx/C5x 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、それぞれの行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れば、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 を指定しなかった場合は、すべての範囲が 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持ちません。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、または 16 進の定数でなければなりません。origin の値を省略すると、origin のデフォルト値として 0 が使用されます。

以下の表は、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のまとめです。

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 と入力することができます。値は、10 進、8 進、または 16 進の定数で指定する必要があります。length の値を省略した場合は、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 幅をビット数で指定します (11-9 ページの 11.4.4 項を参照してください)。ここで値を指定すると、- romwidth オプションはその値によって変更されます。 8 以上の 2 のベキ乗値を10 進、8 進、または 16 進の定数として指定する必要があります。

memwidth アドレス範囲のメモリ幅をビット数で指定します (11-8 ページの 11.4.3 項を参照して下さい)。ここで値を指定すると、 - memwidth オプションはその値によって変更されます。 8 以上の 2 のベキ乗値を 10 進、8 進、または16 進の定数として指定する必要があります。memwidth パラメータを使用する場合は、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する必要があります。

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、あるアドレス範囲のセクション間にあるホール (穴) 部分がこの値で埋め込まれます。ターゲット幅と同じ長さを持つ値を10 進、8 進、または 16 進の定数として指定する必要があります。ここで値を指定すると、 - fill オプションはその値によって変更されます。fill を使用する場合には、 - image コマンド行オプションも使用する必要があります。11 - 25 ページの 11.8.2 項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。出力ファイルの数の計算方法については、11 - 9 ページの 11.4.4 項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指定が任意です。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、すべての範囲について origin と length が必要です。コンマと等号も指定が任意です。

同じページのアドレス範囲が重複してはなりません。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。

11.5.1 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には、プログラムのアドレス空間全体 (PAGE 0) が含まれる単一のデフォルトのメモリ範囲がユーティリティ側で定義されます。これは、ROMS 疑似命令で origin または length を指定せずに単一の範囲を指定した場合と同じです。

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したい場合。**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 つまたは複数) 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、ユーザの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出されません。この方法をとると、SECTIONS 疑似命令でセクションの名前を列挙しなくても、特定のセクションを除外することができます。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、警告が出され、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場合は、必ず ROMS 疑似命令を使用して下さい。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るように各範囲がすべて埋め込まれます。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、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。

11.5.2 ROMS 疑似命令の例

例 11 - 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たものです。

例 11 - 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、次のような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、以下の 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 では、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、次のようなセクションが含まれます。

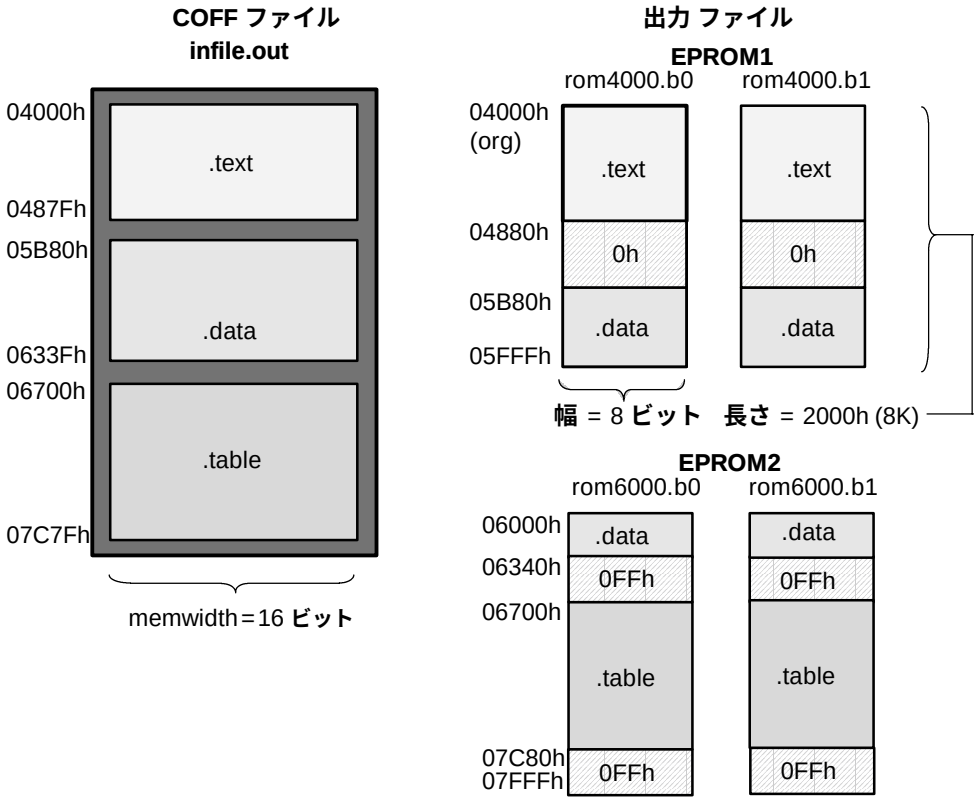
セクション	対応する範囲
.data	6000h〜633Fh
.table	6700h〜7C7Fh

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、以下の 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 11 - 7 は、この ROMS 疑似命令によって、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たものです。

図 11 - 7. **例** 11 - 1 **の** infile.out **ファイルの** 4 **個の出力ファイルへの分割**



11.5.3 ROMS 疑似命令のマップ・ファイルの作成

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している場合に非常に役立ちます。マップ・ファイルには、それぞれのアドレス範囲、各範囲のパラメータ、関連する出力ファイルの名前、およびアドレスごとに分析した内容のリスト (セクション名と埋め込み値) が示されます。例 11 - 1 の例で作成されるマップ・ファイルの一部を以下に示します。

例 11 - 2.　例 11 - 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
```

11.6 SECTIONS 疑似命令

SECTIONS 疑似命令で名前を指定することによって、COFF ファイルの特定のセクションを変換することができます。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したいセクションや、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 に配置したいセクションを指定することもできます。

以下の点に注意して下さい。

- ❑ SECTIONS 疑似命令を使用した場合は、この疑似命令で指定した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かった場合は、構成メモリの範囲内にある初期化されたセクションがすべて変換されます。TMS320C2x/2xx/5x 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、および .cinit 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 コンパイラが生成するセクション

TMS320C2x/C2xx/C5x 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.
- ❑ **初期化されないセクション：** .bss, .stack, .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・ファイルの使用法に関する詳細は、11-5 ページの 11.3 節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出され、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 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、この値によって変更されます。(11-34 ページの 11.10 節を参照して下さい)。この値には、10 進、8 進、または 16 進の定数を指定する必要があります。「**boot**」を指定することもできます（オンチップ・ブート・ローダで使うブート・テーブル・セクションを指定するため）。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 パラメータを使う場合は、すべてのセクションで paddr パラメータを使う必要があります。

= 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** と指定した場合と同じです。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・ローダ特有のコマンド行オプションの両方により決まります。

コンマの指定は任意です。リンカの SECTIONS 疑似命令との共通性を持たせるために、セクション名の後で (boot キーワードの等号の代わりに) コロンを使用することができます。たとえば、以下の 2 つの文は等価です。

```
SECTIONS { .text: .data: boot }
SECTIONS { .text, .data = boot }
```

以下の例では、COFF ファイル内に .text、.data、.const、.vectors、.coeff、および .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 のみを変換したい場合は、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 オプションと SECTIONS 疑似命令の使用法

オンチップ・ブート・ローダに対して SECTIONS 疑似命令を使用した場合、 - 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については、11-27 ページの表11 - 2を参照して下さい。

11.7 出力ファイル名

Hex 変換ユーティリティは、COFF オブジェクト・ファイルをあるデータ形式に変換するときに、データを 1 つまたは複数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って、複数のファイルが作成される場合は、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。

出力ファイル名の割り当て

Hex 変換ユーティリティは、以下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 = {. . .}) が指定されている場合は、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・データが 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って、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することができます。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は、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。以下のように指定すると、前出の 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下さい。

3) **デフォルトのファイル名を割り当てます。**ユーザがファイル名を指定しなかった場合や、指定したファイル名の数が出力ファイルの数より少なかった場合は、ユーティリティ側で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 文字の拡張子で構成されます。拡張子には、以下の 3 つの部分があります。

   a) 以下の出力フォーマットに基づいたフォーマット文字。

      a ASCII - Hex の場合
      i Intel の場合
      t TI - Tagged の場合
      m Motorola - S の場合
      x Tektronix の場合

   b) ROMS 疑似命令での範囲番号。範囲には 1 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、ユーティリティによって coff.i0、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を起動するときに以下の ROMS 疑似命令を指定すると、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.i01	1000h〜1FFFh
coff.i11	2000h〜2FFFh

11.8 イメージ・モードと - 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持つ出力ファイルを作成する方法について説明します。

11.8.1 - 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・イメージのすべてを完全に埋め込むことによって、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。アドレス空間内の各セクション間に、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場合、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 プログラマでは、アドレスの不連続がサポートされていません。

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。 それぞれの出力ファイルには、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、ユーザが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、それぞれの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・レコードを持ちます。ただし、イメージ・モードでは、これらのアドレスが必ず連続したものになります。

注: ターゲット・メモリの範囲の定義

イメージ・モードを使用する場合には、ROMS 疑似命令も指定する必要があります。イメージ・モードでは、それぞれの出力ファイルが、ある範囲のターゲット・メモリに直接対応します。ユーザはこの範囲を定義する必要があります。ユーザがターゲット・メモリの範囲を定義しなかった場合、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。

11.8.2 埋め込み値の指定

- fill オプションでは、セクション間の空き部分の埋め込みに使用する値を指定します。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みなされます。たとえば、 'C2x 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。

ユーザが fill オプションを使って値を指定しなかった場合、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。 - fill オプションが有効なのは、 - image を使用する場合だけです。それ以外の場合、このオプションは無視されます。

11.8.3 イメージ・モードを使用する手順

手順 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 (11 - 14 ページの 11.5 節を参照してください)。

手順 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には、 - 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 にリセットするには、 - 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 11-36 ページの 11.10.3 項を、 - zero オプションの詳細については 11-35 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していないときに、デフォルト以外の値を使用したい場合は、 - fill オプションを使用してください。

11.9 オンチップ・ブート・ローダ用のテーブルの作成

’C26、’C5x 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。そのブート・ローダは、1 つまたは複数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。そのブート・ローダでは、メモリ (EPROM など) に格納されたり、ペリフェラル (シリアル通信ポートなど)からロードされたりする特殊なテーブル (**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ってブート・ローダをサポートします。

11.9.1 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 (あるいはソース・プログラム) 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・ブロックを転送先アドレスにコピーするようにオンチップ・ローダに指示するレコードが入ります。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・テーブルに入ることもあります。ブート・テーブルは、メモリ内に格納できるほか、ペリフェラルから読み込むこともできます。

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・ローダで初期化したい 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することができます。Hex 変換ユーティリティでは、COFF ファイルをもとにターゲット・デバイスの型を識別して、そのデバイスで必要なフォーマットに従ってテーブルの完全なイメージを作成し、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。そして、ユーザはそのテーブルを ROM に書き込んだり、その他の手段でロードしたりすることができます。

ブート・ローダは、通常のメモリ幅より幅が小さいメモリからのローディングをサポートします。たとえば、 - memwidth オプションを使用してブート・テーブルの幅を構成することによって 16 ビットの TMS320C26 を単一の 8 ビット EPROM からブートすることができます。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、TMS320C2x User’s Guide の TMS320C26BFNL ブート・ローダの例を参照してください。

11.9.2 ブート・テーブル・フォーマット

ブート・テーブル・フォーマットは単純です。一般に、テーブルの幅と場合によっては各種の制御レジスタ用の値が入っているヘッダー・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ーがあり、ヘッダーの後にそのブロックのデータがあります。ブロックは、複数入力することができます。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ーを持つことがあります。詳細については、それぞれのデバイスのユーザーズ・マニュアルにあるブート・ローダの節を参照してください。

11.9.3 ブート・テーブルの作成方法

表11 - 2 は、ブート・ローダについて使用可能な Hex 変換ユーティリティ・オプションをまとめたものです。

表11 - 2. ブート・ローダ用のユーティリティ・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 PARALLEL	ブート・ローダ・テーブルのソースにパラレル・ポートを指定します。
-bootorg SERIAL	ブート・ローダ・テーブルのソースにシリアル・ポートを指定します。
- 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
- cg value	メモリ構成レジスタを設定します ('C26 のみ)

ブート・テーブルを作成するには、以下の手順に従います。

手順 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。(11-20 ページの 11.6 節を参照してください) 。

ユーザがブート・ローダ・テーブル内に置く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・フィールドに入れます。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ユーザが考慮する必要はありません。これについては Hex 変換ユーティリティが取り扱います。

手順 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・ローディング用に構成するように Hex 変換ユーティリティに指示することができます。また、SECTIONS 疑似命令を使用して、特定のセクションを選択して構成することもできます。(11-20 ページの 11.6 節を参照してください) 。SECTIONS 疑似命令を使用した場合は、 - boot オプションが無視される点に注意してください。

手順 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、'C26 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・フィールドが 8000h から開始されることになります。

- bootorg SERIAL または - bootorg PARALLEL を使用した場合、または - bootorg オプションをまった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 疑似命令を使用していない場合は、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。

手順 4: **ブート・ローダ特有のオプションを設定します。**必要に応じて、エントリ・ポイント、メモリ制御レジスタなどを設定します。

手順 5: **システム・メモリ構成を記述します。**詳細については、11-7 ページの 11.4 節、および 11-14 ページの 11.5 節を参照してください。

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ー・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」のアドレスがブート・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。

メモリ・マップには必ずブート・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場合には注意が必要です。ブート・テーブルは、他の非ブート・セクションや未構成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・テーブルのために予約されます。このメモリを、ROMS 疑似命令で単に 1 つまたは複数の範囲として構成し、 - 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。

11.9.4 ペリフェラルからのブート

- bootorg オプションで SERIAL または PARALLEL キーワードを使用すると、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。キーワードは、ターゲット・デバイスと使用したいチャネルに応じて選択します。たとえば、'C26 をシリアル・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 - bootorg SERIALと指定します。また、'C5x をパラレル・ポートのどれか 1 つからブートするには、 - bootorg PARALLEL と指定します。

注：

- **メモリ・コンフリクトの可能性**
  ペリフ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。ペリフェラルを通じてブート・テーブルを受け取ります。ただし、11-27 ページの手順 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、このページの範囲と - bootpage オプションを定義して、ブート・テーブルをこのページに置きます。これによってブート・テーブルは、ダミーページの 0 の位置に置かれます。

- **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、親プロセッサが子プロセッサのペリフェラルを通じ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。

11.9.5 ブート・テーブルのエントリ・ポイントの設定

ブート・ロード処理が完了すると、ロードされた最初のブロックのアドレス (デフォルトのエントリ・ポイント) からプログラムの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って、このエントリ・ポイントを別のアドレスに設定することができます。

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には、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、リンカの生成するマップ・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

値は定数でなければなりません。Hex 変換ユーティリティは、c_int00（TMS320C2x/C2xx/C5x C コンパイラが割り当てるデフォルトのエントリ・ポイント）のようなシンボル式は評価できません。

- e オプションを使用すると、ユーティリティは指定されたアドレスにロードする長さ 1 およびデータ値 0 のダミー・ブロックを作成します。ユーザの指定したブロックは、このダミー・ブロックの後に続きます。ダミー・ブロックが最初にブート・ロードされるため、ダミー値の 0 は後続のブロックによって上書きされます。そして、ブート・ロードの完了後に、ブート・ローダは - e オプションのアドレスにジャンプします。

11.9.6 'C26 ブート・ローダの使用法

この項では、'C26 デバイスのブート・ローダで Hex 変換ユーティリティを使用する方法について説明し、例を示します。'C26 のブート・ローダには 4 種類のモードがあります。これらのモードは - bootorg オプションと - memwidth オプションを使って選択することができます。

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
8 ビット EPROM	- bootorg 0x8000	- memwidth 8

複数の出力ファイルを作成したい場合以外は、 - romwidth と - memwidth は等しい設定にする必要があります。

シリアル・モードとパラレル・モード

ブート・テーブルを作成するときのパラレル・モードとシリアル・モードのHex変換ユーティリティの動作について、以下に概要を説明します。

- ❑ 2 つのパラレル・モードでは、ブート・テーブル・ヘッダーは 3 つの 8 ビット・フィールド（STATUS、CONFIG、LENGTH）で構成されます。ユーティリティはこれらのフィールドを自動的にテーブルに組み込みます。
  - ■ STATUS フィールドのビット 0 ～ 2 は、ロード長の MSB 3 ビット を定義します。
  - ■ STATUS フィールドのビット 3 の値は常に 1 で、ブート・ローダを有効にします。
  - ■ STATUS フィールドのビット 4 は、8 ビット・モードの場合は 0 に、16 ビット・モードの場合は 1 に設定されます。
  - ■ ビット 5 ～ 7 は予約ビットです。
  - ■ CONFIG フィールドは、ユーザが指定する - cg オプションに基づいて設定されます。
  - ■ LENGTH フィールドには、ロード長の LSB8 ビット が入っています。
- ❑ シリアル・モードでは、追加の BAUD フィールドがヘッダーの最初のフィールドになります。ブート・ローダは MSB が 1 の任意の値を受け入れます。ユーティリティは 80h を使用します。シリアル・モードの STATUS、CONFIG、LENGTH の各フィールドは、パラレル・モードで使用される場合と同じです（上記を参照）。
- ❑ シリアル・モードとパラレル・モードのどちらの場合も、ブート・テーブル内のデータの後に 1 つの SYNC フィールドが続きます。SYNC の値には意味はありません。ユーティリティは 0 を使用します。
- ❑ どちらの 8 ビット・フォーマットの場合も、データを最初に LSB があるようにフォーマットする必要があります。これはユーティリティが自動的に行います。

以下の制限はすべてのモードに適用します。

- 1 つのセクションしかロードできません。
- セクションの最大長は 2K ワードです。
- 転送先アドレスは固定です（内部 RAM ブロック B0）。

例 11 - 3 に示すプログラムでは、シリアル・ポートを通じて abc.obj の .text セクションをブートしています。まず、.text をオンチップ RAM ブロック B0 にリンクし、長さが 2K ワードを越えないことを確認します。次に、このコマンド・ファイルを使って 16 進フォーマットに変換します。

例 11 - 3. 'C26 のシリアル・ポートからブートするためのコマンド・ファイルの例

```
abc.out             /* input file                          */
-c abc.i            /* output file                         */
-i                  /* Intel format                        */
-memwidth 8         /* 8-bit serial                        */
-romwidth 8         /* outfile is bytes, not words         */
-bootorg SERIAL     /* serial mode boot                    */
-cg 40h             /* value for interrupt/mem config      */

SECTIONS { .text: BOOT }
```

この結果作成される Hex ファイルは、デバイスをブートするためにシリアル・ポートを通じて送信できる 8 ビット・バイトのリストになります。

メモリ構成および割り込みマスク・レジスタの設定（シリアル・モードとパラレル・モード）

'C26 のブート・ローダは、データをメモリにロードするほか、ブート・テーブルに指定されている値に基づいてメモリ構成および割り込みマスク・レジスタ（IMR）を初期化することもできます。Hex 変換ユーティリティの使用時には、下記のように - cg オプションを使ってメモリ構成および IMR 用の値を設定することができます。

-cg *value*

このオプションでは、引数として定数値が必要です。各ビットの定義は以下のとおりです。

- ビット D0 ～ D5 は IMR にロードされます。
- ビット D6 および D7 はメモリ構成を定義します。メモリをどのように構成できるかを次の表に示します。

D7	D6	プログラム・メモり	データ・メモリ
0	0	B0	B1, B2
0	1	B0, B1	B2
1	0	B0, B1, B3	

EPROM モード

'C26 の EPROM モードは、シリアル・モードやパラレル・モードとは大きく異なります。ヘッダー情報もフッター情報もなく、代わりに、ブート・テーブルはオンチップ RAM ブロックのメモリ・イメージとして 8 ビット・メモリに構成され、MSB を先頭にしてロケーション 08000h から配列されます。バイト幅の EPROM から 'C26 をブートするには、Hex 変換ユーティリティを呼び出すときに - bootorg 08000h および - memwidth 8 を指定するだけです。'C26 は、グローバル・メモリのアドレス 08000h で EPROM を検出できるものと予期します。

11.9.7 'C5x Boot Loader の使用

このサブセクションでは、'C5x デバイス用に、Hex 変換ユーティリティをブート・ローダで使用する方法について説明と例を挙げます。'C5x ブート・ローダは、6 つの異なるモードを持っています。これらのモードは、 - bootorg と - memwidth オプションを使用して選択できます。

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

複数の出力ファイルを作成したい場合以外は、 - romwidth と - memwidth は等しい設定にする必要があります。

'C5x は、8 ビットまたは 16 ビットのデータ幅をもつシリアル・インターフェイスまたはパラレル・インターフェイス、のどれを使ってもブートできます。どの組み合わせの場合も、フォーマットは同じです。ブート・テーブルは、転送先アドレスが入っているフィールド、長さ値が入っているフィールド、データが入っているブロックで構成されます。

ブートできるセクションは 1 つだけです。8 ビット・チャネルからブートする場合は、16 ビット・ワードは MSB から先にテーブルに格納されます。Hex 変換ユーティリティがテーブルを正しいフォーマットで自動的に作成します。

- ❑ シリアル・ポートからブートするには、ユーティリティを呼び出すときに - bootorg SERIAL を指定します。そして、 - memwidth 8 または - memwitdh 16 を使用します。
- ❑ パラレル I/O ポートからロードするには、 - bootorg PARALLEL を指定してユーティリティを呼び出します。そして、 - memwidth 8 または - memwitdh 16 を使用します。
- ❑ 外部メモリ（EPROM）からブートするには、 - bootorg オプションを使ってブート・メモリのソース・アドレスを指定します。そして、 - memwidth 8 または - memwitdh 16 を使用します。

たとえば、以下のコマンド・ファイルでは、ロケーション 08000h にあるバイト幅の EPROM から abc.out の .text セクションをブートします。

図 11 - 8. 'C5x 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 8000h       /* external memory boot                */

SECTIONS { .text: BOOT }
```

11.10 ROM デバイス・アドレスの制御

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・アドレスに対応しています。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ロケーションにデータを書き込みます。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ために、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、データを書き込む ROM のロケーションを直接制御する機能があります。

11.10.1 開始アドレスの制御

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って次のように異なります。

- **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って制御されます。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の順に挙げています。

  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、(リンカ・コマンド・ファイルで指定される) ロード・アドレスと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。この関係を以下にまとめます。

$$out\_file\_addr^{\dagger} = load\_addr \times (data\_width \div mem\_width)$$

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	TMS320C1x/C2x/C2xx/C5x デバイスの場合は 16 ビットとして指定されます。11-8 ページの 11.4.2 項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 パラメータで指定することができます。11 - 8 ページの 11.4.3 項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となります。データ幅がメモリ幅よりも大きい場合は、この補正係数によって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 0x1 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 0x2 となります。データはメモリ幅のサイズに相当する 2 つの連続するロケーションに分割されます。

2) **SECTIONS 疑似命令内の - paddr オプション**

あるセクションに対して - paddr 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 オプションとの関係は、次のようにまとめることができます。

$$out\_file\_addr^{\dagger} = paddr\_val + (load\_addr - sect\_beg\_load\_addr) \times (data\_width \div mem\_width)$$

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 オプションで指定された値です。
sec_beg_load_addr	リンカが割り当てたセクションの開始ロード・アドレスです。

†paddr が指定されている場合

データ幅をメモリ幅で割った値は、アドレス生成に関する補正係数です。ロード・アドレスから引いたセクション開始ロードアドレス係数が、セクションの開始点からのオフセットになります。

3) **- zero オプション**

ユーザが - 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がゼロにリセットされます。各ファイルがゼロから開始され、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ためには、このオプションと - image オプションを一緒に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出され、その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は、Hex 変換ユーティリティはブート・テーブル内の別々の COFF セクションを連続するメモリ・ロケーションに入れます。各 COFF セクションが 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・アドレスに等しくなります。

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。アドレス・フィールドは、単に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) を掛けた値を示します。

ユーザが以下のメカニズムのいずれかを使用しない限り、デフォルトによって、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・アドレスがブート・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の順で示してあります。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、優先順位の高い方のメカニズムで設定した値が使われます。

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**

   ユーザが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。 - bootorg PARALLEL も - bootorg SERIAL も、アドレス・フィールドには影響を与えません。

11.10.2 アドレス・インクリメント・インデックスの制御

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。

11.10.3 - byte オプション

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ければ、2 行目はアドレス 8（8h）から始まります。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・レコードがファイル内のバイト位置を参照するようになります。

11.10.4 アドレス・ホールの処理

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掛けられるため、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 にロードしたいとします。リンカ・コマンド・ファイルでこのロード・アドレスを 0x100 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 0x200 とします。EPROM プログラマによって EPROM の開始アドレスを制御しない限り、EPROM 内にホールが生じることになります。EPROM には、0x100 ではなく、 0x200 からデータが書き込まれます。これは、次のような方法で解決することができます。

- **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 11 - 9 は、.sec1 の開始部分にホールが生じるのを避けるようにした Hex コマンド・ファイルを示しています。

図 11 - 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
}
```

注: 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 origen パラメータを使用する（ブート・ローディング時のみ）。**

  11 - 35 ページで説明したように、ブート・ローディングを実行する場合は、- bootorg オプションまたは ROMS 疑似命令の origen によりブート・ローダ・テーブル全体の EPROM アドレスを制御することができます。別の例について、D-9 ページの D.3 節および D-18 ページの D.4 節を参照してください。

11.11 オブジェクト・フォーマットの説明

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、Exteneded Tektronix、および TI - Tagged) のいずれかに変換します。

表 11 - 3 に、フォーマット用のオプションを示します。

- ❑ これらのオプションを 2 つ以上使用した場合は、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11 - 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
- m	Motorola - S	16	8
- t	TI - Tagged	16	16
- x	Tektronix	32	8

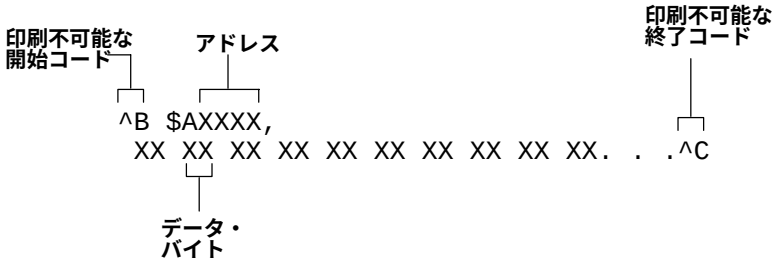
アドレスのビット数は、該当するフォーマットでサポートされるアドレス情報のビット数を示します。16 ビット・アドレスのフォーマットでは、64K までのアドレスのみがサポートされます。Hex 変換ユーティリティは、ターゲット・アドレスを有効なビット数に合わせて切り捨てます。

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 オプションを使用して変更することができます。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。

11.11.1 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 11 - 10 は、ASCII - Hex フォーマットを示したものです。

図 11 - 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) で開始され、ASCIIETX 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)によって示されます。アドレス・レコードが存在するのは、以下の場合だけに限られます。

- ❏ 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することができます。その場合は、単なるバイト値のリストとして出力が作成されます。

11.11.2 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・アドレスと 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

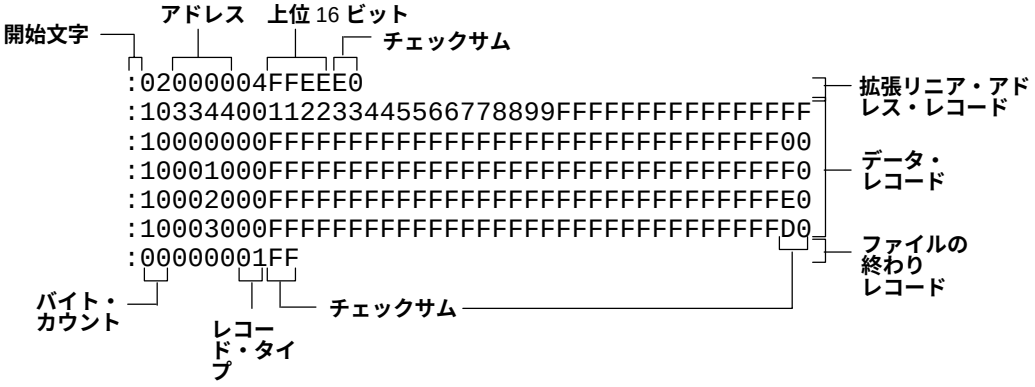
レコード・タイプ 00 (データ・レコード) は、コロン (:) で始まり、その後にバイト・カウント、最初のデータ・バイトのアドレス、レコード・タイプ (00)、およびチェックサムが順に続きます。このアドレスは、32 ビット・アドレスの下位 16 ビットであることに注意して下さい。この値が最新の 04 (拡張リニア・アドレス) レコード内の値と連結されて、完全な 32 ビット・アドレスが作成されます。チェックサムは、バイト・カウント、アドレス、データ・バイトなどのレコード内で先行する各バイトの和の 2 の補数です。

レコード・タイプ 01 (ファイルの終りレコード) は、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 11 - 11 は、Intel - Hex オブジェクト・フォーマットを示したものです。

図 11 - 11. Intel - Hex オブジェクト・フォーマット



11.11.3 Motorola Exorciser オブジェクト・フォーマット (- m オプション)

Motorola - S 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、ファイルの始め (ヘッダー) レコード、データ・レコード、およびファイルの終り (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、以下の 3 つがあります。

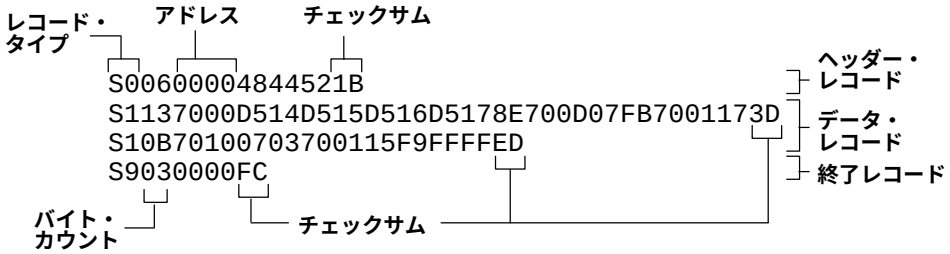
レコード・タイプ	説明
S0	ヘッダー・レコード
S1	コード/データ・レコード
S2	終了レコード

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 11 - 12 は、Motorola - S オブジェクト・フォーマットを示したものです。

図 11 - 12. Motorola - S **フォーマット**



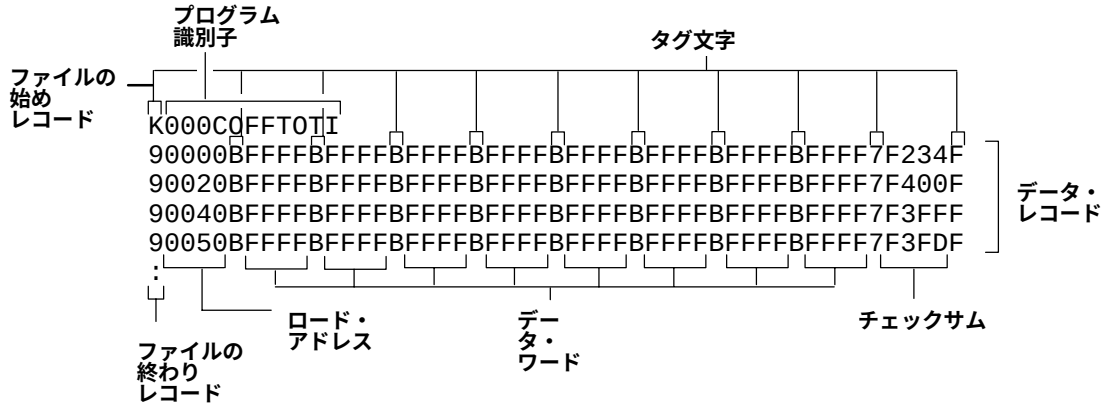
11.11.4 TI-Tagged オブジェクト・フォーマット (- t オプション)

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。それぞれのデータ・レコードは、一連の小フィールドから構成され、タグ文字によって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 11 - 13 は、TI - タグ付きオブジェクト・フォーマットのタグ文字とフィールドを示したものです。

図 11 - 13. TI - タグ付きオブジェクト・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。

11.11.5 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、以下の 2 つのレコード・タイプを持ちます。

データ・レコード　ヘッダー・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。

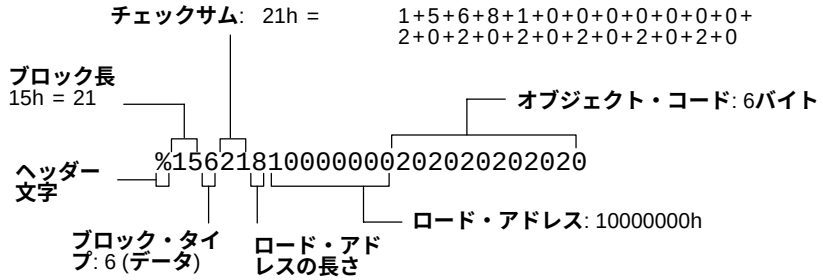
データ・レコード内のヘッダー・フィールドには、以下の情報が入ります。

項目	ASCII 文字の数	説明
%	1	データ・タイプが Tektronix フォーマットであることを示す
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・コードが置かれるアドレスを示します。最初の桁は、アドレス長を示し、常に 8 になります。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 11 - 14 は、Tektronix オブジェクト・フォーマットを示したものです。

図 11 - 14. Extended Tektronix オブジェクト・フォーマット



11.12 Hex 変換ユーティリティのエラー・メッセージ

section mapped to reserved memory

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。

処置 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下さい。有効なメモリ・ロケーションについては、それぞれのプロセッサのユーザーズ・マニュアルを参照してください。

sections overlapping

説明 複数の COFF セクションの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。

処置 この問題の原因には、メモリ幅がデータ幅より小さい場合に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。11-7 ページの 11.4 節、および 11-34 ページの 11.10 節を参照してください。

unconfigured memory error

説明 このエラーの原因には、以下の 2 つのいずれかが考えられます。

- 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。

処置 ROMS 疑似命令で定義した 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、セクションのロード・アドレスまたはブート・ローダ・テーブルのアドレスを修正して下さい。ROMS 疑似命令を使用していない場合は、デフォルトによって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に注意してください。このため、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

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、AT&T 社が UNIX ベースのシステム向けに開発した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。

セクションは COFF の基本概念です。第 2 章では、COFF セクションについて詳しく説明しています。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することができます。

この付録には、COFF オブジェクト・ファイル構造に関する技術的詳細が収録されています。この情報の大部分は、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。

この付録には以下の内容が含まれています。

項目 ページ

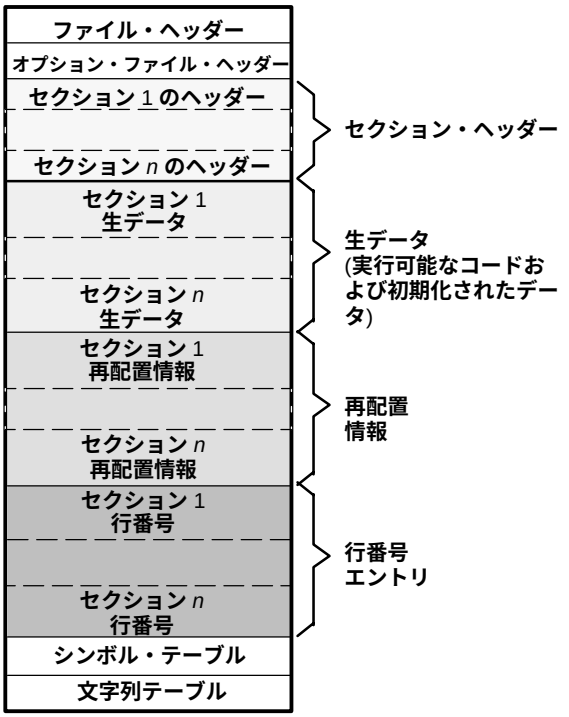
A.1 COFF ファイルの構造

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。次のような要素があります。

- ファイル・ヘッダー
- オプション・ヘッダー情報
- セクション・ヘッダーのテーブル
- 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
- 初期化されたセクションごとの行番号エントリ
- シンボル・テーブル
- 文字列テーブル

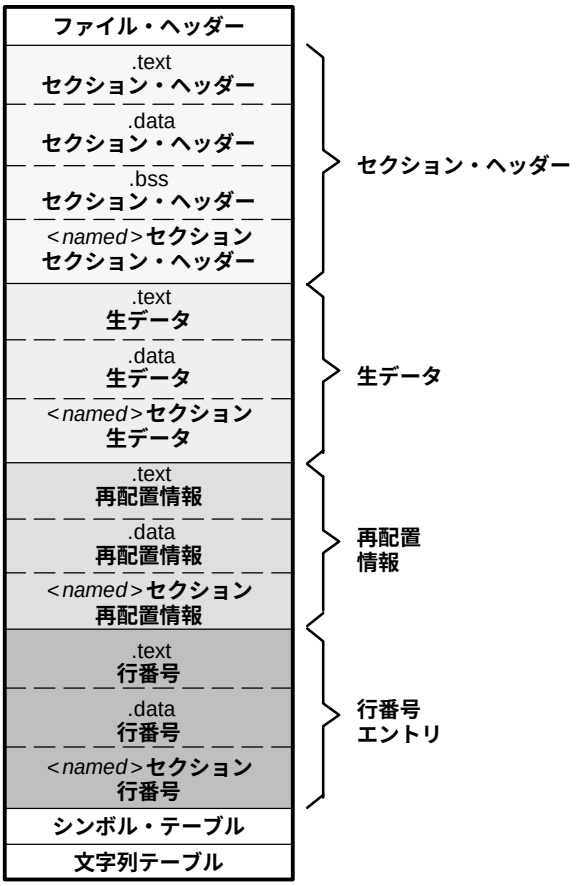
アセンブラとリンカは、同じ COFF 構造を持つオブジェクト・ファイルを作成しますが、最終的にリンクされたプログラムには、再配置エントリは通常は含まれていません。図 A - 1は、オブジェクト・ファイル全体の構造を示したものです。

図 A - 1. COFF **ファイルの構造**



COFF オブジェクト・ファイルの一般的な例を図 A - 2 に示します。このファイルには、.text、.data、および .bss という 3 つのデフォルト・セクションと、名前付きセクション (<named> で示してあります) が含まれています。デフォルトでは、.text、.data、初期化された名前付きセクション、.bss、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は、セクション・ヘッダーはありますが、生データ、再配置情報、および行番号エントリはありません。これは、.bss および .usect 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であるためです。初期化されないセクションには、実際のコードは入りません。

図 A - 2.　COFF オブジェクト・ファイル



COFF の 2 つのバージョン

この付録では、2 つのバージョン（バージョン 0 とバージョン 1）の COFF について説明しています。COFF バージョン 0 は、ツールセットのバージョン 5.00 ～ 6.40 で使用されています。COFF バージョン 1 は、それより後のすべてのバージョンのツールセットで使用されています。COFF バージョン 1 のツールでは、COFF バージョン 0 を読むことができます。ただし、COFF バージョン 0 を出力するためのリンカ・オプション -v0 を使用しない限り、COFF バージョン 1 の出力が生成されます。

A.2 ファイル・ヘッダーの構造

ファイル・ヘッダーには、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が含まれています。表 A - 1 は、バージョン 0 の COFF ファイル・ヘッダー（20 バイト）の構造を示しています。表 A - 2 は、バージョン 1 の COFF ファイル・ヘッダー（22 バイト）の構造を示しています。

表 A - 1. COFF バージョン 0 のファイル・ヘッダーの内容

バイト位置	型	説明
0〜1	unsigned short	このファイルが TMS320C2x/C5x システムで実行可能であることを示すマジック・ナンバ (092h) と、COFF のバージョンを示す 0c0h のいずれか。
2〜3	unsigned short	セクション・ヘッダーの数。
4〜7	long int	時刻および日付スタンプ。ファイルが作成された日時を示す。
8〜11	long int	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る。
12〜15	long int	シンボル・テーブル内のエントリ数。
16〜17	unsigned short	オプション・ヘッダー内のバイト数。このフィールドには、0 と 28 のいずれかが入る。0 の場合、オプション・ファイル・ヘッダーは存在しない。
18〜19	unsigned short	フラグ (表 A - 3 を参照)。

表 A - 2. COFF バージョン 1 のファイル・ヘッダーの内容

バイト位置	型	説明
0〜1	unsigned short	バージョン ID。COFF ファイル構造のバージョンを示す。
2〜3	unsigned short	セクション・ヘッダーの数。
4〜7	long int	時刻および日付スタンプ。ファイルが作成された日時を示す。
8〜11	long int	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る。
12〜15	long int	シンボル・テーブル内のエントリ数。
16〜17	unsigned short	オプション・ヘッダー内のバイト数。このフィールドには、0 か 28 のいずれかが入る。0 の場合は、オプション・ファイル・ヘッダーが存在しないことを示す。
18〜19	unsigned short	フラグ（表 A - 3 を参照）。
20〜21	unsigned short	ターゲット ID。マジック・ナンバー（092h）は、ファイルが TMS320C2x/C2xx/C5x システムで実行可能なことを示す。

表 A - 3 は、ファイル・ヘッダーのバイト 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 - 3. ファイル・ヘッダーのフラグ (バイト 18〜19)

ニーモニック	フラグ	説明
F_10	00h	このファイルは TMS320C1x デバイス用に作成されている。
F_20	010h	このファイルは TMS320C2x デバイス用に作成されている。
F_25	020h	このファイルは TMS320C2x/C5x デバイス用に作成されている。
F_RELFLG	0001h	再配置情報がファイルから除去された。
F_EXEC	0002h	ファイルが再配置可能である（未解決の外部参照が含まれていない）。
F_LNNO	0004h	行番号がファイルから除去された。
F_LSYMS	0008h	ローカル・シンボルがファイルから除去された。
F_LENDIAN	0100h	ファイルは、TMS320C1x、TMS320C2x、TMS320C2xx、および TMS320C5x デバイスが使用するバイト配列を備えている（1 ワード当り 16 ビットで最下位バイトが最初に配置されている）。

A.3 オプション・ファイル・ヘッダーのフォーマット

リンカは、オプション・ファイル・ヘッダーを作成して、ダウンロード時にそのヘッダー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・ヘッダーは組み込まれません。表 A - 4 は、オプション・ファイル・ヘッダーのフォーマットを示したものです。

表 A - 4. オプション・ファイル・ヘッダーの内容

バイト位置	型	説明
0〜1	short	マジック・ナンバ (0108h)
2〜3	short	バージョン・スタンプ
4〜7	long int	実行可能なコードのサイズ (ワード数)
8〜11	long int	初期化されたデータのサイズ (ワード数)
12〜15	long int	初期化されないデータのサイズ
16〜19	long int	エントリ・ポイント
20〜23	long int	実行可能なコードの開始アドレス
24〜27	long int	初期化されたデータの開始アドレス

A.4 セクション・ヘッダーの構造

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・ヘッダーのテーブルが含まれています。各セクションには、そのセクション独自のセクション・ヘッダーがあります。

表 A - 5. セクション・ヘッダーの内容

バイト位置	型	説明
0〜7	char	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る)
8〜11	long int	セクションの物理アドレス
12〜15	long int	セクションの仮想アドレス
16〜19	long int	セクションのサイズ (ワード数)
20〜23	long int	生データへのファイル・ポインタ
24〜27	long int	再配置エントリへのファイル・ポインタ
28〜31	long int	行番号エントリへのファイル・ポインタ
32〜33	unsigned short	再配置エントリの数
34〜35	unsigned short	行番号エントリの数
36〜37	unsigned short	フラグ (表 A - 6 を参照)
38	char	予約済
39	unsigned char	メモリ・ページ番号

セクション・ヘッダーのバイト 36〜37 に格納されるフラグを表 A - 6 に示します。

表 A - 6. セクション・ヘッダーのフラグ (バイト 36〜37)

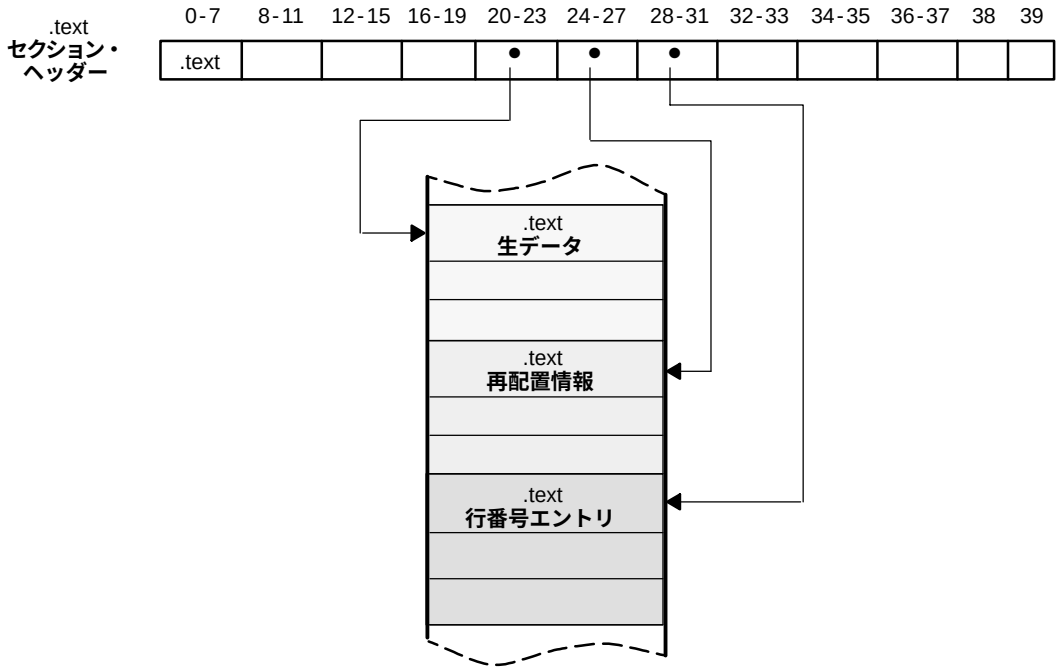
ニーモニック	フラグ	説明
STYP_REG	0000h	通常セクション (割り振り、再配置、およびロードが行われます。)
STYP_DSECT	0001h	ダミー・セクション (再配置が行われ、割り振りおよびロードは行われません。)
STYP_NOLOAD	0002h	非ロード・セクション (割り振りおよび再配置が行われ、ロードは行われません。)
STYP_GROUP	0004h	グループ・セクション (いくつかの入力セクションから構成されます。)
STYP_PAD	0008h	埋め込みセクション（ロードされるだけで、割り振りおよび再配置は行なわれません）
STYP_COPY	0010h	コピー・セクション (再配置およびロードが行われ、割り振りは行われません。再配置エントリおよび行番号エントリは普通に処理されます。)
STYP_TEXT	0020h	実行可能なコードが含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_ALIGN	0700h	ページの境界で位置合わせされるセクション

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。

表 A - 6 の各フラグは、他のフラグと組み合わせて設定されることがあります。たとえば、フラグ・ワードが 024h に設定された場合は、STYP_GROUP と STYP_TEXT が両方とも設定されていることを示します。

図 A - 3 では、セクション・ヘッダー内のポインタが、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し示すかを示します。

図 A - 3.　.text **セクションに関するセクション・ヘッダーのポインタ**



A - 3 **ページの図** A - 2 **で示したように、(**.bss **および** .usect **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・ヘッダーはありますが、生データ、再配置情報、および行番号情報はありません。初期化されないセクションが、オブジェクト・ファイル内で実際に空間を占め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、行番号エントリの数およびファイル・ポインタが** 0 **に設定されます。初期化されないセクションのヘッダーは、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働きしか持ちません。**

A.5 再配置情報の構造

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。この再配置エントリは、アセンブラが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。

バージョン 0 のCOFF のファイル再配置情報エントリでは、表 A - 7 に示す 10 バイトのフォーマットを使用します。バージョン1のCOFFのファイル再配置情報エントリでは、表 A - 8 に示す 12 バイトのフォーマットを使用します。

表 A - 7. COFF バージョン 0 の再配置エントリの内容

バイト位置	型	説明
0〜3	long int	参照の仮想アドレス
4〜5	unsigned short	シンボル・テーブルのインデックス (0 - 65535)
6〜7	unsigned short	予約済み
8〜9	unsigned short	再配置タイプ (表 A - 9 を参照)

表 A - 8. COFF バージョン 1 の再配置エントリの内容

バイト位置	型	説明
0〜3	long int	参照の仮想アドレス
4〜7	unsigned long int	シンボル・テーブルのインデックス
8〜9	unsigned short	予約済
10〜11	unsigned short	再配置タイプ（表 A - 9 を参照）

仮想アドレスとは、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。このアドレスは、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以下に示します。

```
0002                        .global   X
0003   0000   FF80          B         X
       0001   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とは、参照されるシンボルのインデックスです。前出の例では、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ることになります。セクション内におけるシンボルの現行のアドレスとそのシンボルのアセンブリ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、参照している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 0 という値を持ちます。この 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 が .data セクションで定義されており、.data が 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、その再配置は、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したものです。このタイプ・フィールドは、再配置可能な参照を生成するときに使用したアドレッシング・モードによって異なります。前出の例では、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。これは 16 ビットの直接参照なので、再配置タイプは R_RELWORD となります。表 A - 9 に、再配置タイプを示します。

表 A - 9. 再配置タイプ (バイト 8〜9)

ニーモニック	フラグ	再配置タイプ
R_ABS	0000h	再配置なし
R_RELBYTE	000Fh	シンボルのアドレスへの 8 ビット直接参照
R_REL	002Ah	13 ビット直接参照
R_RELWORD	0010h	シンボルのアドレスへの 16 ビット直接参照
R_PARTLS7	0028h	アドレスの 7 LSB
R_PARTMS9	0029h	アドレスの 9 MSB

A.6 行番号テーブルの構造

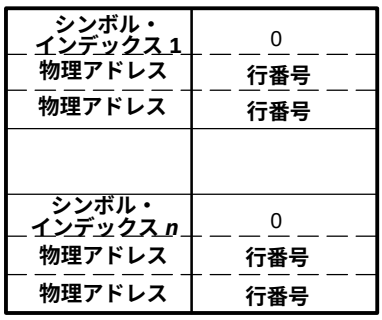
オブジェクト・ファイルには、シンボリック・デバッグに役立つ行番号エントリのテーブルが格納されます。C コンパイラはアセンブリ言語コードを何行か生成する場合は、行番号エントリを作成します。この行番号エントリは、各行のアセンブリ言語コードを、その行を生成した C ソース・コードの元の行にマップします。それぞれの行番号エントリには、6 バイトの情報が含まれています。表 A - 10 は、行番号エントリのフォーマットを示したものです。

表 A - 10. 行番号エントリのフォーマット

バイト位置	型	説明
0〜3	long int	このエントリは、以下の 2 つの値のいずれかを持ちます。 1) 行番号エントリのブロック内の最初のエントリであれば、シンボル・テーブル内のシンボル・エントリを指す値。 2) ブロック内の最初のエントリでない場合は、バイト 4〜5 で示される行の物理アドレス。
4〜5	unsigned short	このエントリは、以下の 2 つの値のいずれかを持ちます。 1) このフィールドが 0 であれば、関数エントリの最初の行であることを示します。 2) このフィールドが 0 でない場合は、C ソース・コードの行番号を示します。

図 A - 4 は、それぞれの行番号エントリがどのようにグループ化されてブロックになるかを示したものです。

図 A - 4. 行番号のブロック



図A - 4 に示すように、それぞれのエントリは以下の 2 つに分けられます。

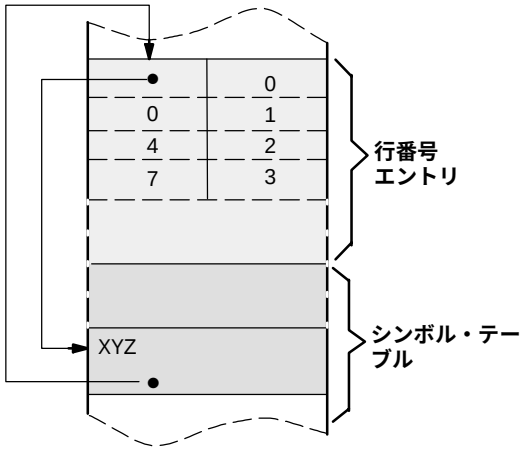
- ある関数の最初の行である場合、バイト 0〜3 はシンボル・テーブル内のシンボル、つまり関数の名前を指し、バイト 4〜5 にはブロックの開始であることを示す0 が入ります。

- ❑ 関数の以降の行である場合、バイト 0〜3 は物理アドレス (C ソースの 1 行で作成されたワード数) を示し、バイト 4〜5 は C ソース・プログラム内での出現箇所を基準とした元の C ソースの相対アドレスを示します。

行エントリ・テーブルには、これらのブロックが多数格納されることがあります。

図 A - 5 は、XYZ という名前の関数の行番号エントリを示したものです。この図にあるように、関数名はシンボル・テーブルにシンボルとして入れられます。行番号エントリの XYZ のブロックの先頭部分は、シンボル・テーブル内の関数名を指します。C ソース内の元の関数に、3 行のコードが含まれているとします。このコードの最初の行では 4 ワードのアセンブリ言語コードが生成され、2 番目の行では 3 ワード、3 番目の行では 10 ワードのアセンブリ言語コードがそれぞれ作成されます。

図 A - 5. 行番号エントリ



(XYZ に関するシンボル・テーブル・エントリには、逆に行番号ブロックの先頭を指すフィールドがあることに注意してください。)

行番号が必要になることはそれほど多くないので、リンカにはオブジェクト・ファイルから行番号情報を除去するオプション (- s) が用意されています。このオプションを使用すると、オブジェクト・モジュールのサイズを小さくできます。

A.7 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 - 6 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 - 6. シンボル・テーブルの内容

静的変数とは、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 を持つものを指します。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、そのシンボルに関する次のような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
- ❑ 基本型 (整数、文字など)
- ❑ 派生型 (配列、構造体など)
- ❑ 次元
- ❑ シンボルを定義したソース・コードの行番号

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・エントリは、クラスおよび型とは無関係に、シンボル・テーブル内で同じフォーマットを持ちます。シンボル・テーブルの各エントリには、表 A - 11 に示す 18 バイトの情報が含まれます。それぞれのシンボルが 18 バイトの補足エントリを持つことがあります。A-16 ページの表 A - 12 に示す特殊シンボルは、必ず補足エントリを持ちます。シンボルの中には、前記の特性を全部は持たないものもあります。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 - 11. シンボル・テーブル・エントリの内容

バイト位置	型	説明
0〜7	char	このフィールドには以下のいずれかがか格納されます。 1) 8 文字のシンボル名 (空き部分にはヌルが埋め込まれます) 2) 文字列テーブルへのオフセット (シンボル名が 8 文字より長い場合)
8〜11	long int	シンボルの値 (記憶クラスにより異なります)
12〜13	short	シンボルのセクション番号
14〜15	unsigned short	基本型および派生型の指定
16	char	シンボルの記憶クラス
17	char	補足エントリの数 (常に 0 か 1)

A.7.1 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。これらのシンボルを表 A - 12 に示します。

表 A - 12. 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
.bb	ブロックの開始のアドレス
.eb	ブロックの終了のアドレス
.bf	関数の開始のアドレス
.ef	関数の終了のアドレス
.target	関数が返す構造体または共用体へのポインタ
.nfake	構造体、共用体、および列挙型のダミー・タグ名
.eos	構造体、共用体、または列挙型の終了
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

これらのシンボルの中には、次のように 2 つ 1 組で使用されるものがあります。

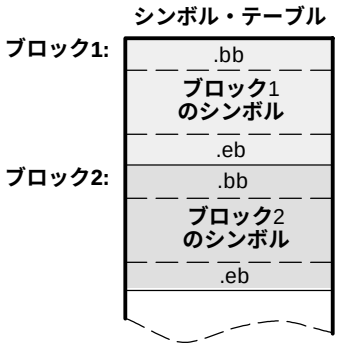
- .bb/.eb は、あるブロックの開始および終了を示します。
- .bf/.ef は、ある関数の開始および終了を示します。
- *n*fake/.eos は、名前が割り当てられていない構造体、共用体、および列挙型の境界に名前を割り当てて定義します。.eos シンボルは、名前付きの構造体、共用体、および列挙型とも組で使用されます。

構造体、共用体、または列挙型がタグ名を持たない場合は、それらをシンボル・テーブルに入力できるように、コンパイラが名前を割り当てます。この名前は、*n*fake (*n* は整数)という形式で指定します。コンパイラは、これらのシンボル名に 0 から番号を付けます。

シンボルとブロック

C におけるブロックは、中括弧で開始し終了する複合文です。ブロックには、必ずシンボルが含まれています。ある特定のブロック用のシンボル定義は、シンボル・テーブル内でグループ化され、.bb/.eb 特殊シンボルで囲んで表現されます。C ではブロックがネストされるので、ブロックのシンボル・テーブル・エントリもそれに応じてネストされることがあります。図 A - 7 は、ブロックのシンボルがシンボル・テーブル内でどのようにグループ化されるかを示したものです。

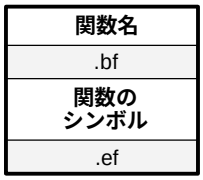
図 A - 7.　ブロックに関するシンボル



シンボルと関数

ある関数に関するシンボル定義は、シンボル・テーブル内でグループ化され、.bf/.ef 特殊シンボルで囲んで表現されます。関数名に関するシンボル・テーブル・エントリの後に .bf 特殊シンボルが置かれます。図 A - 8 は、関数についてのシンボル・テーブル・エントリのフォーマットを示したものです。

図 A - 8.　関数に関するシンボル



関数が構造体または共用体を返す場合は、関数名と .bf 特殊シンボルのエントリの間に、特殊シンボル .target に関するシンボル・テーブル・エントリが現れます。

A.7.2 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、このフィールドは文字 (char) 型になります。この名前は (必要があれば) 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.7.3 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。通常はシンボルの名前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、名前の代わりに、文字列テーブル内にあるシンボルの名前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。

図 A - 9 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たものです。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 になります。

図 A - 9. 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.7.4 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 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 - 13 は、有効な記憶クラスを示したものです。

表 A - 13. 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	共用体タグ
C_AUTO	1	自動変数	C_TPDEF	13	型定義
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	列挙型タグ
C_REG	4	レジスタ変数	C_MOE	16	列挙型のメンバ
C_EXTDEF	5	外部定義	C_REGPARM	17	レジスタ・パラメータ
C_LABEL	6	ラベル	C_FIELD	18	ビット・フィールド
C_ULABEL	7	未定義ラベル	C_BLOCK	100	ブロックの開始または終了 (.bb および .eb 特殊シンボルのみで使用される)
C_MOS	8	構造体のメンバ	C_FCN	101	関数の開始または終了(.bf および .ef 特殊シンボルのみで使用される)
C_ARG	9	関数引数	C_EOS	102	構造体の終了 (.eos 特殊シンボルのみで使用される)
C_STRTAG	10	構造体タグ	C_FILE	103	ファイル名 (.file 特殊シンボルのみで使用される)
C_MOU	11	共用体のメンバ	C_LINE	104	ユーティリティ・プログラムのみで使用

特殊シンボルのなかには、特定の記憶クラスに制限されているものもあります。これらのシンボルとその記憶クラスを表 A - 14 に示します。

表 A - 14. 特殊シンボルとその記憶クラス

特殊シンボル	記憶クラス	特殊シンボル	記憶クラス
.file	C_FILE	.eos	C_EOS
.bb	C_BLOCK	.text	C_STAT
.eb	C_BLOCK	.data	C_STAT
.bf	C_FCN	.bss	C_STAT
.ef	C_FCN		

A.7.5 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。シンボルの値は、そのシンボルの記憶クラスによって異なります。表 A - 15 は、記憶クラスと関連する値をまとめたものです。

表 A - 15. シンボルの値と記憶クラス

記憶クラス	値の説明	記憶クラス	値の説明
C_AUTO	スタック・オフセット (ビット数)	C_UNTAG	0
C_EXT	再配置可能アドレス	C_TPDEF	0
C_STAT	再配置可能アドレス	C_ENTAG	0
C_REG	レジスタ番号	C_MOE	列挙型の値
C_LABEL	再配置可能アドレス	C_REGPARM	レジスタ番号
C_MOS	オフセット (ビット数)	C_FIELD	ビット変位
C_ARG	スタック・オフセット (ビット数)	C_BLOCK	再配置可能アドレス
C_STRTAG	0	C_FCN	再配置可能アドレス
C_MOU	オフセット (ビット数)	C_FILE	0

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になります。このため、.file 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くなった場合、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もそれに応じて変更されます。

A.7.6 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 - 16 に、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ます。

表 A - 16. **セクション番号**

ニーモニック	セクション番号	説明
N_DEBUG	- 2	**特殊なシンボリック・デバッグ用のシンボル**
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の番号**

.text、.data、または .bss **セクションが存在しなかった場合、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、セクション内で定義されていません。セクション番号** - 2 **は、シンボリック・デバッグ用のシンボルを示します。ここには、構造体、共用体、および列挙型のタグ名と、型定義、およびファイル名が含まれます。セクション番号** - 1 **は、値を持っているが、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.7.7 型エントリ

シンボル・テーブル・エントリのバイト 14〜15 **は、シンボルの型を定義しています。それぞれのシンボルには、**1 **つの基本型と** 1〜6 **つの派生型があります。**

型を示すこの 16 **ビットのエントリのフォーマットを以下に示します。**

	派生型 6	派生型 5	派生型 4	派生型 3	派生型 2	派生型 1	基本型
サイズ (ビット数):	2	2	2	2	2	2	4

型フィールドのビット 0〜3 **は、基本型を示します。表** A - 17 **に、有効な基本型を示します。**

表 A - 17. 基本型

ニーモニック	値	型
T_VOID	0	void
T_CHAR	2	char
T_SHORT	3	short
T_INT	4	int
T_LONG	5	long int
T_FLOAT	6	float
T_DOUBLE	7	double
T_STRUCT	8	struct（構造体）
T_UNION	9	union（共用体）
T_ENUM	10	enum（列挙型）
T_MOE	11	列挙型のメンバ
T_UCHAR	12	unsigned char
T_USHORT	13	unsigned short

型フィールドのビット 4〜15 は、6 つの 2 ビット・フィールドとして配置され、1〜6 つの派生型を示します。表 A - 18 に、有効な派生型を示します。

表 A - 18. 派生型

ニーモニック	値	型
DT_NON	0	派生型なし
DT_PTR	1	ポインタ
DT_FCN	2	関数
DT_ARY	3	配列

たとえば、いくつかの派生型を持つシンボルについて 0000000011010011_2、という型を示すエントリがあるとします。このエントリは、そのシンボルが short の配列へのポインタであることを示します。

A.7.8 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) を持ちますが、補足エントリのフォーマットは、シンボルの型と記憶クラスによって異なります。表 A - 19 は、これらの関係をまとめたものです。

表 A - 19. シンボル・テーブル補足エントリのフォーマット

名前	記憶クラス	型エントリ		補足エントリのフォーマット
		派生型 1	基本型	
.file	C_FILE	DT_NON	T_NULL	ファイル名 (表 A - 20 を参照)
.text、.data、.bss	C_STAT	DT_NON	T_NULL	セクション (表 A - 21 を参照)
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	タグ名 (表 A - 22 を参照)
.eos	C_EOS	DT_NON	T_NULL	構造体の終了 (表 A - 23 を参照)
fcname	C_EXT C_STAT	DT_FCN	(注1を参照)	関数 (表 A - 24 を参照)
arrname	(注 2 を参照)	DT_ARY	(注1を参照)	配列 (表 A - 25 を参照)
.bb、.eb	C_BLOCK	DT_NON	T_VOID	ブロックの開始および終了(表 A - 26 および表 A - 27 を参照)
.bf、.ef	C_FCN	DT_NON	T_VOID	関数の開始および終了 (表 A - 26 および表 A - 27 を参照)
構造体、共用体、または列挙型に関連する名前	(注 2 を参照)	DT_PTR DT_ARR DT_NON	T_STRUCT T_UNION T_ENUM	構造体、共用体、または列挙型に関連する名前 (表 A - 28 を参照)

注: 1) T_MOE を除く任意の型
2) C_AUTO、C_STAT、C_MOS、C_MOU、C_TPDEF

表 A - 19 において、tagname は、(特殊シンボル *n*fake を含めた) 任意のシンボル名を示します。fcname および arrname は、任意のシンボル名を示します。

表 A - 19 の複数の条件を満たすシンボルは、共用体フォーマットの補足エントリを持ちます。これらの条件をどれも満たさないシンボルは、補足エントリを持ちません。

ファイル名

ファイル名に関するそれぞれのテーブル補足エントリには、バイト 0〜13 に 14 文字のファイル名が格納されます。バイト 14〜17 は使用されません。

表 A - 20. テーブル補足エントリ用のファイル名フォーマット

バイト位置	型	説明
0〜13	char	ファイル名
14〜17	—	未使用

セクション

表 A - 21 は、テーブル補足エントリのフォーマットを示したものです。

表 A - 21. テーブル補足エントリ用のセクション・フォーマット

バイト位置	型	説明
0〜3	long	セクションの長さ
4〜6	unsigned short	再配置エントリの数
7〜8	unsigned short	行番号エントリの数
9〜17	—	未使用 (ゼロが埋め込まれる)

タグ名

表 A - 22 は、タグ名に関するテーブル補足エントリのフォーマットを示したものです。

表 A - 22. テーブル補足エントリ用のタグ名フォーマット

バイト位置	型	説明
0〜5	—	未使用 (ゼロが埋め込まれる)
6〜7	unsigned short	構造体、共用体、または列挙型のサイズ
8〜11	—	未使用 (ゼロが埋め込まれる)
12〜15	long	この構造体、共用体、または列挙型の範囲外にある次のエントリのインデックス
16〜17	—	未使用 (ゼロが埋め込まれる)

構造体の終了

表 A - 23 は、構造体の終了に関するテーブル補足エントリのフォーマットを示したものです。

表 A - 23. テーブル補足エントリ用の構造体の終了フォーマット

バイト位置	型	説明
0〜3	long	タグ・インデックス
4〜5	—	未使用 (ゼロが埋め込まれる)
6〜7	unsigned short	構造体、共用体、または列挙型のサイズ
8〜17	—	未使用 (ゼロが埋め込まれる)

関数

表 A - 24 は、関数に関するテーブル補足エントリのフォーマットを示したものです。

表 A - 24. テーブル補足エントリ用の関数フォーマット

バイト位置	型	説明
0〜3	long	タグ・インデックス
4〜7	long	関数のサイズ (ビット数)
8〜11	long	行番号へのファイル・ポインタ
12〜15	long	この関数の範囲外にある次のエントリのインデックス
16〜17	—	未使用 (ゼロが埋め込まれる)

配列

表 A - 25 は、配列に関するテーブル補足エントリのフォーマットを示したものです。

表 A - 25. テーブル補足エントリ用の配列フォーマット

バイト位置	型	説明
0〜3	long	タグ・インデックス
4〜5	unsigned short	行番号宣言
6〜7	unsigned short	配列のサイズ
8〜9	unsigned short	1 次元
10〜11	unsigned short	2 次元
12〜13	unsigned short	3 次元
14〜15	unsigned short	4 次元
16〜17	—	未使用 (ゼロが埋め込まれる)

ブロックおよび関数の終了

表 A - 26 は、ブロックおよび関数の終了に関するテーブル補足エントリのフォーマットを示したものです。

表 A - 26. テーブル補足エントリ用のブロックおよび関数の終了フォーマット

バイト位置	型	説明
0〜3	—	未使用 (ゼロが埋め込まれる)
4〜5	unsigned short	C ソースの行番号
6〜17	—	未使用 (ゼロが埋め込まれる)

ブロックおよび関数の開始

表 A - 27 は、ブロックおよび関数の開始に関するテーブル補足エントリのフォーマットを示したものです。

表 A - 27. テーブル補足エントリ用のブロックおよび関数の開始フォーマット

バイト位置	型	説明
0〜3	—	未使用 (ゼロが埋め込まれる)
4〜5	unsigned short	ブロック開始の C ソース行番号
6〜11	—	未使用 (ゼロが埋め込まれる)
12〜15	long int	このブロックの範囲外にある次のエントリのインデックス
16〜17	—	未使用 (ゼロが埋め込まれる)

構造体、共用体、および列挙型に関連する名前

表 A - 28 は、構造体、共用体、および列挙型の名前に関するテーブル補足エントリのフォーマットを示したものです。

表 A - 28. テーブル補足エントリ用の構造体、共用体、および列挙型の名前フォーマット

バイト位置	型	説明
0〜3	long int	タグ・インデックス
4〜5	—	未使用 (ゼロが埋め込まれる)
6〜7	unsigned short	構造体、共用体、または列挙型のサイズ
8〜17	—	未使用 (ゼロが埋め込まれる)

付録 B

シンボリック・デバッグ疑似命令

TMS320C1x/C2x/C2xx/C5x アセンブラは、TMS320C2x/C2xx/C5x C コンパイラがシンボリック・デバッグのために使用するいくつかの疑似命令をサポートしています。

- **.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シンボル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** 疑似命令は、C 関数の最初と最後の行を指定します。
- **.block** および **.endblock** 疑似命令は、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 ソース文の行番号を識別します。

これらのシンボリック・デバッグ疑似命令は、通常、コンパイラで作成するアセンブリ言語ファイルには出力されません。これらの疑似命令を出力したい場合は、次のように -o オプションを指定してコード生成プログラムを起動して下さい。

```
dspcg -o input file
```

この付録には、シンボリック・デバッグ疑似命令についての説明がアルファベット順に記載されています。.file 疑似命令以外は、それぞれの疑似命令について、C ソースの例と結果のアセンブリ言語コードを示してあります。

.block / .endblock　ブロックの定義

構文

.block [*beginning line number*]
.endblock [*ending line number*]

説明

.block および **.endblock** 疑似命令は、C ブロックの開始および終了を指定します。line number は指定が任意で、ソース・ファイル内でブロックが定義されている場所を指定します。

ブロック定義はネストできます。誤ったブロックのネストは、アセンブラで検出されます。

例

ブロックを定義する C ソースの例と、結果のアセンブリ言語コードを以下に示します。

C ソース:

```
      .
      .
      .
{                    /*  Beginning of a block      */
     int  a,b;
     a = b;
}                    /*  End of a block        */
      .
      .
      .
```

結果のアセンブリ言語コード:

```
.block  7
.sym    _a,2,4,1,16
.sym    _b,3,4,1,16
.line   7
LARK    AR2,3
MAR     *0+
LAC     *-
SACL    * ,AR1
.endblock       7
```

構文

```
.file  "filename"
```

説明

.file 疑似命令を使用すると、デバッガでメモリ内のロケーションを逆に C ソース・ファイルの行にマップすることができます。*filename* は、オリジナルの C ソース・プログラムが含まれているファイルの名前を示します。ファイル名の最初の 14 文字が意味を持ちます。

アセンブリ・コードで .file 疑似命令を使用して名前を指定してプログラムを読みやすくすることもできます。

例

以下の例では、text.c という名前のファイルにこの疑似命令を生成したCソースが含まれています。

```
.file     "text.c"
```

構文

.func [*beginning line number*]

.endfunc [*ending line number*]

説明

.func および .endfunc 疑似命令は、C 関数の開始および終了を指定します。line number は指定が任意で、ソース・ファイル内で関数が定義されている場所を指定します。

関数定義はネストできません。

例

関数を定義する C ソースと、結果のアセンブリ言語コードの例を以下に示します。

C ソース:

```
power(x, n)          /*  Beginning of a function   */
int x,n;
{
     int i, p;
     p = 1;
     for (i = 1; i <= n; ++i)
         p = p * x;
     return p;       /*  End of function           */
}
```

結果のアセンブリ言語コード:

```
        .sym    _power,_power,36,2,0
        .globl  _power

        .func   1
*****************************************************
* FUNCTION DEF : _power
*****************************************************
_power:

LF1     .set    0

        POPD    *+
        SAR     AR0,*+
        SAR     AR1,*
        LARK    AR0,3
        LAR     AR0,*0+,AR2

        .sym    _x,-3+LF1,4,9,16
        .sym    _n,-4+LF1,4,9,16
        .sym    _i,1,4,1,16
        .sym    _p,2,4,1,16
        .line   3
        .line   5
        LACK    1
        LARK    AR2,2
        MAR     *0+
        SACL    *-
        .line   6
        SACL    *
        SSXM
        LAC     *
        SBRK    5-LF1
        SUB     *
        BGZ     L2
```

```
        MAR     *+
L1:
        .line   7
        LT      *
        ADRK    5-LF1
        MPY     *
        SPL     *-
        .line   6
        LAC     *
        ADDK    1
        SACL    *
        LAC     *
        SBRK    5-LF1
        SUB     *+
        BLEZ    L1
L2:
        .line   8
        LARK    AR2,2
        MAR     *0+
        LAC     *
EPI0_1:
        .line   9
        MAR     * ,AR1
        SBRK    4
        LAR     AR0,*-
        PSHD    *
        RET

        .endfunc        9
```

構文

.line *line number* [, *address*]

説明

.line 疑似命令は、オブジェクト・ファイル内に行番号エントリを作成します。行番号エントリは、シンボリック・デバッグで使われ、オブジェクト・コードのアドレスとそのコードを生成したソース・コードの行とを関連付けます。

.line 疑似命令には、以下の 2 つのオペランドがあります。

- *line number* は、コードの部分を生成した C ソースの行を示します。行番号は、現行の関数の開始からの相対的な行番号です。これは必須パラメータです。
- *address* は、行番号に関連付けられているアドレスを表す式です。これは指定が任意のパラメータです。アドレスを指定しなかった場合、アセンブラは現行のSPC値を使用します。

例

.line 疑似命令の後には、指定した C ソース行によって生成されるアセンブリ言語ソース文が現れます。たとえば、以下の C ソース行がオリジナルの C ソースの行 5〜7 であるとすれば、行 5〜7 によって以下に示すアセンブリ言語文が生成されます。

C ソース:

```
register int p=1;
for (i=1;i<=n;++i)
     p = p * x;
```

結果のアセンブリ言語コード:

```
        .line   5
        LACK    1
        SACL    * ,AR2
        .line   6
        LARK    AR2,1
        MAR     *0+
        SACL    *
        SSXM
        LAC     *
        SBRK    5-LF1
        SUB     *
        BGZ     L2
        MAR     *+,AR6
L1:
        .line   7
        LT      * ,AR2
        MPY     * ,AR6
        SPL     * ,AR2
        .line   6
        ADRK    4-LF1
        LAC     *
        ADDK    1
        SACL    *
        LAC     *
        SBRK    5-LF1
        SUB     *+,AR6
        BLEZ    L1
L2:
```

構文

.member *name*, *value* [, *type*, *storage class*, *size*, *tag*, *dims*]

説明

.member 疑似命令は、構造体、共用体、または列挙型のメンバを定義します。この疑似命令は、構造体、共用体、または列挙型の定義内で使用された場合にのみ有効です。

- *name* は、シンボル・テーブルに入れるメンバの名前です。名前の最初の 32 文字が意味を持ちます。
- *value* は、メンバに関連付ける値です。正しい任意の式 (絶対または再配置可能) を使用できます。
- *type* は、メンバの C での型です。C の型についての詳しい説明は、付録 A に記載されています。
- *storage class* は、メンバの C での記憶クラスです。C の記憶クラスについての詳しい説明は、付録 A に記載されています。
- *size* は、このメンバを格納するのに必要なメモリのビット数です。
- *tag* は、このメンバが属する型 (型がある場合) または構造体の名前です。この名前は、.stag、.etag、または .utag 疑似命令によって**必ず**事前に定義しなければなりません。
- *dims* には、1〜4 つの式をコンマで区切って指定します。これによって、最大 4 次元までのメンバを指定することができます。

パラメータの順序は意味を持ちます。*name* および *value* は必須パラメータです。それ以外のパラメータは、省略したり空の項目として指定することができます。(コンマを連続して入力すると、空の項目になります)。空の項目を使用すると、あるパラメータをスキップして、リスト内のその後のパラメータを指定することができます。省略したオペランドや空のオペランドは、ヌル値とみなされます。

例

C の構造体定義と、それに対応するアセンブリ言語文の例を以下に示します。

C ソース:

```
struct doc {
     char title;
     char group;
     int  job_number;
} doc_info;
```

結果のアセンブリ言語コード:

```
     .stag   _doc,48
     .member _title,0,2,8,16
     .member _group,16,2,8,16
     .member _job_number,32,4,8,16
     .eos
```

.stag/.etag/.utag/.eos　構造体の定義

構文

```
.stag  name [, size]
    member definitions
.eos
.etag  name [, size]
    member definitions
.eos
.utag  name [, size]
    member definitions
.eos
```

説明

.stag 疑似命令は、構造体定義を開始します。.etag 疑似命令は、列挙型定義を開始します。.utag 疑似命令は、共用体定義を開始します。.eos 疑似命令は、構造体定義、列挙型定義、または共用体定義を終了させます。

- *name* **は、構造体、列挙型、または共用体の名前です。名前の最初の** 32 **文字が意味を持ちます。これは必須パラメータです。**
- *size* **は、構造体、列挙型、または共用体がメモリ内に占めるビット数です。これは指定が任意のパラメータです。省略した場合、サイズは未指定になります。**

.stag**、**.etag**、または** .utag **疑似命令の後に、いくつかの** .member **疑似命令を指定して、構造体のメンバを定義する必要があります。**.member **疑似命令は、構造体、列挙型、または共用体の定義の中で使用できる唯一の疑似命令です。**

構造体、列挙型、および共用体はアセンブラではネストできません。C **コンパイラでは、ネストした構造体をそれぞれ個別に定義し、ネストした構造体を参照している構造体からその構造体を参照することによって、ネストした構造体を展開します。**

例 1

構造体定義の例を以下に示します。

C ソース:

```
struct doc
{
    char title;
    char group;
    int  job_number;
} doc_info;
```

結果のアセンブリ言語コード:

```
     .stag   _doc,48
     .member _title,0,2,8,16
     .member _group,16,2,8,16
     .member _job_number,32,4,8,16
     .eos
```

例 2　　共用体定義の例を以下に示します。

C ソース:

```
union u_tag {
      int   val1;
      float val2;
      char  valc;
}  valu;
```

結果のアセンブリ言語コード:

```
     .utag   _u_tag,32
     .member _val1,0,4,11,16
     .member _val2,0,6,11,32
     .member _valc,0,2,11,16
     .eos
```

例 3　　列挙型定義の例を以下に示します。

Cソース:

```
{
    enum o_ty { reg_1, reg_2, result } optypes;
}
```

結果のアセンブリ言語コード:

```
     .etag   _o_ty,16
     .member _reg_1,0,4,16,16
     .member _reg_2,1,4,16,16
     .member _result,2,4,16,16
     .eos
```

構文

.sym *name*, *value* [, *type*, *storage class*, *size*, *tag*, *dims*]

説明

.sym 疑似命令は、グローバル変数、ローカル変数、または関数についてのシンボリック・デバッグ情報を指定します。

- *name* **は、オブジェクト・シンボル・テーブルに入れる変数の名前です。名前の最初の** 32 **文字が意味を持ちます。**
- *value* **は、変数に関連付ける値です。正しい任意の式 (絶対または再配置可能) を使用できます。**
- *type* **は、変数の** C **での型です。**C **の型についての詳しい説明は、付録** A **に記載されています。**
- *storage class***は、変数の** C **での記憶クラスです。**C **の記憶クラスについての詳しい説明は、付録** A **に記載されています。**
- *size* **は、この変数を格納するのに必要なメモリのビット数です。**
- *tag* **は、この変数が属する型 (型がある場合) または構造体の名前です。この名前は、**.stag**、**.etag**、または** .utag **疑似命令によって事前に定義しなければなりません。**
- *dims* **には、**1**～**4 **つの式をコンマで区切って指定します。これによって、最大** 4 **次元までの変数を指定することができます。**

パラメータの順序は意味を持ちます。*name* **および** *value* **は必須パラメータです。それ以外のパラメータは、省略したり空の項目として指定することができます。(コンマを連続して入力すると、空の項目になります)。空の項目を使用すると、あるパラメータをスキップして、リスト内のその後のパラメータを指定することができます。省略したオペランドや空のオペランドは、ヌル値とみなされます。**

例

以下の C **ソース行によって、次に示す** .sym **疑似命令が生成されます。**

C ソース:

```
struct s { int member1, member2; } str;
int   ext;
int   array[5][10];
long  *ptr;
int   strcmp();

main(arg1,arg2)
   int   arg1;
   char  *arg2;
{
   register r1;
}
```

結果のアセンブリ言語コード:

```
LF1     .set    0
        .sym    _str,_str,8,2,32,_s
        .sym    _ext,_ext,4,2,16
        .sym    _array,_array,244,2,800,,5,10
        .sym    _ptr,_ptr,21,2,16
        .sym    _main,_main,36,2,0
        .sym    _arg1,-3+LF1,4,9,16
        .sym    _arg2,-4+LF1,18,9,16
        .sym    _r1,1,4,1,16
```

付録 C

リンカ・コマンド・ファイルの例

この付録には、TMS320C10、TMS320C25、TMS320C50、および TMS320C51 のリンカ・コマンド・ファイルの例を収めてあります。この付録に示す例では、MEMORY 疑似命令を使って、TMS320C1x User's Guide、TMS320C2x User's Guide、TMS320C5x ユーザーズ・マニュアル で説明されている標準的なメモリ・マップに従ってメモリを定義しています。ここで例に使用している MEMORY 疑似命令と SECTIONS 疑似命令は単なるガイドラインなので、厳密に従う必要はありません。たとえば、別の出力セクション名を使うこともできるし、コードのリンク先として使用しないメモリ（I/O メモリなど）を構成する必要もありません。また、これらの例では、可能な場合は出力セクションをオンチップ・メモリ空間に割り振るようになっていますが、セクションが大きすぎてユーザのオンチップ・メモリ空間に入らない場合は、外部メモリに割り振ってもかまいません。

この付録に示す例は以下のとおりです。

項目 ページ

C.1 TMS320C10 のリンカ・コマンド・ファイル

例 C - 1. **マイクロコンピュータ・モードの** TMS320C10

```
/*****************************************************************/
/*   MC/MP- = 1 (microcomputer mode). Note that program memory     */
/*   addresses 05F4h--05FFh are not configured.                    */
/*****************************************************************/
MEMORY
{
    PAGE 0 :                                    /* Program Memory */
         Int_Prog : origin =    0h , length = 05F4h
         Ext_Prog : origin = 0600h , length = 0A00h
    PAGE 1 :                                      /* Data Memory */
         pg0_Data : origin =    0h , length =  080h
         pg1_Data : origin =  080h , length =  010h
    PAGE 2 :                                       /* I/O Memory */
         ext_IO   : origin =    0h , length =   08h
}
SECTIONS
{
    .text :       > Int_Prog  PAGE = 0
    .data :       > Int_Prog  PAGE = 0
    .bss  :       > pg0_Data  PAGE = 1
}xmps
```

例 C - 2. **マイクロプロセッサ・モードの** TMS320C10

```
/*****************************************************************/
/*                MC/MP- = 0 (microprocessor mode).              */
/*****************************************************************/
MEMORY
{
    PAGE 0 :                                    /* Program Memory */
         Ext_Prog : origin =    0h , length = 01000h
    PAGE 1 :                                      /* Data Memory */
         pg0_Data : origin =    0h , length =   080h
         pg1_Data : origin =  080h , length =   010h
    PAGE 2 :                                       /* I/O Memory */
         ext_IO   : origin =    0h , length =    08h
}
SECTIONS
{
    .text :       > Ext_Prog  PAGE = 0
    .data :       > Ext_Prog  PAGE = 0
    .bss  :       > pg0_Data  PAGE = 1
}
```

C.2 TMS320C25 のリンカ・コマンド・ファイル

例 C - 3. **マイクロプロセッサ・モードでブロック** B0 **がデータ・メモリの** TMS320C25

```
/*****************************************************************/
/*  Block B0 is configured as data memory (CNFP) and MC/MP- = 1    */
/*  (microprocessor mode). Note that data memory locations 6h--5Fh */
/*  and 80h--1FFh are not configured.                              */
/*****************************************************************/
MEMORY
{
    PAGE 0 :                                      /* Program Memory */
         Ints     : origin =    0h , length =   020h
         Ext_Prog : origin =  020h , length = 0FEE0h
    PAGE 1 :                                         /* Data Memory */
         Regs     : origin =    0h , length =    06h
         Block_B2 : origin =  060h , length =   020h
         Int_RAM  : origin = 0200h , length =  0200h    /* B0 & B1 */
         Ext_Data : origin = 0400h , length = 0FC00h
}
SECTIONS
{
    .text :       > Ext_Prog  PAGE = 0
    .data :       > Ext_Prog  PAGE = 0
    .bss  :       > Int_RAM   PAGE = 1
}
```

例 C - 4. **マイクロコンピュータ・モードでブロック** B0 **がデータ・メモリの** TMS320C25

```
/*****************************************************************/
/* Block B0 is defined as data memory (CNFP) and MC/MP- = 0 (micro-*/
/* computer mode). Note: program memory locations 0FB0h--0FFFh &   */
/* data memory locations 6h--5Fh and 80h--1FFh are not configured. */
/*****************************************************************/
MEMORY
{
    PAGE 0 :                                      /* Program Memory */
         Ints     : origin =    0h , length =   020h
         Prog_RAM : origin =  020h , length =  0F90h
         Ext_Prog : origin = 1000h , length = 0F000h
    PAGE 1 :                                         /* Data Memory */
         Regs     : origin =    0h , length =    06h
         Block_B2 : origin =  060h , length =   020h
         Int_RAM  : origin = 0200h , length =  0200h    /* B0 & B1 */
         Ext_Data : origin = 0400h , length = 0FC00h
}
SECTIONS
{
    .text :       > Prog_ROM  PAGE = 0
    .data :       > Prog_ROM  PAGE = 0
    .bss  :       > Int_RAM   PAGE = 1
}
```

例 C - 5. **マイクロプロセッサ・モードでブロック** B0 **がプログラム・メモリの** TMS320C25

```
/*****************************************************************/
/*  Block B0 is defined as program memory (CNFP) and MC/MP- = 1    */
/*  (microprocessor mode). Note that data memory locations 6h--5Fh */
/*  and 80h--2FFh are not configured.                              */
/*****************************************************************/
MEMORY
{
    PAGE 0 :                                    /* Program Memory */
         Ints     : origin =     0h , length =   020h
         Ext_Prog : origin =   020h , length = 0FEE0h
         Block_B0 : origin = 0FF00h , length =  0100h
    PAGE 1 :                                       /* Data Memory */
         Regs     : origin =     0h , length =    06h
         Block_B2 : origin =   060h , length =   020h
         Block_B1 : origin =  0300h , length =  0100h
         Ext_Data : origin =  0400h , length = 0FC00h
}
SECTIONS
{
    .text :       > Block_B0  PAGE = 0
    .data :       > Ext_Prog  PAGE = 0
    .bss  :       > Block_B1  PAGE = 1
}
```

例 C - 6. **マイクロコンピュータ・モードでブロック** B0 **がプログラム・メモリの** TMS320C25

```
/*****************************************************************/
/* Block B0 is defined as program memory (CNFP) and MC/MP- = 0     */
/* microcomputer mode). Program memory locations 0FB0h--0FFFh &    */
/* data memory locations 6h--5Fh and 80h--2FFh are not configured. */
/*****************************************************************/
MEMORY
{
    PAGE 0 :                                    /* Program Memory */
         Ints     : origin =     0h , length =   020h
         Prog_ROM : origin =   020h , length =  0F90h
         Ext_Prog : origin =  1000h , length = 0EF00h
         Block_B0 : origin = 0FF00h , length =  0100h
    PAGE 1 :                                       /* Data Memory */
         Regs     : origin =     0h , length =    06h
         Block_B2 : origin =   060h , length =   020h
         Block_B1 : origin =  0300h , length =  0100h
         Ext_Data : origin =  0400h , length = 0FC00h
}
SECTIONS
{
    .text :       > Prog_ROM  PAGE = 0
    .data :       > Block_B0  PAGE = 0
    .bss  :       > Block_B1  PAGE = 1
}
```

C.3 TMS320C50 のリンカ・コマンド・ファイル

例 C - 7. **マイクロコンピュータ・モードでブロック** B0 **がデータ・メモリの** TMS320C50

```
/*********************************************************************/
/* TMS320C50 in Microcomputer Mode            MC/MP- = 1             */
/* 9K RAM block mapped into program space     RAM, OVLY BITS = 1, 0 */
/* B0 configured as data memory               ST1  - CNF BIT = 0     */
/*********************************************************************/
MEMORY
{
    PAGE 0 :                                        /* Program Memory */
         Prog_ROM     : origin =     0h , length =   800h
         Prog_RAM     : origin =   800h , length =  2400h /* 9k RAM */
         Ext_Prog     : origin =  2C00h , length = 0C400h
    PAGE 1 :                                          /* Data Memory */
         Regs         : origin =     0h , length =    60h
         Scratch_RAM  : origin =    60h , length =    20h /* B2     */
         Int_RAM      : origin =   100h , length =   400h /* B0 & B1*/
         Ext_Data     : origin =   800h , length =  F800h
}
SECTIONS
{
    .text :       > Prog_ROM  PAGE = 0
    .data :       > Prog_ROM  PAGE = 0
    .bss  :       > Int_RAM   PAGE = 1
}
```

例 C - 8. **マイクロプロセッサ・モードでブロック** B0 **がデータ・メモリの** TMS320C50

```
/*********************************************************************/
/* TMS320C50 in Microprocessor Mode           MC/MP = 0              */
/* 9K RAM block mapped into program space     RAM, OVLY BITS = 1, 0 */
/* B0 configured as data memory               ST1  - CNF BIT = 0     */
/*********************************************************************/
MEMORY
{
    PAGE 0 :                                        /* Program Memory */
         Ext_Prog     : origin =     0h , length = 10000h
    PAGE 1 :                                          /* Data Memory */
         Regs         : origin =     0h , length =    60h
         Scratch_RAM  : origin =    60h , length =    20h /* B2     */
         Int_RAM      : origin =   100h , length =   400h /* B0 & B1*/
         Ext_RAM      : origin =   800h , length =  F800h
}
SECTIONS
{
    .text :       > Ext_Prog  PAGE = 0
    .data :       > Ext_Prog  PAGE = 0
    .bss  :       > Int_RAM   PAGE = 1
}
```

例 C - 9. **マイクロコンピュータ・モードでブロック** B0 **がプログラム・メモリの** TMS320C50

```
/*****************************************************************/
/* TMS320C50 in Microcomputer Mode           MC/MP = 1           */
/* 9K RAM block mapped into data space       RAM, OVLY BITS = 0, 1 */
/* B0 configured as program memory           ST1  - CNF BIT = 1    */
/*****************************************************************/
MEMORY
{
    PAGE 0 :                                        /* Program Memory */
         Prog_ROM     : origin =    0h , length =  800h
         Ext_Prog     : origin =  800h , length = F600h
         Int_Prog     : origin = FE00h , length =  200h  /* B0     */
    PAGE 1 :                                          /* Data Memory */
         Regs         : origin =    0h , length =   60h
         Scratch_RAM  : origin =   60h , length =   20h  /* B2     */
         Int_RAM      : origin =  100h , length =  200h  /* B1     */
         Data_RAM     : origin =  800h , length = 2400h  /* 9K RAM */
         Ext_RAM      : origin = 2C00h , length = D400h
}
SECTIONS
{
    .text :       > Prog_ROM  PAGE = 0
    .data :       > Prog_ROM  PAGE = 0
    .bss  :       > Int_RAM   PAGE = 1
}
```

C.4 TMS320C51 のリンカ・コマンド・ファイル

例 C - 10. **マイクロコンピュータ・モードでブロック** B0 **がデータ・メモリの** TMS320C51

```
/*****************************************************************/
/* TMS320C51 in Microprocessor Mode          MC/MP = 0           */
/* 1K RAM block mapped into program space    RAM, OVLY BITS = 1, 0 */
/* B0 configured as data memory              ST1  - CNF BIT = 0    */
/*****************************************************************/
MEMORY
{
    PAGE 0 :                                      /* Program Memory */
         Prog_ROM     : origin =    0h , length = 2000h  /* 8k ROM */
         Prog_RAM     : origin = 2000h , length =  400h  /* 1k RAM */
         Ext_Prog     : origin = 2400h , length = DC00h
    PAGE 1 :                                        /* Data Memory */
         Regs         : origin =    0h , length =   60h
         Scratch_RAM  : origin =   60h , length =   20h  /* B2     */
         Int_RAM      : origin =  100h , length =  400h  /* B0 & B1*/
         Ext_RAM      : origin =  800h , length = F800h
}
SECTIONS
{
    .text :       > Prog_ROM  PAGE = 0
    .data :       > Prog_ROM  PAGE = 0
    .bss  :       > Int_RAM   PAGE = 1
}
```

例 C - 11. **マイクロコンピュータ・モードでブロック** B0 **がプログラム・メモリの** TMS320C51

```
/*****************************************************************/
/* TMS320C51 in Microcomputer Mode           MC/MP = 1           */
/* 1K RAM block mapped into data space       RAM, OVLY BITS = 0, 1 */
/* B0 configured as program memory                               */
/*****************************************************************/
MEMORY
{
    PAGE 0 :                                      /* Program Memory */
         Ext_Prog     : origin =   00h , length = FE00h
         Int_Prog     : origin = FE00h , length =  200h  /* B0     */
    PAGE 1 :                                        /* Data Memory */
         Regs         : origin =    0h , length =   60h
         Scratch_RAM  : origin =   60h , length =   20h  /* B2     */
         Int_RAM      : origin =  100h , length =  200h  /* B1     */
         Data_RAM     : origin =  800h , length =  400h  / 1k RAM  */
         Ext_RAM      : origin = 0C00h , length = F400h
}
SECTIONS
{
    .text :       > Prog_ROM  PAGE = 0
    .data :       > Prog_ROM  PAGE = 0
    .bss  :       > Int_RAM   PAGE = 1
}
```

付録 D

Hex 変換ユーティリティの例

Hex 変換ユーティリティは、多数のオプションと機能を提供する柔軟性の高いユーティリティです。EPROM システムの正しい構成方法と EPROM プログラマの要件を理解すると、簡単にファイルを特定のアプリケーション用に変換できます。

この付録に挙げた 4 つの主な例には、複数の EPROM メモリ・システム用の Hex コマンド・ファイルの開発方法、ホールの回避方法、ブート・テーブルの作成方法が示されています。これらの例では、例 D - 1 に示すアセンブリ・コードを使用しています。

例 D - 1.　Hex 変換ユーティリティの例で使用するアセンブリ・コード

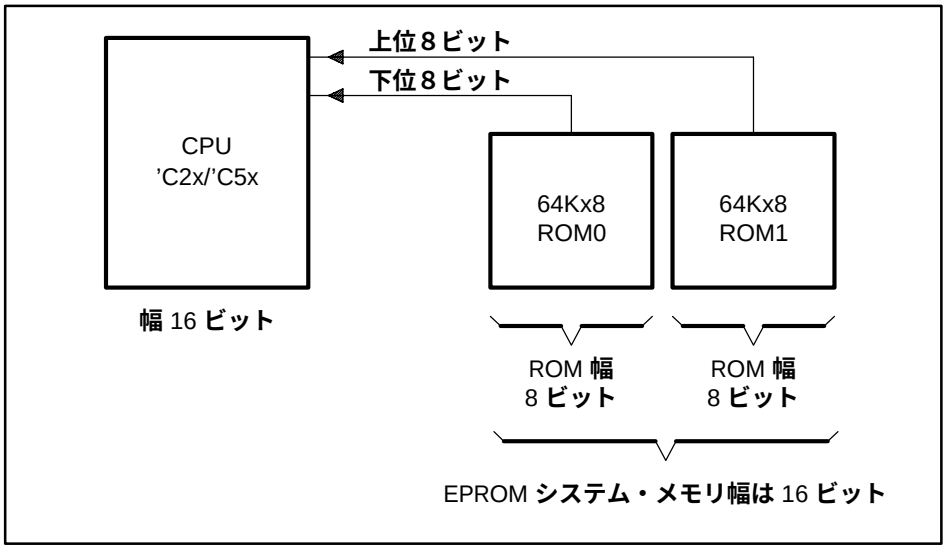
```
*********************************************************
* Assemble two words into section, "sec1"               *
*********************************************************

     .sect "sec1"
     .word 1234h
     .word 5678h

*********************************************************
* Assemble two words into section, "sec2"               *
*********************************************************

     .sect "sec2"
     .word 0aabbh
     .word 0ccddh

     .end
```

項目　　ページ

D.1 例 1 : 2 つの 8 ビット EPROM を対象とした Hex コマンド・ファイルの作成

例 1 では、図 D - 1 に示すメモリ・システム用に COFF オブジェクト・ファイルを変換するのに必要な Hex コマンド・ファイルの作成方法を示します。このシステムには、'C2x または 'C5x ターゲット・プロセッサとインターフェイスする 64K x 8 ビットの 外部 EPROM が 2 つあります。各 EPROM が、ターゲット・プロセッサ用の 16 ビット・ワードのうちの 8 ビットを提供します。

図 D - 1. 2 つの 8 ビット EPROM があるシステム



デフォルトでは、Hex 変換ユーティリティは、変換後の出力ファイルのアドレスを生成するためのベースとして、リンカのロード・アドレスを使用します。ただし、このアプリケーションでは、コードは、リンカで指定されたアドレス（0xF000）でなく、物理 EPROM アドレス 0x0010 にあります。このアドレス空間の変換はターゲット・ボードの回路が取り扱います。paddr パラメータがセクションを割り振り、コードを EPROM アドレス 0x0010 に焼き付けます。

paddr パラメータは、SECTIONS 疑似命令の中で指定します（詳細については、11-20 ページの 11.6 節を参照してください）。変換の際に組み込む 1 セクションのロード・アドレスを paddr パラメータを使って指定した場合は、同時に組み込むすべてのセクションについて、各セクションごとに paddr を指定する必要があります。paddr パラメータを設定するときは、指定するアドレスがリンカが割り当てる後続のセクションのロード・アドレスとオーバーラップしないことを確認しなければなりません。

例 1 では、2 つのセクション、sec1 と sec2 を定義しています。これらの各セクションに SECTIONS 疑似命令の内部から簡単に paddr オプションを追加することができます。ただし、多数のセクションのある大きなアプリケーションの場合やコード開発途中でセクション・サイズが頻繁に変わったりする場合などは、作業の管理ができなくなることがあります。

この問題を回避するには、リンク段階でセクションを結合して、変換用の単一のセクションを作成します。この処理には、例 D - 2 に示すリンカ・コマンド・ファイルを使用してください。

例 D - 2.　2 つの 8 ビット EPROM を対象にしたリンカ・コマンド・ファイル

```
test.obj
-o a.out
-m test.map

MEMORY
{
   PAGE 0 : EXT_PRG ; org = 0xF000 , len = 0x0100
}

SECTIONS
{
   outsec: { *(sec1)
             *(sec2)  } > EXT_PRG PAGE 0
}
```

この例に示す EPROM プログラマには、以下のシステム条件があります。

- ❑ EPROM システム・メモリ幅は 16 ビット
- ❑ ROM1 にはワードの上位 8 ビットを格納する
- ❑ ROM0 にはワードの下位 8 ビットを格納する
- ❑ Hex 変換ユーティリティは、EPROM アドレス 0x0010 から開始するようにコードを配置する
- ❑ Intel フォーマットを使用する
- ❑ Hex 変換ユーティリティの出力ファイルのアドレスにバイト・インクリメントを選択する（メモリ幅はデフォルトで決まっています）

システムの条件を選択するには、以下のオプションを使用します。

オプション	説明
-i	Intel フォーマットを作成します。
-byte	変換後の出力ファイルのアドレス用に、バイト・インクリメントを選択します。
-memwidth 16	EPROM システム・メモリ幅を 16 に設定します。
-romwidth 8	物理 ROM 幅を 8 に設定します。

上記のメモリ幅および ROM 幅の値を使用すると、ユーティリティが自動的に 2 つの出力ファイルを生成します。出力ファイルの数は、ROM 幅に対するメモリ幅の比によって決まります。 ROM0 ファイルには、16 ビットの生データの下位 8 ビットが格納され、ROM1 ファイルには、対応するデータの上位 8 ビットが格納されます。

例 D - 3 に、選択したオプションをすべて指定したHexコマンド・ファイルを示します。

例 D - 3.　2 つの 8 ビット EPROM を対象にした Hex コマンド・ファイル

```
a.out             /* COFF object input file             */
-map examp1.mxp

/*********************************************************/
/* Set parameters for EPROM programmer                   */
/*********************************************************/

-i                /* Select Intel format                */
-byte             /* Select byte increment for addresses */

/*********************************************************/
/* Set options required to describe EPROM memory system  */
/*********************************************************/

-memwidth 16      /* Set EPROM system memory width       */
-romwidth  8      /* Set physical width of ROM device    */

ROMS
{
   PAGE 0 : EPROM : origin = 0x00, length = 0x10000,
                    files = {low8.bit, upp8.bit}
}

SECTIONS
{  outsec: paddr=0x10  }
```

図 D - 2(a) は、下位 8 ビットが格納される ROM0 用の変換後のファイル (low8.bit) の内容を示しています。図 D - 2(b) は、上位 8 ビットのデータが格納される ROM1 用の変換後のファイル (upp8.bit) の内容を示しています。

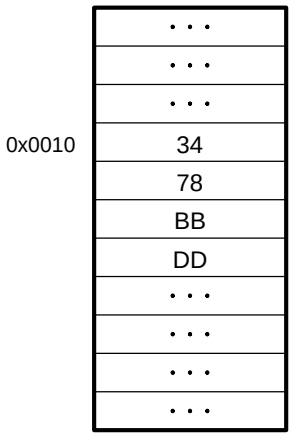
図 D - 2. 出力ファイルのデータ

*(a) low8.bit: (*下位ビット*)*

変換後の出力ファイルのデータ

```
:040010003478BBDDA8
:00000001FF
```

EPROM - ROM0の対応マップ(例 D - 1を参照)

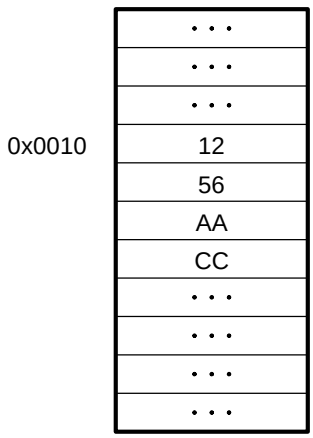


*(b) upp8.bit: (*上位ビット*)*

変換後の出力ファイルのデータ

```
:040010001256AACC0E
:00000001FF
```

EPROM - ROM1の対応マップ(例 D - 1を参照)



ユーティリティがどのように変換を実行したかを正確に知るには、 - map オプションを指定します。 - map オプションは必須ではありませんが、出力に関する有益な情報を生成します。生成されるマップを例 D - 4 に示します。

例 D - 4.　例 D - 3 に示すHex コマンド・ファイルから生成されるマップ・ファイル

```
*******************************************************************
TMS320C1x/C2x/C2xx/C5x Hex Converter                      Version x.xx
*******************************************************************
Fri Feb  3 15:26:11 1995

INPUT FILE NAME: <a.out>
OUTPUT FORMAT:   Intel

PHYSICAL MEMORY PARAMETERS
   Default data width:    16
   Default memory width:  16
   Default output width: 8

OUTPUT TRANSLATION MAP
-------------------------------------------------------
00000000..0000FFFF  Page=0  ROM Width=8  Memory Width=16  ″EPROM″
-------------------------------------------------------
   OUTPUT FILES: low8.bit [b0..b7]
                 upp8.bit [b8..b15]

   CONTENTS: 00000010..00000013  Data Width=2  outsec
```

D.2 例 2 : 複数のセクション間のホールの回避

メモリ幅がデータ幅よりも小さい場合、あるセクションの始めや セクション間にホールが生じることがあります。これは、ロード・アドレスに補正係数を掛けた結果生じるものです。詳細については、11-34 ページの 11.10 節を参照してください。

変換後のセクションの間のホールは除去する必要があります。セクションは、次の方法のどちらかで連続させることができます。

- SECTIONS 疑似命令に指定されたセクションごとに paddr を指定する。この方法を例えば Hex 変換ユーティリティに、出力ファイルのアドレス・フィールドに強制的に特定の値を設定させることができます。セクション・アドレスがオーバーラップしないように注意します。この方法に使用するリンカ・コマンド・ファイルを例 D - 5 に示します。このコマンド・ファイルを使ってリンカを実行し、次に、例 D - 6 に示す一連のコマンドを使って Hex 変換ユーティリティを実行する必要があります。

- 各セクションをリンクして変換用の 1 つの出力セクションにする。この方法に使用するリンカ・コマンド・ファイルを例 D - 7 に示します。このコマンド・ファイルを使ってリンカを実行し、次に、例 D - 8 に示す一連のコマンドを使って Hex 変換ユーティリティを実行する必要があります。

例 D - 5. リンカ・コマンド・ファイル：ホール回避方式１

```
/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
   PAGE 0: PRG: org = 0x0800, len = 0x0100
           EXT: org = 0xF000, len = 0x0100
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
   sec1 : load = EXT PAGE 0
   sec2 : load = EXT PAGE 0, run = PRG PAGE 0
}
```

例 D - 6.　Hex コマンド・ファイル：ホール回避方式１

```
 -i
 a.out
 -map example.mxp

 ROMS
 {
  PAGE 0: ROM : org = 0x0000, length = 0x800, romwidth = 8, memwidth = 8
 }

 SECTIONS
 {
   sec1 : paddr = 0x0000
   sec2 : paddr = 0x0004
 }
```

例 D - 7.　リンカ・コマンド・ファイル：ホール回避方式２

```
* SPECIFY THE SYSTEM MEMORY MAP

MEMORY
{
  PAGE 0: PRG: org = 0x0800, len = 0x0100
          EXT : org = 0xF000, len = 0x0100
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY

SECTIONS
{
  outsec : {  *(sec1)
              *(sec2) } > EXT PAGE 0
}
```

例 D - 8.　Hex コマンド・ファイル：ホール回避方式２

```
-i
a.out
-map example.mxp

ROMS
{
 PAGE 0: ROM : org = 0x0100, length = 0x0800, romwidth = 8, memwidth = 8,
               files = {example.hex}
}

SECTIONS
{
  outsec : paddr = 0x100
}
```

D.3 例 3: 'C50 のブート・テーブルの作成

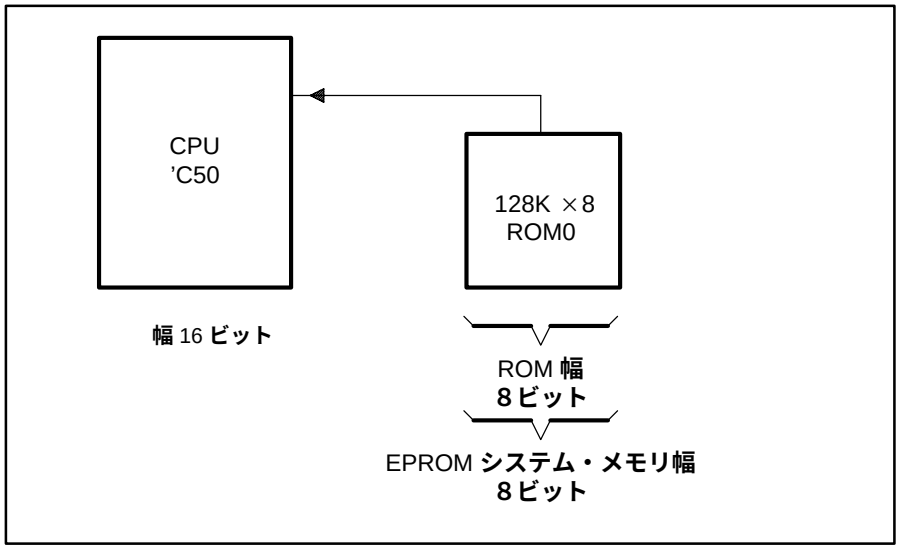
例 3 は、リンカと Hex 変換ユーティリティを使って 'C50 用のブート・ロード・テーブルを作成する方法を示しています。この節で使用している C コードは D-9 ページの例 D - 9 に示されています。

例 D - 9. 'C50 用の C コード

```
int array[]={1,2,3,4};

main()
{
  array[0] = 5;
}
```

この例では、図 D - 3 に示す EPROM メモリ・システム用の出力ファイルを作成します。このアプリケーションでは、単一の 'C50 デバイスを 128K × 8 ビットの EPROM からブートします。システムの条件は、ブート・テーブルが EPROM メモリ・アドレス 0 に入っていなければならないことです。

図 D - 3. 'C50 用の EPROM システム



'C50 のブート・テーブルを作成する場合の考慮事項として次の 2 点があります。

- ブート・ロードに必要なセクションのリンク
- ブート・ルーチン選択ワードの構成と配置

ブート・ロードに必要なセクションのリンク

オンチップ・ブート・ローダにロードされるのは 1 つのブロックだけです。このため、TMS320C2x/C2xx/C5xCコンパイラでコンパイルしたCコードをロードしようとすると、問題が起きることがあります。TMS320C2x/C2xx/C5x C コンパイラは、C ソース・コードをコンパイルするときに、複数のセクションまたはブロックを作成します。アプリケーションによっては、プログラムに関連するすべてのセクションがブートに組み込まれていないと、完全な実行可能プログラムが作成できないこともあります。このような場合は、個々のセクションを結合して 1 つのブート用のセクションにする必要があります。

Hex 変換ユーティリティは、個々のセクションの結合は行いません。したがって、これらのセクションをグループ化するには、リンカを使用しなければなりません。

コンパイラが作成するセクションは、2 つのカテゴリに分類できます。初期化されたセクション（データまたはコードを含むセクション）と初期化されないセクション（空間は確保されているが実データを含んでいないセクション）です。TMS320C2x/C2xx/C5x C コンパイラにより作成される初期化されたセクションには、.text、.cinit、.const、および .data が含まれます。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無視され、変換の対象とはなりません。

ほとんどのアプリケーションの場合、.text セクションと .cinit セクションをブート時に組み込む必要があります。これによって、C ブート・ルーチン（boot.asm で定義されている c_int0）に必要なコードと情報をロードし実行することができます。C ブート・ルーチンは C 環境を初期化し、アプリケーション・コードの main 関数に分岐します。

.text セクションと .cinit セクションをリンクして、リンカ・コマンド・ファイルの中の 1 つのセクションにする必要があります。この .cinit セクションには、初期値付きで宣言された (例えば int x = 5;) すべてのグローバルまたは静的なCシンボルについての初期化データとテーブルが含まれています。リンカは、.cinit セクションを他のセクションとは異なる方法で取り扱うので注意してください。

リンカは、リンク時に**出力セクション**として指定された .cinit セクションを検出すると、自動的に次の処理を実行します。

- ❑ シンボル cinit を、組み込み後の .cinit セクションの開始点を指すように設定する。
- ❑ セクションの終りに 1 ワードを付加する。

付加された最後のワードには、初期化テーブルの終りを示すマークとして使用される 0 が含まれています。ただし、.cinit が**入力セクション**としてしか指定されていない場合には、リンカは cinit を - 1 に設定し、初期化テーブルがロードされなかったことを示します。この場合、C ブート・ルーチン c_int0 は、グローバルまたは静的な C シンボルの初期化を行いません。

.cinit セクションを .cinit 以外の出力セクションにリンクする場合は、リンカは上記に示した自動機能を実行しません。したがって、これらの機能をリンカ・コマンド・ファイルの中で明示的に実現しておく必要があります。D-11 ページの例 D - 10 は、.text と .cinit を boot_sec という名前の単一の出力セクションに配置するリンカ・コマンド・ファイルを示しています。

例 D - 10. 'C50用の単一のブート・セクションを作成するためのリンカ・コマンド・ファイル

```
-c
-l rts50.lib
-m map
-o a.out

MEMORY
{
  PAGE 0 : PROG   : origin = 0x0800h, length = 0x1000h

  PAGE 1 : DATA   : origin = 0x0800h, length = 0x1000h
}

SECTIONS
{
  boot_sec: { *(.text)

              /*************************************/
              /* Set start address for C init table */
              /*************************************/
                 cinit = .;

              /*************************************/
              /* Include all cinit sections         */
              /*************************************/
                 *(.cinit)

              /*************************************/
              /* Reserve a single space for the zero */
              /* word to mark end of C init          */
              /*************************************/
                 .+=1;

    } fill = 0x0000,    /* Make sure fill value is 0 */
      load = PROG PAGE 0

  .data   : {} > DATA   PAGE 1
  .bss    : {} > DATA   PAGE 1
  .const  : {} > DATA   PAGE 1
  .sysmem : {} > DATA   PAGE 1
  .stack  : {} > DATA   PAGE 1
}
```

例 D - 11 は、例 D - 10 に示すコマンド・ファイルを使ってリンカを実行したときに生成されるマップ・ファイルの一部を示しています。

例 D - 11. 例 D - 10 のコマンド・ファイルから作成されるマップ・ファイルのセクション割り振り部分

```
      SECTION ALLOCATION MAP

   output                                  attributes/
   section   page    origin      length       input sections
   --------  ----  ----------  ----------   ----------------
   boot_sec   0    00000800    00000076
                     00000800    00000004     test1.obj (.text)
                     00000804    0000002a     rts50.lib : boot.obj (.text)
                     0000082e    00000041               : exit.obj (.text)
                     0000086f    00000003     test1.obj (.cinit)
                     00000872    00000003     rts50.lib : exit.obj (.cinit)
                     00000875    00000001     --HOLE-- [fill = 0000]

   .data      1    00000800    00000000     UNINITIALIZED
                     00000800    00000000     test1.obj (.data)
                     00000800    00000000     rts50.lib : exit.obj (.data)
                     00000800    00000000               : boot.obj (.data)

   .bss       1    00000800    00000022     UNINITIALIZED
                     00000800    00000021     rts50.lib : exit.obj (.bss)
                     00000821    00000000               : boot.obj (.bss)
                     00000821    00000001     test1.obj (.bss)

   .const     1    00000800    00000000     UNINITIALIZED

   .sysmem    1    00000800    00000000     UNINITIALIZED

   .stack     1    00000822    00000400     UNINITIALIZED
                     00000822    00000000     rts50.lib : boot.obj (.stack)


   GLOBAL SYMBOLS

   address  name                             address  name
   -------- ----                             -------- ----
   00000800 .bss                             00000400 __STACK_SIZE
   00000800 .data                            00000800 edata
   00000400 __STACK_SIZE                     00000800 .bss
   00000869 _abort                           00000800 .data
   00000848 _atexit                          00000800 _main
   00000804 _c_int0                          00000804 _c_int0
   0000082e _exit                            00000821 _flag
   00000821 _flag                            00000822 end
   00000800 _main                            0000082e _exit
   0000086f cinit                            00000848 _atexit
   00000800 edata                            00000869 _abort
   00000822 end                              0000086f cinit
```

リンカが、コマンド・ファイルの指定に従って埋め込み値 0 を使って、セクション boot_sec の終りにホールを配置していることに注意してください。また、グローバル・シンボル cinit は、リンクに組み込まれた最初の .cinit セクションの開始点と一致しています。例D - 10のコマンド・ファイルを使ってリンカを実行すると、リンカは、出力ファイルに .text セクションが含まれていないこと、およびグローバル・シンボル cinit が再定義されることを示す警告を発行します。このような警告は無視してかまいません。

ブート・ルーチン選択ワードの構成と配置

ブート・ルーチン選択ワードがオンチップ・ブート・ローダが稼動するモードを設定します。オンチップ・ブート・ローダが、プログラム・メモリのロケーション 0xFFFF からこのワードを読み込みます。この 'C50アプリケーション用にオンチップ・ブート・ローダが8 ビット・パラレルEPROMモードで稼動することが必要です (オンチップ・ブート・ロードとブート・ルーチン選択ワードに関する詳しい情報は、TMS320C5x ユーザーズ・マニュアルを参照してください) 。

Hex 変換ユーティリティは、ブート・ルーチン選択ワードを自動的に生成し配置することはありません。したがって、ユーザーが確認して、適正な値が適正なメモリ・アドレスに入るようにします。この 'C50 ブート・テーブルの場合は、例 D - 12 に示すリンカ・コマンド・ファイルがこの作業を実行します。

例 D - 12. 'C50 用のブート・ルーチン選択ワードを設定するためのリンカ・コマンド

```
-c  -l rts50.lib  -m map  -o a.out

MEMORY
{
  PAGE 0 : PROG    : origin = 0x0800, length = 0x1000
  PAGE 0 : BRSW    : origin = 0xffff, length = 0x0001
  PAGE 1 : DATA    : origin = 0x0800, length = 0x1000
}

SECTIONS
{
  boot_sec: { *(.text)

    cinit = .;  /* Set start address for C init table. */


    *(.cinit)   /* Include all cinit sections.         */

                /*****************************************/
    .+=1;       /* Reserve a single space for the zero */
                /* word to mark end of C init.         */
                /*****************************************/


    } fill = 0x0000,     /* Make sure fill value is 0  */
      load = PROG PAGE 0

                /*****************************************/
                /* Create a single word section that   */
   .brsw   : {  /* contains the proper fill value for  */
                /* the needed boot routine select mode */
                /*****************************************/

    .+=1;       /* Reserve a word in this section.     */

                /*****************************************/
                /* Select fill value that corresponds  */
                /* to correct boot routine select mode */
                /*****************************************/

       } fill = 0x0001,
         load = 0xffff PAGE 0

  .data   : {} > DATA   PAGE 1
  .bss    : {} > DATA   PAGE 1
  .const  : {} > DATA   PAGE 1
  .sysmem : {} > DATA   PAGE 1
  .stack  : {} > DATA   PAGE 1
}
```

D-14 ページの例 D - 12 に示すコマンド・ファイルでは、.brsw という名前の新しいセクションが作成されます。このセクションには、値が 0x0001 に等しいワードが 1 つだけ入っています。これは、8ビット・パラレルEPROMモードでソース・アドレスが0x0000のブート・ルーチン選択モードに対応します。

例 D - 12 のコマンド・ファイルを使ってリンカを実行すると COFF ファイルが作成されます。このCOFFファイルをHex変換ユーティリティの入力として使用して、必要なブート・テーブルを作成することができます。

オプション

Hex 変換ユーティリティには、EPROM プログラマの要件を説明するオプションと、EPROM メモリ・システムを説明するオプションがあります。例 3 では、EPROM プログラマには、「Hex ファイルは Intel フォーマットであること」という要件が 1 つだけあるものとします。

D-9 ページの図 D - 3 に示す EPROM メモリ・システムでは、EPROM システム・メモリ幅も物理ROM幅も8ビットです。システムの条件を反映するために、以下のオプションが選択されます。

オプション	説明
`-i`	Intel フォーマットを作成します。
`-memwidth 8`	EPROM システム・メモリ幅を 8 に設定します。
`-romwidth 8`	物理 ROM 幅を 8 に設定します。

このアプリケーションではパラレル・ブート・モード用のブート・テーブルを作成する必要があるので、以下のオプションも同様に選択しなければなりません。

オプション	説明
`-boot`	ブート・ロード・テーブルを作成します。
`-bootorg 0x0000`	ブート・テーブルをアドレス 0x0000 に配置します。

.boot_sec と .brsw の 2 つのセクションをロードする必要があります。ブート・テーブルにはこのうちの 1 つだけが組み込まれるので、ブート可能なセクションをHexコマンド・ファイルを使用して指定しなければなりません。

物理ROM内のブート・テーブルの範囲を指定するには、ROMS疑似命令を使用します。例 D - 13 は、COFF ファイルを変換するのに必要な完全なコマンド・ファイルを示しています。

例 D - 13. COFF ファイルを変換するための Hex コマンド・ファイル

```
a.out                  /* Input COFF file               */
-i                     /* Select Intel Format          */
-map boot.mxp
-o boot.hex            /* Name hex output file boot.hex */

-memwidth 8            /* Set EPROM System Memory Width */
-romwidth 8            /* Set physical ROM width        */

-boot                  /* Build a boot table           */
-bootorg 0x0000        /* Place boot table in EPROM    */
                       /* starting at address 0x0000   */



ROMS
{
    PAGE 0 : ROM : origin = 0x0000, length = 0x20000
}


SECTIONS
{
   boot_sec : paddr=boot
   .brsw    : paddr=0x1fffe
}
```

paddrオプションは、指定セクションのリンク済みセクション・アドレスを上書きします。システム・メモリ幅は 8 ビットなので、アドレス空間が係数 2 （データ幅をメモリ幅で割った値）で拡張されます。したがって、.brsw セクションを割り振るためのアドレスには、係数 2 が掛けられます（0xFFFF*2 - > 0x1FFFE）。

例 3 ではメモリ幅と ROM 幅が同じです。したがって、Hex 変換ユーティリティは単一の出力ファイルを作成します。出力ファイルの数は memwidth と romwidth との比で決まります。

D-17 ページの例 D - 14 は、 - map オプションを組み込んだ 例 D - 13 のコマンド・ファイルを実行したときに作成されるマップ・ファイル boot.map を示しています。

例 D - 14.　例 D - 13 に示すコマンド・ファイルから作成されるマップ・ファイル

```
*******************************************************************
TMS320C1x/C2x/C2xx/C5x Hex Converter                      Version x.xx
*******************************************************************
Wed Feb  1 10:36:16 1995

INPUT FILE NAME: <a.out>
OUTPUT FORMAT:   Intel

PHYSICAL MEMORY PARAMETERS
   Default data width:    16
   Default memory width:  8 (MS-->LS)
   Default output width: 8

BOOT LOADER PARAMETERS
   Table Address: 0000, PAGE 0


OUTPUT TRANSLATION MAP
-----------------------------------------------------
00000000..0001ffff  Page=0  ROM Width=8  Memory Width=8  "ROM"
-----------------------------------------------------
   OUTPUT FILES: boot.hex [b0..b7]

   CONTENTS: 00000000..0000000f   BOOT TABLE
                                    .text : dest=00000800
                                     size=00000002  width=00000002
             0001fffe..0001ffff  Data Width=2  .brsw
```

例 D - 13 に示すコマンド・ファイルから作成される Hex 出力ファイル boot.hex を、例 D - 15 に示します。

例 D - 15.　例 D - 13 に示すコマンド・ファイルから作成される Hex 変換ユーティリティの出力ファイル

```
:200000008000075B901BC109021EF00BF080822BF090822BE42BF00BC005E07FFF9BE477D
:20002000BF80086FB801E38808157A8908197A8908007A890869BF80086FBC00A680B801D5
:20004000028A7B990822EF0089090012A680B801028ABEC6082BA6A0B8018B007989081C71
:2000600080A08AA08180B00100E086AEBC1006007B80083A798008468B9886801080BF9012
:200080000801908005 8D1089BE308B8E7B80083A7980084680A08180B00100E0BE47BC100E
:2000A0001000BA20E34408567D800865B9018B001000B8019000BA01BF9008018B88908093
:2000C000058ABF0AFFFE8BE0108D9080B9008B897C020080EF007E80082EB90190A08B90C0
:1000E000EF000001082100000001080000000000EE
:00000001FF
```

D.4 例 4: 'C26 のブート・テーブルの作成

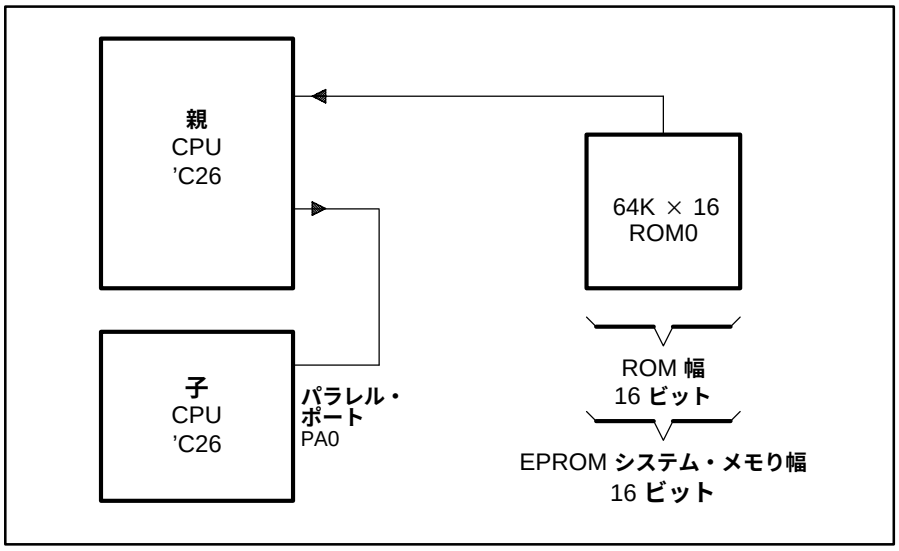
例 4 は、リンカと Hex 変換ユーティリティを使って 'C26 用のブート・ロード・テーブルを作成する方法を示しています。この節で使用している C コードは D-18 ページの例 D - 16 に示されています。

例 D - 16. 'C26 用の C コード

```
int array[]={1,2,3,4};

main()
{
  array[0] = 5;
}
```

このアプリケーションでは、システムに 2 つの 'C26 デバイスがあるものとします。1 つのデバイスは親プロセッサとして動作し、第 2 のプロセッサ（子プロセッサ）をその子のI/Oポートを介してブートします。子プロセッサ用のアプリケーション・コードが入っているブート・テーブルは、図 D - 4 に示すように、親に接続された外部 EPROM に格納されています。

図 D - 4. 'C26 用のサンプル EPROM システム



コードをリンクするときに、ブート可能セクションのロード・アドレスとしてどのアドレスを使用するか疑問を持つ場合があります。ブート可能セクション内のコードはパラレル・ポートを介してロードされるので、リンク用に指定するアドレスは 1 つだけです。つまり、コードが子プロセッサに入る最終アドレスを指定することになります。オンチップ・ブート・ローダは、最初は、データ・メモリ・アドレス 0x0200 から始まるブロック B0 にダウンロードされます。しかし、コードがロードされると、ブロックB0はプログラム・メモリとして構成されます。そして、プログラム・メモリのアドレス0x0FA00のブロック B0 内の最初のロケーションに制御が移されます。このロケーションからコードが取り出され、実行されます。したがってこれがコードのリンクされるべきアドレスになります。

D-19 ページの例 D - 17 では、 D.3 節で説明したとおり、リンカはブート可能コードを単一セクションに割り振っています。

例 D - 17. 親 CPU と子 CPU のある 'C26 のリンカ・コマンド・ファイル

```
-c
-l rts25.lib
-m map

MEMORY
{
   PAGE 0 : PRG_B0 : org = 0xFA00 , length = 0x0100
   PAGE 1 : EXT: org = 0x0800 , length = 0x1000
}

SECTIONS
{
   boot_sec:{ *(.text)

              cinit = .;
              *(.cinit)
              .+=1;

             } fill = 0,
                  ************************************
                  /* Set run address for boot section */
                  ************************************
               load = PRG_B0 PAGE 0

   .bss : {} > EXTPAGE 1
   .sysmem : {} > EXTPAGE 1
   .stack  : {} > EXTPAGE 1
 }
```

例 D - 18 は、例 D - 17 に示すコマンド・ファイルを使ってリンカを実行したときに作成されるマップ・ファイルの一部を示しています。

例 D - 18. 例 D - 17 のコマンドから作成されるマップ・ファイルのセクション割り振り部分

```
        SECTION ALLOCATION MAP

 output                                  attributes/
section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
boot_sec   0    0000fa00    00000072
                  0000fa00    00000004     test1.obj (.text)
                  0000fa04    00000027     rts25.lib : boot.obj (.text)
                  0000fa2b    00000040               : exit.obj (.text)
                  0000fa6b    00000003     test1.obj (.cinit)
                  0000fa6e    00000003     rts25.lib : exit.obj (.cinit)
                  0000fa71    00000001     --HOLE-- [fill = 0000]

.bss       1    00000800    00000022     UNINITIALIZED
                  00000800    00000021     rts25.lib : exit.obj (.bss)
                  00000821    00000000               : boot.obj (.bss)
                  00000821    00000001     test1.obj (.bss)

.sysmem    1    00000800    00000000     UNINITIALIZED

.stack     1    00000822    00000400     UNINITIALIZED
                  00000822    00000000     rts25.lib : boot.obj (.stack)

.data      1    00000000    00000000     UNINITIALIZED
                  00000000    00000000     test1.obj (.data)
                  00000000    00000000     rts25.lib : exit.obj (.data)
                  00000000    00000000               : boot.obj (.data)

GLOBAL SYMBOLS

address  name                             address  name
-------- ----                             -------- ----
00000800 .bss                             00000000 edata
00000000 .data                            00000000 .data
00000400 __STACK_SIZE                     00000400 __STACK_SIZE
0000fa65 _abort                           00000800 .bss
0000fa45 _atexit                          00000821 _flag
0000fa04 _c_int0                          00000822 end
0000fa2b _exit                            0000fa00 _main
00000821 _flag                            0000fa04 _c_int0
0000fa00 _main                            0000fa2b _exit
0000fa6b cinit                            0000fa45 _atexit
00000000 edata                            0000fa65 _abort
00000822 end                              0000fa6b cinit

[12 symbols]
```

例 D - 17 のコマンド・ファイルを使ってリンカを実行すると COFF ファイルが作成されます。このCOFFファイルをHex変換ユーティリティの入力として使用して、必要なブート・テーブルを作成できます。

オプション

Hex 変換ユーティリティには、EPROM プログラマの要件を説明するオプションと、EPROM メモリ・システムを説明するオプションがあります。例 4 では、EPROM プログラマには、「Hex ファイルは Intel フォーマットであること」という要件が 1 つだけあるものとします。

D-18 ページの図 D - 4 に示す EPROM メモリ・システムでは、EPROM システム・メモリ幅は16ビットで、物理ROM幅も 16ビットです。システムの条件を反映するために、以下のオプションが選択されます。

オプション	説明
`-i`	Intel フォーマットを作成します。
`-memwidth 16`	EPROM システム・メモリ幅を 16 に設定します。
`-romwidth 16`	物理 ROM 幅を 16 に設定します。

アプリケーションではパラレル・ブート・モード用のブート・テーブルを作成する必要があり、'C2xではユーザがメモリ構成レジスタを設定する必要があるので、以下のオプションも同様に選択しなければなりません。

オプション	説明
`-boot`	ブート・ロード・テーブルを作成します。
`-bootorg PARALLEL`	パラレル・ブート・テーブル・ヘッダを選択します。
`-cg 0x0000`	ブート・テーブル・ヘッダの割り込み/構成ワードを 0x0000 に設定します。

このアプリケーションでは、ROMS 疑似命令を使ってブート・テーブルのアドレス範囲を指定する必要があります。ブート・テーブルは通信ポートを介した伝送用に編成されていますが、テーブル自体は親プロセッサの外部EPROMメモリに入っています。したがって、このテーブルには、EPROMメモリ空間内のロード・アドレスが必要です。例D - 19 の ROMS 疑似命令により、ブート・テーブルが物理 ROM のアドレス 0x0000 に配置されます。

選択されたオプションがすべて含まれている Hex コマンド・ファイルを例 D - 19 に示します。

例 D - 19.　親 CPU と子 CPU のある 'C26 用の Hex コマンド・ファイル

```
a.out
-o   c26boot.hex
-map c26boot.mxp

-i

-memwidth 16
-romwidth 16


-boot
-bootorg PARALLEL

ROMS
{
   PAGE 0: ROM: org = 0x0000  , length = 0x10000
}
```

- map オプションも使用されています。 - map オプションの使用は、必須ではありませんが、ユーティリティの変換方法を理解するには、便利です。D-23ページの例 D - 20は、例 D - 19 のコマンド・ファイルから作成されるマップ・ファイルを示しています。

例 D - 20.　**例** D - 19 **に示すコマンド・ファイルから作成されるマップ・ファイル（**c26boot.mxp**）**

```
*******************************************************************
   TMS320C1x/C2x/C2xx/C5x Hex Converter                  Version x.xx
*******************************************************************
   Wed Feb  1 17:39:59 1995

   INPUT FILE NAME: <a.out>
   OUTPUT FORMAT:   Intel

   PHYSICAL MEMORY PARAMETERS
      Default data width:    16
      Default memory width:  16
      Default output width: 16

   BOOT LOADER PARAMETERS
      Table Address: 0000, PAGE 0


   OUTPUT TRANSLATION MAP
   ------------------------------------------------------
   00000000..0000ffff  Page=0  ROM Width=16  Memory Width=16  "ROM"
   ------------------------------------------------------
      OUTPUT FILES: c26boot.hex [b0..b15]

      CONTENTS: 00000000..00000076   BOOT TABLE
                                       boot_sec : dest=0000fa00
                                        size=00000072  width=00000002
```

関連の Hex **出力ファイルの内容を 例** D - 21 **に示します。**

例 D - 21.　**例** D - 19 **に示すコマンド・ファイルから作成される出力ファイル**
（c26boot.hex）

```
:20000000001800000071CA01C8106021CE26D0000822D1000822CE02CE08CE07D001FA6B99
:20001000CC01F680FA12FE89FA16FE89FA00FE89FA657E02D001FA6B558858A0CC01588053
:2000200032903 38BFB9AFA2455897F02CE26CC0158ABFB9AFA24CC01FF88FA1A7AA070A025
:200030007180C00130E076AEC8103600FB80FA37FF80FA43559876802080D0020801608016
:20004000358D2089CE24558EFB80FA37FF80FA4370A07180C00130E0CE07C8102000CD206C
:20005000F380FA52CA01FF80FA612000CC016000CD01D00208015588 6080358AD200FFFEEB
:2000600055E0208D6080CA0055897F023080CE26CA0160A0FE80FA2B5590CE260001082180
:0E00700000000001080000000000086A000007
:00000001FF
```

付録 E

アセンブラ・エラー・メッセージ

アセンブラは、次のようなタイプのエラー・メッセージを発行します。

- ❑ 致命的エラー
- ❑ 非致命的エラー
- ❑ マクロ・エラー

アセンブラは、2 回目のパスを完了したときにアセンブリ中に検出したエラーを報告します。また、(リスト・ファイルが作成されている場合は) それらのエラーをリスト・ファイルにも出力します。エラーは、そのエラーの原因となったソース行の後に出力されます。

この付録には、3 つのタイプのアセンブラ・エラー・メッセージをアルファベット順に記載してあります。大部分のエラーは、致命的エラーです。致命的エラーでない場合や、マクロ・エラーの場合は、アセンブラ・リストにそのことが記述してあります。また、大部分のエラー・メッセージについては、「説明」の項で問題を説明するとともに、「処置」の項で考えられる解決策を示してあります。エラー・メッセージそのもので十分な説明が得られる場合は、「処置」の項だけを見ても構いません。説明を読めば (コードの検査および訂正するための) 「処置」がすぐに分かる場合は、「処置」の項を省略してあります。

A

absolute value required

説明　絶対シンボルが必要な場所で再配置可能シンボルが使用されています。

処置　絶対シンボルを使用して下さい。

address required

説明　この命令がオペランドとして要求するアドレスが指定されていません。

an identifier in the expression is invalid

説明　式でつかわれているシンボルの属性が不正です。

処置　演算可能な式であるかどうかを確かめて下さい。

argument must be character constant

説明　文字突き合わせ関数に必要な文字定数が指定されていません。

処置　文字定数を使用して下さい。

B

bad field width

説明 .field 疑似命令で、指定の値が指定のビット数では表現できません。

bad macro library format

説明 指定されたマクロ・ライブラリは、所定のフォーマットではありません。

処置 マクロ・ライブラリが、アセンブルされていないアセンブラ・ソース・ファイルであることを確認してください。また、マクロ名とメンバ名が同じであり、ファイルの拡張子が .asm であることも確認してください。

.break encountered outside loop block

説明 .break 疑似命令がループ・ブロックの外で宣言されています。

処置 コードを調べて、.endloop 疑似命令が誤った場所に置かれていないか確認するか、.break 疑似命令を削除してください。

C

cannot open library

説明 .mlib 疑似命令で指定したライブラリ名が存在しないか、すでに使用されています。

処置 綴り、パス名、環境変数などをチェックしてください。

cannot open listing file : *filename*

説明 指定したファイル名はアクセス不能です。

処置 綴り、パス名、環境変数などをチェックしてください。

cannot open object file : *filename*

説明 指定したファイル名はアクセス不能です。

処置 綴り、パス名、環境変数などをチェックしてください。

cannot open source file : *filename*

説明 指定したファイル名はアクセス不能です。

処置 綴り、パス名、環境変数などをチェックしてください。

character constant overflows a word

説明　　文字定数は 2 文字までに制限する必要があります。

close (")") missing

説明　　括弧が対応していません。

close ("]") missing

説明　　大括弧が対応していません。

close quote missing

説明　　引用符が対応していないか欠落しています。

処置　　すべての文字列を引用符で囲んでください。

comma missing

説明　　アセンブラがコンマを検出すべき場所に、コンマがありませんでした。ほとんどの場合は、命令が必要とするオペランドの数より、検出されたオペランドの数の方が少ないことが原因です。

conditional block nesting level exceeded

説明　　条件付きブロックのネストは、32 レベル以内でなければなりません。

conflicts with previous section definition

説明　　.sect で定義したセクションを usect で再定義しているか、またはその逆の操作をしています。

処置　　両者が同じになるように疑似命令を変更するか、どちらかのセクションの名前を変更してください。

copy file open error

説明　　.copy 疑似命令で指定したファイルが存在しないか、すでに使用されています。

処置　　綴り、パス名、環境変数などをチェックしてください。

D

divide by zero

説明　　式または整合定義式の中に不正な除算があります。

duplicate definition

説明　　REF 文のオペランドとして使用されているシンボルが、ソースのラベル・フィールドでも使用されています。あるいは、ソースのラベル・フィールドに同じシンボルが複数回現れています。

処置　　コードを調べて、上記のエラーを確認してください。ローカル・ラベルを再使用する場合は、.newblock を使用してください。

duplicate definition of a structure component

説明　構造体のタグ、メンバ、またはサイズのシンボルが他の場所で定義されているか、.global 疑似命令で使用されています。

E

.else or .elseif needs corresponding .if

説明　.else または .elseif 疑似命令の前に .if 疑似命令がありません。

empty structure

説明　.struct/.endstruct シーケンスには、1 つ以上のメンバが必要です。

expression not terminated properly

説明　式がコンマ、括弧または空白で区切られていません。式は、いずれかで区切る必要があります。

F

filename missing

説明　指定したファイル名が見つかりません。

処置　綴り、パス名、環境変数などをチェックしてください。

floating - point number not valid in expression

説明　整数式が必要な場所で浮動小数点式が使用されています。

I

illegal label

説明　ラベルとして不正なシンボルが使用されています。

処置　正しいシンボルを使用して下さい。

illegal operation in expression

説明　再配置オペランドと外部オペランドの不当な組み合わせを使用した可能性があります。

処置　3-25 ページの表3 - 2 を参照してください。

illegal shift count

説明　シフト・カウントは 0 〜 16 ビットでなければなりません。

処置　命令によって指定可能なシフト・カウントは異なります。各デバイスのマニュアルを参照して下さい。

illegal structure definition

説明　.stag/.eos 構造体定義がネストされているか、.member 以外のデバッグ疑似命令を含んでいます。

処置　B-8 ページの .stag/.etag/.utag/.eos の説明を参照してください。

illegal structure member

説明　構造体定義に、空間を確保するもの以外の疑似命令が含まれています。

処置　4-2 ページ以降の表4 - 1「定数を初期化するための疑似命令」を参照して下さい。

illegal structure, union, or enumeration tag

説明　.stag 疑似命令は識別子ではないオペランドを持っています。

invalid binary constant

説明　2 進定数に対して 0 または 1 以外の数字が使われています。

invalid condition value

説明　入力されたシンボルをアセンブラが有効な条件シンボルとして認識していません。

invalid control bit

説明　入力されたシンボルをアセンブラが有効な制御ビット・シンボルとして認識していません。

invalid decimal constant

説明　10 進定数に対して 0 ～ 9 以外の数字が指定されています。

処置　このエラーがもっともよく起きるのは、16 進数の接尾部として h または H が使用されていない場合です。すべての 16 進数に h または H の接尾部が付いていることを確認してください。

invalid expression

説明　算術計算では再配置可能なシンボルの使用は無効であることが考えられます。

invalid floating - point constant

説明　浮動小数点定数の指定が誤っているか、浮動小数点定数が必要な場所で整数定数が使用されています。

処置　4-42 ページを参照してください。

invalid hexadecimal constant

説明　有効な 16 進数字は、0～9 の数字と A～F の文字だけです。定数には接尾部として h または H を付ける必要があります。また、数字で始まらなければなりません。

invalid indirect addressing format

説明　指定されているアドレッシング・モードが適正ではありません。

invalid octal constant

説明　有効な 8 進数字は、0～7 の整数だけです。定数には接尾部として q または Q を付ける必要があります。

invalid opcode

説明　ソース・レコードのコマンド・フィールドに未定義の命令、疑似命令、またはマクロ名があります。

invalid opcode for selected version

説明　コマンドは、アセンブルしているデバイスでは無効なものです。

処置　- v オプションでデバイスを指定して下さい。

invalid operand combination - check version

処置　デバイスのユーザーズ・マニュアルに記載されている、この命令に関する説明を確認してください。

invalid operand value

説明　アセンブラ疑似命令にはパラメータを使用するものがあります。不正なパラメータ値が指定されています。

invalid option

説明　.option 疑似命令で指定したオプションが不正です。

invalid register value

説明　レジスタを表すシンボルは、アセンブルしているデバイスでは無効なものです。

invalid symbol

説明　シンボルに無効な文字が含まれています。たとえば、文字や下線の代わりに数字で始まっている可能性があります。

invalid symbol qualifier

説明　シンボル文字列が、32 文字以内の英数字（大文字/小文字の A ～ Z、0 ～ 9、$、および _）で構成されていないか、または数字で始まっています。

invalid version number

説明　アセンブラの有効なバージョン・オプションである - v10、 - v16、 - v20、 - v25、 - v2xx、 - v50（デフォルトは - v25）が使用されていません。

L

label not defined

説明 .label 疑似命令に、必要な *symbol* パラメータがありません。

label required

説明 指摘された疑似命令にラベルがありません。

library not in archive format

説明 .mlib 疑似命令で指定されたファイルがアーカイブ・ファイルではありません。

処置 マクロ・ライブラリが、アセンブルされていないアセンブラ・ソース・ファイルであることを確認してください。また、マクロ名とメンバ名が同じであり、ファイルの拡張子が .asm であることも確認してください。

local macro variable is not a valid symbol

説明 .var 疑似命令のオペランドが有効なシンボルではありません。

処置 6-11 ページの 6.3.7 項を参照してください。

M

macro parameter is not a valid symbol

説明 マクロ・パラメータが有効な識別子ではありません。

処置 マクロ・パラメータについては、6-5 ページ以降の 6.3 節で説明しています。

maximum macro nesting level exceeded

説明 マクロの最大ネスト・レベルは 32 です。

maximum number of copy files exceeded

説明 .copy または .include ファイルの最大ネスト・レベルは 10 です。

.mexit directive encountered outside macro

説明 .mexit 疑似命令がマクロの外部で指定されています。この疑似命令はマクロの内部でのみ有効です。

missing .endif directive

説明　.if 疑似命令に対応する .endif がありません。

missing .endloop directive

説明　.loop 疑似命令に対応する .endloop がありません。

missing .endm directive

説明　.macro 疑似命令に対応する .endm 疑似命令がありません。

missing macro name

説明　.macro 疑似命令に名前がありません。

missing structure tag

説明　.tag 疑似命令にシンボル名がありません。

N

no include/copy files in macro or loop blocks

説明　.include および .copy 疑似命令は、マクロ (.macro/.endm) およびループ・ブロック (.loop/.break/.endloop) の中では使用できません。

no parameters for macro arguments

説明　引数を指定してマクロが呼び出されていますが、マクロ定義には対応するパラメータが見つかりません。

O

open ”(” expected

説明　組み込み関数が正しく使用されていないか、対応していない括弧が見つかりました。

open quote missing

処置　組み込み関数が正しく使用されていないか、または対応していない引用符が見つかりました。

operand is invalid for selected version

処置　指定されたオペランドは、選択されているターゲット・プロセッサでは無効です。

operand missing
処置　　必須オペランドが指定されていません。

overflow in floating - point constant
説明　　浮動小数点値が大き過ぎて表現できません。

P

pass1/pass2 operand conflict
説明　　シンボル・テーブル内のシンボルの値が、パス 1 とパス 2 で異なります。これはアセンブラの内部エラーです。このエラーが繰り返し発生する場合は、アセンブラが破壊されている可能性があります。

positive value required
説明　　正の値の代わりに、負またはゼロの値が使用されました。

R

redefinition of local substitution symbol
説明　　ローカル置換シンボルは1つのマクロで一度しか定義できません。

register required
説明　　この命令がオペランドとして要求するレジスタが指定されていません。

S

string required
説明　　二重引用符で囲まれた必須文字列が指定されていません。
処置　　二重引用符で囲んだ文字列を指定する必要があります。

substitution symbol stack overflow
説明　　ネストした置換シンボルの最大数は 10 です。

substitution symbol string too long
説明　　置換シンボルの最大長は 200 文字です。

symbol required
説明　　.global 疑似命令がオペランドとして要求するシンボルが指定されていません。

symbol used in both REF and DEF
説明　　REF で指定されたシンボルはすでに定義されています。

syntax error
説明　　式の記述形式に誤りがあります。

T

too many local substitution symbols

説明 ローカル置換シンボルの最大数は 32 です。

U

unbalanced symbol table entries

説明 アセンブラが、誤った .block のネストまたは不正な .func のネストを検出しました。

undefined structure member

説明 構造体参照表記 (a.b) によって参照されているシンボルが、.struct/.endstruct シーケンスで宣言されていません。

undefined structure tag

説明 .tag のオペランドが、.struct 疑似命令で定義されていません。

undefined substitution symbol

説明 置換シンボルのオペランドが、マクロ・パラメータとしても定義されておらず、.asg 疑似命令または .eval 疑似命令でも定義されていません。

undefined symbol

説明 整合定義式が必要な場所で、未定義のシンボルが使用されています。

underflow in floating - point constant

説明 浮動小数点値が小さ過ぎて表現できません。

unexpected .endif encountered

説明 .endif 疑似命令の前に .if 疑似命令がありません。

unexpected .endloop encountered

説明 .endloop 疑似命令の前に .loop 疑似命令がありません。

unexpected .endm directive encountered

説明 .endm 疑似命令の前に .macro 疑似命令がありませんでした。

****USER ERROR****

説明　　.emsg 疑似命令からの出力です。

****USER MESSAGE****

説明　　.mmsg 疑似命令からの出力です。

****USER WARNING****

説明　　.wmsg 疑似命令からの出力です。

V

value is out of range

説明　　指定した値が正当な範囲内にありません。

value truncated

説明　　指定の値を表記するには、指摘された命令が提供するよりも多くのビットが必要です。値は、指定のフィールドに収まるように切り捨てられましたが、切り捨て後の値が原因でプログラムが動作を誤る可能性があります。

処置　　もっと小さい値を指定するか、または指摘された命令のもっと多数のビットを指定可能なフォーマットを使用して下さい。

.var directive encountered outside macro

説明　　.var 疑似命令がマクロの外部に指定されています。この疑似命令はマクロ（.macro/.endm）の内部でのみ有効です。

version number changed

説明　　.versio疑似命令を使って、互換性のないターゲット・プロセッサ間でバージョンを変更しようとしました。

W

warning — block open at end of file

説明　　.block または .func 疑似命令に対応する .endblock または .endfunc が欠落しています。

warning — byte value truncated

説明　指定されたフィールドは 8 ビット・フィールドに収まりません。フィールドに収まるように、値が切り捨てられます。

warning - context switch in delayed sequence

説明　遅延シーケンス・コンテキスト変更の中にコンテキスト・スイッチがあります。

warning - delayed jump misses second word

説明　遅延シーケンス・コンテキスト・スイッチの 2 番目のワードの中に 2 ワード命令があります。番目のワードは分岐アドレスからフェッチされます。

warning — function .sym required before .func

説明　関数定義する .sym 疑似命令が .func 疑似命令の前に指定されていません。

warning — illegal relative branch

説明　異なるセクションへの分岐が要求されています。

warning — illegal version — default used

説明　指定されているプロセッサのバージョンが無効です。TMS320C2xがデフォルトとして使用されます。

warning — immediate operand not absolute

説明　提供されたイミディエイト・オペランドは絶対式ではありません。

処置　3-25 ページの表3 - 2 を参照して下さい。

warning — line truncated

説明　入力行の文字数が 200 文字を越えています。入力行の 201 番目以降の文字はすべて無視されます。

warning — null string defined

説明　文字列オペランドが必要な疑似命令に対して、空の文字列（長さが 0 の文字列）が定義されています。

warning — possible pipe conflict with norm

説明　NORM 命令の後に続く 2 ワードに間接アドレッシングが使用されています。これはパイプライン・コンフリクトの原因になることがあります。

warning — possible pipe hit at branch address

説明　　遅延シーケンス・コンテキスト変更時に NORM 命令が見つかりました。NORM命令はパイプライン・コンフリクトの原因になることがあります。

warning — possible pipe hit with arx access

説明　　メモリ・アドレスとして AR レジスタにアクセスしました。これはパイプライン・コンフリクトの原因になることがあります。

warning — register converted to immediate

説明　　オペランドとして定数が必要です。

warning — string length exceeds maximum limit

説明　　.title 疑似命令の最大長は 64 文字です。

warning — symbol truncated

説明　　シンボルの最大長は 8 文字です。アセンブラは余分な文字を無視します。

warning — trailing operand(s)

説明　　指摘された命令で、必要な数よりも多い（または少ない）オペランドが検出されました。

warning — unexpected .end found in macro

説明　　マクロを終了させるのに、.end 疑似命令が使用されました。

処置　　代わりに .endm マクロ疑似命令を使用してください。

warning — value out of range

説明　　指定した値が正当な範囲内にありません。

warning — value truncated

説明　　指定した式が大き過ぎて、オペコードまたは必要なビット数に収まりません。

付録 F

リンカ・エラー・メッセージ

リンカは、次のようなタイプのエラー・メッセージを発行します。

- ❑ 構文エラーおよびコマンド・エラー
- ❑ 割り振りエラー
- ❑ I/O エラー

この付録では、これら 3 つのタイプのエラーについて説明します。それぞれのエラーをカテゴリ別にアルファベット順で示します。以下のリストの中の (...) という記号は、エラーが発生したときにリンカが処理していたオブジェクトの名前を表します。ほとんどのエラー・メッセージには、問題点の「説明」と訂正のためにとれる「処置」を示してあります。エラー・メッセージ自体で問題点が十分に説明されている場合には、処置だけを示してあることもあります。説明を読めば（コードを検査し訂正するための）処置がすぐに分かる場合は、処置を省略してあります。

構文エラーおよびコマンド・エラー

これらのエラーは、リンカ疑似命令を誤って使用した場合、入力式を誤って使用した場合、または無効なオプションがある場合に発生します。すべての式の構文をチェックするとともに、入力した疑似命令が正しいかどうかチェックしてください。また、使用している各種のオプションを確認し、矛盾がないかチェックしてください。

A

absolute symbol (...) being redefined

説明　　絶対シンボルが再定義されています。これは許されません。

adding name (...) to multiple output sections

説明　　SECTIONS 疑似命令で入力セクションが 2 回指定されています。

ALIGN illegal in this context

説明　　シンボルの位置合わせが SECTIONS 疑似命令の外部で実行されています。

alignment for (...) must be a power of 2

説明　セクション位置合わせが 2 のベキ乗ではありませんでした。

処置　16 進数では、すべての 2 のベキ乗が、整数 1、2、4、または 8 とその後に続く単数または複数の 0 で構成されていることを確認してください。

alignment for (...) redefined

説明　1 つのセクションに複数の位置合わせが指定されています。

attempt to decrement "."

説明　. - = のような文は不当です。ドットへの代入は、ホールを作成する場合にのみ使用できます。

B

binding address for (...) redefined

説明　１つのセクションに複数のバインディング値が指定されています。

blocking for (...) must be a power of 2

説明　セクション・ブロッキングが 2 のベキ乗ではありません。

処置　16 進数では、すべての 2 のベキ乗が、整数 1、2、4、または 8 とその後に続く単数または複数の 0 で構成されていることを確認してください。

blocking for (...) redefined

説明　１つのセクションに複数のブロッキング値が指定されています。

C

- c requires fill value of 0 in .cinit (... overridden)

説明　.cinit テーブルは 0 で終了していなければなりません。したがって、.cinit セクションの埋め込み値は 0 でなければなりません。

cannot align a section within GROUP — (...) not aligned

説明　グループ内のセクションが個別位置合わせ用に指定されています。グループは、全体が 1 つの単位として取り扱われるので、グループをアドレスに位置合せしたりバインドしたりすることはできますが、グループを構成している各セクションを個別に取り扱うことはできません。

cannot resize (...), section has initialized definition in (...)

説明　初期化された入力セクション .stack または .heap が存在するため、リンカがセクションのサイズを変更できません。

cannot specify both binding and memory area for (...)

説明　バインディングとメモリの両方が指定されています。この 2 つは互いに排他的で、同時に指定することはできません。

処置　コードを特定のアドレスに置きたい場合は、バインディングだけを使用してください。

cannot specify a page for a section within a GROUP

説明　グループ内の特定のページに対してセクションが指定されています。グループは、全体が1つの単位として取り扱われるので、グループをメモリのページに対して指定することはできますが、グループを構成している各セクションを個別に取り扱うことはできません。

E

8-bit relocation out of range at (...) in section (...), file (...)

説明　'C25 についてのみ有効なエラー・メッセージです。8 ビット再配置のオーバーフローが検出されました。

処置　-l オプションを使ってアセンブラ・リスト・ファイルを作成してください。そして、上記の最初の 2 つのパラメータ、つまり 16 進アドレスとセクション・アドレスに対応する SPC をもつ行をリスト・ファイルで調べてください。

- e flag does not specify a legal symbol name (...)

説明　-e オプションにオペランドとして有効なシンボル名が指定されていません。

entry point other than _c_int00 specified

説明　-c または -cr オプションの場合のみ。_c_int0 以外のプログラム・エントリ・ポイントが指定されました。コンパイラの実行時規則では、_c_int0 が唯一のエントリ・ポイントであることが前提になっています。

entry point symbol (...) undefined

説明　-e オプションで指定されたシンボルが定義されていません。

errors in input - (...) not built

説明　前のエラーによって、出力ファイルが作成されませんでした。

F

fill value for (...) redefined

説明　1 つの出力セクションについて複数の埋め込み値が指定されています。このセクション定義により、個々のホールに異なる値を埋め込むことができます。

fill value redefined for memory area (...)

説明　1 つの出力セクションについて複数の埋め込み値が指定されています。このセクション定義により、個々のホールに異なる値を埋め込むことができます。

I

- i path too long (...)

説明　- i パスに指定できる最大文字数は 256 です。

illegal input character

説明　コマンド・ファイル内に制御文字などの認識されない文字があります。

illegal memory attributes for (...)

説明　属性が R、W、I、および X の組み合わせではありません。

illegal operator in expression

説明　式の演算子が正しいかを確認してください。

invalid path specified with - i flag

説明　- i オプション (フラグ) のオペランドが有効なファイルまたはパス名ではありません。

invalid value for - f option

説明　- f オプション (フラグ) の値が、2 バイトの定数ではありません。

invalid value for - heap option

説明　- heap オプション (フラグ) の値が、2 バイトの定数ではありません。

invalid value for - stack flag

説明　- stack オプション (フラグ) の値が、2 バイトの定数ではありません。

invalid value for - v flag

説明　- v オプション (フラグ) の値が、定数ではありません。

L

length redefined for memory area (...)

説明 MEMORY 疑似命令内の 1 つのメモリ領域に対して 2 つ以上の長さが指定されています。

linking files for incompatible targets (file ...)

説明 異なるターゲット・デバイス用にアセンブルされたオブジェクト・コードをいっしょにリンクすることはできません。

load address for UNION member ignored

説明 UNION 文についてロード・アドレスが指定されています。セクションは実行されますが、共用体としてはロードされません。

load address for uninitialized section (...) ignored

説明 初期化されないセクションについてロード・アドレスが指定されています。初期化されないセクションにはロード・アドレスはありません。実行アドレスだけです。

load allocation required for initialized UNION member (...)

説明 共用体内部の初期化された 1 セクションについてロード・アドレスが指定されています。UNION は実行時の割り振りのみを参照します。同じ共用体内部のすべてのセクションについて、個別にロード・アドレスを指定する必要があります。

M

- m flag does not specify a valid filename

説明 出力マップ・ファイルを書き込むファイルに、有効なファイル名を指定しませんでした。

memory area for (...) redefined

説明 1 つの出力セクションについて複数の名前付きメモリ割り振りが指定されています。

memory page for (...) redefined

説明 1 つのセクションについて複数のページ割り振りが指定されています。

memory attributes redefined for (...)

説明　1 つの出力セクションについてメモリ属性が 2 組以上指定されています。

missing filename on - l

説明　-l（小文字の L）オプションにファイル名オペランドが指定されていません。

misuse of "." symbol in assignment instruction

説明　SECTION疑似命令の外部にある代入文で「.」記号が使用されています。

N

nesting exceeded with file (...)

説明　16 レベルを越えるコマンド・ファイルのネストが指定されています。

no allocation allowed for uninitialized UNION member

説明　共用体内部の初期化されない 1 セクションについてロード・アドレスが指定されています。共用体内の初期化されないセクションは、実行アドレスを UNION 文から入手し、ロード・アドレスは持ちません。したがって、メンバについてのロード割り振りは無効です。

no allocation allowed within a GROUP - allocation for section (...) ignored

説明　グループ内の 1 つのセクションが個別割り振り用に指定されています。グループは、全体が 1 つの単位として取り扱われるので、グループをアドレスに位置合せしたりバインドしたりすることはできますが、グループを構成している各セクションを個別に取り扱うことはできません。

no input files

説明　COFF ファイルが指定されていません。少なくとも 1 つは入力 COFF ファイルがないと、リンカは動作できません。

no load address specified for (...); using run address

説明　初期化されたセクションについてのロード・アドレスが指定されていません。初期化されたセクションに実行アドレスだけが指定されている場合、そのセクションは実行時もロード時も同じアドレスに割り振られます。

O

- o flag does specify a valid file name : *string*

説明　ファイル名は、オペレーティング・システムのファイル命名規則に従っていなければなりません。

origin missing for memory area (...)

説明　MEMORY 疑似命令に起点（origin）が指定されていません。起点は、メモリ範囲の開始アドレスを指定するものです。

origin redefined for memory area (...)

説明　　メモリ領域の起点が 2 回以上指定されています。各メモリ領域に指定できる起点は 1 つだけです。

R

- r incompatible with - s (- s ignored)

説明　　- r オプションと - s オプションの両方が使用されています。- s オプションは再配置情報を除去し、- rオプションは再配置可能オブジェクト・ファイルを要求するものなので、この 2 つのオプションは互いに対立します。

S

section (...) not built

説明　　最も可能性が高いのは、SECTIONS 疑似命令に構文エラーがあることです。

statement ignored

説明　　式に構文エラーがあります。

symbol referencing errors — (...) not built

説明　　シンボルの参照を解決できませんでした。このため、オブジェクト・モジュールを作成できませんでした。

symbol (...) from file (...) being redefined

説明　　代入文で、定義済みのシンボルが再定義されています。

syntax error: scanned line = (...)

説明　　指摘されている行番号に構文エラーが検出されました。

T

too many arguments - use a command file

説明　　コマンド行で、またはプロンプトへの応答として、10 個を越える引数を使用しました。

too many - i options, 7 allowed

処置　　7 つを越える - i オプションを使用しました。追加の探索ディレクトリは、C_DIR または A_DIR 環境変数を使って指定することができます。

type flags for (...) redefined

説明　　1つのセクションについて複数のセクション・タイプが指定されています。タイプ COPY にはタイプ DSECT のすべての属性が備わっているので、DSECT を個別に指定する必要はないことに注意してください。

type flags not allowed for GROUP or UNION

説明　　グループまたは共用体の内部のセクションについて、タイプが指定されています。特殊なセクション・タイプは、個別セクションのみに適用されるものです。

U

- u does not specify a legal symbol name

説明　　-u オプションには、リンクするファイルのいずれかで正しいシンボル名が指定されていません。

unexpected EOF (end of file)

説明　　リンカ・コマンド・ファイルに構文エラーがあります。

undefined symbol in expression

説明　　代入文に未定義のシンボルが含まれています。

Z

zero or missing length for memory area (...)

説明　　MEMORY 疑似命令で定義したメモリ範囲の長さは、ゼロ以外でなければなりません。

割り振りエラー

割り振りエラーのメッセージは、リンクの割り振りフェーズで発行されます。これらのメッセージが発行されるのは、一般的には、あるセクションまたはグループが所定のアドレスに収まらない場合や、MEMORY 疑似命令と SECTIONS 疑似命令が矛盾している場合です。リンカ・コマンド・ファイルを使用している場合は、MEMORY 疑似命令と SECTION疑似命令をチェックして、セクションのオーバーラップが起こったりセクションが未構成メモリに置かれたりしないようにするための十分な余裕があることを確認してください。

B

binding address (...) for section (...) is outside all memory on page (...)

説明 一部のセクションが、MEMORY 疑似命令で構成したメモリの範囲内に収まっていません。

処置 リンカ・コマンド・ファイルを使用している場合は、MEMORY および SECTIONS 疑似命令で、未構成メモリ内にセクションが配置されないように十分な余裕が設けられているかチェックしてください。

binding address (...) for section (...) overlays (...) at (...)

説明 2つのセクションがオーバーラップしているので、割り振れません。

処置 リンカ・コマンド・ファイルを使用している場合は、MEMORY および SECTIONS 疑似命令で、未構成のメモリ内にセクションが配置されないように十分な余裕が設けられているかチェックしてください。

binding address (...) incompatible with alignment for section (...)

説明 バインディング・アドレスが、.align 疑似命令または直前のリンクによるセクションの位置合せ要件に違反しています。

C

can’t allocate output section, (...)

説明 既存の構成メモリ領域にセクションを保持できる大きさがないので、セクションを割り振れません。

処置 リンカ・コマンド・ファイルを使用している場合は、MEMORY 疑似命令と SECTION 疑似命令をチェックして、未構成メモリにセクションが配置されないような十分な余裕があることを確認してください。

I

internal symbol (...) redefined in file (...)

説明 内部シンボルが 2 番目のファイルで再定義されています。リンカは 2 番目の定義を無視します。

L

load address for uninitialized section (...) ignored

説明 初期化されないセクションについてロード・アドレスが指定されています。初期化されないセクションにはロード・アドレスはありません。実行アドレスだけです。

M

memory types (...) and (...) on page (...) overlap

説明 同じページで複数のメモリ範囲がオーバーラップしてはいけません。

処置 リンカ・コマンド・ファイルを使用している場合は、MEMORY および SECTIONS 疑似命令で、未構成メモリ内にセクションが配置されないように十分な余裕が設けられているかチェックしてください。

N

no load address specified for (...); using run address

説明 初期化されたセクションにロード・アドレスが指定されていません。初期化されたセクションが実行アドレスのみを持っている場合、そのセクションは実行時とロード時で同じアドレスに割り振られます。

O

output file (...) not executable

説明 作成された出力ファイルに、別のエラーのために生じる未解決のシンボルまたはその他の問題が含まれている可能性があります。この状態は致命的エラーではありません。

S

section (...) at address (...) overlays previously allocated section (...) at address

説明 2 つのセクションがオーバーラップしているので、割り振れません。

処置 リンカ・コマンド・ファイルを使用している場合は、MEMORY および SECTIONS 疑似命令で、セクションがオーバーラップしないように十分な余裕が設けられているかチェックしてください。

section (...), bound at address (...), won't fit into page (...) of configured memory

説明　既存の構成メモリ領域にセクションを保持できる大きさがないので、セクションを割り振れません。

処置　リンカ・コマンド・ファイルを使用している場合は、MEMORY 疑似命令と SECTION 疑似命令をチェックして、未構成メモリにセクションが配置されないような十分な余裕があることを確認してください。

section (...) enters unconfigured memory at address (...)

説明　既存の構成メモリ領域にセクションを保持できる大きさがないので、セクションを割り振れません。

処置　リンカ・コマンド・ファイルを使用している場合は、MEMORY 疑似命令と SECTION 疑似命令をチェックして、未構成メモリにセクションが配置されないような十分な余裕があることを確認してください。

section (...) in file (...) is too big

説明　既存の構成メモリ領域にセクションを保持できる大きさがないので、セクションを割り振れません。

処置　リンカ・コマンド・ファイルを使用している場合は、MEMORY および SECTIONS 疑似命令で、未構成メモリ内にセクションが配置されないように十分な余裕が設けられているかチェックしてください。

U

undefined symbol (...) first referenced in file (...)

説明　参照シンボルが定義されていないか、または -r オプションが使用されていません。-r オプションを使用していない場合、参照されているすべてのシンボルは定義されていなければなりません。この条件に該当する場合、実行可能出力ファイルは作成されません。

処置　-r オプションを使用してリンクするか、シンボルを定義してください。

I/O エラーおよび内部オーバーフロー・エラー

以下のエラー・メッセージは、入力ファイルが壊れている、存在しない、または読み取り不能であること、あるいは出力ファイルのオープンまたは書き込みができないことを示します。このカテゴリのメッセージで、リンカのメモリ不足またはテーブル・スペースの不足が指摘されることもあります。

C

cannot complete output file (...), write error

説明　通常は、ファイル・システムでスペースが不足していることを示します。

cannot create output file (...)

説明　通常は、ファイル名が不当であることを示します。

処置　綴り、パス名、環境変数などをチェックして下さい。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指定しなければなりません。

can’t open (...)

説明　指定したファイルが存在しません。

処置　綴り、パス名、環境変数などをチェックしてください。

can’t read (...)

説明　ファイルが壊れている可能性があります。

処置　入力ファイルが壊れている場合は、再アセンブルしてください。

can’t seek (...)

説明　ファイルが壊れている可能性があります。

処置　入力ファイルが壊れている場合は、再アセンブルしてください。

can’t write (...)

説明　ディスクが一杯か保護されています。

処置　ディスクの空き容量および保護をチェックしてください。

could not create map file (...)

説明　通常は、ファイル名が不当であることを示します。

処置　綴り、パス名、環境変数などをチェックしてください。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指定しなければなりません。

F

fail to copy (...)

説明　　ファイルが壊れている可能性があります。

処置　　入力ファイルが壊れている場合は、再アセンブルしてください。

fail to read (...)

説明　　ファイルが壊れている可能性があります。

処置　　入力ファイルが壊れている場合は、再アセンブルしてください。

fail to seek (...)

説明　　ファイルが壊れている可能性があります。

処置　　入力ファイルが壊れている場合は、再アセンブルしてください。

fail to skip (...)

説明　　ファイルが壊れている可能性があります。

処置　　入力ファイルが壊れている場合は、再アセンブルしてください。

fail to write (...)

説明　　ディスクが一杯か保護されています。

処置　　ディスクの空き容量および保護をチェックしてください。

file (...) has no relocation information

説明　　-r を指定してリンクされていないファイルを再リンクしようとしました。

file (...) is of unknown type, magic number = (...)

説明　　このバイナリ入力ファイルは COFF ファイルではありません。

I

illegal relocation type (...) found in section(s) of file (...)

説明　　バイナリ・ファイルが壊れています。

internal error : aux table overflow

説明　　このリンカには内部エラーがあります。

invalid archive size for file (...)

説明　　アーカイブ・ファイルが壊れています。

I/O error on output file (...)

説明　　ディスクが一杯か保護されています。

処置　　ディスクの空き容量および保護をチェックしてください。

L

library (...) member (...) has no relocation information

説明　　ライブラリ・メンバに再配置情報がありません。ライブラリ・メンバに再配置情報がないという状態は可能ですが、これは、リンク時に他のファイル内の未解決の参照を満たすことができないことを意味します。

line number entry found for absolute symbol

説明　　入力ファイルが壊れている可能性があります。

処置　　入力ファイルが壊れている場合は、再アセンブルしてください。

M

memory allocation failure

説明　　リンカが作業用に利用するメモリを確保できません。

N

no symbol map produced - not enough memory

説明　　使用可能なメモリが不十分で、シンボル・リストが作成できません。これは非致命的な状況です。マップ・ファイル内にシンボル・リストは生成されません。

R

relocation entries out of order in section (...) of file (...)

説明　　入力ファイルが壊れている可能性があります。

処置　　入力ファイルが壊れている場合は、再アセンブルしてください。

relocation symbol not found: index (...), section (...), file (...)

説明　　入力ファイルが壊れている可能性があります。

処置　　入力ファイルが壊れている場合は、再アセンブルしてください。

S

section (...) not found

説明　　SECTIONS 疑似命令に指定された入力セクションが入力ファイルの中に見つかりませんでした。

sections .text, .data, or .bss not found

説明　　COFF の任意選択ファイル・ヘッダの中の情報が壊れている可能性があります。

処置　　影響を受けるファイルの再アセンブルと再リンクをしてみてください。

seek to (...) failed

説明　　入力ファイルが壊れている可能性があります。

処置　　入力ファイルが壊れている場合は、再アセンブルしてください。

付録 G

用語集

B

.bss: デフォルトの COFF セクションの 1 つ。.bss 疑似命令を使って、メモリ・マップに一定量の空間を予約して、後でそこへデータを格納することができます。通常は、.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 TMS320C2x/C2xx/C5x 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。

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 疑似命令を使うと、コードを .data セクションにアセンブルすることができます。

G

GROUP: SECTIONS 疑似命令のオプションの 1 つで、指定された出力セクションを連続して（グループとして）割り当てます。

H

Hex 変換ユーティリティ: COFF オブジェクト・ファイルを EPROM プログラマにロードできるようにいくつかの標準 ASCII 16 進フォーマットの 1 つに変換するためのユーティリティ。

R

RAM モデル: リンカが C コードをリンクするときに使う自動初期化モデル。 - cr オプションを使ってリンカを呼び出すと、リンカは RAM モデルを使用します。RAM モデルを使うと、実行時ではなくロード時に変数を初期化することができます。

ROM モデル: リンカが C コードをリンクするときに使う自動初期化モデル。 - c オプションを使ってリンカを呼び出すと、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 1 要素。各セクションに 1 つの SPC があります。

static: スコープが 1 つの関数またはプログラムに限定されている変数の一種。静的変数の値は、それが使わ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、実行可能コードを持った初期化されたセクションです。.text 疑似命令を使うと、コードを .text セクションにアセンブルすることができます。

U

UNION: SECTIONS 疑似命令のオプションの 1 つで、複数のセクションに同じ実行アドレスを割り振ります。

unsigned: 実際の符号にかかわらず常に正の数として取り扱われる値の一種。

あ

アーカイブ・ライブラリ: 1 つのファイルにグループ化されたファイルの集合。

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ファイルにグループ化するためのソフトウェア・プログラム。アーカイバを使うと、新しいメンバの追加だけでなく、アーカイブ・ライブラリの中のファイルを削除、抽出、置換することができます。

アセンブラ: アセンブリ言語命令、疑似命令、およびマクロ疑似命令を含むソース・ファイルから機械語のプログラムを作成するソフトウェア・プログラム。アセンブラはシンボリックな命令コードを絶対的な命令コードに代え、シンボリックなアドレスを絶対アドレスまたは再配置可能なコードに代えます。

アセンブル時定数: .set 疑似命令または .equ 疑似命令を使っ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 のベキ乗。位置合わせの指定には、リンカ疑似命令 SECTIONS を使用します。

インクリメンタル・リンク: すでにリンクされているファイルのリンク。

え

エミュレータ: TMS320 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: 同じアドレス範囲に別のメモリ・セクションとしてマップされた物理メモリの中の部分。どの範囲をアクティブとするかは、ハードウェア・スイッチにより決定します。

オブジェクト・ファイル: 機械語オブジェクト・コードを含む、アセンブルまたはリンクされたファイル。

オブジェクト・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、Tektronix フォーマット、TI - Tagged フォーマット、または Motorola - S 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・ファイルが入ったアーカイブ・ライブラリ。

オプション: ソフトウェア・ツールの起動時に、追加の機能、または特定の機能を実行させるために使うコマンド・パラメータ。

オペランド: 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、別のプログラム・モジュールで定義されているシンボル。

き

記憶クラス: シンボル・テーブルのエントリで、シンボルへのアクセス方法を示します。

疑似命令: 特別な目的を持ったコマンドで、ソフトウェア・ツールの動作、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に対する用語）。

共通オブジェクト・ファイル・フォーマット (COFF): バイナリ・オブジェクト・ファイル・フォーマットの一種。セクションの概念を採用することにより、モジュラ・プログラミングを促進できます。

行番号エントリ: COFF 出力モジュールにあるエントリの 1 つで、アセンブリ・コードの各行をそれを作成したオリジナルの C ソース・ファイルにマップする役割を果たします。

共用体: 異なった型とサイズのオブジェクトを保持することのできる変数。

く

グローバル: 次の何れかの条件を満たすシンボルの一種: 1) 現行のモジュールで定義され、別のモジュールでアクセスが可能、または2) 他のモジュールで定義されているが現行のモジュールでアクセスが可能。

クロス・リファレンス・リスト: アセンブラが作成する出力ファイルで、定義されたシンボル、シンボルを定義した行、シンボルを参照している行、およびその最終値が掲載されています。

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（型および関数の定義など）をデバッグ・ツールが使用できるように保持するコンパイラの機能。

構成メモリ: リンカが割り振り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・オプションを保持し、リンカ用の入力ファイルを指定するファイル。

コメント: ソース・ファイルを文書化したり、読みやすくするためのソース文（または、その一部）。コメントは、コンパイル、アセンブル、およびリンクされません。つまり、オブジェクト・ファイルには何の効果も及ぼしません。

さ

再配置: シンボルのアドレスが変更されたときに、リンカがそのシンボルに対するすべての参照を調整すること。

し

式: 算術演算子で区切られた定数、シンボル、または一連の定数とシンボル。

実行アドレス: セクションを実行するアドレス。

実行可能モジュール: リンクされた、TMS320 システムで実行することができるオブジェクト・ファイル。

自動初期化: プログラムの実行を開始する前に、C のグローバル変数（.cinit セクションに保持されている）を初期化すること。

シミュレータ: TMS320 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、最終的な割り振り済みセクション。

出力モジュール: ターゲット・システム上にダウンロードして実行することのできる、リンクされた実行可能オブジェクト・ファイル。

条件付き処理: 指定された式の評価に基づいて、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 疑似命令で構成されます。

初期化されないセクション: メモリ・マップ内に空間は確保されるが実際の内容を持たない COFF セクション。このセクションは .bss と .usect 疑似命令で構成されます。

シンボル: アドレスまたは値を表す英数字の文字列。

シンボリック・デバッグ: シミュレータやエミュレータなどのデバッグ・ツールがシンボルを使うことができるようにシンボルの情報を保持するソフトウェア・ツールの機能。

シンボル・テーブル: COFF オブジェクト・ファイルの中の、ファイルが定義して使用するシンボルについての情報を記述した部分。

せ

整合定義式: すでに定義されているシンボルまたはアセンブル時間定数のみを使った式。

セクション: コードまたはデータの再配置可能なブロックで、最終的には TMS320 メモリ・マップ内の連続したスペースに入れられます。

セクション・プログラム・カウンタ: SPC を参照のこと。

セクション・ヘッダ: COFF オブジェクト・ファイルの中の、そのファイルの中のセクションに関する情報を含んだ部分。各セクションに専用のヘッダがあり、そこにそのセクションの開始アドレスやサイズなどの情報が示されています。

絶対アドレス: TMS320 のメモリ位置に永久に割り当てられたアドレス。

絶対セクション: .asect 疑似命令によって定義された名前付きセクション。.label によって定義されたアドレスを例外として、絶対セクション内のすべてのアドレスは絶対的なものです。

絶対リスタ: 絶対アドレスを含むアセンブラ・リストを作成するために使用できるデバッグ・ツール

そ

ソース・ファイル: C コードまたは TMS320 アセンブリ言語コードを含んだファイルで、これをコンパイルまたはアセンブルしてオブジェクト・ファイルを作成します。

た

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選択の型名。

ターゲット・メモリ: 実行可能なオブジェクト・コードをロードすることのできる物理メモリ。

て

定数: オペランドとして使うことができる数値。

な

名前付きセクション: .sect 疑似命令を使って定義された初期化されたセクション、または、.usect 疑似命令を使って命名された初期化されないセクション。

生データ: 出力セクション内の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: アセンブラが機械コードに変換する命令名。

入力セクション: オブジェクト・ファイルの 1 セクションで、実行可能モジュールにリンクされる部分。

任意選択ヘッダー: COFF オブジェクト・ファイルの中の、リンカがダウンロード時に再配置を行うのに使用する部分。

は

バインディング: 出力セクションあるいはシンボルに異なるアドレスを指定する際のプロセス。

ふ

ファイル・ヘッダー: COFF オブジェクト・ファイルの中の、オブジェクト・ファイルについての一般情報（セクション・ヘッダーの数、オブジェクト・ファイルをダウンロードすることのできるシステムのタイプ、シンボル・テーブルにあるシンボルの数、シンボル・テーブルの開始アドレスなど）を記述した部分。

フィールド: TMS320 においては、ソフトウェアで構成可能なデータ型で、長さを 1〜16 ビットの範囲で自由にプログラミングすることができます。

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

補足エントリ: シンボルがそのシンボル・テーブルに持っている追加のエントリで、そのシンボルについての追加情報（シンボルがファイル名か、セクション名か、関数名か、など）が含まれます。

ホール: 出力セクションを構成する入力セクションと入力セクションの間にあり、実際のコードやデータを含まない部分。

ま

マクロ: ユーザが定義した、命令として使うことができるルーチン。

マクロ・ライブラリ: マクロが入ったアーカイブ・ライブラリ。ライブラリの中の各ファイルは、最低 1 つのマクロを含み、そのファイル名はそのファイルが定義し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。

マクロ定義: マクロを構成する名前とコードを定義するソース文のブロック。

マクロ展開: マクロ呼び出しで展開され、続いて、アセンブルされるソース文。

マクロ呼び出し: マクロを起動すること。

マジック・ナンバ: COFF ファイルのヘッダ・エントリの 1 つで、TMS320 で実行することのできるモジュールとしてオブジェクト・ファイルを識別するためのもの。

マップ・ファイル: リンカによって作成される出力ファイルで、メモリ構成、セクション構成、セクションの割り振り、さらに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ず、コードまたはデータとともにロードできないメモリ。

め

メモリ・マップ: TMS320 ターゲット・システムのメモリ空間のマップで、複数の機能ブロックに区画分けされています。

メンバ: 構造体、共用体、アーカイブ、または列挙型の要素または変数。

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ため、文字列テーブルが使われます）。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で、マクロが呼び出される度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で、その文のアドレスに対応します。

り

リスト・ファイル: アセンブラによって作成される出力ファイルで、ソース文、その行番号、SPC への効果を掲載します。

リンカ: オブジェクト・ファイルを結合して、TMS320 システム・メモリに割り当てて TMS320 で実行できるオブジェクト・モジュールを作成するためのソフトウェア・ツール。

ろ

ローダ: 実行可能モジュールを TMS320 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・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プロセス。

索引

記号

数字

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

あ

い

う

え

お

か

き

く

こ

さ

し

せ

そ

た

ち

つ

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	
大阪営業所	〒530-6026	大阪市北区天満橋1丁目8番30号　OAPオフィスタワー26階	06（6356）4500（代表）
名古屋営業所	〒460-0003	名古屋市中区錦2丁目4番3号　錦パークビル10階	052（232）5601（代表）
松本営業所	〒390-0815	長野県松本市深志1丁目2番11号　松本昭和ビル6階	0263（33）1060（代表）
立川営業所	〒190-0012	東京都立川市曙町2丁目36番2号　ファーレ立川センタースクエア10階	042（527）6760（代表）
京都営業所	〒600-8216	京都市下京区塩小路通西洞院東入東塩小路町843-2　日本生命京都ヤサカビル5階	075（341）7713（代表）
大宮営業所	〒330-0802	埼玉県大宮市宮町1丁目38番1号　野村不動産大宮共同ビル4階	048（647）2622（代表）
米国駐在事務所	TEXAS INSTRUMENTS Inc.,8505 Forest Lane Dallas,75243,U.S.A（TI Japan Ltd.,U.S.A.Sales Office）		1-972-480-7464

工　場　　埼玉県・鳩ヶ谷市　大分県・日出町　茨城県・美浦村　静岡県・小山町（制御機器事業部）

株式会社テキサス・インスツルメンツ筑波研究開発センター　茨城県・つくば市

■お問い合わせ先

プロダクト・インフォメーション・センター（PIC）　0120-81-0026　FAX 0120-81-0036

E-mail: pic-japan@ti.com

TMS320C1x/C2x/C2xx/C5x
アセンブリ言語ツール
ユーザーズ・マニュアル

第2版 1999**年** 4**月**
第1版 1996**年** 11**月**

発行所　**日本テキサス・インスツルメンツ株式会社**
東京都新宿区西新宿6 - 24 - 1 (**西新宿三井ビルディング**)
〒160 - 8366

TEXAS INSTRUMENTS

Printed in Japan
'99.04

日本テキサス・インスツルメンツ株式会社

SCJ2223