TEXAS INSTRUMENTS

TMS320C54x
アセンブリ言語ツール

ユーザーズ・マニュアル

2001 *Digital Signal Processing Products*

SPRU102 翻訳版

JAJU016 WAS SPRU366A

TMS320C54x
アセンブリ言語ツール
ユーザーズ・マニュアル

2001 年 8 月

TEXAS INSTRUMENTS

ご注意

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

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

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

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

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

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

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

以上

IN-0104

弊社半導体製品の取り扱い・保管について

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

1．静電気

- **素手で半導体製品単体を触らないこと。どうしても触る必要がある場合は、リストストラップ等で人体からアースをとり、導電性手袋等をして取り扱うこと。**
- **弊社出荷梱包単位（外装から取り出された内装及び個装）又は製品単品で取り扱いを行う場合は、接地された導電性のテーブル上で（導電性マットにアースをとったもの等）、アースをした作業者が行うこと。また、コンテナ等も、導電性のものを使うこと。**
- **マウンタやはんだ付け設備等、半導体の実装に関わる全ての装置類は、静電気の帯電を防止する措置を施すこと。**
- **前記のリストストラップ・導電性手袋・テーブル表面及び実装装置類の接地等の静電気帯電防止措置は、常に管理されその機能が確認されていること。**

2．温・湿度環境

- **温度：0～40℃、相対湿度：40～85%で保管・輸送及び取り扱いを行うこと。（但し、結露しないこと。）**
- **直射日光があたる状態で保管・輸送しないこと。**

3．防湿梱包

- **防湿梱包品は、開封後は個別推奨保管環境及び期間に従い基板実装すること。**

4．機械的衝撃

- **梱包品（外装、内装、個装）及び製品単品を落下させたり、衝撃を与えないこと。**

5．熱衝撃

- **はんだ付け時は、最低限260℃以上の高温状態に、10秒以上さらさないこと。（個別推奨条件がある時はそれに従うこと。）**

6．汚染

- **はんだ付け性を損なう、又はアルミ配線腐食の原因となるような汚染物質（硫黄、塩素等ハロゲン）のある環境で保管・輸送しないこと。**
- **はんだ付け後は十分にフラックスの洗浄を行うこと。（不純物含有率が一定以下に保証された無洗浄タイプのフラックスは除く。）**

以上

最初にお読みください

本書について

本書は、以下のアセンブリ言語ツールの使用方法について説明しています。

- ❏ 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

本書のご利用方法

本書の目的は、特に TMS320C54x DSP 用に作られた弊社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。

- ❏ **「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、TMS320C54x 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。アセンブラとリンカを使用する前に、第 2 章「共通オブジェクト・ファイル・フォーマットの概要」をお読みください。
- ❏ **「アセンブラの説明」**では、アセンブラの使用方法について詳しく説明します。具体的には、アセンブラの起動方法、ソース文のフォーマット、有効な定数と式、アセンブラ出力、およびアセンブラ疑似命令について説明します。また、マクロ要素についても説明します。
- ❏ **「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・ファイルの作成に役立つ、アセンブラに付属するツールについて、1 つずつ詳細に説明します。たとえば、第 7 章では、リンカの起動方法、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 10 章では、Hex 変換ユーティリティの使用方法について説明します。

❑ **「参考資料」**では、補足的な説明をします。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。C/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、Hex 変換ユーティリティの例、アセンブラおよびリンカのエラー・メッセージ、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、タイプライタの活字に似た特殊な活字 (special typeface) で示してあります。例は、強調のため、ボールド **(bold version)** で示してあります。対話表示についても、ユーザが入力するコマンドとシステムが表示する項目 (プロンプト、コマンド出力、エラー･メッセージなど) との区別のために、ボールド **(bold version)** で示してあります。

プログラム・リストの例を以下に示します。

```
2 000001   002f             x       .byte   47
3 000002   0032             z       .byte   50
4 000003                            .text
```

❑ 構文の記述、命令、コマンド、疑似命令はボールド (**bold**)、パラメータはイタリック体 (*italics*) で示します。構文で**ボールド**の部分は、その表記通り入力する箇所です。構文でイタリック体の部分は、入力する情報の種類を示しています。コマンド行で入力する構文は、以下のように縁取りのある枠囲みの中に示します。

abs500 *filename*

abs500 はコマンドです。このコマンドは、絶対リスタを起動し、*filename* により示されるパラメータを 1 つとります。絶対リスタを起動するときには、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、括弧の内側の情報を指定します。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。

hex500 [*–options*] *filename*

hex500 コマンドには 2 つのパラ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* は複数形で書かれているので、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。

❑ アセンブラ文の構文では、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の指定が**任意**の場合には、通常は表記されません。パラメータの指定が**必須**の場合には、以下の例に示すようにアミ掛け枠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、およびパラメータを 1 カラム目から始めることはできません。

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

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 (ワード) を示す *size in words* 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。*blocking flag* と *alignment flag* は任意指定です。指定する場合は、コンマで区切る必要があります。

❑ 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 が 1 カラム目から始まっていないことに注意してください。

.byte $value_1$ **[,** ... **,** $value_n$**]**

この構文は、.byte には少なくとも 1 つの value パラメータが必要であることを示していますが、コンマで区切って複数の value パラメータを指定することもできます。

❑ 本書で使用するその他の記号と略語を以下に示します。

記号	定義	記号	定義
AR0 ～ AR7	補助レジスタ 0 ～ 7	PC	プログラム・カウンタ・レジスタ
B、b	接尾部 — 2 進数	Q、q	接尾部 — 8 進数
H、h	接尾部 — 16 進数	SP	スタック・ポインタ・レジスタ
LSB	最下位ビット	ST	ステータス・レジスタ
MSB	最上位ビット		

❑ 本書では、サポートされた全ての C54x デバイスの総称として 'C54x を使います。

当社発行の関連文献

以下の文献は、TMS320C54x および関連サポート・ツールの解説書であり、当社から刊行しています。当社の刊行書を入手するには、インターネットよりプロダクト・インフォメーション・センター (PIC) の URL (http://www.tij.co.jp/pic/) にアクセスの上、E−mail もしくは FAX でお問い合わせください。

TMS320C54x DSP Reference Set は、5 巻で構成されており、文献番号 SPRU210 でまとめて発注できます。1 巻だけ発注する場合は、該当の文献番号をお知らせください。

TMS320C54x DSP リファレンス・セット、Volume 1: CPU およびペリフェラル (文献番号 SCJ 2287) は、TMS320C54x™ 16 ビット固定小数点汎用ディジタル・シグナル・プロセッサについて解説しています。このプロセッサのアーキテクチャ、内部レジスタ構造、データおよびプログラムのアドレッシング、命令パイプライン、およびオンチップ・ペリフェラルについて記載しています。また、開発サポート情報、パーツ・リスト、XDS510™ エミュレータを使用する際のデザイン上の考慮事項も含まれています。

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

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

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

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

TMS320C54x 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 SPRU367) は、TMS320C54x™ C コンパイラについて解説しています。この C コンパイラは、ANSI 標準の C ソース・コードを受け入れて、TMS320C54x 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。

Code Composer Studio Getting Started マニュアル (文献番号 SPRU509) は、Code Composer Studio が提供する開発環境を利用してアプリケーションのプログラミングを始める基本的な手順について解説しています。

商標

目次

1 はじめに 1-1
ソフトウェア開発ツールの概要を記述しています。

1.1 ソフトウェア開発ツールの概要 1-2
1.2 各ツールの説明 1-3

2 共通オブジェクト・ファイル・フォーマットの概要 2-1
セクションの基本的な COFF 概念を説明し、セクションを使用すると、アセンブラとリンカをいかに効率よく使用できるかについて説明します。共通オブジェクト・ファイル・フォーマット (COFF) は、ツールで使用するオブジェクト・ファイル・フォーマットです。

2.1 COFF ファイルのタイプ 2-2
2.2 セクション 2-3
2.3 アセンブラによるセクションの処理 2-5
2.3.1 初期化されないセクション 2-5
2.3.2 初期化されたセクション 2-7
2.3.3 名前付きセクション 2-8
2.3.4 サブセクション 2-9
2.3.5 セクション・プログラム・カウンタ 2-9
2.3.6 セクション疑似命令の使用例 2-10
2.4 リンカによるセクションの処理 2-13
2.4.1 デフォルトのメモリ割り当て 2-14
2.4.2 セクションのメモリ・マップへの配置 2-15
2.5 再配置 2-16
2.5.1 再配置に対するメッセージの発行 2-17
2.6 実行時の再配置 2-19
2.7 プログラムのロード 2-20
2.8 COFF ファイルで使用されるシンボル 2-21
2.8.1 外部シンボル 2-21
2.8.2 シンボル・テーブル 2-22

3 アセンブラの説明 3-1
アセンブラの起動方法、ソース文フォーマット、有効な定数と式、およびアセンブラ出力について説明します。

3.1 アセンブラの概要 3-2
3.2 アセンブラと開発フロー 3-3
3.3 アセンブラの起動方法 3-4

3.4 アセンブラ入力のための代替ファイルと代替ディレクトリの命名方法 3-8
3.4.1 –i アセンブラ・オプションの使用方法 3-8
3.4.2 C54X_A_DIR および A_DIR 環境変数の使用方法 3-9
3.5 ソース文のフォーマット 3-11
3.5.1 ソース文の構文 3-11
3.5.2 ラベル・フィールド 3-12
3.5.3 ニーモニック命令フィールド 3-12
3.5.4 代数表記命令フィールド 3-14
3.5.5 コメント・フィールド 3-14
3.6 定数 3-15
3.6.1 2 進整数 3-15
3.6.2 8 進整数 3-15
3.6.3 10 進整数 3-16
3.6.4 16 進整数 3-16
3.6.5 文字定数 3-16
3.6.6 浮動小数点定数 3-17
3.7 文字列 3-18
3.8 シンボル 3-19
3.8.1 ラベル 3-19
3.8.2 シンボル定数 3-19
3.8.3 シンボル定数の定義（–d オプション） 3-20
3.8.4 事前定義されたシンボル定数 3-20
3.8.5 置換シンボル 3-21
3.8.6 ローカル・ラベル 3-22
3.9 式 3-25
3.9.1 演算子 3-26
3.9.2 式のオーバーフローとアンダーフロー 3-26
3.9.3 整合定義式 3-27
3.9.4 条件式 3-27
3.9.5 再配置可能なシンボルと有効な式 3-28
3.10 ビルトイン関数 3-30
3.11 値を拡張プログラム・メモリへロードする方法 3-32
3.12 ソース・リスト 3-33
3.13 クロスリファレンス・リスト 3-37

4 アセンブラ疑似命令 4-1
疑似命令を、機能別、およびアルファベット順に説明します。

4.1 疑似命令のまとめ 4-2
4.2 TMS320C1x/C2x/C2xx/C5x アセンブラ疑似命令との互換性 4-8
4.3 セクションを定義する疑似命令 4-9
4.4 定数を初期化する疑似命令 4-11
4.5 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 4-15
4.6 出力リストのフォーマットを設定する疑似命令 4-17

4.7 他のファイルを参照する疑似命令 4-19
4.8 条件付きアセンブリ疑似命令 4-20
4.9 アセンブル時シンボル疑似命令 4-21
4.10 その他の疑似命令 4-23
4.11 疑似命令のリファレンス 4-25

5 マクロ言語 5-1
マクロのパラメータとして使用されるマクロ疑似命令、および置換シンボルについて説明します。また、マクロの作成方法についても説明します。
5.1 マクロの使用方法 5-2
5.2 マクロの定義方法 5-3
5.3 マクロ・パラメータ/置換シンボル 5-6
5.3.1 置換シンボルを定義するための疑似命令 5-7
5.3.2 ビルトイン置換シンボル関数 5-8
5.3.3 再帰的置換シンボル 5-10
5.3.4 強制置換 5-11
5.3.5 添字付き置換シンボルを使用して個々の文字へアクセスする方法 5-12
5.3.6 マクロでローカル変数として使用される置換シンボル 5-13
5.4 マクロ・ライブラリ 5-14
5.5 マクロでの条件付きアセンブリの使用方法 5-15
5.6 マクロでのラベルの使用方法 5-17
5.7 マクロでのメッセージの作成方法 5-19
5.8 出力リストをフォーマットする方法 5-21
5.9 再帰的なマクロとネストされたマクロの使用方法 5-22
5.10 マクロ疑似命令のまとめ 5-25

6 アーカイバの説明 6-1
アーカイバの起動方法、新しいアーカイブ・ライブラリの作成方法、および既存のライブラリの修正方法について説明します。
6.1 アーカイバの概要 6-2
6.2 アーカイバと開発フロー 6-3
6.3 アーカイバの起動方法 6-4
6.4 アーカイバの例 6-6

7 リンカの説明 7-1
リンカの起動方法について説明し、リンカの操作について詳細な情報を提供し、リンカの疑似命令について説明し、および詳細なリンクの例を示します。
7.1 リンカの概要 7-2
7.2 リンカと開発フロー 7-3
7.3 リンカの起動方法 7-4
7.4 リンカ・オプション 7-6
7.4.1 再配置機能 (−a オプションと −r オプション) 7-8
7.4.2 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) 7-10
7.4.3 C 言語オプション (−c オプションと −cr オプション) 7-10

7.4.4 エントリ・ポイントの定義（−e global_symbol オプション） 7-11
7.4.5 デフォルトの埋め込み値の設定（−f cc オプション） 7-11
7.4.6 シンボルのグローバル化（−g global_symbol オプション） 7-12
7.4.7 すべてのグローバル・シンボルの静的化（−h オプション） 7-12
7.4.8 ヒープ・サイズの定義（−heap constant オプション） 7-12
7.4.9 ライブラリ検索アルゴリズムの変更（−l オプション、−i オプション、および C54X_C_DIR/C_DIR 環境変数） 7-13
7.4.10 条件付きリンクの無効化（−j オプション） 7-16
7.4.11 位置合わせフラグの無視（−k オプション） 7-16
7.4.12 マップ・ファイルの作成（−m filename オプション） 7-16
7.4.13 出力モジュールの名前の指定（−o filename オプション） 7-17
7.4.14 静的な実行の指定（−q オプション） 7-17
7.4.15 シンボル情報の除去（−s オプション） 7-17
7.4.16 スタック・サイズの定義（−stack constant オプション） 7-18
7.4.17 未解決のシンボルの導入（−u symbol オプション） 7-18
7.4.18 COFF フォーマットの指定（−v オプション） 7-19
7.4.19 出力セクション情報のメッセージの表示（−w オプション） 7-19
7.4.20 ライブラリの強制読み取り（−x オプション） 7-20
7.5 リンカ・コマンド・ファイル 7-21
7.5.1 リンカ・コマンド・ファイル内で予約済みの名前 7-24
7.5.2 コマンド・ファイルの定数 7-24
7.6 オブジェクト・ライブラリ 7-25
7.7 MEMORY 疑似命令 7-27
7.7.1 デフォルトのメモリ・モデル 7-27
7.7.2 MEMORY 疑似命令の構文 7-27
7.8 SECTIONS 疑似命令 7-32
7.8.1 デフォルト構成 7-32
7.8.2 SECTIONS 疑似命令の構文 7-32
7.8.3 割り当て 7-35
7.9 セクションの実行（ランタイム）アドレスの指定方法 7-44
7.9.1 ロード・アドレスと実行アドレスの指定方法 7-44
7.9.2 初期化されないセクション 7-45
7.9.3 .label 疑似命令を使用してロード・アドレスを参照する方法 7-45
7.10 UNION 文と GROUP 文の使用方法 7-48
7.10.1 UNION 文によるセクションのオーバーレイ 7-48
7.10.2 出力セクションをグループ化する方法 7-50
7.10.3 UNION および GROUP をネストする方法 7-51
7.10.4 割り当てルーチンの一貫性を調べる方法 7-52
7.11 オーバーレイ・ページ 7-53
7.11.1 MEMORY 疑似命令を使用してオーバーレイ・ページを定義する方法 7-53
7.11.2 SECTIONS 疑似命令を使用してオーバーレイ・ページを使用する方法 7-55
7.11.3 ページ定義構文 7-56

7.12 デフォルトの割り当てアルゴリズム 7-58
7.12.1 割り当てアルゴリズム 7-58
7.12.2 出力セクションの一般的な規則 7-59
7.13 特別なセクションの型 (DSECT、COPY、および NOLOAD) 7-61
7.14 リンク時のシンボルの割り当て方法 7-62
7.14.1 代入文の構文 7-62
7.14.2 SPC をシンボルに割り当てる方法 7-63
7.14.3 代入式 7-63
7.14.4 リンカで定義されるシンボル 7-65
7.14.5 C サポート用のみに定義されるシンボル (−c オプションまたは −cr オプション) 7-65
7.15 ホールの作成方法と埋め込み方法 7-66
7.15.1 初期化されたセクションと初期化されないセクション 7-66
7.15.2 ホールの作成方法 7-66
7.15.3 ホールの埋め込み方法 7-68
7.15.4 初期化されないセクションの明示的な初期化 7-69
7.16 部分 (インクリメンタル) リンク 7-70
7.17 C/C++ コードのリンク 7-72
7.17.1 実行時初期化 7-72
7.17.2 オブジェクト・ライブラリとランタイム・サポート 7-72
7.17.3 スタック・セクションとヒープ・セクションのサイズ設定 7-73
7.17.4 自動初期化 (ROM モデルと RAM モデル) 7-73
7.17.5 −c リンカ・オプションと −cr リンカ・オプション 7-75
7.18 リンカの例 7-76

8 絶対リスタの説明 8-1
絶対リスタを起動して、オブジェクト・ファイルの絶対アドレスのリストを取得する方法について説明します。
8.1 絶対リスト出力の作成方法 8-2
8.2 絶対リスタの起動方法 8-3
8.3 絶対リスタの例 8-5

9 クロスリファレンス・リスタの説明 9-1
クロスリファレンス・リスタを起動して、シンボル、シンボルの定義、およびリンクされたソース・ファイル内でのシンボルの参照を収めたリストを取得する方法について説明します。
9.1 クロスリファレンス・リストの作成方法 9-2
9.2 クロスリファレンス・リスタの起動方法 9-3
9.3 クロスリファレンス・リストの例 9-4

10 Hex 変換ユーティリティの説明 10-1
Hex 変換ユーティリティを起動して、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 16 進フォーマットのいずれかに変換する方法について説明します。
10.1 Hex 変換ユーティリティと開発フロー 10-2
10.2 Hex 変換ユーティリティの起動方法 10-3

10.3 コマンド・ファイル 10-7
10.3.1 コマンド・ファイルの例 10-8
10.4 メモリ幅について 10-9
10.4.1 ターゲット幅 10-10
10.4.2 データ幅 10-10
10.4.3 メモリ幅 10-10
10.4.4 ROM 幅 10-11
10.4.5 メモリ構成の例 10-14
10.4.6 出力ワードのワード配列の指定方法 10-14
10.5 ROMS 疑似命令 10-16
10.5.1 ROMS 疑似命令を指定する場合 10-18
10.5.2 ROMS 疑似命令の例 10-19
10.5.3 ROMS 疑似命令のマップ・ファイルを作成する方法 10-21
10.6 SECTIONS 疑似命令 10-22
10.7 出力ファイル名 10-24
10.7.1 出力ファイル名を割り当てる方法 10-24
10.8 イメージ・モードと −fill オプション 10-26
10.8.1 −image オプション 10-26
10.8.2 埋め込み値を指定する方法 10-27
10.8.3 イメージ・モードを使用する手順 10-27
10.9 オンチップ・ブート・ローダ用のテーブルの作成方法 10-28
10.9.1 ブート・テーブルについて 10-28
10.9.2 ブート・テーブル・フォーマット 10-28
10.9.3 ブート・テーブルの作成方法 10-29
10.9.4 ペリフェラルからのブート方法 10-31
10.9.5 ブート・テーブルのエントリ・ポイントの設定方法 10-32
10.9.6 'C54x ブート・ローダの使用方法 10-32
10.10 ROM デバイス・アドレスの制御方法 10-34
10.10.1 開始アドレスの制御方法 10-34
10.10.2 アドレス・インクリメント・インデックスの制御方法 10-36
10.10.3 −byte オプション 10-36
10.10.4 アドレス・ホールの処理方法 10-37
10.11 オブジェクト・フォーマットの説明 10-38
10.11.1 ASCII−Hex オブジェクト・フォーマット（−a オプション） 10-39
10.11.2 Intel MCS−86 オブジェクト・フォーマット（−i オプション） 10-40
10.11.3 Motorola Exorciser オブジェクト・フォーマット（−m1 オプション、 −m2 オプション、および −m3 オプション） 10-41
10.11.4 TI−SDSMAC オブジェクト・フォーマット（−t オプション） 10-42
10.11.5 Extended Tektronix オブジェクト・フォーマット（−x オプション） 10-43
10.12 Hex 変換ユーティリティのエラー・メッセージ 10-44

11 ニーモニックから代数表記への変換ユーティリティの説明 11-1
ニーモニックから代数表記への変換ユーティリティを起動し、ニーモニック命令が入っているソース・ファイルを、代数表記命令が入ったソース・ファイルに変換する方法について説明します。
11.1 変換ユーティリティの概要 11-2
11.1.1 変換ユーティリティが実行すること 11-2
11.1.2 変換ユーティリティが実行しないこと 11-2
11.2 変換ユーティリティと開発フロー 11-3
11.3 変換ユーティリティの起動方法 11-4
11.4 変換モード 11-5
11.4.1 リテラル・モード (–t オプション) 11-5
11.4.2 リテラル・モードでのシンボル名について 11-5
11.4.3 展開モード (–e オプション) 11-6
11.5 変換ユーティリティでのマクロの処理方法 11-8
11.5.1 マクロ内の疑似命令 11-8
11.5.2 マクロ・ローカル変数 11-9
11.5.3 マクロを起動するときのラベルの定義 11-10

A 共通オブジェクト・ファイル・フォーマット A-1
COFF オブジェクト・ファイルの内部フォーマットおよび構造に関する補足的な技術解説を記述しています。
A.1 COFF ファイルの構造 A-2
A.1.1 オペレーティング・システムの交換による影響 A-4
A.2 ファイル・ヘッダの構造 A-5
A.3 オプション・ファイル・ヘッダのフォーマット A-6
A.4 セクション・ヘッダの構造 A-7
A.5 再配置情報の構造化 A-10
A.6 行番号テーブルの構造 A-12
A.7 シンボル・テーブルの構造と内容 A-14
A.7.1 特殊シンボル A-16
A.7.2 シンボル名のフォーマット A-18
A.7.3 文字列テーブルの構造 A-18
A.7.4 記憶クラス A-19
A.7.5 シンボルの値 A-20
A.7.6 セクション番号 A-21
A.7.7 型エントリ A-21
A.7.8 補足エントリ A-23

B シンボリック・ デバッグ疑似命令 B-1
C コンパイラが使用するシンボリック・デバッグ疑似命令について説明します。

C Hex 変換ユーティリティの例 C-1
各種のメモリ・システム用のコマンド・ファイルの作成方法を、図に示して説明します。
C.1 例で使用する基本コード C-2
C.2 例 1: 2 つの 8 ビット EPROM を対象とした Hex コマンド・ファイルの作成方法 ... C-3

C.3 例 2: 複数のセクション間のホールを回避する方法 C-8
C.4 例 3: ブート・テーブルの作成方法 C-10
C.5 例 4: LP コア・デバイス用のブート・テーブルの作成方法 C-17

D アセンブラのエラー・メッセージ D-1
アセンブラが発行するエラー・メッセージと、それぞれのエラーの原因となった状況に関する説明を記述しています。

E リンカのエラー・メッセージ E-1
リンカが発行するエラー・メッセージと、それぞれのエラーの原因となった状況に関する説明を記述しています。

F 用語集 F-1
本書で使用されている用語や省略語について説明します。

図

図 1−1. TMS320C54x のソフトウェア開発フロー 1-2
図 2−1. メモリの論理ブロックへの区画分け 2-4
図 2−2. 例 2−1 のファイルで生成されたオブジェクト・コード 2-12
図 2−3. 入力セクションの結合による実行可能オブジェクト・モジュールの作成 2-14
図 3−1. アセンブラと開発フロー 3-3
図 4−1. .space 疑似命令と .bes 疑似命令 4-11
図 4−2. .field 疑似命令 4-12
図 4−3. 初期化疑似命令 4-14
図 4−4. .align 疑似命令 4-16
図 4−5. ページ内への .bss ブロックの割り当て方法 4-31
図 4−6. .field 疑似命令 4-49
図 4−7. .usect 疑似命令 4-97
図 6−1. アーカイバと開発フロー 6-3
図 7−1. リンカと開発フロー 7-3
図 7−2. 例 7−3 で定義されたメモリ・マップ 7-31
図 7−3. 例 7−4 で定義されたセクションの割り当て 7-34
図 7−4. 例 7−6 の実行時の様子 7-47
図 7−5. 例 7−7 と例 7−8 で定義されたメモリ割り当て 7-49
図 7−6. 例 7−11 と例 7−12 で定義されているオーバーレイ・ページ 7-54
図 7−7. 自動初期化の RAM モデル 7-74
図 7−8. 自動初期化の ROM モデル 7-74
図 8−1. 絶対リスタと開発のフロー 8-2
図 8−2. module1.lst 8-9
図 8−3. module2.lst 8-9
図 9−1. クロスリファレンス・リスタと開発フロー 9-2
図 10−1. Hex 変換ユーティリティと開発フロー 10-2
図 10−2. Hex 変換ユーティリティの処理フロー 10-9
図 10−3. データ幅とメモリ幅 10-11
図 10−4. データ幅、メモリ幅、および ROM 幅 10-13
図 10−5. ’C54x のメモリ構成の例 10-14
図 10−6. ワード配列の変化 10-15
図 10−7. 例 10−1 の infile.out ファイルを 4 個の出力ファイルに分割する 10-20
図 10−8. ’C54x EPROM からブートするためのコマンド・ファイルの例 10-33
図 10−9. 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 10-37

図 10−10. ASCII−Hex オブジェクト・フォーマット 10-39
図 10−11. Intel−Hex オブジェクト・フォーマット 10-40
図 10−12. Motorola−S フォーマット 10-41
図 10−13. TI−Tagged オブジェクト・フォーマット 10-42
図 10−14. Extended Tektronix オブジェクト・フォーマット 10-43
図 11−1. 変換ユーティリティと開発フロー 11-3
図 11−2. リテラル・モードの処理 11-5
図 11−3. 展開モードの処理 11-6
図 11−4. ラベルの定義方法 11-10
図 11−5. 書き直したソース・コード 11-10
図 A−1. COFF ファイルの構造 A-2
図 A−2. COFF オブジェクト・ファイル A-3
図 A−3. .text セクションのセクション・ヘッダのポインタ A-9
図 A−4. 行番号のブロック A-12
図 A−5. 行番号エントリ A-13
図 A−6. シンボル・テーブルの内容 A-14
図 A−7. ブロックに関するシンボル A-17
図 A−8. 関数に関するシンボル A-17
図 A−9. 文字列テーブル A-18
図 C−1. 2 つの 8 ビット EPROM があるシステム C-3
図 C−2. 出力ファイルのデータ C-6
図 C−3. 'C54x 用の EPROM システム C-10
図 C−4. 'C54xLP 用の EPROM システム C-17

表

表 3−1. 式で使用できる演算子（優先順位）. 3-26
表 3−2. 絶対シンボルおよび再配置可能シンボルをもつ式 3-28
表 3−3. アセンブラ・ビルトイン数学関数 3-30
表 3−4. シンボル属性 3-38
表 4−1. 疑似命令のまとめ 4-2
表 4−2. メモリ・マップド・レジスタ 4-71
表 5−1. 関数と戻り値 5-9
表 5−2. マクロの作成方法 5-25
表 5−3. 置換シンボルの操作方法 5-25
表 5−4. 条件付きアセンブリ 5-25
表 5−5. アセンブル時メッセージの作成方法 5-26
表 5−6. リストをフォーマットする方法 5-26
表 7−1. 式で使用できる演算子（優先順位）. 7-64
表 9−1. シンボルの属性 9-6
表 10−1. Hex 変換ユーティリティのオプション 10-4
表 10−2. ブート・ローダ・オプション 10-29
表 10−3. Hex 変換フォーマットを指定するオプション 10-38
表 A−1. ファイル・ヘッダの内容 A-5
表 A−2. ファイル・ヘッダのフラグ（バイト 18 ～ 19）. A-5
表 A−3. オプション・ファイル・ヘッダの内容 A-6
表 A−4. COFF1 ファイルのセクション・ヘッダの内容 A-7
表 A−5. COFF2 ファイルのセクション・ヘッダの内容 A-7
表 A−6. セクション・ヘッダのフラグ A-8
表 A−7. 再配置エントリの内容 A-10
表 A−8. 再配置タイプ（バイト 8 ～ 9）. A-11
表 A−9. 行番号エントリのフォーマット A-12
表 A−10. シンボル・テーブル・エントリの内容 A-15
表 A−11. シンボル・テーブル内の特殊シンボル A-16
表 A−12. シンボルの記憶クラス A-19
表 A−13. 特殊シンボルとその記憶クラス A-20
表 A−14. シンボルの値と記憶クラス A-20
表 A−15. セクション番号 A-21
表 A−16. 基本型 A-22
表 A−17. 派生型 A-22

表 A−18. シンボル・テーブル補足エントリのフォーマット A-23
表 A−19. テーブル補足エントリ用のファイル名フォーマット A-24
表 A−20. テーブル補足エントリ用のセクション・フォーマット A-24
表 A−21. テーブル補足エントリ用のタグ名フォーマット A-24
表 A−22. テーブル補足エントリ用の構造体の終了フォーマット A-25
表 A−23. テーブル補足エントリ用の関数フォーマット A-25
表 A−24. テーブル補足エントリ用の配列フォーマット A-26
表 A−25. 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 A-26
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開始フォーマット A-27
表 A−27. 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 ... A-27

例

例 2−1. セクション疑似命令の使用方法 2-11
例 2−2. 再配置エントリを生成するコード 2-16
例 3−1. $n ローカル・ラベル 3-22
例 3−2. name? ローカル・ラベル 3-24
例 3−3. 整合定義式 3-27
例 3−4. アセンブラ・リスト 3-35
例 3−5. クロスリファレンス・リストの例 3-37
例 4−1. セクションの疑似命令 4-10
例 5−1. マクロの定義、呼び出し、展開 5-4
例 5−2. 引数の数が異なるマクロ呼び出しの例 5-7
例 5−3. .asg 疑似命令 5-8
例 5−4. .eval 疑似命令 5-8
例 5−5. ビルトイン置換シンボル関数の使用方法 5-9
例 5−6. 再帰的置換 5-10
例 5−7. 強制置換演算子の使用例 5-11
例 5−8. 添字付き置換シンボルを使用して命令を再定義する例 5-12
例 5−9. 添字付き置換シンボルを使用して部分文字列を見つける例 5-13
例 5−10. .loop/.break/.endloop 疑似命令 5-16
例 5−11. ネストされた条件付きアセンブリ疑似命令 5-16
例 5−12. 条件付きアセンブリ・コード・ブロックで使用されている ビルトイン置換シンボル関数 5-16
例 5−13. マクロでの固有のラベル 5-17
例 5−14. マクロでのメッセージの作成方法 5-20
例 5−15. ネストされたマクロの使用方法 5-22
例 5−16. 再帰的なマクロの使用方法 5-23
例 7−1. リンカ・コマンド・ファイル 7-22
例 7−2. リンカ疑似命令を含むコマンド・ファイル 7-23
例 7−3. MEMORY 疑似命令 7-28
例 7−4. SECTIONS 疑似命令 7-34
例 7−5. セクション内容を指定する最も一般的な方法 7-38
例 7−6. セクションを ROM から RAM にコピーする方法 7-46
例 7−7. UNION 文 7-48
例 7−8. UNION セクションに対する別々のロード・アドレス 7-48
例 7−9. セクション・グループの割り当て 7-50
例 7−10. GROUP 文および UNION 文をネストする方法 7-51
例 7−11. オーバーレイ・ページを指定した MEMORY 疑似命令 7-53

例 7−12. 図 7−6 のオーバーレイ用の SECTIONS 疑似命令定義 7-55
例 7−13. TMS320C54x デバイスのデフォルトの割り当て 7-58
例 7−14. リンカ・コマンド・ファイル、 demo.cmd 7-77
例 7−15. 出力マップ・ファイル、 demo.map 7-78
例 10−1. ROMS 疑似命令の例 10-19
例 10−2. 例 10−1 のマップ・ファイル出力によるメモリ範囲 10-21
例 11−1. リテラル・モードでのシンボル名の扱い 11-5
例 11−2. 展開モード 11-7
例 11−3. マクロ内の疑似命令 11-8
例 11−4. マクロ・ローカル変数 11-9
例 C−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 C-2
例 C−2. 2 つの 8 ビット EPROM 用のリンカ・コマンド・ファイル C-4
例 C−3. 2 つの 8 ビット EPROM 用の Hex コマンド・ファイル C-5
例 C−4. C-5 ページの例 C−3 の Hex コマンド・ファイルから出力されるマップ・ファイル . C-7
例 C−5. ホール回避方式 1 . . . . . C-8
例 C−6. ホール回避方式 2 . . . . . C-9
例 C−7. 例 3 で使用する C コード C-10
例 C−8. 1 つのブート・セクションを作成するリンカ・コマンド・ファイル C-12
例 C−9. 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当て部分 . C-13
例 C−10. COFF ファイルを変換するための Hex コマンド・ファイル C-15
例 C−11. 例 C−10 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 C-16
例 C−12. 例 C−10 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 C-16
例 C−13. ’C54xLP 用の C コード C-17
例 C−14. ’C54xLP のリンカ・コマンド・ファイル C-19
例 C−15. 例 C−14 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当て部分 C-19
例 C−16. COFF ファイルを変換するための Hex コマンド・ファイル C-22
例 C−17. 例 C−16 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 C-23
例 C−18. 例 C−16 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 C-23

第 1 章

はじめに

TMS320C54x DSP は、以下のアセンブリ言語ツールによってサポートされています。

- ❑ アセンブラ
- ❑ アーカイバ
- ❑ リンカ
- ❑ 絶対リスタ
- ❑ クロスリファレンス・ユーティリティ
- ❑ Hex 変換ユーティリティ
- ❑ ニーモニックから代数表記への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について、および個々のツールについて簡単に説明します。また、読者の理解を助けるために、C コンパイラとデバッグ・ツールについても要点を説明します。コンパイラとデバッガ、および TMS320C54x デバイスの詳細は 、vi ページの「当社発行の関連文献」を参照してください。

アセンブリ言語ツールでは、モジュラ・プログラミングに便利なように、共通オブジェクト・ファイル・フォーマット (COFF) を使ったオブジェクト・ファイルを生成し、使用しています。オブジェクト・ファイルには、それぞれが独立したコードおよびデータのブロック (セクションと呼ばれる) が含まれており、これを 'C54x 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、'C54x のプログラミングの能率が上がります。第 2 章「共通オブジェクト・ファイル・フォーマットの概要」では、オブジェクト・フォーマットについて詳細に説明しています。

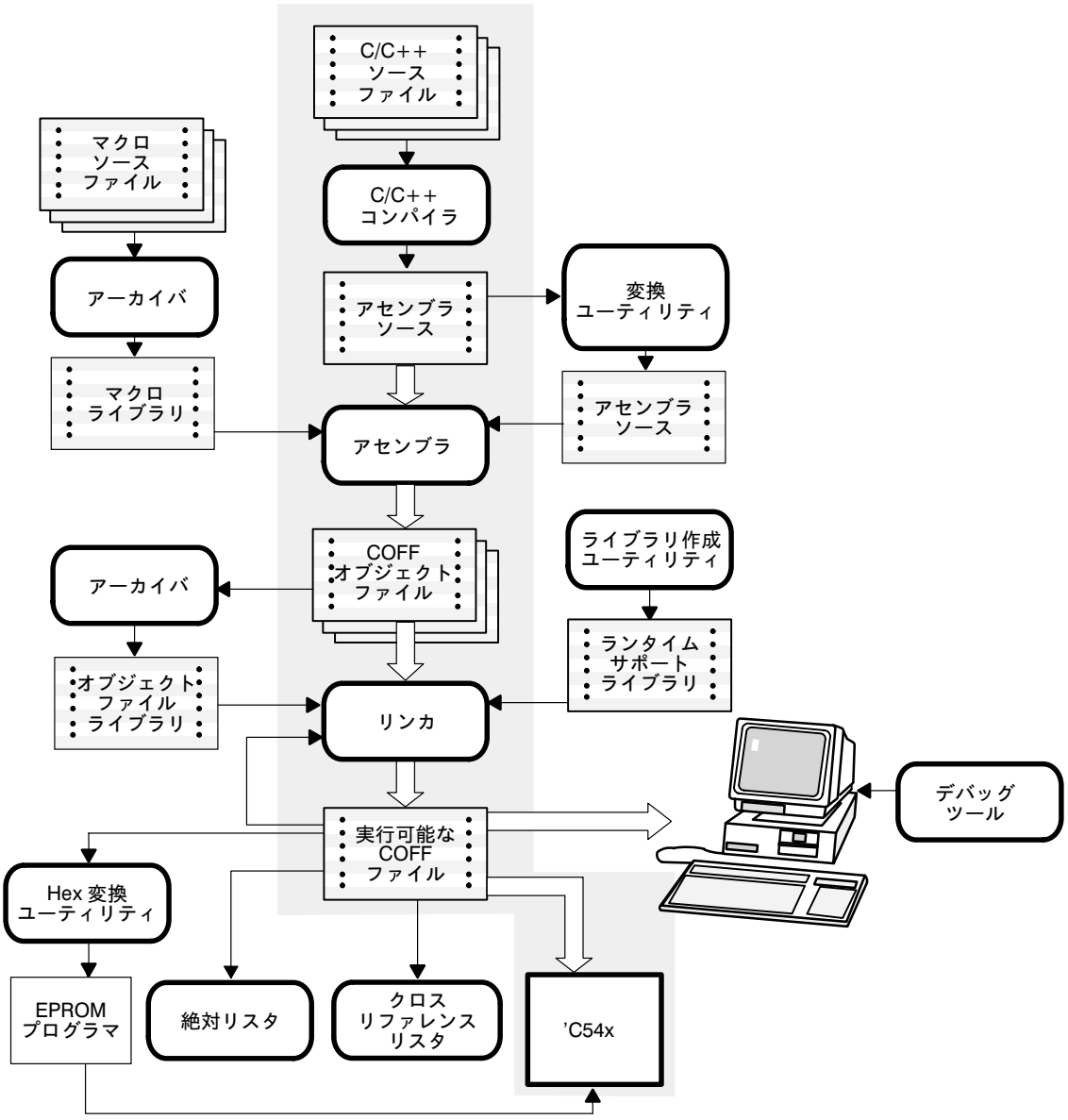
項目 **ページ**

1.1 ソフトウェア開発ツールの概要 1-2
1.2 各ツールの説明 1-3

1.1 ソフトウェア開発ツールの概要

図 1−1 に ’C54x ソフトウェア開発のフローを示します。図の中の陰影を付けた部分は、C 言語プログラムのソフトウェア開発における最も一般的な経路です。それ以外の部分は、必要に応じて使用されます。

図 1−1. TMS320C54x のソフトウェア開発フロー



1.2 各ツールの説明

次に、図 1−1 に示されているツールについて説明します。

- **C/C++ コンパイラ**は、C/C++ ソース・コードを ’C54x のアセンブリ言語のソース・コードに変換します。C コンパイラ・パッケージには、**ライブラリ作成ユーティリティ**が含まれています。このユーティリティを使って、各自のランタイム・ライブラリを作成できます。

- **アセンブラ**は、アセンブリ言語のソース・ファイルを機械語の COFF オブジェクト・ファイルに変換します。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・リストのフォーマット、データの位置合わせ、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。

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

- **アーカイバ**を使用すると、複数のファイルを集めて 1 つのアーカイブ・ファイルを作成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。

- **ニーモニックから代数表記へのアセンブリ変換ユーティリティ**は、ニーモニック命令が記述されているアセンブリ言語ソース・ファイルを、代数表記命令が記述されているアセンブリ言語ソース・ファイルに変換します。

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

- TMS320C54x DSP は、COFF ファイルを入力として受け入れますが、ほとんどの EPROM プログラマは、COFF ファイルを入力として受け入れません。**Hex 変換ユーティリティ**は、COFF オブジェクト・ファイルを、TI−tagged、Intel、Motorola、Tektronix のいずれかのオブジェクト形式に変換します。変換後のファイルは、EPROM プログラマへダウンロードできます。

- **絶対リスタ**は、リンク後のオブジェクト・ファイルを入力として受け入れ、.abs ファイルを出力として生成します。これらの .abs ファイルをアセンブルすると、相対アドレスでなく絶対アドレスが入ったリストを作成できます。絶対リスタがないと、そのようなリストの作成は多くの手動操作が必要となる面倒な作業となります。
- **クロスリファレンス・リスタ**は、オブジェクト・ファイルから、シンボル、シンボルの定義、およびリンク済みソース・ファイルでの参照をリストにしたクロスリファレンス・リストを生成します。

この開発プロセスの主な目的は、'C54x ターゲット・システムで実行できるモジュールを生成することです。次のいずれかのデバッグ・ツールを使用して、生成したコードに改善や修正を加えることができます。使用できるデバッグ・ツールには、次の製品があります。

- 命令の正確度を確認するためのソフトウェアであるシミュレータ
- 評価モジュール (EVM)
- XDS エミュレータ

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

第 2 章

共通オブジェクト・ファイル・フォーマットの概要

アセンブラとリンカを使用して、TMS320C54x™ デバイスで実行可能なオブジェクト・ファイルを作成することができます。これらのオブジェクト・ファイルのフォーマットを共通オブジェクト・ファイル・フォーマット (COFF) と呼びます。

COFF を使用するとアセンブリ言語プログラムを書くときに、コードまたはデータのブロック単位で考えることができるので、モジュラ・プログラミングを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ます。詳細は、付録 A「共通オブジェクト・ファイル・フォーマット」を参照してください。COFF の構造についての説明があります。

項目 **ページ**

2.1 COFF ファイルのタイプ 2-2
2.2 セクション 2-3
2.3 アセンブラによるセクションの処理 2-5
2.4 リンカによるセクションの処理 2-13
2.5 再配置 2-16
2.6 実行時の再配置 2-19
2.7 プログラムのロード 2-20
2.8 COFF ファイルで使用されるシンボル 2-21

2.1 COFF ファイルのタイプ

COFF ファイルには、次のタイプがあります。

- ❏ COFF0
- ❏ COFF1
- ❏ COFF2

各 COFF ファイル・タイプによって、ヘッダのフォーマットが異なります。データ部分はそれぞれの COFF ファイルで共通しています。COFF ファイルの構造の詳細は、付録 A「共通オブジェクト・ファイル・フォーマット」を参照してください。

TMS320C54x アセンブラと C コンパイラでは、COFF2 のタイプのファイルが作成されます。リンカでは、全タイプの COFF ファイルの読み取りと書き込みが可能です。デフォルトで、リンカは COFF2 ファイルを作成します。別のフォーマットを指定するには、−v リンカ・オプションを使用してください。リンカで COFF0 および COFF1 ファイルを使用できるのは、アセンブラと C コンパイラの旧バージョンで作成されたファイルをサポートするためです。

2.2 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別れていて、互いに独立しています。COFF オブジェクト・ファイルには、必ず次の 3 つのデフォルトのセクションが含まれています。

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

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

セクションには 2 つの基本的な型があります。

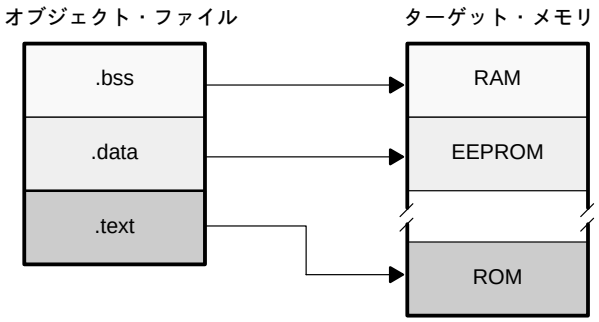
初期化されたセクション	データまたはコードを含みます。.text セクションと .data セクションは初期化されます。.sect アセンブラ疑似命令によって作成された名前付きセクションも初期化されます。
初期化されないセクション	初期化されないデータのための空間を確保します。.bss セクションは初期化されません。.usect アセンブラ疑似命令によって作成された名前付きセクションも初期化されません。

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

リンカの機能の 1 つに、セクションのターゲット・システムのメモリ・マップへの配置があります。これを「割り当て」と呼びます。ほとんどのシステムではいくつかの異なった型のメモリが使われているので、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、独立していて再配置が可能です。それぞれのセクションは、ターゲット・メモリ内の割り当てられたブロックに入れることができます。たとえば、初期化ルーチンをもつセクションを定義して、それをメモリ・マップの ROM を含む部分に割り当てることができます。

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

図 2–1.　メモリの論理ブロックへの区画分け



2.3 アセンブラによるセクションの処理

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

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

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

任意のセクションのサブセクションを作成して、メモリ・マップをより厳密に制御することができます。.sect 疑似命令と .usect 疑似命令を使用して、サブセクションを作成します。サブセクションは、コロンで区切られたベース・セクション名とサブセクション名で識別できます。詳細は、2-9 ページの 2.3.4 項「サブセクション」を参照してください。

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

2.3.1 初期化されないセクション

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

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

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

.bss 疑似命令を呼び出すたびに、アセンブラはさらに多くの空間を適切なセクションに確保します。.usect 疑似命令を呼び出すたびに、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用します。

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

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

symbol .bss または .usect 疑似命令の呼び出しによって確保される空間の最初のワードを指します。*symbol* は、空間を確保する変数の名前に対応します。symbol は、他のどのセクションからも参照でき、また (.global アセンブラ疑似命令を使って) 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 in words* で指定されたワードを確保します。
- ❑ .usect 疑似命令は、*section name* に *size in words* で指定されたワードを確保します。

blocking flag 任意のパラメータです。このパラメータに 0 以外の値を指定すると、アセンブラは、*size in words* で指定されたワードを連続して関連付けます。*size in words* で指定されたワード数が 1 ページより大きくない限り、割り当て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 in words* で指定されたワード数が 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当てられます。

alignment flag 任意のパラメータです。このパラメータに 0 以外の値を指定すると、セクションはロング・ワードの境界に割り当てられます。

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細は、2-8 ページの 2.3.3 項「名前付きセクション」を参照してください。

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

初期化されないサブセクションを作成するには、.usect 疑似命令を使用します。アセンブラは、初期化されないサブセクションを初期化されないセクションと同様に扱います。サブセクションの作成に関する詳細は、2-9 ページの 2.3.4 項「サブセクション」を参照してください。

2.3.2 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、プログラムのロード時にプロセッサ・メモリに入れられます。個々の初期化されたセクションは、別々に再配置することが可能で、他のセクションで定義されているシンボルを参照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。

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

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

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

セクションは、反復工程によって構築されます。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 疑似命令か、.sect 疑似命令のいずれかを見つけるまで) .data セクション内にアセンブルされます。アセンブラは、次の .data 疑似命令を見つけると、その .data 疑似命令に続く文をすでに .data セクションに入っている文に追加します。このようにして、メモリに連続的に配置できる 1 つの .data セクションを作成します。

初期化されたサブセクションは、.sect 疑似命令で作成します。アセンブラは、初期化されたサブセクションを初期化されたセクションと同様に扱います。サブセクションの作成についての詳細は、2-9 ページの 2.3.4 項「サブセクション」を参照してください。

2.3.3 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。名前付きセクションはデフォルトの .text、.data、および .bss セクションと同様に使用できますが、別にアセンブルされます。

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

次の疑似命令を使用して名前付きセクションを作成できます。

- **.usect** 疑似命令を使用すると、.bss セクションと同様に使えるセクションを作成できます。これらのセクションは、変数を入れるための空間を RAM に確保します。
- **.sect** 疑似命令を使用すると、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もった名前付きセクションを作成できます。

各疑似命令の構文は次のとおりです。

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

section name パラメータは、セクションの名前です。最大 32 767 個の独立した名前付きセクションを作成することができます。セクション名は、200 文字まで有効です。COFF1 ファイルについては、最初の 8 文字以外は意味がありません。.sect 疑似命令と .usect 疑似命令では、セクション名がサブセクション名を指す場合があります (詳細は、2.3.4 項「サブセクション」を参照してください)。

これらの疑似命令を新しい名前を付けて呼び出すたびに、新しい名前付きセクションが作成されます。これらの疑似命令をすでに使われた名前を付けて呼び出すたびに、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。

2.3.4 サブセクション

サブセクションはセクション内に存在する小さなセクションです。セクション同様、サブセクションについても、リンカで操作することができます。サブセクションを設定すると、メモリ・マップをより詳細に制御することができます。サブセクションの作成には、.sect または .usect 疑似命令を使用します。サブセクション名の構文は以下のとおりです。

section name:*subsection name*

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

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

_func サブセクションは、個別に割り当てることも、他のすべての .text セクションに割り当てることもできます。

以下の 2 つのタイプのサブセクションを作成することができます。

- ❑ 初期化されたサブセクション。.sect 疑似命令によって作成します (2-7 ページの 2.3.2 項「初期化されたセクション」を参照)。
- ❑ 初期化されないサブセクション。.usect 疑似命令によって作成します (2-5 ページの 2.3.1 項「初期化されないセクション」を参照)。

サブセクションの割り当て方法は、セクションの場合と同じです。詳細は、7-32ページの 7.8 項「SECTIONS 疑似命令」を参照してください。

2.3.5 セクション・プログラム・カウンタ

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

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

アセンブラは、各セクションがアドレス 0 で始まるかのように処理します。リンカは、セクションの最終ロケーションに従って各セクションをメモリ・マップ内に再配置します。詳細は、2-16 ページの 2.5 節「再配置」を参照してください。

2.3.6 セクション疑似命令の使用例

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

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

フィールド 1 ソース・コードの行カウンタが入ります。

フィールド 2 セクション・プログラム・カウンタが入ります。

フィールド 3 オブジェクト・コードが入ります。

フィールド 4 オリジナルのソース文が入ります。

例 2−1.　セクション疑似命令の使用方法

```
      2                 ************************************************
      3                 **  Assemble an initialized table into .data. **
      4                 ************************************************
      5  000000                 .data
      6  000000 0011    coeff   .word       011h,022h,033h
         000001 0022
         000002 0033
      7                 ************************************************
      8                 **   Reserve space in .bss for a variable.    **
      9                 ************************************************
     10  000000                 .bss        buffer,10
     11                ************************************************
     12                **              Still in .data.               **
     13                ************************************************
     14  000003 0123    ptr     .word       0123h
     15                ************************************************
     16                **    Assemble code into the .text section.   **
     17                ************************************************
     18  000000                 .text
     19  000000 100f    add:    LD          0Fh,A
     20  000001 f010    aloop:  SUB         #1,A
         000002 0001
     21  000003 f842            BC          aloop,AGEQ
         000004 0001'
     22                ************************************************
     23                **    Another initialized table into .data.   **
     24                ************************************************
     25  000004                 .data
     26  000004 00aa    ivals   .word       0AAh, 0BBh, 0CCh
         000005 00bb
         000006 00cc
     27                ************************************************
     28                ** Define another section for more variables. **
     29                ************************************************
     30  000000        var2    .usect      "newvars", 1
     31  000001        inbuf   .usect      "newvars", 7
     32                ************************************************
     33                **       Assemble more code into .text.       **
     34                ************************************************
     35  000005                 .text
     36  000005 110a    mpy:    LD              0Ah,B
     37  000006 f166    mloop:  MPY             #0Ah,B
         000007 000a
     38  000008 f868            BC              mloop,BNOV
         000009 0006'
     39                ************************************************
     40                **  Define a named section for int. vectors.  **
     41                ************************************************
     42  000000                 .sect           "vectors"
     43  000000 0011            .word           011h, 033h
     44  000001 0033
```

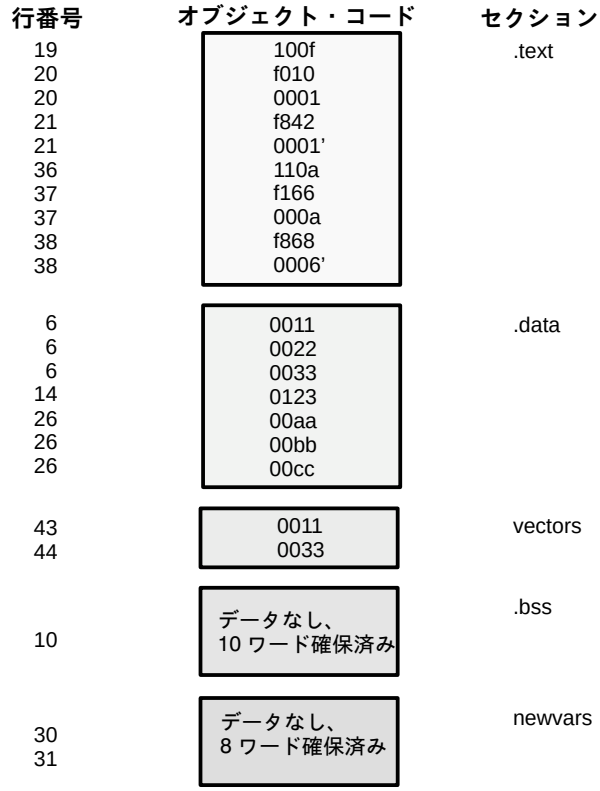
フィールド 1　フィールド 2　フィールド 3　　フィールド 4

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

.text	10 ワードの (各ワード 16 ビット) オブジェクト・コードが入ります。
.data	7 ワードのオブジェクト・コードが入ります。
vectors	.sect 疑似命令によって作成される名前付きセクションです。 2 ワードの初期化されたデータが入ります。
.bss	10 ワードをメモリに確保します。
newvars	.usect 疑似命令を使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 のファイルで生成されたオブジェクト・コード



2.4 リンカによるセクションの処理

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

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

- **MEMORY 疑似命令**を使用すると、ターゲット・システムのメモリ・マップを定義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、入力セクションを結合して出力セクションにする方法、および出力セクションをメモリ内に配置する場所をリンカに伝えます。

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

リンカ疑似命令を必ず使用する必要はありません。リンカ疑似命令を使用しない場合、リンカは 7-58 ページの 7.12 節「デフォルトの割り当てアルゴリズム」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当てアルゴリズムを使用します。リンカ疑似命令を使用する場合には、それをリンカ・コマンド・ファイル内で指定しなければなりません。

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

節	見出し	ページ
7.5	リンカ・コマンド・ファイル	7-21
7.7	MEMORY 疑似命令	7-27
7.8	SECTIONS 疑似命令	7-32
7.12	デフォルトの割り当てアルゴリズム	7-58

2.4.1 デフォルトのメモリ割り当て

図 2−3 は、2 つのファイルのリンク処理を示しています。

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

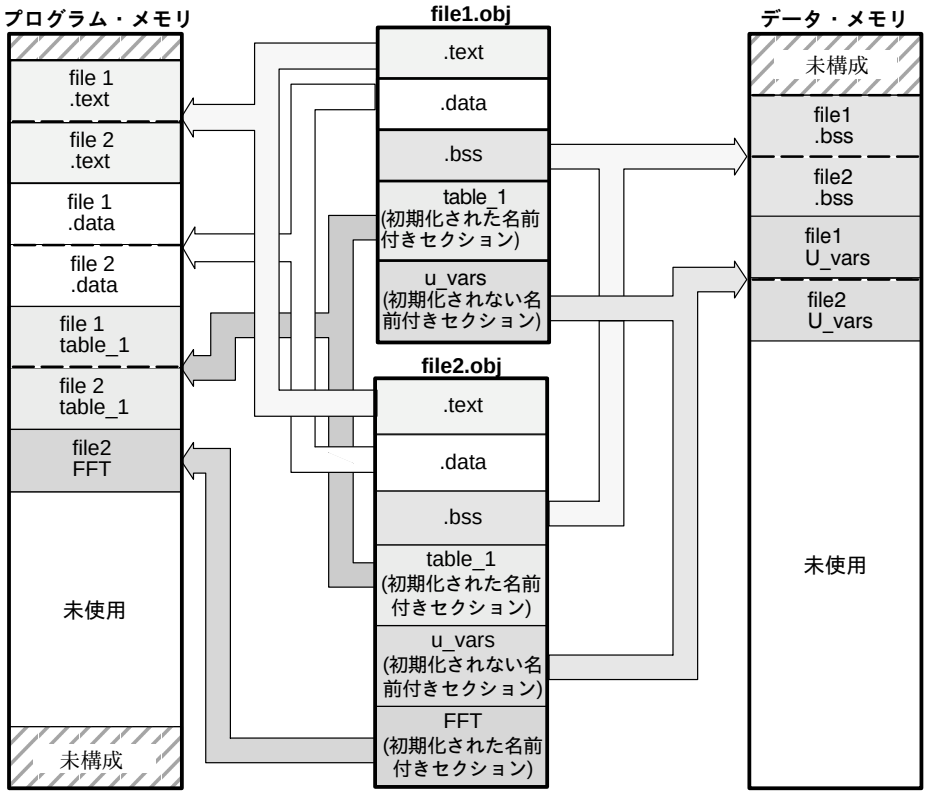


図 2−3 で、file1.obj および file2.obj がアセンブルされていて、リンカ入力として使用されます。各ファイルには、.text、.data、および .bss というデフォルトのセクションと名前付きセクションがあります。実行可能出力モジュールは、これらのセクションの結合を示しています。リンカは、file1.obj の .text と file2.obj の .text を結合して 1 つの .text セクションにします。続いて 2 つの .data セクション、および 2 つの .bss セクションを結合し、最後に、名前付きセクションをモジュールの最後に配置します。メモリ・マ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

2.4.2 セクションのメモリ・マップへの配置

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

メモリ・マップ内のセクションの配置についての詳細は、7-27 ページの 7.7 節「MEMORY 疑似命令」、および 7-32 ページの 7.8 節「SECTIONS 疑似命令」を参照してください。

2.5 再配置

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

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

リンカは、再配置エントリを使用してシンボルの値に対する参照を調整します。アセンブラは、再配置可能なシンボルが参照されるたびに再配置エントリを作成します。リンカはシンボルの再配置が行われた後、このエントリを使って参照をパッチします。例 2−2 には、'C54x の再配置エントリを生成するコード・セグメントが含まれています。

例 2−2. 再配置エントリを生成するコード

(a) ニーモニックの例

```
1                  .ref   X
2                  .ref   Z
3  000000          .text
4  000000 F073     B   Y      ; Generates a Relocation Entry
   000001 0006'
5  000002 F073     B   Z      ; Generates a Relocation Entry
   000003 0000!
6  000004 F020     LD  #X, A  ; Generates a Relocation Entry
   000005 0000!
5  000006 F7E0   Y:  RESET
```

(b) 代数表記の例

```
1                  .ref   X
2                  .ref   Z
3  000000          .text
4  000000 F073     goto   Y   ; Generates a Relocation Entry
   000001 0006'
5  000002 F073     B   Z      ; Generates a Relocation Entry
   000003 0000!
6  000004 F020     A = #X     ; Generates a Relocation Entry
   000005 0000!
7  000006 F7E0   Y:  reset
```

例 2−2 では、X、Y、および Z 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 と Z 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 と Z 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 6 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、Y、および Z のために再配置エントリを生成します。X と Z に対する参照は外部参照です (リスト内の ! 文字によって示されます)。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、X はアドレス 7100h に再配置されるものとします。また、.text セクションは、アドレス 7200h で始まる位置に再配置されるものとします。Y の再配置後の値は 7204h となります。リンカは、2 つの再配置エントリを使用してオブジェクト・コード内の 2 つの参照をパッチします。

```
f073   B  Y          は      f073       となります
0004'                        7204'
f020   LD #X,A       は      f020       となります
0000!                        7100!
```

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

2.5.1 再配置に対するメッセージの発行

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

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

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

いずれのケースでも、リンカは値の高位ビット部を切り捨てます。

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

反対に、幅の広いフィールドをもつ異なるアセンブリ命令を使用している場合を考えてみましょう。または、低ビット数のシンボルのみを必要とする場合は、マスク式を使用して低いビット数をマスク・オフします。

2.6 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パフォーマンスを左右するコードがあるとします。そのコードは ROM にロードしなければなりませんが、実行は RAM を使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 回割り当てるように指示することが可能です。つまり、最初はロード・アドレスを、2 回目は実行アドレスを設定するように指示できます。ロード・アドレスには load キーワードを使用し、実行アドレスには run キーワードを使用します。

ロード・アドレスにより、ローダはセクションの生データを配置する場所を決定します。そのセクションに対するすべての参照 (ラベルの参照など) は実行アドレスを参照します。アプリケーションは、そのセクションをロード・アドレスから実行アドレスへコピーする必要があります。この作業が自動的に行われないのは、別々の実行アドレスを指定するからです。実行時にコードのブロックがどのように移動するかを示す例については、7-46 ページの例 7−6 を参照してください。

1 つのセクションに対して 1 回の割り当て (load または run) しか行わない場合は、そのセクションは 1 回だけ割り当てられ、ロードも実行も同じアドレスで行われます。2 つの割り当てを指定した場合は、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当てられます。

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

実行時の再配置の詳細は、7-44 ページの 7.9 節「セクションの実行 (ランタイム) アドレスの指定方法」を参照してください。

2.7 プログラムのロード

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

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

- ソフトウェア・シミュレータ、XDS51x エミュレータ、ソフトウェア開発システムなどの TMS320C54x デバッグ・ツールには、ローダが組み込まれています。これらのツールには、ローダを起動するための LOAD コマンドが用意されています。ローダは実行可能ファイルを読み取り、プログラムをターゲット・メモリにコピーします。
- Hex 変換ユーティリティ (アセンブリ言語パッケージの一部として出荷されている hex500) を使用して、実行可能 COFF オブジェクト・モジュールを数種類のオブジェクト・ファイル・フォーマットの 1 つに変換することができます。この後、EPROM プログラマを使って変換されたファイルを EPROM に焼き込むことができます。

2.8 COFF ファイルで使用されるシンボル

COFF ファイルには、プログラムで使用されるシンボルに関する情報を格納したシンボル・テーブルがあります。リンカは、このテーブルを使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。

2.8.1 外部シンボル

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

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

以下のコード・セグメントは、これらの定義を示しています。

```
x:   ADD      #56h, A      ; Define x
     B        y            ; Reference y
    .def      x            ; DEF of x
    .ref      y            ; REF of y
```

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

アセンブラは、x と y の両方をオブジェクト・ファイルのシンボル・テーブルに入れます。このファイルが他のオブジェクト・ファイルにリンクされると、x のエントリは、他のファイルからの x に対する未解決の参照を解決します。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ます。リンカがシンボルの定義を見つけられなかった場合は、その未解決の参照についてのエラー・メッセージを出力します。このようなエラーが発行されると、リンカは実行可能なオブジェクト・モジュールを作成できません。

2.8.2 シンボル・テーブル

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

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

第 3 章

アセンブラの説明

アセンブラは、アセンブリ言語のソース・ファイルを機械語のオブジェクト・ファイルに変換します。このファイルは共通オブジェクト・ファイル・フォーマット (COFF) で作成されますが、このフォーマットについては第 2 章「共通オブジェクト・ファイル・フォーマットの概要」、および付録 A「共通オブジェクト・ファイル・フォーマット」に説明があります。ソース・ファイルには、次のアセンブリ言語要素を使用できます。

アセンブラ疑似命令	第 4 章で説明
マクロ疑似命令	第 5 章で説明
アセンブリ言語命令	*TMS320C54x™ Instruction Set Reference Guide* で説明

項目 **ページ**

3.1 アセンブラの概要 3-2
3.2 アセンブラと開発フロー 3-3
3.3 アセンブラの起動方法 3-4
3.4 アセンブラ入力のための代替ファイルと代替ディレクトリの命名方法 3-8
3.5 ソース文のフォーマット 3-11
3.6 定数 3-15
3.7 文字列 3-18
3.8 シンボル 3-19
3.9 式 3-25
3.10 ビルトイン関数 3-30
3.11 値を拡張プログラム・メモリへロードする方法 3-32
3.12 ソース・リスト 3-33
3.13 クロスリファレンス・リスト 3-37

3.1 アセンブラの概要

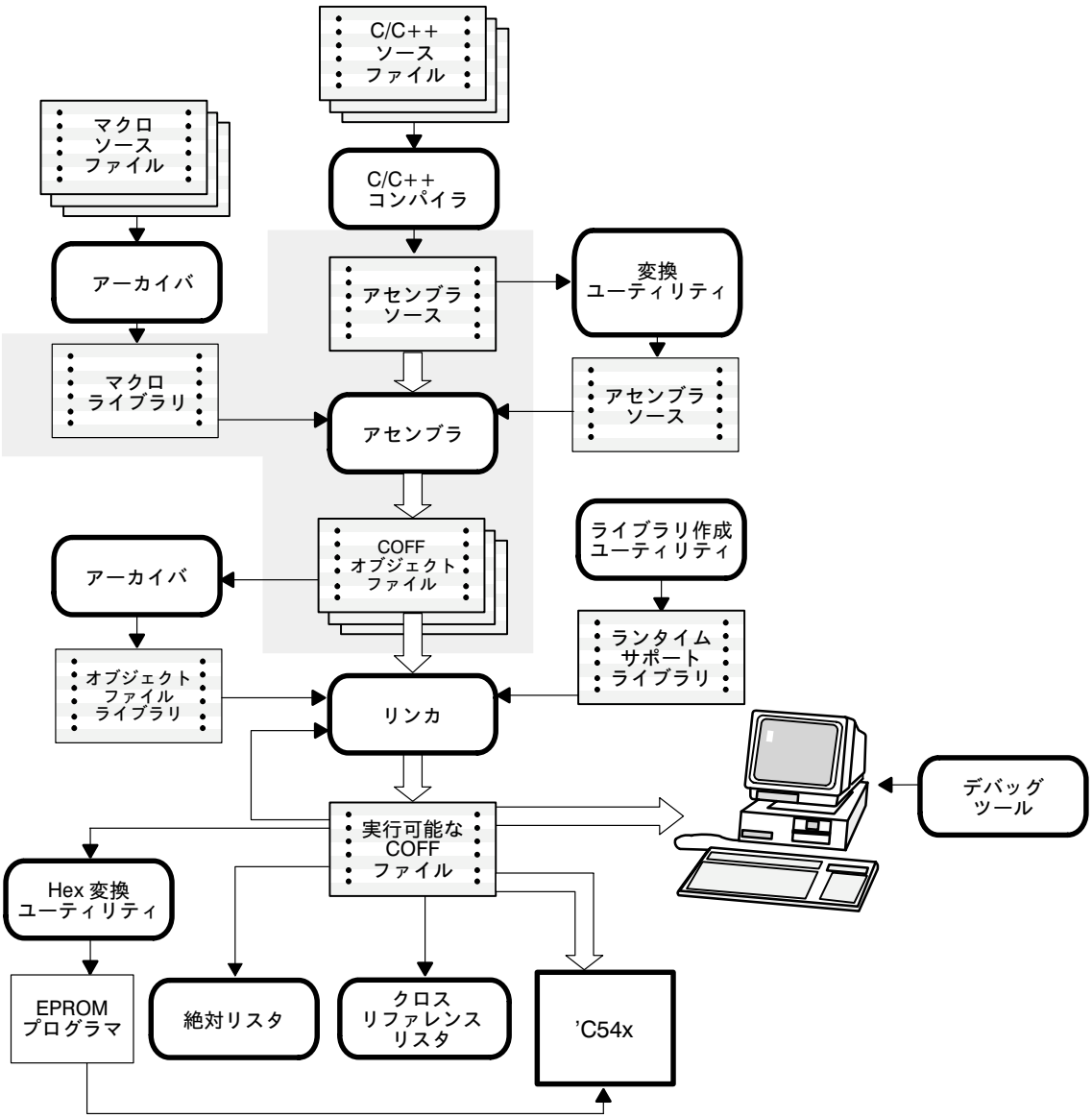
2 パス・アセンブラには次の機能があります。

- ❑ 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 C54x オブジェクト・ファイルを作成する。
- ❑ ソース・リストを必要に応じて作成し、ユーザにリストの制御手段を与える。
- ❑ コードを複数のセクションに分割し、オブジェクト・コードの各セクションに対応する SPC (セクション・プログラム・カウンタ) を保守する。
- ❑ グローバル・シンボルの定義と参照を行い、(必要に応じて) ソース・リストにクロスリファレンス・リストを追加する。
- ❑ 条件ブロックをアセンブルする。
- ❑ マクロをサポートすることによって、インラインかまたはライブラリで、マクロのユーザ定義ができるようにする。

3.2 アセンブラと開発フロー

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、アセンブリ言語自体により作成されたものでも、C/C++ コンパイラにより作成されたものでも、アセンブリ言語のソース・ファイルを入力として受け入れます。

図 3−1. アセンブラと開発フロー



3.3 アセンブラの起動方法

アセンブラを起動するには、次のように入力します。

asm500 [*input file* [*object file* [*listing file*]]] [*–options*]

asm500 アセンブラを起動する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。拡張子を付けないと、−f アセンブラ・オプションが使用されていない場合、アセンブラはデフォルトの拡張子として *.asm* を使用します。入力ファイル名を指定しないと、アセンブラはプロンプトを出して指定を促します。

ソース・ファイルには、ニーモニックまたは代数表記命令のどちらかを入れますが、両方を入れることはできません。デフォルトの命令セットは、ニーモニックです。代数表記命令セットを指定するには、−mg オプションを使用します。

object file アセンブラが作成する C54x 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、アセンブラはデフォルトの *.obj* を付けます。オブジェクト・ファイルを指定しないと、アセンブラは入力ファイル名に拡張子 *.obj* を付けたファイルを作成します。

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

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

options 使用するアセンブラ・オプションを指定します。オプションは、大文字と小文字の区別はありません。アセンブラ名の後ろであれば、コマンド行のどこにでも指定できます。個々のオプションの前にはハイフンを付けます。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、別々に指定する必要があります。

−@ −@*filename* は、*filename* の内容をコマンド行に付けます。このオプションは、ホスト・オペレーティング・システムにより課せられたコマンド行の長さの制限を無効にするために使用できます。コマンド・ファイル内では、組み込みのスペースまたはハイフンを含んでいるファイル名やオプション・パラメータは、引用符で囲まれる必要があります。
たとえば、“this-file.asm” というようにです。

−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わせて使用します。

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

−d **–d***name* [=*value*] は、*name* という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。詳細は、3-20 ページの 3.8.3 項「シンボル定数の定義 (−d オプション)」を参照してください。

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

−g ソース・デバッガでのアセンブラ・ソースのデバッグを可能にします。アセンブリ言語のソース・ファイルの各行ごとに、行情報が、COFF ファイルに出力されます。すでに .line 疑似命令を含むアセンブリ・コード (−g を指定して C/C++ コンパイラを実行することで生成されるコードなど) には、−g オプションを使用できないことに注意してください。

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

−hc **–hc** *filename* は、アセンブラに対し、アセンブリ・モジュールに指定されたファイルをコピーするように指示します。このファイルは、ソース・ファイル文の前に挿入されます。コピーされたファイルは、アセンブリ・リスト・ファイルに表示されます。

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

−i .copy、.include または .mlib 疑似命令で指定されたファイルをアセンブラが検索するディレクトリを指定します。−i オプションのフォーマットは −i *pathname* です。詳細は、3-8 ページの 3.4.1 項「−i アセンブラ・オプションの使用方法」を参照してください。

−l (「L」の小文字) リスト・ファイルを作成します。

−mf アセンブリのコール命令が拡張アドレッシングを使用するように指定します。

−mg ソース・ファイルには代数表記命令が含まれることを指定します。

−q (静的な実行モード) 見出しとすべての進捗情報を抑止します。

−r **−r**[*num*] は、*num* で指定されるアセンブラの注釈を抑止します。注釈はアセンブラの情報メッセージであり、警告ほど深刻ではありません。*num* の値を指定しない場合は、すべての注釈が抑止されます。

−pw アセンブリ・コード上のパイプライン・コンフリクトに関する警告を生成します。アセンブラはすべてのパイプライン・コンフリクトを検出することができるわけではありません。パイプライン・コンフリクトは、直線的コードでのみ検出されます。パイプライン・コンフリクトを検出すると、アセンブラは警告を出力し、コンフリクトを解消するのに必要な (NOP や他の命令による) レイテンシ・スロット (ワード) を報告します。

−s 定義されたすべてのシンボルを、オブジェクト・ファイルのシンボル・テーブルに入れます。通常、アセンブラは、グローバル・シンボルだけをシンボル・テーブルに入れます。−s を使うと、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。

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

−v −v *value* 値は、命令作成の対象となるプロセッサを決定します。*value* には、541、542、543、545、545lp、546lp、548、549 のいずれかを使用します。

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

3.4 アセンブラ入力のための代替ファイルと代替ディレクトリの命名方法

.copy、.include、および .mlib 疑似命令は、アセンブラに対して外部ファイルからのコードを使用するように指示します。.copy および .include 疑似命令は、アセンブラに対して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、および .mlib の疑似命令の例を記載しています。これらの疑似命令の構文は次のとおりです。

.copy *"filename"*
.include *"filename"*
.mlib *"filename"*

filename には、アセンブラが文を読み取るコピー/インクルード・ファイルの名前、またはマクロ定義の入ったマクロ・ライブラリの名前が入ります。filename は、完全パス名、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、以下の順序でファイルを検索します。

1) 現行のソース・ファイルを含んでいるディレクトリ。現行のソース・ファイルとは、.copy、.include、または .mlib のいずれかの疑似命令が見つかったときにアセンブル中だったファイルを指します。

2) –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) C54X_A_DIR および A_DIR 環境変数を使って設定したすべてのディレクトリ

4) C54X_C_DIR および C_DIR 環境変数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 C54X_A_DIR および A_DIR 環境変数を使用することにより、補強することができます。

3.4.1 –i アセンブラ・オプションの使用方法

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

asm500 –i*pathname source filename*

各 –i オプションは 1 つの pathname を指定します。指定できるパス数には制限がありません。アセンブリ・ソースでは、パス情報を指定せずに .copy、.include、または .mlib 疑似命令を使うことができます。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でファイルが見つからない場合、–i オプションで指定されたパスを検索します。

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

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

ファイルは以下のディレクトリにあると仮定します。

Windows™　c:\tools\files\copy.asm

UNIX　/tools/files/copy.asm

O.S.	入力するコマンド
Windows	`asm500 -ic:\tools\files source.asm`
UNIX	`asm500 -i/tools/files source.asm`

source.asm が現行のディレクトリにあるため、アセンブラは、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。

3.4.2 C54X_A_DIR および A_DIR 環境変数の使用方法

環境変数とは、ユーザ定義によっ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、C54X_A_DIR および A_DIR 環境変数を使用してコピー/インクルード・ファイル、またはマクロ・ライブラリを含む代替ディレクトリの名前を指定します。

アセンブラは、最初に C54X_A_DIR 環境変数を検索し、次にその環境変数を読み込んで処理します。この変数が見つからない場合、アセンブラは A_DIR 環境変数を読み込んで処理します。両方の変数が設定されている場合は、プロセッサ固有の変数の設定が使用されます。プロセッサ固有の変数は、異なるプロセッサのために弊社のツールを同時に使用している場合に便利です。

アセンブラは、C54X_A_DIR およびまたは A_DIR を見つけられない場合は、次に C54X_C_DIR および C_DIR を探します。

環境変数を割り当てるためのコマンド構文には、次のようなものがあります。

O.S.	入力するコマンド
Windows	`set A_DIR= pathname;another pathname ...`
UNIX	`setenv A_DIR "pathname;another pathname ..."`

pathname にはコピー/インクルード・ファイル、またはマクロ・ライブラリを含んだディレクトリが入ります。このパス名は、セミコロンまたは空白文字で区切ることができます。アセンブリ・ソースでは、パス情報を指定せずに .copy、.include、または .mlib 疑似命令を使うことができます。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で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"
```

ファイルは以下のディレクトリに保管されていると仮定します。

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

次の表に示すコマンドを使用して、検索パスを設定します。

O.S.	入力するコマンド
Windows	`set A_DIR=c:\dsys` `asm500 –ic:\tools\files source.asm`
UNIX	`setenv A_DIR "/dsys"` `asm500 –i/tools/files source.asm`

source.asm が現行のディレクトリにあるため、アセンブラは、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 オプションを使用し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 を使用して指定されたディレクトリを検索して copy2.asm を検出します。

環境変数は、次にシステムをリブートするか、以下のコマンドを入力して変数をリセットするまでその設定は変わらないことに注意してください。

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

3.5 ソース文のフォーマット

TMS320C54x アセンブリ言語ソース・プログラムは、アセンブラ疑似命令、アセンブリ言語命令、マクロ疑似命令、およびコメントを含むことのできるソース文で構成されます。ソース文の 1 行はソース・ファイル・フォーマットで使える範囲であればいくら長くてもかまいません。

以下にソース文の例を示します。

(a) ニーモニック命令

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

(b) 代数表記命令

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

3.5.1 ソース文の構文

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

ニーモニックの構文:

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

代数表記の構文:

[*label*] [:] *instruction* [;*comment*]

以下の規則に従ってください。

- すべての文は必ず、ラベル、空白、アスタリスク、またはセミコロンのいずれかで始めなければなりません。
- アセンブラ疑似命令を含む文は、必ず 1 つの行で全体を指定しなければなりません。
- ラベルの使用は任意です。使用する場合には、1 カラム目から始めなければなりません。
- 各フィールドは、1 つ以上の空白で区切る必要があります。タブ文字は、空白と同じ働きをします。
- コメントの使用は任意です。1 カラム目から始まるコメントは、アスタリスクまたはセミコロン (* または ;) で始められますが、それ以外のカラムから始まるコメントは、必ずセミコロンで始めなければなりません。

3.5.2 ラベル・フィールド

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

ラベルを使用する場合は、その値はセクション・プログラム・カウンタ (SPC) 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 000000       ; Assume some other code was assembled.
10 000040 000A  Start:  .word 0Ah,3,7
   000041 0003
   000042 0007
```

1 つの行がラベルだけでできていても、それは 1 つの有効な文です。セクション・プログラム・カウンタの現行の値がラベルに割り当てられます。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のアドレスに割り当てられます (SPC はインクリメントされません)。

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

3.5.3 ニーモニック命令フィールド

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

3.5.3.1 ニーモニック・フィールド

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

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

3.5.3.2 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用できるのは、定数 (3-15 ページの 3.6 節「定数」を参照)、シンボル (3-19 ページの 3.8 節「シンボル」を参照)、または式の中の定数とシンボルの組み合わせ (3-25 ページの 3.9 節「式」を参照) です。オペランドとオペランドは、コンマで区切らなければなりません。

❑ **命令のオペランド接頭部**

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

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

```
Label:  ADD #123,B
```

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

■ *** 接頭部 — オペランドが間接アドレスであることを表します。*** 記号を接頭部として使用した場合、アセンブラは、オペランドを間接アドレスとして処理します。つまり、アセンブラは、オペランドの内容をアドレスとして使用します。次の例は、* 接頭部をもつオペランドを使用する命令の例です。

```
Label:  LD *AR4,A
```

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

❑ **疑似命令に対する即値**

即値モードは、主として命令に使われます。場合によっては、疑似命令のオペランドとともに使用することもできます。

通常、疑似命令に対しては即値モードを使用する必要はありません。以下の文を比較してください。

```
ADD #10, A
.byte 10
```

最初の文では、アセンブラに対して、値 10 をアキュムレータ A に加算することを通知するために即値モードが必要です。しかし 2 番目の文では、即値モードが使われていません。アセンブラは、オペランドが値であると想定して、1 バイトを値 10 で初期化します。

3.5.4 代数表記命令フィールド

代数表記命令フィールドは、ニーモニック構文で使用されるニーモニック・フィールドとオペランド・フィールドの組み合わせです。通常、ニーモニックの後にオペランドを続けて使うことはありません。むしろ、オペランドは文全体の一部となっています。次の項目は、代数表記構文において命令フィールドを使用する方法を示しています。

- ❏ 通常、オペランドはコンマで区切られません。ただし、一部の代数表記命令は、ニーモニックとオペランドから構成されます。この種の代数文では、オペランドを区切るためにコンマが使用されます。たとえば、lms (Xmem、Ymem) のようになります。
- ❏ 1 つのオペランドとして使用される複数項の式は、括弧で区切らなければなりません。この規則は、関数コール・フォーマットを使用する文には適用されません。このような文は、すでに括弧で区切られているからです。たとえば、A = B & #(1 << sym) << 5 という式があるとします。1 << sym という式は 1 つのオペランドとして使われるので、括弧によって区切られています。
- ❏ すべてのレジスタ名は、予約済みです。
- ❏ ニーモニックとオペランドから構成される代数表記命令の場合、ニーモニックは予約済みです。

3.5.5 コメント・フィールド

コメントは、どのカラムからも始めることができ、ソース行の最後まで続きます。コメントには、ASCII 文字と空白を使用することができます。コメントは、アセンブリ・ソース・リストには出力されますが、アセンブリには影響を与えません。

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

3.6 定数

アセンブラは、7 つの型の定数をサポートします。

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

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ます。定数は符号拡張されません。たとえば、定数 0FFH は、00FF (底 16)、または 255 (底 10) と同じです。−1 と同じではありません。

3.6.1 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。16 桁未満の数値を指定した場合は、アセンブラは、その数値を右揃えとし、指定されなかったビットをゼロで埋めます。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

3.6.2 8 進整数

8 進整数定数は、0 (ゼロ) という接頭部、あるいは Q または q という接尾部が付いた、最大 6 桁の 8 進数 (0～7) の文字列です。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 $8\,000_{16}$ に等しい定数
226q	150_{10} または 96_{16} に等しい整数

あるいは、8 進定数についての C 表記法を使用することもできます。

010	8_{10} または 8_{16} に等しい定数
0100000	$32\,768_{10}$ または $8\,000_{16}$ に等しい定数
0226	150_{10} または 96_{16} に等しい定数

3.6.3 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 $8\ 000_{16}$ に等しい定数
25	25_{10} または 19_{16} に等しい定数

3.6.4 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 ～ 9 の 10 進数と、A ～ F または a ～ f のアルファベットが使われます。16 進定数の先頭の文字は、10 進数 (0 ～ 9) 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

あるいは、16 進定数についてのC表記法を使用することもできます。

0x78	120_{10} または 0078_{16} に等しい定数
0x0F	15_{10} または $000F_{16}$ に等しい定数
0x37AC	$14\ 252_{10}$ または $37AC_{16}$ に等しい定数

3.6.5 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 文字で表示されます。文字定数の一部として単一引用符を使う場合は、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の文字定数も有効で、値 0 が割り当てられます。1 文字だけを指定した場合は、アセンブラは、ビットの右揃えを行います。有効な文字定数の例を以下に示します。

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

文字定数と文字列は同じではないことに注意してください (文字列については、3-18 ページの 3.7 節「文字列」で説明します)。文字定数が 1 つの整数値を表すのに対して、文字列は文字のリストです。

3.6.6 浮動小数点定数

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

[+|–] [*nnn*] . [*nnn* [E|e [+|–] *nnn*]]

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

3.0
3.14
.3
−0.314e13
+314.59e−2

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

有効な文字列の例を以下に示します。

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

文字列は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

3.8 シンボル

シンボルは、ラベル、定数、および置換シンボルに使用されます。シンボル名は、200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・オプションを使用すると、大文字と小文字は区別されなくなります。この種類のシンボルは、.global 疑似命令を使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ル時にのみ有効です。

3.8.1 ラベル

シンボルをラベルとして使用すると、プログラム内の位置に関連付けられたシンボル・アドレスとなります。ファイルの中でローカルに使われるラベルは、固有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、または .bss の疑似命令のオペランドとして使用することもできます。以下に例を示します。

```
          .global   label1

label2    NOP
          ADD       label1,B
          B         label2
```

3.8.2 シンボル定数

シンボルは定数値に設定することができます。定数を使用すると、意味のある名前を定数値と同様に使用することができます。.set および .struct/.tag/.endstruct の疑似命令を使うと、定数をシンボル名に設定できます。シンボル定数は再定義することができません。これらの疑似命令の使い方を以下に示します。

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

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

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

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

3.8.3 シンボル定数の定義 (–d オプション)

–d オプションにより、定数値とシンボルを等価にすることができます。このオプションで定義したシンボルは、アセンブリ・ソースの中で値の代わりに使うことができます。–d オプションのフォーマットは次のとおりです。

asm500 –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* を省略すると、シンボルは 1 に設定されます。

アセンブラ・ソース内では、次の疑似命令を使用してシンボルをテストできます。

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

$isdefed 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ください。引用符で囲むことによって、引数は、置換シンボルとしてでなく、そのままの文字列として解釈されます。

3.8.4 事前定義されたシンボル定数

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

- セクション・プログラム・カウンタ (SPC) の現行の値を表示する **$** (ドル記号文字)
- AR0 ～ AR7 を含む**レジスタ・シンボル**
- アセンブラによりシンボルとして設定された**メモリマップド・レジスタ**

3.8.5 置換シンボル

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

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

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

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

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

       LD ADDRA, A
       ADD ADDRB, A
       STL A, ADDRB

       .endm

; add2 invocation
       add2 LOC1, LOC2         ;add ”LOC1” argument to a
                               ;second argument ”LOC2”.
; instructions for expanded macro:
;      LD LOC1, A
;      ADD LOC2, A
;      STL A, LOC2
```

マクロについての詳細は、第 5 章「マクロ言語」を参照してください。

3.8.6 ローカル・ラベル

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

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

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

ローカル・ラベルは、次のいずれかの方法で未定義に、またはリセットすることができます。

- .newblock 疑似命令を使用する。
- セクションを変更する (.sect、.text または .data 疑似命令を使用)。
- インクルード・ファイル (.include または .copy 疑似命令で指定された) を入力する。
- インクルード・ファイル (インクルード・ファイルの終りに達する) を去る。

例 3−1 は、ローカル・ラベルの $*n* の形式の使用例です。この例では、シンボル ADDRA、ADDRB、ADDRC がすでに定義されていると仮定しています。

例 3−1. $*n* ローカル・ラベル

(a) ローカル・ラベルを正しく使用したコード

```
Label1:   LD ADDRA, A    ; Load Address A to Accumulator A.
          SUB ADDRB, A   ; Subtract Address B.
          BC $1, ALT     ; If less than zero, branch to $1;
          LD ADDRB, A    ; otherwise, load ADDRB to A
          B $2           ; and branch to $2.
$1        LD ADDRA, A    ; $1: load ADDRA to Accumulator A.
$2        ADD ADDRC, A   ; $2: add ADDRC.
          .newblock      ; Undefine $1 so it can be used
                         ; again.
          BC $1, ALT     ; If less than zero, branch to $1.
          STL A, ADDRC   ; Store ACC low in ADDRC.
$1        NOP
```

(b) ローカル・ラベルを不正に使用したコード

```
Label1:   LD ADDRA, A    ; Load Address A to Accumulator A.
          SUB ADDRB, A   ; Subtract Address B.
          BC $1, ALT     ; If less than zero, branch to $1;
          LD ADDRB, A    ; otherwise, load ADDRB To A
          B $2           ; and Branch to $2.
$1        LD ADDRA, A    ; $1: Load ADDRA To Accumulator A.
$2        ADD ADDRC, A   ; $2: Add ADDRC.
          BC $1, ALT     ; If less than 0, branch to $1.
          STL A, ADDRC   ; Store Acc low in ADDRC.
$1        NOP            ; WRONG: $1 is multiply defined.
```

ローカル・ラベルは、マクロ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、アセンブラは重複定義エラーを発行します。しかし、マクロ内部にローカル・ラベルと .newblock を指定してあれば、マクロを展開するたびに、そのローカル・ラベルが使用され、リセットされます。

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

指定可能な最大ラベル長は、固有の接尾部の長さだけ短くなります。マクロの展開が 10 回未満の場合、最大ラベル長は 126 文字です。マクロの展開が 10 ～ 99 回の場合、最大ラベル長は 125 です。

例 3−2 は、ローカル・ラベルの *name?* の形式の使用例を示しています。

例 3−2.　name? ローカル・ラベル

```
;*********************************************************
;  First definition of local label 'mylab'
;*********************************************************
'
          nop
mylab?    nop
          b mylab?
;*********************************************************
;  Include file has second definition of 'mylab'
;*********************************************************
'
          .copy "a.inc"
;*********************************************************
;Third definition of 'mylab', reset upon exit from include
;*********************************************************
'
mylab?    nop
          b mylab?
;*********************************************************
;  Fourth definition of 'mylab' in macro, macros use
;  different namespace to avoid conflicts
;*********************************************************
'
mymac     .macro
mylab?    nop
          b mylab?
          .endm
;*********************************************************
;  Macro invocation
;*********************************************************
'
          mymac
;*********************************************************
;  Reference to third definition of 'mylab', note that
;  definition is not reset by macro invocation nor
;  conflicts with same name defined in macro
;*********************************************************
'
          b mylab?;
*********************************************************
;  Changing section, allowing fifth definition of 'mylab'
;*********************************************************
'
          .sect "Secto_One"
          nop
mylab?    .word 0
          nop
          nop
          b mylab?
;*********************************************************
;.newblock directive, allowing sixth definition of 'mylab'
;*********************************************************
'
          .newblock
mylab?    .word 0
          nop
          nop
          b mylab?
```

3.9 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定数とシンボルの列です。オペランドは、アセンブリ時定数またはリンク時に再配置可能なシンボルです。有効な式の値の範囲は、−32 768から 32 767 です。式の計算順序に影響を与える主要な 3 つの要素は、次のとおりです。

括弧

括弧で囲まれた式は必ず最初に計算します。

8 / (4 / 2) = 4、ただし 8 / 4 / 2 = 1 です。

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

優先順位グループ

'C54x アセンブラは、C 言語と同じ優先順位を使用します (表 3−1 で概要を説明)。これは、他の TMS320 アセンブラの優先順位とは異なります。括弧により式の計算順位が決められていない場合、優先順位の最も高い演算子が最初に計算されます。

8 + 4 / 2 = 10 (4 / 2 が最初に計算されます)

左から右への計算

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

8 / 4*2 = 4、ただし 8 / (4*2) = 1 です。

3.9.1 演算子

式で使用できる演算子を、表 3−1 に示します。

注: 優先順位における他の TMS320 アセンブラとの違い

他の TMS320 プロセッサの中には、TMS320C54x とは異なる優先順位を使用するものがあります。このため、同一のソース・コードから別の結果が発生することがあります。'C54x は、C 言語と同じ優先順位を使います。

表 3−1. 式で使用できる演算子 (優先順位)

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

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

3.9.2 式のオーバーフローとアンダーフロー

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

3.9.3 整合定義式

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

例 3−3. 整合定義式

(a) 有効な整合定義式

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

X        .set   50h

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

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

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

3.9.4 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		
<	小さい	<=	小さいか等しい
>	大きい	>=	大きいか等しい

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

3.9.5 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、および外部シンボルの有効な演算についてまとめたものです。式では、再配置可能シンボルや外部シンボルで乗算や除算を行うことはできません。また、別のセクションに対して再配置可能な未解決のシンボルを使用することはできません。

.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用することもできます。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、式の中で使用できます。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もつ式

A が... かつ	B が... のとき	A + B は... かつ	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用した式の例を、以下に示します。これらの例では、同じセクション内で定義された 4 つのシンボルを使用します。

```
.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 をもつように定義されています) が使用されています。最初の文は、値 51 をアキュムレータにロードします。

```
LD  #LAB1  +  ((4+3) *  7), A    ;  ACC A = 51
LD  #LAB1  +  4  +  (3*7), A     ;  ACC A = 27
```

❑ **例 2**

有効な式は、すべて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、絶対値にのみ使用することができます。再配置可能シンボルには適用できません。再配置可能シンボルを 1 つだけしか含まない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が、それに続くすべての式は不正です。

```
LD extern_1 – 10, B       ; Legal
LD 10–extern_1, B         ; Can't negate reloc. symbol
LD –(intern_1), B         ; Can't negate reloc. symbol
LD extern_1/10, B         ; / isn't additive operator
LD intern_1 + extern_1, B ; 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るので、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 + 絶対値の形式に整理できます。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。

```
LD intern_1 – intern_2 + extern_1, B   ; Legal
LD intern_1 + intern_2 + extern_1, B   ; Illegal
```

❑ **例 4**

外部シンボルの位置は、式を計算する上で重要です。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 extern_1 に加算しようとするからです。

```
LD intern_1 + extern_1 – intern_2, B   ; Illegal
```

3.10 ビルトイン関数

アセンブラは、変換および超越数学計算を行うための関数をサポートしています。表 3−3 に、ビルトイン関数を示します。*expr* は定数値でなければならないことに注意してください。アセンブラの非数学ビルトイン関数については、5-9 ページの表 5−1 を参照してください。

表 3−3.　アセンブラ・ビルトイン数学関数

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

表 3−3.　アセンブラ・ビルトイン数学関数 (続き)

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

3.11 値を拡張プログラム・メモリへロードする方法

アセンブラは、拡張プログラム・メモリに常駐する（または常駐の可能性のある）ラベル、関数等の値をロードするために、疑似命令 LDX を使用することを許しています。LDX は、24 ビット・アドレスの上位 8 ビットをロードするときに使われます。

たとえば、関数 F1 が拡張プログラム・メモリ（16 ビットではなく 24 ビット）にある場合、F1 の値やアドレスは、次のようにロードされます。

```
LDX    #F1,16,A      ;loads the upper 8 bits of the 24-bit
                     ;address of F1
OR     #F1,A,A       ;adds in the lower 16 bits of the
                     ;address of F1
BACC   A             ;all 24 bits of F1 have been loaded
                     ;into accumulator A
```

24 ビット・アドレス全体をロードするには、LDX と OR の両方を使う必要があることに注意してください。

3.12 ソース・リスト

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

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

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

フィールド 1: ソース文番号

行番号

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

インクルード・ファイル文字

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

ネスト・レベル数

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

フィールド 2: セクション・プログラム・カウンタ

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

フィールド 3: オブジェクト・コード

このフィールドには、16 進数で表したオブジェクト・コードが入ります。すべての機械命令と疑似命令は、このフィールドにオブジェクト・コードを出力します。また、フィールドの最後に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
%	複合再配置式

フィールド 4: ソース文フィールド

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

例 3–4 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 3−4.　アセンブラ・リスト

(a) ニーモニックの例

```
       1                        .global RESET, INT0, INT1, INT2
       2                        .global TINT, RINT, XINT, USER
       3                        .global ISR0, ISR1, ISR2
       4                        .global time, rcv, xmt, proc
       5
       6                initmac .macro
       7                * initialize macro
       8                        SSBX OVM            ; disable oflow
       9                        LD #0, DP           ; dp = 0
      10                        LD #7, ARP          ; arp = ar7
      11                        LD #037h, A         ; acc = 03fh
      12                        RSBX INTM           ; enable ints
      13                        .endm
      14                ****************************************
      15                *       Reset and interrupt vectors       *
      16                ****************************************
      17  000000                .sect ”reset”
      18  000000 F073   RESET:  B  init
          000001 0008+
      19  000002 F073   INT0:   B  ISR0
          000003 0000!
      20  000004 F073   INT1:   B  ISR1
          000005 0000!
      21  000006 F073   INT2:   B  ISR2
          000007 0000!
      22
      23                *
      24  000000                .sect ”ints”
      25  000000 F073   TINT    B  time
          000001 0000!
      26  000002 F073   RINT    B  rcv
          000003 0000!
      27  000004 F073   XINT    B  xmt
          000005 0000!
      28  000006 F073   USER    B  proc
          000007 0000!
      29                ****************************************
      30                *         Initialize processor.         *
      31                ****************************************
      32  000008        init:   initmac
1                       * initialize macro
1         000008 F7B9           SSBX OVM            ; disable oflow
1         000009 EA00           LD #0, DP           ; dp = 0
1         00000a F4A7           LD #7, ARP          ; arp = ar7
1         00000b E837           LD #037h, A         ; acc = 03fh
1         00000c F6BB           RSBX INTM           ; enable ints
```

フィールド 1　フィールド 2　フィールド 3　　フィールド 4

例 3−4.　アセンブラ・リスト (続き)

(b) 代数表記の例

```
       1                        .global RESET, INT0, INT1, INT2
       2                        .global TINT, RINT, XINT, USER
       3                        .global ISR0, ISR1, ISR2
       4                        .global time, rcv, xmt, proc
       5
       6                initmac .macro
       7                * initialize macro
       8                        OVM = 1             ; disable oflow
       9                        DP = #0             ; dp = 0
      10                        ARP = #7            ; arp = ar7
      11                        A = #037h           ; acc = 03fh
      12                        INTM = 0            ; enable ints
      13                        .endm
      14                ****************************************
      15                *       Reset and interrupt vectors       *
      16                ****************************************
      17  000000                .sect ”reset”
      18  000000 F073    RESET:  goto  init
          000001 0008+
      19  000002 F073    INT0:   goto  ISR0
          000003 0000!
      20  000004 F073    INT1:   goto  ISR1
          000005 0000!
      21  000006 F073    INT2:   goto  ISR2
          000007 0000!
      22
      23                *
      24  000000                .sect ”ints”
      25  000000 F073    TINT    goto  time
          000001 0000!
      26  000002 F073    RINT    goto  rcv
          000003 0000!
      27  000004 F073    XINT    goto  xmt
          000005 0000!
      28  000006 F073    USER    goto  proc
          000007 0000!
      29                ****************************************
      30                *         Initialize processor.          *
      31                ****************************************
      32  000008        init:   initmac
 1                      * initialize macro
 1        000008 F7B9           OVM = 1             ; disable oflow
 1        000009 EA00           DP = #0             ; dp = 0
 1        00000a F4A7           ARP = #7            ; arp = ar7
 1        00000b E837           A = #037h           ; acc = 03fh
 1        00000c F6BB           INTM = 0            ; enable ints
```

フィールド 1 フィールド 2　フィールド 3　　フィールド 4

3.13 クロスリファレンス・リスト

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

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

例 3−5.　クロスリファレンス・リストの例

```
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、アセンブル時に定義、または参照された各シンボルが入ります。

Value　このカラムには、シンボルに割り当てられた値を表す 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 3−4 に、このような文字と名前を示します。

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

表 3−4.　シンボル属性

文字または名前	意味
REF	外部参照 (.global シンボル)
UNDF	未定義
’	.text セクションで定義されたシンボル
”	.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

たとえば、以下のソース・ファイルは、

(a) incl0.asm

```
   .global   ABC
   nop
   nop
```

(b) incl1.asm

```
   .global   ABC
   ld  ABC, A
```

(c) incl2.asm

```
   .global   ABC
   stl    A,ABC
```

(d) xref.asm

```
start:
       .include ”incl0.asm”
       .include ”incl1.asm”
       add #10,A
       .include ”incl2.asm”

       nop
       nop
       b   start

       .global   start

       .bss   ABC,2
```

次のクロスリファレンス・リストを生成します。クロスリファレンス・リストの A は、incl0.asm (インクルードされる最初のファイル) を参照します。B は、incl1.asm を参照し、C は、incl2.asm を参照します。

```
xref.asm                    PAGE    1

 1 000000       start:
 2                  .include "incl0.asm"
 3                  .include "incl1.asm"
 4 000003 F000       add #10,A
   000004 000A
 5                  .include "incl2.asm"
 6
 7 000006 F495       nop
 8 000007 F495       nop
 9 000008 F073       b start
   000009 0000!
10
11                   .global start
12
13 000000            .bss ABC,2

xref.asm                    PAGE    2

LABEL           VALUE       DEFN    REF

.TMS320C540     000001       0
.TMS320C541     000000       0
.TMS320C542     000000       0
.TMS320C543     000000       0
.TMS320C544     000000       0
.TMS320C545     000000       0
.TMS320C545LP   000000       0
.TMS320C546     000000       0
.TMS320C546LP   000000       0
.TMS320C548     000000       0
.start          000000'      1
ABC             000000-      12    A  1  B  1  B  2
                                   C  1  C  2
__far_mode      000000       0
__lflags        000000       0
start              REF                  9  11
```

第 4 章

アセンブラ疑似命令

アセンブラ疑似命令は、プログラムにデータを提供し、アセンブリ・プロセスを制御します。アセンブラ疑似命令を使用すると、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。
- ❑ リスト上の表示を制御する。
- ❑ メモリの初期化を行う。
- ❑ 条件付きブロックをアセンブルする。
- ❑ グローバル変数を定義する。
- ❑ 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。

この章は 2 つの部分に分かれています。第 1 部 (4.1 節から 4.10 節) では、疑似命令を機能別に説明し、第 2 部 (4.11 節以降) では、疑似命令のリファレンスをアルファベット順に掲載しています。

項目 **ページ**

4.1 疑似命令のまとめ 4-2
4.2 TMS320C1x/C2x/C2xx/C5x アセンブラ疑似命令との互換性 4-8
4.3 セクションを定義する疑似命令 4-9
4.4 定数を初期化する疑似命令 4-11
4.5 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 4-15
4.6 出力リストのフォーマットを設定する疑似命令 4-17
4.7 他のファイルを参照する疑似命令 4-19
4.8 条件付きアセンブリ疑似命令 4-20
4.9 アセンブル時シンボル疑似命令 4-21
4.10 その他の疑似命令 4-23
4.11 疑似命令のリファレンス 4-25

4.1 疑似命令のまとめ

この節では、アセンブラ疑似命令をまとめています。

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

ここで説明しているアセンブラ疑似命令のほかに、TMS320C54x™ ソフトウェア・ツールは以下のような疑似命令もサポートしています。

- ❑ アセンブラは、いくつかのマクロ用の疑似命令を使用します。この章にマクロ疑似命令のリストがありますが、詳細は第 5 章「マクロ言語」で説明します。
- ❑ 絶対リスタも疑似命令を使用します。絶対リスト疑似命令は、ユーザ入力による命令ではなく、絶対リスタによってソース・プログラムに挿入されます。絶対リスト疑似命令については、第 8 章「絶対リスタの説明」で説明します。この章では説明を省きます。
- ❑ C/C++ コンパイラは、シンボリック・デバッグ用の疑似命令を使用します。他の疑似命令とは違い、ほとんどのアセンブリ言語プログラムではシンボリック・デバッグ疑似命令を使用しません。シンボリック・デバッグ疑似命令については、付録 B「シンボリック・ デバッグ疑似命令」で説明します。この章では説明を省きます。

注： 構文内のラベルとコメント

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

表 4−1. 疑似命令のまとめ

(a) セクションを定義する疑似命令

ニーモニックと構文	説明	ページ
.bss *symbol*, *size in words* [*, blocking*] [**,** *alignment*]	*size in words* に指定されたワード数を、.bss (初期化されていないデータ) セクション内に確保します。	4-30
.clink [”*section name*”]	現行のセクションまたは指定のセクションに対する条件付きリンクを可能にします。	4-34
.data	.data (初期化されたデータ) セクションにアセンブルします。	4-39
.sect ”*section name*”	名前付き (初期化された) セクションにアセンブルします。	4-80
.text	.text (実行可能コード) セクションにアセンブルします。	4-90
symbol **.usect** ”*section name*”**,** *size in words* [**,** *blocking*] [,*alignment flag*]	*size in words* に指定されたワード数を名前付き (初期化されていない) セクション内に確保します。	4-95

表 4−1.　アセンブラ疑似命令のまとめ (続き)

(b) 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	ページ
.bes *size in bits*	*size in bits* に指定されたビット数を現行のセクション内に確保します。ラベルは、確保された空間内の最後のアドレス可能ワードを指します。	4-82
.byte *value*$_1$ [**,** ... **,** *value*$_n$] **.char** *value*$_1$ [**,** ... **,** *value*$_n$]	現行のセクション内の 1 つ以上の連続したワードを初期化します。	4-33
.double *value*$_1$ [**,** ... **,** *value*$_n$] **.ldouble** *value*$_1$ [**,** ... **,** *value*$_n$]	1 つ以上の 64 ビット、IEEE 倍精度、浮動小数点定数を初期化します。	4-40
.field *value* [**,** *size in bits*]	可変長フィールドを初期化します。	4-47
.float *value* [**,** ... **,** *value*$_n$]	1 つ以上の 32 ビット、IEEE 単精度、浮動小数点定数を初期化します。	4-50
.half *value*$_1$ [**,** ... **,** *value*$_n$] **.short** *value*$_1$ [**,** ... **,** *value*$_n$]	1 つ以上の 16 ビット整数を初期化します。	4-54
.int *value*$_1$ [**,** ... **,** *value*$_n$]	1 つ以上の 16 ビット整数を初期化します。	4-58
.long *value*$_1$ [**,** ... **,** *value*$_n$]	1 つ以上の 32 ビット整数を初期化します。	4-64
.pstring "*string*$_1$**"** [**,** ... **,"***string*$_n$**"**]	1 つ以上の (パックされた) テキスト文字列を初期化します。	4-85
.space *size in bits;*	*size in bits* に指定されたビット数を現行のセクション内に確保します。ラベルは、確保された空間の始まりを指します。	4-82
.string "*string*$_1$**"** [**,** ... **, "***string*$_n$**"**]	1 つ以上のテキスト文字列を初期化します。	4-85
.ubyte *value*$_1$ [**,** ... **,** *value*$_n$] **.uchar** *value*$_1$ [**,** ... **,** *value*$_n$]	現行のセクション内の 1 つ以上の連続したワードを初期化します。	4-33
.uhalf *value*$_1$ [**,** ... **,** *value*$_n$] **.ushort** *value*$_1$ [**,** ... **,** *value*$_n$]	1 つ以上の符号なしの 16 ビット整数を初期化します。	4-54
.uint *value*$_1$ [**,** ... **,** *value*$_n$]	1 つ以上の符号なしの 16 ビット整数を初期化します。	4-58
.ulong *value*$_1$ [**,** ... **,** *value*$_n$]	1 つ以上の符号なしの 32 ビット整数を初期化します。	4-64
.uword *value*$_1$ [, ... , *value*$_n$]	1 つ以上の符号なしの 16 ビット整数を初期化します。	4-58
.word *value*$_1$ [, ... , *value*$_n$]	1 つ以上の 16 ビット整数を初期化します。	4-58
.xfloat *value*$_1$ [, ...,*value*$_n$]	1 つ以上の 32 ビット、IEEE 単精度、浮動小数点定数を初期化しますが、倍長ワード境界への位置合わせを行いません。	4-50
.xlong *value*$_1$ [, ...,*value*$_n$]	1 つ以上の 32 ビット整数を初期化しますが、倍長ワード境界への位置合わせを行いません。	4-64

表 4−1.　アセンブラ疑似命令のまとめ (続き)

(c) 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	ページ
.align [*size in words*]	パラメータで指定されたワード境界に SPC を位置合わせします。パラメータは 2 の累乗、またはデフォルト (指定なし：ページ境界) にします。ワード (1)、偶数 (2) などに位置合わせします。	4-27
.even	SPC をワード境界に位置合わせします。	4-27

(d) 出力リストのフォーマットを設定する疑似命令

ニーモニックと構文	説明	ページ
.drlist	すべての疑似命令行のリスト出力を可能にします (デフォルト)。	4-41
.drnolist	特定の疑似命令行のリスト出力を抑止します。	4-41
.fclist	偽の条件付きコード・ブロックのリスト出力を許可します (デフォルト)。	4-46
.fcnolist	偽の条件付きコード・ブロックのリスト出力を抑止します。	4-46
.length *page length*	ソース・リストのページの長さを設定します。	4-61
.list	ソース・リストの出力を再開します。	4-62
.mlist	マクロ・リストとループ・ブロックの出力を許可します (デフォルト)。	4-70
.mnolist	マクロ・リストとループ・ブロックの出力を抑止します。	4-70
.nolist	ソース・リストの出力を停止します。	4-62
.option {**B** | **L** | **M** | **R** | **T** | **W** | **X**}	出力リスト・オプションを選択します。	4-76
.page	ソース・リストのページ替えを行います。	4-78
.sslist	置換シンボルの展開リスト出力を許可します。	4-83
.ssnolist	置換シンボルの展開リスト出力を抑止します (デフォルト)。	4-83
.tab *size*	タブ・サイズを設定します。	4-89
.title "*string***"**	リストのページ見出しにタイトルを出力します。	4-91
.width *page width*	ソース・リストのページの幅を設定します。	4-61

表 4−1.　アセンブラ疑似命令のまとめ (続き)

(e) 他のファイルを参照する疑似命令

ニーモニックと構文	説明	ページ
.copy ["]*filename*["]	別のファイルからソース文をインクルードします。	4-36
.def $symbol_1$ [**,** ... **,** $symbol_n$]	現行のモジュール内に定義されていて、他のモジュールで使用される 1 つ以上のシンボルを特定します。	4-51
.global $symbol_1$ [**,** ... **,** $symbol_n$]	1 つ以上のグローバル (外部) シンボルを特定します。	4-51
.include ["]*filename*["]	別のファイルからソース文をインクルードします。	4-36
.ref $symbol_1$ [**,** ... **,** $symbol_n$]	現行のモジュールで使用されているが、他のモジュールで定義される 1 つ以上のシンボルを特定します。	4-51

(f) マクロを定義する疑似命令

ニーモニックと構文	説明	ページ
.macro	ソース文がマクロ定義の第 1 行目であることを特定します。.macro は、必ずオペコード・フィールドに置いてください。	4-67
.mlib ["]*filename*["]	マクロ・ライブラリを定義します。	4-68
.mexit	.endm に移動します。この疑似命令は、エラーのテストでマクロ展開の失敗が確認された場合に使うと便利です。	5-3
.endm	.macro コード・ブロックを終了します。	4-67
.var	ローカルなマクロ置換シンボルを定義します。	4-98

(g) 条件付きアセンブリを制御する疑似命令

ニーモニックと構文	説明	ページ
.break [*well-defined expression*]	条件が真の場合、.loop のアセンブリを終了します。.break の使用はオプションです。	4-66
.else	.if 条件が偽の場合、コード・ブロックをアセンブルします。.else の使用はオプションです。この疑似命令は、デフォルトの場合として条件付きブロックで使用することができます。	4-56
.elseif *well-defined expression*	.if 条件が偽で、なおかつ .elseif 条件が真の場合、コード・ブロックをアセンブルします。.elseif の使用はオプションです。	4-56
.endif	.if コード・ブロックを終了します。	4-56
.endloop	.loop コード・ブロックを終了します。	4-66
.if *well-defined expression*	条件が真の場合、コード・ブロックをアセンブルします。	4-56
.loop [*well-defined expression*]	コード・ブロックの反復アセンブリを開始します。*well-defined expression* は、ループ・カウントです。	4-66

表 4−1.　アセンブラ疑似命令のまとめ (続き)

(h) アセンブル時にシンボルを定義する疑似命令

ニーモニックと構文	説明	ページ
.asg [**"**]*character string*[**"**], *substitution symbol*	置換シンボルに文字列を割り当てます。	4-28
.endstruct	構造体の定義を終了します。	4-86
.endunion	共用体の定義を終了します。	4-92
symbol **.equ** *value*	値をシンボルと等価にします。	4-81
.eval *well-defined expression*, *substitution symbol*	数値型の置換シンボルの算術計算を行います。	4-28
.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。	4-60
symbol **.set** *value*	値をシンボルと等価にします。	4-81
.struct	構造体の定義を開始します。	4-86
.tag	構造体属性をラベルに割り当てます。	4-86
.union	共用体の定義を開始します。	4-92

表 4−1.　アセンブラ疑似命令のまとめ (続き)

(i)　その他の疑似命令

ニーモニックと構文	説明	ページ
.algebraic	ファイルが代数表記のアセンブリ・ソースを含んでいることを示します。	4-26
.c_mode	コールと分岐が、通常の 16 ビットのアドレス範囲内にあることを示します。	4-35
.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。	4-42
.end	プログラムを終了します。	4-44
.far_mode	コールと分岐がファー・コールであることを示します。	4-45
.mmregs	メモリ・マップド・レジスタをシンボル・テーブルに格納します。	4-71
.mmsg *string*	ユーザ定義のメッセージを出力デバイスに送ります。	4-42
.newblock	ローカル・ラベルの定義を解除します。	4-74
.noremark [*num*]	アセンブラが *num* により特定されるアセンブラ注釈を抑止するコード・ブロックの始まりを確認します。*num* が指定されない場合、すべての注釈は抑止されます。	4-75
.remark [*num*]	.noremark により事前に抑止された注釈を生成するというデフォルト動作を再開します。	4-75
.sblock [**"**]*section name*[**"**] [, ... , "*section name*"]	ブロック化するセクションを指定します。	4-79
.version [*value*]	プロセッサ命令を作成するデバイスを指定します。	4-99
.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。	4-42

4.2 TMS320C1x/C2x/C2xx/C5x アセンブラ疑似命令との互換性

この節では、TMS320C54x アセンブラ疑似命令と TMS320C1x/C2x/C2xx/C5x アセンブラ疑似命令との相違点について説明します。

- ’C54x の .long 疑似命令と .float 疑似命令は、値の最上位のワードを下位アドレスに配置します。’C1x/C2x/C2xx/C5x アセンブラ疑似命令は、値の最下位のワードを下位アドレスに配置します。また、’C54x の .long 疑似命令と .float 疑似命令は、自動的に SPC を偶数ワード境界に位置合わせしますが、’C1x/C2x/C2xx/C5x アセンブラ疑似命令はこの位置合わせを行いません。
- 引数がない .align 疑似命令に対して、’C54x アセンブラと ’C1x/C2x/C2xx/C5x アセンブラは、ともに SPC を次の 128 ワード境界に位置合わせします。ただし、’C54x の .align 疑似命令は定数引数も受け入れます。この定数引数は 2 の累乗でなくてはなりません。SPC はこの引数で指定されたワード境界に位置合わせされます。
’C1x/C2x/C2xx/C5x アセンブラの .align 疑似命令は、この定数引数を受け入れません。
- ’C54x の .field 疑似命令は、ワードの最上位ビットから始まるワードにフィールドをパックします。’C1x/C2x/C2xx/C5x アセンブラの .field 疑似命令は、ワードの最下位のビットから始まるワードにフィールドを配置します。
- ’C54x の .field 疑似命令は、1 ～ 32 ビットの値を処理します。
’C1x/C2x/C2xx/C5x アセンブラは 1 ～ 16 ビットの値を処理します。’C54x アセンブラの場合、16 ビット以上のオブジェクトはワード境界から始まり、下位アドレスに最上位のビットが配置されます。
- ’C54x の .bss 疑似命令と .usect 疑似命令には、位置合わせフラグと呼ばれる追加フラグがあります。このフラグは、偶数ワード境界での位置合わせを指定するものです。’C1x/C2x/C2xx/C5x の .bss 疑似命令と .usect 疑似命令は、このフラグを使用しません。
- ’C54x の .string 疑似命令は、1 ワードあたり 1 文字を初期化します。
’C1x/C2x/C2xx/C5x アセンブラの .string 疑似命令は 1 ワードあたり 2 文字をパックします。新しい .pstring 疑似命令は、従来の .string 疑似命令と同じく 1 ワードあたり 2 文字をパックします。
- ’C54x アセンブラで新しく追加された疑似命令は、以下のとおりです。これらの疑似命令は、’C1x/C2x/C2xx/C5x アセンブラではサポートされていません。

疑似命令	使用方法
.xfloat	.float と同じです。ただし、自動位置合わせは行いません。
.xlong	.long と同じです。ただし、自動位置合わせは行いません。
.pstring	.string と同じです。ただし、1 ワードあたり 2 文字をパックします。

4.3 セクションを定義する疑似命令

次の疑似命令は、アセンブリ言語プログラムのさまざまな部分を適切なセクションと関連付けます。

- **.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。
- **.clink** は、名前付きセクションのタイプ・フィールドに STYP_CLINK フラグを設定します。.clink 疑似命令は、初期化されたセクションまたは初期化されていないセクションのいずれにも適用できます。STYP_CLINK フラグを設定すると、条件付きリンクが可能になります。これは、セクション内シンボルへの参照が検出されない場合に、そのセクションをリンカの最終 COFF 出力から除外するようにリンカに指示することにより可能になります。
- **.data** は、.data セクションのコード部分を特定します。.data セクションは、通常は初期化されたデータを含んでいます。
- **.sect** は、初期化された名前付きセクションを定義し、それに続くコードまたはデータをそのセクションに関連付けます。.sect で定義されたセクションは、実行可能なコードまたはデータを含みます。
- **.text** は、.text セクションのコード部分を特定します。.text セクションは、通常は実行可能コードを含んでいます。
- **.usect** は、初期化されていない名前付きセクションに空間を確保します。.usect 疑似命令の機能は .bss 疑似命令と似ています。ただし、.usect を使うと、.bss セクションとは別に空間を確保できます。

第 2 章「共通オブジェクト・ファイル・フォーマットの概要」で、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用してコードとデータを正しいセクションに関連付ける方法を示しています。この例は出力リストです。カラム 1 は行番号を示し、カラム 2 は SPC 値を示しています (各セクションには専用のプログラム・カウンタ、つまり SPC があります)。最初にコードがセクションに配置されるときの SPC は 0 です。他のコードをアセンブルした後で、セクションへのアセンブルを再開すると、そのセクションの SPC は、介在コードがなかったものとして、カウントを再開します。

例 4−1 の疑似命令は、以下のセクションに対して作業を実行します。

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

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

例 4−1.　セクションの疑似命令

```
 1                    ************************************************
 2                    *     Start assembling into the .text section    *
 3                    ************************************************
 4  000000                      .text
 5  000000 0001                 .word  1,2
    000001 0002
 6  000002 0003                 .word  3,4
    000003 0004
 7
 8                    ************************************************
 9                    *     Start assembling into the .data section    *
10                    ************************************************
11  000000                      .data
12  000000 0009                 .word  9, 10
    000001 000A
13  000002 000B                 .word  11, 12
    000003 000C
14
15                    ************************************************
16                    *       Start assembling into a named,           *
17                    *       initialized section, var_defs            *
18                    ************************************************
19  000000                      .sect  ”var_defs”
20  000000 0011                 .word  17, 18
    000001 0012
21
22                    ************************************************
23                    *    Resume assembling into the .data section    *
24                    ************************************************
25  000004                      .data
26  000004 000D                 .word  13, 14
    000005 000E
27  000000                      .bss   sym, 19   ; Reserve space in .bss
28  000006 000F                 .word  15, 16    ; Still in .data
    000007 0010
29
30                    ************************************************
31                    *    Resume assembling into the .text section    *
32                    ************************************************
33  000004                      .text
34  000004 0005                 .word  5, 6
    000005 0006
35  000000            usym      .usect ”xy”, 20   ; Reserve space in xy
36  000006 0007                 .word  7, 8      ; Still in .text
    000007 0008
```

4.4 定数を初期化する疑似命令

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

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

  ビット数を 16 で割ることによって、指定のワード数を確保できます。

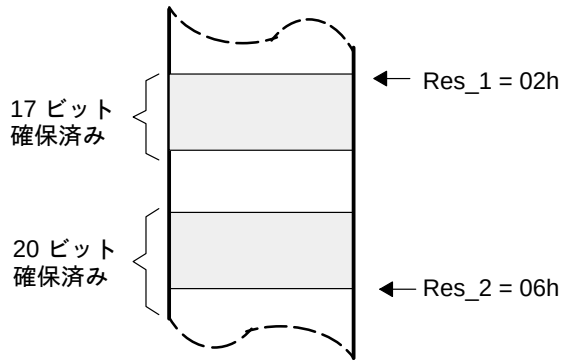
  - .space でラベルを使用すると、そのラベルは確保済みのビットを含む最初のワードを指します。
  - .bes でラベルを使用すると、そのラベルは確保済みのビットを含む最後のワードを指します。

  図 4−1 は .space 疑似命令と .bes 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
 2              ** .space and .bes directives
 3
 4 000000 0100          .word           100h, 200h
   000001 0200
 5 000002       Res_1:  .space          17
 6 000004 000f          .word           15
 7 000006       Res_2:  .bes            20
 8 000007 00ba          .byte           0BAh
 9              ** reserve 3 words
10 000008       Res_3:  .space          3*16
11 00000b 000a          .word           10
```

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

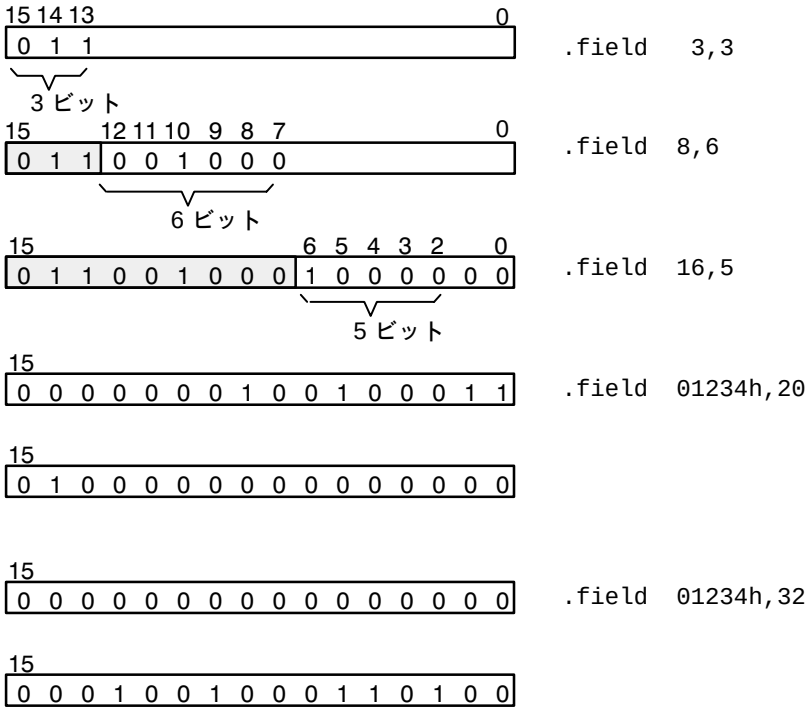


- **.byte**、**.ubyte**、**.char**、および **.uchar** 疑似命令は、1 つ以上の 8 ビット値を現行のセクションの連続するワードに格納します。これらの疑似命令は .word と .uword に似ていますが、各値の幅が 8 ビットに制限されている点が異なります。
- **.field** 疑似命令は、1 つの値を現行のワードの指定のビット数に入れます。.field を使用すると、複数のフィールドを単一のワードにパックできます。アセンブラは、1 つのワードが完全に埋まるまで SPC をインクリメントしません。

  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。最初の 3 つのフィールドで SPC は変わらないことに注目してください (つまり、これらのフィールドは同じワードにパックされます)。

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

図 4−2.　.field 疑似命令



- **.float** と **.xfloat** 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、その結果を現行のセクションの連続する 2 ワードに格納します。最上

位のワードを最初に格納します。.float 疑似命令は最も近くの倍長ワード境界に自動的に位置合わせを行いますが、.xfloat はこの位置合わせを行いません。

- **.int**、**.uint**、**.half**、**.uhalf**、**.short**、**.ushort**、**.word** および **.uword** は、1 つ以上の 16 ビット値を現行のセクション内の連続するワードに入れます。
- **.double** と **.ldouble** は、 1 つ以上の浮動小数点値の単精度 (32 ビット) IEEE 浮動小数点表記を計算して、その結果を現行のセクション内の連続する 2 ワードに格納します。.double 疑似命令は、自動的に倍長ワード境界に位置合わせします。
- **.long**、**.ulong** および **.xlong** は、32 ビット値を現行のセクション内の連続した 2 ワードに格納します。最上位のワードを最初に格納します。.long 疑似命令は倍長ワード境界に自動的に位置合わせを行いますが、.xlong は位置合わせを行いません。
- **.string** と **.pstring** は、1 つ以上の文字列内の 8 ビット文字を現行のセクションに格納します。.string 疑似命令は、.byte と同様、8 ビット文字を現行のセクションの各連続ワードに格納します。.pstring も 8 ビットの幅をもっていますが、1 ワードに 2 文字をパックします。.pstring の場合、文字列内の最終ワードには、必要に応じてヌル文字 (0) が埋め込まれます。

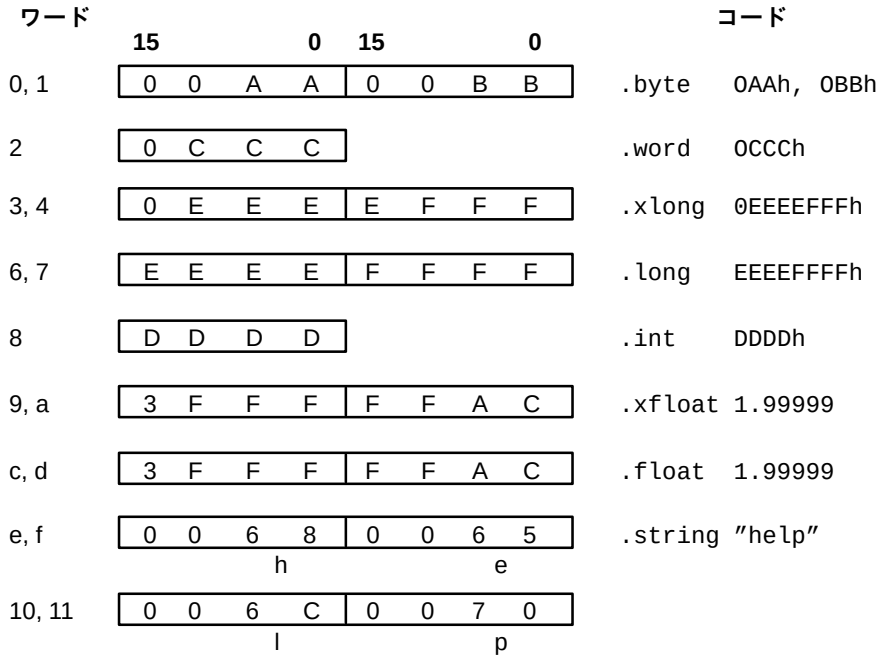
注： .struct/.endstruct シーケンスにおけるこれらの疑似命令

上記の疑似命令が .struct/.endstruct シーケンスの一部である場合、メモリは初期化されず、メンバのサイズが定義されます。.struct/.endstruct 疑似命令の詳細は、4-21 ページの 4.9 節「アセンブル時シンボル疑似命令」を参照してください。

図 4−3 では、.byte、.int、.long、.xlong、.float、.xfloat、.word、および .string の各疑似命令を比較しています。この例では、以下のコードをアセンブルしています。

```
1 000000 00aa          .byte           0AAh, 0BBh
  000001 00bb
2 000002 0ccc          .word           0CCCh
3 000003 0eee          .xlong          0EEEEFFFh
  000004 efff
4 000006 eeee          .long           0EEEEFFFFh
  000007 ffff
5 000008 dddd          .int            0DDDDh
6 000009 3fff          .xfloat         1.99999
  00000a ffac
7 00000c 3fff          .float          1.99999
  00000d ffac
8 00000e 0068          .string         ”help”
  00000f 0065
  000010 006c
  000011 0070
```

図 4−3.　初期化疑似命令



4.5 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、SPC を 1 ワードから 128 ワードの境界に位置合わせします。その結果、疑似命令の後に続くコードは x ワード境界またはページ境界から始まります。SPC が、すでに選択した境界の上に位置合わせされている場合は、SPC はインクリメントされません。.align 疑似命令に対するオペランドは、2^0 ～ 2^{16} の間の 2 の累乗と等しくなります (ただし、2^7 を超えた疑似命令は意味をもちません)。以下に例を示します。

オペランドが	1	の場合、SPC をワード境界に位置合わせします。
	2	の場合、SPC を倍長ワード/偶数境界に位置合わせします。
	128	の場合、SPC をページ境界に位置合わせします。

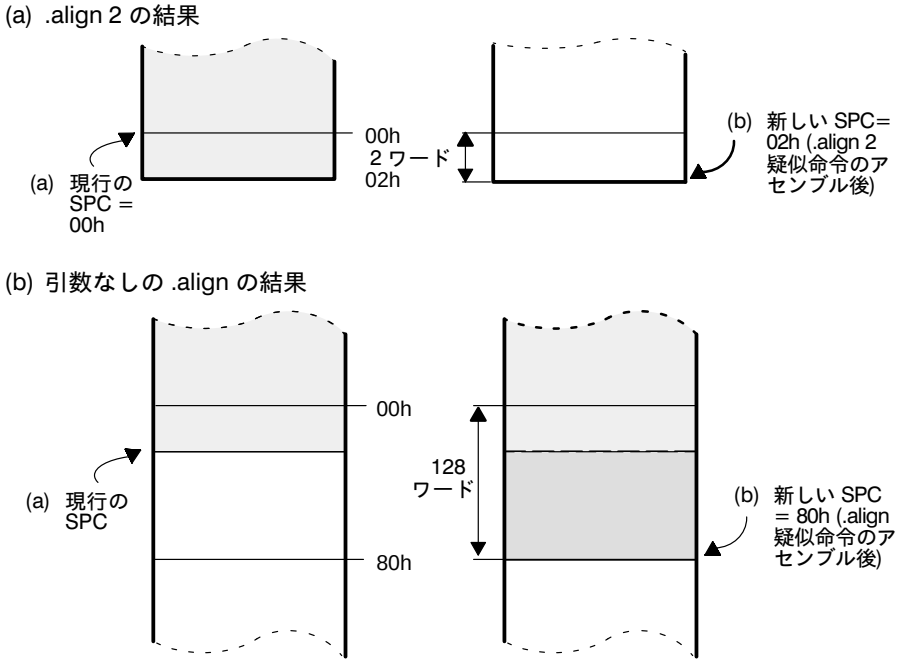
.align 疑似命令にオペランドがない場合、デフォルトの位置合わせはページ境界に設定されます。

.even 疑似命令は、SPC を、次のワード境界を指すように位置合わせします。これは、オペランドが 1 の .align 疑似命令を指定するのと同じことです。オペランドが 2 の .align を使用すると、SPC は次の倍長ワード境界に位置合わせされます。現行のワード内の未使用ビットには、ゼロ (0) が埋め込まれます。

図 4−4 は、.align 疑似命令を表しています。この例では、以下のコードをアセンブルしています。

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

図 4−4.　.align 疑似命令



4.6 出力リストのフォーマットを設定する疑似命令

リスト・ファイルのフォーマットを設定する疑似命令には、次のものがあります。

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

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

  リスト・ファイルへの出力を再開させるには、**.drlist** 疑似命令を使用します。

- リスト・ファイルは、コードを生成しない、偽の条件付きブロックのリストを含んでいます。**.fclist** および **.fcnolist** 疑似命令は、このリスト作成の始動と停止を行います。.fclist 疑似命令を使用すると、ソース・コードどおりに偽の条件付きブロックのリストを作成できます。これはアセンブラのデフォルトの動作です。.fcnolist 疑似命令を使用すると、実際にアセンブルされる条件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。この疑似命令を使用して、さまざまな出力デバイス用にリストを調整できます。

- **.list** および **.nolist** 疑似命令は、リスト・ファイルへの出力の始動と停止を行います。.nolist 疑似命令により、アセンブラが選択したソース文をリスト・ファイルに出力しないようにできます。.list 疑似命令により、リスト・ファイルへの出力を再開させることができます。

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

- **.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できます。

  A すべての疑似命令とデータ、および以降の拡張子、マクロ、ブロックのリスト作成を再開します。

  B .byte 疑似命令によるリスト作成を 1 行に制限します。

  D 特定の疑似命令 (.drnolist と同じ働きをする) のリスト作成を停止します。

  H .half および .short 疑似命令のリスト作成を 1 つの行に制限します。

  L .long 疑似命令によるリスト作成を 1 行に制限します。

  M リストでのマクロ展開を停止します。

  N リスト作成 (.nolistを実行する) を停止します。

O リスト作成 (.listを実行する)を再開します。

R B、M、T、およびW オプションをリセットします。

T .string 疑似命令によるリスト作成を 1 行に制限します。

W .word 疑似命令によるリスト作成を 1 行に制限します。

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

- **.page** 疑似命令により、出力リストのページ替えを行うことができます。
- **.sslist** および **.ssnolist** 疑似命令を使用すると、展開された置換シンボルのリスト作成の始動と停止を行うことができます。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うときに便利です。
- **.tab** 疑似命令を使用すると、タブ・サイズを定義できます。
- **.title** 疑似命令を使用すると、アセンブラは各ページの 1 行目に出力するタイトルを設定できます。
- **.width** 疑似命令を使用すると、リスト・ファイルのページ幅を制御できます。この疑似命令を使用すると、さまざまな出力デバイス用にリストを調整できます。

4.7 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** および **.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、.copy または .include 疑似命令が発生した地点にすぐに続く現行のファイルからのソース文の読み取りを再開します。コピー・ファイルから読み取られた文は、リスト・ファイルに出力されます。インクルード・ファイルから読み取られた文は、リスト・ファイルに出力されません。
- **.def** 疑似命令は、現行のモジュールで定義されているシンボルで、かつ他のモジュールで使用できるシンボルを識別します。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用できるようにします (外部シンボルの詳細は、2-21 ページの 2.8.1 項「外部シンボル」を参照してください)。.global 疑似命令は、定義されたシンボルに対しては .def として機能し、定義されていないシンボルに対しては .ref として機能するという二重の役割を果たします。リンカは、定義されていないグローバル・シンボルがプログラム内で使用されている場合のみ、そのグローバル・シンボルを解決します。
- **.ref** 疑似命令は、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使われているシンボルを識別します。アセンブラは、このようなシンボルを定義されていない外部シンボルとしてマークを付け、リンカがその定義を解決できるようにオブジェクト・シンボル・テーブルに入れます。

4.8 条件付きアセンブリ疑似命令

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

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

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

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

アセンブラは、条件式に対して便利な関係演算子をいくつかサポートしています。関係演算子の詳細は、3-27 ページの 3.9.4 項「条件式」を参照してください。

4.9 アセンブル時シンボル疑似命令

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

- **.asg** 疑似命令は、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
```

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

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

- **.label** 疑似命令は、現行のセクション内のロード・アドレスを参照する特殊シンボルを定義します。この疑似命令は、セクションをロードするアドレスと実行するアドレスが異なる場合に使用すると有効です。たとえば、パフォーマンスを大きく左右するコード・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、実行するときには、それを高速のオンチップ・メモリに移すことができます。

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

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

.set および .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、交換して使用することができます。

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

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

.tag 疑似命令は、構造体特性をラベル・シンボルに関連付けます。これにより、シンボリック表記は簡略化され、さらに他の構造体を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当ては行いません。構造体タグ (stag) を使用するには、最初にそれを定義しておく必要があります。

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

COORD .tag type         ; declare COORD (coordinate)
       ADD   COORD.Y, A

      .bss COORD, T_LEN ; actual memory allocation
```

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

  .tag 疑似命令は共用体特性をラベル・シンボルに関連付けます。そして、.tag 疑似命令はその共用体の開始点に関連付けられます。共用体を定義して、それにタグを付けることが可能です。その後、.tag 疑似命令を使用して、その共用体を構造体のメンバとして宣言できます。共用体はタグを付けずに宣言することもできますが、その場合は、すべてのメンバがシンボル・テーブルに格納されるので、各メンバには必ず固有の名前を付けなければなりません。また、共用体は構造体内で定義することも可能です。その場合、共用体への参照は、その共用体をもつ構造体を介して作成されなければなりません。以下に例を示します。

```
        .data
s2_tag  .struct       ;structure tag definition
         .union       ;union is first structure member
          .struct     ;structure is union member
h1         .half      ;h1, h2, and w1
h2         .uhalf     ;exist in the same memory
          .endstruct
w1        .word       ;word is another union member
         .endunion
w2       .word        ;second structure member
s2_len  .endstruct

XYZ     .tag  s2_tag
        .bss  XYZ,s2_len       ;declare instance of structure
        ADD   XYZ.h2, A
```

4.10 その他の疑似命令

以下の疑似命令は、その他のさまざまな機能を実行します。

- **.algebraic** 疑似命令は、ファイルに代数表記のアセンブリ・ソース・コードが含まれていることをアセンブラに伝えます。−mg アセンブラ・オプションを使用しない場合、この疑似命令をファイルの 1 行目に指定しなければなりません。
- **.c_mode** 疑似命令は、コールと分岐が通常の 16 ビットのアドレス範囲内にあることをアセンブラに伝えます。これは、アセンブラのデフォルトの動作です。
- **.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となります。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。
- **.far_mode** 疑似命令は、コールがファー・コールであることをアセンブラに伝えます。
- **.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 を使用するのは、すべてのメモリ・マップド・レジスタに対して .set を実行するのと同じです。メモリ・マップド・レジスタのリストについては、4-71 ページの表 4−2 を参照してください。
- **.newblock** 疑似命令は、ローカル・ラベルをリセットします。ローカル・ラベルは、$n または name? という書式のシンボルです。ローカル・ラベルは、ラベル・フィールドに現れた時点で定義されます。ローカル・ラベルは、ジャンプ命令のオペランドとして使用できる一時的なラベルです。.newblock 疑似命令は、ローカル・ラベルの定義を解除し、それによってローカル・ラベルの有効範囲を制限します。ローカル・ラベルに関する詳細は、3-22 ページの 3.8.6 項「ローカル・ラベル」を参照してください。
- **.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、ページ位置合わせに似た (ただしそれより制約の弱い) 機能をもつ、アドレス位置合わせメカニズムです。ブロック化されたセクションが 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ます。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対しブロック化を指定できることに注意してください。.usect または .bss の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。
- **.noremark** 疑似命令は、アセンブラが、特定されたアセンブラの注釈 (またはすべての注釈) の出力を抑止するコード・ブロックを開始します。注釈はアセンブラの情報メッセージであり、警告ほど深刻ではありません。**.remark** 疑似命令は、.noremark により事前に抑止された注釈の出力を再び可能にします。
- **.version** 疑似命令は、どのプロセッサに対して命令を構築するかを定めます。各 'C54x デバイスには、それ自体の値があります。

以下の 3 つの疑似命令を使用すると、独自のエラー・メッセージや警告メッセージを作成できます。

- **.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生成させます。つまり、エラー回数をインクリメントして、アセンブラがオブジェクト・ファイルを作成するのを停止させます。
- **.mmsg** 疑似命令は、アセンブル時メッセージを標準出力デバイスに送ります。.mmsg 疑似命令は、.emsg および .wmsg の疑似命令と同じ機能を果たしますが、エラー回数や警告回数をインクリメントしません。またオブジェクト・ファイルの作成に影響しません。
- **.wmsg** 疑似命令は、警告メッセージを標準出力デバイスに送ります。.wmsg 疑似命令は .emsg 疑似命令と同じ機能を果たしますが、エラー回数ではなく警告回数をインクリメントさせます。またオブジェクト・ファイルの作成に影響しません。

4.11 疑似命令のリファレンス

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

疑似命令	ページ	疑似命令	ページ
.algebraic	4-26	.ldouble	4-40
.align	4-27	.length	4-61
.asg	4-28	.list	4-62
.bes	4-82	.long/.ulong	4-64
.break	4-66	.loop	4-66
.bss	4-30	.macro	4-67
.byte/.ubyte	4-33	.mlib	4-68
.c_mode	4-35	.mlist	4-70
.char/.uchar	4-33	.mmregs.	4-71
.clink	4-34	.mmsg	4-42
.copy	4-36	.mnolist	4-70
.data	4-39	.newblock	4-74
.def	4-51	.nolist	4-62
.double	4-40	.option	4-76
.drlist	4-41	.page	4-78
.drnolist	4-41	.pstring	4-85
.else	4-56	.ref	4-51
.elseif	4-56	.sblock	4-79
.emsg	4-42	.sect	4-80
.end	4-44	.set	4-81
.endif	4-56	.short/.ushort	4-54
.endloop	4-66	.space	4-82
.endm	4-67	.sslist	4-83
.endstruct	4-86	.ssnolist	4-83
.endunion	4-92	.string	4-85
.equ	4-81	.struct	4-86
.eval	4-28	.tab	4-89
.even	4-27	.tag	4-86
.far_mode	4-45	.text	4-90
.fclist	4-46	.title	4-91
.fcnolist	4-46	.union	4-92
.field	4-47	.usect	4-95
.float	4-50	.version	4-99
.global	4-51	.width	4-61
.half/.uhalf	4-54	.wmsg	4-42
.if	4-56	.word/.uword	4-58
.include	4-36	.var	4-98
.int/.uint.	4-58	.xfloat	4-50
.label	4-60	.xlong	4-64

構文

```
.algebraic
```

説明

.algebraic 疑似命令は、ファイルに代数表記のアセンブリ・ソース・コードが含まれていることをアセンブラに伝えます。−mg オプションを使用しない場合、この疑似命令はファイルの第 1 行目になくてはなりません。

注： 代数表記のアセンブリ・コードとニーモニック・アセンブリ・コードの混在

代数表記のアセンブリ・コードとニーモニック・アセンブリ・コードを、同じソース・ファイル内に混在させることはできません。

.algebraic 疑似命令は、同じソース・ファイル内に代数表記の文とニーモニック文を混在させるための手段を提供しません。この疑似命令は、−mg アセンブラ・オプションを指定せずに代数表記のアセンブリを選択する方法を提供します。

構文

.align [*size*]
.even

説明

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

オペランドが 128 の場合、SPC は次のページ境界で位置合わせされます。*size* が指定されていない場合、ページ境界がデフォルト値になります。アセンブラは、次の x ワード境界までヌル値 (0) が含まれるワードをアセンブルします。

オペランドが 1　の場合、SPC はワード境界に位置合わせします。
2　の場合、SPC を倍長ワード/偶数境界に位置合わせします。
128　の場合、SPC をページ境界に位置合わせします。

.even 疑似命令は、SPC を倍長ワード境界/偶数境界で位置合わせします。この疑似命令は、オペランドが 2 の .align 疑似命令と同じです。

.align 疑似命令を使用すると、次の 2 つの効果があります。

- アセンブラは、現行のセクション内の境界で SPC を位置合わせします。
- アセンブラは、リンカに、セクションの位置合わせをさせるためのフラグを設定します。これにより、セクションがメモリにロードされたときに、個々の位置合わせに影響が及ばないようにします。

例

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

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

構文

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

説明

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

- アセンブラは、*character string* を置換シンボルに割り当てます。引用符の使用は任意です。引用符がない場合は、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、有効なシンボル名でなくてはなりません。置換シンボルは長さは 32 文字までで、先頭を文字で始める必要があります。シンボルの残りの文字には、英数字、下線 (_)、およびドル記号 ($) を組み合わせて使用できます。

.eval 疑似命令は、置換シンボル・テーブルに格納されている置換シンボルの算術演算を行います。この疑似命令は、式を計算し、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 のブロックのカウンタとして使用すると特に便利です。

- *well–defined expression* は、事前に定義された正しい値で構成されている 、つまり絶対的な結果を得られる英数字式のことをいいます。
- *substitution symbol* は、有効なシンボル名でなくてはなりません。置換シンボルは、長さは 32 文字までで、先頭を文字で始める必要があります。シンボルの残りの文字には、英数字、下線 (_)、およびドル記号 ($) を組み合わせて使用できます。

例　　.asg と .eval の使用例を以下に示します。

```
       1                        .sslist;show expanded sub. symbols
       2              *
       3              *         .asg/.eval example
       4              *
       5                        .asg    *+, INC
       6                        .asg    AR0, FP
       7
       8 000000 f000            ADD     #100, A
         000001 0064
       9 000002 6d90            MAR     *FP+
#                               MAR     *AR0+
      10
      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       000003 0001             .word   x
#                               .word   1
1                               .eval   x+1, x
#                               .eval   1+1, x
1       000004 0002             .word   x
#                               .word   2
1                               .eval   x+1, x
#                               .eval   2+1, x
1       000005 0003             .word   x
#                               .word   3
1                               .eval   x+1, x
#                               .eval   3+1, x
1       000006 0004             .word   x
#                               .word   4
1                               .eval   x+1, x
#                               .eval   4+1, x
1       000007 0005             .word   x
#                               .word   5
```

構文

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

説明

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

- *symbol* (シンボル名) は、必須パラメータです。このパラメータは、疑似命令によって確保される最初の位置を指すラベルを定義します。このシンボル名は、空間を確保する対象となる変数と対応していなければなりません。
- *size in words* は、必須パラメータです。絶対式でなければなりません。アセンブラは、指定されたワード数を .bss セクションに割り当てます。デフォルトのサイズはありません。
- *blocking flag* (ブロック化フラグ) は、任意のパラメータです。このパラメータに 0 以外の値を指定すると、アセンブラは指定されたサイズの空間を連続した空間に割り当てます。つまり、空間のサイズが 1 ページを超えていない限り、割り当てられた空間がページ境界にまたがることはありません。1 ページを超える場合は、オブジェクトはページ境界から開始されます。
- *alignment flag* (位置合わせフラグ) は、任意のパラメータです。このフラグを指定すると、アセンブラは指定された空間を倍長ワード境界に割り当てます。

注： 位置合わせフラグのみを指定する場合

ブロック化フラグを指定しないで位置合わせフラグのみを指定するには、位置合わせフラグの前にコンマを 2 つ挿入するか、またはブロック化フラグに 0 を指定することができます。

アセンブラは次の 2 つの規則に従って、.bss セクションに空間を割り当てます。

規則 1 (図 4−5 に示すように) メモリにホール (穴) が残っていると、.bss 疑似命令は必ずそのホールを埋めようとします。アセンブラは、.bss 疑似命令をアセンブルするときに、前の .bss 疑似命令により残されたホールのリストを検索して、ホールの 1 つに現行のブロックを割り当てようとします (これは、連続割り当てオプションが指定されているかどうかに関係なく実行される標準の手順です)。

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

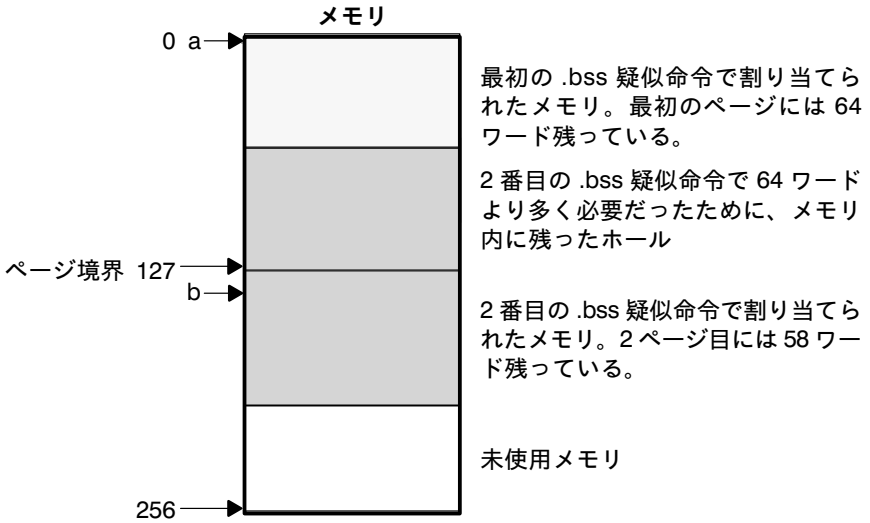
- ブロック化を指定しないと、メモリは現行の SPC の位置に割り当てられます。
- ブロック化を指定すると、アセンブラは、現行の SPC からページ境界までに十分な空間があるかどうかを調べます。ここに十分な空間がなければ、アセンブラはそこに別のホールを作成し、次のページの始まりに空間を割り当てます。

ブロック化オプションを使用すると、.bss セクションに最大 128 ワードを確保して、それがメモリの 1 ページに収まるようにすることができます (もちろん、一度に 128 ワード以上確保できますが、その場合は 1 ページには収まりません)。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 ページの境界内部に入っていなければなりません。しかし、最初のブロックを割り当てた後では、2 番目のブロックは現行のページ内には収まりません。図 4−5 に示すように、2 番目のブロックは次ページに割り当てられます。

図 4−5.　ページ内への .bss ブロックの割り当て方法



初期化されたセクション用のセクション疑似命令 (.text、.data、および .sect) は、現行のセクションを終了させ、別のセクションへのアセンブルを始めるように指示します。ただし、.bss 疑似命令は、現行のセクションに影響を及ぼしません。アセンブラは、.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コードのアセンブルを再開します。COFF セクションの詳細は、第 2 章「共通オブジェクト ・ファイル ・フォーマットの概要」を参照してください。

例

この例では、.bss 疑似命令を使用して、2 つの変数 TEMP と ARRAY のための空間を割り当てます。シンボル TEMP は、(.bss の SPC = 0 にある) 初期化されない 4 ワードの空間を指します。シンボル ARRAY は、(.bss の SPC = 04h にある) 初期化されない 100 ワードの空間を指します。この空間は、1 ページ内に連続して割り当てる必要があります。.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、外部シンボルとして宣言できることに注意してください。

```
 1                ****************************************
 2                **  Assemble into the .text section.     **
 3                ****************************************
 4 000000                 .text
 5 000000 e800            LD      #0, A
 6                ****************************************
 7                **  Allocate 4 words in .bss for TEMP.   **
 8                ****************************************
 9 000000         Var_1:  .bss    TEMP, 4
10
11                ****************************************
12                **           Still in .text             **
13                ****************************************
14 000001 f000           ADD     #56h, A
   000002 0056
15 000003 f066           MPY     #73h, A
   000004 0073
16
17                ****************************************
18                **  Allocate 100 words in .bss for the   **
19                **  symbol named ARRAY; this part of     **
20                **  .bss must fit on a single page.      **
21                ****************************************
22 0000004               .bss    ARRAY, 100, 1
23
24                ****************************************
25                **  Assemble more code into .text.       **
26                ****************************************
27 000005 8000-          STL     A, Var_1
28
29                ****************************************
30                **  Declare external .bss symbols.       **
31                ****************************************
32                       .global ARRAY, TEMP
33                       .end
```

構文

.byte *value*$_1$ [**,** ... **,** *value*$_n$]
.ubyte *value*$_1$ [**,** ... **,** *value*$_n$]
.char *value*$_1$ [**,** ... **,** *value*$_n$]
.uchar *value*$_1$ [**,** ... **,** *value*$_n$]

説明

.byte、**.ubyte**、**.char** および **.uchar** 疑似命令は、1 バイト以上を現行のセクションの連続したワードに挿入します。バイトは、それぞれ単独で 1 つのワードに挿入されます。ワードの MSB 8 ビットにはゼロ (0) が埋め込まれます。*value* には、次のものを指定できます。

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

値は、パックも符号拡張もされません。各バイトは、完全な 16 ビット・ワードの LSB 8 ビットを占めます。アセンブラは、8 ビットよりも大きい値は切り捨てます。value パラメータは 100 個まで使用できますが、行全体の長さは 200 文字以内に収めなくてはなりません。

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

.struct/.endstruct のシーケンスの中で上記の疑似命令を使用する場合、疑似命令はメンバのサイズのみを定義し、メモリの初期化は行いません。注意してください。.struct/.endstruct の詳細は、4-21 ページの 4.9 節「アセンブル時シンボル疑似命令」を参照してください。

例

この例では、8 ビット値 (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 100h で、これは初期化されたワードの先頭位置を表します。

```
1 0000                  .space   100h * 16
2 000100 000a  STRX     .byte    10, -1, "abc", 'a'
  000101 00ff
  000102 0061
  000103 0062
  000104 0063
  000105 0061
```

構文

.clink [*"section name"*]

説明

.clink 疑似命令は、*section name* (セクション名) で指定されるセクションのタイプ・フィールドに STYP_CLINK フラグを設定して、セクションの条件付きリンクを設定します。.clink 疑似命令は、初期化されたセクション、または初期化されていないセクションのいずれにも適用できます。

セクション名を指定しないで .clink を使用する場合、.clink は現在の初期化されたセクションに適用されます。.clink を初期化されないセクションに適用する場合は、セクション名を指定する必要があります。セクション名は、200 文字までが有効です。二重引用符で囲んでください。セクション名には、*section name:subsection name* の書式でサブセクション名を指定することができます。

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

C プログラムのエントリ・ポイントが定義されているセクションには、条件付きでリンクされるセクションとしてマークを付けることはできません。

例

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

```
1 000000                .sect ”Vars”
2               ; Vars section is conditionally linked
3                       .clink
4
5 000000 001A   X:      .word 01Ah
6 000001 001A   Y:      .word 01Ah
7 000002 001A   Z:      .word 01Ah
8 000000                .sect ”Counts”
9               ; Counts section is conditionally linked
10                      .clink
11
12 000000 001A  Xcount: .word 01Ah
13 000001 001A  Ycount: .word 01Ah
14 000002 001A  Zcount: .word 01Ah
15              ; By default, .text is unconditionally linked
16 000000               .text
17              ; Reference to symbol X cause the Vars section
18              ; to be linked into the COFF output
19 000000 E800          LD #0, A
20 000001 8000+         STL A, X
```

構文

```
.c_mode
```

説明

.c_mode 疑似命令は、C コンパイラにより作成されたアセンブリ・ファイルのヘッダとして生成されます。

この疑似命令は、主に −mf アセンブラ・オプションまたは .far_mode 疑似命令とともに使用することで、拡張アドレッシングを採用しているプログラムのリンクを容易にします。

プログラムは拡張アドレッシングを使用するが、アセンブリ・コードが C コンパイラの生成したものでない場合は、アセンブリ・ファイルに .c_mode 疑似命令を追加してください。

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 の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、.copy/.include 疑似命令を検出すると、

1) 現行のソース・ファイルの文のアセンブルを中止します。

2) コピー・ファイルまたはインクルード・ファイルの文をアセンブルします。

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

filename は、ソース・ファイルの名前を指定する必須パラメータです。filename は、二重引用符で囲むこともできます。またその指定は、オペレーティング・システムの規則に従わなければなりません。完全なパス名 (c:\dsp\file1.asm など) を指定することもできます。完全なパス名を指定しない場合、アセンブラは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細は、3-8 ページの 3.4 節「アセンブラ入力のための代替ファイルと代替ディレクトリの命名方法」を参照してください。

.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・システムは別の制限を設定している場合があります。アセンブラは、コピーされたファイルの行番号の前にコピーのレベルを識別する文字コードを置きます。たとえば、A は最初にコピーされたファイルを示し、B は 2 番目にコピーされたファイルを示します。

例 1

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

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

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

copy.asm (ソース・ファイル)	**byte.asm (最初のコピー・ファイル)**	**word.asm (2 番目のコピー・ファイル)**
.space 29 **.copy ”byte.asm”** **Back in original file .pstring ”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 000000          .space 29
      2                 .copy ”byte.asm”
A     1               ** In byte.asm
A     2 000002 0020     .byte 32,1+ ’A’
        000003 0042
A     3                 .copy ”word.asm”
B     1               * In word.asm
B     2 000004 ABCD     .word 0ABCDh, 56q
        000005 002E
A     4               ** Back in byte.asm
A     5 000006 006A     .byte 67h + 3q
      3
      4               ** Back in original file
      5 000007 646F     .pstring ”done”
        000008 6E65
```

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

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

リスト・ファイル:

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

構文

```
.data
```

説明

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

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用しない限り、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 セクションの詳細は、第 2 章「共通オブジェクト ・ファイル ・フォーマットの概要」を参照してください。

例

この例では、コードは .data セクションと .text セクションにアセンブルされます。

```
 1                ****************************************
 2                **  Reserve space in .data.             **
 3                ****************************************
 4 000000                 .data
 5 000000                 .space          0CCh
 6
 7                ****************************************
 8                **  Assemble into .text.                **
 9                ****************************************
10 000000                 .text
11        0000   INDEX    .set            0
12 000000 e800             LD              #INDEX, A
13
14                ****************************************
15                **  Assemble into .data.                **
16                ****************************************
17 00000c         Table:  .data
18 00000d ffff           .word   -1   ; Assemble 16-bit
19                                    ; constant into .data.
20 00000e 00ff           .byte   0FFh ; Assemble 8-bit
21                                    ; constant into .data.
22
23                ****************************************
24                **  Assemble into .text.                **
25                ****************************************
26 000001                 .text
27 000001 000c"            ADD     Table, A
28
29                ****************************************
30                **  Resume assembling into the .data    **
31                **  section at address 0Fh.             **
32                ****************************************
33 00000f                 .data
```

構文

.double *value* [**, ... ,** *value*$_n$]
.ldouble *value* [**, ... ,** *value*$_n$]

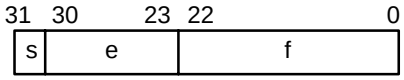
説明

.double 疑似命令と **.ldouble** 疑似命令は、1 つ以上の浮動小数点値の IEEE 単精度浮動小数点表現を現行のセクションに挿入します。各 *value* は浮動小数点定数、または浮動小数点定数と等価なシンボルでなくてはなりません。各定数は、IEEE 単精度 32 ビット形式の浮動小数点値に変換されます。浮動小数点定数は、ワード境界に位置合わせされます。

値は、次の 3 つのフィールドで構成されます。

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

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



.double または .ldouble 疑似命令を .struct/.endstruct のシーケンスで使用すると、その疑似命令はメンバのサイズを定義し、メモリの初期化は行いません。
.struct/.endstruct の詳細は、4-21 ページの 4.9 節「アセンブル時シンボル疑似命令」を参照してください。

例

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

```
  1 000000 E904        .double  -1.0e25
    000001 5951
  2 000002 43E4        .ldouble  456.0
    000003 0000
```

構文

```
.drlist
.drnolist
```

説明

この 2 つの疑似命令を使用すると、アセンブラ疑似命令のリスト・ファイルへの出力を制御できます。

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

.drnolist 疑似命令は、以下の疑似命令のリスト・ファイルへの出力を抑止します。

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

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

例

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

ソース・ファイル:

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

     .drnolist

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

リスト・ファイル:

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

構文

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

説明

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

.emsg 疑似命令は、アセンブラと同様にエラー・メッセージを標準出力デバイスに送ります。つまり、エラー回数をインクリメントし、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す。

.mmsg 疑似命令は、.emsg および .wmsg 疑似命令と同様に、アセンブル時メッセージを標準出力デバイスに送信します。ただし、エラー回数や警告回数を設定せず、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、.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、エラー回数でなく警告回数をインクリメントさせます。また、アセンブラによるオブジェクト・ファイルの作成を禁止しません。

例

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

ソース・ファイル:

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

         MSG_EX PARAM

         MSG_EX
```

リスト・ファイル:

```
       1                        .global PARAM
       2              MSG_EX    .macro  parm1
       3                        .if     $symlen(parm1) = 0
       4                       .emsg   ”ERROR -- MISSING PARAMETER”
       5                        .else
       6                       add     parm1, A
       7                        .endif
       8                        .endm
       9
      10 000000                MSG_EX PARAM
1                               .if     $symlen(parm1) = 0
1                              .emsg   ”ERROR -- MISSING PARAMETER“
1                               .else
1        000000 0000!           add     PARAM, A
1                               .endif
      11
      12 000001                MSG_EX
1                               .if     $symlen(parm1) = 0
1                              .emsg   ”ERROR -- MISSING PARAMETER”
  ***** USER ERROR ***** - : ERROR -- MISSING PARAMETER
1                               .else
1                               add     parm1, A
1                               .endif
 1 Error, No Warnings
```

さらに、アセンブラによって以下のメッセージが標準出力に送られます。

```
TMS320C54x COFF Assembler       Version x.xx
Copyright (c) 2001    Texas Instruments Incorporated
 PASS 1
 PASS 2
”emsg.asm”, ERROR! at line 12: [***** USER ERROR ***** -] ERROR -- MISSING
  PARAMETER
                  .emsg ”ERROR -- MISSING PARAMETER”

 1 Error,  No Warnings
Errors in source - Assembler Aborted
```

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をすべて無視します。

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

例

この例は、.end 疑似命令を使用したアセンブルの終了方法を示しています。アセンブラは、.end 疑似命令以降のソース文をすべて無視します。

ソース・ファイル:

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

リスト・ファイル:

```
       1 000000       START:  .space  300
       2        000F  TEMP    .set    15
       3 000000               .bss    LOC1, 48h
       4 000013 F485          ABS     A
       5 000014 F000          ADD     #TEMP, A
         000015 000F
       6 000016 8000-         STL     A, LOC1
       7                      .end
```

構文

```
.far_mode
```

説明

.far_mode 疑似命令は、アセンブリ・ファイルが拡張アドレッシングを使用すること、つまりコールと分岐が通常の 16 ビット範囲を超えて拡張することをアセンブラに伝えます。この疑似命令は、−mf アセンブラ・オプションと同じ働きをします。

プログラムが拡張アドレッシングを使用し、かつアセンブリ・コードが C コンパイラの生成したものでない場合は、アセンブリ・ファイルに .c_mode 疑似命令を追加してください。

構文

```
.fclist
.fcnolist
```

説明

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

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

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

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

例

この例は、条件付きブロックをリストに出力する場合と、出力しない場合のアセンブリ・ソース・ファイルとリスト・ファイルを示しています。

ソース・ファイル:

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

        .fcnolist
        .if   AAA
        ADD   #1024, A
        .else
        ADD   #1024*10, A
        .endif
```

リスト・ファイル:

```
        1        0001  AAA     .set  1
        2        0000  BBB     .set  0
        3                      .fclist
        4                      .if   AAA
        5 000000 F000          ADD   #1024, A
          000001 0400
        6                      .else
        7                      ADD   #1024*10, A
        8                      .endif
        9
       10                      .fcnolist
       11
       13 000002 F000          ADD   #1024, A
          000003 0400
```

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で構成されてい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、計算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 in bits* の値は 16 でなければなりません。
- *size in bits* (サイズ) は任意のパラメータです。ここにはフィールドを構成するビット数 (1 ～ 32) が入ります。サイズを指定しないと、アセンブラはサイズが 16 ビットであるものとみなします。16 以上の値を指定すると、フィールドはワード境界から始まります。指定したサイズに入り切らない値を指定すると、アセンブラはその値を切り捨てて、エラー・メッセージを発行します。たとえば、.field 3,1 と指定した場合、アセンブラは 3 を 1 に切り捨てます。また、アセンブラは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、現行のワードから始まる指定されたビット数に値をパックします。さらにフィールドが追加されるに従って、フィールドは、ワードの最上位から最下位に向かって順番にパックされます。アセンブラは、現行のワードに入り切らないサイズのフィールドを検出すると、そのワードを書き出して、SPC をインクリメントし、次のワードへのフィールドのパックを開始します。オペランドに 1 を指定して .align 疑似命令を使用すると、次の .field 疑似命令は、新しいワードへパックされるようになります。

ラベルを使用する場合、ラベルは指定されたフィールドを含むワードを指します。

.struct/.endstruct シーケンスで .field を使用すると、.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 の詳細は、4-21 ページの 4.9 節「アセンブル時シンボル疑似命令」を参照してください。

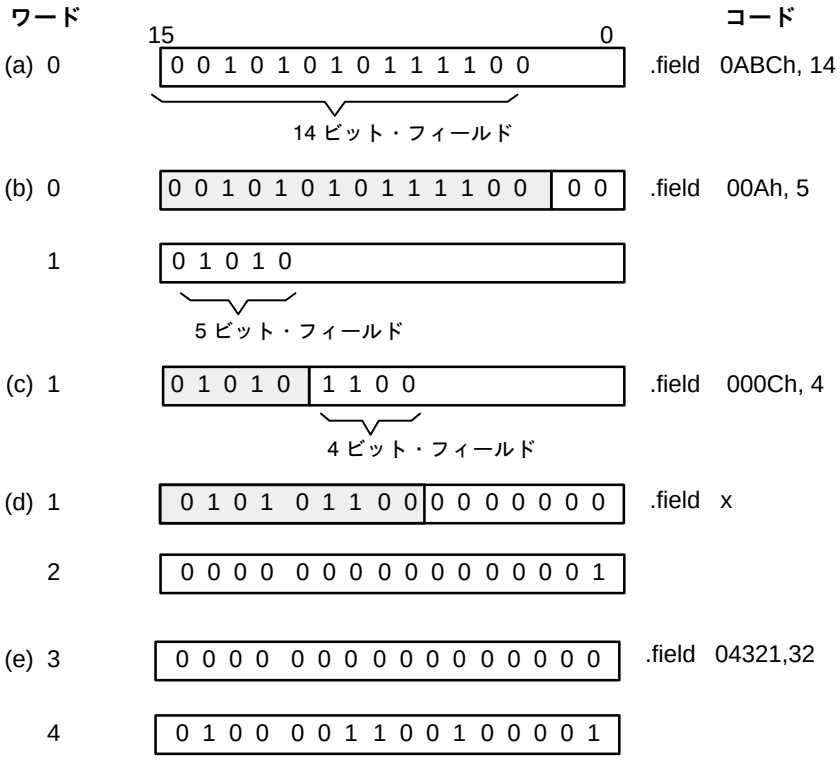
例

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

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

図 4−6 は、この例の疑似命令がどのような効果をメモリに与えるかを示しています。

図 4−6.　.field 疑似命令



構文

.float *value*$_1$ [**,** ... **,** *value*$_n$]
.xfloat *value*$_1$ [**,** ... **,** *value*$_n$]

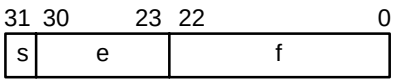
説明

.float 疑似命令と **.xfloat** 疑似命令は、1 つ以上の浮動小数点定数の浮動小数点表現を現行のセクションに挿入します。各 *value* は浮動小数点定数、または浮動小数点定数と等価なシンボルでなくてはなりません。各定数は、IEEE 単精度 (32 ビット) 形式の浮動小数点値に変換されます。浮動小数点定数は、.xfloat 疑似命令が使用されない限りは、倍長ワード境界で位置合わせされます。.xfloat 疑似命令は、.float 疑似命令と同じ働きをしますが、結果を倍長ワード境界に位置合わせしません。

32 ビット値は、次の 3 つのフィールドで構成されます。

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

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



.struct/.endstruct シーケンスで .float を使用すると、.float はメンバのサイズを定義し、メモリの初期化は行いません。.struct/.endstruct の詳細は、4-21 ページの 4.9 節「アセンブル時シンボル疑似命令」を参照してください。

例

.float 疑似命令の例を以下に示します。

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

構文

.global *symbol*$_1$ [**,** *...* **,** *symbol*$_n$]
.def *symbol*$_1$ [**,** *...* **,** *symbol*$_n$]
.ref *symbol*$_1$ [**,** *...* **,** *symbol*$_n$]

説明

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

.def 疑似命令は、現行のモジュールで定義され、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用さ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・シンボルは、他のシンボルと同じ方法で定義されます。つまり、ラベルとして使用されるか、.set、.bss、.usect の疑似命令のいずれかによって定義されます。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用するときだけ、シンボル・テーブル・エントリを作成します。

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

例

この例では 4 つのファイルを使用します。

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

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

file1.lst:

```
 1                ; Global symbol defined in this file
 2                        .global INIT
 3                ; Global symbols defined in file2.lst
 4                        .global X, Y, Z
 5 000000         INIT:
 6 000000 F000            ADD     #56h, A
   000001 0056
 7 000002 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 000000 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 000000         INIT:
 6 000000 F000            ADD     #56h, A
   000001 0056
 7 000002 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 000000 0000!              .word   INIT
 9                  ;        .
10                  ;        .
11                  ;        .
12                           .end
```

構文

.half *value*$_1$ [, ... , *value*$_n$]
.uhalf *value*$_1$ [, ... , *value*$_n$]
.short *value*$_1$ [, ... , *value*$_n$]
.ushort *value*$_1$ [, ... , *value*$_n$]

説明

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

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

value には、絶対式または再配置可能式を指定できます。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、リンカは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用してメモリを初期化できます。

アセンブラは、16 ビットを超える値は切り捨てます。単一行に収まるのであれば、value をいくつでも指定できますが、行全体の長さは 200 文字以内でなくてはなりません。ラベルを使用する場合、ラベルは最初の初期化されたワードを指します。

.half、.uhalf、.short、または .ushort 疑似命令を .struct/.endstruct シーケンスで使用した場合、その疑似命令はメンバのサイズを定義し、メモリの初期化は行いません。.struct/.endstruct の詳細は、4-21 ページの 4.9 節「アセンブル時シンボル疑似命令」を参照してください。

例

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

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

構文

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

説明

これらの 4 つの疑似命令を使用して条件付きアセンブリを行うことができます。

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

- ❏ 式の計算結果が真 (ゼロ以外) の場合、アセンブラは式に続くコードをアセンブルします (.elseif、.else、.endif のいずれかが検出されるまで)。
- ❏ 式の計算結果が偽 (0) の場合、アセンブラは .elseif か .else があればそれに続くコードをアセンブルし、 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 (ゼロ以外) の場合に、アセンブルするコード・ブロックを識別します。.elseif 式が偽のとき、アセンブラは、次の (.elseif があれば) .elseif まで、(.else があれば) .elseif まで、または (.elseif も .else もなければ) .endif へ処理を進めます。条件付きブロックでは .elseif 疑似命令の使用は任意であり、複数の .elseif を使用することもできます。式が偽であり、かつ .elseif 文がない場合、アセンブラは、(.else があれば) .else または .endif の後に続くコードへと処理を進めます。

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

.endif 疑似命令は、条件付きブロックを終了します。

.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。

関係演算子の詳細は、3-27 ページの 3.9.4 項「条件式」を参照してください。

例　　条件付きアセンブリの例を以下に示します。

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

構文

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

説明

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

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

value には、絶対式または再配置可能式を指定できます。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用してメモリを初期化できます。

単一行 (200 文字) に収まるだけの数の value を使用できます。ラベルを使用すると、ラベルは初期化される最初のワードを指します。

これらの疑似命令を .struct/.endstruct シーケンスで使用すると、その疑似命令はメンバのサイズを定義しますが、メモリの初期化は行いません。.struct/.endstruct の詳細は、4-21 ページの 4.9 節「アセンブル時シンボル疑似命令」を参照してください。

例 1　この例では、.int 疑似命令を使用してワードを初期化します。

```
 1 000000            .space 73h
 2 000000            .bss   PAGE, 128
 3 000080            .bss   SYMPTR, 3
 4 000008 E856 INST:  LD     #056h, A
 5 000009 000A       .int   10, SYMPTR, -1, 35 + 'a', INST
   00000a 0080-
   00000b FFFF
   00000c 0084
   00000d 0008'
```

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

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

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。アセンブラで作成されるほとんどのセクションは、再配置可能なアドレスをもっています。アセンブラは、各セクションがゼロで開始されているとしてセクションをアセンブルし、リンカは、それらをロードおよび実行されるアドレスに再配置します。

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

このようなセクションには、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。このセクションで定義されるすべてのラベルは実行アドレスを参照するように再配置され、コード実行時にこのセクションに対する参照 (ブランチなど) は正しくなります。

label 疑似命令は、ロード時のアドレスを参照する特別なラベルを作成します。この機能は主として、セクションの再配置を行うコードに対して、セクションがロードされた位置を指定するときに便利です。

例

この例は、ロード・アドレス・ラベルの使用方法を示しています。

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

リンカで実行アドレスおよびロード・アドレスを割り当てる方法の詳細は、7-44 ページの 7.9 節「セクションの実行 (ランタイム) アドレスの指定方法」を参照してください。

構文

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

説明

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

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

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

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

ここで言う幅とは、リスト・ファイルの完全な 1 行を指します。行カウンタの値、SPC 値、およびオブジェクト・コードは 1 行の幅の一部となります。ページ幅を超えるコメント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 の疑似命令のリストを作成しません。

例

この例では、ページ長とページ幅を変える方法を示しています。

```
*****************************************
**        Page length = 65 lines.      **
**        Page width  = 85 characters. **
*****************************************
          .length    65
          .width     85

*****************************************
**        Page length = 55 lines.      **
**        Page width  = 100 characters.**
*****************************************
          .length    55
          .width     100
```

構文

```
.list
.nolist
```

説明

2 つの疑似命令を使用して、ソース・リストの出力を制御できます。

- **.list** 疑似命令は、ソース・リストの出力を許可します。
- **.nolist** 疑似命令は、.list 疑似命令を検出するまでソース・リストの出力を抑止します。.nolist 疑似命令を使用すると、アセンブル時間とソース・リストのサイズを削減できます。また、この疑似命令をマクロ定義で使用すると、リストへのマクロ展開の出力を抑止できます。

アセンブラは、.list や .nolist 疑似命令、または .nolist 疑似命令より後に現れるソース文を出力しません。ただし、行カウンタはインクリメントしていきます。.list および .nolist の疑似命令はネストできます。リスト出力を回復するには、各 .nolist に対応する .list が必要です。

デフォルトでは、ソース・リストはリスト・ファイルに出力されます。アセンブラは .list 疑似命令が指定されているものとして動作します。ただし、アセンブラを起動する際にリスト・ファイルの生成を指定しないと、アセンブラは .list 疑似命令を無視します。

例

この例では、.copy 疑似命令を使用して他のファイルからのソース文を挿入します。アセンブラは、最初にこの疑似命令を検出すると、コピーされたソース行をリスト・ファイルに出力します。2 回目に .copy を検出したときには、.nolist 疑似命令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・ファイルに出力しません。.nolist 疑似命令、2 番目の .copy 疑似命令、および .list 疑似命令は、リスト・ファイルには出力されません。また、ソース文がリストに出力されない場合でも、行カウンタはインクリメントされることに注意してください。

ソース・ファイル:

```
   .copy  ″copy2.asm″
* Back in original file
   NOP
   .nolist
   .copy  ″copy2.asm″
   .list
* Back in original file
   .string   ″Done″
```

リスト・ファイル:

```
       1                        .copy   ″copy2.asm″
 A     1                    * In copy2.asm (copy file)
 A     2 000000 0020            .word 32, 1 + ′A′
         000001 0042
       2                    * Back in original file
       3 000002 F495            NOP
       7                    * Back in original file
       8 000005 0044            .string ″Done″
         000006 006F
         000007 006E
         000008 0065
```

構文

.long *value*$_1$ [, ... , *value*$_n$]
.ulong *value*$_1$ [, ... , *value*$_n$]
.xlong *value*$_1$ [, ... , *value*$_n$]

説明

.long、**.ulong**、および **.xlong** 疑似命令は、1 つ以上の 32 ビット値を現行のセクションの連続したワードに格納します。最上位のワードを最初に格納します。.long 疑似命令と .ulong 疑似命令は、結果を倍長ワード境界で位置合わせしますが、.xlong 疑似命令はこのような位置合わせは行いません。value には、次のいずれかを指定できます。

- アセンブラが計算して、32 ビットの符号付きまたは符号なしの数値として扱う式
- 二重引用符に囲まれた文字列。文字列内の各文字は、別々の値を表します。

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、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用してメモリを初期化できます。

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。ラベルを使用すると、ラベルは初期化される最初のワードを指します。

これらの疑似命令を .struct/.endstruct シーケンスで使用した場合、その疑似命令はメンバのサイズを定義し、メモリの初期化は行いません。.struct/.endstruct の詳細は、4-21 ページの 4.9 節「アセンブル時シンボル疑似命令」を参照してください。

例　この例は、.long 疑似命令と .xlong 疑似命令が倍長ワードをどのように初期化するかを示しています。

```
1 000000 0000  DAT1: .long  0ABCDh, 'A' + 100h, 'g', 'o'
  000001 ABCD
  000002 0000
  000003 0141
  000004 0000
  000005 0067
  000006 0000
  000007 006F
2 000008 0000'       .xlong  DAT1, 0AABBCCDDh
  000009 0000
  00000a AABB
  00000b CCDD
3 00000c       DAT2:
```

構文

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

説明

これらの 3 つの疑似命令を使用すると、コードのブロックを反復してアセンブルすることができます。

.loop 疑似命令は、反復使用が可能なコード・ブロックを開始します。任意選択の式には、ループ・カウント (ループに含まれるコードがアセンブリを繰り返す回数) を指定します。式の指定がない場合、アセンブラが真 (ゼロ以外) の式をもった .break 疑似命令か、式をもたない .break 疑似命令を最初に検出しない限り、ループ・カウントはデフォルトの 1024 となります。

.break 疑似命令とその式は任意選択です。式が偽 (0) のときは、ループは継続します。式が真 (ゼロ以外)、または省略されたときは、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、反復使用が可能なコード・ブロックを終了します。この疑似命令は、.break 疑似命令が真 (ゼロ以外) 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

この例は、.loop、.break、および .endloop 疑似命令を .eval 疑似命令とともに使用する方法を示しています。

```
       1                    .eval       0,x
       2               COEF **.loop**
       3                    .word       x*100
       4                    .eval       x+1, x
       5                    **.break      x = 6**
       6                    **.endloop**
1        000000 0000        .word       0*100
1                           .eval       0+1, x
1                           **.break      1 = 6**
1        000001 0064        .word       1*100
1                           .eval       1+1, x
1                           **.break      2 = 6**
1        000002 00C8        .word       2*100
1                           .eval       2+1, x
1                           **.break      3 = 6**
1        000003 012C        .word       3*100
1                           .eval       3+1, x
1                           **.break      4 = 6**
1        000004 0190        .word       4*100
1                           .eval       4+1, x
1                           **.break      5 = 6**
1        000005 01F4        .word       5*100
1                           .eval       5+1, x
1                           **.break      6 = 6**
```

構文

macname **.macro** [*parameter*$_1$] [, ... *parameter*$_n$]
model statements or macro directives
.endm

説明

マクロの定義には、**.macro** 疑似命令を使用します。

マクロはプログラム内のどこでも定義できますが、使用する前に必ず定義する必要があります。マクロは、ソース・ファイルの先頭、.include/.copy ファイル、またはマクロ・ライブラリで定義することができます。

macname	マクロに名前を付けます。名前は、必ずソース文のラベル・フィールドに入れなければなりません。
.macro	ソース文がマクロ定義の第 1 行目であることを特定します。.macro は、必ず命令コード・フィールドに置かなければなりません。
[*parameters*]	.macro 疑似命令のオペランドとして使用される、任意選択の置換シンボルです。
model statements	マクロが呼び出されるたびに実行される命令、またはアセンブラ疑似命令です。
macro directives	マクロ展開の制御に使用されます。
.endm	マクロ定義を終了します。

マクロについては、第 5 章「マクロ言語」で詳しく説明します。

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 (c:\dsp\macs.lib など) を指定することもできます。完全なパス名を指定しない場合、アセンブラは以下の位置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の詳細は、3-8 ページの 3.4 節「アセンブラ入力のための代替ファイルと代替ディレクトリの命名方法」を参照してください。

.mlib 疑似命令を検出すると、アセンブラはライブラリをオープンし、その内容のテーブルを作成します。アセンブラは、個々のライブラリ・メンバ名をライブラリ・エントリとして命令コード・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だけが、一度だけ抽出されます。

例

この例では、incr および decr の 2 つのマクロを定義するマクロ・ライブラリを作成します。ファイル incr.asm には incr の定義が、ファイル decr.asm には decr の定義がそれぞれ含まれます。

incr.asm	decr.asm

* Macro for incrementing
incr .macro
ADD #1,A
ADD #1,B
.endm
	* Macro for zero accumulators
	decr .macro
	SUB A,A
	SUB B,B
	.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。

```
ar500 –a mac incr.asm decr.asm
```

これで、.mlib 疑似命令を使用してマクロ・ライブラリを参照し、マクロ incr と decr を定義できるようになります。

```
       1                .mlib    ”mac.lib”
       2 000000         decr             ; Macro call
1        000000 F420       SUB A,A
1        000001 F720       SUB B,B
       3 000002         incr             ; Macro call
1        000002 F000       ADD #1,A
         000003 0001
1        000004 F300       ADD #1,B
         000005 0001
```

構文

```
.mlist
.mnolist
```

説明

この 2 つの疑似命令を使用してリスト・ファイルにあるマクロ展開および反復使用可能なブロックの展開のリスト作成を制御できます。

- **.mlist** 疑似命令を使用すると、マクロと .loop/.endloop ブロックの展開をリスト・ファイルに出力します。
- **.mnolist** 疑似命令は、マクロと .loop/.endloop ブロックの展開のリスト・ファイルへの出力を抑止します。

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

例

この例では、*STR_3* という名前のマクロを定義します。マクロが 2 回目に呼び出されるときには、.mnolist 疑似命令がアセンブルされているので、マクロ展開はリストに出力されません。3 回目に呼び出されるときには、.mlist 疑似命令がアセンブルされているので、マクロ展開はリストに出力されます。

```
       1              STR_3  .macro   P1, P2, P3
       2                     .string ":p1:", ":p2:", ":p3:"
       3                     .endm
       4
       5 000000              STR_3 "as", "I", "am"
1        000000 003A         .string ":p1:", ":p2:", ":p3:"
         000001 0070
         000002 0031
         000003 003A
         000004 003A
         000005 0070
         000006 0032
         000007 003A
         000008 003A
         000009 0070
         00000a 0033
         00000b 003A
       6                     .mnolist
       7 00000c              STR_3 "as", "I", "am"
       8                     .mlist
       9 000018              STR_3 "as", "I", "am"
1        000018 003A         .string ":p1:", ":p2:", ":p3:"
         000019 0070
         00001a 0031
         00001b 003A
         00001c 003A
         00001d 0070
         00001e 0032
         00001f 003A
         000020 003A
         000021 0070
         000022 0033
         000023 003A
```

構文

```
.mmregs
```

説明

.mmregs 疑似命令は、'C54x 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、AL .set 8、AH .set 9 などを実行した場合と同じです。定義されたシンボルはローカルの絶対シンボルです。.mmregs 疑似命令を使用すると、これらのシンボルを定義する必要がなくなります。

表 4−2.　メモリ・マップド・レジスタ

名前	16 進アドレス	説明
IMR	0000	割り込みマスク・レジスタ
IFR	0001	割り込みフラグ・レジスタ
−	2−5	予約済み
ST0	0006	ステータス・レジスタ 0
ST1	0007	ステータス・レジスタ 1
AL	0008	アキュムレータ A 下位ワード (A [15:0])
AH	0009	アキュムレータ A 上位ワード (A [31:16])
AG	000A	アキュムレータ A ガード・ビット (A [39:32])
BL	000B	アキュムレータ B 下位ワード (B [15:0])
BH	000C	アキュムレータ B 上位ワード (B [31:16])
BG	000D	アキュムレータ B ガード・ビット (B [39:32])
T	000E	T レジスタ
TRN	000F	トランジション・レジスタ
BK	0019	サーキュラ・バッファ・サイズ・レジスタ
BRC	001A	ブロック・リピート・カウンタ
RSA	001B	ブロック・リピート開始アドレス
REA	001C	ブロック・リピート終了アドレス
PMST	001D	プロセッサ・モード・ステータス・レジスタ
DRR0	0020	シリアル・ポート・データ受信レジスタ 0
BDRR	0020	BSP データ受信レジスタ

注:　テーブル内のアドレス値が重複している場合、それらは、異なる 'C54x デバイスにインプリメントされているレジスタの異なる名前をサポートします。

表 4−2.　メモリ・マップド・レジスタ (続き)

名前	16 進アドレス	説明
BDDR0	0020	BSP データ受信レジスタ 0
DRR	0020	シリアル・ポート・データ受信レジスタ
DXR0	0021	シリアル・ポート・データ送信レジスタ 0
BDXR	0021	BSP データ送信レジスタ
BDXR0	0021	BSP データ送信レジスタ 0
DXR	0021	シリアル・ポート・データ送信レジスタ
SPC0	0022	シリアル・ポート制御レジスタ 0
BSPC	0022	BSP 制御レジスタ
SPC	0022	シリアル・ポート制御レジスタ
BSPCE	0023	BSP 制御拡張レジスタ
BSPCE0	0023	BSP 制御拡張レジスタ 0
SPCE	0023	シリアル・ポート制御拡張レジスタ
TIM	0024	タイマ・カウンタ・レジスタ
PRD	0025	タイマ・ピリオド・レジスタ
TCR	0026	タイマ制御レジスタ
PDWSR	0028	プログラム / データ・ソフトウェア・ウェイトステート・レジスタ
SWWSR	0028	プログラム / データ・ソフトウェア・ウェイトステート・レジスタ
IOWSR	0029	バンク・スイッチ制御レジスタ
BSCR	0029	バンク切り替え制御レジスタ
HPIC	002C	HPI 制御レジスタ
DRR1	0030	シリアル・ポート・データ受信レジスタ 1
TRCV	0030	TDM データ受信レジスタ
DXR1	0031	シリアル・ポート・データ送信レジスタ 1
TDXR	0031	TDM データ送信レジスタ

注:　テーブル内のアドレス値が重複している場合、それらは、異なる 'C54x デバイスにインプリメントされているレジスタの異なる名前をサポートします。

表 4−2.　メモリ・マップド・レジスタ (続き)

名前	16 進アドレス	説明
SPC1	0032	シリアル・ポート制御レジスタ 1
TSPC	0032	TDM シリアル・ポート制御レジスタ
TRAD	0035	TDM 受信アドレス・レジスタ
AXR	0038	ABU アドレス送信レジスタ
TCSR	0033	TDM チャネル選択レジスタ
TRTA	0034	TDM 送受信アドレス・レジスタ
AXR0	0038	ABU アドレス送信レジスタ 0
ARX	0038	ABU 送信アドレス・レジスタ
BKX	0039	ABU 送信バッファ・サイズ・レジスタ
BKX0	0039	ABU 送信バッファ・サイズ・レジスタ 0
ARR	003A	ABU アドレス受信レジスタ
ARR0	003A	ABU アドレス受信レジスタ 0
BKR	003B	ABU 受信バッファ・サイズ・レジスタ
AXR1	003C	ABU 送信アドレス・レジスタ
BKX1	003D	ABU 送信バッファ・サイズ・レジスタ
ARR1	003E	ABU アドレス受信レジスタ 1
BKR1	003F	ABU 受信バッファ・サイズ・レジスタ 1
BDRR1	0040	BSP データ受信レジスタ 1
BDXR1	0041	BSP データ送信レジスタ
BSPC1	0042	BSP 制御レジスタ 1
BSPCE1	0043	BSP 制御拡張レジスタ 1
CLKMD	0058	クロック・モード・レジスタ
XPC	001E	拡張プログラム・カウンタ

注:　テーブル内のアドレス値が重複している場合、それらは、異なる 'C54x デバイスにインプリメントされているレジスタの異なる名前をサポートします。

構文

```
.newblock
```

説明

.newblock 疑似命令を使用すると、現在定義されているローカル・ラベルの定義が取り消されます。ローカル・ラベルは本来、一時的なものですが、.newblock 疑似命令はローカル・ラベルをリセットし、スコープを終了させます。

ローカル・ラベルは、$n という形式のラベルです。ここで、n は 1 桁の 10 進数字を表します。ローカル・ラベルは、ほかのラベルと同様に命令ワードを指します。しかし、ほかのラベルと違って、式の中で使用することはできません。ローカル・ラベルはシンボル・テーブルには含まれません。

ローカル・ラベルを定義し、使用した後、.newblock 疑似命令を使用してローカル・ラベルをリセットする必要があります。.text、.data、および名前付きセクションもローカル・ラベルをリセットします。インクルード・ファイル内で定義されたローカル・ラベルは、ローカル・ファイルの外部では無効です。

例

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

```
 1                      .ref   ADDRA, ADDRB, ADDRC
 2        0076  B       .set   76h
 3
 4 000000 1000! LABEL1: LD     ADDRA, A
 5 000001 F010          SUB    #B, A
   000002 0076
 6 000003 F843          BC     $1, ALT
   000004 0008'
 7 000005 1000!         LD     ADDRB, A
 8 000006 F073          B      $2
   000007 0009'
 9
10 000008 1000! $1      LD     ADDRA, A
11 000009 0000! $2      ADD    ADDRC, A
12                      .newblock  ; Undefine $1 to reuse
13 00000a F843          BC     $1, ALT
   00000b 000D'
14 00000c 8000!         STL    A, ADDRC
15 00000d F495  $1      NOP
```

構文

.noremark [*num*]
.remark [*num*]

説明

.noremark 疑似命令は、*num* により特定されたアセンブラ注釈を抑止します。*num* が指定されない場合、すべての注釈は抑止されます。注釈はアセンブラの情報メッセージであり、警告ほど深刻ではありません。

この疑似命令は、−r[*num*] アセンブラ・オプションを使用することと同じです。

.remark 疑似命令は、すでに抑止された注釈を再び可能にします。

構文

.option　*option list*

説明

.option 疑似命令では、アセンブラ出力リスト用にいくつかのオプションを選択します。*option list* には、縦線で区切ったオプションのリストを指定します。各オプションはリスト作成機能を選択するために使用します。有効なオプションには、次のものがあります。

B	.byte 疑似命令によるリスト出力を 1 行に制限します。
L	.long 疑似命令によるリスト出力を 1 行に制限します。
M	リストでのマクロ展開を行わないようにします。
R	B、M、T、および W のオプションをリセットします。
T	.string 疑似命令によるリスト出力を 1 行に制限します。
W	.word 疑似命令によるリスト出力を 1 行に制限します。
X	シンボルのクロスリファレンス・リストを作成します (アセンブラを起動するときに、−x オプションを指定してクロスリファレンス・リストを取得することもできます)。

オプションは大文字と小文字の区別をしません。

例　この例は、.byte、.word、.long、および .string 疑似命令によるリスト出力をそれぞれ1行に制限する方法を示しています。

```
 1                ****************************************
 2                ** Limit the listing of .byte, .word, **
 3                ** .long, and .string directives      **
 4                **         to 1 line each.            **
 5                ****************************************
 6                        .option B, W, L, T
 7 000000 00BD           .byte   –'C', 0B0h, 5
 8 000004 AABB           .long   0AABBCCDDh, 536 + 'A'
 9 000008 15AA           .word   5546, 78h
10 00000a 0045           .string "Extended Registers"
11
12                ****************************************
13                **     Reset the listing options.     **
14                ****************************************
15                       .option R
16 00001c 00BD           .byte   –'C', 0B0h, 5
   00001d 00B0
   00001e 0005
17 000020 AABB           .long   0AABBCCDDh, 536 + 'A'
   000021 CCDD
   000022 0000
   000023 0259
18 000024 15AA           .word   5546, 78h
   000025 0078
19 000026 0045           .string "Extended Registers"
   000027 0078
   000028 0074
   000029 0065
   00002a 006E
   00002b 0064
   00002c 0065
   00002d 0064
   00002e 0020
   00002f 0052
   000030 0065
   000031 0067
   000032 0069
   000033 0073
   000034 0074
   000035 0065
   000036 0072
   000037 0073
```

構文

```
.page
```

説明

.page 疑似命令は、リスト・ファイルの新たなページを作成します。.page 疑似命令はソース・リストには出力されませんが、アセンブラは、この疑似命令を検出すると行カウンタをインクリメントします。.page 疑似命令を使用してソース・リストをいくつかの論理部に分割すると、プログラムが読みやすくなります。

例

この例は、.page 疑似命令によって、アセンブラがどのようにソース・リストの新しいページを開始するかを示しています。

ソース・ファイル:

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

リスト・ファイル:

```
TMS320C54x COFF Assembler       Version x.xx
Copyright (c) 2001    Texas Instruments Incorporated
**** Page Directive Example ****                              PAGE    1

       2            ;        .
       3            ;        .
       4            ;        .
TMS320C54x COFF Assembler       Version x.xx
Copyright (c) 2001    Texas Instruments Incorporated
**** Page Directive Example ****                              PAGE    2
```

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、ページ位置合わせに似た（ただしそれより制約の弱い）機能をもつアドレス位置合わせメカニズムです。ブロック化されたセクションは、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。この疑似命令は、初期化されたセクションに対してのみブロック化を指定できます。.usect または .bss 疑似命令で宣言された初期化されないセクションは指定できません。*section name* は引用符で囲んでも囲まなくてもかまいません。

例

この例では、.text セクションと .data セクションがブロック化用に指定されています。

```
1 ****************************************
2 ** Specify blocking for the .text     **
3 ** and .data sections.                **
4 ****************************************
5        .sblock       .text, .data
```

構文

.sect　”*section name***”**

説明

.sect 疑似命令は、デフォルトの .text セクションや .data セクションのように使用される名前付きセクションを定義します。.sect 疑似命令は、名前付きセクションへのソース・コードのアセンブリを開始します。

section name は、アセンブラがアセンブルするコードを書き込むセクションを識別します。名前は、200 文字まで指定でき、二重引用符で囲まなければなりません。セクション名には、*section name : subsection name* の形式でサブセクション名を指定することができます。COFF1 形式のファイルの場合、最初の 8 文字のみが有効です。

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

例

この例は、専用セクション Vars を定義し、そこにコードをアセンブルする方法を示しています。

```
 1                ********************************************
 2                **   Begin assembling into .text section.   **
 3                ********************************************
 4 000000                .text
 5 000000 E878           LD      #78h, A ; Assembled into .text
 6 000001 F000           ADD     #36h, A ; Assembled into .text
   000002 0036
 7                ********************************************
 8                **   Begin assembling into Vars section.    **
 9                ********************************************
10 000000                .sect   ”Vars”
11         0010  WORD_LEN        .set    16
12         0020  DWORD_LEN       .set    WORD_LEN * 2
13         0008  BYTE_LEN        .set    WORD_LEN / 2
14               ********************************************
15               **   Resume assembling into .text section.  **
16               ********************************************
17 000003               .text
18 000003 F000          ADD     #42h, A ; Assembled into .text
   000004 0042
19 000005 0003          .byte   3, 4    ; Assembled into .text
   000006 0004
20               ********************************************
21               **   Resume assembling into Vars section.   **
22               ********************************************
23 000000               .sect   ”Vars”
24 000000 000D          .field  13, WORD_LEN
25 000001 0A00          .field  0Ah, BYTE_LEN
26 000002 0000          .field  10q, DWORD_LEN
   000003 0008
27
```

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、そのシンボルはアセンブリ・ソースの中で値の代わりに使用できるようになります。この方法で、意味のある名前を定数や他の値と等価にすることができます。

- *symbol* は、ラベル・フィールドに指定しなければならないラベルです。
- *value* は、整合定義式でなくてはなりません。つまり、式の中のすべてのシンボルは、前もって現行のソース・モジュール内で定義されていなくてはなりません。

未定義の外部シンボル、またはモジュールで後で定義されるシンボルを式で使用することはできません。式が再配置可能な場合は、その式に割り当てられるシンボルも再配置可能となります。

式の値は、リストのオブジェクト・フィールドに示されます。この値は実際のオブジェクト・コードの一部ではなく、出力ファイルには書き込まれません。

.set または .equ を使用して定義されたシンボルは、.def または .global 疑似命令を使用して外部から参照できます。この方法で、グローバルな絶対定数を定義できます。

例

この例は、.set と .equ を使用したシンボルの割り当て方法を示しています。

```
 1                ********************************************
 2                **   Equate symbol AUX_R1 to register AR1   **
 3                **   and use it instead of the register.    **
 4                ********************************************
 5        0011    AUX_R1  .set    AR1
 6 000000 7711            STM     #56h, AUX_R1
   000001 0056
 7
 8                ********************************************
 9                **   Set symbol index to an integer expr.   **
10                **    and use it as an immediate operand.   **
11                ********************************************
12        0035    INDEX   .equ    100/2 +3
13 000002 F000            ADD     #INDEX, A
   000003 0035
14
15                ********************************************
16                ** Set symbol SYMTAB to a relocatable expr. **
17                **   and use it as a relocatable operand.   **
18                ********************************************
19 000004 000A    LABEL   .word   10
20        0005'   SYMTAB  .set    LABEL + 1
21
22                ********************************************
23                **   Set symbol NSYMS equal to the symbol   **
24                **   INDEX and use it as you would INDEX.   **
25                ********************************************
26        0035    NSYMS   .set    INDEX
27 000005 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、現行のセクションに *size* に指定された数のビットを確保し、それを 0 で埋めます。

.space 疑似命令でラベルを使用すると、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用すると、そのラベルは確保された最後のワードを指します。

例

この例は、.space と .bes 疑似命令を使用してメモリを確保する方法を示しています。

```
 1                ****************************************
 2                **  Begin assembling into .text section.**
 3                ****************************************
 4 000000                    .text
 5
 6                ****************************************
 7                **  Reserve 0F0 bits (15 words in the   **
 8                **            .text section).           **
 9                ****************************************
10 000000                    .space  0F0h
11 00000f 0100               .word   100h, 200h
   000010 0200
12                ****************************************
13                **  Begin assembling into .data section.**
14                ****************************************
15 000000                    .data
16 000000 0049               .string "In .data"
   000001 006E
   000002 0020
   000003 002E
   000004 0064
   000005 0061
   000006 0074
   000007 0061
17                ****************************************
18                **  Reserve 100 bits in the.datasection;**
19                **   RES_1 points to the first word that**
20                **         contains reserved bits.      **
21                ****************************************
22 000008         RES_1:  .space  100
23 00000f 000F            .word   15
24 000010 0008"           .word   RES_1
25
26                ****************************************
27                **  Reserve 20 bits in the .datasection;**
28                **  RES_2 points to the last word that  **
29                **         contains reserved bits.      **
30                ****************************************
31 000012         RES_2:    .bes    20
32 000013 0036              .word   36h
33 000014 0012"             .word   RES_2
```

構文

```
.sslist
.ssnolist
```

説明

この 2 つの疑似命令を使用すると、リスト・ファイル内での置換シンボルの展開を制御できます。

- **.sslist** 疑似命令を使用すると、リスト・ファイルでの置換シンボルの展開が可能になります。展開された行は実際のソース行の下に示されます。
- **.ssnolist** 疑似命令は、リスト・ファイルでの置換シンボルの展開を抑止します。

デフォルトでは、リスト・ファイルでのすべての置換シンボルの展開は抑止されます。(#) 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

この例は、デフォルトで置換シンボルの展開のリスト作成を禁止するコードを示しています。また、.sslist 疑似命令をアセンブルし、アセンブラに対して置換シンボル・コードの展開をリストに出力するように指示しています。

(a) ニーモニックの例

```
     1 000000                .bss    ADDRX, 1
     2 000001                .bss    ADDRY, 1
     3 000002                .bss    ADDRA, 1
     4 000003                .bss    ADDRB, 1
     5               ADD2    .macro  ADDRA, ADDRB
     6                       LD      ADDRA, A
     7                       ADD     ADDRB, A
     8                       STL     A, ADDRB
     9                       .endm
    10
    11 000000 8094           STL     A, *AR4+
    12 000001                ADD2    ADDRX, ADDRY
1      000001 1000-          LD      ADDRX, A
1      000002 0001-          ADD     ADDRY, A
1      000003 8001-          STL     A, ADDRY
    13
    14                       .sslist
    15
    16 000004 8094           STL     A, *AR4+
    17 000005 8090           STL     A, *AR0+
    18
    19 000006                ADD2    ADDRX, ADDRY
1      000006 1000-          LD      ADDRA, A
#                            LD      ADDRX, A
1      000007 0001-          ADD     ADDRB, A
#                            ADD     ADDRY, A
1      000008 8001-          STL     A, ADDRB
#                            STL     A, ADDRY
```

(b) 代数表記の例

```
     1 000000             .bss    ADDRX, 1
     2 000001             .bss    ADDRY, 1
     3 000002             .bss    ADDRA, 1
     4 000003             .bss    ADDRB, 1
     5              ADD2  .macro  ADDRA, ADDRB
     6                    A = @ADDRA
     7                    A = A + @ADDRB
     8                    @ADDRB = A
     9                    .endm
    10
    11 000000 8094        *AR4+ = A
    12 000001             ADD2    ADDRX, ADDRY
1      000001 1000-       A = @ADDRX
1      000002 0001-       A = A + @ADDRY
1      000003 8001-       @ADDRY = A
    13
    14                    .sslist
    15
    16 000004 8094        *AR4+ = A
    17 000005 8090        *AR0+ = A
    18
    19 000006             ADD2    ADDRX, ADDRY
1      000006 1000-       A = @ADDRA
#                         A = @ADDRX
1      000007 0001-       A = A + @ADDRB
#                         A = A + @ADDRY
1      000008 8001-       @ADDRB = A
#                         @ADDRY = A
```

構文

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

説明

.string と **.pstring** 疑似命令は、文字列からの 8 ビット文字を現行のセクションに入れます。.string 疑似命令では、16 ビット・ワードに 1 つの 8 ビット文字が入れられますが、.pstring 疑似命令では、各ワードに 8 ビット文字が 2 つ含まれるようにデータがパックされます。各 *string* には、次のいずれかを指定できます。

- ❑ アセンブラが計算し、16 ビットの符号付き数値として取り扱う式
- ❑ 二重引用符に囲まれた文字列。文字列内の各文字は、別々のバイトを表します。

.pstring では、値がワードにパックされる際に、各ワードの最上位のバイトから埋められていきます。未使用の空間には、ヌル・バイトが埋め込まれます。

アセンブラは、8 ビットよりも大きい値はすべて切り捨てます。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必要があります。

ラベルを使用する場合、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 はメンバのサイズを定義し、メモリを初期化しないことに注意してください。.struct/.endstruct の詳細は、4-21 ページの 4.9 節「アセンブル時シンボル疑似命令」を参照してください。

例

この例では、現行のセクションのワードに 8 ビット値が格納されています。

```
  1 000000 0041  Str_Ptr:  .string  ”ABCD”
    000001 0042
    000002 0043
    000003 0044
  2 000004 0041            .string  41h, 42h, 43h, 44h
    000005 0042
    000006 0043
    000007 0044
  3 000008 4175            .pstring ”Austin”, ”Houston”
    000009 7374
    00000a 696E
    00000b 486F
    00000c 7573
    00000d 746F
    00000e 6E00
  4 00000f 0030            .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体定義の要素に割り当てます。これにより、類似するデータ要素をグループ化してから、アセンブラに要素のオフセットを計算するように指示できます。この方式は、C 構造体または Pascal レコードに似ています。.struct 定義には、.union 定義を含めることができます。また、.struct と .union をネストすることもできます。.struct 疑似命令は、メモリの割り当ては行いません。反復使用が可能な、シンボリックなテンプレートを作成するだけです。

.endstruct 疑似命令は、構造体定義を終了させます。

.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 疑似命令はメモリの割り当ては行いません。.tag 疑似命令の構造体タグ (stag) は、事前に定義しておかなければなりません。

stag　構造体のタグです。この値は、構造体の始まりを示します。stag が指定されていない場合、アセンブラは、構造体のメンバを、構造体の最上部からの絶対オフセットを値とするシンボルとしてグローバル・シンボル・テーブルに入れます。stag は、.struct の場合は任意選択ですが、.tag の場合は必須です。

expr　構造体の始まりのオフセットを示す任意選択の式です。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。このパラメータは、トップレベルの構造体でのみ使用できます。ネストされた構造体を定義しているときには使用できません。

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オフセットと等価です。構造体のメンバに対するラベルはグローバル宣言ができません。

element　次の疑似命令のどれか 1 つを指定します。.byte、.char、.double、.field、.float、.half、.int、.long、.short、.string、.ubyte、.uchar、.uhalf、.uint、.ulong、.ushort、.uword、および .word。また、element（要素）は、ネストされた構造体や共用体の完全な宣言、またはタグで宣言された構造体や共用体でもかまいません。これらの疑似命令は、.struct 疑似命令の後に使用すると、要素のサイズを表します。これらの疑似命令は、メモリを割り当てません。

expr$_n$　記述される要素の数を指定する、任意選択の式です。この値はデフォルトで 1 となります。.string 要素のサイズは 1 ワード、.field 要素は 1 ビットと見なされます。

size　構造体全体のサイズを示す、任意選択のラベルです。

注:　.struct/.endstruct シーケンスで使用できる疑似命令の種類

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

以下の例は、.struct、.tag、および .endstruct 疑似命令のさまざまな使用方法を示しています。

例 1

```
 1                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 000000 0001-             ADD  REAL + REAL_REC.DEN, A
 7                                                   ; access structure element
 8
 9 000000                   .bss REAL, REAL_LEN      ; allocate mem rec
```

例 2

```
10                CPLX_REC  .struct
11          0000  REALI     .tag REAL_REC            ; stag
12          0002  IMAGI     .tag REAL_REC            ; member1 = 0
13          0004  CPLX_LEN  .endstruct               ; cplx_len = 4
14
15                COMPLEX   .tag CPLX_REC            ; assign structure attrib
16
17 000002                  .bss COMPLEX, CPLX_LEN
18
19 000001 0002-            ADD  COMPLEX.REALI, A    ; access structure
20 000002 8002-            STL  A, COMPLEX.REALI
21
22 000003 0104-            ADD  COMPLEX.IMAGI, B    ; allocate space
```

例 3

```
 1                          .struct                 ; no stag puts mems into
 2                                                    ; global symbol table
 3          0000  X         .int                    ; create 3 dim templates
 4          0001  Y         .int
 5          0002  Z         .int
 6          0003            .endstruct
```

例 4

```
 1                BIT_REC   .struct                 ; stag
 2          0000  STREAM    .string 64
 3          0040  BIT7      .field  7               ; bits1 = 64
 4          0040  BIT9      .field  9               ; bits2 = 64
 5          0041  BIT10     .field  10              ; bits3 = 65
 6          0042  X_INT     .int                    ; x_int = 66
 7          0043  BIT_LEN   .endstruct              ; length = 67
 8
 9                BITS      .tag BIT_REC
10 000000 0040-            ADD  BITS.BIT7, A       ; move into acc
11 000001 f030             AND  #007Fh, A          ; mask off garbage bits
   000002 007f
12
13 000000                  .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。デフォルトのタブのサイズは、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 命令から構成されています。

ソース・ファイル:

```
; default tab size
   NOP
   NOP
   NOP

     .tab 4
   NOP
   NOP
   NOP

     .tab 16
   NOP
   NOP
   NOP
```

リスト・ファイル:

```
       1             ; default tab size
       2 000000 F495           NOP
       3 000001 F495           NOP
       4 000002 F495           NOP
       5
       7 000003 F495       NOP
       8 000004 F495       NOP
       9 000005 F495       NOP
      10
      12 000006 F495                   NOP
      13 000007 F495                   NOP
      14 000008 F495                   NOP
```

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書き込まれます。まだ .text セクションへのアセンブルが行われていない場合、セクション・プログラム・カウンタは 0 に設定されます。すでにコードが .text セクションにアセンブルされている場合は、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、異なるセクション疑似命令 (.data または .sect) を指定しなければ、アセンブル開始時にアセンブラは、.text セクションにコードをアセンブルします。

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

例

この例では、コードを .text セクションと .data セクションにアセンブルします。.data セクションには整数定数が書き込まれ、.text セクションには実行可能なコードが書き込まれます。

```
 1                ****************************************
 2                ** Begin assembling into .data section.**
 3                ****************************************
 4 000000                 .data
 5 000000 000a            .byte   0Ah, 0Bh
   000001 000b
 6
 7                ****************************************
 8                ** Begin assembling into .text section. **
 9                ****************************************
10 000000                 .text
11 000000 0041   START:   .string "A","B","C"
   000001 0042
   000002 0043
12 000003 0058   END:     .string "X","Y","Z"
   000004 0059
   000005 005a
13
14 000006 0000'           ADD     START, A
15 000007 0003'           ADD     END, A
16               ****************************************
17               **  Resume assembling into .data section.**
18               ****************************************
19 000002                 .data
20 000002 000c            .byte   0Ch, 0Dh
   000003 000d
21               ****************************************
22               **  Resume assembling into .text section.**
23               ****************************************
24 000008                 .text
25 000008 0051            .string "Quit"
   000009 0075
   00000a 0069
   00000b 0074
```

構文

.title　”*string***”**

説明

.title 疑似命令は、各リスト・ページのヘッダ部分に出力されるタイトルを提供します。.title 文自体はリストに出力されませんが、行カウンタはインクリメントされます。

string には、引用符で囲まれたタイトルを 65 文字まで指定できます。65 文字を超えると、アセンブラは文字列を切り捨て、警告を発行します。

アセンブラは、.title 疑似命令に続くページにタイトルを出力します。そして別の .title 疑似命令が処理されるまで、以降のページに出力し続けます。リストの最初のページにタイトルを出力する場合は、最初のソース文に .title 疑似命令を指定しなければなりません。

例

この例では、あるタイトルは 1 ページ目に出力され、別のタイトルは後続ページに出力されます。

ソース・ファイル:

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

リスト・ファイル:

```
TMS320C54x COFF Assembler       Version x.xx
Copyright (c) 2001    Texas Instruments Incorporated
**** Fast Fourier Transforms ****                              PAGE    1

       2            ;        .
       3            ;        .
       4            ;        .
TMS320C54x COFF Assembler       Version x.xx
Copyright (c) 2001    Texas Instruments Incorporated
**** Floating-Point Routines ****                              PAGE    2
```

構文

[*utag*]	**.union**	[*expr*]
[mem_0]	*element*	[$expr_0$]
[mem_1]	*element*	[$expr_1$]
.	.	.
.	.	.
.	.	.
[mem_n]	**.tag**	$utag_n$[, $expr_n$]
.	.	.
.	.	.
.	.	.
[mem_N]	*element*	[$expr_N$]
[*size*]	**.endunion**	
label	**.tag**	*utag*

説明

.union 疑似命令は、シンボリックなオフセットを、同じメモリ空間に割り当てられる代替データ構造定義の要素に割り当てます。この疑似命令を使用すると、いくつかの代替構造を定義して、アセンブラに要素オフセットを計算させることができます。この機能は、C の共用体に似ています。.union 疑似命令は、メモリの割り当ては行いません。反復使用が可能な、シンボリックなテンプレートを作成するだけです。

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

.endunion 疑似命令は、共用体の定義を終了します。

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

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

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

mem$_n$　共用体のメンバを示す任意選択のラベルです。このラベルは、絶対的であり、共用体の始まりからのオフセットと等価です。共用体のメンバに対するラベルはグローバル宣言できません。

element　次の疑似命令のどれか 1 つを指定します。.byte、.char、.double、.field、.float、.half、.int、.long、.short、.string、.ubyte、.uchar、.uhalf、.uint、.ulong、.ushort、.uword、および .word。また、element（要素）は、ネストされた構造体や共用体の完全な宣言、またはタグで宣言された構造体や共用体でもかまいません。これらの疑似命令は、.union 疑似命令の後に使用すると、要素のサイズを表します。これらの疑似命令は、メモリを割り当てません。

expr$_n$　記述される要素の数を指定する、任意選択の式です。この値は、デフォルトで 1 となります。.string 要素のサイズは 1 ワード、.field 要素は 1 ビットと見なされます。

size　共用体全体のサイズを示す、任意選択のラベルです。

注:　.union/.endunion シーケンスで使用できる疑似命令の種類

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

次の例は、タグがある共用体と、タグがない共用体を示しています。

例 1

```
 1                          .global employid
 2              xample      .union                      ; utag
 3        0000  ival        .word                       ; member1 = int
 4        0000  fval        .float                      ; member2 = float
 5        0000  sval        .string                     ; member3 = string
 6        0002  real_len    .endunion                   ; real_len = 2
 7
 8 000000                   .bss  employid, real_len ;allocate memory
 9
10              employid    .tag  xample                ; name an instance
11 000000 0000-             ADD   employid.fval, A      ; access union element
```

例 2

```
 1
 2                          .union                      ; utag
 3        0000  x           .long                       ; member1 = long
 4        0000  y           .float                      ; member2 = float
 5        0000  z           .word                       ; member3 = word
 6        0002  size_u      .endunion                   ; real_len = 2
 7
```

構文

symbol **.usect "***section name***",** *size in words* **[,** [*blocking flag*] [, *alignment 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用の空間を確保します。この疑似命令は、.bss 疑似命令に似ています。両方ともデータ用の空間を確保するだけで、内容はありません。ただし、.usect は、.bss セクションとは無関係に、メモリ内の任意の場所に配置できる追加のセクションを定義します。

symbol	.usect 疑似命令の実行によって確保される最初の位置を指します。空間を確保する変数の名前に対応するシンボルを指定してください。
section name	二重引用符で囲まなければなりません。このパラメータは、初期化されないセクションに名前を付けます。名前は、200 文字まで指定できます。COFF1 フォーマットのファイルの場合、最初の 8 文字のみが有効です。セクション名には、*section name:subsection name* の形式でサブセクション名を書き込むこともできます。
size in words	*section name* で指定された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	任意のパラメータです。指定した値がゼロ以外の場合、このフラグは、このセクションがブロック化されることを表します。ブロック化は、位置合わせに似た (ただしそれより制約の弱い) 機能をもつアドレス指定メカニズムです。セクションは、1 ページより小さい場合はページ境界 (128 ワード) にまたがらないように設定され、1 ページより大きい場合はページ境界から始まるように設定されます。このブロック化はセクションに適用されるのであって、.usect 疑似命令のこのインスタンスで宣言したオブジェクトに適用されるのではありません。
alignment flag	任意のパラメータです。このフラグを指定すると、アセンブラは指定されたサイズの空間を倍長ワード境界に割り当てます。

注： 位置合わせフラグのみを指定する方法

ブロック化フラグを指定しないで、位置合わせフラグのみを指定するには、構文が示すように、位置合わせフラグの前にコンマを 2 つ挿入する必要があります。

その他のセクション疑似命令 (.text、.data、および .sect) は、現行のセクションを終了して、アセンブラに別のセクションへのアセンブルを開始するように伝えます。ただし、.usect 疑似命令と .bss 疑似命令は現行のセクションに影響を与えません。アセンブラは、.usect 疑似命令と .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。

メモリ内に隣接して配置される変数は、指定した同じセクション内に定義することができます。これをするには、同じセクション名を使用して .usect 疑似命令を繰り返します。

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

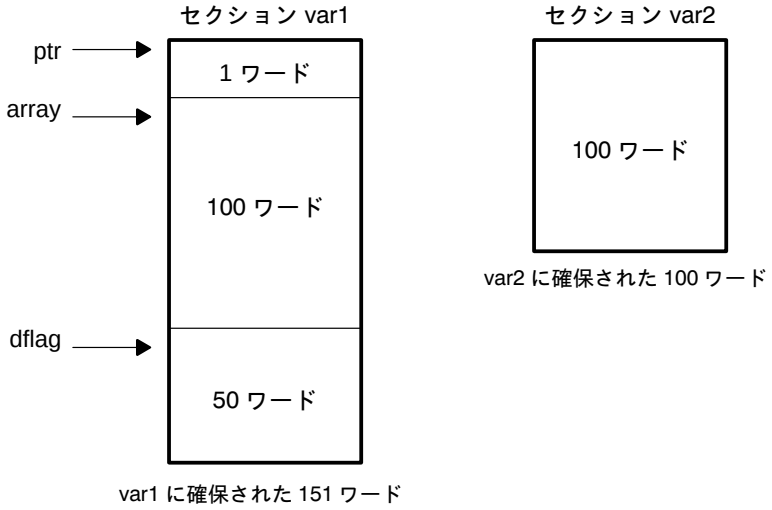
例

この例では、.usect 疑似命令を使用して、2 つの初期化されない名前付きセクション var1 と var2 を定義します。シンボル ptr は、var1 セクションに確保された最初のワードを指します。シンボル配列は、var1 に確保された 100 ワード・ブロック内の最初のワードを指します。また dflag は、var1 内にある 50 ワードのブロックの最初のワードを指します。シンボル vec は、var2 セクションに確保された最初のワードを指します。

4-97 ページの図 4−7 は、この例で 2 つの初期化されないセクション var1 と var2 に、どのように空間が確保されるかを示しています。

```
 1                ****************************************
 2                **     Assemble into .text section.     **
 3                ****************************************
 4 000000                 .text
 5 000000 E803            LD      #03h, A
 6
 7                ****************************************
 8                **       Reserve 1 word in var1.        **
 9                ****************************************
10 000000         ptr     .usect  ”var1”, 1
11
12                ****************************************
13                **      Reserve 100 words in var1.      **
14                ****************************************
15 000001         array   .usect  ”var1”, 100
16
17 000001 F000            ADD     #037h, A  ; Still in .text
   000002 0037
18
19                ****************************************
20                **      Reserve 50 words in var1.       **
21                ****************************************
22 000065         dflag   .usect  ”var1”, 50
23
24 000003 F000            ADD     #dflag, A ; Still in .text
   000004 0065-
25
26                ****************************************
27                **      Reserve 100 words in var2.      **
28                ****************************************
29 000000         vec     .usect  ”var2”, 100
30
31 000005 F000            ADD     #vec, A   ; Still in .text
   000006 0000-
32                ****************************************
33                **  Declare an external .usect symbol.  **
34                ****************************************
35                        .global array
```

図 4−7.　.usect 疑似命令



構文

.var　sym_1 [,sym_2, ... , sym_n]

説明

.var 疑似命令を使用すると、マクロ内で置換シンボルをローカル変数として使用できます。この疑似命令を使用して、マクロ当たり 32 個までローカル・マクロ置換シンボル (パラメータを含む) を定義できます。

.var 疑似命令は、ヌル文字列の初期値をもつ一時的な置換シンボルを作成します。このシンボルはパラメータとして渡されることはなく、展開後は消滅します。

マクロの詳細は、第 5 章「マクロ言語」を参照してください。

構文

.version　*value*

説明

.version 疑似命令は、どのプロセッサ用の命令を生成するかを決定します。*value* には、次のいずれかを指定してください。

541
542
543
545
545LP
546LP
548
54_ext (拡張プログラム・スペースをもつ、その他すべての C54x デバイス用)

第 5 章

マクロ言語

アセンブラでは、ユーザ独自の「命令」を作ることができるマクロ言語をサポートしています。特にプログラムに何度も同じような作業を実行させる場合に、この機能を使用すると便利です。マクロ言語を使用すると、以下のことができます。

- ❑ 独自のマクロを定義し、既存のマクロを再定義する。
- ❑ 長い、または複雑なアセンブリ・コードを単純化する。
- ❑ アーカイバを使って作成したマクロ・ライブラリにアクセスする。
- ❑ マクロ内に条件付きブロック、および繰り返しブロックを定義する。
- ❑ マクロ内で文字列を操作する。
- ❑ 展開リスト出力を制御する。

項目 **ページ**

5.1 マクロの使用方法 5-2
5.2 マクロの定義方法 5-3
5.3 マクロ・パラメータ/置換シンボル 5-6
5.4 マクロ・ライブラリ 5-14
5.5 マクロでの条件付きアセンブリの使用方法 5-15
5.6 マクロでのラベルの使用方法 5-17
5.7 マクロでのメッセージの作成方法 5-19
5.8 出力リストをフォーマットする方法 5-21
5.9 再帰的なマクロとネストされたマクロの使用方法 5-22
5.10 マクロ疑似命令のまとめ 5-25

5.1 マクロの使用方法

プログラムには、何回か実行されるルーチンが含まれていることがしばしばあります。このような場合、ルーチンのソース文を繰り返す代わりに、ルーチンをマクロとして定義し、通常はルーチンを繰り返す場所でそのマクロを呼び出すことができます。この方法により、短く簡潔なソース・プログラムを書くことができます。

同じマクロを何度か呼び出す場合、ただし呼び出しごとに別のデータを使用するときには、マクロ内部でデータにパラメータを割り当てることができます。パラメータを割り当てることにより、マクロを呼び出すたびに、異なる情報をマクロに渡すことができます。マクロ言語では、置換シンボルと呼ばれる特殊シンボルがサポートされています。置換シンボルは、マクロ・パラメータで使用されます。

マクロの使用には、次の 3 つの手順があります。

手順 1: **マクロの定義。**プログラムでマクロを使用するためには、まずマクロを定義する必要があります。マクロを定義するには、次の 2 つの方法があります。

- ❑ マクロは、**ソース・ファイル**の先頭で定義することも、.copy/.include ファイルで定義することもできます。詳細は、5.2 節「マクロの定義方法」を参照してください。
- ❑ マクロは**マクロ・ライブラリ**で定義することができます。マクロ・ライブラリとは、アーカイバを使用して作成したアーカイブ・フォーマットのファイルの集合のことです。アーカイブ・ファイル (マクロ・ライブラリ) の個々のメンバには、そのメンバ名に対応したマクロ定義が 1 つ入っています。マクロ・ライブラリにアクセスするには、.mlib 疑似命令を使用します。詳細は、5-14 ページの 5.4 節「マクロ・ライブラリ」を参照してください。

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・プログラムの中で、ニーモニックとして使用できます。これを「マクロ呼び出し (マクロ・コール)」と言います。

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、アセンブラはそのマクロを展開します。展開時に、アセンブラは、引数を変数によってマクロ・パラメータに渡し、マクロ呼び出し文をマクロ定義に置き換え、ソース・コードをアセンブルします。デフォルトでは、マクロ展開はリスト・ファイルに出力されます。.mnolist 疑似命令を使用すると、展開リストの出力を止めることができます。詳細は、5-21 ページの 5.8 節「出力リストをフォーマットする方法」を参照してください。

アセンブラはマクロ定義を見つけると、そのマクロ名を命令コード・テーブルに入れます。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、または命令ニーモニックは、見つかったマクロと同じ名前のものがあれば再定義されます。これにより、新しい命令を追加するだけでなく、疑似命令および命令の関数を展開することができます。

5.2 マクロの定義方法

マクロはプログラム内のどこにでも定義することができますが、使用する前に定義する必要があります。マクロは、ソース・ファイルの先頭でも、.include/.copy ファイルでも、またはマクロ・ライブラリでも定義することができます。マクロ・ライブラリの詳細は、5-14 ページの 5.4 節「マクロ・ライブラリ」を参照してください。

マクロ定義は、ネストさせることも、別のマクロを呼び出すこともできます。ただし、マクロのすべての要素が同じファイルの中で定義されていなければなりません。ネストされたマクロの詳細は、5-22 ページの 5.9 節「再帰的なマクロとネストされたマクロの使用方法」を参照してください。

マクロの定義は、次のようなフォーマットの一連のソース文です。

macname	**.macro** [*parameter*$_1$] [**,** ... **,** *parameter*$_n$]
	model statements or macro directives
	[.mexit]
	.endm

macname	マクロに名前を付けます。名前は、必ずソース文のラベル・フィールドに入れます。マクロ名の最初の 32 文字が有効です。アセンブラは、マクロ名を内部命令コード・テーブルに入れ、すでに同じ名前をもった命令やマクロ定義があれば、それを置き換えます。
.macro	ソース文がマクロ定義の第 1 行目であることを特定します。.macro は、必ず命令コード・フィールドに入れます。
[*parameters*]	.macro 疑似命令のオペランドとして使用される、指定が任意の置換シンボルです。パラメータについては、5-6 ページの 5.3 節「マクロ・パラメータ/置換シンボル」を参照してください。
model statements	マクロが呼び出されるたびに実行される命令、またはアセンブラ疑似命令です。
macro directives	マクロ展開の制御に使用されます。
.mexit	goto .endm 文として機能します。エラー・テストでマクロ展開が失敗すること、および残りのマクロが不要であることが分かった場合、.mexit 疑似命令を使用すると便利です。
.endm	マクロ定義を終了します。

マクロ定義にコメントを含め、そのコメントをマクロ展開には表示したくない場合、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表示する場合は、アスタリスクかセミコロンを使用します。マクロのコメントの詳細は、5-19 ページの 5.7 節「マクロでのメッセージの作成方法」を参照してください。

例 5−1 は、マクロの定義、呼び出し、および展開の例を示しています。

例 5−1. マクロの定義、呼び出し、展開

(a) ニーモニックの例

```
       1              *
       2
       3              *        add3
       4              *
       5              *        ADDRP = P1 + P2 + P3
       6
       7              add3     .macro P1, P2, P3, ADDRP
       8
       9                       LD      P1, A
      10                       ADD     P2, A
      11                       ADD     P3, A
      12                       STL     A, ADDRP
      13                       .endm
      14
      15
      16                       .global abc, def, ghi, adr
      17
      18 000000                add3 abc, def, ghi, adr
1
1        000000 1000!          LD      abc, A
1        000001 0000!          ADD     def, A
1        000002 0000!          ADD     ghi, A
1        000003 8000!          STL     A, adr
```

(b) 代数表記の例

```
      1              *
      2
      3              *        add3
      4              *
      5              *        ADDRP = P1 + P2 + P3
      6
      7              add3     .macro P1, P2, P3, ADDRP
      8
      9                       A = @P1
     10                       A = A + @P2
     11                       A = A + @P3
     12                       @ADDRP = A
     13                       .endm
     14
     15
     16                       .global abc, def, ghi, adr
     17
     18 000000       add3 abc, def, ghi, adr
1
1       000000 1000!          A = @abc
1       000001 0000!          A = A + @def
1       000002 0000!          A = A + @ghi
1       000003 8000!          @adr = A
```

5.3 マクロ・パラメータ/置換シンボル

同じマクロを何度か呼び出す場合、ただし呼び出しごとに別のデータを使用するときは、そのマクロ内部でパラメータを割り当てることができます。マクロ言語では、置換シンボルと呼ばれる特殊シンボルをサポートしています。マクロ・パラメータにこの置換シンボルを使用します。

マクロ・パラメータは文字列を表す置換シンボルです。これらのシンボルは、文字列をシンボル名で代用するという用途でマクロ以外の場所で使用することもできます。

有効な置換シンボルは長さが最高 32 文字までで、先頭を文字で始める必要があります。先頭の文字に続く部分には、英数字、下線、ドル記号を使用できます。

マクロ・パラメータとして使用される置換シンボルは、それが定義されているマクロ内でしか効果がありません。.var 疑似命令を使って定義する置換シンボルを含めて、1 つのマクロにつき 32 個までのローカルな置換シンボルを定義できます。.var 疑似命令の詳細は、5-13 ページの 5.3.6 項「マクロでローカル変数として使用される置換シンボル」を参照してください。

マクロ展開時に、アセンブラは、引数をマクロ・パラメータに渡します。各引数の文字列は、対応するパラメータに割り当てられます。対応する引数をもたないパラメータは、ヌル文字列に設定されます。引数の数がパラメータの数よりも多い場合は、最後のパラメータに残りのすべての引数の文字列が割り当てられます。

引数のリストを 1 つのパラメータに渡す場合、またはコンマやセミコロンをパラメータに渡す場合は、その語を引用符で囲みます。

アセンブル時に、アセンブラは、まず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 5−2 は、引数の数が異なるマクロ展開の例を示したものです。

例 5−2.　引数の数が異なるマクロ呼び出しの例

マクロ定義

```
Parms  .macro a,b,c
;         a = :a:
;         b = :b:
;         c = :c:
       .endm
```

マクロの呼び出し:

```
       Parms  100,label           Parms  100,label,x,y
;         a = 100                        ;   a = 100
;         b = label                      ;   b = label
;         c = ” ”                        ;   c = x,y

       Parms  100, , x            Parms  ”100,200,300”,x,y
;         a = 100                        ;   a = 100,200,300
;         b = ” ”                        ;   b = x
;         c = x                          ;   c = y

       Parms  ”””string”””,x,y
;         a = ”string”
;         b = x
;         c = y
```

5.3.1　置換シンボルを定義するための疑似命令

置換シンボルは、**.asg** 疑似命令および **.eval** 疑似命令を使用して操作できます。

.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除きます。どちらの場合でも、文字列が読み取られて置換シンボルに割り当てられます。

例 5−3 は、置換シンボルに割り当てられた文字列を示しています。

例 5−3.　.asg 疑似命令

```
.asg  AR0,FP              ;  frame pointer
.asg  *AR1+,Ind           ;  indirect addressing
.asg  *AR1+0b,Rc_Prop     ;  reverse carry propagation
.asg  """string""",strng  ;  string
.asg  "a,b,c",parms       ;  parameters
```

.eval 疑似命令は、数値置換シンボルに対して算術を実行します。

.eval 疑似命令の構文は、次のとおりです。

.eval *well-defined expression, substitution symbol*

.eval 疑似命令は、式を計算し、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。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 5−4 は、置換シンボルへの算術の実行を示しています。

例 5−4.　.eval 疑似命令

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

例 5−4 では、.asg 疑似命令を .eval 疑似命令と置き換えても、出力結果は変わりません。このような単純なケースでは、.eval と .asg のどちらを使用してもかまいません。ただし、式から値を計算する場合は、.eval 疑似命令を使用する必要があります。.asg 疑似命令は、文字列を置換シンボルに割り当てるだけですが、.eval 疑似命令は式を計算し、置換シンボルに等価の文字列を割り当てます。

5.3.2　ビルトイン置換シンボル関数

以下のビルトイン置換シンボル関数を使用すると、置換シンボルの文字列値に基づいて判断を行うことができます。これらの関数は、必ず値を戻し、式の中で使用することもできます。ビルトイン置換シンボル関数は、アセンブリの条件式に使用すると非常に便利です。これらの関数のパラメータは、置換シンボルか文字列定数です。

表 5−1 の関数定義では、a と b はパラメータで、置換シンボルまたは文字列定数を表します。文字列という用語は、パラメータの文字列値を表します。*ch* というシンボルは文字列定数を表します。

表 5−1. 関数と戻り値

関数	戻り値
$symlen*(a)*	文字列 *a* の長さ
$symcmp*(a,b)*	< 0 : *a* < *b* の場合 0 : *a* = *b* の場合 > 0 : *a* > *b* の場合
$firstch*(a,ch)*	文字列 *a* の中で文字定数 *ch* が最初に現れる位置
$lastch*(a,ch)*	文字列 *a* の中で文字定数 *ch* が最後に現れる位置
$isdefed*(a)*	1 : 文字列 *a* がシンボル・テーブルで定義されている場合 0 : 文字列 *a* がシンボル・テーブルで定義されていない場合
$ismember*(a,b)*	リスト *b* の先頭のメンバが文字列 *a* に割り当てられます。 0 : *b* がヌル文字列の場合
$iscons*(a)*	1 : 文字列 *a* が 2 進定数の場合 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
$structsz(a)	構造体タグ *a* が表わす構造体のサイズ
$structacc(a)	構造体タグ *a* が表わす構造体の参照点

† 事前定義レジスタ名についての詳細は、3-19 ページの 3.8 節「シンボル」を参照してください。

例 5−5 は、ビルトイン置換シンボル関数を示しています。

例 5−5. ビルトイン置換シンボル関数の使用方法

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

5.3.3 再帰的置換シンボル

アセンブラは置換シンボルを見つけると、対応する文字列の値に置換しようとします。その文字列も置換シンボルである場合は、アセンブラは再度置換を行います。アセンブラは、置換シンボルではないトークンを検出するまで、またはすでに計算の過程で検出されたことのある置換シンボルを検出するまで、置換を続けます。

例 5−6 では、x が z に置換されます。z は y に置換され、y は x に置換されます。アセンブラは、これが無限に再帰的であると判断し、置換を中止します。

例 5−6. 再帰的置換

```
      .asg   ″x″,z  ; declare z and assign z = ″x″
      .asg   ″z″,y  ; declare y and assign y = ″z″
      .asg   ″y″,x  ; declare x and assign x = ″y″
       add      x, A


  *    add      x, A   ; recursive expansion
```

5.3.4 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用すると、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囲めば、置換を強制できます。コロンとシンボルの間には、空白は入れないでください。

強制置換演算子の構文は、次のとおりです。

```
:symbol:
```

アセンブラは、まずコロンで囲まれた置換シンボルを拡張してから、他の置換シンボルを拡張します。

強制置換演算子はマクロの中でのみ使用できますが、別の強制置換演算子でネストすることはできません。

例 5−7 は、強制置換演算子がどのように使用されるかを示しています。

例 5−7. 強制置換演算子の使用例

```
force  .macro x
       .loop  8
AUX:x: .set   x
       .eval  x+1,x
       .endloop
       .endm
       force  0
```

強制マクロは次のソース・コードを生成します。

```
AUX0   .set   0
AUX1   .set   1
   .
   .
   .
AUX7   .set   7
```

5.3.5 添字付き置換シンボルを使用して個々の文字へアクセスする方法

マクロでは、添字付き置換シンボルを使用して置換シンボルの個々の文字（部分文字列）にアクセスできます。その際は、明確に指示するために強制置換演算子を使用する必要があります。

部分文字列へアクセスする方法は 2 つあります。

- :*symbol* (*well-defined expression*):

  この方法で添字を付けると、計算結果は 1 文字の文字列に表されます。

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

  この方法では、expression$_1$ は部分文字列の開始位置を示し、expression$_2$ は部分文字列の長さを表します。添字を付ける正確な位置と、結果の文字列の正確な長さを指定できます。部分文字列のインデックスは、0 ではなく 1 から始まります。

例 5−8 と例 5−9 は、添字付き置換シンボルとともに使用されるビルトイン置換シンボル関数の例を示しています。

例 5−8 では、添字付き置換シンボルは、add 命令を再定義して、この命令がショート型の即値を処理できるようにしています。

例 5−8. 添字付き置換シンボルを使用して命令を再定義する例

```
ADDX    .macro   ABC
        .var     TMP
        .asg     :ABC(1):,TMP
        .if      $symcmp(TMP,"#") = 0
        ADD      ABC, A
        .else
        .emsg    "Bad Macro Parameter"
        .endif
        .endm

        ADDX     #100          ;macro call
        ADDX     *AR1          ;macro call
```

例 5−9 では、添字付き置換シンボルは、文字列 strg2 内の start 位置から始めて、部分文字列 strg1 を検索するために使用されています。部分文字列 strg1 の位置は、置換シンボル pos に割り当てられています。

例 5−9.　添字付き置換シンボルを使用して部分文字列を見つける例

```
substr    .macro    start,strg1,strg2,pos
          .var      LEN1,LEN2,I,TMP
          .if       $symlen(start) = 0
          .eval     1,start
          .endif
          .eval     0,pos
          .eval     1,i
          .eval     $symlen(strg1),LEN1
          .eval     $symlen(strg2),LEN2
          .loop
          .break    i = (LEN2 – LEN1 + 1)
          .asg      ":strg2(i,LEN1):",TMP
          .if       $symcmp(strg1,TMP) = 0
          .eval     i,pos
          .break
          .else
          .eval     i + 1,i
          .endif
          .endloop
          .endm

          .asg      0,pos
          .asg      "ar1 ar2 ar3 ar4",regs
          substr    1,"ar2",regs,pos
          .data
          .word     pos
```

5.3.6　マクロでローカル変数として使用される置換シンボル

置換シンボルをマクロ内でローカル変数として使用する場合は、**.var** 疑似命令を使用します。マクロ 1 つにつき、(パラメータを含めて) 32 個までのローカル・マクロ置換シンボルを定義できます。.var 疑似命令は、ヌル文字列の初期値をもつ一時的な置換シンボルを作成します。このシンボルはパラメータとして渡されることはなく、展開後は消えてしまいます。

.var sym_1 [,sym_2] ... [,sym_n]

.var 疑似命令は、例 5−8 と例 5−9 で使用されています。

5.4 マクロ・ライブラリ

マクロを定義する方法の 1 つは、マクロ・ライブラリを作成することです。マクロ・ライブラリは、マクロ定義を含んだファイルの集合です。このようなファイル、つまりメンバを集めて 1 つのファイル (アーカイブと呼ばれる) にまとめるには、アーカイバを使用する必要があります。マクロ・ライブラリのどのメンバにも、マクロ定義が 1 つ含まれます。マクロ・ライブラリ内のファイルは、アセンブルされていないソース・ファイルでなければなりません。マクロ名とそのメンバ名は同一で、マクロ・ファイル名の拡張子は .asm でなければなりません。次に例を示します。

マクロ名	マクロ・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用します。
.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、ライブラリをオープンし、その内容のテーブルを作成します。アセンブラは、ライブラリ内の個々のメンバ名を、ライブラリ・エントリとして命令コード・テーブルに入れます。もし同じ名前の命令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出力は、.mlist 疑似命令を使用して制御できます。.mlist 疑似命令の詳細は、5-21 ページの 5.8 節「出力リストをフォーマットする方法」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。

アーカイバを使用して必要なファイルをアーカイブに入れるだけで、マクロ・ライブラリを作成できます。アセンブラが、マクロ・ライブラリにはマクロ定義が入っているものとみなす点を除けば、マクロ・ライブラリは他のアーカイブと違いはありません。アセンブラは、マクロ・ライブラリの中にはマクロ定義だけが入っているものと考えます。ライブラリにオブジェクト・コードやいろいろなソース・ファイルを入れると、結果は保証されません。

5.5 マクロでの条件付きアセンブリの使用方法

条件付きアセンブリ疑似命令には、**.if/.elseif/.else/.endif** および **.loop/.break/.endloop** があります。条件付きアセンブリ疑似命令は、32 個のレベルまでネストできます。条件付きブロックのフォーマットは、次のとおりです。

.if *well-defined expression*
[**.elseif** *well-defined expression*]
[**.else**]
.endif

条件付きアセンブリでは **.elseif** 疑似命令と **.else** 疑似命令の指定は任意で、条件付きアセンブリ・コード・ブロック内で 2 回以上使用できます。この 2 つの疑似命令が省略されて、かつ .if 式が偽 (0) の場合、アセンブラは .endif 疑似命令に続くコードのアセンブルへと進みます。.if/ .elseif/.else/.endif 疑似命令の詳細は、4-56 ページを参照してください。

.loop/.break/.endloop 疑似命令を使用すると、コード・ブロックを繰り返してアセンブルできます。繰り返しが可能なブロックのフォーマットは、次のとおりです。

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

.loop 疑似命令の任意の式が計算されて、ループ・カウント (実行するループの数) が設定されます。この式を省略すると、式が真 (非ゼロ) になる .break 疑似命令をアセンブラが検出しない限り、ループ・カウントはデフォルトの 1024 となります。
.loop/.break/.endloop 疑似命令の詳細は、4-66 ページを参照してください。

.break 疑似命令とその式の使用は任意です。式が計算されて偽となった場合は、ループは継続します。アセンブラは、.break 式が真と計算された場合、または .break 式が省略されている場合は、ループを終了します。ループを抜け出すと、アセンブラは .endloop 疑似命令の後のコードのアセンブルに移ります。

例 5−10、例 5−11、および例 5−12 は、.loop/.break/.endloop 疑似命令、適切なネストが行われている条件付きアセンブリ疑似命令、およびビルトイン置換シンボル関数が、条件付きアセンブリ・コード・ブロックでどのように使用されているかを示しています。

例 5−10.　.loop/.break/.endloop 疑似命令

```
       .asg   1,x
       .loop

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

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

例 5−11.　ネストされた条件付きアセンブリ疑似命令

```
       .asg   1,x
       .loop

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

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

例 5−12.　条件付きアセンブリ・コード・ブロックで使用されているビルトイン置換シンボル関数

```
       .ref OPZ
       .fcnolist
*
*Double Add or Subtract
*
DBL    .macro ABC, ADDR, src      ; add or subtract double

       .if $symcmp(ABC,"+") == 0
       dadd   ADDR, src           ; add double

       .elseif $symcmp(ABC,"-") == 0
       dsub   ADDR, src           ; subtract double

       .else
       .emsg  "Incorrect Operator Parameter"

       .endif

       .endm

*Macro Call
       DBL  -, OPZ, A
```

条件付きアセンブリ疑似命令の詳細は、4-20 ページの 4.8 節「条件付きアセンブリ疑似命令」を参照してください。

5.6 マクロでのラベルの使用方法

アセンブリ言語プログラムで使用されるラベルは、どれも固有なものである必要があります。マクロで使用されるラベルについても同様に固有なものにします。マクロが 2 回以上展開されていれば、ラベルも 2 回以上定義されています。ただし、ラベルを 2 回以上定義することは不正です。マクロ言語は、マクロで定義するラベルが必ず固有なものになる方法を提供しています。ラベルの最後に疑問符を付ければ、アセンブラはその疑問符を固有の数字に変更します。マクロを展開しても、リスト・ファイル上でこの固有の数字を参照することはできません。ラベルは、マクロ定義で行われたように疑問符がついて表示されます。このラベルをグローバルとして宣言することはできません。

固有の添字を使用できるようにすると、最大ラベル長が短くなります。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。この固有の添字をもつラベルは、クロスリスティング・ファイルに出力されます。

固有のラベルの構文は、次のとおりです。

label?

例 5−13 は、マクロでの固有のラベルの作成方法を示しています。

例 5−13. マクロでの固有のラベル

(a) ニーモニックの例

```
        1              ; define macro
        2              MIN      .macro AVAR, BVAR ; find minimum
        3
        4                       LD       AVAR, A
        5                       SUB      #BVAR, A
        6                       BC       M1?, ALT
        7                       LD       #BVAR, A
        8                       B        M2?
        9              M1?      LD       AVAR, A
       10              M2?
       11                      .endm
       12
       13              ; call macro
       14 000000               MIN 50, 100
1
1         000000 1032           LD       50, A
1         000001 F010           SUB      #100, A
          000002 0064
1         000003 F843           BC       M1?, ALT
          000004 0008'
1         000005 E864           LD       #100, A
1         000006 F073           B        M2?
          000007 0009'
1         000008 1032   M1?     LD       50, A
1         000009        M2?
```

例 5−13.　マクロでの固有のラベル (続き)

(b) 代数表記の例

```
       1              ; define macro
       2              MIN      .macro AVAR, BVAR ; find minimum
       3
       4                       A = @AVAR
       5                       A = A – #BVAR
       6                       if (ALT) goto M1?
       7                       A = #BVAR
       8                       goto M2?
       9              M1?      A = @AVAR
      10              M2?
      11                       .endm
      12
      13              ; call macro
      14 000000                MIN 50, 100
1
1        000000 1032            A = @50
1        000001 F010            A = A – #100
         000002 0064
1        000003 F843            if (ALT) goto M1?
         000004 0008'
1        000005 E864            A = #100
1        000006 F073            goto M2?
         000007 0009'
1        000008 1032   M1?      A = @50
1        000009        M2?
```

5.7 マクロでのメッセージの作成方法

マクロ言語では、独自のアセンブル時エラー・メッセージと警告メッセージを定義できるように、3 つの疑似命令をサポートしています。この 3 つの疑似命令は、ユーザが独自に必要とする特別のメッセージを作成するのに便利です。リスト・ファイルの最後の行に、エラーと警告の発行された回数が示されます。この回数によってコードの問題点を知ることができるので、特にデバッグ中に役立ちます。

.emsg エラー・メッセージをリスト・ファイルに送ります。.emsg 疑似命令は、アセンブラと同様にエラーを発生させます。つまり、エラー・カウントを 1 つずつインクリメントさせ、アセンブラがオブジェクト・ファイルを作成するのを禁止します。

.mmsg アセンブル時メッセージをリスト・ファイルに送ります。.mmsg 疑似命令は .emsg 疑似命令と同じ機能を果たしますが、エラー回数の設定をせず、かつオブジェクト・ファイルの作成を禁止しない点が異なります。

.wmsg 警告メッセージをリスト・ファイルに送ります。.wmsg 疑似命令は .emsg 疑似命令と同じ機能を果たしますが、警告の回数をインクリメントさせる点と、オブジェクト・ファイルの作成を禁止しない点が異なります。

マクロ・コメントは、マクロの定義で使用されるコメントですが、マクロ展開の中には表示されません。1 カラム目の感嘆符 (!) は、その行がマクロ・コメントであることを示しています。コメントをマクロ展開にも表示する場合は、コメントの前にアスタリスクかセミコロンを付けます。

例 5–14 は、マクロでのユーザ・メッセージと、マクロ展開には表示されないマクロ・コメントを示しています。

例 5−14.　マクロでのメッセージの作成方法

```
    1             testparam   .macro x,y
    2
    3                         .if ($symlen(x) == 0)
    4                         .emsg ”ERROR –– Missing Parameter”
    5                          .mexit
    6                         .elseif ($symlen(y) == 0)
    7                          .emsg ”ERROR == Missing Parameter”
    8                           .mexit
    9                          .else
    10                           LD y, A
    11                           LD x, B
    12                           ADD A, B
    13                         .endif
    14                         .endm
    15
    16 000000                  testparam 1,2
1
1                             .if ($symlen(x) == 0)
1                              .emsg ”ERROR –– Missing Parameter”
1                               .mexit
1                             .elseif ($symlen(y) == 0)
1                              .emsg ”ERROR == Missing Parameter”
1                               .mexit
1                              .else
1     000000 1002                LD 2, A
1     000001 1101                LD 1, B
1     000002 F500                ADD A, B
1                              .endif
    17
    18 000003                  testparam
1
1                             .if ($symlen(x) == 0)
1                             .emsg ”ERROR –– Missing Parameter”
  ***** USER ERROR ***** – : ERROR –– Missing Parameter
1                              .mexit

 1 Error, No Warnings
```

5.8 出力リストをフォーマットする方法

マクロ、置換シンボル、および条件付きアセンブリ疑似命令により、情報が非表示になることがあります。この隠された情報を参照しなければならない場合があるので、マクロ言語は展開リスト作成機能をサポートしています。

デフォルトでは、アセンブラはリスト出力ファイルに、マクロ展開と偽の条件付きブロックを出力します。リスト・ファイル内で、このリスト出力のオン・オフを切り換えることができます。4 組の疑似命令を使用して、この情報のリスト出力を制御できます。

❑ **マクロ展開とループ展開のリスト**

.mlist マクロと .loop/.endloop ブロックを展開します。.mlist 疑似命令は、これらのブロックで検出したすべてのコードをリストに出力します。

.mnolist マクロ展開と .loop/.endloop ブロックのリスト作成が抑止されます。

マクロ展開とループ展開のリスト出力に関しては、.mlist がデフォルトです。

❑ **偽の条件付きブロックのリスト**

.fclist アセンブラがコードを生成しないすべての条件付きブロック (偽の条件付きブロック) をリスト・ファイルに含めるようにします。条件付きブロックは、ソース・コード内と全く同じようにリストに出力されます。

.fcnolist 偽の条件付きブロックのリスト出力を抑止します。条件付きブロック内の、実際にアセンブルを行うコードのみがリストに出力されます。.if、.elseif、.else、および .endif 疑似命令はリストには出力されません。

偽の条件付きブロックのリスト出力に関しては、.fclist がデフォルトです。

❑ **置換シンボル展開リスト**

.sslist リスト内で置換シンボルを展開します。置換シンボルの展開のデバッグに有効です。展開された行は、実際のソース行の下に示されます。

.ssnolist リスト内の置換シンボルの展開を抑止します。

置換シンボル展開のリスト出力に関しては、.ssnolist がデフォルトです。

❑ **疑似命令のリスト**

.drlist アセンブラが、すべての疑似命令行をリスト・ファイルに出力するようにします。

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

疑似命令のリスト出力に関しては、.drlist がデフォルトです。

5.9 再帰的なマクロとネストされたマクロの使用方法

マクロ言語では、再帰的なマクロ呼び出しと、ネストされたマクロ呼び出しをサポートしています。したがって、1 つのマクロ定義の中から別のマクロを呼び出すことができます。マクロは 32 個のレベルまでネストできます。再帰的なマクロを使用すると、マクロをその定義自体 (マクロ呼び出し自体) から呼び出すことができます。

再帰的なマクロやネストされたマクロを作成する際、アセンブラではパラメータの動的な範囲設定が使用されるので、マクロ・パラメータに渡す引数には十分な注意が必要です。これは、呼び出されたマクロは、それを呼び出したマクロの環境を利用するということです。

例 5−15 は、ネストされたマクロを示しています。in_block マクロにある y が out_block マクロにある y を隠すことに注意してください。ただし、out_block マクロにある x と z には、in_block マクロからアクセスできます。

例 5−15. ネストされたマクロの使用方法

```
in_block  .macro y,a
             .          ; visible parameters are y,a and
             .          ;   x,z from the calling macro
          .endm

out_block .macro x,y,z
             .          ; visible parameters are x,y,z
             .
          in_block x,y  ; macro call with x and y as
                        ;   arguments
             .
             .
          .endm
          out_block     ; macro call
```

例 5−16 は、再帰的なマクロを示しています。fact マクロは、n の階乗を計算するのに必要なアセンブリ・コードを作成します。ここで n は即値です。結果はデータ・メモリ・アドレス loc に入ります。fact マクロは、fact1 を呼び出すことによ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 5−16. 再帰的なマクロの使用方法

(a) ニーモニックの例

```
fact   .macro N, loc    ; n is an integer constant
                        ; loc memory address = n!
       .if    N < 2     ; 0! = 1! = 1

       ST     #1, loc
       .else
       ST     #N, loc   ; n >= 2 so, store n at 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
       LD     loc, T    ; multiply present factorial
       MPY    #N, A     ;      by present position
       STL    A, loc    ; save result
       .eval  N – 1, N  ; decrement position
       fact1            ; recursive call
       .endif

       .endm
```

例 5−16. 再帰的なマクロの使用方法 (続き)

(b) 代数表記の例

```
fact   .macro N, loc   ; n is an integer constant
                       ; loc memory address = n!
       .if N < 2       ; 0! = 1! = 1

       @AR0 = #1
       .else
       @AR0 = #N       ; n >= 2 so, store n at loc
                       ; decrement n, and do the
       .eval N – 1, N  ;      factorial of n – 1
       fact1           ; call fact1 with current
                       ;                 environment
       .endif

       .endm

fact1  .macro

       .if N > 1
       T = @AR0        ; multiply present factorial
       A = T * #N      ;      by present position
       @AR0 = A        ; save result
       .eval N – 1, N  ; decrement position
       fact1           ; recursive call
       .endif

       .endm
```

5.10 マクロ疑似命令のまとめ

表 5−2.　マクロの作成方法

ニーモニックと構文	説明
macname **.macro** [*parameter*$_1$]...[*parameter*$_n$]	マクロを定義します。
.mlib *filename*	マクロ定義を含むライブラリを特定します。
.mexit	.endm に移動します。
.endm	マクロ定義を終了します。

表 5−3.　置換シンボルの操作方法

ニーモニックと構文	説明
.asg [**"**]*character string*[**"**], *substitution symbol*	文字列を置換シンボルに割り当てます。
.eval *well-defined expression, substitution symbol*	数値置換シンボルの算術計算を実行します。
.var *substitution symbol*$_1$...[*substitution symbol*$_n$]	ローカル・マクロ・シンボルを定義します。

表 5−4.　条件付きアセンブリ

ニーモニックと構文	説明
.if *well-defined expression*	条件付きアセンブリを開始します。
.elseif *well-defined expression*	任意の条件付きアセンブリ・ブロックです。
.else	任意の条件付きアセンブリ・ブロックです。
.endif	条件付きアセンブリを終了します。
.loop [*well-defined expression*]	反復使用が可能なブロック・アセンブリを開始します。
.break [*well-defined expression*]	任意の反復使用が可能なブロック・アセンブリです。
.endloop	反復使用が可能なブロック・アセンブリを終了します。

表 5−5.　アセンブル時メッセージの作成方法

ニーモニックと構文	説明
.emsg	エラー・メッセージを標準出力に送ります。
.wmsg	警告メッセージを標準出力に送ります。
.mmsg	警告メッセージまたはアセンブル時メッセージを標準出力に送ります。

表 5−6.　リストをフォーマットする方法

ニーモニックと構文	説明
.fclist	偽の条件付きコード・ブロックのリスト出力を許可します (デフォルト)。
.fcnolist	偽の条件付きコード・ブロックのリスト出力を抑止します。
.mlist	マクロ・リストの出力を許可します (デフォルト)。
.mnolist	マクロ・リストの出力を抑止します。
.sslist	置換シンボルの展開リスト出力を許可します。
.ssnolist	置換シンボルの展開リスト出力を抑止します (デフォルト)。

第 6 章

アーカイバの説明

TMS320C54xアーカイバを使用すると、いくつかのファイルを集めて 1 つのアーカイブ・ファイルにできます。たとえば、いくつかのマクロを集めて 1つのマクロ・ライブラリにできます。アセンブラは、このライブラリを検索し、ソース・ファイルによりマクロとして呼び出されたメンバを使用します。アーカイバを使用して、いくつかのオブジェクト・ファイルを集めて 1 つのオブジェクト・ライブラリにすることもできます。リンカは、リンク中に、外部参照を解決するメンバをライブラリから取り込みます。

項目 **ページ**

6.1 アーカイバの概要 6-2
6.2 アーカイバと開発フロー 6-3
6.3 アーカイバの起動方法 6-4
6.4 アーカイバの例 6-6

6.1 アーカイバの概要

TMS320C54xアーカイバを使用すると、いくつかのファイルを結合してアーカイブまたはライブラリと呼ばれる 1 つのファイルを作成できます。アーカイブ内の個々のファイルはメンバと呼ばれます。アーカイブを作成すると、アーカイバを使用してメンバの追加、削除、または抽出を行うことができます。

ライブラリは、どのような種類のファイルからも作成できます。アセンブラとリンカは、どちらもアーカイブ・ライブラリを入力として受け入れます。アセンブラは個々のソース・ファイルが入ったライブラリを使用でき、リンカは個々のオブジェクト・ファイルが入ったライブラリを使用できます。

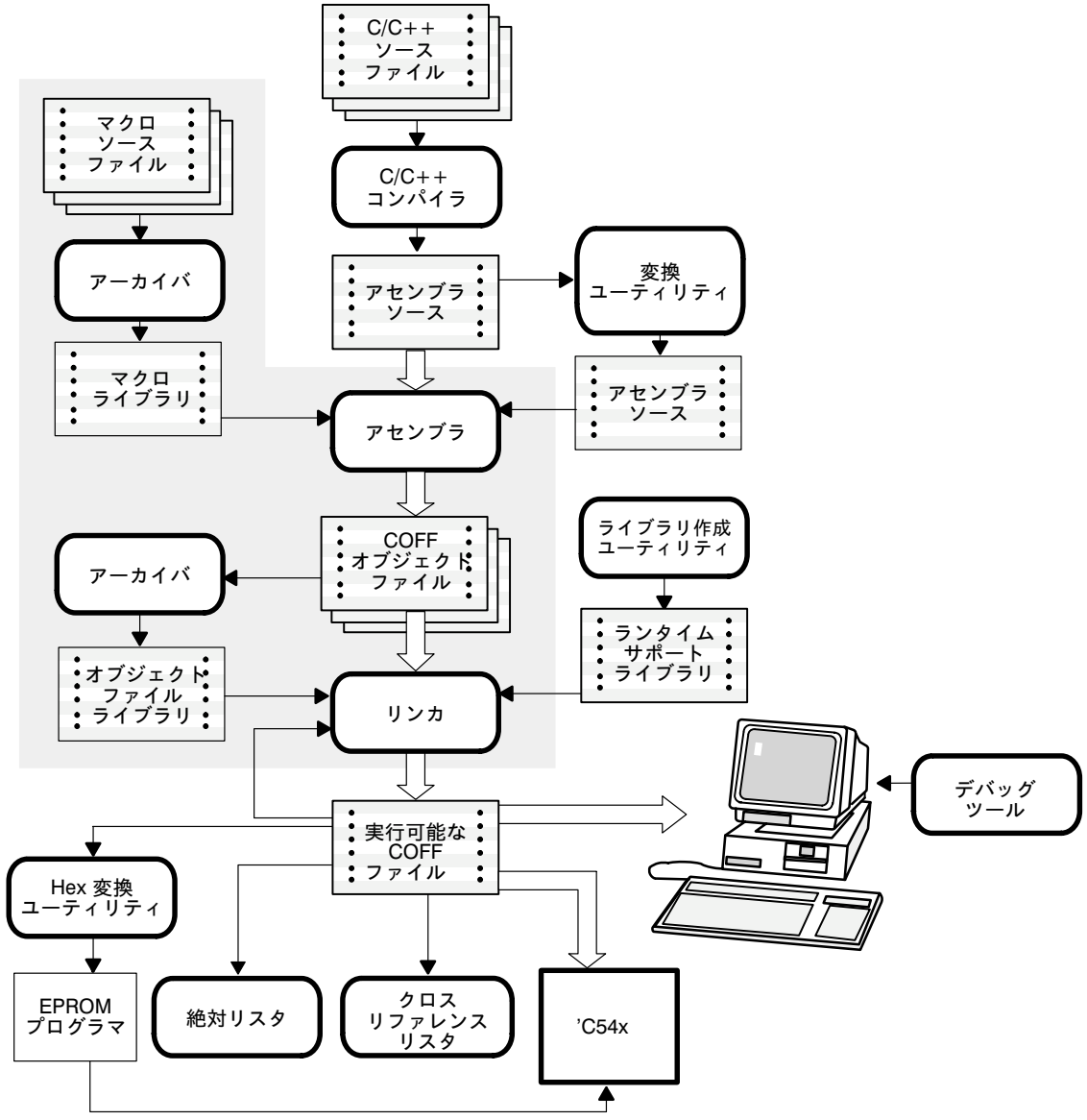
アーカイバの最も便利な使い方の 1 つは、オブジェクト・モジュールのライブラリを作成することです。たとえば、いくつかの算術ルーチンを書いて、アセンブルし、さらにアーカイバを使ってオブジェクト・ファイルを集めて 1つの論理グループを作成します。次に、オブジェクト・ライブラリをリンカ入力として指定します。リンカは、ライブラリ全体を検索し、外部参照を解決するメンバがあれば取り込みます。

アーカイバを使用してマクロ・ライブラリを作成することもできます。それぞれが 1 つのマクロを含むソース・ファイルをいくつか作成し、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用して、マクロ・ライブラリの名前を指定します。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に呼び出されたマクロを検索します。マクロとマクロ・ライブラリの詳細は、第 5 章「マクロ言語」を参照してください。

6.2 アーカイバと開発フロー

図 6−1 は、アセンブリ言語開発プロセスにおけるアーカイバの役割を示したものです。アセンブラとリンカは、どちらもライブラリを入力として受け入れます。

図 6−1. アーカイバと開発フロー



6.3 アーカイバの起動方法

アーカイバを起動するには、次のように入力します。

ar500 [–]*command* [*option*] *libname* [*filename*$_1$ ... *filename*$_n$]

ar500　アーカイバを起動するコマンドです。

command　アーカイバに対して、ライブラリ・メンバの操作方法を指示します。コマンドの前には、オプションでハイフンを付けることができます。アーカイバを起動するときには、以下のコマンドのいずれかを必ず指定してください。ただし、1回の起動で指定できるのは1つのコマンドだけです。有効なアーカイバ・コマンドは以下のとおりです。

- **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もつメンバがすでにあっても、そのメンバを置換しません。単に新しいメンバをアーカイブの最後に追加するだけです。
- **d**　指定されたメンバをライブラリから削除します。
- **r**　ライブラリ内の指定されたメンバを置換します。ファイル名を指定しない場合、アーカイバは、ライブラリ・メンバを現行のディレクトリ内にある同じ名前のファイルに置換します。ライブラリ内に指定されたファイルが存在しない場合、アーカイバは、そのメンバを置換せずに追加します。
- **t**　ライブラリの内容の一覧を出力します。ファイル名を指定する場合、指定されたファイルのみのリストを出力します。ファイル名を指定しない場合、アーカイバは指定されたすべてのライブラリ・メンバのリストを作成します。
- **x**　指定されたファイルを抽出します。メンバ名を指定しない場合、アーカイバはライブラリのすべてのメンバを抽出します。アーカイバはメンバを 1 つ抽出すると、単にそのメンバを現行のディレクトリにコピーするだけで、そのメンバをライブラリから除去することはありません。

option アーカイバに対して動作方法を指示します。以下のオプションは同じに指定できます。

−q (quiet: 静的な実行) タイトル画面と状況メッセージの出力を抑止します。

−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、および −d についてのみ有効です)。

−v (verbose: 補足) 古いライブラリとその構成メンバから、新しいライブラリを作成するときに、ファイルごとにその説明を表示します。

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

filename ライブラリに関連付けられる個々のメンバ・ファイルの名前を指定します。拡張子がある場合は、拡張子付きの完全なファイル名を指定する必要があります。

1 つのライブラリに同じ名前の複数のメンバを組み込むことは可能です (望ましくはありませんが)。メンバを削除、置換、または抽出しようとし、かつライブラリ内に指定した名前のメンバが複数存在する場合、アーカイバはその名前をもった最初のメンバを削除、置換、または抽出します。

6.4 アーカイバの例

アーカイバの使用例を以下に示します。

- sine.obj、cos.obj、および flt.obj のファイルを格納する function.lib という名前のライブラリを作成する場合は、次のように入力します。

```
ar500 –a function sine.obj cos.obj flt.obj
TMS320C54x Archiver              Version x.xx
Copyright (c) 2001    Texas Instruments Incorporated
   ==>    new archive 'function.lib'
   ==>    building archive 'function.lib'
```

- –t オプションを指定すると、function.lib の内容一覧を出力できます。

```
ar500 –t function
TMS320C54x Archiver              Version x.xx
Copyright (c) 2001    Texas Instruments Incorporated
       FILE NAME     SIZE      DATE
----------------     ----      ------------------------
        sine.obj      248      Mon Nov 19 01:25:44 2001
         cos.obj      248      Mon Nov 19 01:25:44 2001
         flt.obj      248      Mon Nov 19 01:25:44 2001
```

- ライブラリに新しいメンバを追加する場合は、次のように入力します。

```
ar500  –as function atan.obj
TMS320C54x Archiver              Version x.xx
Copyright (c) 2001    Texas Instruments Incorporated
   ==>    symbol defined: 'symbol_name'
   ==>    symbol defined: 'symbol_name'
   ==>    building archive 'function.lib'
```

上記の例では libname の拡張子が指定されていないため、アーカイバは、function.lib という名前のライブラリにファイルを追加します。function.lib が存在しない場合、アーカイバは function.lib を作成します (–s オプションはアーカイバに対して、ライブラリで定義されているグローバル・シンボルのリストを出力するように指示します)。

- ライブラリ・メンバを変更する場合は、変更するメンバを抽出し、編集して、置換します。次の例では、macros.lib という名前のライブラリに、push.asm、pop.asm、および swap.asm というメンバが含まれていると仮定します。

```
ar500 –x macros push.asm
```

アーカイバは、push.asm のコピーを作成し、そのコピーを現行のディレクトリに入れます。ただし、ライブラリから push.asm が除去されることはありません。これで、抽出したファイルを編集できます。ライブラリの push.asm を編集した push.asm コピーと置換するには、次のように入力します。

```
ar500 –r macros push.asm
```

第 7 章

リンカの説明

TMS320C54xリンカは、複数の COFF オブジェクト・ファイルを結合して実行可能モジュールを作成します。COFF セクションの概念は、リンカの動作の基本になっています。第 2 章「共通オブジェクト・ファイル・フォーマットの概要」に、COFF フォーマットの詳細を記述しています。

項目 **ページ**

7.1 リンカの概要 7-2
7.2 リンカと開発フロー 7-3
7.3 リンカの起動方法 7-4
7.4 リンカ・オプション 7-6
7.5 リンカ・コマンド・ファイル 7-21
7.6 オブジェクト・ライブラリ 7-25
7.7 MEMORY 疑似命令 7-27
7.8 SECTIONS 疑似命令 7-32
7.9 セクションの実行 (ランタイム) アドレスの指定方法 7-44
7.10 UNION 文と GROUP 文の使用方法 7-48
7.11 オーバーレイ・ページ 7-53
7.12 デフォルトの割り当てアルゴリズム 7-58
7.13 特別なセクションの型 (DSECT、COPY、および NOLOAD) 7-61
7.14 リンク時のシンボルの割り当て方法 7-62
7.15 ホールの作成方法と埋め込み方法 7-66
7.16 部分 (インクリメンタル) リンク 7-70
7.17 C/C++ コードのリンク 7-72
7.18 リンカの例 7-76

7.1 リンカの概要

TMS320C54x リンカを使用すると、出力セクションを効率良くメモリ・マップに割り当てることによって、システム・メモリを構成できます。リンカは、オブジェクト・ファイルを結合するとき、以下の作業を実行します。

- ❏ セクションをターゲット・システムの構成メモリに割り当てる。
- ❏ シンボルとセクションを再配置して、最終アドレスに割り当てる。
- ❏ 入力ファイル間の未定義の外部参照を解決する。

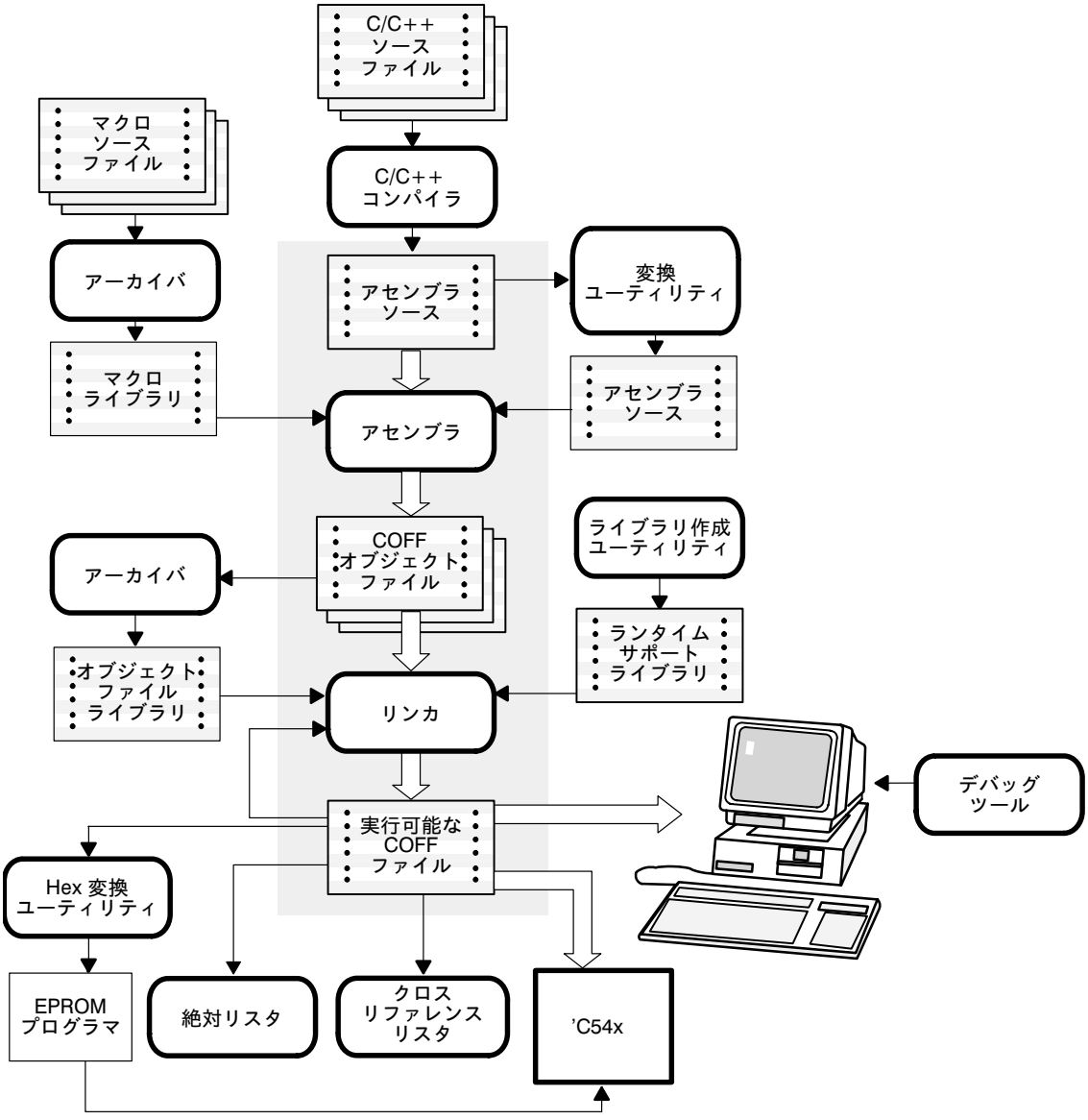
リンカ・コマンド言語は、メモリ構成、出力セクション定義、およびアドレスのバインディングを制御します。この言語は、式の代入と計算をサポートします。設計したメモリ・モデルを定義し、作成することにより、システム・メモリを構成できます。2 つの強力な疑似命令 MEMORY および SECTIONS を使用すると、以下の操作を行うことができます。

- ❏ セクションをメモリの特定の領域に割り当てる。
- ❏ オブジェクト・ファイルのセクションを結合する。
- ❏ リンク時にグローバル・シンボルを定義、または再定義する。

7.2 リンカと開発フロー

図 7−1 は、アセンブリ言語開発プロセスにおけるリンカの役割を示したものです。リンカは、オブジェクト・ファイル、コマンド・ファイル、ライブラリ、部分的にリンクされたファイルなど数種類のファイルを入力として受け入れます。リンカが作成する実行可能な COFF オブジェクト・モジュールは、いくつかの開発ツールの 1 つにダウンロードでき、または TMS320C54x デバイスを使って実行できます。

図 7−1. リンカと開発フロー



7.3 リンカの起動方法

リンカを起動する一般的な構文は次のとおりです。

lnk500 [*–options*] *filename*$_1$*. ... filename*$_n$

lnk500	リンカを起動するコマンドです。
options	コマンド行またはリンカ・コマンド・ファイル内の任意の場所に指定できます (オプションについての詳細は、7-6 ページの 7.4 節「リンカ・オプション」にあります)。
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の拡張子は *.obj* であり、それ以外の拡張子は明示的に指定しなければなりません。リンカは、入力ファイルがオブジェクト・ファイルか、リンカ・コマンドを含む ASCII ファイルかを判断します。デフォルトの出力ファイル名は *a.out* です。

リンカを起動するには、以下の 3 つの方法のいずれかを使用します。

- コマンド行でオプションとファイル名を指定します。この例では、file1.obj と file2.obj の 2 つのファイルをリンクして、link.out という名前の出力モジュールを作成します。

```
lnk500 file1.obj file2.obj –o link.out
```

- **lnk500** コマンドをファイル名とオプションを付けずに入力します。この場合、リンカはそれらに対するプロンプトを表示します。

```
Command files :
Object files [.obj] :
Output file [a.out] :
Options :
```

  - *command files* には、1 つまたは複数のコマンド・ファイル名を入力します。
  - *object files* には、1 つまたは複数のオブジェクト・ファイル名を入力します。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。
  - *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、他のプロンプトに対して入力された –o オプションを無効にします。–o オプションを使用せず、このプロンプトにも応答しない場合、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・ファイルに追加オプションを入れることもできます。入力するには、コマンド行に入力する場合と同様にハイフンを使用してください。

❑ 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起動できます。コマンド・ファイル名を入力ファイルとして指定してください。

```
lnk500 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、次のように入力できます。

```
lnk500 –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。

7.4 リンカ・オプション

リンカ・オプションは、リンク操作を制御するオプションで、コマンド行かコマンド・ファイルに置くことができます。リンカ・オプションの前には必ずハイフン (−) を付けなければなりません。−l (小文字の L) と −i オプションを除き、オプションはどのような順序で指定してもかまいません。引数がある場合には、引数とオプションの間に空白を入れて離すこともできます。リンカ・オプションの要約を以下に示します。

−a	絶対的な実行可能モジュールを作成します。このオプションはデフォルトです。−a も −r も指定されていない場合、リンカは −a が指定されているものとして動作します。
−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	TMS320C54x C/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	TMS320C54x C/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め込み値を設定します。*fill_value* は 16 ビットの定数です。
−g *global_symbol*	*global_symbol* をグローバルのままにします (−h を無効にします)。
−h	すべてのグローバル・シンボルを静的にします。
−help **−?**	使用できるすべてのリンカ・コマンド行オプションのリストを表示します。
−heap *size*	ヒープ・サイズ (C の動的メモリ割り当て用) 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。デフォルトのサイズは 1K ワードです。
−i *dir*	ライブラリ検索アルゴリズムを変更し、デフォルトの場所を検索する前に *dir* の中を検索するようにします。このオプションは、−l オプションより前に指定しなければなりません。ディレクトリまたはファイル名は、オペレーティング・システムの規則に従っていなければなりません。
−j	条件付きリンクを抑制します。
−k	入力セクション内の位置合わせフラグを無視します。
−l *filename*	リンカの入力としてアーカイブ・ライブラリ・ファイルの名前を指定します。つまり、*filename* にはアーカイブ・ライブラリ名が入ります。このオプションは、−i オプションより後に指定しなければなりません。ディレクトリまたはファイル名は、オペレーティング・システムの規則に従っていなければなりません。

–m *filename*	ホール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。ディレクトリまたはファイル名は、オペレーティング・システムの規則に従っていなければなりません。
–q	静的な実行 (見出しの抑止) を要求します。
–r	再配置可能な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。デフォルトのサイズは 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。
–v*n*	COFF の出力フォーマットを指定します。ただし、*n* は 0、1、または 2 です。デフォルトのフォーマットは COFF2 です。
–w	未定義の出力セクションが作成されたときに、メッセージを表示します。
–x	ライブラリの再読み取りを強制実行します。後方参照を解決します。

7.4.1 再配置機能 (−a オプションと −r オプション)

リンカは再配置を実行します。再配置は、あるシンボルのアドレスが変わったときに、そのシンボルに対するすべての参照を調整するプロセスです。リンカは 2 つのオプション (−a と −r) をサポートしており、それを使用して絶対出力モジュールや再配置可能な出力モジュールを作成できます。−a オプションも −r オプションも指定されなかった場合、リンカはデフォルトにより、−a オプションが指定されたものとして動作します。

❑ **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 オプションを使用すると、リンカは実行可能な絶対出力モジュールを作成します。絶対ファイルには、再配置情報は何も含まれません。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殊シンボル (これらのシンボルについては、7-65 ページの 7.14.4 項「リンカで定義されるシンボル」に説明があります)。
- プログラムのエントリ・ポイントなどの情報を記述するオプション・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 がリンクされて、a.out という絶対出力モジュールが作成されます。

```
lnk500 -a file1.obj file2.obj
```

注: −a オプションと −r オプション

−a と −r のどちらのオプションも使用しなかった場合、リンカは −a が指定されたものとして動作します。

❑ **再配置可能な出力モジュールの作成 (–r オプション)**

–a オプションを指定せずに –r オプションを使用すると、リンカは出力モジュールに再配置エントリを保持します。出力モジュールが (ロード時に) 再配置されるか、(他のリンカの実行によって) 再リンクされる場合は、–r オプションを指定して再配置エントリを保持します。

–a オプションを指定せずに –r オプションを使用すると、リンカは実行不可能なファイルを作成します。実行できないファイルには、特別なリンカ・シンボルもオプション・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、これが原因で出力モジュールが作成されなくなることはありません。

次の例では、file1.obj と file2.obj がリンクされて、a.out という再配置可能な出力モジュールが作成されます。

```
lnk500 -r file1.obj file2.obj
```

出力ファイル a.out は、他のオブジェクト・ファイルと再リンクしたり、またはロード時に再配置したりできます (他のファイルと再リンクされるファイルをリンクすることを、部分リンクといいます)。詳細は、7-76 ページの 7.18 節「リンカの例」を参照してください。

❑ **実行可能で再配置可能な出力モジュールの作成 (–ar)**

–a オプションと –r オプションの両方を指定してリンカを起動すると、リンカは実行可能で再配置可能なオブジェクト・モジュールを作成します。この出力ファイルには、特別なリンカ・シンボル、オプション・ヘッダ、および解決されたすべてのシンボル参照が含まれています。ただし、再配置情報は保持されます。

次の例では、file1.obj と file2.obj がリンクされて、xr.out という名前の実行可能で再配置可能な出力モジュールが作成されます。

```
lnk500 -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 (lnk500 –ar) ことも、区切って入力する (lnk500 –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)。絶対的なファイルを再リンクすることは、各入力ファイルに再配置する必要がある情報が含まれていないときのみ可能です (つまり、どのファイルにも未解決の参照がなく、リンカがそのファイルを作成したときにバインドされた同じ仮想アドレスにバインドされるときのみです)。

7.4.2 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、デフォルトでは、シンボリック・デバッグ情報に重複エントリができないようにします。このような重複情報が作成されるのは、通常は、C プログラムをデバッグ用にコンパイルした場合です。この例を次に示します。

```
-[ header.h ]-
typedef struct
{
   <define some structure members>
} XYZ;

-[ f1.c ]-
#include ”header.h”
...

-[ f2.c ]-
#include ”header.h”
...
```

この例のファイルをデバッグ用にコンパイルすると、f1.obj と f2.obj の両方にタイプ XYZ を記述するシンボリック・デバッグ・エントリができます。最終出力ファイルで必要なのは、これらのエントリのセットのうち 1 つだけです。リンカは、重複エントリを自動的に削除します。

−b オプションはリンカに対して、このような重複エントリを保存するように指定する場合に使用されます。−b オプションを使用すると、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。

7.4.3 C 言語オプション (−c オプションと −cr オプション)

−c および −cr オプションを指定すると、リンカは C/C++ コンパイラに必要なリンク規則を使用するようになります。

- ❏ −c オプションはリンカに対して、ROM 自動初期化モデルを指定するように指示します。
- ❏ −cr オプションはリンカに対して、RAM 自動初期化モデルを指定するように指示します。

C/C++ コードのリンク方法の詳細は、7-72 ページの 7.17 節「C/C++ コードのリンク」、および 7-75 ページの 7.17.5 項「−c リンカ・オプションと −cr リンカ・オプション」を参照してください。

7.4.4 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・カウンタはプログラムの始まりを指示します。

リンカは、エントリ・ポイントに 4 種類の値を割り当てることができます。以下に示した 4 つの値について、リンカはこの順に使おうとします。最初の 3 つの値のうちのいずれかを使用する場合、その値はシンボル・テーブル内の外部シンボルでなければなりません。

- −e オプションで指定された値。構文は以下のとおりです。

  −e *global_symbol*

  global_symbol はエントリ・ポイントを定義し、入力ファイル内で外部シンボルとして定義されなければなりません。

- シンボル _c_int00 の値 (指定されている場合)。C/C++ コンパイラによって生成されたコードをリンクしている場合、_c_int00 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 (使用されている場合)
- ゼロ (デフォルト値)

この例では、file1.obj と file2.obj をリンクします。シンボル begin はエントリ・ポイントです。begin は、file1 か file2 で外部シンボルとして定義されている必要があります。

```
lnk500 -e begin file1.obj file2.obj
```

7.4.5 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形成されたホールを埋めます。または、初期化されないセクションと初期化されたセクションを結合するときに、初期化されないセクションを初期化します。この機能を使用すると、ソース・ファイルを再アセンブルせずに、リンク時にメモリ領域を初期化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 を使用します。

この例では、ホールは 16 進値 ABCD で埋められます。

```
lnk500 -f 0ABCDh file1.obj file2.obj
```

7.4.6 シンボルのグローバル化 (−g *global_symbol* オプション)

−h オプションは、すべてのグローバル・シンボルを静的にします。グローバルにして残したいシンボルがあり、−h オプションを使用する場合は、−g オプションを使用してそのシンボルをグローバルに宣言できます。−g オプションは、指定したシンボルについて、−h オプションによる効果を無効にします。−g オプションの構文は次のとおりです。

−g *global_symbol*

7.4.7 すべてのグローバル・シンボルの静的化 (−h オプション)

−h オプションは、.global アセンブラ疑似命令で定義されたすべてのグローバル・シンボルを静的にします。静的シンボルは、外部からリンクしたモジュールには見えません。グローバル・シンボルを静的にすると、グローバル・シンボルは実質的に隠されます。これにより、同じ名前の (別のファイル内にある) 外部シンボルを固有のものとして扱うことができます。

−h オプションは、すべての .global アセンブラ疑似命令を実質的に無効にします。すべてのシンボルは、そのシンボルが定義されているモジュールのローカル・シンボルになるので、外部参照が不可能になります。たとえば、b1.obj、b2.obj、および b3.obj は互いに関連していて、グローバル変数 GLOB を参照しているとします。また、d1.obj、d2.obj、および d3.obj は互いに関連していて、別のグローバル変数 GLOB を参照しているとします。この場合、−h オプションと部分リンクを使用すれば、関連ファイルを競合なしにリンクできます。

```
lnk500 -h -r b1.obj b2.obj b3.obj -o bpart.out
lnk500 -h -r d1.obj d2.obj d3.obj -o dpart.out
```

−h オプションにより、bpart.out と dpart.out はグローバル・シンボルをもたないこと、つまり GLOB は 2 つの別々の GLOB として取り扱われることが保証されます。−r オプションを指定すると、bpart.out と dpart.out はそれぞれの再配置エントリを保持できます。こうすると、この 2 つの部分的にリンクされたファイルは、次のコマンドを使ってリンクしても安全です。

```
lnk500 bpart.out dpart.out -o system.out
```

7.4.8 ヒープ・サイズの定義 (−heap *constant* オプション)

C/C++ コンパイラは、.sysmem と呼ばれる初期化されないセクションを、malloc() の使用する C ランタイム・メモリ・プールに使用します。このメモリ・プールのサイズは、リンク時に −heap オプションを使用して設定できます。このオプションの直後に、定数でサイズ (ワード) を指定します。

```
lnk500 -heap 0x0400 /* defines a heap size */
```

リンカは、入力ファイルのいずれかに .sysmem セクションがある場合のみ、.sysmem セクションを作成します。

リンカはまた、グローバル・シンボル __SYSMEM_SIZE を作成し、このシンボルにヒープ・サイズを割り当てます。デフォルトのサイズは 1K ワードです。

C コードのリンク方法の詳細は、7-72 ページの 7.17 節「C/C++ コードのリンク」を参照してください。

7.4.9 ライブラリ検索アルゴリズムの変更 (–l オプション、–i オプション、および C54X_C_DIR/C_DIR 環境変数)

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、file1.obj で参照されているシンボルが定義されているものとします。このファイルのリンク方法を次に示します。

```
lnk500 file1.obj object.lib
```

現行のディレクトリにないライブラリを使用する場合は、–l (小文字の L) リンカ・オプションを使用します。このオプションの構文は次のとおりです。

–l *filename*

filename はアーカイブ・ライブラリの名前です。–l とファイル名の間の空白は、入れても入れなくてもかまいません。

–i リンカ・オプションまたは C_DIR あるいは C54X_C_DIR 環境変数を使用して、リンカのディレクトリ検索アルゴリズムを補強させることができます。リンカは、次の順序でオブジェクト・ライブラリを検索します。

1) –i リンカ・オプションを使用して指定されたディレクトリを検索します。
2) C_DIR および C54X_C_DIR 環境変数を使用して指定されたディレクトリを検索します。
3) C_DIR および C54X_C_DIR が設定されていない場合は、アセンブラの C54X_A_DIR および A_DIR 環境変数を使用して指定されたディレクトリを検索します。
4) 現行のディレクトリを検索します。

7.4.9.1 代替ライブラリ・ディレクトリの名前の指定 (–i オプション)

–i オプションを使用して、オブジェクト・ライブラリが入っている代替ディレクトリの名前を指定します。このオプションの構文は次のとおりです。

–i *dir*

dir は、オブジェクト・ライブラリが入っているディレクトリの名前を指定します。–i とディレクトリ名の間の空白は、入れても入れなくてもかまいません。

リンカは、–l オプションによ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は、–i で指定されたディレクトリ内を最初に検索します。各 –i オプションは、1 つのディレクトリしか指定できませんが、1 回の起動につき複数の –i オプションを指定できます。–i オプションを使用して代替ディレクトリを指定する場合は、コマンド行またはコマンド・ファイルで –l オプションの前に指定する必要があります。

たとえば、r.lib と lib2.lib という 2 つのアーカイブ・ライブラリがあるとします。次の表は、r.lib と lib2.lib があるディレクトリ、環境変数の設定方法、およびリンク時における 2 つのライブラリの使用方法を示しています。使用しているオペレーティング・システムに従って、次のように選択してください。

O.S.	パス名	起動コマンド
DOS	\ldと\ld2	`lnk500 f1.obj f2.obj –i\ld –i\ld2 –lr.lib –llib2.lib`
UNIX	/ldと/ld2	`lnk500 f1.obj f2.obj –i/ld –i/ld2 –lr.lib –llib2.lib`

7.4.9.2 代替ライブラリ・ディレクトリの名前の指定 (C_DIR 環境変数)

環境変数は、ユーザ定義によって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 および C54X_C_DIR という環境変数を使用して、オブジェクト・ライブラリが入った代替ディレクトリの名前を指定します。次のコマンドを使用して環境変数を割り当てます。

O.S.	入力
DOS	`set C_DIR= pathname;another pathname ...`
UNIX	`setenv C_DIR "pathname;another pathname ..."`

pathnames は、オブジェクト・ライブラリが入っているディレクトリです。コマンド行またはコマンド・ファイルで –l オプションを使用して、リンカに検索するライブラリ名を指示します。

次の例では、r.lib と lib2.lib という 2 つのアーカイブ・ライブラリが、ld と ld2 というディレクトリに入っているとします。次の表は、r.lib と lib2.lib が入っているディレクトリ、環境変数の設定方法、およびリンク時における 2 つのライブラリの使用方法を示しています。使用しているオペレーティング・システムに従って、次のように選択してください。

O.S.	パス名	起動コマンド
DOS	\ld と \ld2	`set  C_DIR=\ld;\ld2` `lnk500 f1.obj f2.obj –l r.lib –l lib2.lib`
UNIX	/ld と /ld2	`setenv C_DIR "/ld ;/ld2"` `lnk500 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。

O.S.	入力
DOS	`set C_DIR=`
UNIX	`unsetenv C_DIR`

アセンブラは、A_DIR および C54X_A_DIR という環境変数を使用して .copy/.include ファイルまたはマクロ・ライブラリを含んだ代替ディレクトリの名前を指定します。C_DIR が設定されていない場合、リンカは、A_DIR および C54X_A_DIR で指定されたディレクトリ内のオブジェクト・ライブラリを検索します。オブジェクト・ライブラリの詳細は、7-25 ページの 7.6 節「オブジェクト・ライブラリ」にあります。

7.4.10 条件付きリンクの無効化 (–j オプション)

–j オプションを指定すると、アセンブラの .clink 疑似命令で設定された条件付きリンクは無効になります。デフォルトでは、すべてのセクションは無条件にリンクされます。

7.4.11 位置合わせフラグの無視 (–k オプション)

–k オプションを指定すると、SECTIONS 疑似命令の位置合わせ指定が強制的に無視されます。SECTIONS 疑似命令の詳細は、7.8 節「SECTIONS 疑似命令」を参照してください。

7.4.12 マップ・ファイルの作成 (–m *filename* オプション)

–m オプションを指定すると、リンカ・マップ・リストが作成され、そのマップは *filename* に入れられます。–m オプションの構文は次のとおりです。

–m *filename*

リンカ・マップには、以下の事項が記述されます。

- ❑ メモリ構成
- ❑ 入出力セクションの割り当て
- ❑ 外部シンボルの再配置後のアドレス

マップ・ファイルには、出力モジュール名とエントリ・ポイントが入っています。また、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 がリンクされて、file.map という名前のマップ・ファイルが作成されます。

```
lnk500 file1.obj file2.obj -m file.map
```

マップ・ファイルの例は、7-78 ページの例 7−15 に示しています。

7.4.13 出力モジュールの名前の指定 (−o *filename* オプション)

エラーが検出されないと、リンカは出力モジュールを作成します。出力モジュールのファイル名を指定しない場合、リンカはデフォルト名の a.out を使用します。出力モジュールを別のファイルに書き込む場合は、−o オプションを指定します。−o オプションの構文は次のとおりです。

−o *filename*

filename は、新しい出力モジュールの名前です。

この例では、file1.obj と file2.obj がリンクされて、run.out という名前の出力モジュールが作成されます。

```
lnk500 -o run.out file1.obj file2.obj
```

7.4.14 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、リンカでは見出し情報が出力されません。このオプションは、バッチ操作の際に使用すると便利です。

7.4.15 シンボル情報の除去 (−s オプション)

−s オプションを指定すると、シンボル・テーブル情報と行番号エントリが省略され、出力モジュールのサイズを小さくすることができます。実働アプリケーションでは、出力モジュールを最も小さくしたいときに −s オプションを使用すると有効です。

この例では、file1.obj と file2.obj をリンクして、行番号とシンボル・テーブル情報が除去された nosym.out という名前の出力モジュールを作成します。

```
lnk500 -o nosym.out -s file1.obj file2.obj
```

−s オプションを指定することによって、後でシンボリック・デバッガの使用が制限されたり、ファイルの再リンクができなくなったりすることがあります。

7.4.16 スタック・サイズの定義 (−stack *constant* オプション)

TMS320C54x C/C++ コンパイラは、初期化されない .stack セクションを使用してランタイム・スタックの空間を割り当てます。リンク時に、−stack オプションを指定して .stack セクションのサイズを設定できます。このオプションの直後に、定数でサイズ (ワード) を指定します。

```
lnk500 -stack 0x1000 /* defines a stack size */
```

入力セクションに異なるスタック・サイズを指定した場合、入力セクションのスタック・サイズは無視されます。入力セクションで定義されているシンボルは、すべて有効なままです。スタック・サイズのみが変わります。

リンカは、.stack セクションを定義するときにグローバル・シンボル __STACK_SIZE を定義し、このシンボルにセクションのサイズを割り当てます。デフォルトのスタック・サイズは 1K ワードです。

7.4.17 未解決のシンボルの導入 (−u *symbol* オプション)

−u オプションを指定すると、未解決のシンボルをリンカのシンボル・テーブルに入れることができます。これにより、リンカは、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取り込むように強制されます。リンカに参照のないシンボルを定義するメンバをリンクさせるためには、リンカに対して −u オプションを指定しなければなりません。

たとえば、rts.lib という名前のライブラリに symtab というシンボルを定義したメンバがあり、リンクするどのオブジェクト・ファイルも symtab を参照しないとします。しかし、出力モジュールを再リンクする必要があり、このリンクに symtab を定義したライブラリ・メンバを取り込もうとしているとします。以下に示すように −u オプションを指定すると、リンカは、rts.lib で symtab を定義したメンバを検索し、そのメンバをリンクするように強制されます。

```
lnk500 -u symtab file1.obj file2.obj rts.lib
```

−u オプションを指定しない場合、file1.obj または file2.obj のどちらからもこのメンバを明示的に参照しないので、このメンバは取り込まれません。

7.4.18 COFF フォーマットの指定 (–v オプション)

–v オプションは、リンカの作成する COFF オブジェクト・ファイルのフォーマットを指定します。COFF オブジェクト・ファイルは、リンカの出力です。このフォーマットは、オブジェクト・ファイル内に情報を配置する方法を指定します。

リンカは、COFF0、COFF1、および COFF2 の各フォーマットを読み書きできます。デフォルトでは、リンカは COFF2 ファイルを作成します。異なる出力フォーマットを作成するには、–v オプションを使用します。ただし、*n* は、COFF0 の場合には 0、COFF1 の場合には 1 です。

COFF の詳細は、第 2 章「共通オブジェクト・ファイル・フォーマットの概要」、および付録 A「共通オブジェクト・ファイル・フォーマット」に説明があります。

7.4.19 出力セクション情報のメッセージの表示 (–w オプション)

–w オプションを指定すると、メモリ・セクションの作成に関連した追加メッセージが表示されます。追加メッセージは、以下の環境で表示されます。

- リンカ・コマンド・ファイルの中で、入力セクションをどのように結合して出力セクションにするかを記述した SECTIONS 疑似命令を設定できます。しかし、リンカは、SECTIONS 疑似命令の中で、対応する出力セクションが定義されていない入力セクションを 1 つ以上検出すると、同じ名前をもつ複数の入力セクションを、その名前が付いた 1 つの出力セクションに統合します。デフォルトでは、リンカはこの統合が発生したことを知らせるメッセージを表示しません。

  この状況が発生し、かつ –w オプションを使用している場合、リンカは、新しい出力セクションを作成したときにメッセージを表示します。

- –heap オプションと –stack オプションを使用しなかった場合、リンカは自動的に、それぞれ .sysmem セクションと .stack セクションを作成します。各セクションのデフォルト・サイズは 0x400 ワードです。これらのセクションのどちらか、または両方に割り当てる十分なメモリが存在しない場合も考えられます。その場合、リンカは、セクションの割り当てができなかったことを知らせるエラー・メッセージを発行します。

  –w オプションを使用した場合、リンカはさらに詳しい別のメッセージを表示します。そのメッセージには、ユーザ自身が .system または .stack セクションを割り当てるための疑似命令の名前が含まれています。

SECTIONS 疑似命令の詳細は、7-32 ページの 7.8 節「SECTIONS 疑似命令」を参照してください。リンカのデフォルト動作の詳細は、7-58 ページの 7.12 節「デフォルトの割り当てアルゴリズム」を参照してください。

7.4.20 ライブラリの強制読み取り (–x オプション)

リンカは通常、入力ファイル (アーカイブ・ライブラリを含む) を、コマンド行またはコマンド・ファイルで検出したときに 1 回だけ読み取ります。アーカイブが読み取られると、未定義のシンボルに対する参照を解決するすべてのメンバがリンクに入れられます。その後、入力ファイルがすでに読み取ったアーカイブ・ライブラリで定義されているシンボルを参照すると、その参照は解決されません。

–x オプションを使用すると、リンカに対してすべてのライブラリを再読み取りするように強制できます。リンカは、解決されていない参照がなくなるまでライブラリを再度読み取ります。–x オプションを使用すると、リンクの速度が低下します。したがって、使用するのは必要なときだけにしてください。たとえば、a.lib には b.lib で定義されているシンボルに対する参照が含まれていて、b.lib には a.lib で定義されているシンボルへの参照が含まれている場合、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。

```
lnk500 –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。

```
lnk500 -x –la.lib –lb.lib
```

7.5 リンカ・コマンド・ファイル

リンカ・コマンド・ファイルを使用すると、リンク制御情報をファイルに置くことができます。リンカを同じ情報で何度も起動するときに、この機能を使用すると便利です。また、リンカ・コマンド・ファイルを使用すると、MEMORY および SECTIONS 疑似命令を使用してアプリケーションをカスタマイズできるので便利です。この 2 つの疑似命令は、コマンド・ファイルでのみ使用でき、コマンド行では使用できません。

リンカ・コマンド・ファイルは、以下の要素を 1 つ以上含む ASCII ファイルです。

- ❏ 入力ファイル名。オブジェクト・ファイル、アーカイブ・ライブラリ、または他のコマンド・ファイルを指定します。
- ❏ リンカ・オプション。このオプションは、コマンド行で使用するのと同様にコマンド・ファイルで使用できます。
- ❏ MEMORY と SECTIONS 疑似命令。MEMORY 疑似命令を使用すると、ターゲット・メモリの構成を定義できます。SECTIONS 疑似命令を使用すると、セクションの構築と割り当ての方法が制御できます。
- ❏ 代入文。グローバル・シンボルを定義し、それに値を割り当てます。

コマンド・ファイルでリンカを起動するには、コマンド **lnk500** を入力し、その後にコマンド・ファイル名を入力します。

lnk500 *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、検出したファイルをオブジェクト・ファイルとして認識すると、そのファイルをリンクします。検出したファイルがオブジェクト・ファイルでない場合、リンカは、そのファイルをコマンド・ファイルとみなし、その中のコマンドを読み取って処理します。コマンド・ファイル名は、使用しているシステムに関係なく大文字と小文字を区別します。

例 7−1 は、link.cmd という名前のリンカ・コマンド・ファイルのサンプルを示しています (2-15 ページの 2.4.2 項「セクションのメモリ・マップへの配置」に、リンカ・コマンド・ファイルの別の例があります)。

例 7−1.　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 7−1 はファイル名とオプションのみを示しています。コマンド・ファイルには、/* と */ で囲むことによりコメントを指定できます。このコマンド・ファイルでリンカを起動するには、次のように入力します。

```
lnk500 link.cmd
```

コマンド・ファイルを使用するときには、コマンド行に別のパラメータを入れることができます。

```
lnk500 -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します。したがって、a.obj と b.obj は c.obj と d.obj よりも先に出力モジュールにリンクされます。

複数のコマンド・ファイルを指定できます。たとえば、ファイル名が入っている names.lst という名前のファイルと、リンカ疑似命令が入っている dir.cmd という名前の別のファイルがある場合は、次のように入力します。

```
lnk500 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。これは、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。

注　スペースまたはハイフンが付いたファイル名とオプション・パラメータ

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

例 7−2 は、リンカ疑似命令が入ったコマンド・ファイルの例を示しています (リンカ疑似命令のフォーマットについては、この後の節で説明します)。

例 7−2.　リンカ疑似命令を含むコマンド・ファイル

```
a.obj b.obj c.obj          /* Input filenames       */
–o prog.out –m prog.map    /* Options               */

MEMORY                    /* MEMORY directive      */
{
  RAM:  origin = 100h      length = 0100h
  ROM:  origin = 01000h    length = 0100h
}

SECTIONS                  /* SECTIONS directive    */
{
  .text:  > ROM
  .data:  > RAM
  .bss:   > RAM
}
```

7.5.1 リンカ・コマンド・ファイル内で予約済みの名前

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。これらの名前は、コマンド・ファイル内でシンボル名やセクション名として使用することはできません。

align	GROUP	origin
ALIGN	l (小文字の L)	ORIGIN
attr	len	page
ATTR	length	PAGE
block	LENGTH	range
BLOCK	load	run
COPY	LOAD	RUN
DSECT	MEMORY	SECTIONS
f	NOLOAD	spare
fill	o	type
FILL	org	TYPE
group		UNION

7.5.2 コマンド・ファイルの定数

定数は、2 つの構文形式のどちらかを使用して指定できます。1 つはアセンブラで使用される 10 進、8 進、16 進のいずれかの定数を指定するために使用する形式 (3-15 ページの 3.6 節「定数」を参照) で、もう 1 つは、C の構文で整数定数に使用する形式です。

例:

	10 進数	8 進数	16 進数
アセンブラ・フォーマット：	32	40q	20h
C フォーマット:	32	040	0x20

7.6 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はグループとしてライブラリに集められます。オブジェクト・ライブラリをリンカ入力として指定すると、リンカは、ライブラリのメンバで、既存の未解決のシンボルに対する参照を定義しているメンバをすべて組み込みます。アーカイバを使用して、ライブラリを作成し、保守することができます。第 6 章「アーカイバの説明」に、アーカイバについての詳細が記載されています。

オブジェクト・ライブラリを使用すると、リンク時間を短縮し、実行可能なモジュールのサイズを小さく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、その同じ関数がアーカイブ・ライブラリに入っている場合は、参照されるときだけ組み込まれます。

ライブラリを指定する順序には注意を払う必要があります。これは、リンカは、ライブラリの検索時に未定義のシンボルを解決するメンバのみを組み込むからです。同じライブラリは、必要に応じて何度でも指定できます。ライブラリは組み込まれるたびに検索されます。この代わりに、−x オプションを指定することもできます。ライブラリには、このライブラリで定義されているすべての外部シンボルを格納したテーブルがあります。リンカは、このライブラリがこれ以上参照を解決するのに役に立たないと判断するまで、このテーブルを検索します。

次の例では、いくつかのオブジェクト・ファイルがライブラリとリンクされます。次の条件を前提とします。

- ❑ 入力ファイル f1.obj と f2.obj は、両方とも clrscr という名前の外部関数を参照している。
- ❑ 入力ファイル f1.obj は、シンボル origin を参照している。
- ❑ 入力ファイル f2.obj は、シンボル fillclr を参照している。
- ❑ ライブラリ libc.lib のメンバ 0 は、origin の定義を含んでいる。
- ❑ ライブラリ liba.lib のメンバ 3 は、fillclr の定義を含んでいる。
- ❑ 両方のライブラリのメンバ 1 は、clrscr を定義している。

たとえば、次のように入力すると、参照は以下に示すように解決されます。

```
lnk500 f1.obj liba.lib f2.obj libc.lib
```

- ❑ liba.lib のメンバ 1 は、clrscr に対する両方の参照を満たします。これは、このライブラリの検索が行われ、clrscr は f2.obj が参照する前に定義されるからです。
- ❑ libc.lib のメンバ 0 は、origin に対する参照を満たします。
- ❑ liba.lib のメンバ 3 は、fillclr に対する参照を満たします。

ただし、次のように入力した場合、clrscr へのすべての参照は、libc.lib のメンバ 1 によって満たされます。

```
lnk500 f1.obj f2.obj libc.lib liba.lib
```

リンクされたどのファイルもライブラリで定義されたシンボルを参照していない場合は、–u オプションを指定することによって、リンカに強制的にライブラリ・メンバを組み込ませることができます。リンカのグローバル・シンボル・テーブルに未定義シンボル rout1 を作成する例を、次に示します。

```
lnk500 –u rout1 libc.lib
```

rout1 を定義しているメンバが libc.lib にある場合は、リンカはそのメンバを組み込みます。

個々のライブラリ・メンバの割り当てを制御することはできません。メンバの割り当ては、SECTIONS 疑似命令のデフォルトの割り当てアルゴリズムに従って行われます。

7-13 ページの 7.4.9 項「ライブラリ検索アルゴリズムの変更 (–l オプション、–i オプション、および C54X_C_DIR/C_DIR 環境変数)」に、オブジェクト・ライブラリが入っているディレクトリを指定する方法が説明されています。

7.7 MEMORY 疑似命令

リンカは、出力セクションをメモリのどこに割り当てるべきかを決定します。この作業を実行するには、リンカにターゲット・メモリのモデルを指定する必要があります。MEMORY 疑似命令を使用すると、ターゲット・メモリのモデルを指定することができ、使用しているシステムで保持するメモリの種類と、メモリが占めるアドレス範囲を定義できます。リンカは、出力セクションを割り当てるときにモデルを保持し、そのモデルを使用して、どのメモリ・ロケーションにオブジェクト・コードを配置するかを決定します。

TMS320C54x システムのメモリ構成は、アプリケーションによって異なります。MEMORY 疑似命令を使用すると、さまざまな構成を指定できます。MEMORY 疑似命令を使用してメモリ・モデルを指定した後で、SECTIONS 疑似命令を使用して、定義されたメモリに出力セクションを割り当てることができます。

リンカがセクションを処理する方法の詳細は、2-13 ページの 2.4 節「リンカによるセクションの処理」を参照してください。セクションの再配置については、2-16 ページの 2.5 節「再配置」を参照してください。

7.7.1 デフォルトのメモリ・モデル

アセンブラを使用すると、コードを TMS320C54x デバイス用にアセンブルできます。アセンブラは、このデバイスを識別するフィールドを出力ファイルのヘッダに挿入します。リンカは、この情報をオブジェクト・ファイルのヘッダから読み取ります。MEMORY 疑似命令を指定しない場合、リンカは名前を指定されたデバイスに固有のデフォルトのメモリ・モデルを使用します。デフォルト・メモリ・モデルの詳細は、7-58 ページの 7.12.1 項「割り当てアルゴリズム」を参照してください。

7.7.2 MEMORY 疑似命令の構文

MEMORY 疑似命令は、ターゲット・システム内に物理的に存在し、プログラムが使用できるメモリの範囲を特定します。それぞれのメモリ範囲には、名前、開始アドレス、および長さがあります。

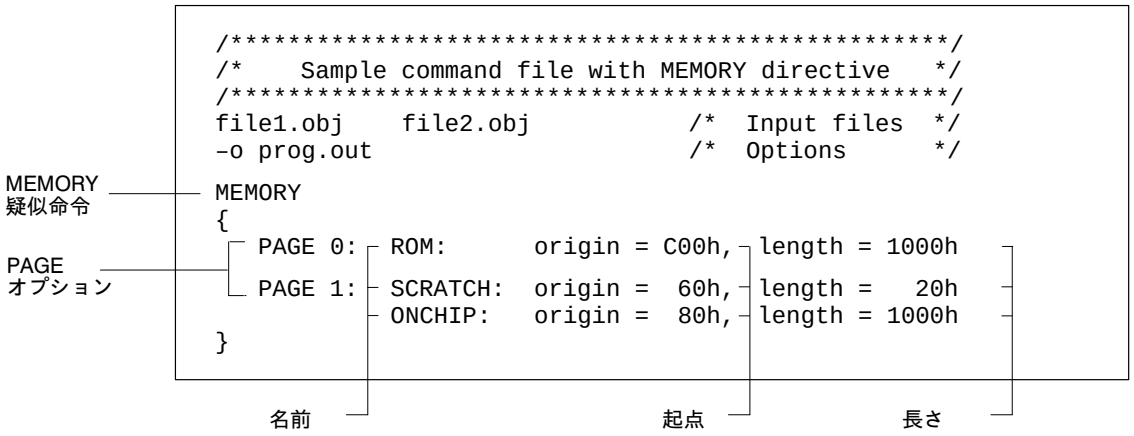
'C54x デバイスには、同じアドレス範囲を占める別々のメモリ空間があります。デフォルト・モデルでは、1 つの空間がプログラム専用となり、別の空間がデータ専用となります (別々のアドレス空間の数は、使用しているチップのカスタマイズされた構成によって異なります。詳細は、TMS320C54x リファレンス・セット マニュアルを参照してください)。

リンカを使用すると、MEMORY 疑似命令の PAGE オプションを使用して、これらのアドレス空間を別々に構成できます。デフォルト・モデルでは、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、これらの 2 つの PAGE を完全に別個のメモリ空間として取り扱います。'C54x は、最大 255 の PAGE をサポートしますが、使用できる数は選択した構成によって異なります。

MEMORY 疑似命令を使用するときは、オブジェクト・コード用に使用できるすべてのメモリ範囲を必ず特定してください。MEMORY 疑似命令によって定義されたメモリは構成メモリであり、MEMORY 疑似命令で明示的に定義しなかったメモリは未構成メモリです。リンカは、未構成メモリにはプログラムを配置しません。存在しないメモリ空間を表すには、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には、MEMORY (大文字) という語の後に、メモリ範囲指定リストを中括弧に入れて指定します。例 7−3 に示した MEMORY 疑似命令は、プログラム・メモリのアドレス C00h 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、およびデータ・メモリのアドレス 80h に 4K ワードをもつシステムを定義しています。

例 7−3. MEMORY 疑似命令



MEMORY 疑似命令の一般的な構文は次のとおりです。

```
MEMORY
{
   PAGE 0 :  name 1 [(attr)]  :  origin =  constant  ,  length =  constant [, fill = constant] ;
   PAGE n :  name n [(attr)]  :  origin =  constant  ,  length =  constant [, fill = constant] ;
}
```

PAGE メモリ空間を特定します。指定した構成によって異なりますが、最大 255 ページまで指定できます。通常は、PAGE 0 はプログラム・メモリを、PAGE 1 はデータ・メモリを指定します。PAGE オプションを指定しない場合、リンカは PAGE 0 が指定されているものとして動作します。それぞれの PAGE は、完全に独立したアドレス空間を表します。PAGE 0 の構成メモリは、PAGE 1 の構成メモリと重ねることができます。

name メモリ範囲に名前を付けます。メモリ名には 1 ～ 64 文字まで指定できます。使用できる文字は、A ～ Z、a ～ z、$、.、および _ です。メモリ範囲に付けた名前は、リンカにとっては特別な意味がありません。単にメモリ範囲を特定するための記号にすぎません。メモリ範囲名はリンカが内部的に使用するもので、出力ファイルやシンボル・テーブルには保持されません。別のページであれば、メモリ範囲に同じ名前を付けることができます。ただし、同じページではどのメモリ範囲にも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はできません。

attr 名前を付けた範囲に関連する 1 ～ 4 個の属性を指定します。属性の使用は任意ですが、使用するときは括弧に入れなければなりません。属性を使用すると、出力セクションの割り当てを特定のメモリ範囲に制限できます。属性を全く使用しない場合は、すべての出力セクションはどのメモリ範囲にでも自由に割り当てることができます。属性が指定されていないメモリは (デフォルトのモデルにあるものを含めて)、4 つの属性をすべてもちます。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ることができ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。値はワード単位で指定する 16 ビットの定数です。10 進数、8 進数、16 進数のいずれも使用できます。

length メモリ範囲の長さを指定します。*length*、*len*、または *l* を入力します。値はワード単位で指定する 16 ビットの定数です。10 進数、8 進数、16 進数のいずれも使用できます。

fill メモリ範囲の埋め込み値を指定します。*fill* または *f* と入力します。埋め込み値の指定は任意です。値は 2 バイトの整数定数で、10 進数、8 進数、16 進数のいずれも使用できます。埋め込み値を使用して、セクションに割り当てられないメモリ範囲の領域を埋めることができます。

注: メモリ範囲の埋め込み

大きなメモリ範囲に埋め込み値を指定した場合、出力ファイルは非常に大きくなります。その理由は、メモリ範囲の埋め込みを指定すると、(埋め込み値が 0 でも) その範囲内にあるすべての未割り当てメモリ・ブロック用に生データが生成されるからです。

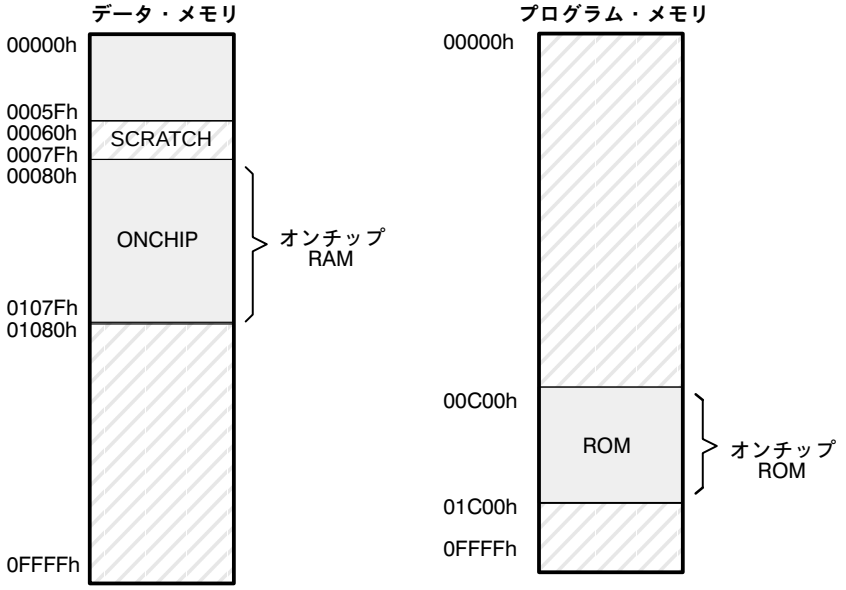
次の例は、R 属性と W 属性、および埋め込み定数 0FFFFh をもつメモリ範囲を指定しています。

```
MEMORY
{
       RFILE (RW) : o = 02h, l = 0FEh, f = 0FFFFh
}
```

MEMORY 疑似命令は、通常、出力セクションの割り当てを制御するために SECTIONS 疑似命令とともに使用します。MEMORY 疑似命令を使用してターゲット・システムのメモリ・モデルを指定した後、SECTIONS 疑似命令を使用して、出力セクションを特定の名前のメモリ範囲内か、特定の属性をもつメモリ内に割り当てることができます。たとえば、.text および .data セクションを ROM という名前の領域に割り当て、.bss セクションを ONCHIP という名前の領域に割り当てる、といった使い方ができます。

図 7−2 は、例 7−3 で定義されたメモリ・マップを示しています。

図 7−2.　例 7−3 で定義されたメモリ・マップ



7.8 SECTIONS 疑似命令

SECTIONS 疑似命令は、次の働きをします。

- ❑ 入力セクションを結合して出力セクションに入れる方法を指示する。
- ❑ 実行可能なプログラム内の出力セクションを定義する。
- ❑ メモリ内の出力セクションを配置する位置を指定する (相互に対照的に、および全メモリ空間について)。
- ❑ 出力セクション名の変更を可能にする。

リンカがセクションを処理する方法の詳細は、2-13 ページの 2.4 節「リンカによるセクションの処理」を参照してください。セクションの再配置については、2-16 ページの 2.5 節「再配置」を参照してください。サブセクションの定義については、2-9 ページの 2.3.4 項「サブセクション」を参照してください。サブセクションを使用すると、セクションをより正確に操作できます。

7.8.1 デフォルト構成

SECTIONS 疑似命令を指定しない場合、リンカはデフォルトのアルゴリズムを使用してセクションを結合し、セクションの割り当てを行ないます。7-58 ページの 7.12 節「デフォルトの割り当てアルゴリズム」で、このアルゴリズムについて詳しく説明します。

7.8.2 SECTIONS 疑似命令の構文

SECTIONS 疑似命令をコマンド・ファイル内で指定する場合は、SECTIONS (大文字) という文字に続けて、出力セクションの指定を中括弧に入れて指定します。

SECTIONS 疑似命令の一般的な構文は、次のとおりです。

```
SECTIONS
{
        name : [property, property, property,...]
        name : [property, property, property,...]
        name : [property, property, property,...]
}
```

各セクションの指定は、*name* で始まり、出力セクションを定義します (出力セクションとは、出力ファイルのセクションです)。セクション名の後ろには、セクションの内容とその割り当て方法を定義する属性のリストが続きます。属性と属性の間はコンマで区切ってもかまいません。セクションに対して指定できる属性には次のものがあります。

- **ロード割り当て：**メモリ内のセクションをロードする位置を指定します。

  構文: **load =** *allocation* **または**
  allocation **または**
  > *allocation*

- **実行割り当て：**メモリ内のセクションを実行する位置を指定します。

  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指定します。

  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指定します。

  構文: **type = COPY** **または**
  type = DSECT **または**
  type = NOLOAD

  セクションの型の詳細は、7-61 ページの 7.13 節「特別なセクションの型 (DSECT、COPY、および NOLOAD)」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指定します。

  構文: **fill =** *value* **または**
  name : ... { ... } **=** *value*

  ホールの作成と埋め込みの詳細は、7-66 ページの 7.15 節「ホールの作成方法と埋め込み方法」を参照してください。

例 7−4 は、サンプルのリンカ・コマンド・ファイルでの SECTIONS 疑似命令を示しています。図 7−3 は、これらのセクションがどのようにメモリに割り当てられるかを示しています。

例 7−4. SECTIONS 疑似命令

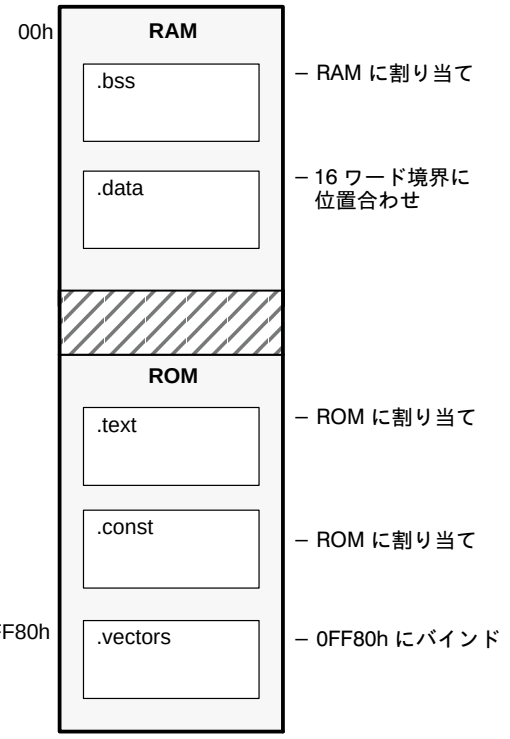
SECTIONS 疑似命令

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, run = 800h
  .const:    load = ROM
  .bss:      load = RAM
  .vectors:  load = FF80h
     {
        t1.obj(.intvec1)
        t2.obj(.intvec2)
        endvec = .;
     }
  .data:     align = 16
}
```

図 7−3 は、例 7−4 で SECTIONS 疑似命令によって定義された .vectors、.text、.const、.bss、および .data の 5 つの出力セクションを示しています。

図 7−3. 例 7−4 で定義されたセクションの割り当て



.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。

.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。

.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。

.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。

.vectors セクションは、t1.obj からの .intvec1 セクションと、t2.obj からの .intvec2 セクションにより構成されます。

7.8.3 割り当て

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もっていると考えられます。いずれの場合でも、ターゲット・メモリ内に出力セクションを置き、そのアドレスを割り当てることを「割り当て」と呼びます。ロードと実行に別の割り当てを使用する方法の詳細は、7-44 ページの 7.9 節「セクションの実行 (ランタイム) アドレスの指定方法」を参照してください。

リンカに対してセクションの割り当て方法を指定しない場合、リンカはデフォルトのアルゴリズムによってセクションの割り当てを行ないます。一般的には、リンカは構成メモリ内のセクションを配置できる任意の場所にセクションを入れます。このセクションのデフォルトの割り当ては、SECTIONS 疑似命令内でセクションを定義して、その割り当て方法を指示することで無効にできます。

1 つ以上の割り当てパラメータを指定して、割り当てを制御します。各パラメータは、キーワード、(=) や (>) 記号、および任意に括弧で囲んだ値から構成されます。ロードと実行に別々の位置を割り当てる場合、キーワード LOAD の後ろにあるすべてのパラメータはロード割り当てに適用され、キーワード RUN の後ろにあるすべてのパラメータは実行割り当てに適用されます。使用可能な割り当てパラメータには次のものがあります。

Binding セクションを特定のアドレスに割り当てます。

```
.text: load = 0x1000
```

Memory セクションを、MEMORY 疑似命令で定義された、指定された名前 (ROM など) または属性をもった範囲に割り当てます。

```
.text: load > ROM
```

Alignment セクションがアドレス境界から始まるように指定するには、align キーワードを使用します。

```
.text: align = 0x80
```

代入を含んでいる出力セクションも強制的に位置合わせするには、. (ドット) に align 式を代入します。たとえば、次のように指定すると、bar.obj が位置合わせされ、outsect が強制的に 0x40 ワード境界で位置合わせされます。

```
SECTIONS
{
   outsect:  {   bar.obj(.bss)
                 . = align(0x40);
             }
}
```

Blocking セクションが 2 つのアドレス境界の間に入っていなければならないことを指定するには、block キーワードを使用します。セクションが大きすぎる場合は、アドレス境界から開始されます。

```
.text: block(0x80)
```

Page 使用されるメモリ・ページを指定します (7-53 ページの 7.11 節「オーバーレイ・ページ」を参照)。

```
.text: PAGE 0
```

ロード割り当ての場合は (通常はこの割り当てのみ)、単に (>) 記号を使用して、LOAD キーワードを省略することができます。

```
.text: > ROM            .text: {...} > ROM
.text: > 0x1000
```

パラメータを 2 つ以上使用する場合は、次のように連続させることができます。

```
.text: > ROM align 16 PAGE 2
```

または、読みやすいように括弧を使用することもできます。

```
.text: load = (ROM align(16) page (2))
```

7.8.3.1 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 のワード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 (十分な空間があれば) のどこにでもバインドできますが、重複させることはできません。指定したアドレスにセクションをバインドするための十分な空間がない場合、リンカはエラー・メッセージを発行します。

注: 「バインディング」と「位置合わせ」または「名前付きメモリ」を同時に使用することはできません。

位置合わせ、または名前付きメモリを使用する場合は、セクションをアドレスにバインドすることはできません。バインドしようとした場合、リンカはエラー・メッセージを発行します。

7.8.3.2 Named memory (名前付きメモリ)

セクションは、MEMORY 疑似命令で定義したメモリ範囲に割り当てることができます。この例では、メモリ範囲に名前を付け、セクションをその範囲内にリンクしています。

```
MEMORY
{
   ROM (RIX)  :  origin = 0C00h,  length = 1000h
   RAM (RWIX) :  origin = 0080h,  length = 1000h
}

SECTIONS
{
   .text  :             > ROM
   .data ALIGN(128) :   > RAM
   .bss   :             > RAM
}
```

この例では、リンカは .text を ROM という領域に入れています。.data と .bss の出力セクションは RAM に割り当てられています。セクションを名前付きメモリ範囲内で位置合わせすることもできます。この例では、.data セクションは RAM というメモリ範囲内の 128 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もつメモリ領域にリンクできます。これを行なうには、メモリ名の代わりに一連の属性を (括弧に入れて) 指定します。同じ MEMORY 疑似命令宣言を使用して、次のように指定できます。

```
SECTIONS
{
   .text:  > (X)   /* .text --> executable memory      */
   .data:  > (RI)  /* .data --> read or init memory    */
   .bss :  > (RW)  /* .bss --> read or write memory    */
}
```

この例では、ROM 領域と RAM 領域のどちらも X 属性をもっているので、.text 出力セクションを両方の領域にリンクすることができます。また、ROM と RAM のどちらにも R 属性と I 属性があるので、.data セクションは両方の領域に入れることができます。ただし、W 属性をもっているのは RAM だけなので、.bss 出力セクションは RAM にしか入れることができません。

セクションを名前付きメモリ範囲の中のどの位置に割り当てるかは、制御することができません。ただし、リンカは、まず下位のメモリ・アドレスを使用し、さらに、断片化をできるだけ避けようとします。これまでの例では、衝突する割り当てがないことを前提にして、.text セクションはアドレス 0 から開始します。セクションを特定のアドレスで開始しなければならない場合は、名前付きメモリの代わりにバインディングを使用します。

7.8.3.3 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できます。ここで、n は 2 の累乗です。次に例を示します。

```
.text: load = align(128)
```

この例では、.text は 128 ワード境界上にくるように割り当てられます。

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当てられます。セクションがブロック・サイズより大きい場合は、セクションはブロックの境界上で始まります。位置合わせの場合と同様に、n は 2 の累乗でなければなりません。次に例を示します。

```
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、またはページ上で始まるように割り当てられます。

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用することはできません。

7.8.3.4 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できます。出力セクションのサイズは、出力セクションを構成する入力セクションのサイズの合計となります。リンカは、入力セクションのいずれにも位置合わせまたはブロック化が指定されていない場合には、指定された順序で連結することによって入力セクションを結合します。

どんな入力セクションでも位置合わせまたはブロック化が指定されている場合、出力セクション内の入力セクションは次の順序になります。

1) 続けられる最大から最小までのすべての位置合わせされたセクション

2) 最大から最小までのすべてのブロック化セクション

3) 最大から最小までのその他すべての入力セクション

例 7−5 は、最も一般的なセクションの指定方法を示しています。入力セクションのリストが 1 つも指定されていないことに注意してください。

例 7−5. 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 7−5 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用できます。

出力セクションを形成する入力セクションを、明示的に指定することができます。各入力セクションは、ファイル名とセクション名を使用して特定されます。

```
SECTIONS
{
   .text :             /* Build .text output section        */
   {
    f1.obj(.text)      /* Link .text section from f1.obj    */
    f2.obj(sec1)       /* Link sec1 section from f2.obj     */
    f3.obj             /* Link ALL sections from f3.obj     */
    f4.obj(.text,sec2) /* Link .text and sec2 from f4.obj   */
   }
}
```

入力セクションは、互いに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組み込まれます。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はあるが、SECTIONS 疑似命令によって明示的に指定されていない場合、その入力セクションは自動的に出力セクションの最後にリンクされます。たとえば前の例で、リンカがさらに .text セクションを検出し、その .text セクションが SECTIONS 疑似命令のどこにも指定されていない場合、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 7−5 の指定は、実際は次の指定方法を簡略化したものです。

```
SECTIONS
{
   .text: { *(.text) }
   .data: { *(.data) }
   .bss:  { *(.bss)  }
}
```

*(.text) という指定は、すべての入力ファイルからの割り当てられていない .text セクションを意味します。この形式は次のような場合に便利です。

- ❑ 特定の名前をもつすべての入力セクションを出力セクションに組み込みたいが、出力セクションの名前が入力セクションの名前と違う場合
- ❑ リンカに対して、追加の入力セクションや括弧に入ったコマンドを処理する前に入力セクションを割り当てさせたい場合

次の例は、上記で説明した 2 つの場合を示しています。

```
SECTIONS
{
   .text  :  {
                    abc.obj(xqt)
                       *(.text)
             }
   .data  :  {
                    *(.data)
                    fil.obj(table)
             }
}
```

この例では、.text 出力セクションには abc.obj ファイルの名前付きセクション xqt が組み込まれ、その後にすべての .text 入力セクションが続いています。.data セクションには、すべての .data 入力セクションが含まれ、その後に fil.obj ファイルの名前付きセクション table が続いています。この方法では、すべての割り当てられていないセクションが組み込まれます。たとえば、リンカが *(.text) を検出したときに .text 入力セクションの 1 つがすでに他の出力セクションに組み込まれている場合は、リンカはその .text 入力セクションをこの .text 出力セクションに組み込むことはしません。

7.8.3.5 複数のメモリ領域を使用した割り当て

リンカを使用すると、メモリ領域の明示的なリストを、割り当て可能な出力セクション内に指定することができます。次の例を考えてみましょう。

```
MEMORY
{
   P_MEM1 :  origin = 02000h,  length = 01000h
   P_MEM2 :  origin = 04000h,  length = 01000h
   P_MEM3 :  origin = 06000h,  length = 01000h
   P_MEM4 :  origin = 08000h,  length = 01000h
}

SECTIONS
{
   .text  : { } > P_MEM1 | P_MEM2 | P_MEM4
}
```

演算子 “|” は、複数のメモリ領域に指定するために使用されます。.text の出力セクションは、メモリ領域が当てはまる最初のメモリ領域内に全体が割り当てられます。そのメモリ領域は、指定された順でアクセスされます。この例では、リンカはまず P_MEM1 内のセクションを割り当てようとします。この試みが失敗すると、リンカはそのセクションを P_MEM2、およびその他に置こうとします。出力セクションがどの名前付きメモリ領域でも完全に割り当てられない場合、リンカはエラー・メッセージを発行します。

このタイプの SECTIONS 疑似命令の指定では、リンカは、本来割り当てられるメモリ領域の利用可能な空間を超えて増大する出力セクションを均一に取り扱うことができます。リンカ・コマンド・ファイルを変更せずに、リンカに、このセクションを他の範囲のどこかに移動させるようにすることができます。

7.8.3.6 出力セクションを不連続なメモリ領域に自動的に分割する方法

リンカは、出力セクションを複数のメモリ領域に分割して、効率的な割り当てを実現することができます。演算子 >> を使用して、出力セクションが (必要であれば) 指定された複数のメモリ領域内に分割できることを示します。次に例を示します。

```
MEMORY
{
   P_MEM1 :  origin = 02000h,  length = 01000h
   P_MEM2 :  origin = 04000h,  length = 01000h
   P_MEM3 :  origin = 06000h,  length = 01000h
   P_MEM4 :  origin = 08000h,  length = 01000h
}

SECTIONS
{
  .text: { *(.text) } >> P_MEM1 | P_MEM2 | P_MEM3 | P_MEM4
}
```

この例で演算子 >> は、.text 出力セクションが、リストされたどのメモリ範囲の中でも分割できることを示しています。.text セクションが P_MEM1 内の利用可能なメモリを超えて増大する場合は、入力セクションの境界上で分割されます。また、出力セクションの残りの部分は P_MEM2 | P_MEM3 | P_MEM4 に割り当てられます。

演算子 "|" は、複数のメモリ領域のリストを指定するために使用されます。

さらに、演算子 >> を使用して、出力セクションが単一のメモリ領域内で分割されうることを示すようにできます。この機能は複数の出力セクションが同じメモリ領域内に割り当てられなければならないときには便利ですが、ひとつの出力セクションによってそのメモリ領域が区分されるという制限があります。次の例を考えてみましょう。

```
MEMORY
{
   RAM :  origin = 01000h,  length = 08000h
}

SECTIONS
{
  .special: { f1.obj(.text) } = 04000h
  .text: { *(.text) } >> RAM
}
```

.special 出力セクションは、RAM メモリ領域の中央付近に割り当てられます。この出力セクションは 01000h から 04000h まで、および f1.obj(.text) の終わりから 08000h までの、RAM 内の 2 つの未使用範囲を残しておきます。.text セクション用のこの指定により、リンカは .special セクションの周囲にある .text セクションを分割し、および .special のいずれか側の RAM で利用可能な空間を使用できるようになります。

演算子 >> はまた、指定された属性の結合にマッチするすべてのメモリ領域の間にある出力セクションを分割するために使用できます。次に例を示します。

```
MEMORY
{
  P_MEM1 (RWX) : origin = 01000h,  length = 02000h
  P_MEM2 (RWI) : origin = 04000h,  length = 01000h
}

SECTIONS
{
  .text: { *(.text) } >> (RW)
}
```

リンカは、出力セクションのすべてまたは一部を、SECTIONS 疑似命令に指定された属性にマッチする属性をもつすべてのメモリ領域に割り当てようとします。

この SECTIONS 疑似命令は、次のものと同じ働きをします。

```
SECTIONS
{
  .text: { *(.text) } >> P_MEM1 | P_MEM2
}
```

特定の出力セクションは分割してはいけません。

- C/C++ プログラム用の自動初期化表を含む .cinit
- C++ プログラム用のグローバル・コンストラクタのリストを含む .pinit
- 独立したロード割り当ておよび実行割り当てをもつ出力セクション。出力セクションのロード時間割り当てから実行時の場所までの出力セクションをコピーするコードは、その出力セクション内での分割を受け入れることができません。
- 計算される式を含む入力セクションが指定された出力セクション。その式は、実行時に出力セクションを管理するプログラムで使用される記号を定義する可能性があります。

演算子 >> をこれらすべてのセクションで使用する場合、リンカは警告を発行し、その演算子を無視します。

7.8.3.7 アーカイブ・メンバを出力セクションに割り当てる方法

リンカを使用すると、1 つ以上のアーカイブ・ライブラリを特定の出力セクションに割り当てることができます。このような割り当ての構文は、次のとおりです。

```
SECTIONS
{
   .output_sec
   {
     [-l]lib_name<obj1 [obj2...objn]>  (.sec_name)
    }
}
```

この構文では、*lib_name* がアーカイブ・ライブラリです。ライブラリ検索アルゴリズムが常にこのアーカイブを検索するために使用されるので、–l はオプションです。–l オプションの詳細は、7-13 ページの 7.4.9 節「ライブラリ検索アルゴリズムの変更」を参照してください。不等号括弧 () は、アーカイブ・メンバを指定するために使用されます。不等号括弧は、1 つ以上のオブジェクト・ファイルを含み、スペースで区切られています。*sec_name* は、割り当てられるアーカイブ・セクションです。

次に例を示します。

```
SECTIONS
{
   .boot > BOOT1
   {
     –l rts.lib<boot.obj exit.obj strcpy.obj> (.text)
    }
   .rts > BOOT2
   {
     –l rts.lib (.text)
   }
   .text > RAM
   {
      * (.text)
   }
}
```

上記の指定では、rts.lib からの boot.obj、exit.obj、および strcpy.obj の .text セクションが .boot セクション内に置かれることになります。

rts.lib からの .text セクションの剰余は、rts. セクション内に置かれることになります。

その他すべての割り当てられない .text セクションは、.text セクション内に置かれることになります。

7.9 セクションの実行 (ランタイム) アドレスの指定方法

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といった使い方をしたいときがあります。たとえば、ROM ベースのシステムにパフォーマンスを大きく左右するコードがあるとします。そのコードは ROM にロードしなければなりませんが、実行は RAM を使用した方が速くなります。

リンカを使用すると、この指定を簡単に行うことができます。SECTIONS 疑似命令を使用すると、リンカに同じセクションを 2 回割り当てるように指示できます。つまり、最初はロード・アドレスを、2 回目は実行アドレスを設定するように指示できます。次に例を示します。

```
.fir: load = ROM, run = RAM
```

ロード・アドレスには load キーワードを使用し、実行アドレスには run キーワードを使用します。

実行時の再配置の概要については、2-19 ページの 2.6 節「実行時の再配置」を参照してください。

7.9.1 ロード・アドレスと実行アドレスの指定方法

ロード・アドレスにより、ローダがセクションの生データを配置する位置を決定します。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、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 回の割り当て (ロードまたは実行) しか行わない場合は、そのセクションは 1 回しか割り当てられず、ロードも実行も同じアドレスで行われます。2 つの割り当てを指定した場合は、そのセクションは同じサイズの 2 つの異なったセクションであるかのように割り当てられます。これは、両方の割り当てがそれぞれメモリ・マップ内の空間を占め、互いに、または他のセクションとオーバーレイできないことを意味します (UNION 疑似命令を使用すると、セクションをオーバーレイさせることができます。7-48 ページの 7.10.1 項「UNION 文によるセクションのオーバーレイ」を参照してください)。

ロード・アドレスか実行アドレスに位置合わせやブロック化などの追加のパラメータを指定する場合は、該当するキーワードの後ろに指定してください。load キーワードの後ろにあって run キーワードが見つかるまでの割り当てに関係があるすべてのものは、ロード・アドレスに影響を与えます。run キーワードの後にあるすべてのものは、実行アドレスに影響を与えます。ロード割り当てと実行割り当ては完全に独立しているため、一方の条件 (位置合わせなど) は他方の条件にまった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い、ロード用ではどこかの場所で 16 ワードに位置合わせを行います)

7.9.2 初期化されないセクション

初期化されないセクション (.bss など) はロードされないので、有効なアドレスは実行アドレスだけです。リンカは、初期化されないセクションを 1 回だけ割り当てます。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・アドレスを無視します。また、1 つのアドレスのみを指定すると、リンカは、実行アドレスまたはロード・アドレスに関係なくそのアドレスを実行アドレスとして取り扱います。次の例では、初期化されないセクション用のロード・アドレスと実行アドレスを指定しています。

```
.bss: load = 0x1000, run = RAM
```

この例では警告が発行され、ロードは無視され、空間は RAM に割り当てられます。次のすべての例では、同じ結果が得られます。.bss セクションは、RAM に割り当てられます。

```
.bss: load = RAM
.bss: run = RAM
.bss: > RAM
```

7.9.3 .label 疑似命令を使用してロード・アドレスを参照する方法

セクション内のシンボルへの参照は、通常、その実行アドレスを示します。ただし、実行時にロード時の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。.label 疑似命令は、セクションのロード・アドレスを示す特殊シンボルを定義します。したがって、通常のシンボルが実行アドレスを基準に再配置されるのに対して、.label シンボルはロード・アドレスを基準に再配置されます。.label 疑似命令の詳細は、4-60 ページを参照してください。

例 7−6 は、.label 疑似命令の使い方を示しています。

例 7−6.　セクションを ROM から RAM にコピーする方法

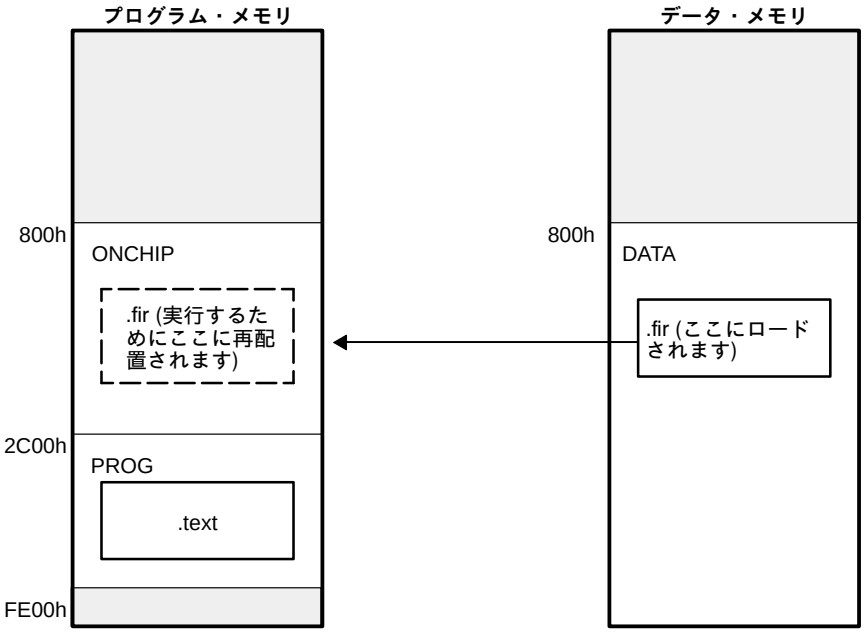
```
;-------------------------------------------------------
;  define a section to be copied from ROM to RAM
;-------------------------------------------------------
    .sect  ”.fir”
    .label fir_src        ; load address of section
fir:                      ; run address of section
    <code here>           ; code for the section

    .label fir_end        ; load address of section end

;-------------------------------------------------------
;  copy .fir section from ROM into RAM
;-------------------------------------------------------
    .text

    STM   fir_src, AR1    ; get load address
    RPT   #(fir_end – fir_src – 1)
    MVDP  *AR1+, fir      ; copy address to program memory

;-------------------------------------------------------
;  jump to section, now in RAM
;-------------------------------------------------------
      CALL  fir
```

リンカ・コマンド・ファイル

```
/*************************************************/
/*  PARTIAL LINKER COMMAND FILE FOR FIR EXAMPLE   */
/*************************************************/

MEMORY
{
   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 7−4 は、この例の実行時の様子を示しています。

図 7−4.　例 7−6 の実行時の様子



7.10 UNION 文と GROUP 文の使用方法

GROUP と UNION という 2 つの SECTIONS 文を使用して、メモリを節約できます。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当てることができます。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当てることができます。

7.10.1 UNION 文によるセクションのオーバーレイ

アプリケーションによっては、同じアドレスで複数のセクションを実行できるように割り当てることもあります。たとえば、プログラム実行のさまざまな段階で、オンチップ RAM にいくつかのルーチンを置きたい場合があります。また、同時にはアクティブに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当てることができます。

例 7−7 では、file1.obj と file2.obj の .bss セクションは RAM 内の同じアドレスに割り当てられています。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当てられます。

例 7−7.　UNION 文

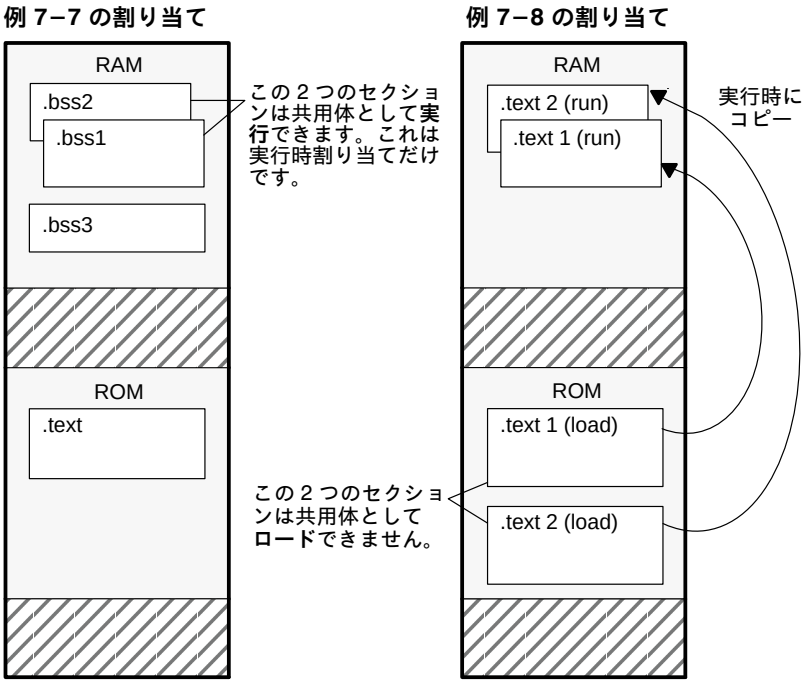
```
SECTIONS
{
   .text: load = ROM
   UNION: run = RAM
   {
      .bss1: { file1.obj(.bss) }
      .bss2: { file2.obj(.bss) }
   }
      .bss3: run = RAM { globals.obj(.bss) }
```

1 つのセクションを共用体の一部として割り当てることは、その実行アドレスにのみ影響を与えます。どのような場合でも、ロード時に複数のセクションをオーバーレイさせることはできません。初期化されたセクションが共用体のメンバである場合は (.text などの初期化されたセクションには生データがあります)、そのロード割り当ては必ず別々に指定しなければなりません。次に例を示します。

例 7−8.　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 7−5.　例 7−7 と例 7−8 で定義されたメモリ割り当て



.text セクションにはデータが入っているので、これを共用体として実行することはできますがロードすることはできません。したがって、各セクションに専用のロード・アドレスが必要となります。共用体の中の初期化されたセクションのロード割り当てに失敗した場合、リンカは警告を発行し、構成メモリ内の可能な場所にロード空間を割り当てます。

初期化されないセクションはロードされず、ロード・アドレスも必要ありません。

UNION 文は実行アドレスの割り当てにのみ使用するので、共用体そのもののロード・アドレスを指定する必要はありません。割り当てに対して、共用体は初期化されないセクションとして取り扱われます。指定された割り当ては、すべて実行アドレスとみなされます。また、両方の割り当てが指定された場合、リンカは警告を発行し、ロード・アドレスを無視します。

ひとつの共用体の位置合わせ属性とブロック化属性は、そのメンバすべての最大の位置合わせ属性およびブロック化属性です。

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 (7-53 ページの 7.11 節「オーバーレイ・ページ」を参照) は、どちらもオーバーレイを取り扱うので同じように見えます。しかし、実際にはまったく異なるものです。UNION 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。これに対して、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。ページ機能を使用して UNION の機能に近づけることは可能ですが、手間がかかります。

7.10.2 出力セクションをグループ化する方法

SECTIONS 疑似命令には、いくつかの出力セクションを連続して強制的に割り当てるための GROUP オプションがあります。たとえば、*term_rec* というセクションに .data セクション内のテーブルの終了レコードがあるとします。この場合、リンカに .data と term_rec を一緒に割り当てるように強制できます。

例 7−9. セクション・グループの割り当て

```
SECTIONS
{
   .text          /* Normal output section             */
   .bss           /* Normal output section             */
   GROUP 1000h :  /* Specify a group of sections       */
   {
      .data       /* First section in the group        */
      term_rec    /* Allocated immediately after .data */
   }
}
```

バインディング、位置合わせ、または名前付きメモリを使用して、単一の出力セクションの場合と同じ方法で GROUP を割り当てることができます。上の例では、GROUP はアドレス 1000h にバインドされています。これは、メモリ内で .data はワード 1000h に割り当てられ、term_rec はその後に続いて割り当てられるということです。

ひとつの GROUP の位置合わせ属性とブロック化属性は、そのメンバのすべての最大の位置合わせ属性およびブロック化属性です。

GROUP 用の割り当てルーチンは、7.10.4 に掲載されている一貫する検査規則に従います。

7.10.3 UNION および GROUP をネストする方法

リンカを使用すると、SECTIONS 疑似命令を使用して GROUP 文と UNION 文を任意にネストすることができます。GROUP 文と UNION 文をネストすることにより、セクションを階層的にオーバーレイしたりグループ化したりして表現することができます。例 7−10 は、セクションの 2 つのオーバーレイを相互にグループ化する方法を示しています。

例 7−10. GROUP 文および UNION 文をネストする方法

```
SECTIONS
{
   GROUP 1000h : run = RAM
   {
      UNION:
      {
       mysect1: load = ROM
       mysect2: load = ROM
       }
      UNION:
      {
       mysect3: load = ROM
       mysect4: load = ROM
      }
   }
}
```

この例では上記のリンカ制御ファイルが与えられていて、そのリンカは次の割り当てを実行します。

- 4 つのセクション (mysect1、mysect2、mysect3、mysect4) は、ROM メモリ領域の中で、固有で、オーバーラップしないロード・アドレスに割り当てられています。この割り当ては、各セクションに与えられている特定のロード割り当てにより決定されます。
- mysect1 および mysect2 セクションは、RAM 上で同じ実行アドレスが割り当てられています。
- mysect3 および mysect4 セクションは、RAM 上で同じ実行アドレスが指定されています。
- mysect1/mysect2 および mysect3/mysect4 の実行アドレスは、(位置合わせおよびブロックの制限に従った) GROUP 文に導かれて、隣接して割り当てられています。

グループおよび共用体を参照するために、リンカの診断メッセージは次の表示法を使用します。

GROUP_*n*
UNION_*n*

この表示法では、「*n*」は (1 で始まる) 連続番号であり、リンカ制御ファイルにおいて、ネストすることを考慮せずに、グループまたは共用体の語彙の順序を表しています。グループと共用体は、それぞれ固有のカウンタをもっています。

7.10.4 割り当てルーチンの一貫性を調べる方法

リンカは、共用体、グループ、およびセクション用のロード割り当てと実行割り当ての一貫性を調べます。使用される規則は次のとおりです。

- ❑ 実行割り当ては、トップレベルのセクション、グループ、または共用体 (その他のどのグループや共用体でもネストされないセクション、グループ、または共用体) でのみ行われます。リンカは、トップレベルの構造体の実行アドレスを使用して、グループおよび共用体の中でコンポーネントの実行アドレスを計算します。
- ❑ 7.10.1 節の記述にあるように、リンカは UNION に対するロード割り当てを受け入れません。
- ❑ 7.10.1 節の記述にあるように、リンカは初期化されないセクションに対するロード割り当てを受け入れません。
- ❑ ほとんどのケースでは、初期化されたセクションのためのロード割り当てを提供する必要があります。ただし、リンカは、ロード割り当てルーチンを既に定義したグループ内に配置される初期化されたセクションのためのロード割り当てを受け入れません。
- ❑ ショートカットによると、グループ全体用にロード割り当てを指定して、そのグループ内でネストされたすべての初期化されたセクションまたはサブグループ用のロード割り当てを判別することができます。ただし、ロード割り当ては次の条件がすべて真である場合にのみ、グループ全体用として受け入れられます。
  - ■ グループが初期化されている (たとえば、最低でも 1 つの初期化されたメンバをもっている)。
  - ■ グループがロード割り当てルーチンをもつ異なるグループ内でネストされていない。
  - ■ グループが、初期化されたセクションを含んでいる共用体を含まない。

  グループが初期化されたセクションをもつ共用体を含む場合、そのグループ内にネストされた初期化されたセクションのそれぞれに、ロード割り当てを指定する必要があります。次の例を考えてみましょう。

```
SECTIONS
{
  GROUP: load = ROM, run = ROM
  {
   .text1:
   UNION:
   {
    .text2:
    .text3:
   }
  }
}
```

グループ用に与えられたロード割り当て指示では、共用体の中の要素 .text2 および .text2 のためのロード割り当てを特別に指定していません。この場合、リンカは診断メッセージを発行して、これらのロード割り当てが明示的に指定されるように要求します。

7.11 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る、といったメモリ構成を使用するものがあります。この場合、システムは、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 つのアドレス範囲内やその範囲外にマップさせることができます。言い換えれば、物理メモリの複数のバンクは 1 つのアドレス範囲で互いにオーバーレイします。リンカを使用して、さまざまな出力セクションをこれらのバンクの各々にロードさせたり、ロード時にマップされないバンクにロードさせたりできます。

リンカは、オーバーレイ・ページを提供することによりこの機能をサポートします。各ページは、MEMORY 疑似命令を使用して別々に構成する必要がある 1 つのアドレス範囲を表します。SECTIONS 疑似命令を使用して、さまざまなページにマップされるセクションを指定できます。

7.11.1 MEMORY 疑似命令を使用してオーバーレイ・ページを定義する方法

リンカにとって、各オーバーレイ・ページは、完全な 24 ビットのアドレス指定可能な位置から構成されている完全に独立したメモリを表します。したがって、複数のセクションが異なるページにある場合には、それらを同じ (またはオーバーラップする) アドレスにリンクできます。

ページには、0 から始まる連続した番号が付けられます。PAGE オプションを指定しない場合、リンカは初期化されるセクションを PAGE 0 (プログラム・メモリ) に割り当て、初期化されないセクションを PAGE 1 (データ・メモリ) に割り当てます。

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 A00h ～ 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、各バンクは異なるデータで初期化できます。以降に、この構成を得るための MEMORY 疑似命令の使い方を示します。

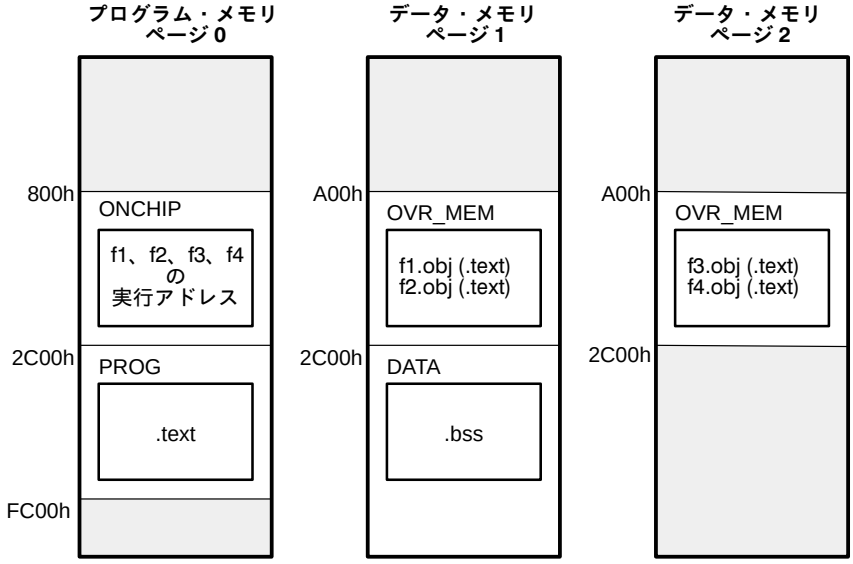
例 7−11. オーバーレイ・ページを指定した MEMORY 疑似命令

```
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 7−11 は、3 つの別々のアドレス範囲を定義しています。PAGE 0 は、オンチップ・プログラム・メモリの一領域と残りのプログラム・メモリ空間を定義します。PAGE 1 は、最初のオーバーレイ・メモリ領域と残りのデータ・メモリ空間を定義します。PAGE 2 は、データ空間の別のオーバーレイ・メモリ領域を定義します。両方の OVR_MEM 範囲は、同じアドレス範囲をカバーしています。これは、それぞれの範囲が異なるページにあって、別々のメモリ空間を表しているために可能となるのです。

図 7−6 は、例 7−11 に示す MEMORY 疑似命令と、例 7−12 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 7−6.　例 7−11 と例 7−12 で定義されているオーバーレイ・ページ



7.11.2 SECTIONS 疑似命令を使用してオーバーレイ・ページを使用する方法

MEMORY 疑似命令を、例 7−11 に示すように使用しているとします。さらに、コードには通常のセクションの他に、データ・メモリ空間にロードしてプログラム・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・モジュールが存在しているとします。例 7−12 は、この場合の SECTIONS 疑似命令によるオーバーレイの使い方を示しています。

例 7−12.　図 7−6 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、および f4 です。モジュール f1 および f2 は結合されて出力セクション S1 になり、f3 および f4 は結合されて出力セクション S2 になります。S1 および S2 用の PAGE 指定は、リンカにこれらのセクションを対応するページにリンクするように指示します。この結果、この 2 つのセクションはどちらもロード・アドレス A00h にリンクされますが、メモリ空間は異なります。プログラムがロードされるとき、ローダは、各セクションが適切なメモリ・バンクにロードされるようにハードウェアを構成できます。

出力セクション S1 および S2 は、オンチップ RAM 内に実行アドレスをもつ共用体の中に配置されます。アプリケーションは、実行時に、これらのセクションを転送してから実行する必要があります。セクション S1 の転送には、シンボル s1_load および s1_length を使用でき、セクション S2 の転送にはシンボル s2_load および s2_length を使用できます。特殊シンボル「.」は、現行のロード・アドレスではなく、現行の実行アドレスを指します。

ページ内では、通常の方法で、出力セクションをバインドしたり名前付きメモリ領域を使用したりできます。例 7−12 では、S1 を次のように割り当てることができます。

```
S1 : load = 01200h, page = 1  { . . . }
```

これは、S1 をページ 1 のアドレス 1200h にバインドします。ページをアドレスの修飾子として使用することもできます。次に例を示します。

```
S1 : load = (01200h PAGE 1)  { . . . }
```

セクションにバインディングや名前付きメモリ範囲を指定しない場合、リンカは（メモリ空間が 1 つだけの場合に通常行われるのと同様に）そのセクションを、割り当てることができる任意のページ内に割り当てます。たとえば、S2 は次のように指定することもできます。

```
S2 : PAGE 2  { . . . }
```

OVR_MEM はページ 2 の唯一のメモリなので、このセクションに = OVR_MEM を指定する必要はありません（ただし、指定してもかまいません）。

7.11.3 ページ定義構文

オーバーレイ・ページを例 7−11 および例 7−12 に示すように指定するには、MEMORY 疑似命令に次の構文を使用します。

MEMORY
{
PAGE 0 : *name 1* [*(attr)*] **: origin =** *constant* **, length =** *constant;*
PAGE n : *name n* [*(attr)*] **: origin =** *constant* **, length =** *constant;*
}

それぞれのページを定義するには、PAGE キーワードとページ番号を指定し、その後にコロンと、そのページに含まれるメモリ範囲リストを指定します。**ボールド体**の部分は、ここに示したとおりに入力する必要があります。メモリ範囲は、通常の方法で指定します。最大 255 ページまでのオーバーレイ・ページを定義できます。

それぞれのページは完全に独立したアドレス空間を表しているので、異なるページ上のメモリ範囲が同じ名前でもかまいません。あるページの構成メモリが、他のページの構成メモリとオーバーラップしてもかまいません。ただし、1 つのページ内では、すべてのメモリ範囲は固有の名前をもっていなければならず、オーバーラップも許されません。

PAGE 指定の有効範囲外に一覧表示されたメモリ範囲は、デフォルトで PAGE 0 になります。次の例について考えてみます。

```
MEMORY
{           ROM   : org = 0h     len = 1000h
            EPROM : org = 1000h  len = 1000h
            RAM   : org = 2000h  len = 0E000h
   PAGE1:   XROM  : org = 0h     len = 1000h
            XRAM  : org = 2000h  len = 0E000h
}
```

ROM、EPROM、および RAM の各メモリ範囲は、すべて PAGE 0 上にあります (ページが指定されていないため)。XROM と XRAM は PAGE 1 上にあります。PAGE 1 上の XROM は PAGE 0 上の ROM と重なっており、PAGE 1 上の XRAM は PAGE 0 上の RAM に重なっていることに注意してください。

出力リンク・マップ (リンカ・オプション −m によって取得) では、メモリ・モデルのリストにページ別にキーが付けられます。したがって、メモリ・モデルを正しく指定したかどうかを簡単に確認できます。また、出力セクションのリストには、各セクションがロードされるメモリ空間を示す PAGE の欄があります。

7.12 デフォルトの割り当てアルゴリズム

MEMORY および SECTIONS 疑似命令を使用すると、セクションを柔軟に作成し、結合し、割り当てることができます。しかし、指定しなかったメモリ・ロケーションやセクションも、リンカは処理する必要があります。リンカは指定された仕様の範囲内で、デフォルトのアルゴリズムを使用してセクションを作成し、割り当てます。7.12.1 項「割り当てアルゴリズム」、および 7.12.2 項「出力セクションの一般的な規則」にデフォルトの割り当てに関する説明があります。

7.12.1 割り当てアルゴリズム

MEMORY および SECTIONS 疑似命令を使用しない場合、リンカは以下の定義が指定されたものとして出力セクションを割り当てます。

例 7−13. TMS320C54x デバイスのデフォルトの割り当て

```
MEMORY
{
   PAGE 0: PROG: origin = 0x0080    length = 0xFF00
   PAGE 1: DATA: origin = 0x0080    length = 0xFF80
}
SECTIONS
{
   .text:     PAGE = 0
   .data:     PAGE = 0
   .cinit:    PAGE = 0    ;cflag option only
   .bss:      PAGE = 1
}
```

すべての .text 入力セクションは連結され、実行可能出力ファイルの中で 1 つの .text 出力セクションを形成します。また、すべての .data 入力セクションは結合され、1 つの .data 出力セクションを形成します。.text および .data のセクションは、プログラム・メモリ空間である PAGE 0 上の構成メモリの中に割り当てられます。すべての .bss セクションは結合され、.bss 出力セクションを形成します。この .bss セクションは、データ・メモリ空間である PAGE 1 上の構成メモリの中に割り当てられます。

入力ファイルの中に、初期化された名前付きセクションが含まれている場合、リンカは、これらのセクションをプログラム・メモリ内の .data セクションの直後に割り当てます。入力ファイルの中に、初期化されない名前付きセクションが組み込まれている場合、リンカは、これらのセクションをデータ・メモリ内の .bss セクションの直後に割り当てます。これは、SECTIONS 疑似命令の中で明示的に PAGE を指定することによって無効にできます。

SECTIONS 疑似命令を使用する場合、リンカは、デフォルト割り当てを一切行いません。割り当ては、7.12.2 項「出力セクションの一般的な規則」に述べる、SECTIONS 疑似命令で指定した規則、および一般的なアルゴリズムに従って実行されます。

7.12.2 出力セクションの一般的な規則

出力セクションは、次の 2 つの方法のいずれかで作成できます。

規則 1 SECTIONS 疑似命令の定義の結果として。

規則 2 SECTIONS 疑似命令で定義されていない同じ名前をも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 疑似命令の結果として作成される場合 (規則 1)、この定義は完全にセクションの内容を決定します (出力セクションの内容を定義する方法の例については、7-32 ページの 7.8 節「SECTIONS 疑似命令」を参照してください)。

出力セクションは、入力セクションが SECTIONS 疑似命令によって指定されていないときにも作成できます (規則 2)。この場合、リンカは同じ名前をもつすべての入力セクションを結合し、その名前をもつ出力セクションを作成します。たとえば、f1.obj と f2.obj の両方のファイルに Vectors という名前のセクションが入っており、SECTIONS 疑似命令はそれらのための出力セクションを定義していないとします。リンカは、入力ファイルに入っている 2 つの Vectors セクションを結合して Vectors という名前の 1 つの出力セクションを作成し、その出力セクションをメモリ内に割り当て、出力ファイルに組み込みます。

リンカは、すべての出力セクションの組成を決定した後、これらの出力セクションを構成メモリ内に割り当てる必要があります。MEMORY 疑似命令は、メモリのどの部分を構成するかを指定します。MEMORY 疑似命令がない場合、リンカはデフォルトの構成を使用します。

リンカの割り当てアルゴリズムは、メモリの断片化を最小限に抑えようとします。これにより、メモリを効率的に使用でき、プログラムがメモリ内に収まる可能性も高くなります。このアルゴリズムは、次のとおりです。

1) 特定のバインディング・アドレスが指定された出力セクションは、メモリ内のそのアドレスに配置されます。

2) 特定の、名前付きメモリ範囲に含まれるか、またはメモリ属性の制約がある出力セクションが割り当てられます。それぞれの出力セクションは、名前が指定された領域内の最初に使用可能な空間に配置され、その際、必要であれば位置合わせが考慮されます。

3) 残りのセクションがある場合、これらのセクションは定義された順に割り当てられます。SECTIONS 疑似命令で定義されていないセクションは、検出された順に割り当てられます。それぞれの出力セクションは、最初に使用可能なメモリ空間に配置され、その際、必要であれば位置合わせが考慮されます。

注: PAGE オプション

PAGE オプションを使用して出力セクションのメモリ空間を明示的に指定しない場合、リンカはそのセクションを PAGE 0 に割り当てます。これは、PAGE 0 に空きがなく、他のページに空きがある場合でも同様です。PAGE 0 以外のページを使用するには、SECTIONS 疑似命令でページを指定する必要があります。

7.13 特別なセクションの型 (DSECT、COPY、および NOLOAD)

出力セクションには、DSECT、COPY、および NOLOAD の特別な 3 つの型を割り当てることができます。これらの型は、プログラムのリンク時およびロード時に、そのプログラムの取り扱い方に影響を及ぼします。セクションに型を割り当てるには、セクション定義の後に、型を (丸括弧で囲んで) 指定します。次にこの例を示します。

```
SECTIONS
{
   sec1 2000h   (DSECT)   : {f1.obj}
   sec2 4000h   (COPY)    : {f2.obj}
   sec3 6000h   (NOLOAD)  : {f3.obj}
}
```

- DSECT 型は、以下の性質を備えたダミー・セクションを作成します。
  - ダミー・セクションは、出力セクションのメモリ割り当てに含まれません。メモリを占有せず、メモリ・マップ・リストにも含まれません。
  - ダミー・セクションは、他の出力セクション、他の DSECT、および未構成メモリにオーバーレイできます。
  - ダミー・セクション内で定義したグローバル・シンボルは、通常どおり再配置されます。これらのシンボルは、DSECT が実際にロードされた場合にもつことになる値をもって、出力モジュールのシンボル・テーブル内に置かれます。これらのシンボルは、別の入力セクションから参照できます。
  - DSECT 内で未定義の外部シンボルが検出されると、指定したアーカイブ・ライブラリが検索されます。
  - ダミー・セクションの内容、再配置情報、および行番号情報は、出力モジュール内には配置されません。

  上記の例では、f1.obj に入っている各セクションは割り当てられませんが、すべてのシンボルは、これらのセクションがワード・アドレス 2000h でリンクされているものとして再配置されます。他のセクションは、sec1 内のすべてのグローバル・シンボルを参照できます。
- COPY セクションは、その内容と関連情報が出力モジュールに書き込まれる点を除けば、DSECT セクションと同様のものです。TMS320C54x C/C++ コンパイラ用の初期化テーブルが入っている .cinit セクションは、RAM モデルでは、この属性を備えています。プラグマ IDENT により作成された .comment セクションは、COPY セクションです。
- NOLOAD セクションは、通常の出力セクションと 1 つだけ異なる点があります。それは、セクションの内容、再配置情報、および行番号情報が出力モジュールに配置されない点です。リンカは、このセクション用に空間を割り当て、このセクションはメモリ・マップ・リストに含まれます。

7.14 リンク時のシンボルの割り当て方法

リンカ代入文を使用すると、外部 (グローバル) シンボルを定義し、リンク時にこれらのシンボルに値を代入できます。この機能を使用して、変数やポインタを割り当てに依存した値に初期化できます。

7.14.1 代入文の構文

リンカでの代入文の構文は、C 言語の代入文の構文と同様のものです。

symbol	=	*expression;*	シンボルに式の値を代入します。
symbol	+=	*expression;*	シンボルに式の値を加算します。
symbol	–=	*expression;*	シンボルから式の値を減算します。
symbol	*=	*expression;*	シンボルに式の値を乗算します。
symbol	/=	*expression;*	式の値でシンボルを除算します。

シンボルは、外部で定義されなければなりません。外部が定義されない場合は、リンカは新しいシンボルを定義し、これをシンボル・テーブルに入れます。式は、7.14.3 項「代入式」で定義されている規則に従っていなければなりません。代入文は、必ずセミコロンで終了しなければなりません。

リンカは、すべての出力セクションを割り当てた後に代入文を処理します。したがって、式にシンボルが含まれている場合、そのシンボルに使用されたアドレスは、そのシンボルの実行可能出力ファイル内のアドレスを反映します。

たとえば、プログラムが Table1 と Table2 という 2 つの外部シンボルによって表される 2 つのテーブルのいずれかからデータを読み取るとします。このプログラムは、現行のテーブルのアドレスとして cur_tab というシンボルを使用します。cur_tab は、必ず Table1 と Table2 のいずれかを指していなければなりません。これは、アセンブリ・コードの中でも実現できますが、テーブルを変更するには、プログラムを再度アセンブルする必要があります。これを避けるため、次のようにリンカの代入文を使用して、リンク時に cur_tab に値を割り当てることができます。

```
prog.obj            /* Input file                          */
cur_tab = Table1;   /* Assign cur_tab to one of the tables */
```

7.14.2 SPC をシンボルに割り当てる方法

1 つのドット (.) で表記する特殊記号は、割り当て時の SPC の現行値を表します。リンカの「.」記号は、アセンブラの $ 記号に似ています。「.」記号は、SECTIONS 疑似命令の中の代入文にのみ使用できます。この理由は、「.」は割り当てのときにだけ意味をもち、SECTIONS は割り当てプロセスを制御するからです (7-32 ページの 7.8 節「SECTIONS 疑似命令」を参照)。「.」記号は 1 つの出力セクションを定義する大括弧の外側で使用できないことに注意してください。

「.」は、セクションの現行のロード・アドレスではなく、現行の実行アドレスを指します。

たとえば、プログラムで .data セクションの開始アドレスを知る必要があるとします。この場合、.global 疑似命令を使用することにより、プログラム内に Dstart という未定義の外部変数を作成できます。その後、次のように Dstart に「.」の値を代入します。

```
SECTIONS
{
   .text:   {}
   .data:   { Dstart = .; }
   .bss:    {}
}
```

これにより、Dstart は .data セクションの最初にリンクされるアドレスとして定義されます (Dstart には、.data が割り当てられる前に代入が行われます)。リンカは、Dstart へのすべての参照を再配置します。

「.」記号に値を代入する特別な種類の代入があります。この代入により、出力セクション内の SPC が調整され、2 つの入力セクションの間にホールが作成されます。ホールを作成するために「.」に代入される値は、実際に「.」で表されるアドレスではなく、セクションの先頭からの相対アドレスです。「.」への代入とホールについては、7-66 ページの 7.15 節「ホールの作成方法と埋め込み方法」に説明があります。

7.14.3 代入式

リンカ式には、以下の規則が適用されます。

- 式には、グローバル・シンボル、定数、および表 7−1 に示す C 言語の演算子を使用できます。
- すべての数値は、ロング (32 ビット) 整数として扱われます。
- 定数がリンカによって識別される方法は、アセンブラの場合と同じです。つまり、数値は接尾部 (16 進数を表す H または h、8 進数を表す Q または q) がなければ、10 進数と見なされます。C 言語の接頭部 (8 進数を表す 0、16 進数を表す 0x) も認識されます。16 進定数の最初の文字は、数字でなければなりません。2 進定数は使用できません。

- ❑ 式の中のシンボルは、自身のアドレスだけを値としてもつことができます。型のチェックは行われません。
- ❑ リンカ式は絶対的でも再配置可能でもかまいません。式に 1 つでも再配置可能シンボル (およびゼロ個以上の定数または絶対シンボル) が含まれていれば、この式は再配置可能です。それ以外の場合、式は絶対的です。シンボルに再配置可能な式の値が代入されていれば、このシンボルは再配置可能です。また、絶対的な式の値が代入されていれば、このシンボルは絶対的です。

リンカは、表 7–1 に示した C 言語の演算子を、この表の優先順位でサポートします。同じグループの演算子は、同じ優先順位をもちます。表 7–1 に示した演算子のほか、リンカには align という演算子もあり、これを使用すると、シンボルを出力セクション内で n ワード境界に位置合わせできます (n は 2 の累乗です)。たとえば、

```
. = align(16);
```

という式は、現行のセクション内の SPC を次の 16 ワード境界に位置合わせします。align 演算子は現行の SPC の関数なので、「.」と同じ文脈、つまり、SECTIONS 疑似命令の中でのみ使用できます。

表 7–1. 式で使用できる演算子 (優先順位)

シンボル	演算子	評価
+ – ~	単項のプラス、マイナス、1 の補数	右から左
* / %	乗算、除算、剰余	左から右
+ –	加算、減算	左から右
<< >>	左シフト、右シフト	左から右
< <= > >=	より小、以下、より大、以上	左から右
!= =[=]	等しくない、等しい	左から右
&	ビット単位の AND	左から右
^	ビット単位の XOR	左から右
|	ビット単位の OR	左から右

注: 単項の +、–、および * は、2 項の形より優先順位が高くなります。

7.14.4 リンカで定義されるシンボル

リンカは、プログラムが実行時にセクションのリンク先を決めるために使用できるいくつかのシンボルを自動的に定義します。これらのシンボルは外部シンボルなので、リンク・マップの中に示されます。これらのシンボルは、.global 疑似命令で宣言されていれば、どのアセンブリ言語モジュールからでもアクセスできます。これらのシンボルには、次のように値が代入されます。

.text .text 出力セクションの最初のアドレスが代入されます。
(実行可能コードの先頭を示します)

etext .text 出力セクションの直後にある最初のアドレスが代入されます。
(実行可能コードの終わりを示します)

.data .data 出力セクションの最初のアドレスが代入されます。
(初期化されたデータ・テーブルの先頭を示します)

edata .data 出力セクションの直後にある最初のアドレスが代入されます。
(初期化されたデータ・テーブルの終わりを示します)

.bss .bss 出力セクションの最初のアドレスが代入されます。
(初期化されないデータの先頭を示します)

end .bss 出力セクションの直後にある最初のアドレスが代入されます。
(初期化されないデータの終わりを示します)

7.14.5 C サポート用のみに定義されるシンボル (−c オプションまたは −cr オプション)

__STACK_SIZE .stack セクションのサイズが代入されます。

__SYSMEM_SIZE .sysmem セクションのサイズが代入されます。

7.15 ホールの作成方法と埋め込み方法

リンカは、出力セクション内に、どこからもリンクされていない領域を作成する機能を備えています。このような領域を**ホール**と呼びます。特別なケースとして、初期化されないセクションをホールとして扱うこともできます。この節では、リンカがこのようなホールを処理する方法と、ホール（および初期化されないセクション）に値を埋め込む方法について説明します。

7.15.1 初期化されたセクションと初期化されないセクション

出力セクションは、次のどちらかの状態にあります。

- ❑ セクション全体の生データを含んでいる。
- ❑ 生データを含んでいない。

生データをもつセクションを初期化されたセクションといいます。これは、オブジェクト・ファイルにそのセクションの実際のメモリ・イメージ内容が含まれることを意味します。このイメージは、セクションがロードされるとき、そのセクションの指定された開始アドレスのメモリ内にロードされます。.text および .data のセクションは、その中にアセンブルされたものがあれば、必ず生データを含んでいます。.sect アセンブラ疑似命令で定義された名前付きセクションも、生データをもっています。

デフォルトでは、.bss セクション、および .usect 疑似命令で定義されたセクション（初期化されないセクションです）には、生データは含まれません。これらのセクションはメモリ・マップ内の空間を占めますが、実際の内容はありません。初期化されないセクションは、通常、変数用の空間を RAM 内に予約します。オブジェクト・ファイル内では、初期化されないセクションは通常のセクション・ヘッダをもち、その中に定義されたシンボルをもつ場合があります。しかし、セクション内にメモリ・イメージは格納されません。

7.15.2 ホールの作成方法

初期化された出力セクションにホールを作成できます。ホールが作成されるのは、1 つの出力セクションの中で入力セクション相互の間に余分の空間を残すようにリンカに強制するときです。このようなホールが作成されるとき、リンカは上記の最初のガイドラインに従って、ホール用の生データを提供する必要があります。

ホールが作成されるのは、出力セクションの中だけです。出力セクション相互の間にも空間を残すことができますが、このような空間はホールではありません。SECTIONS 疑似命令を使用して、出力セクション相互間の空間を埋めたり初期化したりすることはできません。

出力セクション内にホールを作成するには、出力セクション定義の中で特別なリンカ代入文を使用する必要があります。この代入文は、SPC（「.」で表される）に加算を行うか、より大きい値を代入するか、あるいは SPC をアドレス境界上に位置合わせするかします。代入文の演算子、式、および構文については、7-62 ページの 7.14 節「リンク時のシンボルの割り当て方法」に説明があります。

次の例では、代入文を使用して出力セクション内にホールを作成します。

```
SECTIONS
{
   outsect:
   {
      file1.obj(.text)
      . += 100h;  /* Create a hole with size 100h words */
      file2.obj(.text)
      . = align(16);  /* Create a hole to align the SPC */
      file3.obj(.text)
   }
}
```

出力セクション outsect は、次のように作成されます。

- file1.obj に入っている .text セクションがリンクされます。
- リンカは 256 ワードのホールを作成します。
- file2.obj に入っている .text セクションがホールの後にリンクされます。
- リンカは SPC を 16 ワード境界上に位置合わせすることによって、別のホールを作成します。
- 最後に、file3.obj に入っている .text セクションがリンクされます。

セクション内で「.」記号に代入されたすべての値は、セクション内の相対アドレスを参照します。リンカは「.」記号への代入を、セクションが (バインディング・アドレスが指定された場合でも) アドレス 0 から始まっているものとして処理します。たとえば、この例の . = align(16) という文について考えてみます。この文は、file3.obj の .text を outsect 内の 16 ワード境界に位置合わせする効果があります。outsect が最終的に、位置合わせされていないアドレス上から始まるように割り当てられた場合、file3.obj の .text も位置合わせされません。

「.」記号は、セクションの現行のロード・アドレスではなく、現行の実行アドレスを指すことに注意してください。

「.」をデクリメントする式は、不正です。たとえば、「.」への代入に − = 演算子を使用しても無効です。「.」への代入に最もよく使用される演算子は、+= と align です。

ある出力セクションに含まれるすべての入力セクションが 1 種類 (.text など) の場合、以下の文を使用して、出力セクションの始めか終わりにホールを作成できます。

```
.text:   {  .+= 100h; }    /* Hole at the beginning */
.data:   {
            *(.data)
            . += 100h; }   /* Hole at the end       */
```

出力セクションにホールを作成するもう 1 つの方法は、初期化されないセクションを初期化されたセクションに結合して、1 つの出力セクションにする方法です。この場合、リンカは初期化されないセクションをホールとして扱い、そのセクションにデータを提供します。次の例は、この方法を示しています。

```
SECTIONS
{
   outsect:
   {
   file1.obj(.text)
   file1.obj(.bss)        /* This becomes a hole */
   }
}
```

.text セクションには生データがあるので、すべての outsect には生データが入っていなければなりません (規則 1)。したがって、初期化されない .bss セクションはホールになります。

初期化されないセクションがホールになるのは、それらのセクションを初期化されたセクションに結合した場合だけです。複数の初期化されないセクションをまとめてリンクしても、結果として生成される出力セクションは、やはり初期化されないセクションです。

7.15.3 ホールの埋め込み方法

初期化された出力セクションにホールが存在する場合、リンカはそのホールを埋める生データを提供しなければなりません。リンカは、16 ビットの埋め込み値を使用してホールを埋めます。この値はホール全体がいっぱいになるまで、メモリを介してコピーされます。リンカは、次のようにして埋め込み値を決定します。

1) 初期化されないセクションと初期化されたセクションを結合してホールが作成される場合、初期化されないセクションの埋め込み値を指定できます。次のように、セクション名の後に = 記号と 16 ビット定数を指定してください。

```
SECTIONS
{
   outsect:
   {
      file1.obj(.text)
      file2.obj(.bss) = 00FFh   /* Fill this hole */
   }                            /* with 0FFh      */
}
```

2) 次のように出力セクションの定義の後に埋め込み値を指定することで、その出力セクション内のすべてのホールの埋め込み値を指定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) ホールの初期値を指定しない場合、リンカは –f オプションで指定された値でホールを埋めます。たとえば、コマンド・ファイル link.cmd に次のような SECTIONS 疑似命令が入っているとします。

```
SECTIONS
{
   .text: { .= 100; }      /* Create a 100-word hole */
}
```

ここで、次のように –f オプションを指定してリンカを起動します。

```
lnk500 –f 0FFFFh link.cmd
```

これによって、ホールは 0FFFFh で埋められます。

4) リンカを起動するときに –f オプションを使用しなi場合、リンカは 0 でホールを埋めます。

ホールが作成され、初期化された出力セクション内でホールが埋められた場合、そのホールは必ず、リンカが埋め込みに使用した値とともにリンク・マップ内に示されます。

7.15.4 初期化されないセクションの明示的な初期化

初期化されないセクションがホールになるのは、初期化されたセクションに結合された場合だけです。初期化されないセクションを互いに結合しても、結果として生成される出力セクションは、初期化されないセクションのままです。

しかし、SECTIONS 疑似命令の中で明示的に埋め込み値を指定することにより、リンカに対して初期化されないセクションを初期化させるように強制することができます。これにより、セクション全体が生データ (埋め込み値) をもつようになります。次に例を示します。

```
SECTIONS
{
   .bss: fill = 1234h      /* Fills .bss with 1234h */
}
```

注: セクションの埋め込み

セクションに埋め込みを行うと、たとえ埋め込み値が 0 でも、出力ファイル内のセクション全体を埋める生データが生成されます。このため、大きなセクションやホールに埋め込み値を指定すると、出力ファイルは非常に大きくなります。

7.16 部分 (インクリメンタル) リンク

すでにリンクが済んだ出力ファイルは、追加モジュールとともに再度リンクできます。これを、部分リンクまたはインクリメンタル・リンクと呼びます。部分リンクを使用すると、大きなアプリケーションを分割し、個々の部分を別々にリンクしてからすべての部分をまとめてリンクし、最終的な実行可能プログラムを作成することができます。

再リンクするファイルを作成するためには、次のガイドラインに従ってください。

- ❏ 中間ファイルには、再配置情報が必要です。ファイルを初回にリンクするときは、−r オプションを使用してください。
- ❏ 中間ファイルには、シンボル情報が必要です。デフォルトでは、リンカの出力にはシンボル情報が保持されます。ファイルを再リンクする予定がある場合は、−s オプションを使用してはなりません。−s は出力モジュールからシンボル情報を除去するからです。
- ❏ 中間のリンク段階は、出力セクションの編成だけに関係します。割り当てには関係しません。すべての割り当て、バインディング、および MEMORY 疑似命令は最終リンクの段階で実行してください。
- ❏ 中間ファイルに、別のファイル内のグローバル・シンボルと同じ名前のグローバル・シンボルが存在し、それらのシンボルを静的シンボルとして (中間ファイルの中にだけ見えるように) 扱う場合は、−h オプションを使用してファイルをリンクする必要があります (7-12 ページの 7.4.7 項「すべてのグローバル・シンボルの静的化 (−h オプション)」を参照してください)。
- ❏ C コードをリンクする場合は、最終リンクの段階まで −c または −cr を使用しないでください。−c または −cr オプションを指定してリンカを起動するたびに、リンカはエントリ・ポイントを作成しようとします。

以下は、部分リンクの使い方を示しています。

手順 1: file1.com ファイルをリンクします。−r オプションを使用して、出力ファイル tempout1.out 内に再配置情報を保持します。

```
lnk500 –r –o tempout1 file1.com
```

file1.com の内容は次のとおりです。

```
SECTIONS
{
   ss1:   {
          f1.obj
          f2.obj
           .
           .
           .
          fn.obj
          }
}
```

手順 2: file2.com ファイルをリンクします。−r オプションを使用して、出力ファイル tempout2.out 内に再配置情報を保持します。

```
lnk500 –r –o tempout2 file2.com
```

file2.com の内容は次のとおりです。

```
SECTIONS
{
   ss2:   {
          g1.obj
          g2.obj
           .
           .
           .
          gn.obj
          }
}
```

手順 3: tempout1.out と tempout2.out をリンクします。

```
lnk500 –m final.map –o final.out tempout1.out tempout2.out
```

7.17 C/C++ コードのリンク

TMS320C54x C/C++ コンパイラは、アセンブルとリンクができるアセンブリ言語ソース・コードを生成します。たとえば、次のように prog1、prog2 などのモジュールで構成される C/C++ プログラムをアセンブルしてからリンクし、prog.out という名前の実行可能ファイルを作成できます。

```
lnk500 -c -o prog.out prog1.obj prog2.obj ... rts.lib
```

−c オプションは、C/C++ 環境によって定義された特別な規則を使用するようにリンカに指示します。ランタイム・ライブラリには、C/C++ のランタイムサポート関数が入っています。

C/C++ の詳細は、ランタイム環境とランタイムサポート関数に関する説明も含め、TMS320C54x オプティマイジング C/C++ コンパイラ ユーザーズ・マニュアルを参照してください。

7.17.1 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・モジュールとリンクしなければなりません。プログラムは、実行を開始すると、最初に boot.obj を実行します。boot.obj には、ランタイム環境を初期化するためのコードとデータが入っています。このモジュールは、以下の作業を実行します。

- ❏ システム・スタックをセットアップする。
- ❏ 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 rts.lib には、boot.obj が入っています。以下のことができます。

- ❏ アーカイバを使用してライブラリから boot.obj を抽出し、このモジュールを直接リンクする。
- ❏ rts.lib を入力ファイルとして組み込む (−c または −cr オプションを使用した場合、リンカは自動的に boot.obj を抽出します)。

7.17.2 オブジェクト・ライブラリとランタイム・サポート

TMS320C54x 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、rts.lib に含まれている追加のランタイムサポート関数に関する説明があります。プログラムでこれらの関数のいずれかを使用する場合は、rts.lib をオブジェクト・ファイルにリンクする必要があります。

ユーザ独自のオブジェクト・ライブラリを作成し、それをリンクすることもできます。リンカは、未定義の参照を解決するライブラリ・メンバだけを組み込んでリンクします。

7.17.3 スタック・セクションとヒープ・セクションのサイズ設定

C は、.sysmem と .stack という 2 つの初期化されないセクションをメモリ・プールとして使用します。メモリ・プールの .sysmem は malloc( ) 関数が、メモリ・プールの .stack はランタイム・スタックがそれぞれ使用します。これらのサイズを設定するには、−heap オプションまたは −stack オプションを使用し、これらのオプションの直後にセクションのサイズを定数として指定します。この 2 つのセクションのデフォルト・サイズはどちらも 1K ワードです。

詳細は、7-12 ページの 7.4.8 項「ヒープ・サイズの定義 (−heap constant オプション)」、および 7-18 ページの 7.4.16 項「スタック・サイズの定義 (−stack constant オプション)」を参照してください。

7.17.4 自動初期化 (ROM モデルと RAM モデル)

C/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生成します。それらのテーブルは、*.cinit* という名前付きセクションに入っています。初期化テーブルは、次に示す 2 つのどちらかの方法で使用できます。

- ❑ **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、ブート時間は短縮され、初期化テーブルに使用するメモリも節約できるので、パフォーマンスが向上します。自動初期化の RAM モデルを利用するには、スマート・ローダ (つまり、変数を初期化する機能があるローダ) を使用する必要があります。

  −cr を使用する場合、リンカは .cinit セクションに特別な属性を付加します。この属性は、リンカにその .cinit セクションをメモリ内にロードしないように指示します。また、リンカは cinit シンボルを −1 に設定します。これは、C/C++ ブート・ルーチンに、初期化テーブルがメモリ内に存在しないことを知らせます。これにより、ブート時に実行時初期化は行われません。

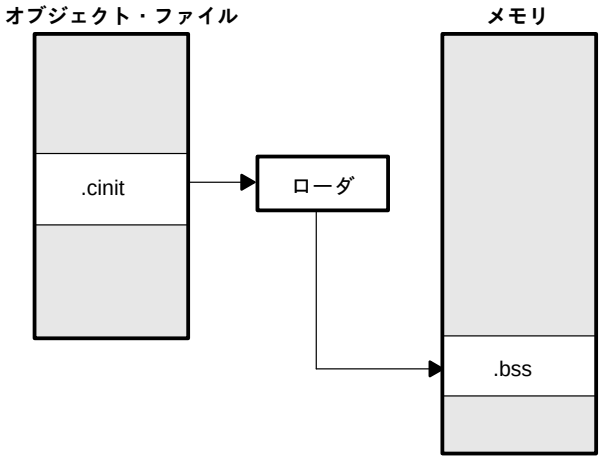
  プログラムをロードするとき、ローダは以下のことができなければなりません。

  - ■ オブジェクト・ファイル内の .cinit セクションの存在を検出する。
  - ■ .cinit セクションをコピーしないように指示する属性の存在を検出する。
  - ■ 初期化テーブルのフォーマットを認識する (このフォーマットについては、TMS320C54x オプティマイジング C/C++ コンパイラ ユーザーズ・マニュアルに説明があります)。

  この後、ローダは初期化テーブルをオブジェクト・ファイルから直接使用して、.bss 内の変数を初期化します。

図 7−7 は、RAM 自動初期化モデルを示しています。

図 7−7.　自動初期化の RAM モデル

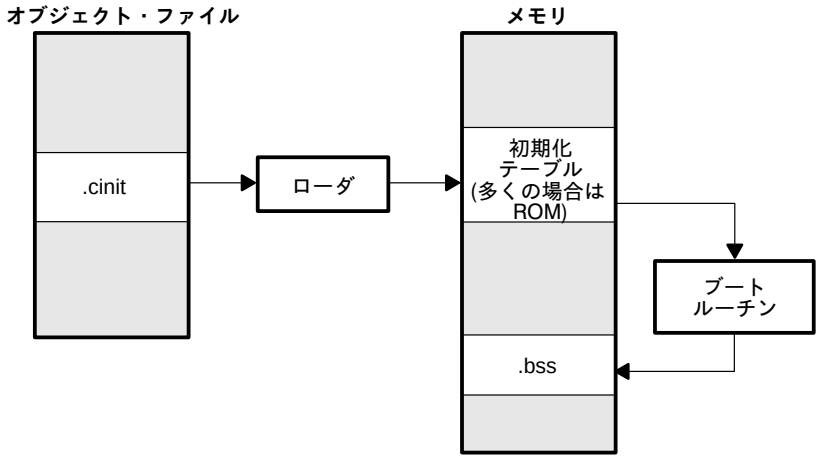


❑ **ROM モデル** (−c オプション)

変数は、実行時に初期化されます。.cinit セクションは、他のセクションとともにメモリ内にロードされます。リンカは、メモリ内のテーブルの始めを指す cinit という特殊シンボルを定義します。プログラムが実行を開始すると、C/C++ ブート・ルーチンは、テーブルに入っているデータを .bss セクション内の指定された変数の中にコピーします。これにより、初期化データは ROM 内に格納され、プログラムが起動されるたびに RAM にコピーされることができるようになります。

図 7−8 は、ROM 自動初期化モデルを示しています。

図 7−8.　自動初期化の ROM モデル



7.17.5 −c リンカ・オプションと −cr リンカ・オプション

以下に、−c または −cr のオプションを指定してリンカを起動したときに何が起きるかをまとめます。

- シンボル _c_int00 はプログラムのエントリ・ポイントとして定義されます。_c_int00 は boot.obj 内の C/C++ ブート・ルーチンの開始点です。_c_int00 を参照することにより、boot.obj はランタイムサポート・ライブラリ rts.lib から自動的にリンクされます。
- .cinit 出力セクションには終端レコードが埋め込まれ、ブート・ルーチン (ROM モデルの場合) またはローダ (RAM モデルの場合) に初期化テーブルの読み取りを停止する時期を知らせます。
- ROM モデル (−c オプション) では、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/C++ ブート・ルーチンは、このシンボルを自動初期化の開始点として使用します。
- 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内に存在しないことを示し、したがって初期化は実行時に行われません。
  - STYP_COPY フラグ (0010h) は .cinit セクションのヘッダ内に設定されます。STYP_COPY は特別な属性で、ローダに対して、自動初期化を直接実行し、.cinit セクションをメモリ内にロードしないように指示します。リンカは、メモリ内に .cinit セクション用の空間を割り当てません。

7.18 リンカの例

この例では、demo.obj、fft.obj、および tables.obj という 3 つのオブジェクト・ファイルをリンクし、demo.out というプログラムを作成します。シンボル SETUP は、プログラムのエントリ・ポイントです。

ターゲット・メモリは次のような構成であるとします。

プログラム・メモリ

アドレス範囲	内容
0080 ～ 7000	オンチップ RAM_PG
C000 ～ FF80	オンチップ ROM

データ・メモリ

アドレス範囲	内容
0080 ～ 0FFF	RAM ブロック ONCHIP
0060 ～ FFFF	マップされた外部アドレス EXT

出力セクションは、以下の入力セクションから構築されます。

- ❑ demo.obj、fft.obj、および tables.obj の .text セクションに入っている実行可能コードは、プログラム ROM にリンクされなければなりません。
- ❑ demo.obj の var_defs セクションに入っている変数は、ONCHIP ブロック内のデータ・メモリの中へリンクされなければなりません。
- ❑ demo.obj、tables.obj、および fft.obj の .data セクションに入っている係数テーブルは、データ・メモリ内の RAM ブロック ONCHIP の中へリンクされなければなりません。長さが 100 ワードで、埋め込み値が 07A1Ch のホールが作成されます。ONCHIP ブロックの残りの部分は、07A1Ch の値に初期化されなければなりません。
- ❑ demo.obj、tables.obj、および fft.obj の変数が入った .bss セクションは、プログラム RAM の RAM_PG ブロックの中へリンクされなければなりません。この RAM の未使用部分は、0FFFFh に初期化されなければなりません。
- ❑ demo.obj のバッファと変数が入った xy セクションは、明示的にリンクされなかったので、デフォルトとしてデータ RAM の ONCHIP ブロックの中へリンクされます。

例 7−14 は、この例のリンカ・コマンド・ファイルを示しています。例 7−15 は、マップ・ファイルを示しています。

例 7−14.　リンカ・コマンド・ファイル、demo.cmd

```
/************************************************************/
/***                Specify Linker Options                ***/
/************************************************************/
-e coeff                    /* Define the program entry point */
-o demo.out                 /* Name the output file           */
-m demo.map                 /* Create an output map           */

/************************************************************/
/***                 Specify the Input Files              ***/
/************************************************************/

demo.obj
fft.obj
tables.obj

/************************************************************/
/***           Specify the Memory Configurations          ***/
/************************************************************/

MEMORY
{
        PAGE 0: RAM_PG:  origin=00080h   length=06F80h
                   ROM:  origin=0C000h   length=03F80h

        PAGE 1: ONCHIP:  origin=00080h   length=0F7Fh
                   EXT:  origin=01000h   length=0EFFFh
}

/************************************************************/
/***             Specify the Output Sections              ***/
/************************************************************/

SECTIONS
{
        .text: load = ROM, page = 0    /*  link .text into ROM */

        var_defs: load = ONCHIP, page=1 /*     defs in RAM     */

        .data: fill = 07A1Ch, load=ONCHIP, page=1
         {
                tables.obj(.data) /* .data input */
                fft.obj(.data)    /* .data input */
                . = 100h;     /* create hole, fill with 07A1Ch */
         }                    /* and link with ONCHIP          */

        .bss: load=RAM_PG,page=0,fill=0FFFFh
                              /* Remaining .bss; fill and link */
}

/************************************************************/
/***                  End of Command File                 ***/
/************************************************************/
```

次のコマンドを使用してリンカを起動します。

```
lnk500 demo.cmd
```

これにより、例 7−15 に示したマップ・ファイルと、TMS320C54x 上で実行できる demo.out という出力ファイルが作成されます。

例 7−15. 出力マップ・ファイル、demo.map

```
OUTPUT FILE NAME:   <demo.out>
ENTRY POINT SYMBOL: 0

MEMORY CONFIGURATION
        name    origin      length     used      attributes  fill
        ------  ----------  ---------  --------  ----------  --------
PAGE 0: RAM_PG  00000080    000006f80  00000002  RWIX
        ROM     0000c000    000003f80  00000011  RWIX
PAGE 1: ONCHIP  00000080    000000f7f  00000105  RWIX
        EXT     00001000    00000efff  00000000  RWIX

SECTION ALLOCATION MAP
 output                                  attributes/
section   page  origin      length       input sections
--------  ----  ----------  ----------   ----------------
.text      0    0000c000    00000011
                0000c000    00000008     demo.obj (.text)
                0000c008    00000004     fft.obj (.text)
                0000c00c    00000005     tables.obj (.text)
var_defs   1    00000080    00000002
                00000080    00000002     demo.obj (var_defs)
.data      1    00000082    00000100
                00000082    00000001     tables.obj (.data)
                00000083    00000004     fft.obj (.data)
                00000087    000000fb     --HOLE-- [fill = 7a1c]
                00000182    00000000     demo.obj (.data)
.bss       0    00000080    00000002
                00000080    00000002     demo.obj (.bss) [fill=ffff]
                00000082    00000000     tables.obj (.bss)
                00000082    00000000     fft.obj (.bss)
xy         1    00000182    00000003     UNINITIALIZED
                00000182    00000003     demo.obj (xy)

GLOBAL SYMBOLS
address  name                             address  name
-------- ----                             -------- ----
00000080 .bss                             00000080 .bss
00000082 .data                            00000082 .data
0000c000 .text                            00000082 TEMP
0000c002 ARRAY                            0000c002 ARRAY
00000082 TEMP                             00000182 end
00000182 edata                            0000018a edata
00000082 end                              0000c000 .text
0000c011 etext                            0000c011 etext

[8 symbols]
```

第 8 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。これを手動で行うと、多数の操作を要する繁雑なプロセスが必要になります。しかし、絶対リスタ・ユーティリティを使用すると、このような操作は自動的に行われます。

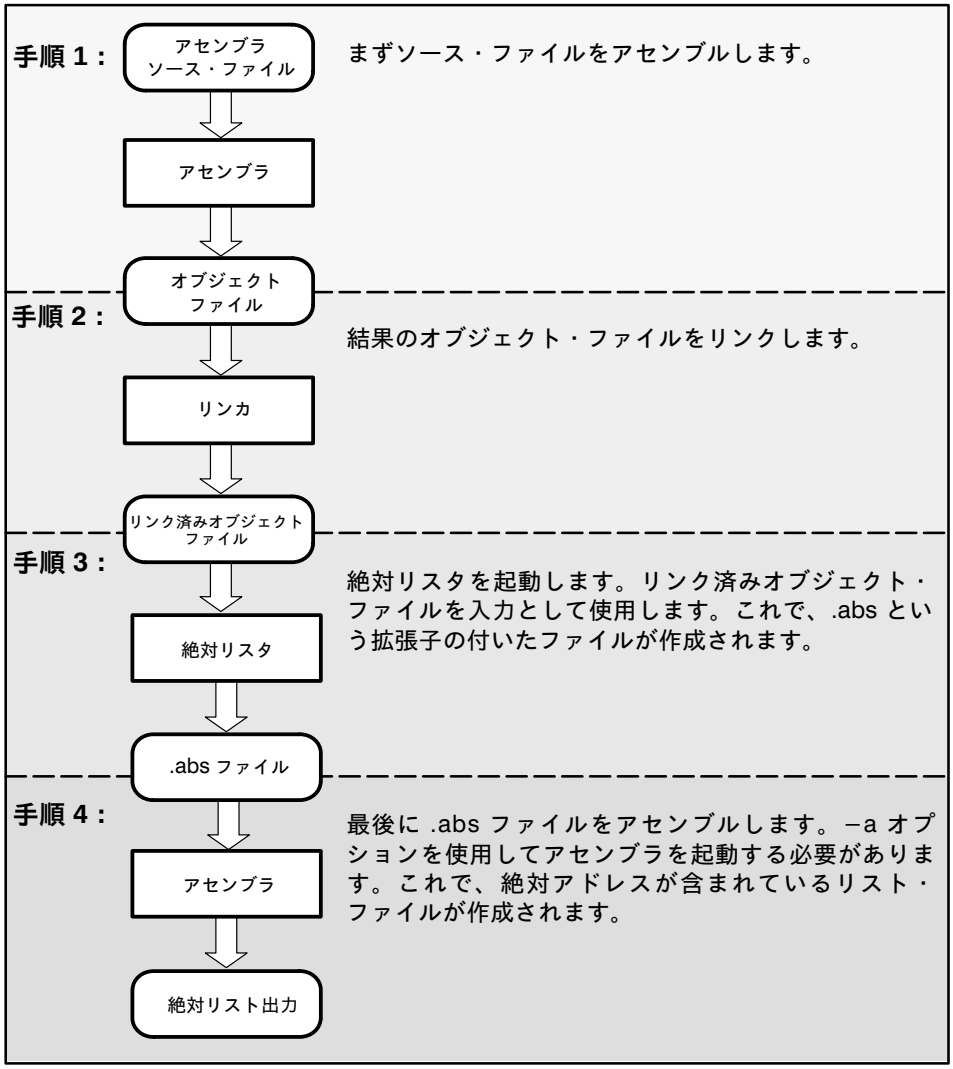
項目 **ページ**

8.1 絶対リスト出力の作成方法 8-2
8.2 絶対リスタの起動方法 8-3
8.3 絶対リスタの例 8-5

8.1 絶対リスト出力の作成方法

図 8–1 は、絶対リスト出力を作成するために必要な手順を示しています。

図 8–1. 絶対リスタと開発のフロー



8.2 絶対リスタの起動方法

絶対リスタを起動するには、次のように入力します。

abs500 [–*options*] *input file*

abs500 絶対リスタを起動するコマンドです。

options 使用する絶対リスタ・オプションを識別します。オプションでは大文字と小文字は区別されません。オプションは、コマンドの後なら、そのコマンド行のどこにでも指定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 アセンブリ・ファイル、C ソース・ファイル、および C ヘッダ・ファイルのファイル名拡張子についてのデフォルトの命名規則を変更できます。3 つのオプションを以下に示します。

- –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・ファイル用 (デフォルトは .h)

拡張子の「.」とオプションと拡張子の間の空白は、指定してもしなくてもかまいません。

–q (静的な実行) 見出しとすべての進捗情報を出力しません。

input file リンク済みオブジェクト・ファイルの名前を指定します。拡張子を指定しなければ、絶対リスタは、入力ファイルがデフォルトの拡張子 .out をもつものと解釈します。絶対リスタの起動時に入力ファイル名を指定しないと、絶対リストは入力を求めるプロンプトを表示します。

絶対リスタは、リンクされた各ファイルにつき 1 つの出力ファイルを作成します。出力ファイルの名前は、入力ファイルの名前に拡張子 .abs を付けたものとなります。ただし、ヘッダ・ファイルについては、対応する .abs ファイルは作成されません。

絶対リスト出力を作成するには、以下のような –a アセンブラ・オプションを使用して、上記の出力ファイルをアセンブルします。

```
asm500 –a filename.abs
```

–e オプションは、コマンド行のファイル名の解釈、および出力ファイルの名前の両方に作用します。これらのオプションは必ず、コマンド行においてすべてのファイル名の前に指定する必要があります。

−e オプションは、デバッグ・オプション (−g コンパイラ・オプション) を指定してコンパイルした C ファイルから作成された、リンク済みオブジェクト・ファイルに使用すると便利です。デバッグ・オプションを設定すると、結果のリンク済みオブジェクト・ファイルには、その作成に使用したソース・ファイルの名前が組み込まれます。この場合、絶対リスタは、C ヘッダ・ファイルについては対応する .abs ファイルを作成しません。また、C ソース・ファイルに対応する .abs ファイルは、C ソース・ファイルそのものでなく、C ソース・ファイルから作成されたアセンブリ・ファイルを使用します。

たとえば、C ソース・ファイル hello.csr をデバッグ・オプションを指定してコンパイルするとします。この場合、アセンブリ・ファイル hello.s が作成されます。hello.csr には hello.hsr も組み込まれています。作成された実行可能ファイルの名前が hello.out だとすれば、次のコマンドにより適正な .abs ファイルを作成できます。

```
abs500 –ea s –ec csr –eh hsr hello.out
```

hello.hsr (ヘッダ・ファイル) については .abs ファイルは作成されません。また、hello.abs には、C ソース・ファイル hello.csr ではなく、アセンブリ・ファイル hello.s に対する絶対リストが含まれます。

8.3 絶対リスタの例

この例では、3 つのソース・ファイルを使用しています。module1.asm と module2.asm の両方に、ファイル globals.def が含まれています。

module1.asm

```
.text
.bss   array,100
.bss   dflag, 2
.copy  globals.def
ld     #offset, A
ld     dflag, A
```

module2.asm

```
.bss   offset, 2
.copy  globals.def
ld     #offset, A
ld     #array, A
```

globals.def

```
.global dflag
.global array
.global offset
```

以下の手順で、module1.asm と module2.asm の 2 つのファイルについての絶対リスト出力を作成します。

手順 1: まず、module1.asm と module2.asm をアセンブルします。

```
asm500 module1
asm500 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、bttest.cmd という名前の以下のようなリンカ・コマンド・ファイルを使用して、module1.obj と module2.obj をリンクします。

```
/************************************************/
/* File bttest.cmd -- COFF linker command file  */
/*       for linking TMS320C54x modules         */
/********************************** ************/
-o bttest.out           /* Name the output file */
-m bttest.map           /* Create an output map */

/************************************************/
/*            Specify the Input Files           */
/************************************************/
module1.obj
module2.obj

/************************************************/
/*       Specify the Memory Configurations      */
/************************************************/
MEMORY
{
   PAGE 0:   ROM:  origin=2000h   length=2000h
   PAGE 1:   RAM:  origin=8000h   length=8000h
}

/************************************************/
/*         Specify the Output Sections          */
/************************************************/
SECTIONS
{
   .data:  >RAM PAGE 1
   .text:  >ROM PAGE 0
   .bss:   >RAM PAGE 1
}
```

リンカを起動します。

lnk500 bttest.cmd

これで、bttest.out という名前の実行可能オブジェクト・ファイルが作成されます。この新しいファイルを、絶対リスタの入力として使用します。

手順 3: 絶対リスタを起動します。

```
abs500 bttest.out
```

これで、module1.abs と module2.abs という 2 つのファイルが作成されます。

module1.abs:

```
          .nolist
array     .setsym     08000h
dflag     .setsym     08064h
offset    .setsym     08066h
.data     .setsym     08000h
edata     .setsym     08000h
.text     .setsym     02000h
etext     .setsym     02007h
.bss      .setsym     08000h
end       .setsym     08068h
          .setsect    ".text",02000h
          .setsect    ".data",08000h
          .setsect    ".bss",08000h
          .list
          .text
          .copy       "module1.asm"
```

module2.abs:

```
          .nolist
array     .setsym     08000h
dflag     .setsym     08064h
offset    .setsym     08066h
.data     .setsym     08000h
edata     .setsym     08000h
.text     .setsym     02000h
etext     .setsym     02007h
.bss      .setsym     08000h
end       .setsym     08068h
          .setsect    ".text",02003h
          .setsect    ".data",08000h
          .setsect    ".bss",08066h
          .list
          .text
          .copy       "module2.asm"
```

この 2 つのファイルには、手順 4 でアセンブラを起動するときにアセンブラが必要とする以下の情報が入っています。

- .setsym 疑似命令。これは、値をグローバル・シンボルと等価にします。どちらのファイルにも、シンボル *dflag* 用のグローバルな等価値が含まれています。シンボル dflag は、module1.asm と module2.asm. に組み込まれているファイル globals.def の中で定義されています。
- .setsect 疑似命令。これは、セクションの絶対アドレスを定義しています。
- .copy 疑似命令。これは、どのアセンブリ言語ソース・ファイルを組み込むかをアセンブラに指示します。

.setsym 疑似命令と .setsect 疑似命令は、通常のアセンブリで使用しても役に立ちません。この 2 つの疑似命令が役に立つのは、絶対リスト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 (アセンブラを起動するときに –a オプションを使用しなければならないことを忘れないでください)。

```
asm500  -a  module1.abs
asm500  -a  module2.abs
```

これで、module1.lst と module2.lst という 2 つのリスト・ファイルが作成されます。オブジェクト・コードは生成されません。これらのリスト・ファイルは、通常のリスト・ファイルに似ていますが、表示されているアドレスは絶対アドレスです。

module1.lst (図 8–2 参照) と、module2.lst (図 8–3 参照) という 2 つの絶対リスト・ファイルが作成されます。

図 8−2. module1.lst

```
TMS320C54x COFF Assembler       Version x.xx     Wed Oct 16 12:00:05 2001
 Copyright (c) 2001    Texas Instruments Incorporated

module1.abs                                                          PAGE    1

      22 002000              .text
      23                     .copy       ”module1.asm”
 A     1 002000              .text
 A     2 008000              .bss   array, 100
 A     3 008064              .bss   dflag, 2
 A     4                     .copy  globals.def
 B     1                     .global dflag
 B     2                     .global array
 B     3                     .global offset
 A     5 002000 F020         ld     #offset, A
         002001 8066!
 A     6 002002 1064–        ld     dflag, A

 No Errors, No Warnings
```

図 8−3. module2.lst

```
TMS320C54x COFF Assembler       Version x.xx     Wed Oct 16 12:00:17 2001
 Copyright (c) 2001    Texas Instruments Incorporated

module2.abs                                                          PAGE    1

      22 002003              .text
      23                     .copy       ”module2.asm”
 A     1 008066              .bss   offset, 2
 A     2                     .copy  globals.def
 B     1                     .global dflag
 B     2                     .global array
 B     3                     .global offset
 A     3 002003 F020         ld     #offset, A
         002004 8066–
 A     4 002005 F020         ld     #array, A
         002006 8000!

 No Errors, No Warnings
```

第 9 章

クロスリファレンス・リスタの説明

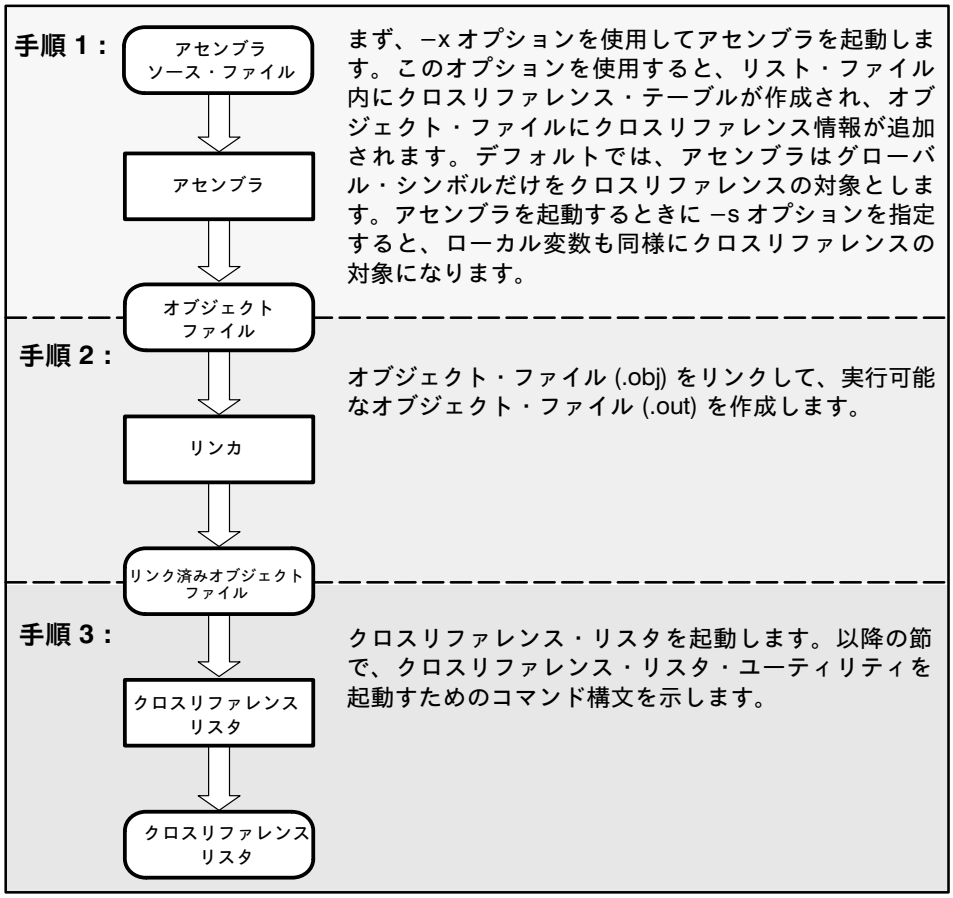
クロスリファレンス・リスタはデバッグ・ツールです。このユーティリティは、リンク済みオブジェクト・ファイルを入力として受け入れ、クロスリファレンス・リストを出力として作成します。このリストには、シンボル、シンボルの定義、リンクされたソース・ファイル内でのシンボルの参照が示されています。

項目 **ページ**

9.1 クロスリファレンス・リストの作成方法 9-2
9.2 クロスリファレンス・リスタの起動方法 9-3
9.3 クロスリファレンス・リストの例 9-4

9.1 クロスリファレンス・リストの作成方法

図 9−1. クロスリファレンス・リスタと開発フロー



9.2 クロスリファレンス・リスタの起動方法

クロスリファレンス・ユーティリティを使用するには、正しいオプションを指定してファイルをアセンブルし、リンクして実行可能ファイルにする必要があります。–x オプションを使用してアセンブリ言語ファイルをアセンブルします。このオプションを指定すると、クロスリファレンス・リストが作成され、クロスリファレンス情報がオブジェクト・ファイルに追加されます。デフォルトでは、アセンブラがクロスリファレンスの対象にするのはグローバル・シンボルだけですが、–s オプションを使用してアセンブラを起動した場合、ローカル・シンボルも追加されます。実行可能ファイルを作成するには、オブジェクト・ファイルをリンクします。

クロスリファレンス・リスタを起動するには、以下のように入力します。

xref500 [–*options*] [*input filename* [*output filename*]]

xref500 クロスリファレンス・ユーティリティを起動するコマンドです。

options 使用するクロスリファレンス・リスタ・オプションを識別します。オプションでは大文字と小文字は区別されません。オプションは、コマンドの後なら、そのコマンド行のどこにでも指定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。クロスリファレンス・リスタ・オプションは以下のとおりです。

- **–l** (L の小文字) 出力ファイルの 1 ページ当りの行数を指定します。–l オプションのフォーマットは –l*num*で、num は 10 進定数です。たとえば、–l30 を指定すると、出力ファイルの 1 ページ当りの行数は 30 行に設定されます。オプションと 10 進定数との間には、空白を入れても入れなくてもかまいません。デフォルトは 1 ページ当り 60 行です。
- **–q** (静的な実行) 見出しとすべての進捗情報を出力しません。

input filename リンク済みオブジェクト・ファイルです。入力ファイル名を省略すると、ユーティリティはファイル名の入力を求めるプロンプトを表示します。

output filename クロスリファレンス・リスト・ファイルの名前です。出力ファイル名を省略すると、デフォルトのファイル名は、入力ファイルの名前に拡張子 .xrf を付けたものになります。

9.3 クロスリファレンス・リストの例

```
================================================================================

Symbol: INIT

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____   ________   ________    ______  _______  _______  _______
file1.asm        EDEF   '000000    000080      3       1
file2.asm        EREF    000000    000080              2         11
================================================================================

Symbol: X

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____   ________   ________    ______  _______  _______  _______
file1.asm        EREF    000000    000001              2         5
file2.asm        EDEF    000001    000001      5       1
================================================================================

Symbol: Y

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____   ________   ________    ______  _______  _______  _______
file2.asm        EDEF    -000000   000080      7       1

================================================================================

Symbol: Z

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____   ________   ________    ______  _______  _______  _______
file2.asm        EDEF    000003    000003      9       1

================================================================================
```

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 リストされているシンボルの名前

Filename シンボルが使用されているファイルの名前

RTYP このファイル内でのシンボルの参照タイプ。参照タイプには以下のものがあります。

- **STAT** シンボルはこのファイルで定義されていて、グローバルとして宣言されていません。
- **EDEF** シンボルはこのファイルで定義されていて、グローバルとして宣言されています。
- **EREF** シンボルはこのファイルで定義されていませんが、グローバルとして参照されています。
- **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。表 9−1 に、このような文字と意味を掲載しています。

LnkVal この 16 進数は、リンク後にシンボルに割り当てられた値です。

DefLn シンボルが定義されている文の行番号です。

RefLn シンボルが参照されている文の行番号です。行番号の後にアスタリスク (*) が付いている場合は、参照によりオブジェクトの内容が変更される可能性があります。行番号の後に文字 (A、B、C など) が付いている場合は、アセンブリ・ソースの .include 疑似命令で指定されているファイルでこのシンボルが参照されています。「A」は、.include 疑似命令で 1 番目に指定されているファイルに割り当てられており、「B」は、2 番目に指定されているファイルに割り当てられていて、以降も同様です。この欄が空白であれば、このシンボルが一度も使用されなかったことを示します。

表 9−1.　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 10 章

Hex 変換ユーティリティの説明

TMS320C54xのアセンブラおよびリンカは、共通オブジェクト・ファイル・フォーマット (COFF) のオブジェクト・ファイルを作成します。COFF は、モジュラ・プログラミングを促進するとともに、コード・セグメントおよびターゲット・システム・メモリを管理するための柔軟な方式を提供する 2 進オブジェクト・ファイル・フォーマットです。

EPROM プログラマの多くは、COFF オブジェクト・ファイルを入力として受け入れません。Hex 変換ユーティリティは、COFF オブジェクト・ファイルを、EPROM プログラマへのロードに適したいくつかの標準 ASCII 16 進フォーマットのいずれかに変換します。このユーティリティは、COFF オブジェクト・ファイルの 16 進変換を必要とするその他のアプリケーション (デバッガやローダなど) を使用する場合にも役立ちます。このユーティリティは、ターゲット・デバイスに組み込まれているオンチップ・ブート・ローダもサポートしているため、'C54x のコード生成プロセスは自動化されます。

Hex 変換ユーティリティでは、次の出力ファイル・フォーマットを作成できます。

- ❑ 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、24 ビット、および 32 ビットの各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弊社の SDSMAC (TI−Tagged)

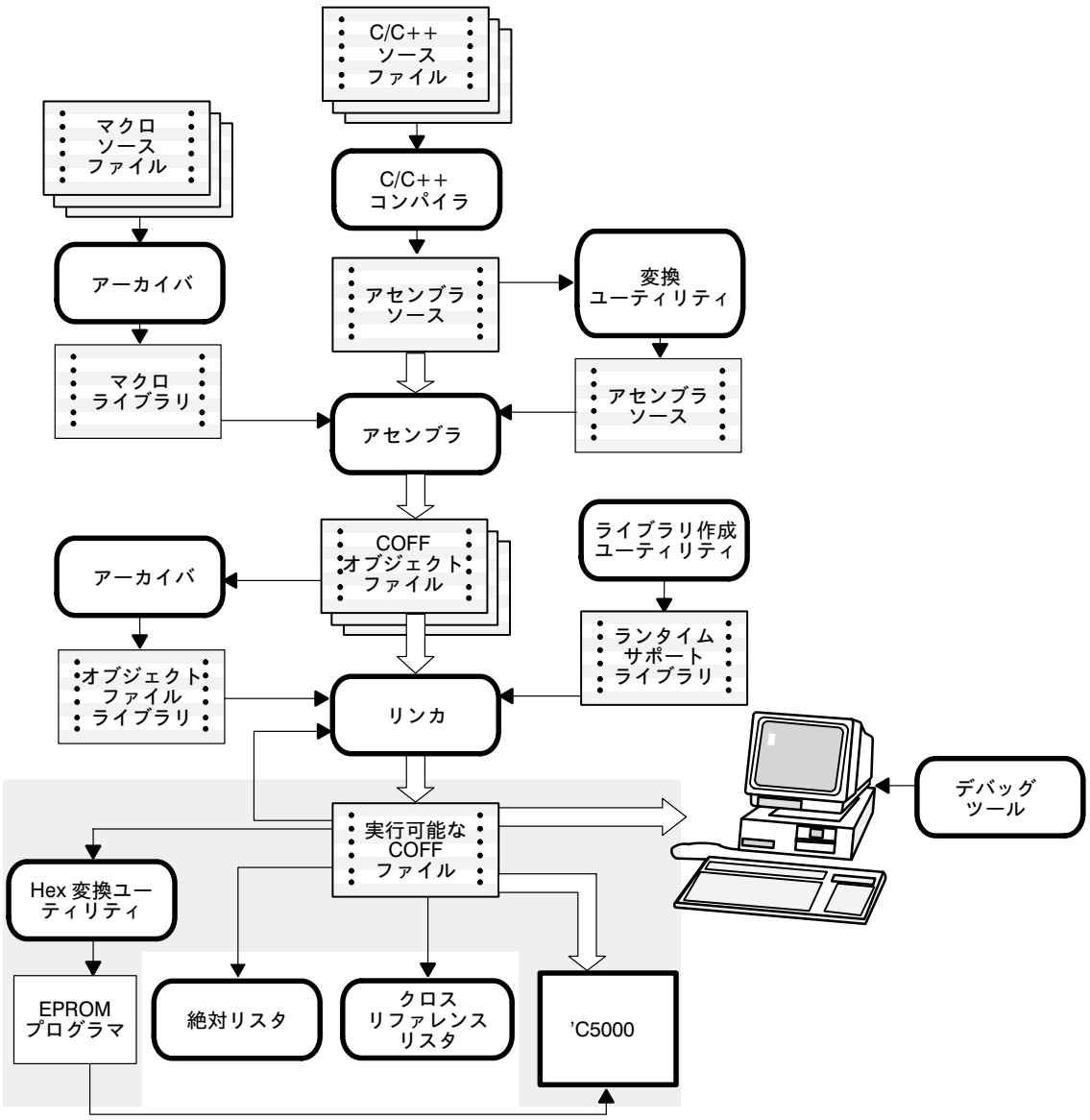
項目 **ページ**

10.1 Hex 変換ユーティリティと開発フロー 10-2
10.2 Hex 変換ユーティリティの起動方法 10-3
10.3 コマンド・ファイル 10-7
10.4 メモリ幅について 10-9
10.5 ROMS 疑似命令 10-16
10.6 SECTIONS 疑似命令 10-22
10.7 出力ファイル名 10-24
10.8 イメージ・モードと –fill オプション 10-26
10.9 オンチップ・ブート・ローダ用のテーブルの作成方法 10-28
10.10 ROM デバイス・アドレスの制御方法 10-34
10.11 オブジェクト・フォーマットの説明 10-38
10.12 Hex 変換ユーティリティのエラー・メッセージ 10-44

10.1 Hex 変換ユーティリティと開発フロー

図 10−1 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 10−1. Hex 変換ユーティリティと開発フロー



10.2 Hex 変換ユーティリティの起動方法

Hex 変換ユーティリティを起動するには、次の 2 つの基本的な方法があります。

- **コマンド行でオプションとファイル名を指定する方法。**次の例では、ファイル firmware.out を TI−Tagged フォーマットに変換し、firm.lsb と firm.msb という 2 つの出力ファイルを作成します。

  ```
  hex500 –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方法。**Hex 変換ユーティリティを起動するためのコマンド行オプションとファイル名を格納したバッチ・ファイルを作成できます。次の例では、hexutil.cmd というコマンド・ファイルを使用してユーティリティを起動しています。

  ```
  hex500 hexutil.cmd
  ```

  コマンド・ファイルの中では、通常のコマンド行情報のほかに、Hex 変換ユーティリティの ROMS 疑似命令と SECTIONS 疑似命令を使用できます。

Hex 変換ユーティリティを起動するには、次のように入力します。

hex500 [*–options*] *filename*

hex500 Hex 変換ユーティリティを起動するコマンドです。

–options Hex 変換ユーティリティの処理を制御する追加情報を指定します。オプションは、コマンド行で使用できるほか、コマンド・ファイルの中でも使用できます。

- オプションの前には必ずダッシュを付けます。また、どのオプションも大文字と小文字は区別されません。
- 一部のオプションには追加のパラメータがあります。これらのパラメータは、1 つ以上の空白でオプションから区切る必要があります。
- 複数文字のオプション名は、本書に記載されているとおり正確に指定する必要があります。省略形は使用できません。
- オプションの指定順序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。

filename COFF オブジェクト・ファイルまたはコマンド・ファイルの名前を指定します (コマンド・ファイルの詳細は、10-7 ページの 10.3 節「コマンド・ファイル」を参照してください)。

表 10−1. Hex 変換ユーティリティのオプション

(a) 一般オプション

一般オプションは、Hex 変換ユーティリティの動作全体を制御します。

オプション	説明	ページ
−byte	バイトを順にカウントします。	10-36
−map *filename*	マップ・ファイルを作成します。	10-21
−o *filename*	出力ファイル名を指定します。	10-24
−q	静的な実行 (使用するときは他のオプションより前に指定する必要があります)。	10-7

(b) イメージ・オプション

イメージ・オプションは、ある範囲のターゲット・メモリを連続したイメージとして作成します。

オプション	説明	ページ
−fill *value*	ホールに値を埋め込みます。	10-27
−image	イメージ・モードを指定します。	10-26
−zero	アドレス起点をゼロにリセットします。	10-35

(c) メモリ・オプション

メモリ・オプションは、出力ファイルのメモリ幅を設定します。

オプション	説明	ページ
−memwidth *value*	システム・メモリのワード幅を定義します (デフォルト値 = 16 ビット)。	10-10
−order {LS | MS}	メモリ・ワードの順番を指定します。	10-14
−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)。	10-11

(d) 出力フォーマット

出力フォーマットは、出力ファイルのフォーマットを指定します。

オプション	説明	ページ
−a	ASCII−Hex を選択します。	10-39
−i	Intel を選択します。	10-40
−m1	Motorola−S1 を選択します。	10-41
−m2 または −m	Motorola−S2 を選択します (デフォルト)。	10-41
−m3	Motorola−S3 を選択します。	10-41
−t	TI−Tagged を選択します。	10-42
−x	Tektronix を選択します。	10-43

(e) すべての ’C54x デバイス用のブートローダ・オプション

すべての ’C54x デバイス用のブートローダ・オプションは、Hex 変換ユーティリティを使用してブート・テーブルを作成する方法を制御します。

オプション	説明	ページ
−boot	全セクションをブート可能な形式に変換します (SECTIONS 疑似命令の代わりに使用します)。	10-29
−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。	10-29
−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。	10-29
−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。	10-29
−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。	10-29
−e *value*	ブート・ロード後に実行を始めるエントリ・ポイントを指定します。*value* は、アドレスまたはグローバル・シンボルになります。	10-29

(f) 'C54x LP デバイス専用のブートローダ・オプション

オプション	説明	ページ
−bootorg WARM または −warm	ブート・ローダ・テーブルのソースに、現在メモリ内にあるテーブルを指定します。	10-29
−bootorg COMM	ブート・ローダ・テーブルのソースに通信ポートを指定します。	10-29
−spc *value*	シリアル・ポート制御レジスタの値を設定します。	10-29
−spce *value*	シリアル・ポート制御拡張レジスタの値を設定します。	10-29
−arr *value*	ABU 受信アドレス・レジスタ値を設定します。	10-29
−bkr *value*	ABU 送信バッファ・サイズ・レジスタの値を設定します。	10-29
−tcsr *value*	TDM シリアル・ポート・チャネル選択レジスタの値を設定します。	10-29
−trta *value*	TDM シリアル・ポート送受信アドレス・レジスタの値を設定します。	10-29
−swwsr *value*	PARALLEL/WARM ブート・モードのソフトウェア・ウェイト・ステート・レジスタの値を設定します。	10-29
−bscr *value*	PARALLEL/WARM ブート・モードのバンク・スイッチ制御レジスタの値を設定します。	10-29

10.3 コマンド・ファイル

同じ入力ファイルとオプションを指定して、このユーティリティを繰り返し起動する場合は、コマンド・ファイルを使用すると便利です。また、ROMS および SECTIONS Hex 変換ユーティリティ疑似命令を使用して変換処理をカスタマイズする場合も、コマンド・ファイルが役立ちます。

コマンド・ファイルは、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全く同じ方法でコマンド・ファイル内に指定します。
- **ROMS 疑似命令。** ROMS 疑似命令は、システムの物理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定義します (ROMS 疑似命令の詳細は、10-16 ページの 10.5 節「ROMS 疑似命令」を参照してください)。
- **SECTIONS 疑似命令。** SECTIONS 疑似命令は、COFF オブジェクト・ファイルからどのセクションを選択するかを指定します (SECTIONS 疑似命令の詳細は、10-22 ページの 10.6 節「SECTIONS 疑似命令」を参照してください)。

  この疑似命令を使用すると、オンチップ・ブート・ローダで初期化する特定のセクションを識別することもできます (オンチップ・ブート・ローダの詳細は、10-28 ページの 10.9 節「オンチップ・ブート・ローダ用のテーブルの作成方法」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。

```
hex500 command_filename
```

コマンド行には、その他のファイルおよびオプションも指定できます。コマンド・ファイルとコマンド行オプションの両方を使用して、次のようにユーティリティを起動することもできます。

```
hex500 firmware.cmd –map firmware.mxp
```

これらのオプションおよびファイル名を指定する順序に、意味はありません。ユーティリティは、変換処理を開始する前に、コマンド行からすべての入力を読み込むとともに、コマンド・ファイルからすべての情報を読み込みます。ただし、–q オプションを使用する場合は、–q オプションをコマンド行またはコマンド・ファイルの最初のオプションとして指定する必要があります。

–q オプションは、このユーティリティの通常の見出しおよび進捗情報の表示を抑止します。

10.3.1 コマンド・ファイルの例

- firmware.cmd という名前のコマンド・ファイルに以下の行が含まれているとします。

```
firmware.out   /* input file  */
-t             /* TI-Tagged   */
-o   firm.lsb  /* output file 1, LSBs of ROM */
-o   firm.msb  /* output file 2, MSBs of ROM*/
```

以下のように入力すれば、Hex 変換ユーティリティを起動することができます。

```
hex500 firmware.cmd
```

- この例では、appl.out という名前のファイルを Intel フォーマットの 4 つの 16 進ファイルに変換します。各出力ファイルの幅は 1 バイトで、長さは 16K バイトです。.text セクションはブート・ローダ・フォーマットに変換されます。

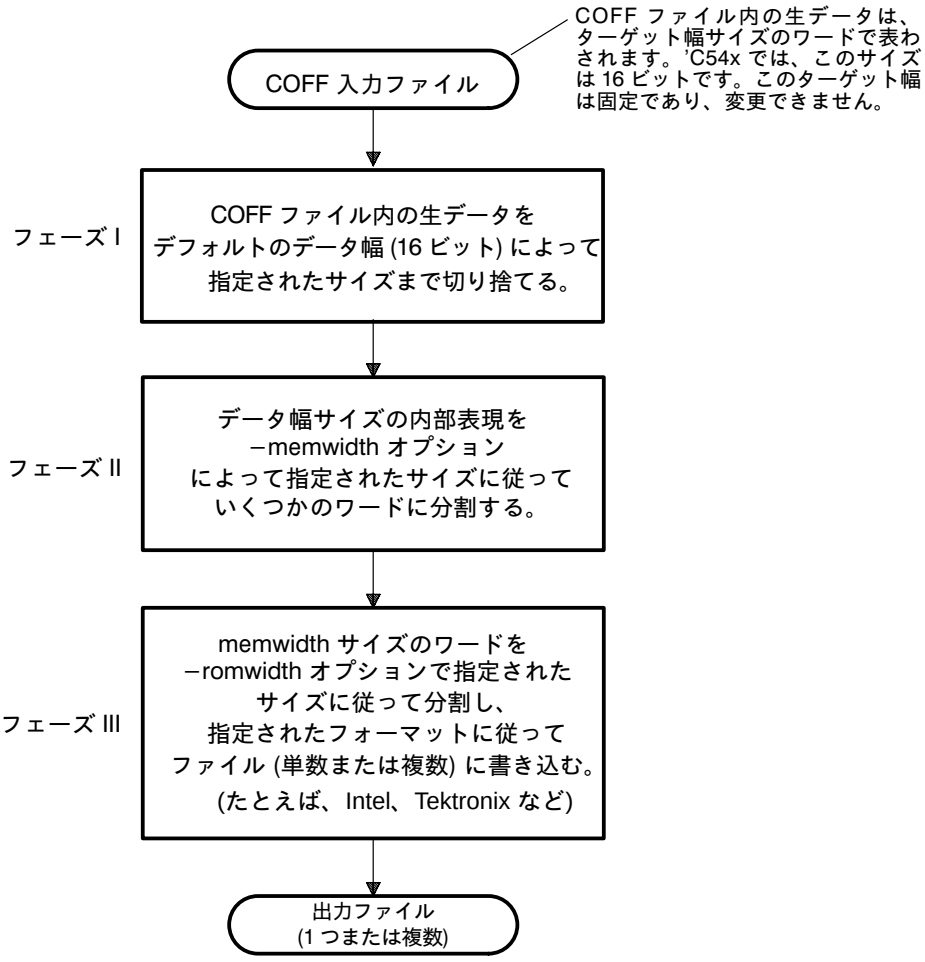
```
appl.out         /* input file   */
-i               /* Intel format */
-map appl.mxp    /* map file     */

ROMS
{
   ROW1: origin=01000h len=04000h romwidth=8
         files={ appl.u0 appl.u1 }
   ROW2: origin 05000h len=04000h romwidth=8
         files={ app1.u2 appl.u3 }
}

SECTIONS
{
    .text: BOOT
    .data, .cinit, .sect1, .vectors, .const:
}
```

10.4 メモリ幅について

Hex 変換ユーティリティは、ユーザ自身がメモリ幅と ROM 幅を指定することを認めることによりユーザのメモリ・アーキテクチャをより柔軟にします。Hex 変換ユーティリティを使用するには、ユーティリティがワード幅をどのように扱うかを理解しておく必要があります。変換プロセスでは、ターゲット幅、データ幅、メモリ幅、および ROM 幅の 4 つの幅が重要な意味をもちます。ターゲット・ワード、データ・ワード、メモリ・ワード、および ROM ワードという用語は、このような幅をもつワードを意味します。

図 10−2 は、Hex 変換ユーティリティの処理フローにおける 3 つの独立した個別フェーズを示したものです。

図 10−2. Hex 変換ユーティリティの処理フロー



10.4.1 ターゲット幅

ターゲット幅は、COFF ファイル内の生データ・フィールドの単位サイズ (ビット数) です。これは、ターゲット・プロセッサ上の命令コードのサイズに対応します。この幅はターゲットごとに固定であり、変更することはできません。C54x のターゲット幅は 16 ビットです。

10.4.2 データ幅

データ幅は、COFF ファイルの特定のセクションに格納されるデータ・ワードの論理幅 (ビット数) です。通常、論理データ幅はターゲット幅と同じになります。TMS320C54x 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。

10.4.3 メモリ幅

メモリ幅は、メモリ・システムの物理的な幅 (ビット数) です。通常、メモリ・システムは、ターゲット・プロセッサの幅と物理的に同じ幅になります。つまり、16 ビットのプロセッサであれば、16 ビットのメモリ・アーキテクチャをもちます。ただし、一部のアプリケーションでは、ターゲット・ワードを分割して、連続した複数のより狭い幅のメモリ・ワードを作成する必要があります。さらに、'C54x のような特定のプロセッサでは、メモリ幅はターゲット幅より狭くなることがあります。

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、次の方法で変更できます。

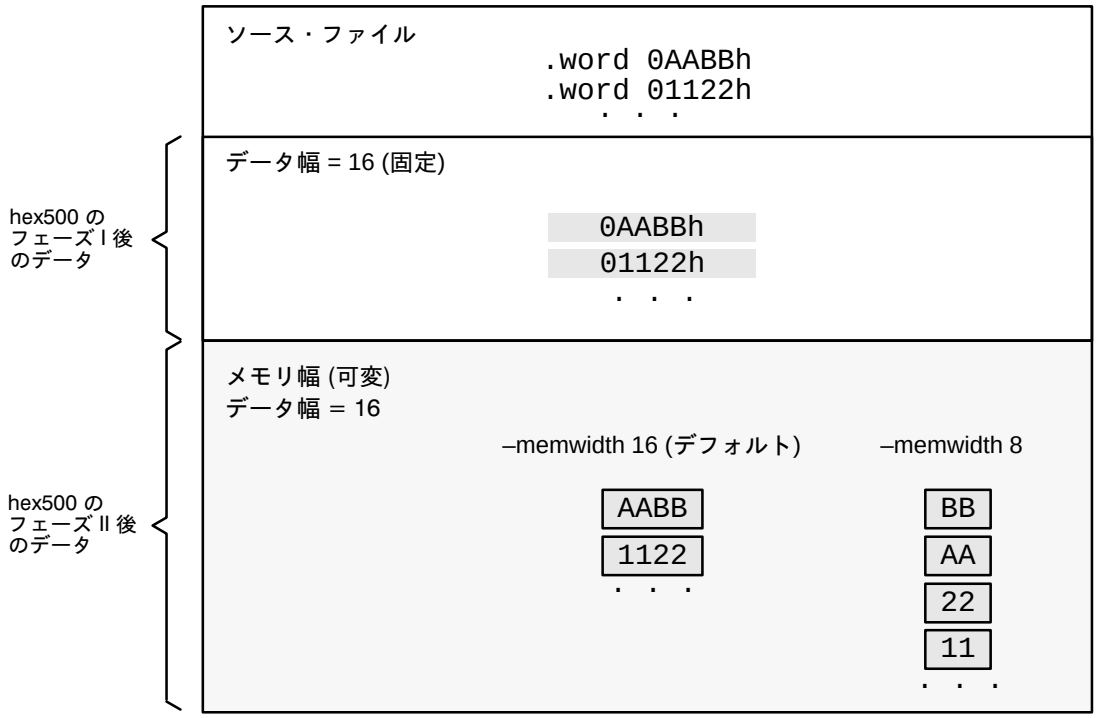
- **−memwidth** オプションを使用する方法。この方法では、ファイル全体についてメモリ幅が変更されます。
- ROMS 疑似命令の **memwidth** パラメータを設定する方法。この方法では、ROMS 疑似命令で指定したアドレス範囲についてメモリ幅が変更されます。そのアドレス範囲に対する −memwidth オプションは無効になります。10-16 ページの 10.5 節「ROMS 疑似命令」を参照してください。

どちらの場合も、使用する値は、8 以上の 2 の累乗値でなければなりません。

メモリ幅のデフォルト値である 16 は、たとえば 1 つのターゲット・ワードをそれより狭い幅の連続した複数のメモリ・ワードに分割する必要がある場合などの、例外的な状況がある場合以外は変更しないでください。メモリ・ワードがターゲット・ワードより狭いという状況がよく起こるのは、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・ワードよりも狭い幅をもつメモリ・ワードからのブートをサポートするものがあります。たとえば、16 ビットの TMS320C54x 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、それぞれの 16 ビット値は 2 つのメモリ・ロケーションを占めます (これは、−memwidth 8 として指定されます)。

図 10−3 は、メモリ幅がデータ幅とどのように関連するかを示しています。

図 10−3. データ幅とメモリ幅



10.4.4 ROM 幅

ROM 幅は、各 ROM デバイスと対応する出力ファイルの物理的な幅 (ビット数) を指定します (通常は 1 バイト、つまり 8 ビットです)。Hex 変換ユーティリティがデータをどのように分割して出力ファイルに入れるかは、ROM 幅によって決まります。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 つ以上の出力ファイルに分割されます。アドレス領域ごとの出力ファイルの数は、次の式で求めることができます。ただし、メモリ幅 ≧ ROM 幅です。

ファイル数 = メモリ幅 ÷ ROM 幅

たとえば、メモリ幅が 16 である場合に、ROM 幅の値として 16 を指定すると、16 ビットの各ワードが格納された出力ファイルが 1 つ作成されます。または、ROM 幅の値として 8 を使用して、2 つの出力ファイルを作成し、それぞれのファイルに 8 ビットの各ワードを格納することもできます。

アドレス領域ごとのファイル数の計算については、10-16 ページの 10.5 節「ROMS 疑似命令」を参照してください。

Hex 変換ユーティリティが使用するデフォルトの ROM 幅は、出力フォーマットによって次のように異なります。

- TI−Tagged フォーマットを除くすべての Hex フォーマットは、8 ビットの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。このため、TI−Tagged 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 フォーマットは、16 ビットの ROM 幅だけをサポートします。

ROM 幅 (TI−Tagged フォーマットの場合を除き) は次の方法で変更することができます。

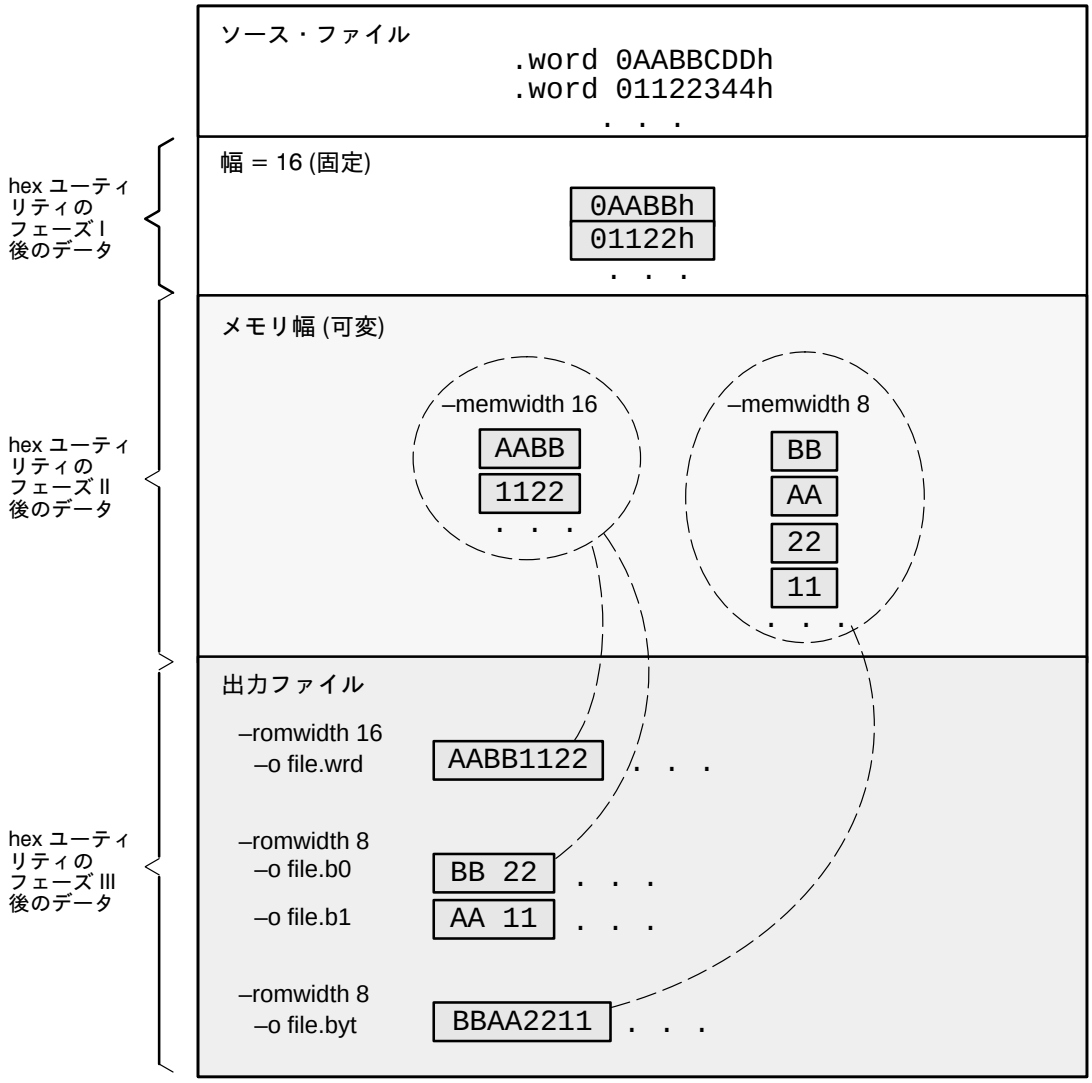
- **−romwidth** オプションを使用する方法。この方法では、COFF ファイル全体について ROM 幅の値が変更されます。
- ROMS 疑似命令の **romwidth** パラメータを設定する方法。この方法では、特定の ROM アドレス範囲について ROM 幅の値が変更されます。そのアドレス範囲に対する −romwidth オプションは無効になります。10-16 ページの 10.5 節「ROMS 疑似命令」を参照してください。

上記のどちらの場合も、8 以上の 2 の累乗値を使用します。

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) より広い ROM 幅を選択する場合、ユーティリティは、単に複数バイトのフィールドをファイルに書き込みます。

図 10−4 は、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示しています。

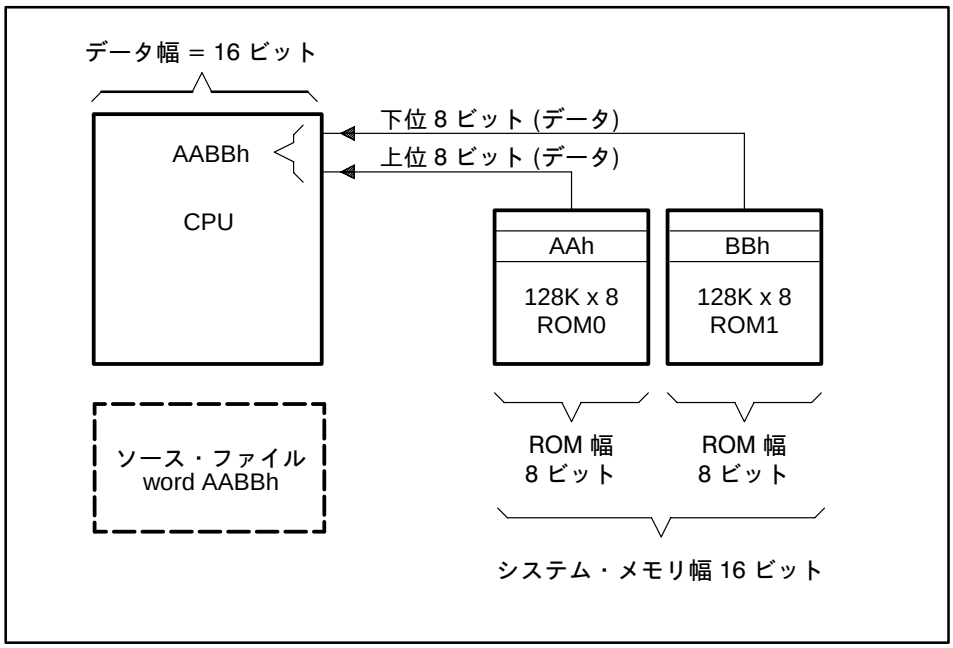
図 10−4. データ幅、メモリ幅、および ROM 幅



10.4.5 メモリ構成の例

図 10−5 は、一般的なメモリ構成の例を示しています。このメモリ・システムは、128K × 8 ビットの 2 つの ROM デバイスから構成されます。

図 10−5. 'C54x のメモリ構成の例



10.4.6 出力ワードのワード配列の指定方法

メモリ・ワードがターゲット・ワードよりも幅が狭い場合 (メモリ幅 < 16)、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**配列を指定する方法。この場合、連続したロケーションの先頭に幅の広いワードの最上位部分が置かれます。
- **−order LS** によって**リトル・エンディアン**配列を指定する方法。この場合、連続したロケーションの先頭に幅の広いワードの最下位部分が置かれます。

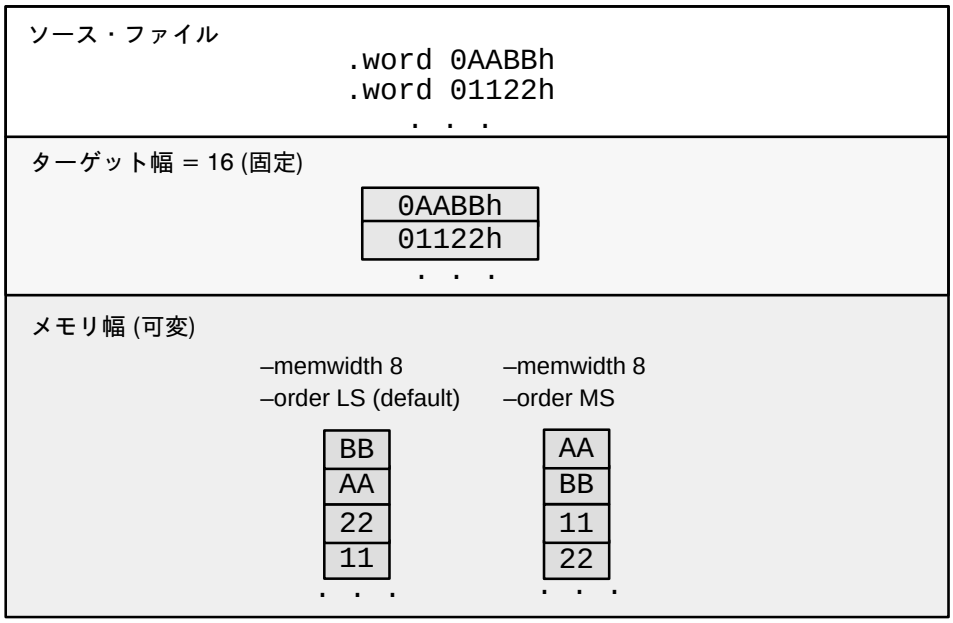
'C54x のブート・ローダはリトル・エンディアン方式を前提としているので、デフォルトでは、Hex 変換ユーティリティはリトル・エンディアン方式を使用します。
−order MS は、独自のブート・ローダ・プログラムを使用している場合以外は使用しないでください。

注: −order オプションを適用する場合

- 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のみ使用します。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・ワードが出力ファイルにどのように分割されるかには影響を与えません。これらのファイルは、1 つの集合であると考えてください。この集合には、最下位のファイルと最上位のファイルが含まれていますが、集合全体について配列が決まっているわけではありません。この集合に含まれるファイルの名前を指定するときには、−order オプションとは関係なく、必ず最下位のファイルの名前を最初に指定します。

図 10−6 は、−order オプションが変換処理にどのような影響を与えるかを示しています。この図と、前出の図 10−4 は、Hex 変換ユーティリティ出力ファイル内のデータの状態を示しています。

図 10−6. ワード配列の変化



10.5 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。各ファイルを使用して、単一の ROM デバイスのプログラムを行います。

ROMS 疑似命令を使用しない場合、このユーティリティは、2 つのアドレス空間 (PAGE 0 と PAGE 1) を組み込んだデフォルトのメモリ構成を定義します。それぞれのアドレス空間は、1 つのアドレス領域を含みます。PAGE 0 はプログラム・アドレス空間全体のデフォルト領域を含み、PAGE 1 はデータ・アドレス空間全体のデフォルト領域を含みます。

ROMS 疑似命令は、TMS320C54x リンカの MEMORY 疑似命令に類似しています。どちらの疑似命令も、ターゲット・アドレス空間のメモリ・マップを定義するものです。ROMS 疑似命令では、各行エントリで特定のアドレス範囲を定義します。一般的な構文は次のとおりです。

```
ROMS
{
   [PAGE n:]
      romname: [origin=value,] [length=value,] [romwidth=value,]
               [memwidth=value,] [fill=value,]
               [files={filename1, filename2, ...}]

      romname: [origin=value,] [length=value,] [romwidth=value,]
               [memwidth=value,] [fill=value,]
               [files={filename1, filename2, ...}]
   ...
}
```

ROMS 疑似命令定義を開始します。

PAGE プログラム・アドレス空間とデータ・アドレス空間を使用するターゲット用のメモリ空間を識別します。プログラムが普通にリンクされている場合、PAGE 0 でプログラム・メモリを指定し、PAGE 1 でデータ・メモリを指定します。PAGE コマンドの後に定義した各メモリ範囲は、別の PAGE コマンドを指定するまで、そのページに属します。PAGE を指定しない場合は、すべての範囲は PAGE 0 に属します。

romname メモリ範囲を識別します。メモリ範囲の名前は、1 から 8 文字までの長さで指定します。この名前は、プログラムに対しては特に重要な意味をもちません。単に範囲を識別するだけです (重複したメモリ範囲名も許されます)。

origin メモリ範囲の開始アドレスを指定します。これは、origin、org、あるいは o と入力できます。指定する値は、10 進、8 進、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、デフォルト値として 0 が使用されます。

次の表は、10 進、8 進、または 16 進の定数を指定するときに使用できる表記法についてまとめたものです。

定数	表記法	例
16 進数	接頭部 0x または接尾部 h を付ける	0x77 または 077h
8 進数	接頭部 0 を付ける	077
10 進数	接頭部も接尾部も付けない	77

length メモリ範囲の長さを ROM デバイスの物理的な長さとして指定します。これは、length、len、あるいは l と入力することができます。値は、10 進、8 進、または 16 進の定数で指定しなければなりません。length の値を省略すると値は、デフォルトでアドレス空間全体の長さになります。

romwidth 該当するアドレス範囲の物理的な ROM 幅をビット単位で指定します (10-11 ページの 10.4.4 項「ROM 幅」を参照)。ここで値を指定すると、−romwidth オプションはその値によって変更されます。8 以上の 2 の累乗値を 10 進、8 進、または 16 進の定数として指定しなければなりません。

memwidth 該当するアドレス範囲のメモリ幅をビット単位で指定します (10-10 ページの 10.4.3 項「メモリ幅」を参照)。ここで値を指定すると、−memwidth オプションはその値によって変更されます。8 以上の 2 の累乗値を 10 進、8 進、または 16 進の定数として指定しなければなりません。memwidth パラメータを使用するときは、同時に SECTIONS 疑似命令で paddr パラメータをセクションごとに指定しなければなりません。

fill アドレス範囲で使用する埋め込み値を指定します。イメージ・モードでは、Hex 変換ユーティリティは、あるアドレス範囲のセクション間にあるホール (穴) 部分をこの値で埋め込みます。ターゲット幅と同じ幅をもつ値を 10 進、8 進、または 16 進の定数として指定しなければなりません。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 を使用するときは、−image コマンド行オプションも使用しなければなりません。10-27 ページの 10.8.2 項「埋め込み値を指定する方法」を参照してください。

files アドレス範囲に対応する出力ファイルの名前を指定します。名前のリストは中括弧で囲み、最下位から最上位へと順に出力ファイルを指定してください。

ファイル名の数は、該当するアドレス範囲で生成される出力ファイルの数と同じでなければなりません。出力ファイルの数を計算するには、10-11 ページの 10.4.4 項「ROM 幅」を参照してください。指定したファイル名が多すぎたり少なすぎたりする場合は、Hex 変換ユーティリティから警告が出されます。

−image オプションを使用している場合を除いて、アドレス範囲を定義するパラメータはすべて任意選択です。コンマと等号も任意選択です。origin または length なしでアドレス範囲を指定すると、アドレス空間全体が定義されます。イメージ・モードでは、すべてのアドレス範囲について origin と length が必要です。

同じページのアドレス範囲が重複しないようにしてください。また、アドレス範囲は小さい方から順に指定する必要があります。

10.5.1 ROMS 疑似命令を指定する場合

ROMS 疑似命令を使用しない場合、このユーティリティは、2 つのアドレス空間 (PAGE 0 と PAGE 1) を組み込んだデフォルトのメモリ構成を定義します。それぞれのアドレス空間は、1 つのアドレス領域を含みます。PAGE 0 はプログラム・アドレス空間全体のデフォルト領域を含み、PAGE 1 はデータ・アドレス空間全体のデフォルト領域を含みます。特定のページに何もロードされない場合は、そのページに対する出力は作成されません。

ROMS 疑似命令は、次のような場合に使用します。

- ❏ **大量のデータを固定サイズの ROM に書き込む場合。**ROM の長さに対応したメモリ範囲を指定すると、出力は ROM に収まるいくつかのブロックに自動的に分割されます。
- ❏ **出力を特定のセグメントだけに制限する場合。**ROMS 疑似命令を使用して、ターゲット・アドレス空間の特定のセグメント (1 つ以上) だけに変換を制限することもできます。ROMS 疑似命令で定義した範囲にないデータは変換されません。セクションが範囲の境界にまたがることもありますが、そのような場合は、境界で複数の範囲に分割されます。あるセクションが、ユーザ定義したどの範囲にもまったく入らない場合、そのセクションは変換されず、メッセージも警告も発行されません。この方法では、SECTIONS 疑似命令でセクションの名前を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、警告が発行され、範囲内にある部分だけが変換されます。
- ❏ **image モードを使用する場合。**−image オプションを使用するときは、必ず ROMS 疑似命令を使用してください。各範囲はすべて埋め込まれ、ある範囲に対応するそれぞれの出力ファイルに、その範囲全体のデータが入ります。セクションの前後またはセクション間にある空き部分には、ROMS 疑似命令で指定した埋め込み値、−fill オプションで指定した値、またはデフォルト値のゼロが埋め込まれます。

10.5.2 ROMS 疑似命令の例

例 10−1 の ROMS 疑似命令は、16 ビット・メモリの 16K のワードを 4 個の 8K × 8 ビット EPROM に分割する方法を示しています。

例 10−1. ROMS 疑似命令の例

```
infile.out
-image
-memwidth 16

ROMS
{
   EPROM1: org = 04000h, len = 02000h, romwidth = 8
           files = { rom4000.b0, rom4000.b1 }

   EPROM2: org = 06000h, len = 02000h, romwidth = 8,
           fill = 0FFh,
           files = { rom6000.b0, rom6000.b1 }
}
```

この例では、EPROM1 で 4000h から 5FFFh までのアドレス範囲を定義しています。この範囲には、次のセクションが含まれます。

セクション	対応する範囲
.text	4000h ～ 487Fh
.data	5B80H ～ 5FFFh

このアドレス範囲の残り部分には、0h (デフォルトの埋め込み値) が埋め込まれます。この範囲にあるデータは、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 は、6000h から 7FFFh までのアドレス範囲を定義しています。この範囲には、次のセクションが含まれます。

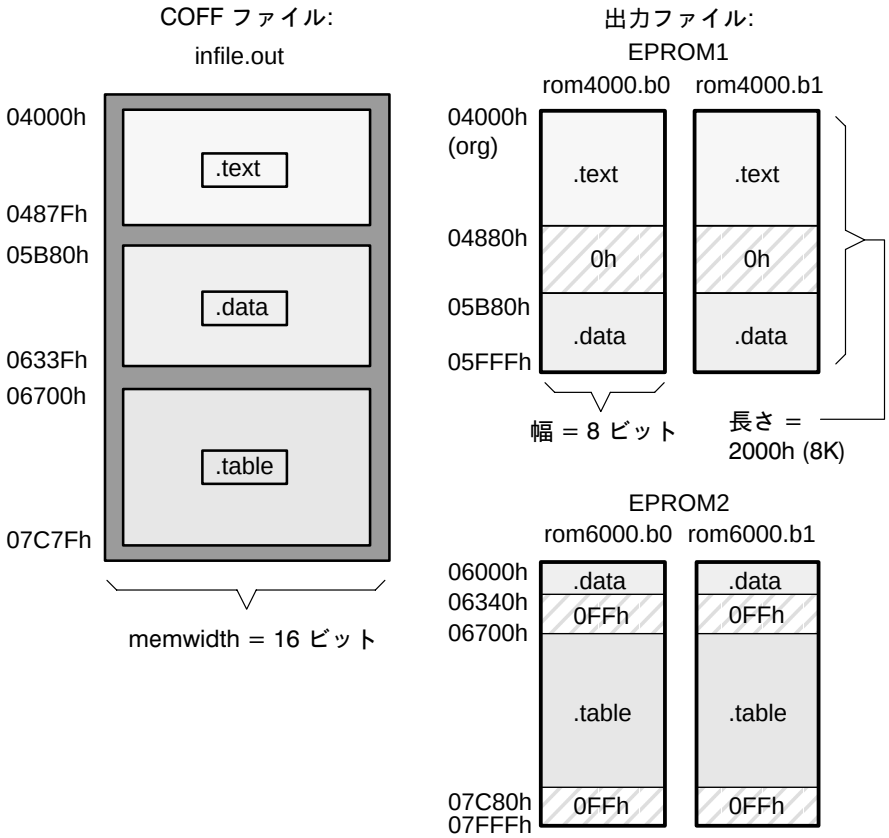
セクション	対応する範囲
.data	6000h ～ 633Fh
.table	6700h ～ 7C7Fh

このアドレス範囲の残りの部分には、0FFh (指定した埋め込み値) が埋め込まれます。この範囲にあるデータは、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 10−7 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ています。

図 10−7. 例 10−1 の infile.out ファイルを 4 個の出力ファイルに分割する



10.5.3 ROMS 疑似命令のマップ・ファイルを作成する方法

マップ・ファイル (−map オプションで指定) は、ROMS 疑似命令で複数の範囲を指定する場合に非常に有効です。マップ・ファイルには、各アドレス範囲、各範囲のパラメータ、関連付けられた出力ファイルの名前、およびアドレスごとに分割した内容のリスト (セクション名と埋め込み値) が示されます。例 10−1 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 10−2.　例 10−1 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
```

10.6 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できます。また、オンチップ・ブート・ローダからロードするように構成するセクションや、リンカ・コマンド・ファイルで指定したロード・アドレスとは異なるアドレスで ROM に配置するセクションを指定することもできます。

- ❑ SECTIONS 疑似命令を使用する場合、この疑似命令でリストを作成するセクションのみがユーティリティによって変換されます。COFF ファイル内のその他のセクションは、すべて無視されます。
- ❑ SECTIONS 疑似命令を使用しない場合、構成メモリの範囲内にある初期化されたセクションはすべて変換されます。TMS320C54x コンパイラが生成する初期化されたセクションには、.text、.const、.cinit、および .data があります。

初期化されないセクションは、SECTIONS 疑似命令に指定してあるかどうかにかかわらず、決して変換されることはありません。

注: C/C++ コンパイラが生成するセクション

TMS320C54x C/C++ コンパイラは、以下のセクションを自動的に生成します。

- ❑ **初期化されたセクション:** .text、.const、.cinit、および .data
- ❑ **初期化されないセクション:** .bss、.stack、および .sysmem

SECTIONS 疑似命令は、コマンド・ファイル内で使用します (コマンド・ファイルの使用方法の詳細は、10-7 ページの 10.3 節「コマンド・ファイル」を参照してください)。SECTIONS 疑似命令の一般的な構文は次のとおりです。

```
SECTIONS
{
   sname: [paddr=value]
   sname: [paddr=boot]
   sname: [= boot ],
   ...
}
```

SECTIONS 疑似命令定義を開始します。

sname COFF 入力ファイル内のセクションを識別します。存在しないセクションを指定すると、ユーティリティから警告が発行され、その名前は無視されます。

paddr 該当するセクションを配置する物理的な ROM アドレスを指定します。リンカで与えられるセクションのロード・アドレスは、この値によって変更されます (10-34 ページの 10.10 節「ROM デバイス・アドレスの制御方法」を参照してください)。値は、10 進、8 進、または 16 進の定数で指定しなければなりません。「**boot**」を指定することもできます (オンチップ・ブート・ローダで使用するブート・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 パラメータを使用する場合は、すべてのセクションで paddr パラメータを使用しなければなりません。

= boot オンチップ・ブート・ローダでロードするためにセクションを構成します。これは、**paddr=boot** 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・ローダ固有のコマンド行オプションの両方によって決まります。

セクション名を区切るコンマは、任意選択です。リンカの SECTIONS 疑似命令との共通性をもたせるために、セクション名の後で (boot キーワードの等号の代わりに) コロンを使用できます。たとえば、次の 2 つの文は同じです。

```
SECTIONS { .text: .data: boot }

SECTIONS { .text, .data = boot }
```

次の例では、COFF ファイル内に .text、.data、.const、.vectors、.coeff、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 のみを変換する場合は、SECTIONS 疑似命令を使用して次のように指定します。

```
SECTIONS { .text, .data }
```

この 2 つのセクションを両方ともブート・ローディング用に構成するには、boot キーワードを追加して次のように指定します。

```
SECTIONS { .text = boot, .data = boot }
```

注: −boot オプションと SECTIONS 疑似命令の使用方法

オンチップ・ブート・ローダに対して SECTIONS 疑似命令を使用すると、−boot オプションは無視されます。この場合は、SECTIONS 疑似命令の中でブート・セクションを明示的に指定する必要があります。−boot オプションと、オンチップ・ブート・ローダに関連するその他のコマンド行オプションの詳細は、10-29 ページの表 10−2 を参照してください。

10.7 出力ファイル名

Hex 変換ユーティリティは、使用している COFF オブジェクト・ファイルをあるデータ・フォーマットに変換するときに、データを 1 つ以上の出力ファイルに分割します。データをバイト幅またはワード幅のファイルに分割することによって、複数のファイルが作成されるときは、必ず最下位のファイルから最上位のファイルへの順序でファイル名が割り当てられます。これは、ターゲットまたは COFF のエンディアン配列や、−order オプションとは無関係に行われます。

10.7.1 出力ファイル名を割り当てる方法

Hex 変換ユーティリティは、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 = {. . .}) が組み込まれている場合は、そのリストにあるファイル名が使用されます。

   たとえば、ターゲット・データは 16 ビットのワードで、8 ビット幅の 2 つのファイルに分割されるとします。ROMS 疑似命令を使用してこの出力ファイルに名前を付けるには、次のように指定します。

   ```
   ROMS
   {
    RANGE1: romwidth=8, files={ xyz.b0 xyz.b1 }
   }
   ```

   ユーティリティは、最下位ビット (LSBs) を xyz.b0 に書き込み、最上位ビット (MSBs) を xyz.b1 に書き込むことによって、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、Hex 変換ユーティリティは −o オプションのリストからファイル名を取り出します。次のように指定すると、前述の ROMS 疑似命令の使用例と同じ結果が得られます。

   ```
   -o xyz.b0 -o xyz.b1
   ```

   ROMS 疑似命令と −o オプションを同時に指定した場合は、−o オプションよりも ROMS 疑似命令の方が優先されることに注意してください。

3) **デフォルトのファイル名を割り当てます。**ファイル名を指定しない場合や、指定したファイル名の数が出力ファイルの数より少ない場合は、ユーティリティはデフォルトのファイル名を割り当てます。デフォルトのファイル名は、COFF 入力ファイルからの基本名と、2 ～ 3 文字の拡張子 (ファイル名 .abc など) で構成されます。拡張子には、以下の 3 つの部分があります。

   a) 出力フォーマットに基づいたフォーマット文字

      a ASCII−Hex 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 0 から順に番号が振られます。ROMS 疑似命令がない場合、または範囲が 1 つだけの場合、この文字は省略されます。

   c) その範囲に関するファイルの集合でのファイル番号。この番号は、0 (最下位のファイルを示す) から始まります。

   たとえば、16 ビットのターゲット・プロセッサ用の coff.out があり、intel フォーマットの出力を作成するとします。出力ファイル名を指定しなければ、Hex 変換ユーティリティによって coff.i0 と coff.i1 という名前の 2 つの出力ファイルが作成されます。

   Hex 変換ユーティリティの起動時に、次の ROMS 疑似命令を指定した場合、2 個の出力ファイルが作成されます。

```
ROMS
{
   range1: o = 1000h l = 1000h
   range2: o = 2000h l = 1000h
}
```

出力ファイル	格納されるデータ
coff.i00	1000h ～ 1FFFh
coff.i10	2000h ～ 2FFFh

10.8 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もつ出力ファイルを作成する方法について説明します。

10.8.1 –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された範囲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。つまり、アドレス空間内のセクション間に、データのない空き部分が生じます。イメージ・モードを使用せずにこのようなファイルを変換すると、Hex 変換ユーティリティは、出力ファイル内のアドレス・レコードを使用して次のセクションの先頭までスキップすることで、これらの空き部分を処理します。つまり、出力ファイルのアドレスに不連続が生じることが考えられます。一部の EPROM プログラマは、アドレスの不連続をサポートしていません。

イメージ・モードでは、このような不連続が生じることはありません。各出力ファイルには、ターゲット・メモリ内のアドレス範囲に厳密に対応する、連続したデータ・ストリームが格納されます。セクションの前後またはセクション間にある空き部分には、指定した埋め込み値が埋め込まれます。

大部分の 16 進フォーマットは、各行にアドレスを必要とするので、イメージ・モードを使用して変換した出力ファイルもアドレス・レコードをもちます。ただし、イメージ・モードでは、これらのアドレスは必ず連続したものになります。

注: ターゲット・メモリの範囲を定義する方法

イメージ・モードを使用する場合は、ROMS 疑似命令も指定しなければなりません。イメージ・モードでは、各出力ファイルが、ある範囲のターゲット・メモリに直接対応します。ユーザは、この範囲を定義する必要があります。ターゲット・メモリの範囲を定義しない場合、ユーティリティはターゲット・プロセッサのアドレス空間全体のメモリ・イメージを作成しようとします。場合によっては、非常に大量の出力データが作成されます。このような状態を避けるために、ROMS 疑似命令を使用して、ユーザ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。

10.8.2 埋め込み値を指定する方法

–fill オプションは、セクション間のホールに埋め込む値を指定します。この埋め込み値は、–fill オプションの後に整数定数として指定する必要があります。定数の幅は、ターゲット・プロセッサ上のワード幅であるものとみなされます。たとえば、'C54x で –fill 0FFh と指定すると、埋め込みパターンは 00FFh となります。この定数値では、符号拡張は行われません。

fill オプションで値を指定しないと、Hex 変換ユーティリティはデフォルトの埋め込み値として 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。

10.8.3 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。詳細は、10-16 ページの 10.5 節「ROMS 疑似命令」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ために、–byte オプションを使用します。各出力ファイルについてアドレス起点を 0 にリセットするために、–zero オプションを使用します。–byte オプションの詳細は、10-36 ページの 10.10.3 項「–byte オプション」を、–zero オプションの詳細は、10-35 ページを参照してください。ROMS 疑似命令で埋め込み値を指定せず、デフォルト (0) 以外の埋め込み値を使用する場合は、–fill オプションを使用してください。

10.9 オンチップ・ブート・ローダ用のテーブルの作成方法

'C54x などの一部の DSP デバイスには、組み込みのブート・ローダが搭載されています。このブート・ローダは、1 つ以上のコードまたはデータ・ブロックによってメモリを初期化します。ブート・ローダは、メモリ (EPROM など) に格納されるか、またはペリフェラル (シリアル・ポートまたは通信ポートなど) からロードされる特別なテーブル (ブート・テーブル) を使用して、コードまたはデータを初期化します。Hex 変換ユーティリティは、ブート・テーブルを自動的に作成することによって、ブート・ローダをサポートします。

10.9.1 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」と呼ばれます。ブート・テーブルには、テーブル内に含まれているデータ・ブロックを指定された転送先アドレスにコピーするためのオンチップ・ローダに指示するレコードが含まれています。また、各種のプロセッサ制御レジスタを初期化するための値がブート・テーブルに含まれることもあります。ブート・テーブルは、メモリ内に格納することも、ペリフェラルを通じて読み取ることもできます。

Hex 変換ユーティリティは、ブート・ローダ用のブート・テーブルを自動的に作成します。このユーティリティを使用すると、ブート・ローダで初期化する COFF セクション、テーブルのロケーション、および任意の制御レジスタの値を指定できます。Hex 変換ユーティリティでは、COFF ファイルを基にターゲット・デバイスの型を識別して、デバイスに必要なフォーマットに従って完全なイメージ・テーブルを作成し、そのテーブルを出力ファイル内で 16 進フォーマットに変換します。その後、テーブルを ROM に焼き込んだり、その他の手段でロードしたりできます。

ブート・ローダは、通常のメモリ幅より幅が狭いメモリからのロードをサポートします。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 TMS320C54x 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、TMS320C54x リファレンス・セット マニュアルのブート・ローダの例を参照してください。

10.9.2 ブート・テーブル・フォーマット

ブート・テーブルのフォーマットは単純です。一般に、各種の制御レジスタ用の値が入っているヘッダ・レコードがあります。また、後続の各ブロックには、そのブロックのサイズおよび転送先アドレスが含まれたヘッダがあり、ヘッダの後にそのブロックのデータがあります。ブロックは、複数入力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をもつことができます。詳細は、TMS320C54x リファレンス・セット マニュアルのブート・ローダの節を参照してください。

10.9.3 ブート・テーブルの作成方法

表 10−2 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 10−2. ブート・ローダ・オプション

(a) すべての 'C54x デバイス用のオプション

オプション	説明
−boot	全セクションをブート可能な形式に変換します (SECTIONS 疑似命令の代わりに使用します)。
−bootorg 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。
−e *value*	ブート・ロード後に実行を開始するエントリ・ポイントを指定します。*value* は、アドレスまたはグローバル・シンボルになります。

(b) 'C54xLP デバイス専用のオプション

オプション	説明
−bootorg WARM または −warm	ブート・ローダ・テーブルのソースに、現在メモリ内にあるテーブルを指定します。
−bootorg COMM	ブート・ローダ・テーブルのソースに通信ポートを指定します。
−spc *value*	シリアル・ポート制御レジスタの値を設定します。
−spce *value*	シリアル・ポート制御拡張レジスタの値を設定します。
−arr *value*	ABU 受信アドレス・レジスタ値を設定します。
−bkr *value*	ABU 送信バッファ・サイズ・レジスタの値を設定します。
−tcsr *value*	TDM シリアル・ポート・チャネル選択レジスタの値を設定します。
−trta *value*	TDM シリアル・ポート送受信アドレス・レジスタの値を設定します。
−swwsr *value*	PARALLEL/WARM ブート・モードのソフトウェア・ウェイト・ステート・レジスタの値を設定します。
−bscr *value*	PARALLEL/WARM ブート・モードのバンクスイッチ制御レジスタの値を設定します。

10.9.3.1 ブート・テーブルの作成方法

ブート・テーブルを作成するには、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 (10-22 ページの 10.6 節「SECTIONS 疑似命令」を参照してください)。

ブートローダ・テーブル内に入れるセクションを選択すると、Hex 変換ユーティリティは、そのセクションのロード・アドレスをブート・テーブル内の該当するブロックの転送先アドレス・フィールドに配置します。セクションの内容は、該当するブロックの生データとして扱われます。

Hex 変換ユーティリティは、セクション実行アドレスを使用しません。リンク時には、ROM アドレスやブート・テーブルの構成について考慮する必要はありません。この構成は、Hex 変換ユーティリティが取り扱います。

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・ロード用に設定するように Hex 変換ユーティリティに指示できます。または、SECTIONS 疑似命令を使用して、特定のセクションを選択して構成することもできます (10-22 ページの 10.6 節「SECTIONS 疑似命令」を参照してください)。SECTIONS 疑似命令を使用した場合は、−boot オプションは無視されることに注意してください。

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、'C54x を使用していて、メモリ位置 8000h からブートする場合は、−bootorg 8000h と指定します。このように指定すると、Hex 変換ユーティリティ出力ファイルのアドレス・フィールドは 8000h から開始します。

−bootorg 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 疑似命令を使用しないと、最初のセクションのロード・アドレスからテーブルが開始されます。テーブルを PAGE 0 以外から開始できるようにするために、−bootpage オプションも用意されています。

手順 4: **ブートローダ固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のオプションを設定します。

手順 5: **システム・メモリ構成を記述します。**詳細は、10-9 ページの 10.4 節「メモリ幅について」、および 10-16 ページの 10.5 節「ROMS 疑似命令」を参照してください。

10.9.3.2 ブート・テーブル用の空きを確保する方法

完成したブート・テーブルは、ブート・ローダに関するヘッダ・レコードおよびデータがすべて含まれている単一のセクションに類似しています。この「セクション」のアドレスはブート・テーブルの起点となります。Hex 変換ユーティリティは、通常の変換処理の一部としてブート・テーブルを 16 進フォーマットに変換し、それを他のセクションと同じように出力ファイルにマップします。

システムのメモリには、必ずブート・テーブル用の空きを残しておいてください。特に、ROMS 疑似命令を使用しているときは注意が必要です。ブート・テーブルは、他の非ブート・セクションや未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・テーブルのために確保されています。このメモリを、ROMS 疑似命令で 1 つまたは複数の範囲として構成し、−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。

10.9.4 ペリフェラルからのブート方法

−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・デバイスと使用するチャネルに応じて選択します。たとえば、'C54x 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、'C54x をパラレル・ポートのいずれか 1 つからブートするには、−bootorg PARALLEL と指定します。

注: オンチップ・ブート・ローダについて

- **メモリ・コンフリクトの可能性**
  ペリ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。ブート・テーブルはペリフェラルを介して受け取ります。ただし、10-30 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、未使用のページの範囲を定義し、−bootpage オプションを使用してブート・テーブルをこのページに入れます。これによってブート・テーブルは、ダミー・ページの 0 の位置に表示されます。

- **ペリフェラルのブート・ローダ・アドレスに対する EPROM フォーマットが必要な理由**
  通常のシステムでは、親プロセッサは子プロセッサのペリフェラルを介して、その子プロセッサをブートします。ブート・ローダ・テーブル自体が親プロセッサのメモリ・マップ内の空間を占めることがあります。EPROM フォーマットと ROMS 疑似命令のアドレスは、子プロセッサが使用するものではなく、親プロセッサが使用するフォーマットとアドレスに対応するものです。

10.9.5 ブート・テーブルのエントリ・ポイントの設定方法

ブート・ロード処理が完了すると、リンカにより指定され、COFF ファイルに含まれたデフォルトのエントリ・ポイントで実行が開始されます。Hex 変換ユーティリティで −e オプションを使用することによって、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は、コマンド行またはコマンド・ファイルで −e 0123h と指定します。−e のアドレスは、リンカが生成するマップ・ファイルを見て決めることができます。

注: 有効なエントリ・ポイント

値は定数、またはアセンブリ・ソース内で対外的に定義された (たとえば .global を使って) シンボルでなければなりません。

−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。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。

−bootorg WARM オプションを使用する場合、−e オプションは ROM 内のブート・テーブルのロード・アドレスを設定します。

10.9.6 'C54x ブート・ローダの使用方法

この項では、'C54x デバイスのブート・ローダによる Hex 変換ユーティリティの使用方法について、例を示して説明します。'C54x のブート・ローダには複数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 オプションを使用し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
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
8 ビット・パラレル	−bootorg WARM	−memwidth 8
16 ビット・パラレル	−bootorg WARM	−memwidth 16
8 ビット I/O	−bootorg COMM	−memwidth 8

複数の出力ファイルを作成する場合以外は、−romwidth と −memwidth の設定は同じにするべきです。

'C54x 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。フォーマットはどのような組み合わせでも同じであり、ブート・テーブルは転送先アドレスが入ったフィールド、長さが入ったフィールド、およびデータが入ったブロックから構成されます。

1 つのセクションだけをブートできます。8 ビット・チャネルからブートする場合、16 ビットのワードは 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は、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、ユーティリティを起動するときに −bootorg SERIAL を指定します。−memwidth 8 または −memwidth 16 のどちらかを使用してください。
- ❑ 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起動します。−memwidth 8 または −memwidth 16 のどちらかを使用してください。
- ❑ 外部メモリ (EPROM) からブートするには、−bootorg オプションを使用してブート・メモリのソース・アドレスを指定します。−memwidth 8 または −memwidth 16 のどちらかを使用してください。

たとえば、図 10−8 のコマンド・ファイルを使用すると、0x8000 の位置にあるバイト幅の EPROM から abc.out の .text セクションをブートできます。

図 10−8. 'C54x EPROM からブートするためのコマンド・ファイルの例

```
abc.out              /* input file                          */
-o abc.i             /* output file                         */
-i                   /* Intel format                        */
-memwidth 8          /* 8-bit memory                        */
-romwidth 8          /* outfile is bytes, not words         */
-bootorg 0x8000      /* external memory boot                */

SECTIONS { .text: BOOT }
```

10.10 ROM デバイス・アドレスの制御方法

Hex 変換ユーティリティの出力アドレス・フィールドは、ROM デバイス・アドレスに対応します。EPROM プログラマは、Hex 変換ユーティリティ出力ファイルのアドレス・フィールドで指定されている位置にデータを焼き付けます。Hex 変換ユーティリティには、各セクションの ROM での開始アドレスを制御したり、アドレス・フィールドをインクリメントするのに使用するアドレス・インデックスを制御したりする、いくつかのメカニズムが用意されています。ただし、多くの EPROM プログラマには、データを焼き込む ROM の位置を直接制御する機能があります。

10.10.1 開始アドレスの制御方法

Hex 変換ユーティリティ出力ファイルの制御メカニズムは、ブート・ローダを使用しているかどうかによって異なります。

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順に示します。

1) **リンカ・コマンド・ファイル**

Hex 変換ユーティリティ出力ファイルのアドレス・フィールドは、デフォルトでは、ロード・アドレス (リンカ・コマンド・ファイルで指定される) とHex 変換ユーティリティのパラメータ値の関数になります。この関係を以下にまとめます。

$$\text{out\_file\_addr}^{\dagger} = \text{load\_addr} \times (\text{data\_width} \div \text{mem\_width})$$

out_file_addr	出力ファイルのアドレスです。
load_addr	リンカが割り当てたロード・アドレスです。
data_width	TMS320C54x デバイスの場合は 16 ビットとして指定されます。10-10 ページの 10.4.2 項「データ幅」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。10-10 ページの 10.4.3 項「メモリ幅」を参照してください。

† paddr が指定されていない場合

データ幅をメモリ幅で割った値は、アドレス生成のための補正係数です。データ幅がメモリ幅よりも大きい場合は、この補正係数によ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 0×1 で、データ幅をメモリ幅で割った値が 2 の場合、出力ファイル・アドレス・フィールドは 0×2 となります。データは、メモリ幅のサイズに相当する 2 つの連続する位置に分割されます。

2) **SECTIONS 疑似命令内の paddr パラメータ**
あるセクションに対して paddr パラメータが指定されている場合、Hex 変換ユーティリティは、セクションのロード・アドレスをバイパスして、paddr で指定されたアドレスにそのセクションを配置します。Hex 変換ユーティリティ出力ファイルのアドレス・フィールドと paddr パラメータとの関係は、次のようにまとめることができます。

$$out\_file\_addr^{\dagger} = paddr\_val + (load\_addr - sect\_beg\_load\_addr) \times (data\_width \div mem\_width)$$

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 パラメータで指定された値です。
sec_beg_load_addr	リンカにより割り当てられたセクションのロード・アドレスです。

† paddr が指定されていない場合

データ幅をメモリ幅で割った値は、アドレス生成のための補正係数です。ロード・アドレスからセクション開始ロードアドレス係数を引いたものは、セクションの開始点からのオフセットです。

3) **−zero オプション**
−zero オプションを使用すると、すべての出力ファイルについて、アドレス起点はゼロにリセットされます。各ファイルはゼロから開始され、上位方向にカウントされるため、アドレス・レコードは、データの実際のターゲット・アドレスではなく、ファイルの先頭からのオフセット (ROM 内でのアドレス) を表します。

各出力ファイル内の開始アドレスを強制的にゼロにするには、−zero オプションと −image オプションを組み合わせて使用しなければなりません。−image オプションを指定せずに −zero オプションを指定した場合は、ユーティリティから警告が発行され、−zero オプションは無視されます。

ブートローダ・モード
ブート・ローダを使用している場合、Hex 変換ユーティリティはブート・テーブル内の異なる COFF セクションを連続するメモリ位置に入れます。各 COFF セクションは 1 つのブート・テーブル・ブロックになり、それぞれのブロックの転送先アドレスは、リンカが割り当てたセクション・ロード・アドレスと同じになります。

ブート・テーブルでは、Hex 変換ユーティリティ出力ファイルのアドレス・フィールドは、リンカが割り当てたセクションのロード・アドレスとは無関係です。ブート・テーブルのアドレス・フィールドは、テーブルの開始に対するオフセットに、補正係数 (メモリ幅で割ったデータ幅) を掛けた値を示しているだけです。リンカにより割り当てられたセクションのロード・アドレスは、そのセクションのサイズ、およびそのセクション内に含まれたデータと一緒にブート・テーブルにエンコードされます。これらのアドレスは、ブート・ロードの処理中に、データをメモリに保存するために使用されます。

以下のメカニズムのいずれかを使用しない限り、COFF 入力ファイル内にある最初のブート可能なセクションのリンク後のロード・アドレスは、デフォルトによってブート・テーブルの開始アドレスになります。それぞれのメカニズムは、優先順位の低い方から高い方へ順に示してあります。重複している範囲では、優先順位の低いオプションで設定した値よりも、優先順位の高いメカニズムで設定した値が使用されます。

1) **ROMS 疑似命令で指定した ROM 起点**
Hex 変換ユーティリティは、ROMS 疑似命令内の最初のメモリ範囲の起点にブート・テーブルを配置します。

2) **−bootorg オプション**
メモリからのブートを選択している場合、Hex 変換ユーティリティは、−bootorg オプションで指定されたアドレスにブート・テーブルを配置します。−bootorg PARALLEL と −bootorg SERIAL のどちらも、アドレス・フィールドには影響を与えません。

10.10.2 アドレス・インクリメント・インデックスの制御方法

デフォルトでは、Hex 変換ユーティリティは、出力ファイルのアドレス・フィールドをメモリ幅の値に基づいてインクリメントします。メモリ幅が16 ビットの場合は、出力ファイルの各行にある 16 ビット・ワードの個数に基づいて、アドレスがインクリメントされます。

10.10.3 −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いと、2 行目はアドレス 8 (8h) から始まり、−byte オプションを使用すると、2 行目はアドレス 16 (010h) から始まります。両方の例のデータは同じです。−byte は、出力ファイル・アドレス・フィールドの計算に影響を及ぼすだけで、変換後のデータの実際のターゲット・プロセッサ・アドレスには影響しません。

−byte オプションを使用すると、ターゲット・プロセッサでバイト単位のアドレス指定が可能であるかどうかにかかわらず、出力ファイル内のアドレス・レコードはファイル内のバイト位置を参照します。

10.10.4 アドレス・ホールの処理方法

メモリ幅がデータ幅と異なる場合は、自動的にロード・アドレスに補正係数がかけられるので、あるセクションの開始、または 2 つのセクションの間にホールが作成されることがあります。

たとえば、COFF セクション (.sec1) を 8 ビット EPROM のアドレス 0x0100 にロードするとします。リンカ・コマンド・ファイルでこのロード・アドレスを 0x0100 と指定すると、Hex 変換ユーティリティは、そのアドレスに 2 (データ幅をメモリ幅で割った値 = 16/8 = 2) を掛けて、出力ファイルの開始アドレスを 0x0200 とします。EPROM プログラマを使用して EPROM の開始アドレスを制御しない限り、EPROM 内にホールが作成されます。EPROM には、0x0100 ではなく、0x0200 からデータが焼き込まれます。これは、次のような方法で解決できます。

- **SECTIONS 疑似命令の paddr パラメータを使用する方法**
  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 10−9 は、.sec1 の開始部分にホールができないようにした Hex コマンド・ファイルを示しています。

図 10−9. 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 origin パラメータを使用する方法 (ブート・ロード時のみ)**
  10-35 ページで説明したように、ブート・ロードを実行する場合は、−bootorg オプションまたは ROMS 疑似命令の origin により、ブートローダ・テーブル全体の EPROM アドレスを制御できます。

- 別の例に関しては、C-10 ページの C.4「例 3: ブート・テーブルの作成方法」を参照してください。

10.11 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、Extended Tektronix、および TI−Tagged) のいずれかに変換します。

表 10−3 は、フォーマットを指定するオプションを示しています。

- ❑ これらのオプションを 2 つ以上使用する場合は、最後に指定したオプションは有効になります。
- ❑ 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 10−3. Hex 変換フォーマットを指定するオプション

オプション	フォーマット	アドレスのビット数	デフォルト幅
−a	ASCII−Hex	16	8
−i	Intel	32	8
−m1	Motorola−S1	16	8
−m2 または −m	Motorola−S2	24	8
−m3	Motorola−S3	32	8
−t	TI−Tagged	16	16
−x	Tektronix	32	8

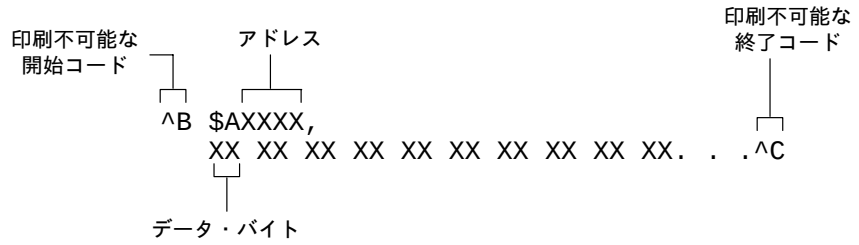
アドレスのビット数は、該当するフォーマットでサポートされるアドレス情報のビット数を示します。16 ビット・アドレスのフォーマットでは、64K までのアドレスのみがサポートされます。Hex 変換ユーティリティは、ターゲット・アドレスを有効なビット数に合わせて切り捨てます。

デフォルト幅は、該当するフォーマットのデフォルト出力幅を示します。このデフォルト幅は、−romwidth オプションを使用するか、ROMS 疑似命令で romwidth パラメータを使用して変更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。

10.11.1 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 10−10 は、ASCII−Hex フォーマットを示しています。

図 10−10. ASCII−Hex オブジェクト・フォーマット



このファイルは、ASCII STX 文字 (ctrl−B、02h) で開始し、ASCII ETX 文字 (ctrl−C、03h) で終了します。アドレス・レコードは、$AXXXX (XXXX は 4 桁 (16 ビット) の 16 進アドレス) で示されます。アドレス・レコードが存在するのは、次の場合に限られます。

- ❑ 不連続が生じた場合
- ❑ バイト・ストリームがアドレス 0 から開始されていない場合

−image オプションと −zero オプションを使用すると、不連続およびアドレス・レコードをすべて回避することができます。これらが回避される場合は、バイト値のリストとして出力されます。

10.11.2 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

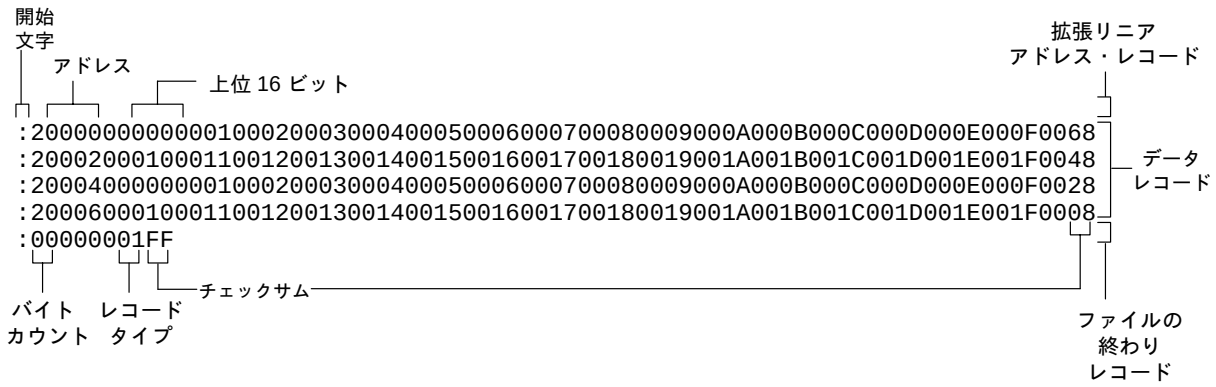
レコード・タイプ 00 (データ・レコード) は、コロン (:) で始まり、その後にバイト・カウント、最初のデータ・バイトのアドレス、レコード・タイプ (00) およびチェックサムが順に続きます。このアドレスは、32 ビット・アドレスの最下位 16 ビットです。この値が最新の 04 (拡張リニア・アドレス) レコード内の値と連結されて、完全な 32 ビット・アドレスが作成されます。チェックサムは、バイト・カウント、アドレス、データ・バイトなどのレコード内で先行する各バイトの和の 2 の補数 (2 進数) です。

レコード・タイプ 01 (ファイルの終わりレコード) 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 10−11 は、Intel−Hex オブジェクト・フォーマットを示しています。

図 10−11. Intel−Hex オブジェクト・フォーマット



10.11.3 Motorola Exorciser オブジェクト・フォーマット (−m1 オプション、−m2 オプション、および −m3 オプション)

Motorola S1、S2、および S3 のフォーマットは、それぞれ 16 ビット、24 ビット、および 32 ビットのアドレスをサポートします。これらのフォーマットは、ファイルの始め (ヘッダ) レコード、データ・レコード、およびファイルの終わり (終了) レコードから構成されます。各レコードには、レコード・タイプ、バイト・カウント、アドレス、データ、およびチェックサムの 5 つのフィールドがあります。レコード・タイプには、以下のものがあります。

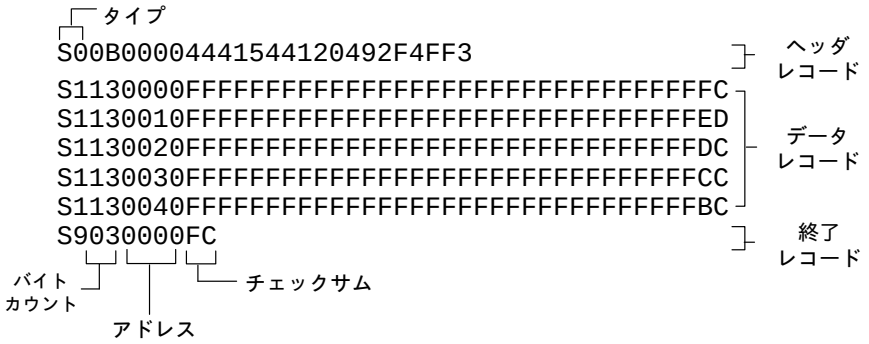
レコード・タイプ	説明
S0	ヘッダ・レコード
S1	16 ビット・アドレス用のコード／データ・レコード (S1 フォーマット)
S2	24 ビット・アドレス用のコード／データ・レコード (S2 フォーマット)
S3	32 ビット・アドレス用のコード／データ・レコード (S3 フォーマット)
S7	32 ビット・アドレス用の終了レコード (S3 フォーマット)
S8	24 ビット・アドレス用の終了レコード (S2 フォーマット)
S9	16 ビット・アドレス用の終了レコード (S1 フォーマット)

バイト・カウントは、レコード・タイプとバイト・カウントを除いた、レコード内の文字のペアのカウントです。

チェックサムは、バイト・カウント、アドレス、およびコード/データの各フィールドを構成している文字のペアで表される値の合計の 1 の補数の最下位バイトです。

図 10−12 は、Motorola−S オブジェクト・フォーマットを示しています。

図 10−12. Motorola−S フォーマット



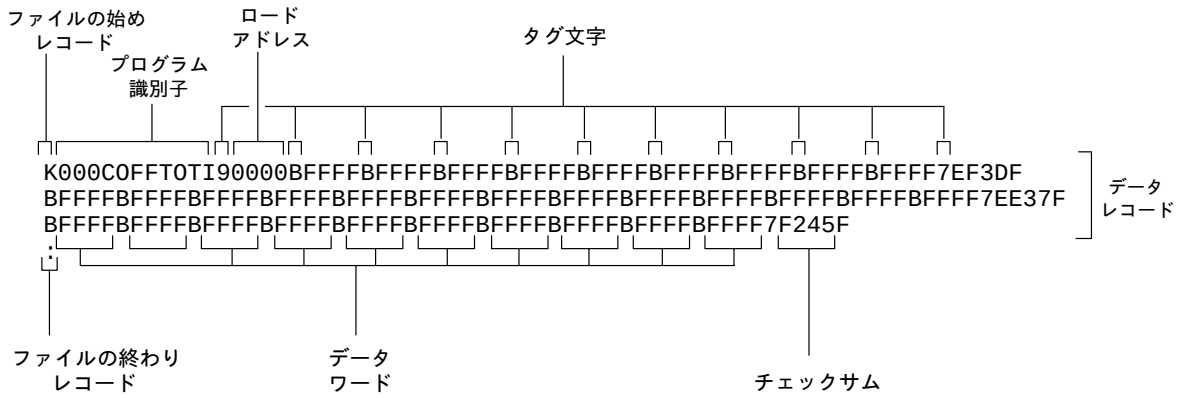
10.11.4 TI−SDSMAC オブジェクト・フォーマット (−t オプション)

TI−Tagged オブジェクト・フォーマットは、16 ビット・アドレスをサポートします。このフォーマットは、ファイル開始レコード、データ・レコード、およびファイルの終わりレコードから構成されます。各データ・レコードは、一連の小フィールドから構成され、タグ文字によ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 10−13 は、TI−Tagged オブジェクト・フォーマットのタグ文字とフィールドを示しています。

図 10−13. TI−Tagged オブジェクト・フォーマット



最初のアドレスより前にデータ・フィールドがある場合は、その最初のフィールドにアドレス 0000h が割り当てられます。どのデータ・バイトについてもアドレス・フィールドを作成できますが、必須のアドレス・フィールドはありません。タグ文字 7 の後にあるチェックサム・フィールドは、最初のタグ文字からチェックサムのタグ文字 (7 または 8) までにある文字の、8 ビット ASCII 値の合計の 2 の補数です。ファイルの終わりレコードは、コロン (:) です。

10.11.5 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、次の 2 つのレコード・タイプがあります。

データ・レコード ヘッダ・フィールド、ロード・アドレス、およびオブジェクト・コードが格納されます。

終了レコード モジュールの終了を示します。

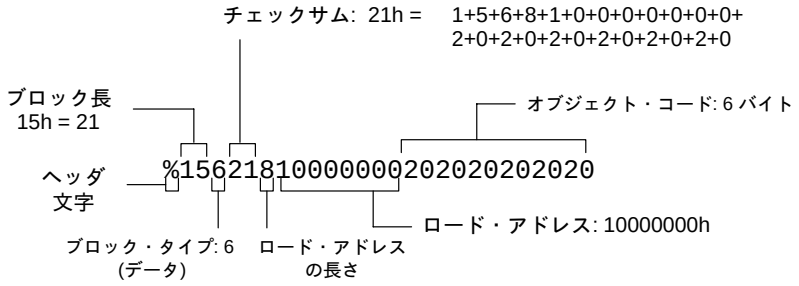
データ・レコード内のヘッダ・フィールドには、以下の情報が入ります。

項目	ASCII 文字の数	説明
%	1	Tektronix フォーマットのデータ・タイプ
ブロック長	2	% を除いたレコード内の文字数
ブロック・タイプ	1	6 = データ・レコード 8 = 終了レコード
チェックサム	2	% とチェックサムを除いたレコード内のすべての値の合計の 256 に対する剰余 (2 桁の 16 進数)

データ・レコード内のロード・アドレスは、オブジェクト・コードが指定されたアドレスを示します。最初の桁はアドレス長を示していて、常に 8 です。データ・レコードの残りの文字には、オブジェクト・コード (1 バイトについて 2 文字) が含まれています。

図 10−14 は、Tektronix オブジェクト・フォーマットを示しています。

図 10−14. Extended Tektronix オブジェクト・フォーマット



10.12 Hex 変換ユーティリティのエラー・メッセージ

section mapped to reserved memory message

説明　セクションまたはブートローダ・テーブルが、プロセッサのメモリ・マップに示されている予約メモリ領域にマップされています。

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ください。有効なメモリ位置については、TMS320C54x リファレンス・セット マニュアルを参照してください。

sections overlapping

説明　複数の COFF セクションのロード・アドレスが重複しているか、ブート・テーブルのアドレスが他のセクションと重複しています。

処置　この問題の原因には、メモリ幅がデータ幅より狭い場合に、 Hex 変換ユーティリティで実行したロード・アドレスから 16 進出力ファイル・アドレスへの変換が正しく行われなかったことが考えられます。10-9 ページの 10.4 節「メモリ幅について」、および 10-34 ページの 10.10 節「ROM デバイス・アドレスの制御方法」を参照してください。

unconfigured memory error

説明　このエラーは、以下のどちらかが原因で起きたと考えられます。

- ロード・アドレスが ROMS 疑似命令で定義したメモリ範囲内にないセクションが、COFF ファイルに含まれている。
- ブートローダ・テーブルのアドレスが、ROMS 疑似命令で定義したメモリ範囲内にない。

処置　ROMS 疑似命令で定義された ROM 範囲を訂正して、各アドレスがメモリ範囲内に収まるようにするか、またはセクションのロード・アドレスかブートローダ・テーブルのアドレスを修正してください。ROMS 疑似命令を使用していない場合は、デフォルトによ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を忘れないでください。この理由により、ROMS 疑似命令を除去する方法も次の解決策として使用できます。

第 11 章

ニーモニックから代数表記への変換ユーティリティの説明

TMS320C54x のニーモニックから代数表記への変換ユーティリティは、ニーモニック命令セットで書かれたアセンブリ・コードを、代数命令セットで書かれたコードに変換します。

項目 **ページ**

11.1 変換ユーティリティの概要 11-2
11.2 変換ユーティリティと開発フロー 11-3
11.3 変換ユーティリティの起動方法 11-4
11.4 変換モード 11-5
11.5 変換ユーティリティでのマクロの処理方法 11-8

11.1 変換ユーティリティの概要

TMS320C54x ニーモニックから代数表記への変換ユーティリティは、ニーモニックのアセンブリ命令を代数表記のアセンブリ命令に変換します。ニーモニック命令は通常、キーワードとオペランドから構成されています。代数表記命令は、通常、オペランドと演算子から構成されています。代数表記命令は、高水準プログラミング言語の命令によく似ています。

変換ユーティリティは、エラーのないコードを必要とします。認識できない命令やマクロの起動を検出すると、変換ユーティリティは標準出力にメッセージを出力し、コード行を変換しません。

変換ユーティリティはニーモニック命令が入ったアセンブリ・コードのソース・ファイルを受け取り、代数表記命令が入ったアセンブリ・コードのソース・ファイルを生成します。入力ファイルには、拡張子を付けないか、.asm という拡張子を付けることができます。出力ファイルは入力ファイルと同じ名前になり、.cnv という拡張子が付きます。

11.1.1 変換ユーティリティが実行すること

変換ユーティリティは、以下のことを実行します。

- ニーモニックを、言語仕様で定義されている、その命令の機能に相当する代数表記表現に置き換えます。代数表記表現は、複数のコード行から構成される場合もあります。
- ニーモニック命令のオペランドのフォーマットを、言語仕様に記述されている代数表記表現の構文のフォーマットに設定し直します。フォーマットを設定し直す場合、以下のことが行われます。
  - データ・メモリ・アドレス (dma) へのアクセスを示す場合、dma の前に @ 記号が付きます。
  - ニーモニックの間接省略表現 * は、*AR0 に置き換えられます。
  - 必要な場合、定数の前に # 記号が付きます。
  - 単一のオペランドとして使用され、複数の用語をもつ代数表記式は、小括弧で囲まれます。

11.1.2 変換ユーティリティが実行しないこと

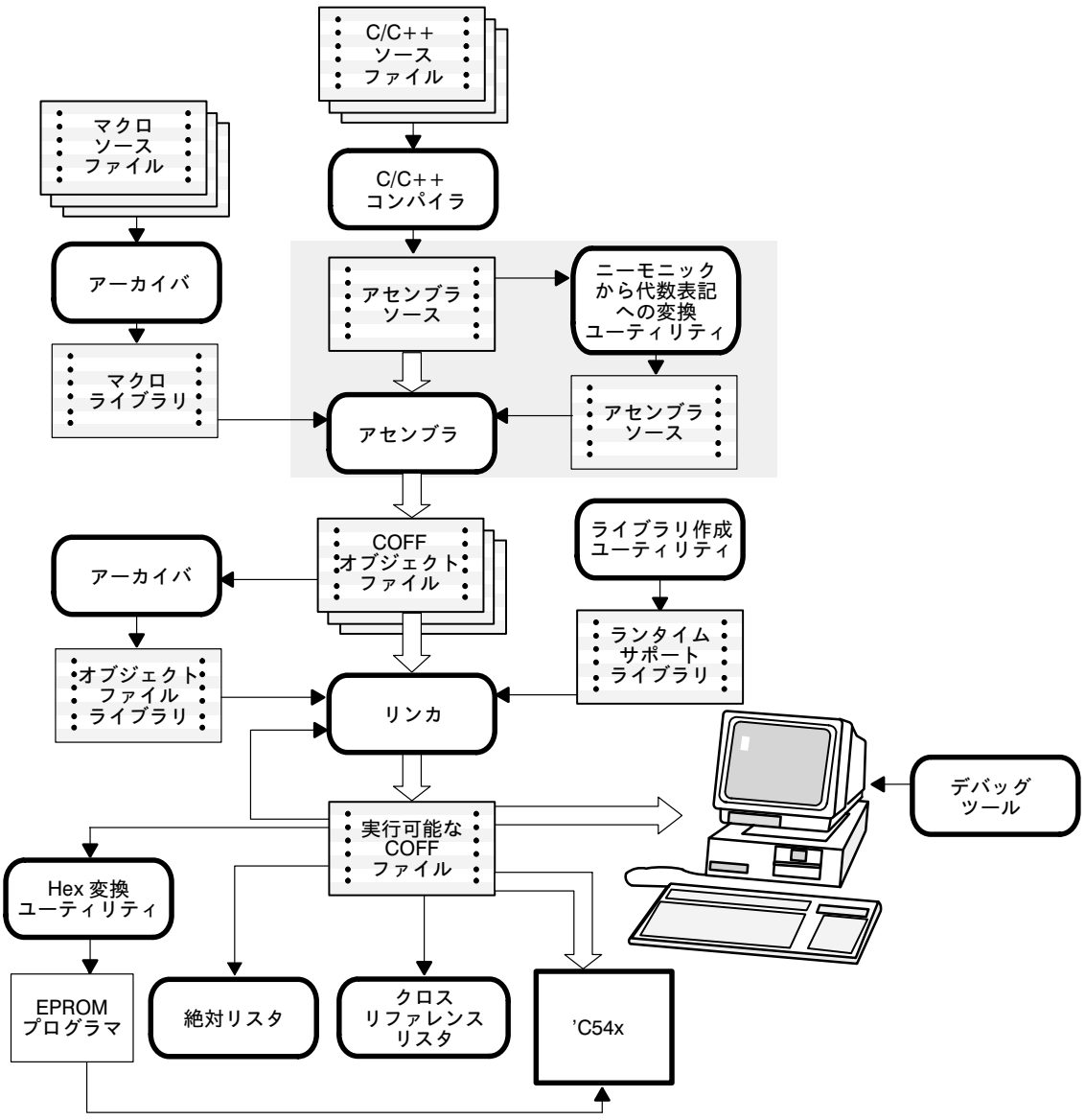
変換ユーティリティには、以下の制限事項があります。

- 変換ユーティリティは、マクロ定義を変換できません。マクロ定義は無視されます。変換ユーティリティはオプションとして、マクロの起動を展開されたマクロに置き換え、仮パラメータを実際の起動で使用された引数に置き換えます。
- 変換ユーティリティは、ニーモニック命令と同じ名前のマクロを変換しようとします。マクロ名は、ニーモニック命令と同じにならないようにする必要があります。

11.2 変換ユーティリティと開発フロー

図 11−1 は、アセンブリ言語開発プロセスでの変換ユーティリティの役割を示しています。アセンブラは、ニーモニックまたは代数表記の構文を受け取ります。

図 11−1. 変換ユーティリティと開発フロー



11.3 変換ユーティリティの起動方法

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。

mnem2alg [*option*] *inputfile*

mnem2alg 変換ユーティリティを起動するコマンドです。

option 変換ユーティリティのモードを指定します (11-5 ページの 11.4 節「変換モード」を参照してください)。次のオプションがあります。

−t リテラル・モード。これは、option を指定しなかった場合のデフォルトです。

−e 展開モード

inputfile 変換するアセンブリ・ソース・ファイルの名前を指定します。拡張子を指定しない場合は、.asm を拡張子と見なします。

11.4 変換モード

変換ユーティリティは、次のいずれかのモードで作動します。

リテラル	元のニーモニック命令をコメントにして残し、変換した命令をその後に続けます。
展開	マクロの起動を展開して前処理し、置換シンボルを置換します。

11.4.1 リテラル・モード (–t オプション)

デフォルトのリテラル・モード (–t オプション) で実行すると、変換ユーティリティは前処理せずに命令を変換します。変換ユーティリティはマクロを処理せず、置換シンボルの展開も行いません。マクロの起動や命令を認識できないと、変換ユーティリティは標準出力にメッセージを出力し、そのコードを変換しません。変換ユーティリティは、アセンブラ・ソース・ファイルと同じ名前で .cnv の拡張子が付いたファイルを作成します。

図 11–2. リテラル・モードの処理

11.4.2 リテラル・モードでのシンボル名について

リテラル・モードでは、変換ユーティリティは .asg によって定義されたシンボル名をラベルとして扱い、シンボル名が表す値としては扱いません。次の例では、ソース・コードの変換が行われ、このとき sym はデータ・メモリ・アドレスとして扱われます。

例 11–1. リテラル・モードでのシンボル名の扱い

(a) ソース・コード:

```
sym    .asg   *AR2
       LD  sym,B
```

(b) 変換後のコード:

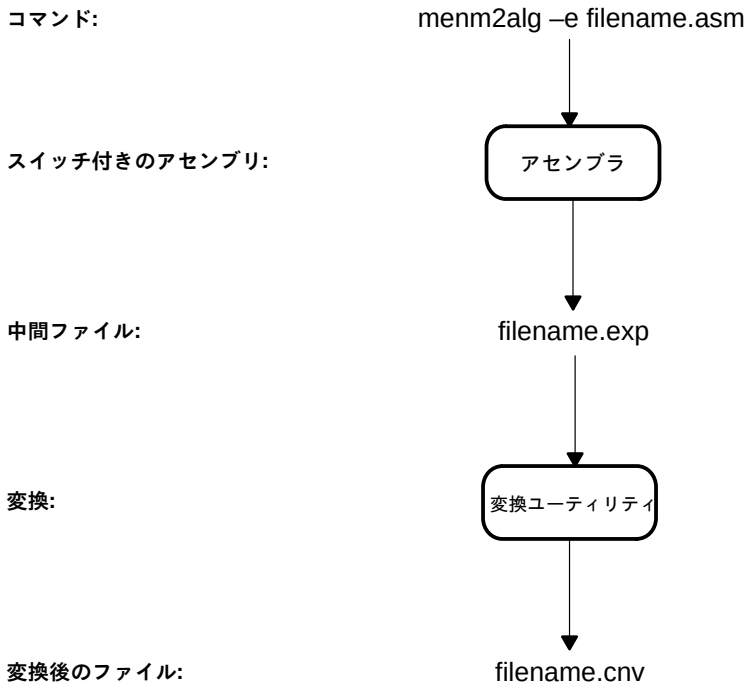
```
sym    .asg *AR2
       B = @sym
```

11.4.3 展開モード (–e オプション)

展開モードを起動するには、–e オプションを使用します。展開モードでは、変換ユーティリティはマクロと置換シンボルを前処理してから命令を変換します。マクロを展開し、置換シンボルを挿入するために、変換ユーティリティは入力を前処理するスイッチを指定してアセンブラを起動します。アセンブラは、.exp という拡張子が付いたファイルを作成します。.exp ファイルは、変換ユーティリティに戻され、処理されます。変換ユーティリティは、アセンブラ・ソース・ファイルと同じ名前で .cnv の拡張子が付いたファイルを作成します。

展開モードでは、変換ユーティリティはアセンブラを起動するので、アセンブラの実行可能ファイルをパスに含めておく必要があります。アセンブラの実行可能ファイルのバージョンは、1.11 以降でなければなりません。前処理中にアセンブラがエラーを検出した場合、変換ユーティリティは実行を中止し、出力を生成しません。

図 11–3. 展開モードの処理



次の例は、展開モードでの処理方法の例を示しています。中間ファイルでは、マクロの起動はコメント化され、展開されたマクロはその場所に挿入され、実際の引数への置換も行われています。マクロ定義は変換されていませんが、結果として生成される .cnv ファイルは、代数表記アセンブラによってアセンブルでき、出力を生成できます。アセンブラはマクロ定義を処理しません (変換ユーティリティは定義を変換しないからです)。アセンブラが生成した .exp 中間ファイルは、変換の完了後に削除されます。

例 11−2. 展開モード

(a) ソース・コード

```
file.asm
*******************************
          .asg *AR0,sym

mymac     .macro parm1,parm2
          LD     parm1,parm2
          ADD    sym,5,parm2,B
          .endm

          mymac  sym,A
*******************************
```

(b) 中間コード

```
file.exp – after preprocessing
           before translation
*******************************
          .asg *AR0,sym

mymac     .macro parm1,parm2
          LD     parm1,parm2
          ADD    sym,5,parm2,B
          .endm

;           mymac  sym,A
          LD     *AR0,A
          ADD    *AR0,5,A,B
*******************************
```

(c) 変換後のコード

```
file.cnv – after translation
*******************************
          .asg *AR0, sym

mymac     .macro parm1,parm2
          LD     parm1,parm2
          ADD    sym,5,parm2,B
          .endm

;           mymac  sym,A
          A = *AR0
          B = A + *AR0 << 5
*******************************
```

11.5 変換ユーティリティでのマクロの処理方法

この節では、変換ユーティリティでのマクロの処理について説明します。説明する内容は次のとおりです。

- ❑ マクロ内の疑似命令
- ❑ マクロ・ローカル変数
- ❑ マクロを起動するときのラベルの定義

11.5.1 マクロ内の疑似命令

マクロの起動が展開されると、マクロ定義内の疑似命令は中間ファイルにコピーされません。その代わり、マクロはインライン化され、コードはマクロ環境内に存在しなくなります。次のソース・コードは、この例に示した中間コードに前処理されます。

例 11−3. マクロ内の疑似命令

(a) ソース・コード

```
mymac  .macro parm1
       .var temp
       .eval parm1, temp
       .word temp
       .endm

       mymac 5
```

(b) 中間コード

```
mymac  .macro parm1
       .var   temp
       .eval parm1,temp
       .word temp
       .endm

;        mymac 5
       .word 5
```

11.5.2 マクロ・ローカル変数

マクロ・ローカル変数が検出されると、それらの変数は、そのマクロを繰り返し呼び出しても同じラベルが生成されないように変更されます。次の図に示す例では、ソース・コードは、中間コードに前処理されます。

例 11−4. マクロ・ローカル変数

(a) ソース・コード

```
mymac  .macro  parm1
lab?   .word   parm1
       .endm

       mymac 4
       mymac 40
       mymac 400
```

(b) 中間コード

```
mymac     .macro  parm1
lab?      .word   parm1
          .endm

;           mymac 4
lab$1$    .word   4
;           mymac 40
lab$2$    .word   40
;           mymac 400
lab$3$    .word   400
```

ローカル・ラベル名は、前に $*n*$ が付加されます。この *n* はマクロの起動の番号です。他のラベルが、生成されるマクロ・ローカル・ラベルと同じものにならないようにする必要があります。

11.5.3 マクロを起動するときのラベルの定義

マクロの起動へ関連付けられているラベルが存在する場合、このラベルは、展開と変換が行われた後は使用されません。なぜなら、このラベルは、マクロの起動とともにコメント化されるからです。次に示す例では、ソース・コードは、中間コードに前処理されます。

図 11−4. ラベルの定義方法

(a) ソース・コード

```
mymac  .macro
       .word F403
       .endm
LABEL  mymac
```

(b) 中間コード

```
 mymac  .macro
        .word F403
        .endm
 ;LABEL    mymac
        .word F403
```

LABEL は、このコードをアセンブルしたときは定義されません。ラベル定義は、マクロの起動と同じ行に置かないようにする必要があります。この例のソース・コードは、次のように書き直してください。

図 11−5. 書き直したソース・コード

```
mymac  .macro
       .word F403
       .endm
LABEL
       mymac
```

付録 A

共通オブジェクト・ファイル・フォーマット

コンパイラ、アセンブラ、およびリンカは、共通オブジェクト・ファイル・フォーマット (COFF) のオブジェクトを作成します。COFF は、AT&T 社が UNIX ベースのシステム向けに開発したフォーマットと同名のオブジェクト・ファイル・フォーマットを実現したものです。このオブジェクト・ファイル・フォーマットが使用された理由は、モジュラ・プログラミングが促進される点と、コード・セグメントおよびターゲット・システムのメモリを管理する強力かつ柔軟な方式が提供される点にあります。

COFF の基本概念は「セクション」です。第 2 章「共通オブジェクト・ファイル ・フォーマットの概要」では、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・ファイルの構造に関する技術的な詳細情報が収録されています。この情報の大部分は、C/C++ コンパイラで生成されるシンボリック・デバッグ情報に関するものです。この付録の目的は、COFF オブジェクト・ファイルの内部フォーマットに関する補足情報を提供することです。

項目 **ページ**

A.1 COFF ファイルの構造 A-2
A.2 ファイル・ヘッダの構造 A-5
A.3 オプション・ファイル・ヘッダのフォーマット A-6
A.4 セクション・ヘッダの構造 A-7
A.5 再配置情報の構造化 A-10
A.6 行番号テーブルの構造 A-12
A.7 シンボル・テーブルの構造と内容 A-14

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情報とシンボリック・デバッグ情報を記述しています。次の要素があります。

- ファイル・ヘッダ
- オプション・ヘッダの情報
- セクション・ヘッダのテーブル
- 初期化されたセクションごとの生データ
- 初期化されたセクションごとの再配置情報
- 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によって作成されるオブジェクト・ファイルは、同一の COFF 構造をもちます。ただし、最終的にリンクされるプログラムには、通常、再配置エントリ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ています。

図 A−1. COFF ファイルの構造

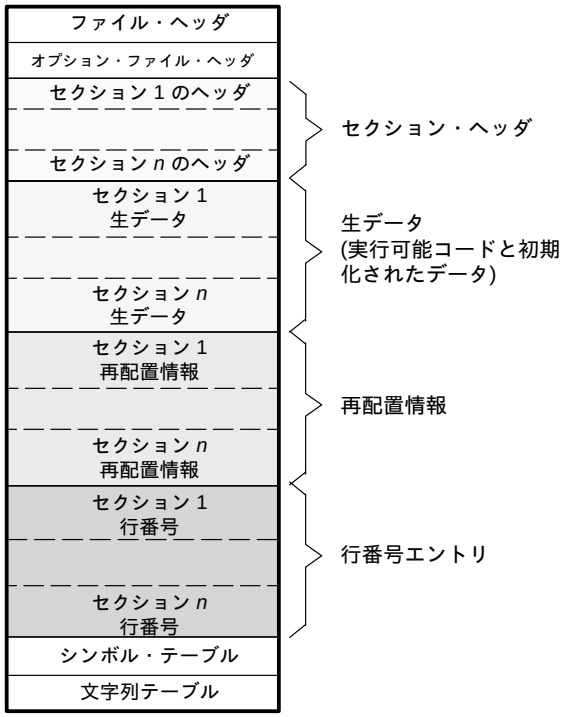
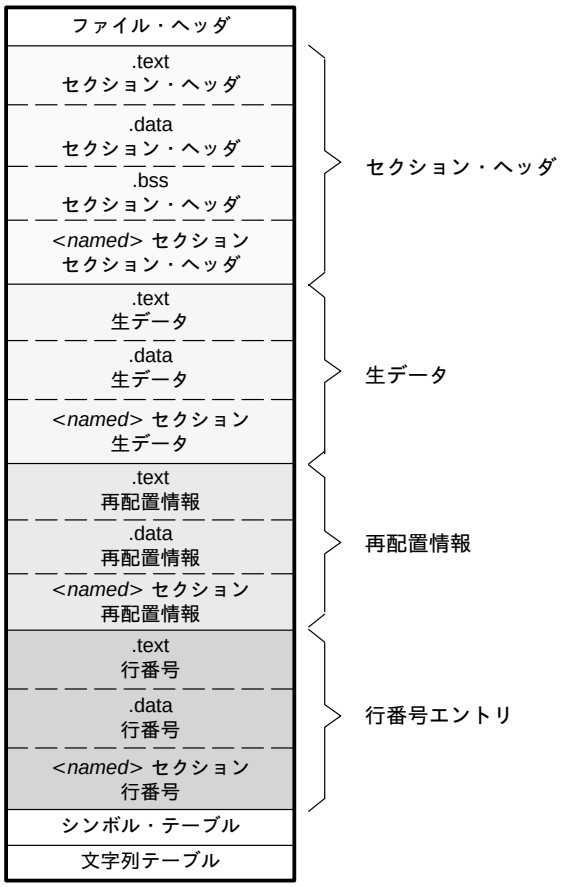


図 A−2 は、COFF オブジェクト・ファイルの一般的な例を示しています。このファイルには、.text、.data、および .bss の 3 つのデフォルト・セクションと、名前付きセクション（図中では <named> と示されています）が含まれています。デフォルトでは、.text セクション、.data セクション、初期化された名前付きセクション、.bss セクション、初期化されない名前付きセクションという順序で、オブジェクト・ファイル内に各セクションが配置されます。初期化されないセクションにも、セクション・ヘッダはあります。ただし、生データ、再配置情報、および行番号エントリはありません。これは、.bss および .usect の疑似命令が、初期化されないデータ用に単に空間を確保するだけだからです。初期化されないセクションには、実際のコードは含まれません。

図 A−2.　COFF オブジェクト・ファイル



A.1.1 オペレーティング・システムの交換による影響

’C54x COFF ファイルは、対応する各オペレーティング・システムに関係なく、すべての開発ツールによって認識されます。使用するオペレーティング・システムを別のオペレーティング・システムに変更する際は、COFF ファイルのファイル・ヘッダ情報のみのバイトをスワップする必要があります。COFF ファイルの生データを変更する必要は全くありません。

’C54x 開発ツールは、ファイル・ヘッダの違いを検出し、この違いに合わせて自動的に補正を行います。この処理は、’C54x 開発ツールだけを使用している場合には正しく行われます。

COFF ファイルの違いを識別するには、オプション・ファイル・ヘッダのマジック・ナンバを確認します。バイト 0～1 にマジック・ナンバが格納されています。SunOS™ または HP−UX™ のオペレーティング・システムの場合、マジック・ナンバは 108h です。DOS オペレーティング・システムの場合、マジック・ナンバは 801h です。

A.2 ファイル・ヘッダの構造

ファイル・ヘッダには、オブジェクト・ファイルの一般的なフォーマットを記述した情報 (22 バイト) が含まれています。表 A−1 は、COFF ファイル・ヘッダの構造を示しています。

表 A−1. ファイル・ヘッダの内容

バイト位置	型	説明
0 ～ 1	unsigned short int	バージョン ID。COFF ファイル構造のバージョンを示します。
2 ～ 3	unsigned short int	セクション・ヘッダの数
4 ～ 7	long int	時刻および日付スタンプ。ファイルの作成日時を示します。
8 ～ 11	long 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ります。
12 ～ 15	long int	シンボル・テーブル内のエントリの数
16 ～ 17	unsigned short int	オプション・ヘッダのバイト数。このフィールドには、0 か 28 のいずれかが入ります。0 の場合、オプション・ファイル・ヘッダは存在しません。
18 ～ 19	unsigned short int	フラグ (表 A−2 を参照)
20 ～ 21	unsigned short int	ターゲット ID。マジック・ナンバは、ファイルが TMS320C54x™ システムで実行可能なことを示します。

表 A−2 は、ファイル・ヘッダのバイト 18～19 に格納されるフラグのリストです。これらのフラグは任意の数および組み合わせで同時に設定される場合があります (たとえば、バイト 18～19 に 0003h が設定された場合は、F_RELFLG と F_EXEC が両方とも設定されたことを示します)。

表 A−2. ファイル・ヘッダのフラグ (バイト 18 ～ 19)

ニーモニック	フラグ	説明
F_RELFLG	0001h	再配置情報はファイルから除去されました。
F_EXEC	0002h	ファイルは再配置可能です (未解決の外部参照は含まれていません)。
F_LNNO	0004h	行番号はファイルから除去されました。
F_LSYMS	0008h	ローカル・シンボルはファイルから除去されました。
F_LITTLE	0100h	このファイルのバイト配列は、'C54x デバイスが使用できる形式です (1 ワード当たり 16 ビットで最下位のバイトが最初に置かれます)。
F_SYMMERGE	1000h	重複するシンボルは除去されました。

A.3 オプション・ファイル・ヘッダのフォーマット

リンカは、オプション・ファイル・ヘッダを作成して、ダウンロード時にそのヘッダを使用して再配置を実行します。部分的にリンクされるファイルには、オプション・ファイル・ヘッダは含まれません。表 A−3 は、オプション・ファイル・ヘッダのフォーマットを示しています。

表 A−3. オプション・ファイル・ヘッダの内容

バイト位置	型	説明
0 〜 1	short int	マジック・ナンバ (SunOS または HP−UX の場合は 108h、DOS の場合は 801h)
2 〜 3	short int	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 .data セクションのサイズ (ワード数)
12 〜 15	long int	初期化されない .bss セクションのサイズ (ワード数)
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・ヘッダの構造

COFF オブジェクト・ファイルには、オブジェクト・ファイル内の各セクションの開始位置を定義するセクション・ヘッダのテーブルが含まれています。それぞれのセクションには、独自のセクション・ヘッダがあります。COFF1 ファイルと COFF2 ファイルは、それぞれ異なるセクション・ヘッダの情報を含んでいます。表 A−4 は、COFF1 ファイルのセクション・ヘッダの内容を示しています。表 A−5 は、COFF2 ファイルのセクション・ヘッダの内容を示しています。

表 A−4. COFF1 ファイルのセクション・ヘッダの内容

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ます)
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 int	再配置エントリの数
34 ～ 35	unsigned short int	行番号エントリの数
36 ～ 37	unsigned short int	フラグ (表 A−6 を参照)
38	char	予約済み
39	char	メモリ・ページ番号

表 A−5. COFF2 ファイルのセクション・ヘッダの内容

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ます)
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 ～ 35	unsigned long	再配置エントリの数
36 ～ 39	unsigned long	行番号エントリの数
40 ～ 43	unsigned long	フラグ (表 A−6 を参照)
44 ～ 45	short	予約済み
46 ～ 47	unsigned short	メモリ・ページ番号

表 A−6 は、セクション・ヘッダに設定されるフラグのリストを示しています。複数のフラグを組み合わせて設定できます。たとえば、フラグ・ワードが 024h に設定されている場合は、STYP_GROUP と STYP_TEXT の両方が設定されていることを示します。

表 A−6. セクション・ヘッダのフラグ

ニーモニック	フラグ	説明
STYP_REG	0000h	通常セクション (割り当て、再配置、およびロードが行われます)
STYP_DSECT	0001h	ダミー・セクション (再配置は行われますが、割り当てとロードは行われません)
STYP_NOLOAD	0002h	非ロード・セクション (割り当てと再配置は行われますが、ロードは行われません)
STYP_GROUP	0004h	グループ・セクション (いくつかの入力セクションから構成されます)
STYP_PAD	0008h	埋め込みセクション (ロードは行われますが、割り当てと再配置は行われません)
STYP_COPY	0010h	コピー・セクション (再配置とロードは行われますが、割り当ては行われません。再配置エントリと行番号エントリは通常どおり処理されます)
STYP_TEXT	0020h	実行可能なコードが書き込まれているセクション
STYP_DATA	0040h	初期化されたデータが含まれているセクション
STYP_BSS	0080h	初期化されないデータが含まれているセクション
STYP_CLINK	4000h	条件付きでリンクされているセクション

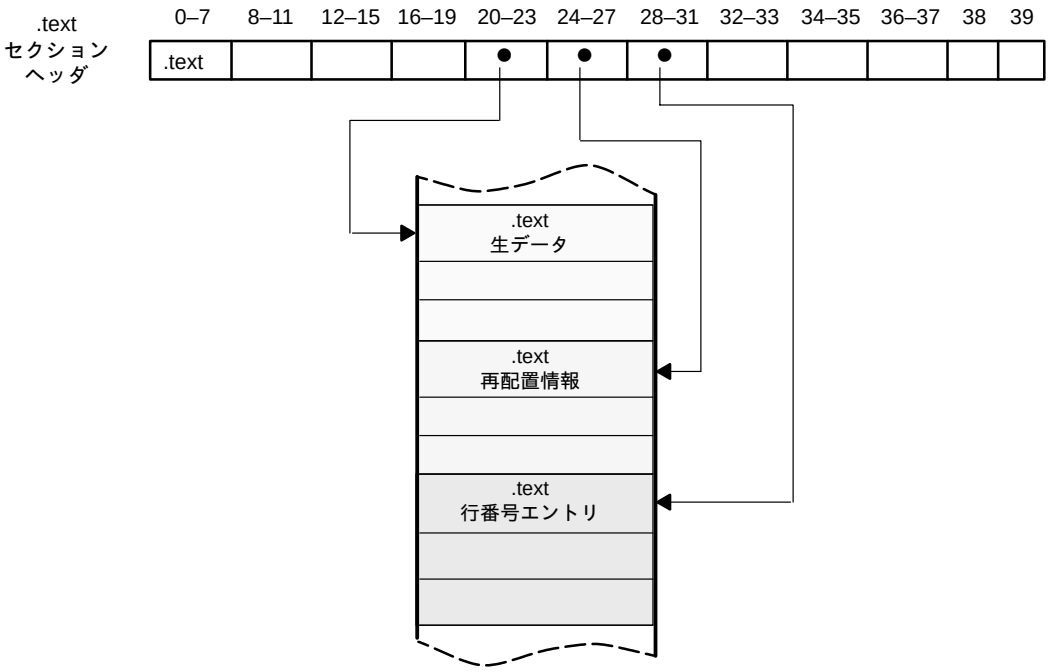
注: ロードという用語は、セクションの生データがオブジェクト・ファイル内に現れることを意味します。

フラグは次のバイト位置に格納されます。

バイト	対応する COFF フォーマット
36 ～ 37	COFF1
40 ～ 43	COFF2

図 A−3 は、セクション・ヘッダ内のポインタが、オブジェクト・ファイル内の .text セクションに関連する要素をどのように指すかを示しています。

図 A−3.　.text セクションのセクション・ヘッダのポインタ



A-3 ページの図 A−2 に示すように、(.bss および .usect の疑似命令によって作成される) 初期化されないセクションは、このフォーマットとは異なります。初期化されないセクションには、セクション・ヘッダはありますが、生データ、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は、オブジェクト・ファイル内で実際に空間を占有することはありません。このため、初期化されないセクションについては、再配置エントリの数、行番号エントリの数、およびファイル・ポインタは 0 に設定されます。初期化されないセクションのヘッダは、変数のための空間をメモリ・マップ内にどれだけ確保するかをリンカに指示するだけです。

A.5 再配置情報の構造化

COFF オブジェクト・ファイルには、再配置可能な参照ごとに 1 つの再配置エントリがあります。アセンブラは、この再配置エントリを自動的に生成します。リンカは、各入力セクションを読み取るときに再配置エントリを読み取り、再配置を実行します。再配置エントリにより、各入力セクション内の参照がどのように取り扱われるかが決まります。

COFF ファイルの再配置情報エントリでは、表 A−7 に示す 12 バイトのフォーマットが使用されます。

表 A−7. 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 int	COFF1 ファイル: 予約済み COFF2 ファイル: 拡張アドレス計算のための追加バイト
10 ～ 11	unsigned short int	再配置タイプ (表 A−8 を参照)

仮想アドレスは、再配置前の現行のセクション内でのシンボルのアドレスです。仮想アドレスは、再配置が必要な場所を指定します(これは、オブジェクト・コード内でのパッチが必要なフィールドのアドレスです)。

再配置エントリを生成するコードの例を以下に示します。

```
2                       .global   X
3  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 000001 となります。

シンボル・テーブルのインデックスは、参照されるシンボルのインデックスです。上記の例では、このフィールドにはシンボル・テーブル内の X のインデックスが格納されます。セクション内のシンボルの現行のアドレスと、そのシンボルのアセンブル時のアドレスとの差が、再配置の大きさです。再配置可能フィールドは、参照シンボルと同じ大きさだけ再配置する必要があります。例では、X は再配置前に値 0 をもちます。X がアドレス 2000h に再配置される場合を考えてみましょう。これは再配置の大きさ (2000h − 0 = 2000h) なので、アドレス 1 の再配置フィールドはパッチされ、2000h が加算されます。

あるシンボルがどのセクションで定義されているかが分かれば、そのシンボルの再配置後のアドレスを求めることができます。たとえば、X は .data セクションで定義されており、.data は 2000h に再配置されるとすれば、X は 2000h に再配置されます。

再配置エントリ内でシンボル・テーブルのインデックスが −1 (0FFFFh) であれば、この再配置は、内部再配置と呼ばれます。この場合、再配置の大きさは、単に現行のセクションが再配置される大きさになります。

再配置タイプは、パッチされるフィールドのサイズを指定するとともに、パッチする値の計算方法を記述するものです。このタイプ・フィールドは、再配置可能な参照を生成するのに使用されたアドレッシング・モードによって異なります。上記の例では、参照されるシンボル (X) の実アドレスは、オブジェクト・コード内の 16 ビット・フィールドに置かれます。この実アドレスは 16 ビットの直接参照なので、再配置タイプは R_RELWORD となります。表 A−8 は、再配置タイプを示しています。

表 A−8. 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_REL24	0005h	シンボルのアドレスへの 24 ビット直接参照
R_RELBYTE	000Fh	シンボルのアドレスへの 8 ビット直接参照
R_REL13	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_RELLONG	0021h	シンボルのアドレスへの 32 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・デバッグに便利な行番号エントリのテーブルが格納されます。C/C++ コンパイラがアセンブリ言語コードを何行か生成する際に、行番号エントリが作成されます。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/C++ ソース・コードの元の行に逆にマップします。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−9 は、行番号エントリのフォーマットを示しています。

表 A−9. 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、次の 2 つの値のいずれかをもちます。 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・エントリを指します。 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です。
4 ～ 5	unsigned short int	このエントリは、次の 2 つの値のいずれかをもちます。 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/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ています。

図 A−4. 行番号のブロック

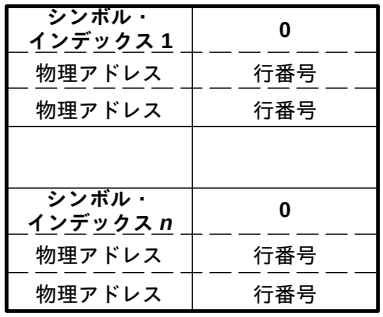


図 A−4 に示すように、それぞれのエントリは以下のように 2 つに分けられます。

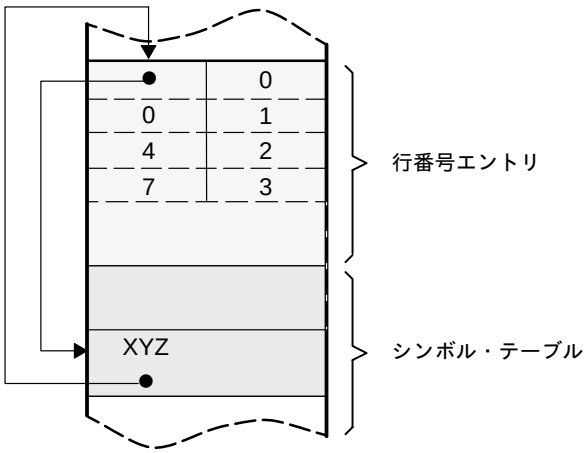
- ある関数の最初の行である場合、バイト 0～3 はシンボル・テーブル内のシンボル、つまり関数の名前を指し、バイト 4～5 にはブロックの開始であることを示す 0 が入ります。

- 関数の 2 行目以降の行である場合、バイト 0～3 は物理アドレス (C/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/C++ ソース・プログラム内での出現箇所に相対的な元の C/C++ ソースのアドレスを示します。

行エントリ・テーブルには、これらのブロックを多数格納できます。

図 A–5 は、XYZ という名前の関数の行番号エントリを示しています。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/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生成されます。

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より小さくできます。

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/C++ で定義されるシンボルのうち、関数外部の記憶クラス static をもつ変数を指します。同じ名前のシンボルを複数のモジュールで使用する必要がある場合は、それらのシンボルを静的にして、それぞれのシンボルのスコープを、該当するシンボルを定義しているモジュールだけに限定します (こうすると、複数の定義が矛盾しなくなります)。

シンボル・テーブル内のそれぞれのシンボル・エントリには、次の情報が格納されます。

- ❑ 名前 (または文字列テーブルへのポインタ)
- ❑ 型
- ❑ 値
- ❑ シンボルが定義されたセクション
- ❑ 記憶クラス
- ❑ 基本型 (整数、文字など)
- ❑ 派生型 (配列、構造など)
- ❑ 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・テーブル内のシンボル・エントリは、クラスおよび型とは無関係に、同一のフォーマットです。シンボル・テーブルの各エントリには、表 A−10 に示す 18 バイトの情報が格納されています。それぞれのシンボルは 18 バイトの補足エントリをもつことがあります。A-16 ページの表 A−11 に示す特殊シンボルは、必ず補足エントリをもちます。シンボルの中には、前述の特性を全部はもたないシンボルもあります。フィールドが設定されない場合、そのフィールドはヌルになります。

表 A−10. 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次のいずれかが格納されます。 1) 8 文字のシンボル名 (未使用部分にはヌルが埋め込まれます) 2) 文字列テーブルへのポインタ (シンボル名が 8 文字よりも長い場合)
8 ～ 11	long int	シンボルの値 (記憶クラスにより異なります)
12 ～ 13	short int	シンボルのセクション番号
14 ～ 15	unsigned short int	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 または 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。表 A−11 は、これらの特殊シンボルを示しています。

表 A−11. 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
.*n*fake	構造体、共用体、または列挙型のダミー・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 つ一組で使用されるものがあります。

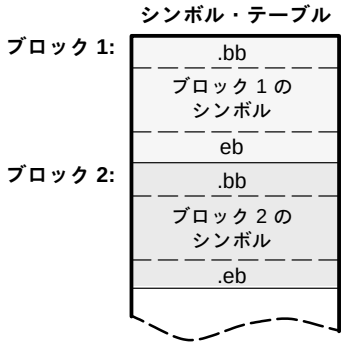
- .bb/.eb は、あるブロックの開始と終了を示します。
- .bf/.ef は、ある関数の開始と終了を示します。
- *n*fake/.eos は、名前が割り当てられていない構造体、共用体、および列挙型の境界に名前を割り当てて定義します。.eos シンボルは、名前付きの構造体、共用体、および列挙型とともに使用されます。

構造体、共用体、または列挙型がタグ名をもたない場合は、これらをシンボル・テーブルに入力できるように、コンパイラは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。

A.7.1.1 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、特殊シンボル .bb/.eb に囲まれて表現されます。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ています。

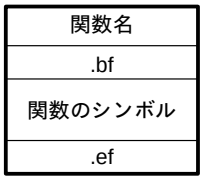
図 A−7. ブロックに関するシンボル



A.7.1.2 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、特殊シンボル .bf/.ef に囲まれて表現されます。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ています。

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・エントリが挿入されます。

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。このシンボル名は、必要に応じてヌルが埋め込まれ、バイト 0～7 に格納されます。
- ❑ シンボル名が 8 文字を超えている場合、このフィールドは 2 つの倍長整数 (long) 型として扱われます。この場合は、シンボル名全体が文字列テーブルに格納されます。バイト 0～3 には 0 が入り、バイト 4～7 は文字列テーブルへのオフセットとなります。

A.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、シンボル名が格納されるシンボル・テーブル・エントリ内のフィールドには、シンボル名の代わりに、文字列テーブル内にあるシンボル名へのポインタが格納されます。文字列テーブルでは、それぞれの名前がヌル・バイトで区切られ、連続して格納されます。文字列テーブルの最初の 4 バイトには、文字列テーブルのサイズがバイト数単位で入ります。このため、文字列テーブルへのオフセットは、4 以上になります。

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ています。この文字列テーブル内のインデックスは、Adaptive−Filter については 4、Fourier−Transform については 20 となります。

図 A−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/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−12 は、有効な記憶クラスを示しています。

表 A−12. 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_EXTREF	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 (特殊シンボル .bb と .eb でのみ使用)
C_MOS	8	構造体のメンバ	C_FCN	101	関数の開始または終了 (特殊シンボル .bf と .ef でのみ使用)
C_ARG	9	関数引数	C_EOS	102	構造体の終了 (特殊シンボル .eos でのみ使用)
C_STRTAG	10	構造体タグ	C_FILE	103	ファイル名 (特殊シンボル .file でのみ使用)
C_MOU	11	共用体のメンバ	C_LINE	104	ユーティリティ・プログラムでのみ使用

特殊シンボルの中には、特定の記憶クラスに制限されているものもあります。表 A−13 は、このようなシンボルとその記憶クラスを示しています。

表 A−13. 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−14 は、記憶クラス、および関連する値をまとめたものです。

表 A−14. 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です。このため、シンボル・テーブル内に .file シンボルによる片方向のリンク・リストが作成されます。.file シンボルがそれ以上ない場合、最後の .file シンボルはシンボル・テーブル内の最初の .file シンボルを指します。

再配置可能なシンボルの値は、そのシンボルの仮想アドレスです。リンカがあるセクションを再配置すると、再配置可能なシンボルの値はそれに応じて変更されます。

A.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−15 は、これらの番号とそれぞれの番号が示すセクションを示しています。

表 A−15. 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)

.text、.data、または .bss の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、セクション内では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、ファイル名が含まれます。セクション番号 −1 は、値をもっているが、再配置可能でないシンボルであることを示します。セクション番号 0 は、現行のファイルでは定義されていない再配置可能な外部シンボルであることを示します。

A.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。

型フィールドのビット 0～3 は、基本型を示します。表 A−16 は、有効な基本型を示しています。

表 A−16. 基本型

ニーモニック	値	型
T_NULL	0	割り当てられていない型
T_CHAR	2	char
T_SHORT	3	short int
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 int

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−17 は、有効な派生型を示しています。

表 A−17. 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もつシンボルには、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 int 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 つもつ**場合と、**もたない**場合があります。このシンボル・テーブル補足エントリは、シンボル・テーブル・エントリと同じバイト数 (18) をもちますが、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−18 は、これらの関係をまとめたものです。

表 A−18. シンボル・テーブル補足エントリのフォーマット

		型エントリ		
名前	**記憶クラス**	**派生型 1**	**基本型**	**補足エントリのフォーマット**
.file	C_FILE	DT_NON	T_NULL	ファイル名 (表 A−19 を参照)
.text、.data、.bss	C_STAT	DT_NON	T_NULL	セクション (表 A−20 を参照)
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	タグ名 (表 A−21 を参照)
.eos	C_EOS	DT_NON	T_NULL	構造体の終了 (表 A−22 を参照)
fcname	C_EXT C_STAT	DT_FCN	(注 1 を参照)	関数 (表 A−23 を参照)
arrname	(注 2 を参照)	DT_ARY	(注 1 を参照)	配列 (表 A−24 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−25 および表 A−26 を参照)
.bf、.ef	C_FCN	DT_NON	T_VOID	関数の開始および終了 (表 A−25 および表 A−26 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−27 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−18 において、tagname は (特殊シンボル *n*fake を含む) 任意のシンボル名を示します。fcname と arrname は、任意のシンボル名を示します。

表 A−18 の複数の条件を満たすシンボルは、補足エントリの共用体フォーマットをもちます。これらの条件をどれも満たさないシンボルは、補足エントリをもちません。

A.7.8.1 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−19. 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

A.7.8.2 セクション

表 A−20 は、テーブル補足エントリのフォーマットを示しています。

表 A−20. 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 int	セクションの長さ
4 ～ 6	unsigned short int	再配置エントリの数
7 ～ 8	unsigned short int	行番号エントリの数
9 ～ 17	—	未使用 (ゼロが埋め込まれます)

A.7.8.3 タグ名

表 A−21 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ています。

表 A−21. 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 (ゼロが埋め込まれます)
6 ～ 7	unsigned short int	構造体、共用体、または列挙型のサイズ
8 ～ 11	—	未使用 (ゼロが埋め込まれます)
12 ～ 15	long int	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 (ゼロが埋め込まれます)

A.7.8.4 構造体の終了

表 A−22 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ています。

表 A−22. 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 int	タグ・インデックス
4 ～ 5	—	未使用 (ゼロが埋め込まれます)
6 ～ 7	unsigned short int	構造体、共用体、または列挙型のサイズ
8 ～ 17	—	未使用 (ゼロが埋め込まれます)

A.7.8.5 関数

表 A−23 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ています。

表 A−23. 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 int	タグ・インデックス
4 ～ 7	long int	関数のサイズ (ビット数)
8 ～ 11	long int	行番号へのファイル・ポインタ
12 ～ 15	long int	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 (ゼロが埋め込まれます)

A.7.8.6 配列

表 A−24 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ています。多次元配列は、4 次元に制限されます。この制限は、(−gw シェル・オプションでコンパイルされる) DWARF フォーマットを使用することにより回避できます。

表 A−24. 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 int	タグ・インデックス
4 ～ 5	unsigned short int	行番号宣言
6 ～ 7	unsigned short int	配列のサイズ
8 ～ 9	unsigned short int	1 次元
10 ～ 11	unsigned short int	2 次元
12 ～ 13	unsigned short int	3 次元
14 ～ 15	unsigned short int	4 次元
16 ～ 17	—	未使用 (ゼロが埋め込まれます)

A.7.8.7 ブロックおよび関数の終了

表 A−25 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ています。

表 A−25. 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 (ゼロが埋め込まれます)
4 ～ 5	unsigned short int	C ソースの行番号
6 ～ 17	—	未使用 (ゼロが埋め込まれます)

A.7.8.8 ブロックおよび関数の開始

表 A−26 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ています。

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 (ゼロが埋め込まれます)
4 ～ 5	unsigned short int	C ソースの行番号
6 ～ 11	—	未使用 (ゼロが埋め込まれます)
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 (ゼロが埋め込まれます)

A.7.8.9 構造体、共用体、および列挙型に関連する名前

表 A−27 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ています。

表 A−27. 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 (ゼロが埋め込まれます)
6 ～ 7	unsigned short int	構造体、共用体、または列挙型のサイズ
8 ～ 17	—	未使用 (ゼロが埋め込まれます)
16 ～ 17	—	未使用 (ゼロが埋め込まれます)

付録 B

シンボリック・デバッグ疑似命令

TMS320C54x™ アセンブラは、TMS320C54x C/C++ コンパイラがシンボリック・デバッグに使用するいくつかの疑似命令をサポートします。

- **.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** の疑似命令は、それぞれ構造体、列挙型、および共用体を定義します。**.member** 疑似命令は、構造体、列挙型、または共用体のメンバを指定します。**.eos** 疑似命令は、構造体、列挙型、または共用体の定義を終了させます。
- **.func** および **.endfunc** の疑似命令は、C/C++ 関数の最初と最後の行を指定します。
- **.block** および **.endblock** の疑似命令は、C/C++ ブロックの境界を指定します。
- **.file** 疑似命令は、現行のソース・ファイル名を識別するシンボルを、シンボル・テーブル内に定義します。
- **.line** 疑似命令は、C/C++ ソース文の行番号を識別します。

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 −g オプションを指定してコンパイラ・シェルを起動します。

```
cl500 –g input file
```

この付録では、シンボリック・デバッグ疑似命令についての説明をアルファベット順に記載しています。.file 疑似命令以外は、それぞれの疑似命令について、C/C++ ソースの例と結果のアセンブリ言語コードを示しています。

構文

.block *[beginning line number]*
.endblock *[ending line number]*

説明

.block および **.endblock** の疑似命令は、C/C++ ブロックの開始および終了を指定します。*line number* は任意選択で、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次に示します。

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 8
.sym   _a,2,4,1,16
.sym   _b,3,4,1,16
.line  9
     LD        *SP(3),A              ; cycle 3
     STL       A,*SP(2)              ; cycle 4
.endblock    10
```

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場所を C/C++ ソース・ファイルの行に逆にマップできます。*filename* は、オリジナルの C/C++ ソース・プログラムが入っているファイルの名前です。ファイル名の最初の 14 文字が意味をもちます。

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、プログラムを読みやすくすることもできます。

例

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 ソースが格納されている例を次に示します。

```
.file      ″text.c″
```

構文

.func *[beginning line number]*
.endfunc *[ending line number]*

説明

.func および **.endfunc** の疑似命令は、C/C++ 関数の開始および終了を指定します。*line number* は任意選択で、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。関数定義はネストさせることができません。

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次に示します。

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:

```
8                       .global _power
9                       .sym    _power,_power,36,2,0
10                      .func   3
11
12              ;***************************************************
13              ;* FUNCTION DEF: _power                            *
14              ;***************************************************
15 000000       _power:
16 000000 eefd          FRAME     #-3
17 000001 f495          nop
18              ;* A     assigned to _x
19                      .sym    _x,0,4,17,16
20                      .sym    _n,4,4,9,16
21                      .sym    _x,0,4,1,16
22                      .sym    _i,1,4,1,16
23                      .sym    _p,2,4,1,16
24                      .line   3
25 000002 8000          STL       A,*SP(0)
26                      .line   5
27 000003 7602          ST        #1,*SP(2)
   000004 0001
28                      .line   6
29 000005 7601          ST        #1,*SP(1)
   000006 0001
30 000007 f7b8          SSBX      SXM
31 000008 f495          nop
32 000009 1004          LD        *SP(4),A
33 00000a 0801          SUB       *SP(1),A
34 00000b f843          BC        L3,ALT
   00000c 0018'
35                      ; branch occurs
36 00000d       L2:
37                      .line   7
38 00000d 4400          LD        *SP(0),16,A
39 00000e 3102          MPYA      *SP(2)
40 00000f 8102          STL       B,*SP(2)
41                      .line   6
42 000010 6b01          ADDM      #1,*SP(1)
   000011 0001
43 000012 f7b8          SSBX      SXM
44 000013 f495          nop
45 000014 1004          LD        *SP(4),A
46 000015 0801          SUB       *SP(1),A
47 000016 f842          BC        L2,AGEQ
   000017 000d'
48                      ; branch occurs
49 000018       L3:
50                      .line   8
51 000018 1002          LD        *SP(2),A
52                      .line   9
53 000019 ee03          FRAME     #3
54 00001a fc00          RET
55                      ; return occurs
56                      .endfunc        9,000000000h, 3
```

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・デバッグで使用され、オブジェクト・コード内のアドレスを、そのコードを生成したソース・コードの行に関連付けます。

.line 疑似命令には、次の 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/C++ ソースの行を示します。行番号は、カレント関数の始めから数えた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。これは任意のパラメータです。アドレスを指定しない場合、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たアセンブリ言語のソース文が置かれます。たとえば、次に示す C ソース行がオリジナルの C ソースの行 6 と行 7 である場合、行 6 と 7 には以下に示すアセンブリ言語のソース文が生成されます。

C ソース:

```
for (i = 1; i <= n; ++i)
     p = p * x;
```

結果のアセンブリ言語コード:

```
31                      .line   7
32 00000d 4400          LD        *SP(0),16,A          ; cycle 1
33 00000e 3102          MPYA      *SP(2)               ; cycle 2
34 00000f 8102          STL       B,*SP(2)             ; cycle 3
35                      .line   6
36 000010 6b01          ADDM      #1,*SP(1)            ; cycle 4
   000011 0001
37 000012 f7b8          SSBX      SXM                  ; cycle 6
38 000013 f495          nop
39 000014 1004          LD        *SP(4),A             ; cycle 8
40 000015 0801          SUB       *SP(1),A             ; cycle 9
41 000016 f842          BC        L2,AGEQ              ; cycle 10
   000017 000d'
42                      ; branch occurs                ; cycle 15
43 000018       L3:
44                      .line   8
45 000018 1002          LD        *SP(2),A
46                      .line   9
47 000019 ee03          FRAME     #3                   ; cycle 1
48 00001a fc00          RET                            ; cycle 2
49                      ; branch occurs                ; cycle 7
50                      .endfunc        9,000000000h,3
```

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したときのみ有効です。

- *name* は、シンボル・テーブルの中に入れるメンバの名前です。名前の最初の 32 文字が意味をもちます。
- *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/C++ の型です。C/C++ の型の詳細は、付録 A「共通オブジェクト・ファイル・フォーマット」を参照してください。
- *storage class* は、メンバの C/C++ の記憶クラスです。C/C++ の記憶クラスの詳細は、付録 A「共通オブジェクト・ファイル・フォーマット」を参照してください。
- *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宣言しておか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できます。これによって、最大 4 次元までのメンバを指定できます。

パラメータの順序は意味をもちます。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したりできます(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、リスト内の後続のパラメータを指定できます。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、これに対応するアセンブリ言語文の例を次に示します。

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
```

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、列挙型、または共用体の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 文字が意味をもちます。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。これは任意選択のパラメータです。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。

アセンブラは、構造体、列挙型、または共用体をネストさせることはできません。C/C++ コンパイラでは、ネストした構造体を個別に定義し、参照元の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

構造体定義の例を次に示します。

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　共用体定義の例を次に示します。

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　列挙型定義の例を次に示します。

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・テーブルの中に入れる変数の名前です。名前の最初の 32 文字が意味をもちます。
- *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/C++ の型です。C/C++ の型の詳細は、付録 A「共通オブジェクト・ファイル・フォーマット」を参照してください。
- *storage class* は、変数の C/C++ の記憶クラスです。C/C++ の記憶クラスの詳細は、付録 A「共通オブジェクト・ファイル・フォーマット」を参照してください。
- *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 疑似命令によって必ず事前に宣言しておかなければなりません。
- *dims* には、最大 4 つまでの式をコンマで区切って指定できます。これによって、最大 4 次元までの変数を指定できます。

パラメータの順序は意味をもちます。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したりできます (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、リスト内の後続のパラメータを指定できます。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:

```
    .global   _array
    .bss      _array,50,0,0
    .sym      _array,_array,244,2,800,,5,10
    .global   _ptr
    .bss      _ptr,1,0,0
    .sym      _ptr,_ptr,21,2,16
    .global   _str
    .bss      _str,2,0,0
    .sym      _str,_str,8,2,32,_s
    .global   _ext
    .bss      _ext,1,0,0
    .sym      _ext,_ext,4,2,16
```

付録 C

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、ファイルを特定のアプリケーションに変換することが容易になります。

項目 **ページ**

C.1 例で使用する基本コード C-2

C.2 例 1: 2 つの 8 ビット EPROM を対象とした Hex コマンド・ファイルの作成方法 C-3

C.3 例 2: 複数のセクション間のホールを回避する方法 C-8

C.4 例 3: ブート・テーブルの作成方法 C-10

C.5 例 4: LP コア・デバイス用のブート・テーブルの作成方法 C-17

C.1 例で使用する基本コード

この付録では 3 つの主な例を使用して、複数の EPROM メモリ・システム用の Hex コマンド・ファイルを開発する方法、ホールを回避する方法、およびブート・テーブルを作成する方法を説明します。最初の 2 つの例では、例 C−1 に示すアセンブリ・コードを使用しています。

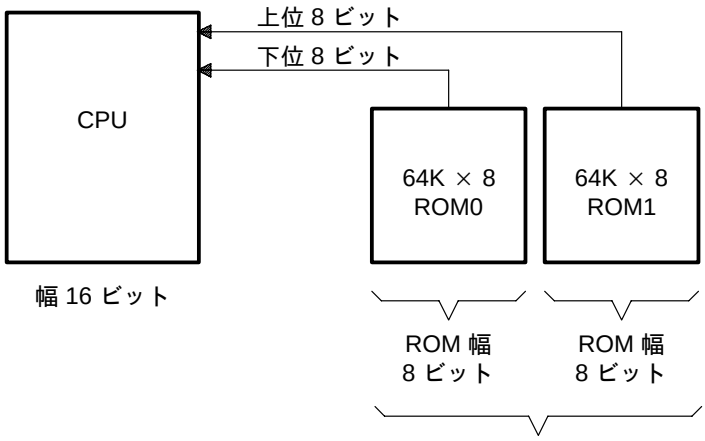
例 C−1. Hex 変換ユーティリティの例で使用するアセンブリ・コード

```
*********************************************************
* 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
```

C.2 例 1: 2 つの 8 ビット EPROM を対象とした Hex コマンド・ファイルの作成方法

例 1 は、図 C−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説明しています。このメモリ・システムには、'C54x ターゲット・プロセッサとのインターフェイスをとる 64K × 8 ビットの外部 EPROM が 2 つあります。各 EPROM は、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 C−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 (0x1400) ではなく、物理 EPROM アドレス 0x0010 にあります。ターゲット・ボードの回路により、このアドレス空間の変換が処理されます。paddr パラメータにより、セクションを割り当てて、コードを EPROMの 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 (詳細は、10-22 ページの 10.6 節「SECTIONS 疑似命令」を参照してください)。変換時に組み込む 1 つのセクションに対してロード・アドレスを paddr パラメータを使用して指定する場合は、同時に組み込む各セクションに対して paddr パラメータ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、sec1 と sec2 の 2 つのセクションが定義されています。これらの各セクションに、SECTIONS 疑似命令の内部から paddr パラメータを容易に追加できます。ただし、多数のセクションから構成されている大きなアプリケーションの場合や、コード開発の途中でセクション・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、セクションをリンク段階で結合して、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 C−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 C−2.　2 つの 8 ビット EPROM 用のリンカ・コマンド・ファイル

```
test.obj
-o test.out
-m test.map

MEMORY
{
   PAGE 0 : EXT_PRG : org = 0x1400 , len = 0xEB80
}

SECTIONS
{
   outsec: { *(sec1)
             *(sec2)  } > EXT_PRG PAGE 0
}
```

この例に示す EPROM プログラマには、以下のシステム要件があります。

- ❏ EPROM システム・メモリ幅は 16 ビットに設定する。
- ❏ 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用に、バイト・インクリメントを選択する (メモリ幅がデフォルト)。

システムの条件をセットアップするには、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、ユーティリティは自動的に 2 つの出力ファイルを作成します。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 C−3 は、選択オプションをすべて指定している Hex コマンド・ファイルを示しています。

例 C−3.　2 つの 8 ビット EPROM 用の Hex コマンド・ファイル

```
test.out        /* COFF object input file               */
-map example1.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 C−2 (a) は、下位 8 ビットが格納されている ROM0 用の変換ファイル (low8.bit) の内容を示しています。図 C−2 (b) は、上位 8 ビットのデータが格納されている ROM1 用の変換ファイル (upp8.bit) の内容を示しています。

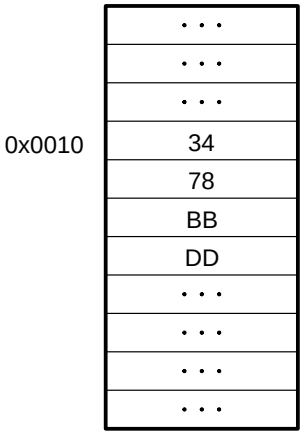
図 C−2.　出力ファイルのデータ

(a) low8.bit (下位ビット)

変換出力ファイルのデータ

```
:0400100034 78BBDDA8
:00000001FF
```

EPROM — ROM0 の対応マップ (C-2 ページの例 C−1 を参照)

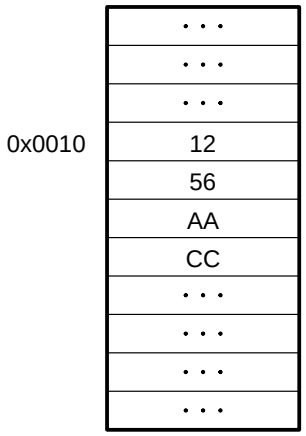


(b) upp8.bit (上位ビット)

変換出力ファイルのデータ

```
:0400100012 56AACC0E
:00000001FF
```

EPROM — ROM1 の対応マップ (C-2 ページの例 C−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出力します。例 C−4 は、結果として生成されるマップを示しています。

例 C−4. C-5 ページの例 C−3 の Hex コマンド・ファイルから出力されるマップ・ファイル

```
*******************************************************
TMS320C54x COFF/Hex Converter             Version x.xx
*******************************************************
Fri Oct 11 15:10:53 2001

INPUT FILE NAME: <test.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 Memory Width=16 ROM Width=8 ”EPROM”
-------------------------------------------------------
   OUTPUT FILES: low8.bit [b0..b7]
                 upp8.bit [b8..b15]

   CONTENTS: 00000010..00000017  outsec Data Width=2
```

C.3 例 2: 複数のセクション間のホールを回避する方法

メモリ幅がデータ幅よりも小さいと、セクションの始めやセクション間にホールが作成されることがあります。このホールは、ロード・アドレスに補正係数を掛けた結果作成されます。詳細は、10-34 ページの 10.10 節「ROM デバイス・アドレスの制御方法」を参照してください。

変換後のセクション間のホールは除去しなければなりません。セクションは、次のいずれかの方法で連続させることができます。

- SECTIONS 疑似命令に指定される各セクションに対して paddr 値を指定します。これにより、Hex 変換ユーティリティは、出力ファイルのアドレス・フィールドにこの特定の値を使用するように強制されます。セクション・アドレスがオーバーラップしないようにする必要があります。例 C−5 (a) は、この方式に使用するリンカ・コマンド・ファイルを示しています。このコマンド・ファイルを使用してリンカを実行します。そのあと、例 C−5 (b) に示されている一連のコマンドを使用して、Hex 変換ユーティリティを実行します。
- 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します。例 C−6 (a) は、この方式に使用するリンカ・コマンド・ファイルを示しています。このコマンド・ファイルを使用してリンカを実行します。そのあと、例 C−6 (b) に示されている一連のコマンドを使用して、Hex 変換ユーティリティを実行します。

例 C−5. ホール回避方式 1

(a) リンカ・コマンド・ファイル

```
/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
   PAGE 0: DARAM: org = 0x0080  , length = 0x1370
             EXT: org = 0x1400  , length = 0xEB80
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
   sec1 : load = EXT PAGE 0
   sec2 : load = EXT PAGE 0
}
```

(b) Hex コマンド・ファイル

```
-i
test.out
-map example.mxp

ROMS
{
 PAGE 0: ROM: org = 0x0000,  length = 0x800, romwidth = 8, memwidth = 8
}

SECTIONS
{
  sec1: paddr = 0x0000
  sec2: paddr = 0x0004
}
```

例 C−6.　ホール回避方式 2

(a) リンカ・コマンド・ファイル

```
/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
  PAGE 0: DARAM: org = 0x0080  , length = 0x1370
            EXT: org = 0x1400  , length = 0xEB80
}

SECTIONS
{
   outsec: { *(sec1)
             *(sec2) } > EXT PAGE 0
}
```

(b) Hex コマンド・ファイル

```
-i
test.out
-map example.mxp

ROMS
{
  PAGE 0: ROM : org = 0x0100, length = 0x0800, romwidth = 8, memwidth = 8,
                files = {examp2_2.hex}
}

SECTIONS
{
 outsec: paddr = 0x100
}
```

C.4 例 3: ブート・テーブルの作成方法

例 3 は、リンカと Hex 変換ユーティリティを使用して、'C54x デバイス用のブート・ロード・テーブルを作成する方法を説明しています。例 C−7 は、この節で使用している C コードを示しています。

注 :

この例は、LP 'C54x 以外のデバイスを対象としています。

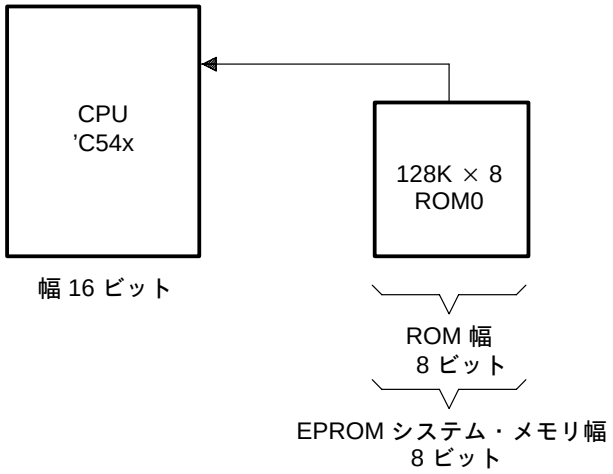
'C54xLP デバイスの場合は、C-17 ページの C.5 節「例 4: LP コア・デバイス用のブート・テーブルの作成方法」を参照してください。

例 C−7. 例 3 で使用する C コード

```
int array[]={1,2,3,4};

main()
{
  array[0] = 5;
}
```

図 C−3 は、出力ファイルが作成される EPROM メモリ・システムを示しています。このアプリケーションでは、単一の 'C54x デバイスは 128K × 8 ビットの EPROM からブートされます。システム要件は、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 C−3. 'C54x 用の EPROM システム



オンチップ・ブート・ローダは、1 つのブロックだけをロードします。このため、TMS320C54x C コンパイラでコンパイルされた C コードをロードしようとすると、問題が発生することがあります。TMS320C54x C コンパイラは、C ソース・コードのコンパイル時に、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、プログラムに関連付けられた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ブート用の 1 つのセクションに結合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ないません。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されます。このカテゴリとは、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、実データは含まれていないセクション) です。TMS320C54x 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は、.text、.cinit、.const、および .data を含みます。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、変換されません。

ほとんどのアプリケーションでは、.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。これにより、C ブート・ルーチン (boot.asm で定義されている c_int00) のコードと情報をロードして実行できます。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 セクションは、リンカ・コマンド・ファイルで 1 つのセクションにリンクされなければなりません。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 シンボルの初期化データとテーブルが設定されています。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処理することに注意してください。

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。
- ❑ セクションの終わりに 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わりを示すマークとして使用されるゼロが含まれます。ただし、.cinit が入力セクションとしてのみ組み込まれている場合は、リンカは cinit を −1 に設定します。これは、初期化テーブルがロードされなかったことを示します。したがって、C ブート・ルーチンである c_int00 は、グローバルまたは静的な C シンボルの初期化を行ないません。

.cinit セクションを .cinit 以外の出力セクションにリンクすると、リンカは前述の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。次の例は、.text と .cinit を boot_sec という名前の 1 つ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 C−8.　1 つのブート・セクションを作成するリンカ・コマンド・ファイル

```
-c
-l rts.lib
-m boot1.map
-o boot.out

MEMORY
{
   PAGE 0 : PROG     : origin = 001400h, length = 01000h
   PAGE 1 : DATA     : origin = 0080h,   length = 01000h
}

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 C−9 は、上記のコマンド・ファイルを使用してリンカを実行するときに出力されるマップ・ファイルの一部を示しています。

例 C−9. 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当て部分

```
SECTION ALLOCATION MAP

 output                                  attributes/
section    page   origin     length       input sections
--------   ----   ---------- ----------   ----------------
boot_sec   0      00001400   0000006e
                    00001400   00000004    boot.obj (.text)
                    00001404   0000002b    rts.lib  : boot.obj (.text)
                    0000142f   00000035             : exit.obj (.text)
                    00001464   00000006    boot.obj (.cinit)
                    0000146a   00000003    rts.lib : exit.obj (.cinit)
                    0000146d   00000001    --HOLE-- [fill = 0000]

.data      1      00000080   00000000     UNINITIALIZED
                    00000080   00000000    boot.obj (.data)
                    00000080   00000000    rts.lib  : exit.obj (.data)
                    00000080   00000000             : boot.obj (.data)

.bss       1      00000080   00000025     UNINITIALIZED
                    00000080   00000004    boot.obj (.bss)
                    00000084   00000000    rts.lib  : boot.obj (.bss)
                    00000084   00000021             : exit.obj (.bss)

.const     1      00000080   00000000     UNINITIALIZED

.sysmem    1      00000080   00000000     UNINITIALIZED

.stack     1      000000a5   00000400     UNINITIALIZED
                    000000a5   00000000    rts.lib : boot.obj (.stack)

GLOBAL SYMBOLS

address  name                            address  name
-------- ----                            -------- ----
00000080 .bss                            00000080 edata
00000080 .data                           00000080 .data
00000400 __STACK_SIZE                    00000080 _array
0000145e _abort                          00000080 .bss
00000080 _array                          000000a5 end
00001448 _atexit                         00000400 __STACK_SIZE
00001404 _c_int00                        00001400 _main
0000142f _exit                           00001404 _c_int00
00001400 _main                           0000142f _exit
00001464 cinit                           00001448 _atexit
00000080 edata                           0000145e _abort
000000a5 end                             00001464 cinit


[12 symbols]
```

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 を使用してセクション boot_sec の終わりにホールを配置したことに注目してください。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。C-12 ページの例 C−8 のコマンド・ファイルを使用し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 が再定義されつつあることを示す警告を発行します。この場合、このような警告は無視してください。

C-12 ページの例 C−8 に示すコマンド・ファイルを使用してリンカを実行すると、COFF ファイルが出力されます。このファイルは、必要なブート・テーブルを作成する Hex 変換ユーティリティへの入力ファイルとして使用できます。

Hex 変換ユーティリティには、EPROM プログラマに関する要件を指定するオプションと、EPROM メモリ・システムを指定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」という唯一の要件があるものとしています。

C-10 ページの図 C−3 に示す EPROM メモリ・システムでは、EPROMシステム・メモリの幅は 8 ビット、物理 ROM 幅は 8 ビットです。システムの要件を反映するには、Hex コマンド・ファイルで次のオプションを設定する必要があります。

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	物理的な ROM 幅を 8 に設定します。

このアプリケーションでは、外部メモリからのブート用のブート・テーブルを作成する必要があるので、システムの要件を反映するには、Hex コマンド・ファイルで次のオプションを設定する必要があります。

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。

例 C−10. COFF ファイルを変換するための Hex コマンド・ファイル

```
boot.out            /* Input COFF file              */
-i                  /* Select Intel format          */

-map boot2.map

-o boot.hex         /* Name the hex output file     */

-memwidth 8         /* Set EPROM system memory width */
-romwidth 8         /* Set physical ROM width       */

-boot               /* Make all sections bootable   */
-bootorg 0x0000     /* Place boot table in EPROM    */
                    /* starting at address 0x0000   */



ROMS
{
   PAGE 0 : ROM : origin = 0x0000, length = 0x20000
}
```

例 3 では、メモリ幅と ROM 幅は同じです。したがって、Hex 変換ユーティリティは 1 つ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 との比によって決まります。

例 C−11 は、−map オプションを指定する例 C−10 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 boot2.map を示しています。

例 C−11. 例 C−10 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

```
*******************************************************
TMS320C54x COFF/Hex Converter             Version x.xx
*******************************************************
Fri Oct 11 15:27:46 2001

INPUT FILE NAME: <boot.out>
OUTPUT FORMAT:   Intel

PHYSICAL MEMORY PARAMETERS
   Default data width:     16
   Default memory width:   8 (MS-->LS)
   Default output width:   8

BOOT LOADER PARAMETERS

OUTPUT TRANSLATION MAP
-------------------------------------------------------------------
00000000..0001ffff  Page=0  Memory Width=8  ROM Width=8  ″ROM″
-------------------------------------------------------------------
   OUTPUT FILES: boot.hex [b0..b7]

   CONTENTS: 00000000..00000138   BOOT TABLE
                     boot_sec : dest=00001400  size=00000139 width=00000002
```

例 C−12 は、例 C−10 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出力ファイル boot.hex を示しています。

例 C−12. 例 C−10 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

```
:20000000140000797 6F800800005FC00771800A66BF8001803FF68F80018FFFEF7B8F7BED9
:20002000F4A0F6B7F6B5F6B6F020146DF1000001F84D142BF07314257EF80012F00000010C
:2000400047F800117E9200F80011F00000017EF80011F00000016C89141AF0741400F074CF
:2000600014674A117211008410F80011FA4514444A16EEFF4811F00000868816F495F49527
:2000800010EEFFFFF4E36CE9FFFF143E10F80085F845144B10F80085F4E3F495F073144C0F
:2000A000F7B811F80084F3100020FA4B145BF4954A11F2731465F495E80172110084491198
:2000C00080E10086F3000001E80081F800848A11FC00EEFFE801F074142FEE01FC0000045D
:1800E000008000010002000300040001008400000001008500000000073
:00000001FF
```

C.5 例 4 : LP コア・デバイス用のブート・テーブルの作成方法

例 4 は、リンカと Hex 変換ユーティリティを使用して、'C54xLP デバイスのブート・ロード・テーブルを作成する方法について説明しています。'C54xLP デバイスには、複数のセクションを指定できます。したがって、LP 以外のデバイスのように、リンク時にセクションをまとめる必要はありません。

注:

この例は、'C54xLP デバイスだけを対象としています。

LP 'C54x 以外のデバイスについては、C-10 ページの C.4 節「例 3: ブート・テーブルの作成方法」を参照してください。

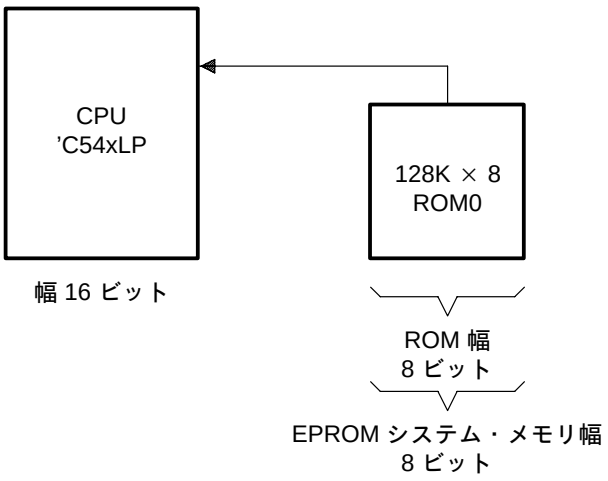
例 C−13. 'C54xLP 用の C コード

```
int array[]={1,2,3,4};

main()
{
  array[0] = 5;
}
```

この例では、−v548 シェル・オプションを使用してコンパイルされています。

図 C−4 は、出力ファイルが作成される EPROM メモリ・システムを示しています。このアプリケーションでは、単一の 'C54xLP デバイスは 128K × 8 ビット EPROM からブートされます。システム要件は、ブート・テーブルが EPROM メモリ・アドレス 0 に位置していることです。

図 C−4. 'C54xLP 用の EPROM システム



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されます。このカテゴリとは、初期化されたセクション (データまたはコードが格納されているセクション) と、初期化されないセクション (空間は確保されているが、実データは含まれていないセクション) です。TMS320C54x C/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は、.text、.cinit、.const、および .data を含みます。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、変換されません。

ほとんどのアプリケーションでは、.text セクションと .cinit セクションはブート時に組み込まれます。これにより、C ブート・ルーチン (boot.asm で定義されている c_int00) のコードと情報をロードして実行できます。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。

.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 シンボルの初期化データとテーブルが設定されています。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処理することに注意してください。

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。
- ❑ セクションの終わりに 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わりを示すマークとして使用されるゼロが含まれています。ただし、.cinit が入力セクションとしてのみ組み込まれている場合は、リンカは cinit を −1 に設定します。これは、初期化テーブルがロードされなかったことを示します。したがって、C ブート・ルーチンである c_int00 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と、リンカは前述の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。例 C−14 は、'C54xLP デバイスのリンカ・コマンド・ファイルを示しています。

例 C−14. ’C54xLP のリンカ・コマンド・ファイル

```
-c
c54xlp.obj
-l rts.lib
-m c54xlp.map
-o c54xlp.out

MEMORY
{
   PAGE 0 : PROG     : origin = 001400h, length = 01000h

   PAGE 1 : DATA     : origin = 0080h,   length = 01000h
}

SECTIONS
{
   .text     : {} > PROG   PAGE 0
   .cinit    : {} > PROG   PAGE 0
   .data     : {} > DATA   PAGE 1
   .bss      : {} > DATA   PAGE 1
   .const    : {} > DATA   PAGE 1
   .sysmem   : {} > DATA   PAGE 1
   .stack    : {} > DATA   PAGE 1
}
```

例 C−15 は、例 C−14 のコマンド・ファイルを使用してリンカを実行したときに作成されるマップ・ファイルを示しています。このコマンド・ファイルを使用してリンクを実行した結果作成される COFF ファイルは、必要なブート・テーブルを作成する Hex 変換ユーティリティへの入力として使用できます。

例 C−15. 例 C−14 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当て部分

```
OUTPUT FILE NAME:   <c54xlp.out>
ENTRY POINT SYMBOL: ”_c_int00”  address: 00001404


MEMORY CONFIGURATION

         name     origin    length      used    attributes    fill
        --------  --------  ---------- --------  ---------- --------
PAGE 0: PROG      00001400  000001000 0000007a  RWIX

PAGE 1: DATA      00000080  000001000 00000426  RWIX
```

例 C−15. 例 C−14 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当て部分 (続き)

```
SECTION ALLOCATION MAP

output                                  attributes/
section  page   origin        length       input sections
-------- ----   ----------    ----------   ----------------
.text    0      00001400      0000006d
                  00001400      00000004    c54xlp.obj (.text)
                  00001404      0000002b    rts.lib : boot.obj (.text)
                  0000142f      0000003e            : exit.obj (.text)
.cinit   0      0000146d      0000000d
                  0000146d      00000006    c54xlp.obj (.cinit)
                  00001473      00000006    rts.lib : exit.obj (.cinit)
                  00001479      00000001    --HOLE-- [fill = 0000]
.data    1      00000080      00000000    UNINITIALIZED
                  00000080      00000000    c54xlp.obj (.data)
                  00000080      00000000    rts.lib : exit.obj (.data)
                  00000080      00000000            : boot.obj (.data)
.bss     1      00000080      00000026    UNINITIALIZED
                  00000080      00000004    c54xlp.obj (.bss)
                  00000084      00000000    rts.lib : boot.obj (.bss)
                  00000084      00000022            : exit.obj (.bss)
.const   1      00000080      00000000    UNINITIALIZED
.sysmem  1      00000080      00000000    UNINITIALIZED
.stack   1      000000a6      00000400    UNINITIALIZED
                  000000a6      00000000    rts.lib : boot.obj (.stack)

GLOBAL SYMBOLS

address  name                           address   name
-------- ----                           --------  ----
00000080 .bss                           00000001  __lflags
00000080 .data                          00000080  _array
00001400 .text                          00000080  .data
0000144b C$$EXIT                        00000080  .bss
00000400 __STACK_SIZE                   00000080  edata
00000085 __cleanup_ptr                  00000085  __cleanup_ptr
00000001 __lflags                       000000a6  end
00001467 _abort                         00000400  __STACK_SIZE
00000080 _array                         00001400  .text
0000144e _atexit                        00001400  _main
00001404 _c_int00                       00001404  _c_int00
0000142f _exit                          0000142f  _exit
00001400 _main                          0000144b  C$$EXIT
0000146d cinit                          0000144e  _atexit
00000080 edata                          00001467  _abort
000000a6 end                            0000146d  etext
0000146d etext                          0000146d  cinit
ffffffff pinit                          ffffffff  pinit

[18 symbols]
```

Hex 変換ユーティリティには、EPROM プログラマに関する要件を指定するオプションと、EPROM メモリ・システムを指定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」という唯一の要件があるものとします。

C-17 ページの図 C−4 に示す EPROM メモリ・システムでは、EPROMシステム・メモリの幅は 8 ビット、物理 ROM 幅は 8 ビットです。システムの要件を反映するには、次のオプションを選択する必要があります。

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	物理的な ROM 幅を 8 に設定します。

このアプリケーションでは、外部メモリからのブート用のブート・テーブルを作成する必要があるので、次のオプションも同様に選択する必要があります。

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。

例 C−16.　COFF ファイルを変換するための Hex コマンド・ファイル

```
c54xlp.out          /* Input COFF file                */
-i                  /* Select Intel format            */

-map c54xlp.mxp     /* Name hex utility map file      */

-o c54xlp.hex       /* Name the hex output file       */

-memwidth 8         /* Set EPROM system memory width  */
-romwidth 8         /* Set physical ROM width         */

-boot               /* Make all sections bootable     */
-bootorg 0x0000     /* Place boot table in EPROM      */
                    /* starting at address 0x0000     */



ROMS
{
   PAGE 0 : ROM : origin = 0x0000, length = 0x20000
}
```

例 4 では、メモリ幅と ROM 幅は同じです。したがって、Hex 変換ユーティリティは 1 つ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 との比によって決まります。

例 C−17 は、−map オプションを指定する例 C−16 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 c54xlp.mxp を示しています。

例 C−17. 例 C−16 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

```
*****************************************************
TMS320C54x COFF/Hex Converter             Version x.xx
*****************************************************
Sat Sep 21 17:01:13 2001

INPUT FILE NAME: <c54xlp.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  Memory Width=8  ROM Width=8  ”ROM”
------------------------------------------------------------------------
   OUTPUT FILES: c54xlp.hex [b0..b7]

   CONTENTS: 00000000..00000109   BOOT TABLE
                                    .text : dest=00001400  size=0000006d
width=00000002
                                    .cinit : dest=0000146d  size=0000000d
width=00000002
```

例 C−18 は、例 C−16 のコマンド・ファイルから作成される Hex 変換ユーティリティ出力ファイル C54xlp.hex を示しています。

例 C−18. 例 C−16 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

```
:2000000008AA7FFFF800FFFFFFFF00870000140076F800800005FE00E800F495771800A68A
:200020006BF8001803FF68F80018FFFEF7B8F7BEF6B9F4A0F6B7F6B5F6B6F0201487F10087
:200040000001F84D1430F273142AF6B8F4957EF80012F000000147F800117E9200F800115A
:20006000F00000017EF80011F00000016C89141FF7B8EEFCF020FFFFF1000001F84D1444B9
:20008000F273143E4E02F495F5E356027E001100FA4C143C6B030001F6B8EE04F0741400F4
:2000A000F074144A4A114A16721100A410F80011FA451460F495EEFF4811F00000848816EF
:2000C000F495F49510EEFFFFF4E36CE9FFFF145A10F800A5F845146710F800A5F4E3F0742D
:2000E0001484EE018A168A11FC00F7B8E9204A1109F800A4F84E1478F2731482F495E8014B
:1E010000721100A4491180E10084F3000001E80081F800A48A11FC00F495F073148566
:20011E00000D000014870004008000010002000300040001 00A40000000100A50000000040
:02013E000000BF
:00000001FF
```

付録 D

アセンブラのエラー・メッセージ

アセンブラは、2 番目のパスを完了した時点で、アセンブル実行中に検出されたエラーを報告します。また、リスト・ファイルを作成しておけば、これらのエラーはこのリスト・ファイルにも出力されます。エラーは、エラーが発生したソース行の後に出力されます。コードに最初に発生した最初のエラーを修正してみる必要があります。単一のエラー状況は、偽のエラーのカスケードの原因になります。

アセンブラのエラー・メッセージを受け取った場合は、この付録の記述を読んで、遭遇した問題に対する可能な解決法を探してください。まず、エラー・メッセージのクラス番号の位置を見つけます。 その後、そのクラス内で遭遇したエラー・メッセージの位置を見つけます。それぞれのクラス番号には、関連するエラー・メッセージのアルファベット順のリストがあります。
各クラスにはその問題についての説明があり、考えられる解決法を示した処置の項目があります。

E0000

Attempt to nest repeat block
Comma required to separate arguments
Comma required to separate parameters
Commas must separate directive elements
Illegal combination of shift operands
Illegal identifier after keyword unsigned
Illegal instruction
Illegal keyword
Illegal repeat block open – check delay slot
Illegal repeat block open – missing 'repeat'
Illegal shift for parallel operation
Left parenthesis expected
Left parenthesis is missing
Matching right parenthesis is missing
Missing comma
Missing left parenthesis
Missing opening brace
Missing right parenthesis
Missing right parenthesis for value specification
Missing right quote of string constant
No matching right parenthesis
Open repeat block at EOF
Right parenthesis expected
Syntax Error
Syntax requires parentheses
Unrecognized character type

説明　一般的な構文に関するエラーです。必要な構文がありません。

処置　エラー・メッセージ・テキストに従ってソースを修正してください。

E0002

Invalid mnemonic specification

説明　無効なニーモニックに関するエラーです。指定された命令、マクロ、または疑似命令は認識されませんでした。

処置　使用した疑似命令または命令を調べてください。

Invalid instruction for specified processor version

説明　.version 疑似命令で指定されたプロセッサ・バージョンに対して、無効な命令が指定されています。

処置　命令、および .version 疑似命令を調べてください。

E0003

Cluttered character operand encountered
Cluttered identifier operand encountered
Cluttered register operand encountered
Cluttered string constant operand encountered
Conditionals cannot begin in the first column
Condition must be EQ, LT, GT, or NEQ
Condition must be srcEQ, NEQ, LT, LEQ, GT, or GEQ
Expecting ARn for src,dst
Expecting shift or accumulator
Illegal auxiliary register specified
Illegal condition operand
Illegal condition operand or combination
Illegal indirect memaddr specification
Illegal operand
Illegal smem operand
Immediate value out of range
Invalid binary constant specified
Invalid constant specification
Invalid decimal constant specified
Invalid float constant specified
Invalid hex constant specified

Invalid immediate or shift value
Invalid octal constant specified
Invalid Operand 2
Invalid operand *x*
Invalid shift value
Must add AR0 to destination
Must subtract AR0 from destination
Only labels and comments may begin in first column
Operand must be the A accumulator
Operand must be the B accumulator
Section is not defined
Shift value must be 16
Syntax error – Operand nnn
The accumulator arguments must be the same
The accumulator arguments must not be the same
The dst accumulator arguments must be the same
The dst,src1 arguments must be the same
The smem operands must be the same
Unexpected parallel instruction delimiters

説明　無効なオペランドに関するエラーです。指定された命令、パラメータ、または他のオペランドは認識されませんでした。

処置　エラー・メッセージ・テキストに従ってソースを修正してください。

E0004

Absolute, well-defined integer value expected
Expecting accumulator A or B
Expecting ASM or shift value
Expecting dual memory addressing
Identifier expected
Identifier operand expected
Illegal character argument specified
Illegal combination of Smem operands
Illegal floating-point expression
Illegal operand
Illegal shift operation
Illegal structure reference
Incorrect bit symbol for specified status register
Invalid data size for relocation
Invalid float constant specified
Invalid identifier, *sym*, specified
Invalid macro parameter specified
Invalid operand, ”char”
Must add to the destination operand
No parameters available for macro arguments
Not expecting direct operand *op*
Not expecting immediate value operand *op*
Not expecting indirect operand *op*
Offset Addressing modes not legal for MMR addressing
Operand must be auxiliary register or SP
Operand must be auxiliary register
Operand must be T
Register must be ARn or SP
Single character operand expected
String constant or substitution symbol expected
String operand expected
Structure/Union tag symbol expected
Substitution symbol operand expected
The accumulator operands must be different
The operands must be SP

説明　不正なオペランドに関するエラーです。指定された命令、パラメータ、または他のオペランドは、この構文では正しくありません。

処置　エラー・メッセージ・テキストに従ってソースを修正してください。

E0005

Missing field value operand
Missing operand(s)
Operand missing
Tag identification operand required
Tag symbol identifier required

説明　オペランドの欠如に関するエラーです。必要なオペランドが指定されていません。

処置　必要なすべてのオペランドが宣言されるように、ソースを修正してください。

E0006

.break must occur within a loop
Conditional assembly mismatch
Matching .endloop missing
No matching .if specified
No matching .endif specified
No matching .endloop specified
No matching .if specified
No matching .loop specified
Open block(s) inside macro
Unmatched .endloop directive
Unmatched .if directive

説明　対応していない条件付きアセンブリ疑似命令に関するエラーです。ある疑似命令には一致する疑似命令が必要でしたが、アセンブラはこれを見つけられませんでした。

処置　エラー・メッセージ・テキストに従ってソースを修正してください。

E0007

Conditional nesting is too deep
Loop count out of range

説明　条件付きアセンブリ・ループに関するエラーです。条件ブロックの最大ネスト・レベルは 32 レベルです。

処置　.macro/.endmacro ソース、.if/.elseif/.else/.endif ソース、または .loop/.break/.endloop ソースを修正してください。

E0008

Bad use of .access directive
Matching .struct directive is not present
Matching .union directive is not present

説明　一致しない構造体定義疑似命令に関するエラーです。.struct/.endstruct シーケンス内で、ある疑似命令には一致する疑似命令が必要でしたが、アセンブラはこれを見つけられませんでした。

処置　ソースを調べて、一致しない構造体定義疑似命令を修正してください。

E0009

Cannot apply bitwise NOT to floats
Illegal struct/union reference dot operator
Missing structure/union member or tag
Section ”name” is not an initialized section
Structure or union tag symbol expected
Structure or union tag symbol not found

説明　演算子が不正に使用されていることを示すエラーです。指定されたオペランドにとって不当な演算子が指定されました。

処置　エラー・メッセージ・テキストに従って、必要なオペランドがすべて宣言されるように、ソースを修正してください。

E0100

Label missing
Label required
.setsym requires a label

説明　必要なラベルに関するエラーです。指定された疑似命令にはラベルが必要でしたが、これが指定されていません。

処置　ソースを修正して必要なラベルを指定してください。

E0101

Labels are not allowed with this directive
Standalone labels not permitted in structure/union defs

説明　無効なラベルがあることを示すエラーです。ラベルを指定できない疑似命令に、ラベルが指定されています。

処置　無効なラベルを削除してください。

E0102

Local label *number* defined differently in each pass
Local label *number* is multiply defined
Local label *number* is not defined in this section
Local labels can't be used with directives

説明　ローカル・ラベルが不当に使用されていることを示すエラーです。

処置　エラー・メッセージ・テキストに従ってソースを修正してください。ローカル・ラベルを再利用するには .newblock を使用します。

E0200

Bad term in expression
Binary operator can't be applied
Cannot resolve symbol in expression
Difference between segment symbols not permitted
Expression evaluation failed
Illegal divide by zero
Illegal remainder by zero
Integer divide by zero
Integer remainder by zero
Offset expression must be integer value
Operation cannot be performed on given operands
Unable to compose expression
Unary operator can't be applied
Value of expression has changed due to jump expansion
Well-defined expression required

説明　一般的な式に関するエラーです。使用されているオペランドの組み合わせが無効であるか、または必要な算術型が指定されていません。

処置　エラー・メッセージ・テキストに従ってソースを修正してください。

E0201

Absolute operands required for FP operations!
Floating–point divide by zero
Floating–point overflow
Floating–point underflow
Floating–point expression required
Illegal floating–point expression
Invalid floating–point operation

説明　浮動小数点式に関するエラーです。整数式が必要な箇所に浮動小数点式が使用されたか、浮動小数点式が必要な箇所に整数式が使用されたか、または浮動小数点値が無効であったかです。

処置　エラー・メッセージ・テキストに従ってソースを修正してください。

E0300

Cannot equate an external symbol to an external
Cannot redefine this section name
Cannot tag an undefined symbol
Empty structure or union definition
Illegal structure or union tag
Missing closing '}' for repeat block
Redefinition of "sym" attempted
Structure tag can't be global
Symbol can't be defined in terms of itself
Symbol expected
Symbol expected in label field
Symbol, *sym*, has already been defined
Symbol, *sym*, is not defined in this source file
Symbol, *sym*, is operand to both .ref and .def
Structure/union member, *sym*, not found
The following symbols are undefined:
Union member previously defined
Union tag can't be global

説明　一般的なシンボルに関するエラーです。シンボルを再定義しようとしたか、またはシンボルを不正な方法で定義しようとしました。

処置　エラー・メッセージ・テキストに従ってソースを修正してください。

E0301

Cannot redefine local substitution symbol
Substitution Stack Overflow
Substitution symbol not found

説明　一般的な置換シンボルに関するエラーです。シンボルを再定義しようとしたか、またはシンボルを不正な方法で定義しようとしました。

処置　エラー・メッセージ・テキストに従ってソースを修正してください。置換シンボルのオペランドが、マクロ・パラメータとして定義されているか、または .asg 疑似命令あるいは .eval 疑似命令によって定義されているかどうかを確認してください。

E0400

Symbol table entry is not balanced

説明　シンボリック・デバッグ疑似命令に、これを補足する疑似命令がありません (.block に対して .endblock がない場合など)。

処置　ソース内の一致しない条件付きアセンブリ疑似命令を調べてください。

E0500

Macro argument string is too long
Missing macro name
Too many variables declared in macro

説明　一般的なマクロに関するエラーです。マクロ定義に誤りがある可能性があります。

処置　エラー・メッセージ・テキストに従ってソースを修正してください。

E0501

Macro definition not terminated with .endm
Matching .endm missing

Matching .macro missing
.mexit directive outside macro definition
No active macro definition

説明　マクロ定義疑似命令に関するエラーです。マクロ疑似命令に、これを補足する疑似命令がありません (.macro に対して .endm がない場合など)。

処置　エラー・メッセージ・テキストに従ってソースを修正してください。

E0600

Bad archive entry for *macro name*
Bad archive name
Can’t read a line from archive entry
***library name* macro library not found**
***library name* is not in archive format**

説明　マクロ・ライブラリへのアクセスに関するエラーです。マクロ・ライブラリのアーカイブ・ファイルからの読み取り、またはアーカイブ・ファイルへの書き込み中に問題が発生しました。アーカイブ・ファイルが正しく作成されなかった可能性があります。

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、ファイルの拡張子が .asm であることを確認してください。

E0700

Can’t use –g on assembly code with .line directives
Illegal structure/union member
No structure/union currently open
.sym not allowed inside structure/union

説明　シンボリック・デバッグ疑似命令が不正に使用されていることを示すエラーです。シンボリック・デバッグ疑似命令が適切な位置で使用されていません。

処置　エラー・メッセージ・テキストに従ってソースを修正してください。

E0800

Control flow change in delayed branch slot
Instructions not permitted in structure/union definitions
Parallel operator without instruction
Too many words in delayed branch slot

説明　　ブランチ命令に関するエラーです。これらのエラーは通常、ターゲット固有です。

処置　　エラー・メッセージ・テキストに従ってソースを修正してください。

E0801

Too many parallel instructions

説明　　多くの命令が同時に存在することが原因のエラーです。

処置　　並行する命令の問題についてソースを調べ、エラー・メッセージ・テキストに従って修正してください。

E0900

Cannot change version after 1st instruction
Cannot change parsing mode after 1st instruction
Can’t include a file inside a loop or macro
Illegal structure member
Illegal structure definition contents
Illegal union member
Illegal union definition contents
Invalid load-time label
Invalid structure/union contents
.setsect only valid if absolute listing produced (use –a option)

.setsym only valid if absolute listing produced (use –a option)
.var allowed only within macro definitions

説明 疑似命令が不正に使用されていることを示すエラーです。特定の疑似命令が、許可されていない場所で使用されています。疑似命令がその場所での使用を禁止されているのは、オブジェクト・ファイルを壊す可能性があるからです。多くの疑似命令は、構造体または共用体の定義内では使用禁止です。

処置 エラー・メッセージ・テキストに従ってソースを修正してください。

E1000

Include/Copy file not found or opened

説明 指定されたファイル名が見つかりません。

処置 ファイル名のスペル、パス名、環境変数などを確認してください。

E1300

Copy limit has been reached
Exceeded limit for macro arguments
Macro nesting limit exceeded

説明 一般的なアセンブラの制限を超えたことを示すエラーです。.copy/.include ファイルの最大ネスト・レベルは 10 レベルです。マクロ引数として指定できるパラメータの最大数は 32 です。マクロの最大ネスト・レベルは 32 レベルです。

処置 ソースを調べて、制限の超過状態を確認してください。

E9999

Pass conflict

説明 アセンブラの内部エラーです。繰り返して発生する場合は、アセンブラが壊れているか、または混乱しているかもしれません。

処置 より小さいファイルをアセンブルしてください。小さいファイルがアセンブルしない場合は、アセンブラを再インストールしてください。

Pipeline conflict detected

説明 パイプライン・コンフリクトを報告するエラーです。

処置 ソースを調べて問題の原因を判別し、ソースを修正してください。

Illegal use of *(EXPR) notation
Illegal use of *SP notation

説明 記述方法の問題を報告するエラーです。

処置 ソースを調べて問題の原因を判別し、ソースを修正してください。

W0000

Ambiguous assignment
Invalid page number specified – ignored
Macro parameter conflict
No operands expected. Operands ignored
Specified alignment is outside accessible memory – ignored
Trailing operand does not exist
Trailing operands ignored
Unrecognized operand, ignored
***+ARn addressing is for write-only**

説明 オペランドに関する警告です。アセンブラが、予期しなかったオペランドを検出しました。

処置 ソースを調べて、問題の原因、およびソースを修正する必要があるかどうかを判別してください。

W0001

Field value truncated to *value*
Field width truncated to *size in bits*
Immediate value out of range
Legal shift values are ...
Maximum alignment is to 32K boundary – alignment ignored
Offset expression – value out of range
Power of 2 required, *next larger power of 2* assumed
Section Name is limited to 8 characters
Section name, *name*, truncated to 8 chars
Set value must be 0 or 1
Shift value out of range
Status bit value out of range
Status register must be 0 or 1
String is too long – will be truncated
Valid values are 1 and 2
Value values to *xxx* are *nnn*
Value truncated
Value truncated to x–bit width
Value truncated to byte size
Value out of range

説明　切り捨てられた値に関する警告です。指定された式は長すぎて、命令コードまたは必要なビット数内に収まりません。

処置　ソースを調べて、式の結果が許容可能であることを確認してください。または、エラーが発生した場合はソースを変更してください。

W0002

Address expression will wrap–around
Expression will overflow, value truncated

説明　算術式に関する警告です。この結果の原因は、アセンブラが行った計算です。結果は、許容できるものと許容できないものがあります。

処置　結果が許容できるものであることを検証してください。エラーが発生している場合は、ソースを変更してください。

W0003

Incorrect size for the type
.sym for *function name* required before .func

説明　　シンボリック・デバッグ疑似命令の問題に関する警告です。.func 疑似命令の前に、関数を定義する .sym 疑似命令がありません。

処置　　ソースを修正してください。

W0004

.access only allowed in top-most structure definition
Access point has already been defined
Open block(s) at EOF

説明　　構造体定義の問題に関する警告です。

処置　　エラー・メッセージ・テキストに従ってソースを修正してください。

W9999

A branch to an empty label just inside the loop-closing brace is a branch out of the loop
Far mode valid only for C548
First instruction following XC must be a 1-word instruction
Open branch delay slot at end of section
Power of 2 required, *next larger power of 2* assumed
Second instruction following XC must be a 1-word instruction
Section *name* absolute address set to 0
Section *name* is larger than 64K
Value truncated to byte size

説明　　一般的な警告です。

処置　　警告メッセージ・テキスト従ってソースを修正してください。

付録 E

リンカのエラー・メッセージ

本付録では、リンカのエラー・メッセージをアルファベット順に掲載しています。このリストでは、シンボル (...) は、エラー発生時にリンカが処理しようとしていたオブジェクト名を示しています。

A

absolute symbol (...) being redefined

説明 絶対シンボルを再定義することはできません。

処置 すべての式の構文を調べてから、入力疑似命令が正確であるかどうかを調べてください。

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 SECTIONS 疑似命令の外部で、シンボルの位置合わせが実行されています。

alignment for (...) must be a power of 2

説明 セクションの位置合わせが 2 の累乗値ではありませんでした。

処置 16 進数で、すべての 2 の累乗値が、整数の 1、2、4 または 8 と、それに続くゼロの列から構成されていることを確認してください。

alignment for (...) redefined

説明　1 つのセクションに対して、複数の位置合わせが指定されています。

attempt to decrement DOT

説明　**.**–= *value* などの文が指定されていますが、これは正しくありません。ドットへの代入は、ホールの作成にのみ使用できます。

B

bad fill value

説明　埋め込み値は 16 ビットの定数でなければなりません。

binding address (...) for section (...) is outside all memory on page (...)

説明　MEMORY 疑似命令によって構成されたメモリの外側に配置されているセクションがあります。

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 SECTIONS 疑似命令で定義された空間が十分であるかをチェックして、未構成のメモリにセクションが配置されないようにしてください。

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ため、この 2 つのセクションを割り当てることができません。

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 SECTIONS 疑似命令で定義された空間が十分であるかをチェックして、セクションがオーバーラップして配置されないようにしてください。

binding address for (...) redefined

説明　1 つのセクションに対して、複数のバインディング値が指定されています。

binding address (...) incompatible with alignment for section (...)

説明　セクションに、.align 疑似命令、または直前のリンクからの位置合わせ要件があります。バインディング・アドレスはこの要件に違反しています。

blocking for (...) must be a power of 2

説明　セクションのブロック化が 2 の累乗値ではありません。

処置　16 進数で、すべての 2 の累乗値が、整数の 1、2、4、または 8 と、それに続くゼロの列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 つのセクションに対して、複数のブロック化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 で終了している必要があります。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 complete output file (...), write error

説明　通常このメッセージは、ファイル・システムに空き空間がないことを示します。

cannot create output file (...)

説明　通常このメッセージは、ファイル名が正しくないことを示します。

処置　ファイル名のスペル、パス名、環境変数などを確認してください。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ファイル名を付けてください。

cannot resize (...), section has initialized definition in (...)

説明　.stack または .heap という名前の初期化された入力セクションが存在するので、リンカはセクションのサイズを変更できません。

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セクションが指定されました。グループ全体が 1 つの単位として扱われるので、メモリのページにグループを指定することはできます。ただし、グループを構成するセクションを個別に操作することはできません。

cannot specify both binding and memory area for (...)

説明　バインディングと名前付きメモリの両方が指定されています。バインディングと名前付きメモリは、相互に排他的です。

処置　特定のアドレスにコードを配置したい場合は、バインディングだけを使用します。

can’t align a section within GROUP – (...) not aligned

説明　個別の位置合わせの対象として、グループ内のセクションが指定されています。グループ全体が 1 つの単位として扱われるので、アドレスにグループをバインドまたは位置合わせすることができます。ただし、グループを構成するセクションは個別に操作することができません。

can’t align within UNION – section (...) not aligned

説明　個別の位置合わせの対象として、共用体内のセクションが指定されています。共用体全体が 1 つの単位として扱われるので、アドレスに共用体をバインドまたは位置合わせすることはできます。ただし、共用体を構成するセクションを個別に操作することはできません。

can’t allocate (...), size ... (page ...)

説明　セクションを保持できるだけの容量をもつ構成メモリがないので、セクションを割り当てることができません。

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 SECTIONS 疑似命令で定義された空間が十分であるかをチェックして、空いている領域にセクションを割り当ててください。

can’t create map file (...)

説明　通常このメッセージは、ファイル名が正しくないことを示します。

処置　ファイル名のスペル、パス名、環境変数などを確認してください。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、ファイル名を付ける必要があります。

can’t find input file *filename*

説明　ファイル *filename* が PATH 上に存在しないか、またはファイル名のスペルが誤っています。

処置　ファイル名のスペル、パス名、環境変数などを確認してください。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、ファイル名を付ける必要があります。

can’t open (...)

説明　指定されたファイルは存在しません。

処置　ファイル名のスペル、パス名、環境変数などを確認してください。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、ファイル名を付ける必要があります。

can’t open *filename*

説明　　指定されたファイルは存在しません。

処置　　ファイル名のスペル、パス名、環境変数などを確認してください。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、ファイル名を付ける必要があります。

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、この入力ファイルを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、この入力ファイルを再アセンブルしてください。

can’t write (...)

説明　　ディスクが満杯か、または保護されている可能性があります。

処置　　ディスクの容量と保護設定を確認してください。

command file nesting exceeded with file (...)

説明　　コマンド・ファイルの最大ネスト・レベルは 16 レベルです。

E

–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 オプションまたは –cr オプションを使用している場合にのみ表示されるメッセージです。値 _c_int00 以外のプログラム・エントリ・ポイントが指定されています。コンパイラの実行時規則は、_c_int00 が唯一のエントリ・ポイントであることを前提としています。

entry point symbol (...) undefined

説明　–e オプションで使用されているシンボルは、未定義のシンボルです。

errors in input – (...) not built

説明　これまでに発生したエラーが原因で、出力ファイルを作成できません。

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、この入力ファイルを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、この入力ファイルを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、この入力ファイルを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、この入力ファイルを再アセンブルしてください。

fail to write (...)

説明　ディスクが満杯か、または保護されている可能性があります。

処置　ディスクの容量と保護設定を確認してください。

file (...) has no relocation information

説明　−r によっ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　バイナリ入力ファイルが COFF ファイルではありません。

fill value for (...) redefined

説明　1 つの出力セクションに対して、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を使用して、個々のホールにそれぞれ異なる値を埋め込むことができます。

I

–i path too long (...)

説明　−i パスの最大文字数は 256 です。

illegal input character

説明　コマンド・ファイルに制御文字、または他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、Xの組み合わせではありません。

illegal operator in expression

説明　式の中で正しい演算子を使用してください。

illegal option within SECTIONS

説明　−l (小文字の L) オプションは、SECTIONS 疑似命令に指定できる唯一のオプションです。

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 (...)

説明　リンカで内部エラーが発生しました。

invalid archive size for file (...)

説明　アーカイブ・ファイルが壊れています。

invalid path specified with –i flag

説明　−i オプション (フラグ) のオペランドが、有効なファイル名またはパス名ではありません。

invalid value for –f flag

説明　−f オプション (フラグ) の値が 2 バイト定数値ではありません。

invalid value for –heap flag

説明　−heap オプション (フラグ) の値が 2 バイト定数値ではありません。

invalid value for –stack flag

説明　−stack オプション (フラグ) の値が 2 バイト定数値ではありません。

invalid value for –v flag

説明　−v オプション (フラグ) の値が定数値ではありません。

I/O error on output file (...)

説明　ディスクが満杯か、または保護されている可能性があります。

処置　ディスクの容量と保護設定を確認してください。

L

length redefined for memory area (...)

説明　MEMORY 疑似命令のメモリ領域に対して、複数の長さが定義されています。

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可能性があります。これは、リンク時に、ライブラリ・メンバが他のファイルの未解決の参照を解決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、この入力ファイルを再アセンブルしてください。

load address for uninitialized section (...) ignored

説明　初期化されないセクションに対して、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実行アドレスがありますが、ロード・アドレスはありません。

load address for UNION ignored

説明　UNION は、セクションの実行アドレスだけを参照します。

load allocation required for initialized UNION member (...)

説明　共用体の初期化されたセクションに対して、ロード・アドレスが指定されています。UNION は、実行時割り当てだけを参照します。共用体のすべてのセクションに対して、ロード・アドレスを個別に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・ファイルの書き込み先ファイルとして指定したファイル名は無効です。

making aux entry *filename* for symbol *n* out of sequence

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、この入力ファイルを再アセンブルしてください。

memory area for (...) redefined

説明　1 つの出力セクションに対して、複数の名前付きメモリ割り当てが指定されています。

memory page for (...) redefined

説明　1 つのセクションに対して、複数のページ割り当て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対して、2 組以上のメモリ属性が指定されています。

memory types (...) and (...) on page (...) overlap

説明　同じページ上の複数のメモリ範囲がオーバーラップしています。

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 SECTIONS 疑似命令で定義された空間が十分であるかをチェックして、セクションがオーバーラップして配置されないようにしてください。

missing filename on –l; use –l <filename>

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 DOT symbol in assignment instruction

説明　SECTIONS 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

no allocation allowed for uninitialized UNION member

説明　共用体の初期化されないセクションに対して、ロード・アドレスが指定されています。共用体の初期化されないセクションは、UNION 文から実行アドレスを取得します。また、このセクションにはロード・アドレスはありません。したがって、メンバに対するロード割り当ては無効です。

no allocation allowed with a GROUP–allocation for section (...) ignored

説明　個別の割り当ての対象として、グループ内のセクションが指定されています。グループ全体が 1 つの単位として扱われるので、アドレスにグループをバインドまたは位置合わせすることはできます。ただし、グループを構成するセクションを個別に操作することはできません。

no input files

説明　COFF ファイルが指定されていません。1 つ以上の入力 COFF ファイルが指定されていないと、リンカは稼動できません。

no load address specified for (...); using run address

説明　初期化されたセクションに対して、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だけをもつ場合は、この実行アドレスで実行およびロードされるように割り当てられます。

no run allocation allowed for union member (...)

説明　UNION は、共用体のすべてのメンバの実行アドレスを定義します。したがって、メンバへの個別の実行アドレスの割り当ては不正な処理です。

no string table in file *filename*

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、この入力ファイルを再アセンブルしてください。

no symbol map produced – not enough memory

説明　シンボル・リストを作成できるだけの使用可能メモリ容量がありません。この状況は致命的ではありませんが、この状況では、マップ・ファイル内でシンボル・リストが生成されなくなります。

O

–o flag does not specify a valid file name : *string*

説明　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、ファイル名を付ける必要があります。

origin missing for memory area (...)

説明　MEMORY 疑似命令で起点が指定されていません。起点によって、メモリ範囲の開始アドレスが指定されます。

out of memory, aborting

説明　システムに、必要なタスクをすべて実行できるだけのメモリ容量がありません。

処置　アセンブリ言語ファイルを小さなファイルに分割してから、部分リンクを実行してください。7-70 ページの 7.16 節「部分 (インクリメンタル) リンク」を参照してください。

output file has no .bss section

説明　これは警告です。通常、.bss セクションは COFF ファイルに記述されます。ただし、このセクションは必須ではありません。

output file has no .data section

説明　これは警告です。通常、.data セクションは COFF ファイルに記述されます。ただし、このセクションは必須ではありません。

output file has no .text section

説明　これは警告です。通常、.text セクションは COFF ファイルに記述されます。ただし、このセクションは必須ではありません。

output file (...) not executable

説明　作成された出力ファイルに、その他のエラーが原因で発生した問題、または未解決のシンボルがある可能性があります。これは致命的エラーではありません。

overwriting aux entry *filename* of symbol *n*

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、この入力ファイルを再アセンブルしてください。

P

PC-relative displacement overflow at address (...) in file (...)

説明　PC 相対分岐を再配置した結果、分岐の変位が大きすぎて命令内にエンコードできませんでした。

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 オプションを指定すると再配置情報が削除されますが、–r オプションを指定すると再配置可能なオブジェクト・ファイルが要求されます。このため、両方のオプションが指定されている場合、両オプションは互いに対立します。

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、この入力ファイルを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、この入力ファイルを再アセンブルしてください。

S

section (...) at (...) overlays at address (...)

説明　2 つのセクションがオーバーラップしているため、この 2 つのセクションを割り当てることができません。

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 SECTIONS 疑似命令で十分な空間が定義されているかをチェックし、セクションがオーバーラップしないようにしてください。

section (...) enters unconfigured memory at address (...)

説明　セクションを保持できるだけの容量をもつ構成メモリがないので、セクションを割り当てることができません。

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 SECTIONS 疑似命令で十分な空間が定義されているかをチェックし、セクションが未構成のメモリ内に配置されないようにしてください。

section (...) not built

説明　ほとんどの場合、SECTIONS 疑似命令に構文エラーがあることを示します。

section (...) not found

説明　SECTIONS 疑似命令に指定されている入力セクションが、入力ファイルで見つかりませんでした。

section (...) won't fit into configured memory

説明　セクションを保持できるだけの容量をもつ構成メモリがないので、セクションを割り当てることができません。

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 SECTIONS 疑似命令で十分な空間が定義されているかをチェックし、セクションが未構成のメモリ内に配置されないようにしてください。

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、この入力ファイルを再アセンブルしてください。

semicolon required after assignment

説明　コマンド・ファイルに構文エラーがあります。

statement ignored

説明　式に構文エラーがあります。

symbol referencing errors — (...) not built

説明　シンボル参照を解決できませんでした。したがって、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　定義済みのシンボルが、代入文で再定義されています。

T

too few symbol names in string table for archive *n*

説明　アーカイブ・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、アーカイブを再作成してください。

too many arguments – use a command file

説明　1 つのコマンド行またはプロンプトに対する応答で、11 以上の引数を指定しました。

too many –i options, 7 allowed

処置　8 つ以上の –i オプションが指定されました。追加の検索ディレクトリを指定する場合は、環境変数 C_DIR または A_DIR を使用して指定します。

type flags for (...) redefined

説明　1 つのセクションに対して、複数のセクション型が指定されました。COPY 型は DSECT 型のすべての属性をもっています。このため、別個に DSECT を指定する必要が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セクションに対して、型が指定されています。特殊なセクション型は、個別のセクションにのみ適用されます。

U

–u does not specify a legal symbol name

説明　–u オプションで指定されたシンボル名は、リンク先ファイルのいずれかに存在する正しいシンボル名ではありません。

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 (...) first referenced in file (...)

説明　参照されたシンボルが定義されていません。または –r オプションが使用されていません。–r オプションが使用されない場合、リンカは、参照されているシンボルがすべて定義済みであることを要求します。この状況では、実行可能な出力ファイルを作成できません。

処置　–r オプションを使用してリンクするか、またはシンボルを定義します。

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。

unrecognized option (...)

処置　　有効なオプションのリストを調べてください。

Z

zero or missing length for memory area (...)

説明　　MEMORY 疑似命令で定義されているメモリ範囲の長さがゼロです。

付録 F

用語集

A

ASCII: 情報交換用米国標準コード
(American Standard Code for Information Exchange)。英数字の情報を表現、交換するための標準コンピュータ・コード

B

.bss: デフォルトの COFF セクションの 1 つ。.bss 疑似命令を使用すると、メモリ・マップに一定量の空間を確保でき、その空間に後でデータを格納することができます。.bss セクションは初期化されません。

C

COFF: 共通オブジェクト・ファイル・フォーマット
(Common Object File Format)。セクションの概念をサポートすることによってモジュラ・プログラミングを促進する、バイナリ・オブジェクト・ファイルのフォーマット

C コンパイラ: C のソース文をアセンブリ言語のソース文に変換するためのプログラム

D

.data: デフォルトの COFF セクションの 1 つ。.data セクションは、初期化されたデータを含む初期化されたセクションです。.data 疑似命令を使用すると、コードを .data セクションにアセンブルできます。

G

GROUP: SECTIONS 疑似命令のオプションの 1 つ。指定された出力セクションを、(グループとして)連続して強制的に割り当てます。

H

Hex 変換ユーティリティ: COFF ファイルを受け取り、そのファイルを、EPROM プログラマにロードできる標準 ASCII 16 進フォーマットの 1 つに変換するプログラム

R

RAM モデル: リンカが C コードをリンクするときに使用する自動初期化モデル。−cr オプションを指定してリンカを起動するとき、リンカは RAM モデルを使用します。RAM モデルを使用すると、実行時ではなくロード時に変数を初期化することができます。

ROM 幅: 各出力ファイルの幅 (ビット単位)。厳密には、ファイル内の 1 つのデータ値の幅を意味します。ユーティリティがデータをどのように分割して出力ファイルに入れるかは、ROM 幅によって決まります。ターゲット・ワードがメモリ・ワードにマップされた後、メモリ・ワードは 1 つまたは複数の出力ファイルに分割されます。出力ファイル数は ROM 幅によって決まります。

ROM モデル: リンカが C コードをリンクするときに使用する自動初期化モデル。−c オプションを指定してリンカを起動するときに、リンカは ROM モデルを使用します。ROM モデルでは、リンカはデータ・テーブルの .cinit セクションをメモリにロードし、変数は実行時に初期化されます。

S

SPC (Section Program Counter): セクション内の現在位置を追跡するためのアセンブラの一要素。各セクションには専用の SPC があります。

static: 関数またはプログラムに限定されているスコープをもつ変数の一種。静的変数の値は、それが使用されている関数またはプログラムが終了しても破棄されません。再度その関数またはプログラムを開始すると、以前の値が使用されます。

T

.text: デフォルトの COFF セクションの 1 つ。.text セクションは、実行可能なコードを含んだ初期化されたセクションです。.text 疑似命令を使用すると、コードを .text セクションにアセンブル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。このオプションを使用すると、リンカは複数のセクションに同じ実行アドレスを割り当てます。

unsigned: 実際の符号にかかわらず、正数として取り扱われる値の一種

あ

アーカイバ: 複数のファイルをアーカイブ・ライブラリと呼ばれる単一のファイルにグループ化するためのソフトウェア・プログラム。アーカイバを使用すると、新しいメンバを追加するだけでなく、アーカイブ・ライブラリのメンバを削除、抽出、または置換することができます。

アーカイブ・ライブラリ: 単一のファイル内にグループ化されたファイルの集合

アセンブラ: アセンブリ言語命令、疑似命令、およびマクロ疑似命令を含んだソース・ファイルから、機械語のプログラムを作成するソフトウェア・プログラム。アセンブラは、シンボリックな命令コードを絶対的な命令コードに置換し、シンボリックなアドレスを絶対アドレスまたは再配置可能なアドレスに置換します。

アセンブル時定数: .set 疑似命令を指定して定数値を割り当てたシンボル

い

位置合わせ: リンカが、出力セクションを n− ビット境界上のアドレスに置くこと。ここで、n は 2 の累乗。位置合わせの指定には、SECTIONS リンカ疑似命令を使用します。

インクリメンタル・リンク: 複数のパスでファイルをリンクする手法。この手法を使用すると、アプリケーションを分割して各部分を別々にリンクし、その後すべての部分をまとめてリンクすることができるので、大きなアプリケーションを開発する場合に有効です。

え

エミュレータ: TMS320C54x の機能をエミュレートするハードウェア開発システム

エントリ・ポイント: ターゲット・メモリでの実行開始点

お

オーバーレイ・ページ: 同じアドレス範囲にマップされる物理メモリの部分。どのページをアクティブにするかは、ハードウェア・スイッチにより決定されます。

オブジェクト・ファイル: 機械語オブジェクト・コードを含む、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 フォーマットや Tektronix フォーマットなどのオブジェクト・ファイルに変換するプログラム

オブジェクト・ライブラリ: 複数のオブジェクト・ファイルで構成されたアーカイブ・ライブラリ

オプション: ソフトウェア・ツールの起動時に、追加の機能や特定の機能を実行させるために使用するコマンド・パラメータ

オプション・ヘッダ: COFF オブジェクト・ファイルの一部分。リンカが、ダウンロード時に再配置を行うために使用します。

オペランド: アセンブリ言語命令、アセンブラ疑似命令、またはマクロ疑似命令の引数またはパラメータ

か

外部シンボル: 現行のプログラム・モジュールで使用されるけれども、その定義が異なるプログラム・モジュールで行われるシンボル

き

記憶クラス: シンボルへのアクセス方法を示すシンボル・テーブルのエントリ

疑似命令: 特別な目的をもつ複数のコマンド。ソフトウェア・ツールの動作や機能を制御します(デバイスの動作を制御する「アセンブリ言語命令」と対比)。

共通オブジェクト・ファイル・フォーマット: COFF を参照

行番号エントリ: COFF 出力モジュールにあるエントリの 1 つ。アセンブリ・コードの各行を、これを作成した元の C ソース・ファイルに逆にマップします。

共用体: 型とサイズが異なるオブジェクトを保持できる変数

く

クロスリファレンス・リスト: アセンブラにより作成される出力ファイル。定義されたシンボル、シンボルを定義した行、シンボルを参照する行、およびその最終値が表示されます。

グローバル: 次のいずれかの条件を満たすシンボルの一種です。
1) 現行のモジュールで定義されていて、他のモジュールでアクセスされている、または
2) 現行のモジュールでアクセスされているが、他のモジュールで定義されている。

こ

高水準言語デバッグ: シンボル情報、および高水準言語情報 (型や関数の定義など) を保持して、デバッグ・ツールがこれらの情報を使用できるようにするコンパイラの機能

構成メモリ: リンカが割り当てを行うために指定したメモリ

構造体: グループ化されて 1 つの名前を付けられた、単数または複数の変数の集まり

コマンド・ファイル: リンカまたは Hex 変換ユーティリティのオプション、ファイル名、疑似命令、またはコメントが記述されているファイル

コメント: ソース・ファイルを文書化したり、読みやすくするためのソース文(またはその一部)。コメントは、コンパイル、アセンブル、またはリンクされません。つまり、オブジェクト・ファイルには何の影響も及ぼしません。

さ

再配置: シンボルのアドレスが変更されるときに、リンカがそのシンボルに対するすべての参照を調整すること

サブセクション: セクション内のさらに小さなセクション。メモリ・マップをセクションよりも細かく制御する場合に使用します。「セクション」を参照してください。

し

式: 定数、シンボル、または算術演算子によって区切られた一連の定数とシンボル

実行アドレス: セクションが実行されるアドレス

実行可能モジュール: TMS320C54x システムで実行できる、リンクされたオブジェクト・ファイル

自動初期化: プログラムの実行の前に、C のグローバル変数 (.cinit セクションに保持されている) を初期化すること

シミュレータ: TMS320C54x の機能をシミュレートするソフトウェア開発システム

出力セクション: リンクされた実行可能モジュールの中の、最終的な割り当て済みセクション

出力モジュール: ターゲット・システム上にダウンロードして実行できる、リンクされた実行可能オブジェクト・ファイル

条件付き処理: ソース・コードの 1 ブロック、またはソース・コードの代替ブロックを指定された式の計算に基づいて処理する方法

初期化されたセクション: 実行可能コード、または初期化されたデータを含むCOFF セクション。初期化されたセクションは、.data 疑似命令、.text 疑似命令、または .sect 疑似命令から構成されます。

初期化されないセクション: メモリ・マップ内に空間は確保されるが、実際の内容はもたない COFF セクション。このセクションは、.bss 疑似命令と .usect 疑似命令から構成されます。

シンボリック・デバッグ: シンボル情報を保持して、シミュレータやエミュレータなどのデバッグ・ツールがこの情報を使用できるようにするソフトウェア・ツールの機能

シンボル: アドレスまたは値を表す英数字の文字列

シンボル・テーブル: ファイルで定義して使用されるシンボルについての情報を含んだ COFF オブジェクト・ファイルの部分

せ

整合定義式: すでに定義されているシンボルまたはアセンブル時定数のみを指定した式

静的な実行: 通常の見出しと進捗情報の出力を抑止すること

セクション: コードまたはデータの再配置可能なブロック。最終的には TMS320C54x メモリ・マップ内に連続した空間を占めます。

セクション・プログラム・カウンタ: 「SPC」を参照

セクション・ヘッダ: COFF オブジェクト・ファイルの一部分。そのファイルのセクションに関する情報が含まれます。各セクションには専用のヘッダがあります。ヘッダは、そのセクションの開始アドレスを示し、サイズなどの情報を含んでいます。

絶対アドレス: TMS320C54x™ のメモリ位置に永続的に割り当てられるアドレス

絶対リスタ: リンク済みファイルを入力として受け入れ、出力として.abs ファイルを作成するデバッグ・ツール。.abs ファイルをアセンブルすることによって、オブジェクト・コードの絶対アドレスを表示するリストを作成できます。絶対リスタを使用しないと、絶対リストの作成作業には多くの手動操作が必要になります。

そ

ソース・ファイル: C コードまたはアセンブリ言語コードを含んだファイル。コードをコンパイルまたはアセンブルすることによって、オブジェクト・ファイルを作成します。

た

代数表記: アセンブラによって機械コードに変換される命令

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・コードをロードできる、TMS320C54x システムの物理メモリ

て

定数: オペランドとして使用できる数値

な

名前付きセクション: .sect 疑似命令を使用して定義される、初期化されたセクション

生データ: 出力セクション内の、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: アセンブラによって機械コードに変換される命令名

入力セクション: オブジェクト・ファイルのセクションの 1 つ。このセクションは、実行可能モジュールにリンクされます。

は

バインディング: 出力セクションまたはシンボルに異なるアドレスを指定するプロセス

ふ

ファイル・ヘッダ: COFF オブジェクト・ファイルの一部分。オブジェクト・ファイルに関する一般情報 (セクション・ヘッダの数、オブジェクト・ファイルをダウンロードできるシステムの種類、シンボル・テーブルにあるシンボルの数、およびシンボル・テーブルの開始アドレス) が含まれます。

フィールド: TMS320C54x においては、ソフトウェアにより設定可能なデータ型で、 1～16 ビットの任意の値に長さをプログラミングできます。

符号拡張: 値を構成する未使用の MSB を、その値の符号ビットで埋めること

部分リンク: 再度リンク可能なようにファイルをリンクすること

ブロック: 中括弧 {} でまとめられた一連の宣言または文

ほ

ホール: 出力セクションを構成する入力セクション間にある領域。実際のコードやデータは含みません。

補足エントリ: シンボル・テーブルにあるシンボルの追加エントリ。そのシンボルに関する追加情報（ファイル名、セクション名、関数名など）が含まれます。

ま

マクロ: 命令として使用できるユーザ定義ルーチン

マクロ呼び出し: マクロを起動すること

マクロ定義: マクロを構成する名前とコードを定義する、ソース文のブロック

マクロ展開: マクロ呼び出しに置換され、続いてアセンブルされるソース文

マクロ・ライブラリ: マクロで構成されたアーカイブ・ライブラリ。ライブラリの各ファイルには、1 つのマクロが含まれていなければなりません。また、ファイル名はファイルに定義されているマクロ名と同じで、拡張子は .asm でなければなりません。

マジック・ナンバ: COFF ファイルのヘッダ・エントリの 1 つ。TMS320C54x で実行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です。

マップ・ファイル: リンカの作成する出力ファイル。メモリ構成、セクション構成、セクションの割り当て、およびシンボルとシンボルが定義されているアドレスを示します。

み

未構成メモリ: メモリ・マップの一部として定義されていないメモリ。コードやデータをロードすることはできません。

め

メモリ・マップ: ターゲット・システムのメモリ空間のマップ。複数の機能ブロックに区画分けされています。

メンバ: 構造体、共用体、アーカイブ、または列挙型の要素または変数

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ので、文字列テーブルが使用されます)。シンボルのエントリの名前の部分は、文字列テーブルの文字列の位置を指します。

モデル文: マクロ定義の中の命令またはアセンブラ疑似命令。マクロが起動されるたびにアセンブルされます。

ら

ラベル: ソース文の 1 カラム目で始まるシンボル。その文のアドレスに対応します。

り

リスト・ファイル: アセンブラの作成する出力ファイル。ソース文、その行番号、および SPC への効果が記述されています。

リンカ: オブジェクト・ファイルを結合して、オブジェクト・モジュールを生成するソフトウェア・ツール。生成されたモジュールは TMS320C54x システム・メモリに割り当てられ、デバイスにより実行できます。

ろ

ローダ: 実行可能モジュールを TMS320C54x システム・メモリにロードするデバイス

わ

ワード: ターゲット・メモリ内の 16 ビットのアドレッシングが可能な位置

割り当て: リンカが出力セクションの最終的なメモリ・アドレスを計算するプロセス

索引

記号

−?
  アセンブラ・オプション， 3-5
  リンカ・オプション， 7-6

オペランド接頭部， 3-13

−@，アセンブラ・オプション， 3-4

* オペランド接頭部， 3-13

数字

10 進整数定数， 3-16

16 進整数定数， 3-16

2 進整数定数， 3-15

8 進整数定数， 3-15

A

−a
  Hex 変換ユーティリティ・オプション， 10-39
  アーカイバ・コマンド， 6-4
  アセンブラ・オプション， 3-5
  リンカ・オプション， 7-8

A_DIR 環境変数， 3-9， 7-13， 7-15

abs500 コマンド⇒絶対リスタ, 起動を参照

.algebraic 疑似命令， 4-23， 4-26

.align 疑似命令， 4-15， 4-27
  C1x/C2x/C2xx/C5x との互換性， 4-8

−ar リンカ・オプション， 7-9

ar500 コマンド， 6-4

−arr Hex 変換ユーティリティ・オプション，
  10-29

ASCII，定義， F-1

ASCII−Hex オブジェクト・フォーマット，
  10-39

.asg 疑似命令， 4-21， 4-28
  マクロでの使用， 5-7
  リスト制御， 4-17， 4-41

asm500 コマンド， 3-4

attr MEMORY 指定， 7-29

B

−b リンカ・オプション， 7-10

.bes 疑似命令， 4-11， 4-82

−bkr Hex 変換ユーティリティ・オプション，
  10-29

.block シンボリック・デバッグ疑似命令， B-2

−boot Hex 変換ユーティリティ・オプション，
  10-29

boot.obj， 7-72， 7-75

−bootorg Hex 変換ユーティリティ・オプション， 10-29， 10-31

−bootpage Hex 変換ユーティリティ・オプション， 10-29

.break 疑似命令， 4-20， 4-66
  マクロでの使用， 5-15
  リスト制御， 4-17， 4-41

−bscr Hex 変換ユーティリティ・オプション，
  10-29

.bss 疑似命令， 4-9， 4-30
  C1x/C2x/C2xx/C5x との互換性， 4-8
  セクションの， 2-5
  リンカ定義， 7-65

.bss セクション， 4-9， 4-30， A-3
  初期化， 7-68
  定義， F-1
  ホール， 7-68

−byte，Hex 変換ユーティリティ・オプション， 10-36

.byte 疑似命令， 4-12， 4-33
  .option 疑似命令でリスト作成を制限する，
    4-17， 4-76

C

C
  システム・スタック， 7-18， 7-73
  メモリ・プール， 7-12， 7-73

−c
  アセンブラ・オプション， 3-5

−c (続き)
リンカ・オプション，7-10，7-65

C コード，リンク，7-72–7-75

C コンパイラ
COFF 技術詳細情報，A-1
関数定義，B-4
記憶クラス，A-19
行番号エントリ，B-6
行番号情報，A-12–A-13
共用体定義，B-8
構造体定義，B-8
シンボル・テーブル・エントリ，B-10
定義，F-1
特殊シンボル，A-16–A-17
ファイルの識別，B-3
ブロック定義，B-2
メンバ定義，B-7
リンク，7-10，7-72–7-75
列挙型定義，B-8

C_DIR 環境変数，7-13–7-15

_c_int00，7-11，7-75

.c_mode 疑似命令，4-23，4-35

C54X_A_DIR 環境変数，3-9，7-13

C54X_C_DIR 環境変数，7-13–7-15

.char 疑似命令，4-12，4-33

.cinit
セクション，7-73–7-74
テーブル，7-73

cinit シンボル，7-73–7-74

.clink 疑似命令，4-9，4-34

COFF
Hex フォーマットへの変換，10-1–10-44
⇒Hex 変換ユーティリティも参照
オブジェクト・ファイルの例，A-3
開発フローで，7-3，10-2
型エントリ，A-21
記憶クラス，A-19
技術詳細情報，A-1–A-28
行番号エントリ，B-6
再配置，2-16–2-18，A-10–A-11
実行時の再配置，2-19
初期化されたセクション，2-7
初期化されないセクション，2-5–2-6
シンボリック・デバッグ，A-12–A-13
シンボル，2-21–2-22，A-16–A-17
シンボル・テーブル
構造と内容，A-14–A-28
シンボルの値，A-20
セクション
アセンブラ，2-5–2-12

COFF (続き)
初期化されない，2-5–2-6
説明，2-3–2-4
特別な型，7-61
名前付き，2-8，7-66
リンカ，2-13–2-15
割り当て，2-3
定義，F-1
デフォルト割り当て，7-58
配列の制限，A-26
ファイル・タイプ，2-2
ファイルの構造，A-2–A-4
プログラムのロード，2-20
ヘッダ
オプション，A-6
セクション，A-7–A-9
ファイル，A-5
補足エントリ，A-23–A-28
文字列テーブル，A-18
リンカ，7-1

.const，7-34

.copy 疑似命令，3-8，4-19，4-36

COPY セクション，7-61

−cr リンカ・オプション，7-10，7-65

D

−d
アーカイバ・コマンド，6-4
アセンブラ・オプション，3-5，3-20

.data 疑似命令，4-9，4-39

.data セクション，2-5，4-9，4-39，A-3
シンボル，7-65
定義，F-1

.def 疑似命令，4-19，4-51
外部シンボルの識別，2-21

.double 疑似命令，4-13，4-40

.drlist 疑似命令，4-17，4-41
マクロでの使用，5-21

.drnolist 疑似命令，4-17，4-41
.option 疑似命令を使用する場合と同じ結果
マクロでの使用，5-21

DSECT セクション，7-61

E

−e
Hex 変換ユーティリティ・オプション，10-32
絶対リスタ・オプション，8-3
リンカ・オプション，7-11

.edata リンカ・シンボル，7-65

.else 疑似命令，4-20，4-56
  マクロでの使用，5-15

.elseif 疑似命令，4-20，4-56
  マクロでの使用，5-15

.emsg 疑似命令，4-24，4-42，5-19
  リスト制御，4-17，4-41

.end，リンカ・シンボル，7-65

.end 疑似命令，4-23，4-44

.endblock シンボリック・デバッグ疑似命令，
    B-2

.endfunc シンボリック・デバッグ疑似命令，B-4

.endif 疑似命令，4-20，4-56
  マクロでの使用，5-15

.endloop 疑似命令，4-20，4-66
  マクロでの使用，5-15

.endm 疑似命令，5-3

.endstruct 疑似命令，4-21，4-86

.endunion 疑似命令，4-22，4-92

.eos シンボリック・デバッグ疑似命令，B-8

.equ 疑似命令，4-21，4-81

.etag シンボリック・デバッグ疑似命令，B-8

.etext リンカ・シンボル，7-65

.eval 疑似命令，4-21，4-28
  マクロでの使用，5-8
  リスト制御，4-17，4-41

.even 疑似命令，4-15，4-27

F

−f リンカ・オプション，7-11

.far_mode 疑似命令，4-23，4-45

.fclist 疑似命令，4-17，4-46
  マクロでの使用，5-21
  リスト制御，4-17，4-41

.fcnolist 疑似命令，4-17，4-46
  マクロでの使用，5-21
  リスト制御，4-17，4-41

.field 疑似命令，4-12，4-47
  C1x/C2x/C2xx/C5x との互換性，4-8

.file シンボリック・デバッグ疑似命令，B-3

files ROMS 定義，10-18

fill
  MEMORY 指定，7-30

fill（続き）
  ROMS 指定，10-17
  埋め込み値
    設定，7-11
    デフォルト，7-11
    明示的な初期化，7-69

−fill Hex 変換ユーティリティ・オプション，
    10-27

.float 疑似命令，4-13，4-50
  C1x/C2x/C2xx/C5x との互換性，4-8

.func シンボリック・デバッグ疑似命令，B-4

G

−g
  アセンブラ・オプション，3-5
  リンカ・オプション，7-12

.global 疑似命令，4-19，4-51
  外部シンボルの識別，2-21

GROUP
  定義，F-1
  リンカ疑似命令，7-50

H

−h
  アセンブラ・オプション，3-5
  リンカ・オプション，7-12

.half 疑似命令，4-13，4-54
  .option 疑似命令を使用してリスト・ファイル
  の作成を制限する

−hc アセンブラ・オプション，3-5

−heap リンカ・オプション
  .sysmem セクション，7-73
  説明，7-12

−help
  アセンブラ・オプション，3-5
  リンカ・オプション，7-6

Hex 変換ユーティリティ
  ROMS 疑似命令，10-16–10-21
  ROM デバイス・アドレスの制御，
      10-34–10-37
  ROM 幅，10-11–10-13
  SECTIONS 疑似命令，10-22–10-23
  イメージ・モード，10-26–10-27
  エラー・メッセージ，10-44
  オブジェクト・フォーマット，10-38–10-43
  オプション，10-4–10-6
  オンチップ・ブート・ローダ，10-28–10-33
  開発フロー，10-2
  起動，10-3–10-6

Hex 変換ユーティリティ (続き)
コマンド・ファイル，10-7–10-8
出力ファイル名，10-24
説明，1-3
ターゲット幅，10-10
定義，F-2
データ幅，10-10
メモリ幅，10-10–10-11
メモリ・ワードの順序指定，10-14–10-15
例
2 つの 8 ビットEPROM を対象とした Hex コマンド・ファイルの作成方法，C-3–C-7
複数のセクション間のホールを回避する方法，C-8–C-9
ブート・テーブルの作成方法，C-10–C-16，C-17–C-24

hex500 コマンド，10-3

−hi アセンブラ・オプション，3-5

I

−i
Hex 変換ユーティリティ・オプション，10-40
アセンブラ・オプション，3-6，3-8
リンカ・オプション，7-14

I MEMORY 属性，7-29

.if 疑似命令，4-20，4-56
マクロでの使用，5-15

−image Hex 変換ユーティリティ・オプション，10-26

.include 疑似命令，3-8，4-19，4-36

.int 疑似命令，4-13，4-58

Intel オブジェクト・フォーマット，10-40

J

−j，リンカ・オプション，7-16

K

−k，リンカ・オプション，7-16

L

−l
アセンブラ・オプション，3-6，3-33
クロスリファレンス・リスタのオプション，9-3

−l (続き)
リンカ・オプション，7-13

.label 疑似命令，4-21，4-60

.ldouble 疑似命令，4-13，4-40

LDX 疑似命令，3-32

length
MEMORY 指定，7-30
ROMS 指定，10-17

.length 疑似命令，4-17，4-61
リスト制御，4-17

.line シンボリック・デバッグ疑似命令，B-6

.list 疑似命令，4-17，4-62
.option 疑似命令を使用した場合と同じ結果

lnk500 コマンド，7-4

load リンカ・キーワード，2-19，7-44–7-46

.long 疑似命令，4-13，4-64
C1x/C2x/C2xx/C5x との互換性，4-8
.option 疑似命令でリスト作成を制限する，4-17，4-76

.loop 疑似命令，4-20，4-66
マクロでの使用，5-15

M

−m，リンカ・オプション，7-16

−m1，Hex 変換ユーティリティ・オプション，10-41

−m2，Hex 変換ユーティリティ・オプション，10-41

−m3，Hex 変換ユーティリティ・オプション，10-41

.macro 疑似命令，4-67，5-3
まとめの表，5-25

_main，7-11

malloc()，7-12，7-73

.member シンボリック・デバッグ疑似命令，B-7

MEMORY リンカ疑似命令
PAGE オプション，7-27–7-29，7-60
オーバーレイ・ページ，7-53–7-57
構文，7-27–7-31
説明，2-13，7-27–7-31
デフォルト・モデル，7-27，7-58

.mexit 疑似命令，5-3

−mf アセンブラ・オプション，3-6

−mg アセンブラ・オプション，3-6

.mlib 疑似命令，4-68，5-14
  マクロでの使用，3-8
.mlist 疑似命令，4-17，4-70
  マクロでの使用，5-21
  リスト制御，4-17，4-41
.mmregs 疑似命令，4-23，4-71
.mmsg 疑似命令，4-24，4-42，5-19
  リスト制御，4-17，4-41
mnem2alg コマンド，11-4
.mnolist 疑似命令，4-17，4-70
  マクロでの使用，5-21
  リスト制御，4-17，4-41
Motorola−S オブジェクト・フォーマット，
    10-41

N

name MEMORY 指定，7-29
.newblock 疑似命令，4-23，4-74
.no_remark 疑似命令，4-23
.nolist 疑似命令，4-17，4-62
  .option 疑似命令を使用した場合と同じ結果
NOLOAD セクション，7-61
.noremark 疑似命令，4-75

O

−o リンカ・オプション，7-17
.option 疑似命令，4-17，4-76
−order Hex 変換ユーティリティ・オプション，
    10-15
origin
  MEMORY 指定，7-29
  ROMS 指定，10-17

P

paddr SECTIONS 指定，10-23
PAGE ROMS 指定，10-16
PAGE オプション MEMORY 疑似命令，7-27，
    7-56–7-58，7-60
.page 疑似命令，4-18，4-78
.pstring 疑似命令，4-13，4-85
−pw，アセンブラ・オプション，3-6

Q

−q
  アーカイバ・オプション，6-5
  アセンブラ・オプション，3-6
  クロスリファレンス・リスタのオプション，
    9-3
  絶対リスタのオプション，8-3
  リンカ・オプション，7-17

R

−r
  アーカイバ・コマンド，6-4
  アセンブラ・オプション，3-6
  リンカ・オプション，7-9，7-70–7-71
R MEMORY 属性，7-29
−r アセンブラ・オプション，4-75
RAM モデル
  自動初期化，7-73
  定義，F-2
.ref 疑似命令，4-19，4-51
  外部シンボルの識別，2-21
.remark 疑似命令，4-23，4-75
ROM
  デバイス・アドレス，10-34–10-37
  幅
    説明，10-11–10-13
    定義，F-2
  モデル
    自動初期化，7-74
    定義，F-2
romname ROMS 指定，10-16
ROMS Hex 変換ユーティリティ疑似命令，
    10-16–10-21
romwidth ROMS 指定，10-17
rts.lib，7-72，7-75
run リンカ・キーワード，2-19，7-44

S

−s
  アーカイバ・オプション，6-5
  アセンブラ・オプション，3-6
  リンカ・オプション，7-17，7-70–7-71
.sblock 疑似命令，4-23，4-79
.sect 疑似命令，2-5，4-9，4-80
.sect セクション，4-9，4-80

SECTIONS
Hex 変換ユーティリティ疑似命令, 10-22–10-23
リンカ疑似命令
GROUP, 7-50
.label 疑似命令, 7-45–7-47
MEMORY 疑似命令と組み合わせて使用, 7-27
UNION, 7-48–7-52
位置合わせ, 7-38
埋め込み値, 7-33
オーバーレイ・ページ, 7-53–7-57
構文, 7-32
実行割り当て, 7-33
指定, 2-19, 7-44–7-47
出力セクションの分割, 7-41
初期化されないセクション, 7-45
セクション指定, 7-33
セクションの型, 7-33
説明, 2-13, 7-32–7-43
デフォルトの割り当て, 7-58–7-60
デフォルト・モデル, 7-29
入力セクション, 7-33, 7-38–7-40
バインディング, 7-36
複数のメモリ領域を使用した割り当て, 7-40
ブロック化, 7-38
メモリ, 7-37–7-78
予約語, 7-24
ロード割り当て, 7-33
割り当て, 7-35–7-43

.set 疑似命令, 4-21, 4-81

.setsect 疑似命令, 8-8

.setsym 疑似命令, 8-8

.short 疑似命令, 4-13, 4-54

sname SECTIONS 指定, 10-23

.space 疑似命令, 4-11, 4-82

SPC
アセンブラ・シンボル, 3-12
アセンブラの影響, 2-10–2-12
値
ソース・リストに示された, 3-33
ラベルに関連付けられた, 3-12
位置合わせ
ホールの作成による, 7-66
ワード境界, 4-15–4-16, 4-27
最大値, 2-9
事前定義されたシンボル, 3-20
説明, 2-9
定義, F-2
ラベルを割り当てる, 3-12
リンカ・シンボル, 7-63, 7-66

−spc Hex 変換ユーティリティ・オプション, 10-29

−spce Hex 変換ユーティリティ・オプション, 10-29

SPC への $ シンボル, 3-20

.sslist 疑似命令, 4-18, 4-83
マクロでの使用, 5-21
リスト制御, 4-17, 4-41

.ssnolist 疑似命令, 4-18, 4-83
マクロでの使用, 5-21
リスト制御, 4-17, 4-41

.stack, 7-18, 7-19, 7-73

−stack リンカ・オプション, 7-18, 7-73

__STACK_SIZE, 7-18, 7-65

.stag, シンボリック・デバッグ疑似命令, B-8

.string 疑似命令, 4-13, 4-85
C1x/C2x/C2xx/C5x との互換性, 4-8
.option 疑似命令でリスト作成を制限する, 4-18, 4-76

.struct 疑似命令, 4-21, 4-86

−swwsr Hex 変換ユーティリティ・オプション, 10-29

.sym シンボリック・デバッグ疑似命令, B-10

__SYSMEM_SIZE, 7-12, 7-65

.sysmem セクション, 7-12

T

−t
Hex 変換ユーティリティ・オプション, 10-42
アーカイバ・コマンド, 6-4

.tab 疑似命令, 4-18, 4-89

.tag 疑似命令, 4-21, 4-22, 4-86, 4-92

−tcsr Hex 変換ユーティリティ・オプション, 10-29

Tektronix オブジェクト・フォーマット, 10-43

.text 疑似命令, 2-5, 4-9
リンカ定義, 7-65

.text セクション, 4-9, 4-90, A-3
定義, F-2

TI−Tagged オブジェクト・フォーマット, 10-42

.title 疑似命令, 4-18, 4-91

−trta Hex 変換ユーティリティ・オプション, 10-29

U

−u
アセンブラ・オプション，3-6
リンカ・オプション，7-18
.ubyte 疑似命令，4-12，4-33
.uchar 疑似命令，4-12，4-33
.uhalf 疑似命令，4-13，4-54
.uint 疑似命令，4-13，4-58
.ulong 疑似命令，4-13，4-64
UNION
定義，F-3
リンカ疑似命令，7-48–7-52
.union 疑似命令，4-22，4-92
unsigned，定義，F-3
.usect 疑似命令，2-5，4-9，4-95
C1x/C2x/C2xx/C5x との互換性，4-8
.usect セクション，4-9
.ushort 疑似命令，4-13，4-54
.utag シンボリック・デバッグ疑似命令，B-8
.uword 疑似命令，4-13，4-58

V

−v
アーカイバ・オプション，6-5
アセンブラ・オプション，3-7
リンカ・オプション，7-19
.var 疑似命令，4-98，5-13
リスト制御，4-17，4-41
.vectors，7-34
.version 疑似命令，4-23，4-99

W

W MEMORY 属性，7-29
−w リンカ・オプション，7-19
.width 疑似命令，4-18，4-61
リスト制御，4-17
.wmsg 疑似命令，4-24，4-42，5-19
リスト制御，4-17，4-41
.word 疑似命令，4-13
.option 疑似命令でリスト作成を制限する，
4-18，4-76

X

−x
Hex 変換ユーティリティ・オプション，10-43
アーカイバ・コマンド，6-4
アセンブラ・オプション，3-7，3-37
リンカ・オプション，7-20
X MEMORY 属性，7-29
.xfloat 疑似命令，4-13，4-50
.xlong 疑似命令，4-13，4-64
xref500 コマンド，9-3

あ

アーカイバ，1-3
オプション，6-5
開発フローで，6-3
概要，6-2
起動，6-4
コマンド，6-4
定義，F-3
例，6-6
アーカイブ・ライブラリ
オブジェクト，7-25–7-26
強制読み取り，7-20
後方参照，7-20
個々のメンバに割り当てる，7-43
代替ディレクトリ，7-13
定義，F-3
ファイルのタイプ，6-2
マクロ，4-68
アセンブラ
COFF セクションの取り扱い，2-5–2-12
LDX 疑似命令，3-32
エラー・メッセージ，D-1–D-18
オプション，3-4
追加使用情報，3-8，3-20，3-33，3-37
開発フローで，3-3
概要，3-2
拡張アドレッシング・サポート，3-32
起動，3-4
クロスリファレンス・リスト，3-7，3-37
再配置
実行時，2-19
説明，2-16–2-18
リンク時，7-8
式，3-25，3-26，3-27
出力リスト
疑似命令リスト，4-17–4-18，4-41–4-99
例，3-35
シンボル，3-19，3-21
セクション疑似命令，2-5–2-12

アセンブラ (続き)
説明，1-3
ソース・リスト，3-33–3-36
注釈の抑止，3-6，4-75
定義，F-3
定数，3-15–3-17
パイプライン・コンフリクトの処理， 3-6
ビルトイン関数，3-30，5-8
マクロ，5-1–5-26
文字列，3-18
アセンブラ疑似命令， 4-1–4-25
C1x/C2x/C2xx/C5x との互換性， 4-8
アセンブル時シンボルの定義， 4-21–4-22
.asg，4-21，4-28
.endstruct，4-21，4-86
.endunion，4-22，4-92
.equ，4-21，4-81
.eval，4-21，4-28
.label，4-21，4-60
.set，4-21，4-81
.struct，4-21，4-86
.tag，4-21，4-22，4-86，4-92
.union，4-22，4-92
出力リストのフォーマットの設定， 4-17–4-18
.drlist，4-17，4-41
.drnolist，4-17，4-41
.fclist，4-17，4-46
.fcnolist，4-17，4-46
.length，4-17，4-61
.list，4-17，4-62
.mlist，4-17，4-70
.mnolist，4-17，4-70
.nolist，4-17，4-62
.option，4-17，4-76
.page，4-18，4-78
.sslist，4-18，4-83
.ssnolist，4-18，4-83
.tab，4-18，4-89
.title，4-18，4-91
.width，4-18，4-61
条件付きアセンブリを可能にする， 4-20
.break，4-20，4-66
.else，4-20，4-56
.elseif，4-20，4-56
.endif，4-20，4-56
.endloop，4-20，4-66
.if，4-20，4-56
.loop，4-20，4-66
セクションの定義， 4-9–4-10
.bss，2-5，4-9，4-30
.clink，4-9，4-34
.data，2-5，4-9，4-39
.sect，2-5，4-9，4-80
.text，2-5，4-9，4-90
.usect，2-5，4-9，4-95
アセンブラ疑似命令 (続き)
セクション・プログラム・カウンタ (SPC) の位置合わせ，4-15–4-16
.align，4-15，4-27
.even，4-15，4-27
絶対リスタ
.setsect，8-8
.setsym，8-8
その他，4-23–4-24
.algebraic，4-23，4-26
.c_mode，4-23，4-35–4-99
.emsg，4-24，4-42
.end，4-23，4-44
.far_mode，4-23，4-45
.mmregs，4-23，4-71
.mmsg，4-24，4-42
.newblock，4-23，4-74
.noremark，4-23，4-75
.remark，4-23，4-75
.sblock，4-23，4-79
.version，4-23
.wmsg，4-24，4-42
他のファイルを参照する， 4-19
.copy，4-19，4-36
.def，4-19，4-51
.global，4-19，4-51
.include，4-19，4-36
.ref，4-19，4-51
定数の初期化，4-11–4-14
.bes，4-11，4-82
.byte，4-12，4-33
.char，4-12，4-33
.double，4-13，4-40
.field，4-12，4-47
.float，4-13，4-50
.half，4-13，4-54
.int，4-13，4-58
.ldouble，4-13，4-40
.long，4-13，4-64
.pstring，4-13，4-85
.short，4-13，4-54
.space，4-11，4-82
.string，4-13，4-85
.ubyte，4-12，4-33
.uchar，4-12，4-33
.uhalf，4-13，4-54
.uint，4-13，4-58
.ulong，4-13，4-64
.ushort，4-13，4-54
.uword，4-13，4-58
.word，4-13，4-58
.xfloat，4-13，4-50
.xlong，4-13，4-64
デフォルト疑似命令， 2-5
まとめの表，4-2–4-7
例，2-10–2-12

アセンブラ注釈の抑止，4-75
アセンブリ言語ソース・コードの ;，3-14
アセンブリ言語ソース・コードの *，3-14
アセンブル時定数，4-81
　定義，F-3

い

位置合わせ，4-15–4-16，4-27
　定義，F-3
　リンカ，7-38
インクリメンタル・リンク
　説明，7-70–7-71
　定義，F-3
インクルード・ファイル，3-5，3-8，4-36

う

埋め込み値 ⇒ホールを参照

え

エミュレータ，定義，F-3
エラー・メッセージ
　Hex 変換ユーティリティ，10-44
　アセンブラにより表示される，D-1–D-18
　生成，4-24，4-42
　マクロでの作成方法，5-19
　リンカにより表示される，E-1–E-16
演算子の優先順位，3-26
エントリ・ポイント
　定義，F-3
　割り当て値，7-11，7-75

お

オーバーレイ・ページ
　SECTIONS 疑似命令の使用方法，7-55–7-56
　説明，7-53–7-57
　定義，F-3
オブジェクト
　コード・ソース・リスト，3-34
　フォーマット
　　ASCII−Hex，10-39
　　Intel，10-40
　　Motorola−S，10-41
　　Tektronix，10-43
　　TI−Tagged，10-42
オブジェクト (続き)
　　アドレス・ビット，10-38
　　出力幅，10-38
　ライブラリ
　　アーカイバを使用して作成する，6-2
　　検索アルゴリズムの変更，7-13
　　説明，7-25–7-26
　　定義，F-4
　　ランタイム・サポート，7-72
オブジェクト・ファイル，定義，F-4
オブジェクト・フォーマット変換プログラム，定義，F-4
オプション
　Hex 変換ユーティリティ，10-4–10-6
　アーカイバ，6-5
　アセンブラ，3-4
　クロスリファレンス・リスタ，9-3
　絶対リスタ，8-3
　定義，F-4
　トランスレータ，11-4
　リンカ，7-6–7-20
オプション・ヘッダ
　定義，F-4
　フォーマット，A-6
オペランド
　間接アドレッシング，3-13
　接頭部，3-13
　ソース文のフォーマット，3-13
　定義，F-4
　フィールド，3-13
　ラベル，3-19
　ローカル・ラベル，3-22
オペランドの接頭部，3-13
オンチップ・ブート・ローダ
　EPROM からのブート方法，10-33
　ROM デバイス・アドレスの制御方法，10-35
　エントリ・ポイントの設定方法，10-32
　オプション
　　−e，10-32
　　要約，10-29
　シリアル・ポートからのブート方法，10-33
　説明，10-28，10-32
　パラレル・ポートからのブート方法，10-33
　ブート・テーブル，10-28–10-33
　ブート・ローダの使用方法，10-32
　ペリフェラルからのブート方法，10-31
　モード，10-32

か

開発
　ツール，1-2
　フロー，1-2，6-3，7-3

外部シンボル，2-21
再配置可能，3-28
定義，F-4

拡張アドレッシング・サポート， 3-32

型エントリ，A-21

環境変数
A_DIR，3-9，7-13
C_DIR，7-13，7-15
C54X_A_DIR，3-9，7-13
C54X_C_DIR，7-13

関係演算子，3-27

関数定義，A-17，A-26，A-27，B-4

き

記憶クラス
説明，A-19
定義，F-4

疑似命令
⇒アセンブラ疑似命令も参照
シンボリック・デバッグ， B-2–B-12
定義，F-4
リンカ
MEMORY，2-13，7-27–7-31
SECTIONS，2-13，7-32–7-43

共通オブジェクト・ファイル・フォーマット ⇒ COFFを参照

行番号，テーブルの構造， A-12–A-13

行番号エントリ
疑似命令，B-6
定義，F-4

共用体
.tag，4-22，4-92
シンボリック・デバッグ疑似命令， B-8
定義，F-4

キーワード
load，2-19，7-35，7-44
run，2-19，7-35，7-44
割り当てパラメータ， 7-35

く

クロスリファレンス・リスタ
オプション，9-3

クロスリファレンス・リスタ（続き）
開発フローで，9-2
起動，9-3
クロスリファレンス・リストの作成方法， 9-2
シンボルの属性，9-6
例，9-4

クロスリファレンス・リスト
.option 疑似命令で生成する，4-18，4-76
アセンブラ・オプション， 3-7
クロスリファレンス・リスタを使用した作成方法，9-1–9-6
説明，3-37
定義，F-4

グローバル
シンボル，7-12
定義，F-4

こ

高水準言語デバッグ，定義， F-5

構成メモリ
説明，7-59
定義，F-5

構造体
.tag，4-21，4-86
定義，A-25，B-8，F-5

構文
ソース文，3-11
代入文，7-62

コマンド・ファイル
Hex 変換ユーティリティ， 10-7–10-8
定義，F-5
リンカ
起動，7-4
説明，7-21–7-24
定数，7-24
予約語，7-24
例，7-76–7-78

コメント
アセンブリ言語ソース･コードの， 3-14
定義，F-5
フィールド，3-14
ページ幅の拡張，4-61

コメント (続き)
マクロで，5-19
リンカ・コマンド・ファイルで， 7-22

さ

再帰的なマクロ，5-22

再配置
機能，7-8–7-9
実行時，2-19
情報の構造化，A-10–A-11
セクション，2-16–2-18
定義，F-5

再配置可能
出力モジュール，7-9
シンボル，3-28

サブセクション
概要，2-9
初期化された，2-7
初期化されない，2-6
定義，F-5

算術演算子，3-26

し

式
アンダーフロー，3-26
演算子の優先順位， 3-25
オーバーフロー，3-26
再配置可能シンボル，3-28
算術演算子，3-26
条件，3-27
条件演算子，3-27
整合定義式，3-27
絶対および再配置可能， 3-28
説明，3-25
定義，F-5
不正，3-28
リンカ，7-63–7-64

式におけるアンダーフロー， 3-26

式におけるオーバーフロー， 3-26

式の計算，3-25

式の中の括弧，3-25

システム・スタック，7-18，7-73

事前定義された名前， −d アセンブラ・オプション，3-5

実行アドレス
セクションの，7-44
定義，F-5

実行可能出力，7-8，7-9

実行可能モジュール，定義， F-5

実行時初期化とランタイム・サポート， 7-72

自動初期化
型の指定，7-10
説明，7-73–7-74
定義，F-5

シミュレータ，定義， F-5

出力
Hex 変換ユーティリティ， 10-24
実行可能，7-8–7-9
セクション
規則，7-59
定義，F-5
メッセージの表示，7-19
割り当て，7-35–7-43
モジュール
定義，F-5
名前，7-17
リンカ，7-3，7-17，7-76

出力セクション，分割， 7-41

出力リスト，4-17–4-18

条件付き処理
アセンブリ疑似命令
最大ネスト・レベル，5-15
マクロで，5-15–5-16
式，3-27
定義，F-5

条件ブロック，5-15
アセンブリ疑似命令， 4-20
偽の条件ブロックのリスト， 4-46

初期化されたセクション， 2-7
.data，2-7，4-39
.sect，2-7，4-80
.text，2-7，4-90
説明，7-66
定義，F-6

初期化されないセクション， 2-5–2-6
.bss，2-6，4-30
.usect，2-6，4-95
実行アドレスの指定方法， 7-45
初期化，7-69
説明，7-66
定義，F-6

除去
行番号エントリ，7-17
シンボル情報，7-17

シンボリック・デバッグ
−b リンカ・オプション，7-10
−s アセンブラ・オプション，3-6
関数定義，B-4

除去 (続き)
行番号エントリ，B-6
共用体定義，B-8
構造体定義，B-8
シンボル，B-10
シンボル情報の除去，7-17
テーブルの構造と内容，A-14–A-28
ファイルの識別，B-3
マクロでのエラー・メッセージの作成方法，5-19
メンバ定義，B-7
リンカでのマージの抑止，7-10
列挙型定義，B-8

シンボル
アセンブラにより定義された，3-5
値の割り当て，4-21，4-22，4-81，4-86，4-92
リンク時，7-62–7-65
大文字と小文字，3-5
外部，2-21，4-51
クロスリファレンス・リスタ，9-6
クロスリファレンス・リスト，3-37
グローバル，7-12
参照する文の番号，3-37
式の再配置可能シンボル，3-28
事前定義された
$ 記号，3-20
メモリマップド・レジスタ，3-20
説明，2-21–2-22，3-19
属性，3-38
置換，3-21
定義，A-17，F-6
定義される
C サポートのためだけに，7-65
アセンブラによる，2-21–2-22
リンカで，7-65
定義済みアセンブラ，3-5
定義する文の番号，3-37
定数値に設定，3-19
名前，A-18
未解決の，7-18
文字列，3-18
予約語，7-24
ラベルとして使用される，3-19
割り当てられた値，3-37

シンボル定数，3-20

シンボル・テーブル
.sym 疑似命令からのエントリ，B-10
値，A-20
インデックス，A-10
エントリの作成，2-22
エントリの除去，7-17
構造と内容，A-14–A-28
使用される特殊シンボル，A-16–A-17

シンボル・テーブル (続き)
説明，2-22
定義，F-6
未解決のシンボルの挿入，7-18

す

数学関数，3-30

スタイルと記号の規則，v

せ

整合定義式
説明，3-27
定義，F-6

静的
シンボル，7-12
定義，F-2
変数，A-14

静的な実行，3-6
定義，F-6
リンカ，7-17

セクション
COFF，2-3–2-4
UNION 疑似命令でのオーバーレイ，7-48–7-49
再配置，2-16–2-18，2-19
指定，7-33
初期化された，2-7
初期化されない，2-5–2-6
実行アドレスの指定方法，7-45
初期化，7-69
定義，F-6
特別な型，7-61
名前付き，2-3，2-8
ユーザ・セクションの作成方法，2-8
ランタイム・アドレスの指定方法，7-44–7-47
リンカ入力セクションの指定方法，7-38
リンカの SECTIONS 疑似命令で，7-33
割り当て，7-58–7-60

セクションのオーバーレイ，7-48–7-49

セクションのロード・アドレス
説明，7-44
ラベルによる参照，7-45–7-47

セクション番号，A-21

セクション・プログラム・カウンタ
⇒SPCを参照
定義，F-2

セクション・ヘッダ
説明，A-7–A-9
定義，F-6

絶対アドレス，定義， F-6

絶対出力モジュール
  再配置可能， 7-9
  作成，7-8

絶対リスタ
  オプション，8-3
  開発フロー，8-2
  起動，8-3
  絶対リスト・ファイルの作成方法， 3-4，8-2
  説明，1-4
  定義，F-6
  例，8-5–8-10

絶対リスト
  –a アセンブラ・オプション， 3-5
  作成方法，8-2

そ

属性，3-38，7-29

ソース・ファイル
  代数表記命令の指定方法， 3-6
  定義，F-6
  リスト，3-33–3-36

ソース文
  構文，3-11
  ソース・リスト内の番号， 3-33
  フィールド，3-34
  フォーマット，3-12–3-14

た

代数
  ソース・ファイル，3-6
  定義，F-6
  ニーモニックからの変換， 11-1–11-10

代替ディレクトリ
  A_DIR による命名，3-9
  –i オプションによる命名，3-8
  疑似命令による命名，3-8–3-10
  リンカ，7-14

代入文
  式，7-63–7-64
  定義，F-7

タグ，定義，F-7

ターゲット幅， 10-10

ターゲット・メモリ，定義， F-7

ダミー・セクション， 7-61

ち

置換シンボル
  .var macro 疑似命令，5-13
  強制置換，5-11
  コンマとセミコロンの引き渡し， 5-6
  再帰的置換，5-10
  算術演算，4-21，5-8
  説明，3-21
  添字付き置換，5-12–5-13
  定義するための疑似命令， 5-7–5-8
  展開リスト，4-18，4-83
  ビルトイン関数，5-8
  マクロ，5-6–5-13
  マクロ当たりの最大数，5-6
  マクロのローカル変数として使用する， 5-13
  文字列の割り当て，3-21，4-21

注釈，抑止，4-75

て

定数
  10 進整数，3-16
  16 進整数，3-16
  2 進整数，3-15
  8 進整数，3-15
  アセンブル時，4-81
  コマンド・ファイルで， 7-24
  シンボル，3-19，3-20
  説明，3-15
  定義，F-7
  浮動小数点，4-50
  文字，3-16

ディレクトリ検索アルゴリズム
  アセンブラ，3-8
  リンカ，7-13

データ・メモリ， 7-27

デフォルト
  MEMORY 構成，7-58
  MEMORY モデル，7-27
  SECTIONS 構成，7-32，7-58
  セクション⇒COFF, セクションを参照
  ホールの埋め込み値， 7-11
  メモリ割り当て，2-14
  割り当て，7-58

と

特殊シンボル，A-16–A-17

特別なセクションの型， 7-61

な

名前付きセクション，2-8
  COFF フォーマット，A-3

名前付きセクション（続き）
.sect 疑似命令，2-8，4-80
.usect 疑似命令，2-8，4-95
定義，F-7

生データ，定義，F-7

に

ニーモニック
代数表記への変換，11-1–11-10
定義，F-7
フィールド，3-12

ニーモニックから代数表記への変換ユーティリティ ⇒変換ユーティリティを参照

入力
セクション
説明，7-38
定義，F-7
リンカ，7-3，7-25–7-26

ね

ネストしたマクロ，5-22

は

パイプライン・コンフリクト，3-6

配列の定義，COFF の制限，A-26

バインディング
セクション，7-36
定義，F-7
名前付きメモリ，7-36

パス⇒代替ディレクトリ、環境変数を参照

幅⇒メモリ幅を参照

ひ

ビッグ・エンディアン配列，10-14

ビルトイン関数，3-30，5-8

ふ

ファイル
インクルード，3-5
コピー，3-5
識別，B-3

ファイルのコピー
.copy 疑似命令，3-8，4-36
−hc アセンブラ・オプション，3-5
−i オプション，3-6，3-8

ファイル・ヘッダ
構造，A-5
定義，F-7

ファイル名
オブジェクト・コード，3-4
拡張子，デフォルトの変更，8-3
コピー/インクルード・ファイル，3-8
マクロ，マクロ・ライブラリでの，5-14
文字列として，3-18
リスト・ファイル，3-4

ファイル ROM 指定，10-18

フィールド，定義，F-7

符号拡張，定義，F-7

浮動小数点定数，4-50

ブート・テーブル⇒オンチップ・ブート・ローダ，ブート・テーブルを参照

ブート・ローダ⇒オンチップ・ブート・ローダを参照

部分リンク
説明，7-70–7-71
定義，F-7

プログラム・カウンタ⇒SPCを参照

プログラムのロード，2-20

プログラム・メモリ，7-27

ブロック
参照，B-2
説明，A-17
定義，F-7
テーブル補足エントリ，A-26，A-27

ブロック化，4-30，7-38

へ

ページ
PAGE 構文，7-56–7-58
オーバーレイ，7-53–7-57
タイトル，4-91
長さ，4-61
排出，4-78
幅，4-61

変換ユーティリティ
オプション，11-4
開発フロー，11-3
起動，11-4
出力ファイル，11-2

変換ユーティリティ (続き)
制限事項，11-2
説明，11-2
入力ファイル，11-2
モード，11-5–11-7
変数，ローカル，使用される置換シンボル， 5-13

ほ

補足エントリ
説明，A-23–A-28
定義，F-8
ホール
埋め込み，7-68–7-69
埋め込み値，リンカの SECTIONS 疑似命令，7-33
作成方法，7-66–7-68
出力セクションで，7-66–7-69
初期化されないセクションでの， 7-69
定義，F-8
デフォルトの埋め込み値， 7-11

ま

マクロ
.mlib アセンブラ疑似命令， 3-8
.mlist アセンブラ疑似命令， 4-70
疑似命令のまとめ， 5-25
コメント，5-19
再帰的，5-22–5-24
出力リストのフォーマット設定， 5-21
条件付きアセンブリ，5-15–5-16
説明，5-2
置換シンボル，5-6–5-13
定義，5-3，F-8
ネストした，5-22–5-24
パラメータ，5-6–5-13
マクロ展開のリスト作成を禁止する， 4-17，4-76
マクロの使用方法， 5-2
メッセージの作成方法， 5-19
ライブラリ，5-14，6-2
ラベル，5-17–5-18
マクロ・コール，定義， F-8
マクロ定義，定義， F-8
マクロ展開，定義， F-8
マクロ・ライブラリ，定義， F-8
マジック・ナンバ，定義， F-8
マップ・ファイル
作成，7-16
マップ・ファイル (続き)
定義，F-8
例，7-78

み

未構成メモリ
DSECT 型，7-61
説明，7-28
定義，F-8

め

メッセージ
エラー，D-1–D-18
リンカ，E-1–E-16
メモリ
名前付き，7-37
幅
ROM 幅，10-11–10-13，10-17
説明，10-10–10-11
ターゲット幅，10-10
メモリ・ワードの順序指定， 10-14–10-15
プール，C 言語，7-12，7-73
マップ
説明，2-15
定義，F-8
未構成，7-28
モデル，7-27
ワードの順序指定， 10-14–10-15
割り当て
説明，7-58–7-60
デフォルト，2-14
メモリ領域
MEMORY 疑似命令，7-29
定義，7-27
複数の割り当て，7-40
メモリ・ワードの順序指定， 10-14–10-15
メンバ，定義， F-8

も

文字
定数，3-16
文字列，3-18
文字列関数，5-9
文字列テーブル
説明，A-18
定義，F-8
モデル文，定義， F-9

ゆ

優先順位グループ，3-25

よ

予約語，リンカ，7-24

ら

ライブラリ検索アルゴリズム，7-13
ライブラリ作成ユーティリティ，説明，1-3
ラベル
  .byte 疑似命令で使用する，4-33
  アセンブリ言語ソース，3-12
  大文字と小文字の区別，3-5
  クロスリファレンス・リスト，3-37
  構文，3-12
  使用されるシンボル，3-19
  定義，F-9
  フィールド，3-12
  ローカル，3-22，4-74

り

リスタ
  クロスリファレンス，9-1–9-6
  絶対，8-1–8-10
リスト
  改ページ，4-78
  クロスリファレンス・リスト，4-18，4-76
  使用可能にする，4-62
  タイトル，4-91
  タブ・サイズ，4-89
  置換シンボル，4-83
  ファイル，4-17–4-18，4-41
    –l オプションを使用した作成方法，3-6
    定義，F-9
    フォーマット，3-33–3-36
  ページの長さ，4-61
  ページの幅，4-61
  マクロ・リスト，4-68，4-70
  抑止する，4-62
  リスト・オプション，4-76
リトル・エンディアン配列，10-14
リンカ
  C コード，7-10，7-72–7-75
  COFF，7-1
  COFF セクションの取り扱い，2-13–2-15
  GROUP 文，7-48，7-50
  MEMORY 疑似命令，2-13，7-27–7-31
  SECTIONS 疑似命令，2-13，7-32–7-43
  UNION 文，7-48–7-49
  アーカイブ・メンバ、割り当てる，7-43
  エラー・メッセージ，E-1–E-16
リンカ (続き)
  | 演算子，7-40
  >>演算子，7-41
  演算子，7-64
  オーバーレイ・ページ，7-53
  オブジェクト・ライブラリ，7-25–7-26
  オプション
    説明，7-8–7-20
    要約テーブル，7-6–7-7
  開発フローで，7-3
  概要，7-2
  起動，7-4–7-5
  キーワード，7-24，7-44，7-56
  構成メモリ，7-59
  コマンド・ファイル，7-4，7-21–7-24，7-76
  出力，7-3，7-17，7-76
  出力セクションを自動的に分割する，7-41
  シンボル，2-21–2-22，7-62，7-65
  シンボルの割り当て方法，7-62
  セクション
    出力，7-59
    特別な，7-61
    メモリ・マップ内の，2-15
  セクションのランタイム・アドレス，7-44
  説明，1-3
  代入式，7-62，7-63–7-64
  定義，F-9
  入力，7-3，7-21–7-24
  複数のメモリ領域を使用した割り当て，7-40
  部分リンク，7-70–7-71
  プログラムのロード，2-20
  未構成メモリ，7-61
  例，7-76–7-78

れ

レジスタ・シンボル，3-20
列挙型定義，B-8

ろ

ローカル・ラベル，3-22
ローカル・ラベルのリセット，4-74
ローダ，定義，F-9
論理演算子，3-26

わ

ワード，定義，F-9
ワードの位置合わせ，4-27
割り当て，4-30
  GROUP，7-50

割り当て (続き)
UNION，7-48
位置合わせ，4-27，7-38
セクション，7-35–7-43
説明，2-3
割り当て (続き)
定義，F-9
デフォルトのアルゴリズム，7-58–7-60
バインディング，7-36–7-78
ブロック化，7-38
メモリ・デフォルト，2-14，7-37

日本テキサス・インスツルメンツ株式会社

本　　社　〒160-8366　東京都新宿区西新宿6丁目24番1号　西新宿三井ビルディング3階　03(4331)2000(番号案内)

西日本ビジネスセンター　〒530-6026　大阪市北区天満橋1丁目8番30号　OAPオフィスタワー26階　06(6356)4500(代　表)
横浜オフィス　〒222-0033　横浜市港北区新横浜2丁目3番12号　新横浜スクエアビル3階　045(476)7870(代　表)
工　　場　大分県・日出町／茨城県・美浦村／静岡県・小山町（センサーズ＆コントロールズ事業部）
研究開発センター　茨城県・つくば市（筑波テクノロジー・センター）／神奈川県・厚木市（厚木テクノロジー・センター）

■お問い合わせ先
プロダクト・インフォメーション・センター（PIC）　FAX 0120-81-0036
URL: http: //www.tij.co.jp/pic/

TMS320C54x

アセンブリ言語ツール

ユーザーズ・マニュアル

第 2 版　2001年 8月
第 1 版　1999年10月

発行所　**日本テキサス・インスツルメンツ株式会社**
〒160−8366
東京都新宿区西新宿6−24−1(西新宿三井ビルディング)

Made in Japan
’01.08

日本テキサス・インスツルメンツ株式会社

SPRU366A