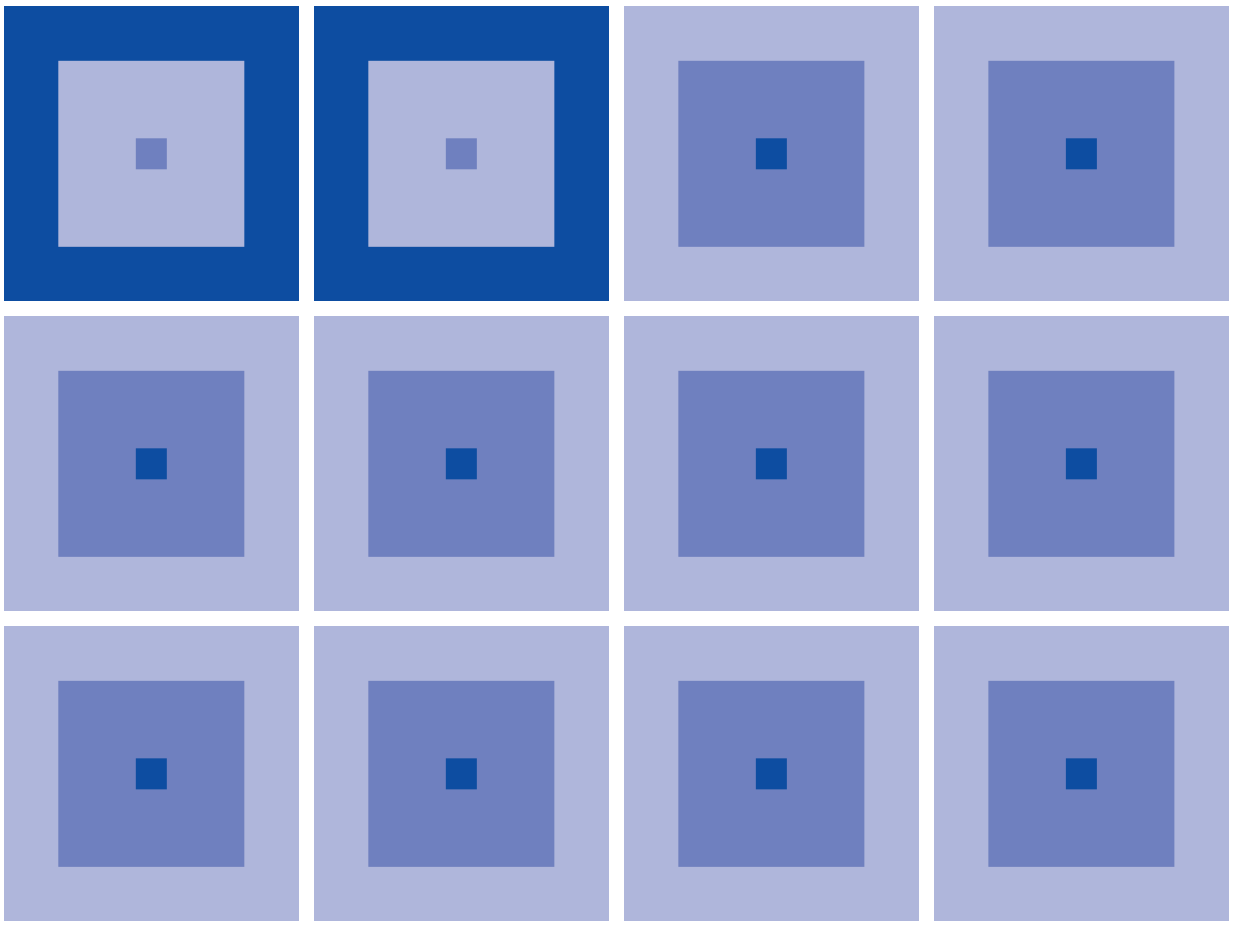
EPSON
EXCEED YOUR VISION

CMOS 8-BIT SINGLE CHIP MICROCOMPUTER

S5U1C88000C Manual I

(S1C88 Family統合ツールパッケージ)

Cコンパイラ/アセンブラ/リンカ



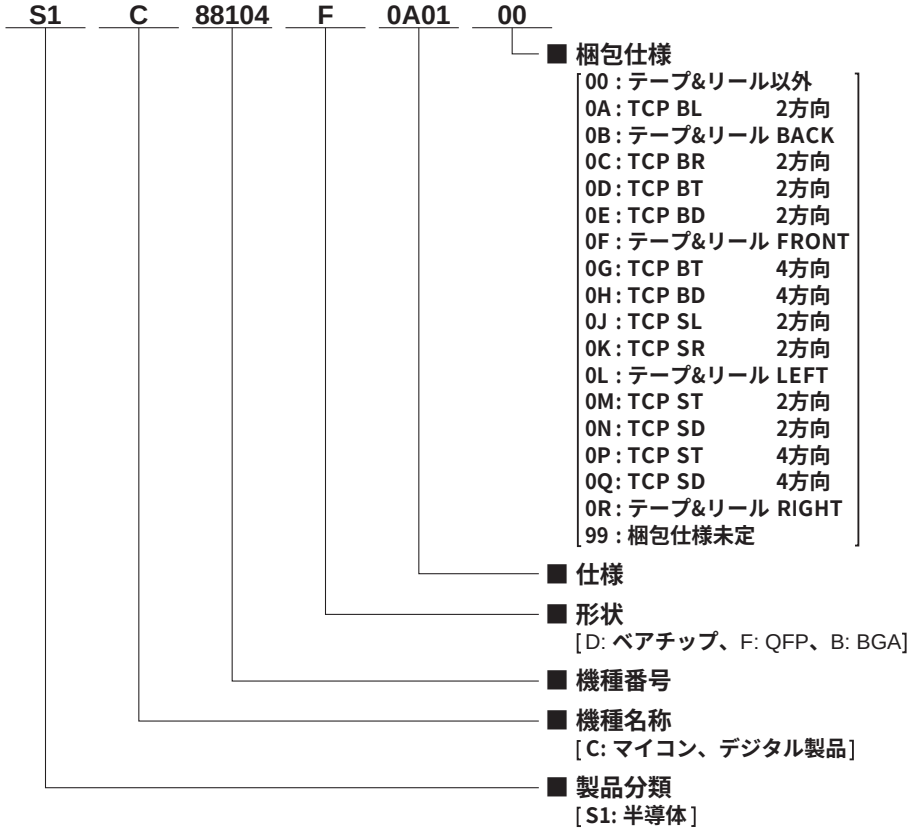
セイコーエプソン株式会社

本資料のご使用につきましては、次の点にご留意願います。

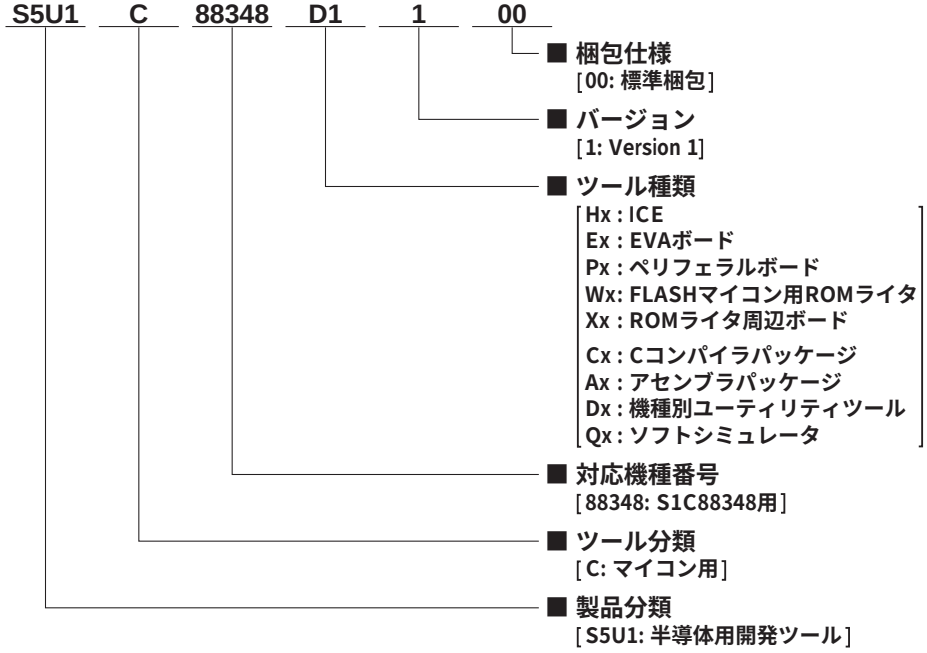
1. 本資料の内容については、予告なく変更することがあります。
2. 本資料の一部、または全部を弊社に無断で転載、または、複製など他の目的に使用することは堅くお断りします。
3. 本資料に掲載される応用回路、プログラム、使用方法等はあくまでも参考情報であり、これらに起因する第三者の権利(工業所有権を含む)侵害あるいは損害の発生に対し、弊社は如何なる保証を行うものではありません。また、本資料によって第三者または弊社の工業所有権の実施権の許諾を行うものではありません。
4. 特性表の数値の大小は、数直線上の大小関係で表しています。
5. 本資料に掲載されている製品のうち、「外国為替及び外国貿易法」に定める戦略物資に該当するものについては、輸出する場合、同法に基づく輸出許可が必要です。
6. 本資料に掲載されている製品は、生命維持装置その他、きわめて高い信頼性が要求される用途を前提としていません。よって、弊社は本(当該)製品をこれらの用途に用いた場合の如何なる責任についても負いかねます。

製品型番体系

●デバイス



●開発ツール



マニュアルの構成

S1C88 Family統合ツールパッケージには、 S1C88 Familyマイクロコンピュータのソフトウェア開発に必要なツールが含まれています。S5U1C88000C Manual（S1C88 Family統合ツールパッケージ）は、これらのツールの機能と使用方法を説明します。マニュアルは次のとおり2冊で構成されています。

I. Cコンパイラ/アセンブラ/リンカ（本書）
Cコンパイラを中心とした、ツールチェーン（次ページの図の[Main Tool Chain][注]）を解説しています。

II. ワークベンチ/Development Tools/旧アセンブラパッケージ
統合開発環境を提供するワークベンチ、アドバンスドロケータ、マスクデータ作成用ツール（次ページの図の[Development Tool Chain]）、デバッガ、構造化アセンブラ（次ページの図の[Sub Tool Chain]）を解説しています。

このマニュアルは、読者にCおよびアセンブリ言語の知識があることを前提として書かれています。

S1C88 Familyマイクロコンピュータの開発時は、必要に応じて以下のマニュアルも参照してください。

S1C88xxxテクニカルマニュアル
デバイスの仕様、Flash EEPROMプログラミング方法等を解説しています。

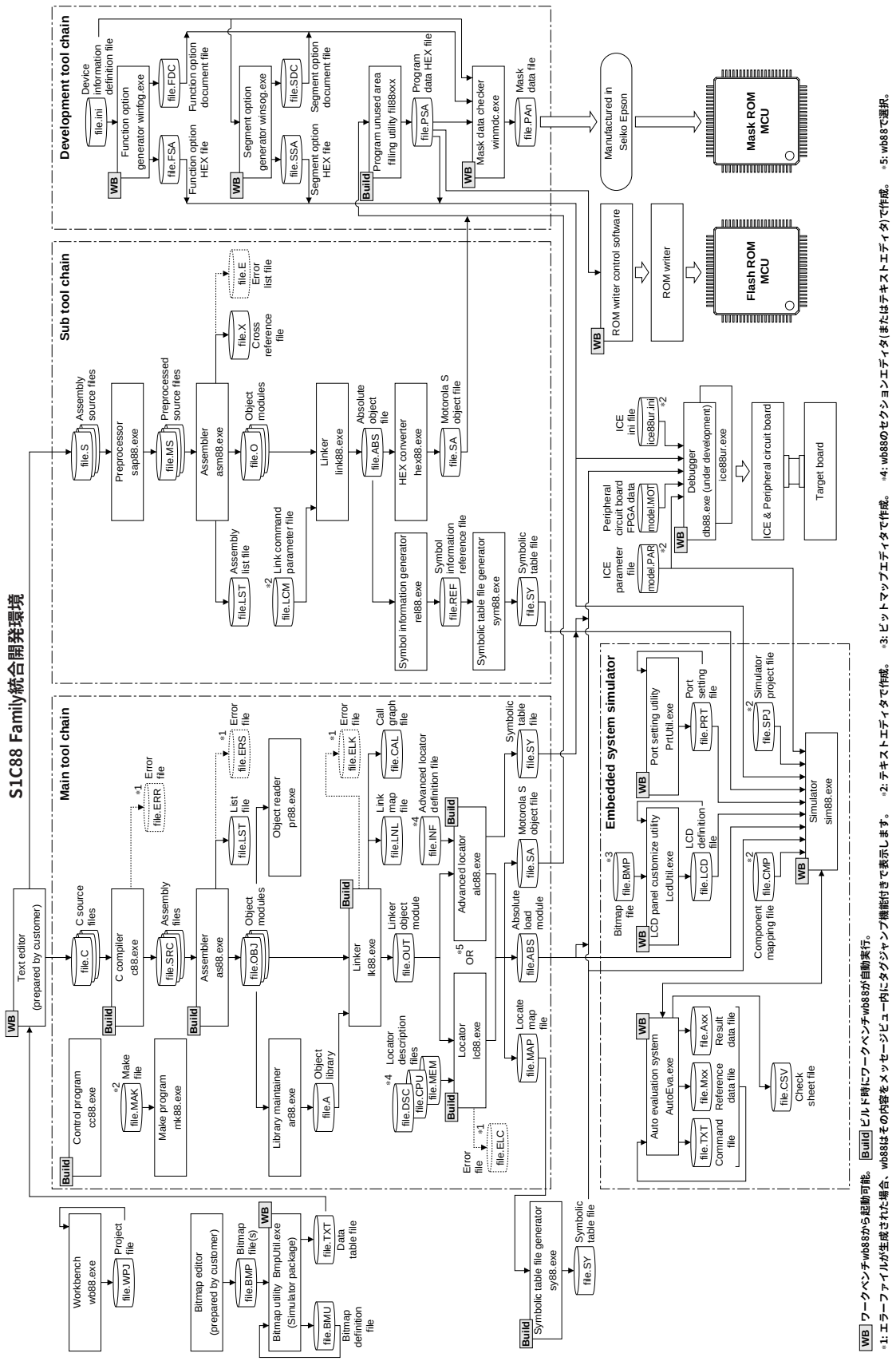
S5U1C88000Q Manual
シミュレータパッケージに含まれるツールの操作方法を解説しています。

S5U1C88000H5 Manual
ICE（S5U1C88000H5）の取り扱い方法を解説しています。

S5U1C88xxxP Manual
ICEに装着する周辺回路ボードの取り扱い方法を解説しています。

注: [Main Tool Chain]にはVer.3よりアドバンスドロケータalc88が追加され、ロケータlc88の代わりに使用できるようになっています。ただし、本書はTASKINGツールチェーンに対応しており、Cコンパイラ、アセンブラ、リンカ、ロケータおよびその関連情報のみを掲載しています。TASKINGツールチェーンに含まれていないアドバンスドロケータは、"II. ワークベンチ/Development Tools/旧アセンブラパッケージ"で説明されていますので、そちらを参照してください。

なお、ロケータ記述ファイルを含む既存の資産を使用する場合を除き、再配置用の記述言語を修得する必要がなく、また分岐最適化機能を持つアドバンスドロケータの使用を推奨します。したがって、S1C88 Familyのアプリケーションを新たに開発される場合(アドバンスドロケータを使用する場合)、特に本書に記載のロケータlc88（4章）およびロケータ記述言語DELFEE（5章）の説明をお読みいただく必要はありません。その他の章に出てくるロケータの機能および動作はアドバンスドロケータで代替されるものとして本書をお読みください。



WB ワークベンチwb88から起動可能。 Build ビルド時にワークベンチwb88が自動実行。

*1: エラーファイルが生成された場合、wb88はその内容をメッセージビュー内にタグジャンプ機能付きで表示します。 *2: テキストエディタで作成。 *3: ビットマップエディタで作成。 *4: wb88のセクションエディタ(またはテキストエディタ)で作成。 *5: wb88で選択。

－目　次－

1 Cコンパイラ

1.1 概要

1.1.1 S1C88 Cクロスコンパイラの概要

このマニュアルでは、S1C88 Cクロスコンパイラの機能について説明します。このマニュアルでは、"S1C88 Cコンパイラ"を単に**c88**(バイナリの名前)と表記します。

セイコーエプソンはS1C88プロセッサファミリ用のツールチェーンを提供しています。S1C88プロセッサファミリおよび派生製品を示すときは、単にS1C88と表記します。

S1C88 Cコンパイラでは、ANSI Cで書かれたソースプログラムを、S1C88アセンブラソースコードファイルに変換します。このコンパイラは、"ネイティブ"モードで動作するS1C88用のコードを生成します。また、言語拡張機能をサポートしているため、コードのパフォーマンスを向上させると同時に、典型的なS1C88アーキテクチャをCレベルで効率的に使用できるようになっています。このコンパイラはANSI C互換で、*プリプロセッサ*、S1C88 C*フロントエンド*、それに対応する*バックエンド*つまりコードジェネレータの、3つの主要な部分で構成されています。これらはすべて単一のプログラムに統合されており、中間ファイルも必要としないため、コンパイルプロセスのスピードを向上させることができます。また同時に、フロントエンドバックエンド結合最適化ストラテジおよびプリプロセッサプラグマのインプリメンテーションが簡単になります。このように、このコンパイラはワンパスコンパイラとして動作するため、ファイル入出力のオーバーヘッドも最小限にとどめることができます。

コンパイラは、一度に1つのC関数を処理しながら、ソースモジュール全体を読み込みます。関数が解析され、意味的な正しさがチェックされてから、中間コードツリーに変形され、これがメモリに格納されます。コードの最適化は、中間コードの構築時に実行されますが、関数が完全に処理されたときにも適用されます。後者は、通常、*グローバル*な最適化と呼ばれます。

c88は、S1C88アセンブラ言語仕様を使用してアセンブラコードを生成します。このコードは、S1C88クロスアセンブラを使用してアセンブルする必要があります。このマニュアルでは、"S1C88 クロスアセンブラ"を単に**as88**と表記します。

生成されたオブジェクトは、**lk88** S1C88リンカを使用して、他のオブジェクトやライブラリとリンクすることができます。このマニュアルでは、"**lk88** S1C88リンカ"を単に**lk88**と表記します。リンクされたオブジェクトは、**lc88** S1C88ロケータを使用することにより、完全なアプリケーションにロケートすることができます。このマニュアルでは、"**lc88** S1C88ロケータ"を単に**lc88**と表記します。

cc88はコントロールプログラムです。このコントロールプログラムには、S1C88ツールチェーンのさまざまなコンポーネントを起動する機能があります。**cc88**は、さまざまなファイル名拡張子を認識できるようになっています。Cソースファイル(`.c`)はコンパイラに渡されます。また、アセンブラソース(`.asm`)は前処理されてアセンブラに渡されます。リロケータブルオブジェクトファイル(`.obj`)およびライブラリ(`.a`)は、リンカの入力ファイルとして認識されます。拡張子`.out`および`.dsc`が付いているファイルは、ロケータの入力ファイルとして処理されます。コントロールプログラムでは、コンパイルプロセスのどの時点でも停止できるようになっており、中間ファイルを作成したり保存したりするオプションもあります。

1.1.2 一般的なインプリメンテーション

この節では、コンパイラおよびターゲットに依存しない最適化のさまざまなフェーズについて説明します。

1.1.2.1 コンパイラのフェーズ

Cプログラムのコンパイル時には、いくつかの異なるフェーズがあります。これらのフェーズは、*フロントエンド*および*バックエンド*の2つのグループに分けることができます。

フロントエンド：

プリプロセッサフェーズ：

ファイルのインクルードやマクロの置換は、Cプログラムの解析を始める前にプリプロセッサで実行されます。マクロプリプロセッサの構文は、Cの構文に依存せず、ANSI X3.159-1989標準で記述されているものになります。

スキャナフェーズ：

スキャナが、プリプロセッサの出力をトークンのストリームに変換します。

パーサフェーズ：

Cの文法では、トークンはパーサに送られます。パーサは、プログラムの構文と意味を解析し、プログラムの中間表現を生成します。

フロントエンド最適化フェーズ：

ターゲットプロセッサに依存しない最適化は、中間コードを変形することによって実行されます。次の節では、フロントエンドの最適化について説明します。

バックエンド：

バックエンド最適化フェーズ：

ターゲットプロセッサに応じた最適化を実行します。多くの場合、これは中間コードの再変換およびアクションを意味します。たとえば、変数にレジスタを割り当てるテクニック、式の評価、アドレッシングモードの有効利用などのようなものがあります。"1.2 言語仕様"では、この項目について詳細に説明しています。

コードジェネレータフェーズ：

このフェーズでは、中間コードを、S1C88アセンブラ命令を表す内部命令コードに変換します。

ピープホールオプティマイザ/パイプラインスケジューラフェーズ：

このフェーズでは、パターンマッチングテクニックを使用し、内部コードについてピープホール最適化を実行します。パイプラインスケジューラは、命令の順序を変更して結合し、命令の数を最小にします。最終的には、ピープホールオプティマイザが、内部命令コードを**as88**用のアセンブラコードに変換します。生成されたアセンブラコードには、マクロが含まれていません。アセンブラにも、オプティマイザが付属しています。

コンパイラの(フロントエンドとバックエンドの)すべてのフェーズは、1つのプログラムに結合されています。そのため、コンパイラは、コンパイルの異なるフェーズ間での通信に使用するための中間ファイルを使用しません。バックエンド部分は、Cの命令文単位で呼び出されず、C関数がフロントエンドによって(メモリ内で)完全に処理された後、開始されます。そのため、最適化のレベルを高くすることができます。このコンパイラでは、入力ファイルを1回のパスで処理できるため、コンパイルが比較的高速になります。

1.1.2.2 フロントエンドの最適化

コマンド行オプション**-O**は、Cソースに適用される最適化の程度をコントロールします。ソースファイル内では、プラグマ`#pragma optimize`がコンパイラの最適化レベルを設定します。プラグマを使用すると、プログラムの特定の部分について、特定の最適化のオン、オフを切り換えることができます。ただし一部の最適化は、個別にコントロールすることができません。たとえば、定数の畳み込みは常に実行されます。

コンパイラは、中間コードに対して次の最適化を実行します。これらの最適化は、ターゲットプロセッサおよびコード生成ストラテジに依存しません。

定数の畳み込み

定数のみを含む式が、その結果の値によって置換されます。

式の再構成

式を再構成して、定数を畳み込みます。たとえば`1+(x-3)`を`x+(1-3)`のように変形して、畳み込みます。

式の簡素化

0または1の乗算、および0の加算や減算が削除されます。このような無用の式は、マクロ、またはコンパイラ自体によって生成される可能性があります(たとえば配列の添字など)。

論理式の最適化

"&&"、"‖"、"!"を含む式は、一連の条件ジャンプに変換されます。

ループローテーション

`for`ループおよび`while`ループの場合、式がループの"上"で一度だけ評価され、後は"下"で評価されます。この最適化を実行した場合、コードは短くなりませんが、実行速度は短くなります。

switchの最適化

switch文については、さまざまな最適化が実行されます。冗長なcaseラベルが削除される他、場合によってはswitch文も削除されます。

コントロールフローの最適化

ジャンプ条件を反転させて、コードを移動させることにより、ジャンプ命令の大部分を縮小することができます。この最適化により、コードサイズが小さくなると同時に、実行時間も短くなります。

ジャンプのチェーン

無条件ジャンプの直後にあるラベルへの条件ジャンプまたは無条件ジャンプは、2番目のジャンプの宛先ラベルへのジャンプで置換されます。この最適化を実行した場合、コードは短くなりませんが、実行速度は短くなります。

無用なジャンプの削除

ジャンプの直後にあるラベルへの無条件ジャンプは削除されます。このようなラベルへの条件ジャンプは、ジャンプ条件の評価で置換されます。ジャンプには副作用がつきものであるため、評価は必要になります。

条件ジャンプの反転

無条件ジャンプを越える条件ジャンプは、ジャンプ条件を反転させた1つの条件ジャンプに変形されます。この最適化により、コードサイズが小さくなると同時に、実行時間も短くなります。

相互ジャンプと分岐末尾のマージ

2つの異なる実行パスに同じコードシーケンスがある場合、余分な命令を追加せずにマージできるときはマージされます。この変形により、実行時間はあまり短縮されませんがコードサイズは縮小されます。また一定の環境の場合、ジャンプを1回、回避できることもあります。

明示的代入文の定数式化

内容がわかっている変数を参照する場合、これらの変数はその内容で置換されます。

共通式の削除

このコンパイラには、同じ式が繰り返されて使用される場合、その(部分)式を検出する機能があります。このような"共通"式は、再計算を避けるため一時的に保存されます。この方法は、"共通式の削除"と呼ばれ、省略されてCSEと呼ばれます。

デッドコードの削除

無効コードは、プログラムに影響を与えずに、中間コードから削除することができます。ただし、コーディングエラーのために、無効コードが生成された可能性もあるため、警告メッセージが表示されることになっています。

ループの最適化

不変式は、ループから取り去られます。また、インデックス変数を含む式は、効率的に最適化されます。

ループの展開

短いループは、コピーで置き換えられます。

1.1.2.3 バックエンドの最適化

次の最適化は、ターゲットに依存するものであるため、バックエンドで実行されます。

割り当てグラフ

変数、パラメータ、中間結果、共通部分式が、割り当てユニットで表現されます。コンパイラは、関数ごとに、どのユニットがいつ必要かを示す割り当てユニットのグラフを構築します。レジスタのアロケータは、このグラフを使用することにより、使用可能なレジスタをもっとも効率的に占有できるようになります。コンパイラも、この割り当てグラフを使用して、アセンブラコードを生成します。

ピープホール最適化

生成されたアセンブラコードでは、命令シーケンスを、高速で短い同等のシーケンスに置き換えるか、不要な命令を削除することでパフォーマンスを向上することができます。

リーフ関数の処理

リーフ関数(他の関数を呼び出さない関数)は、スタックフレームの構築または疑似レジスタで特別に処理されます。

デッドストアの削除

ある式の結果が使用されない場合、その式は削除されます。

末端再帰の削除

その文の初めに分岐する再帰文は置き換えられます。

1.1.3 コンパイラの構造

S1C88アプリケーションを構築したい場合、次のプログラムを直接、またはコントロールプログラム経由で起動する必要があります。

- Cコンパイラ(**c88**)。接尾辞.cを持つファイルからアセンブラソースファイルを生成します。コンパイラ出力ファイルの接尾辞は、.srcです。ただし、**-o**オプションを付けることによって別のファイルに出力したりすることもできます。**-s**オプションを付けると、Cソース行と、生成されたアセンブラ文とを混合させることもできます。高レベル言語のデバッグ情報は、**-g**オプションを付けることにより生成することができます。ただし、生成されたアセンブラソースコードを調べるときは、**-g**オプションを付けない方が賢明です。というのも、このようなソースコードには、"読み取り不可能な"高レベル言語のデバッグ指示語が多く含まれているためです。Cコンパイラは、どのファイルについても1回だけパスを行います。この1回のパスで、構文をチェックし、コードを生成してコードの最適化を実行します。
- 対応するクロスアセンブラ(**as88**)。生成されたアセンブラソースファイルを処理して、接尾辞.objを持つリロケータブルオブジェクトファイルにします。
- **lk88**リンカ。生成されたリロケータブルオブジェクトファイルとCライブラリをリンクします。その結果、接尾辞.outを持つリロケータブルオブジェクトファイルが生成されます。この段階を終わると、接尾辞.lnlを持つリンカのマップファイルが生成されます。
- **lc88**ロケータ。生成されたリロケータブルオブジェクトファイルをロケートします。その結果、接尾辞.absを持つ絶対ローダブルファイルが生成されます。この段階を終わると、接尾辞.mapを持つフルアプリケーションマップファイルが生成されます。

接尾辞.absを持つ、ロケータの出力ファイルは、デバッガに直接ロードできます。

次の図は、S1C88ツールチェーンのさまざまなパーツ間の関係を示すものです。

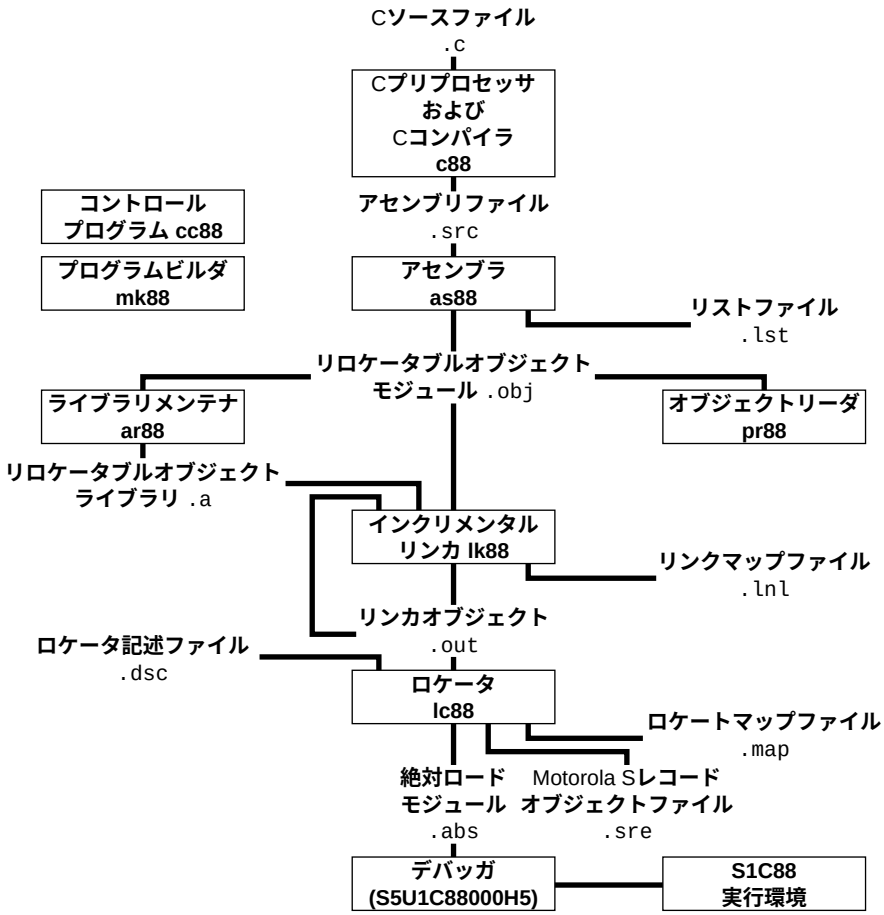


図1.1.3.1 S1C88開発フロー

プログラム**cc88**が、いわゆるコントロールプログラムで、S1C88ツールチェーンのさまざまなコンポーネントを起動します。Cソースプログラムはコンパイラによってコンパイルされ、アセンブラソースファイルはアセンブラに渡されます。Cプリプロセッサプログラムは、Cコンパイラの統合パーツとして使用することができます。コントロールプログラムは、ファイル拡張子.aと.objをリンカの入力ファイルとして認識します。またコントロールプログラムは、拡張子.outおよび.dscを持つファイルをロケータに渡します。他のすべてのファイルは、オブジェクトファイルと見なされ、リンカに渡されます。コントロールプログラムには、ロケート段階(**-cl**)の他、リンカ段階(**-c**)やアセンブラ段階(**-cs**)を抑制するオプションもあります。

ロケータ**lc88**は、オプションにより出力ファイルをMotorola Sレコード形式で生成します。デフォルトの出力フォーマットはIEEE-695です。

通常、コントロールプログラムは、次のフェーズが無事に完了したら、中間コンパイル結果を削除します。すべての中間ファイルをそのまま保持したい場合は、オプション**-tmp**を付けることで、これらのファイルが削除されないようにすることができます。

使用できるすべてのユーティリティおよびロケータの出力フォーマットについては、それぞれの章を参照してください。

1.1.4 環境変数

この節では、S1C88ツールチェーンが使用する環境変数について概説します。

環境変数	説　明
AS88INC	アセンブラのインクルードファイルが置かれる代替パスを指定します。
C88INC	Cコンパイラ**c88**のインクルードファイルが置かれる代替パスを指定します。
C88LIB	リンカ**lk88**が使用するライブラリファイルの検索パスを指定します。
CC88BIN	この変数が設定されているとき、コントロールプログラム**cc88**は、この変数で指定されているディレクトリを、起動するツール名の前に追加します。
CC88OPT	**cc88**を起動するたびに、余分なオプションと引数を指定します。コントロールプログラムは、コマンド行の引数の前に、この変数の引数を処理します。
PATH	実行可能ファイルの検索パスを指定します。
TMPDIR	プログラムが一時ファイルを作成できる代替ディレクトリを指定します。**c88**、**cc88**、**as88**、**lk88**、**lc88**、**ar88**が使用します。

1.1.4.1 コントロールプログラムの使用

次に、サンプルプログラムcalc.cを使用したプロセスについて説明します。この手順は、自身の実行可能ファイルをビルドしてデバッグする場合の参考になります。

1. examplesディレクトリのcサブディレクトリを現在の作業ディレクトリにします。
2. バイナリのディレクトリがPATH環境変数に入っていることを確認します。
3. コントロールプログラム**cc88**を呼び出し、モジュールのコンパイル、アセンブル、リンク、ロケートを実行します。

   ```
   cc88  -g -M -M1 calc.c -o calc.abs
   ```

   -gオプションは、シンボリックデバッグ情報を生成するときに指定します。デバッガを使用してデバッグするときは、常にこのオプションを付ける必要があります。

   一部の最適化を実行すると、高レベル言語デバッガでコードをデバッグする機能に影響が出ます。そのため、これらの最適化をオフにする目的で、**-g**オプションと一緒に**-O0**オプションを設定する必要があります。高いレベルの最適化を実行するとき、コンパイラに**-g**オプションを指定すると、警告メッセージW555が表示されます。

   -Mオプションは、マップファイルを生成するときに指定します。

   -M1オプションは、大容量メモリモデルを使用するときに指定します。

   -oオプションは、出力ファイルの名前を指定します。

手順3のコマンドにより、オブジェクトファイルcalc.obj、リンカマップファイルcalc.lnl、ロケータマップファイルcalc.map、絶対出力ファイルcalc.absが生成されます。calc.absファイルは、IEEE 695標準フォーマットであるため、デバッガから直接使用することができます。フォーマッタは別途必要ありません。

以上により、制御プログラムを呼び出すことで、デバッガを使用してデバッグするときに必要なすべてのファイルが作成されました。

コントロールプログラムがコンパイラ、アセンブラ、リンカ、ロケータを呼び出す方法を調べたい場合、**-v0**オプションまたは**-v**オプションを使用することができます。**-v0**オプションを使用すると、実際には実行されずに、実行の状況のみが表示されます。**-v**オプションを付けると、実行の表示と同時に実際にコマンドが実行されます。

```
cc88  -g -M -M1 calc.c -o calc.abs -v0
```

コントロールプログラムは、実際にはコマンドを実行せずに、次のようなコマンド実行の状況を表示します。

```
S1C88 control program va.b rc         SN00000000-003 (c) year TASKING, Inc.
+ c88 -e -g -M1 -o /tmp/cc24611b.src calc.c
+ as88 -e -g -o calc.obj /tmp/cc24611b.src
+ lk88 -e -M calc.obj -lcl -lrt -lfp -ocalc.out -Ocalc
+ lc88 -e -M -ocalc.abs calc.out
```

-eオプションは、エラーが発生した場合、出力ファイルを削除します。リンカの**-O**オプションは、マップファイルのベースネームを指定します。リンカのオプション**-lcl**、**-lrt**、**-lfp**は、適切なCライブラリ、実行時ライブラリ、浮動小数点ライブラリをリンクするよう指定します。

これを見るとわかるように、各ツールは、中間の結果を保持するために一時ファイルを使用します。中間ファイルを保持したい場合、**-tmp**オプションを使用することができます。次のコマンドは、この状況を示しています。

```
cc88  -g -M -M1 calc.c -o calc.abs -v0 -tmp
```

このコマンドにより、次の出力が生成されます。

```
S1C88 control program va.b rc        SN00000000-003 (c) year TASKING, Inc.
+ c88 -e -g -M1 -o calc.src calc.c
+ as88 -e -g -o calc.obj calc.src
+ lk88 -e -M calc.obj -lcl -lrt -lfp -ocalc.out -Ocalc
+ lc88 -e -M -ocalc.abs calc.out
```

これを見るとわかるように、**-tmp**オプションを使用する場合、カレントディレクトリにもアセンブラソースファイルとリンカ出力ファイルが作成されます。

もちろん、コントロールプログラムと同じような方法で、ツールを個別に実行しても同じ結果になります。

これを見るとわかるように、コントロールプログラムは、正しいオプションとコントロールを付けて、それぞれのツールを自動的に呼び出します。コントロールプログラムについては、"1.4 コンパイラの使用"で詳細に説明します。

1.1.4.2 makeファイルの使用

examplesディレクトリのサブディレクトリには、それぞれ**mk88**が処理できるmakeファイルが含まれています。また、それぞれのサブディレクトリには、readme.txtファイルがあり、サンプルプログラムを構築する方法について説明しています。

calcデモのサンプルを構築するときは、次の手順を実行します。この手順は、自身の実行可能ファイルを構築してデバッグする場合の参考になります。

1. examplesディレクトリのasmサブディレクトリを現在の作業ディレクトリにします。

   このディレクトリには、calcデモサンプルを構築するためのmakeファイルが含まれます。デフォルトの**mk88**規則を使用します。

2. バイナリのディレクトリがPATH環境変数に入っていることを確認します。

3. プログラムビルダ**mk88**を呼び出し、モジュールのコンパイル、アセンブル、リンク、ロケートを実行します。

   ```
   mk88
   ```

   このコマンドにより、makeファイルが使用されて、サンプルが構築されます。

mk88によって起動されたコマンドを、実際に実行せずに確認するときは、次のコマンドを実行します。

```
mk88 -n
```

このコマンドにより、次の出力が生成されます。

```
S1C88 program builder vx.y rz         SN00000000-003 (c) year TASKING, Inc.
cc88 -g -M -M1 calc.c -o calc.abs
```

makeファイルの**-g**オプションは、Cコンパイラにシンボリックデバッグ情報を生成するよう命令するときに使用します。この情報を活用することにより、Cで記述されたアプリケーションを、簡単にデバッグできるようになります。

makeファイルの**-M**オプションは、リンカリストファイル(.lnl)およびロケータマップファイル(.map)を作成するときに使用します。

-M1オプションは、大容量メモリモデルを使用するときに指定します。

-oオプションは、出力ファイルの名前を指定します。

生成されたすべてのファイルを削除するときは、次のコマンドを入力します。

```
mk88 clean
```

1.2 言語仕様

1.2.1 はじめに

Cクロスコンパイラ(**c88**)では、S1C88 Family用の高レベル言語プログラミングに対する新しいアプローチを提供しています。このコンパイラはANSI標準に準拠していますが、同時にS1C88で用意されているCの特殊機能を制御できるようにもなっています。

この章では、S1C88アーキテクチャに関連するC言語仕様について説明します。

c88で使用するC言語仕様の拡張機能には、次のようなものがあります。

_sfrbyte and _sfrword
特殊機能レジスタの宣言に使用するデータ型。_sfrbyteまたは_sfrwordには、メモリが割り当てられません。

_at
絶対アドレスで変数を指定することができます。

記憶タイプ
それぞれの宣言内では、メモリの種類(extern、staticなど)に関係なく、記憶タイプを指定することができます。このようにすることにより、メモリモデルに依存しない方法で、さまざまな範囲の変数にアドレッシングできます(_near、_far、_rom)。

メモリ固有ポインタ
c88では、特定のターゲットメモリをポイントするポインタを定義することができます。ポインタは、_near、_farのいずれかのメモリをポイントすることができます。それぞれのポインタは、その種類に応じて効率的なコードを生成します。

共通関数
関数が共通関数(_commonキーワード)として宣言された場合、関数の内容がメモリの下位の32K(共有コードバンク)に置かれるようになります。

アセンブラ関数
アセンブラ関数の呼び出しについては、"1.2.15 アセンブラ関数の呼び出し"を参照してください。

割り込み関数
C言語で割り込みベクタを使用することにより、割り込み関数を直接指定することができます(_interruptキーワード)。

組み込み関数
組み込み関数呼び出し時にアセンブラの記述コードを生成するとき、さまざまな宣言済み関数を使用することができます。呼び出される関数を実行する前には、通常パラメータの受け渡しやコンテキスト保存を実行するためオーバーヘッドが発生しますが、こうすることにより、このようなオーバーヘッドを回避することができます。

1.2.2 メモリへのアクセス

S1C88では、異なるバンク機構で、CODEとDATAにアクセスします。コンパイラには、これを処理する機能があります。

実際には、完成したアプリケーションの大部分のCコードは、(言語拡張機能を使用している点を除いて)標準Cになっています。アプリケーションのこのような部分は、まったく変更せずに、システムの要件(コードの密度、外部RAMの容量など)にもっとも適合するメモリモデルを使用してコンパイルすることができます。

ごく一部のアプリケーションでは、言語拡張機能を使用しています。これらの箇所には、多くの場合次のような特性があります。

- I/Oへアクセスするとき、特殊機能レジスタを使用。
- 高い実行速度が必要。
- 高いコード密度が必要。
- デフォルト以外のメモリに対してアクセス。
- 割り込みのサービスに使用。

1.2.2.1 記憶タイプ

プロセッサのアドレッシング空間の特定メモリ領域に静的オブジェクトを割り当てるとき、静的記憶指定子を使用することができます。静的記憶域をとるすべてのオブジェクトは、明示的な記憶指定子を使用して宣言できます。デフォルトでは、静的変数は、ラージモデルおよびコンパクトコードモデルの場合、_farメモリに割り当てられ、スモールモデルおよびコンパクトデータモデルの場合、_nearメモリに割り当てられます。

c88は、次の記憶タイプ指定子を認識します。

記憶タイプ	説　明
_near	データメモリの最下位64Kでアドレッシングします。
_far	データメモリの任意の場所でアドレッシングできますが、64Kページ以内でなければなりません。
_rom	ROM内にロケートされます。

例：

```
int _near         Var_in_near;           /* fast accessible integer in low
                                            (64K) address of _near Memory */
int _near * _far Ptr_in_far_to_near;     /* allocate pointer in _far Memory,
                                            used to point to integers in
                                            _near */
char _rom         string[] = "S1C88"     /* string in ROM Memory*/
```

_nearアドレッシング修飾子を使用すると、頻繁に使用される変数について、コンパイラが高速にアクセスできるコードが生成されます。_nearメモリに対するポインタも、_farメモリに対するポインタよりも高速に使用できるようになります。

関数はデフォルトでROMメモリ内に割り当てられます。その場合、記憶指定子は省略されます。また、関数の戻り値は、記憶域に割り当てることができません。

静的記憶指定子を使用する方法の他に、_atキーワードを使用しても、静的オブジェクトを固定メモリアドレスに割り当てることができます。

```
int _near       myvar _at(0x100);
```

これは、固定メモリ方式を使用したオブジェクトプログラムとインタフェースをとるときに使用すると便利です。

記憶指定子の使用例：

記憶指定子の例

```
int _near *P            // pointer to int in _near memory
                        // (pointer has 16-bit size)
int _far *g             // pointer to int in _far memory
                        // (pointer has 24-bit size)

g = P;                  /* the compiler issues a warning */
```

ライブラリ関数で次の宣言が行われ、

```
extern int _near foo;   //extern int in _near memory
```

同時にデータオブジェクトが次のように宣言された場合、

```
int _far foo;           //int in _far memory
```

リンカはこれにエラーのフラグを立てます。変数は、常に記憶指定子を付けないで使用しなければなりません。

```
char _far example;      /* define a char in _far memory*/
example = 2;            /* assign example       */
```

生成されるアセンブラコードは、次のようになります。

```
LD      ep,#@dpag(_example)
LD      a,#2
LD      [@doff(_example)],a
```

同じ記憶指定子を持つすべての割り当ては、"セクション"と呼ばれるユニットに集められます。_near属性を持つセクションは、最初の64Kに割り当てられます。セクションの位置を手動で指定することも可能です。

記憶域とセクションの関係

次の表は、生成されるアセンブラセクションタイプとそれぞれのC記憶タイプの属性を示しています。

記憶タイプ	S1C88セクションタイプ/属性
_near	DATA, SHORT
_far	DATA, FIT 10000H
_rom	CODE, ROMDATA, FIT 10000H
const _near	CODE, ROMDATA
const _far	CODE, ROMDATA, FIT 10000H

1.2.2.2 メモリモデル

c88は、スモール、コンパクトコード、コンパクトデータ、ラージの4つのメモリモデルをサポートしています。**-M**オプションを使用することで、これらのモデルのうちいずれかを選択することができます。S1C88のプログラムは常に、再入可能なモデルを使用してコンパイルされます。静的モデル関数は、ソース内で指定しなければなりません。

次の表は、各メモリモデルの概要を示しています。コマンド行でメモリモデルが指定されていない場合、スモールモデルが使用されます。これは、スモールモデルがもっとも効率的なコードであるためです。各コンパイラモデルは、プログラム/データサイズを次のように想定します。

メモリモデル	プログラムサイズ	データサイズ	説　明
スモール(s) (デフォルト)	≤ 64K	≤ 64K	"LD NB"がありません。拡張ページレジスタは0と見なされます。
コンパクトコード(c)	≤ 64K	> 64K	"LD NB"がありません。拡張ページレジスタは、必要に応じてロードされます。
コンパクトデータ(d)	> 64K	≤ 64K	"LD NB"が挿入されます。拡張ページレジスタは0と見なされます。
ラージ(l)	> 64K	> 64K	"LD NB"が挿入されます。拡張ページレジスタは、必要に応じてロードされます。

ただし、アセンブラでは同じモデルを使用することが前提になっています。そのため、異なるメモリモデルで構築された2つのオブジェクトは一緒にリンクできなくなります。

サポートされているすべてのモデルには、独立したバージョンのCライブラリとランタイムライブラリが提供されています。このため、特定のモデルを使用するときに、コンパイルや構築をやり直す必要がありません。

S1C88のさまざまなCPUモードで使用するためにさまざまなモデルが用意されています。プッシュされた戻りアドレスは、CPUモードごとに異なるため、コンパイラがこれを処理しなければなりません。コンパイラモデル、"スモール"と"コンパクトコード"では、CALL命令で2バイトの戻りアドレスをプッシュすることが前提になり、他のモデルでは、3バイトの戻りアドレスが前提になっています。また、スタートアップモジュールは、独自の状況に応じて適合させなければならないこともあります。

それぞれのCコンパイラモデルは、以下のCPUモデルに合わせて設計されています。

コンパイラモデル	CPUモード
スモール	シングルチップ (MCU), 64K (MPU)
コンパクトコード	512Kミニマム
コンパクトデータ	512Kマックス
ラージ	512Kマックス

コンパイラモデルによってPAGEレジスタの処理が異なり、戻りアドレスのサイズも異なるため、S1C88ツールは、1つのアプリケーション内にメモリモデルが混在する場合、それを受け付けません。リンカは、1つのアプリケーション内にモデルが混在している場合、そのことを通知します。

どのモデルでも、C関数のパラメータとオートマチックはスタック経由で渡されます。リンカは、このような目的で、アプリケーション全体で関数呼び出しグラフを使用しています。互いの関数を呼び出さない関数の場合、これらの関数が同時にアクティブになることはないため、それぞれのデータ領域は重ね書きすることができます。ただし、ポインタを使用して呼び出す関数の場合、その限りではありません。

_MODEL

c88では、定義済みのプリプロセッサシンボル_MODELが採用されています。_MODELの値は、選択されているメモリモデル(**-M**オプション)を表します。これは、異なるメモリモデルを持つ複数のアプリケーションで1つのソースモジュールを使用し、そこで条件Cコードを作成するとき、非常に便利になります。インクルードファイル`c88.h`について説明している"1.2.21 移植性の高いCコード"も参照してください。

_MODELの値は、次のようになります。

スモールモデル	's'
コンパクトコードモデル	'c'
コンパクトデータモデル	'd'
ラージモデル	'l'

例：

```
#if _MODEL == 's'     /* small model */
...
#endif
```

1.2.2.3 _at( )属性

S1C88のCでは、特定の変数を絶対アドレスに置くことができます。このとき、アセンブラコードを記述する代わりに、`_at()`属性を使用して、絶対アドレスに配置することができます。

例：

```
_far unsigned char Display _at( 0x2000 );
```

上記の例は、`Display`という名前の変数をアドレス0x2000に作成します。生成されるアセンブラコードでは、絶対セクションが現れます。この位置には、変数`Display`のための空間が予約されます。

変数を絶対アドレスに置く場合、次のようにさまざまな制限が適用されます。

- 絶対アドレスには、グローバル変数のみを置くことができます。関数のパラメータ、または関数内のauto変数は、絶対アドレスに置くことができません。
- 変数が"extern"として宣言されている場合、コンパイラはその変数を割り当てません。同じ変数が別のモジュール内で別のアドレスに割り当てられる場合、コンパイラ、アセンブラ、リンカはそれを認識できません。
- 変数が"static"として宣言されている場合、パブリックなシンボルは生成されません(標準的なCの動作)。
- 絶対変数は、ROM内で宣言された場合を除き、初期化できません。
- 関数は、絶対関数として宣言できません。
- 絶対変数は、お互いに重なり合うことができません。同じアドレスに2つの絶対変数を宣言すると、アセンブラまたはリンカによってエラーが生成されますが、コンパイラは、このことをチェックしません。
- 2つのモジュールで同じ絶対変数を宣言すると、リンク時に競合が発生します(ただしモジュールのいずれかで変数を"extern"として宣言している場合はこの限りではない)。

1.2.3 データ型

以下に示すANSI Cのデータ型がサポートされています。ANSI Cのデータ型の他にも、_sfrbyteと_sfrwordの型が追加されています。ポインタは2つの型が認識されます。

データ型	サイズ (バイト)	範　囲
signed char	1	-128から+127
unsigned char	1	0から255U
_sfrbyte	1	0から255U
signed short	2	-32768から+32767
unsigned short	2	0から65535U
signed int	2	-32768から+32767
unsigned int	2	0から65535U
_sfrword	2	0から65535U
signed long	4	-2147483648から+2147483647
unsigned long	4	0から4294967295UL
enum	2	0から65535U
_near pointer	2	0から65535U
_far pointer	3	0から16M

- _char、_sfrbyte、_sfword、short、int、longは、すべてint型で、すべての暗黙的(自動的)な変換をサポートしています。
- **c88**は、char型演算を使用してchar型式を評価するのが適切な場合、(8ビットの)char型演算を使用して命令を生成します。こうすると、int演算の場合と比べてコード密度が高くなります。これについては、"1.2.3.2 char型演算"で詳細に説明します。
- charとshortは、それぞれ8ビットと16ビットのintとして扱われます。
- 比較的高いメモリアドレスの最上位部分に変数を格納するときは、S1C88の規則が使用されます(リトルエンディアン)。

1.2.3.1 ANSI Cの型変換

ANSI C X3.159-1989標準によると、char型、short int型、 ビットフィールド(符号付き、符号なしの両方)、列挙型のオブジェクトは、整数が使用できる場合すべての式で使用できます。signed intで、元の型のすべての値を表現できる場合、値はsigned intに変換されます。それ以外の場合、値はunsigned intに変換されます。このプロセスは、整数の昇格と呼ばれます。

整数の昇格は、古いスタイルの宣言を使用すれば、int型の関数ポインタや関数パラメータでも実行されます。ただし暗黙的な型変換によって発生する問題を避けるため、関数プロトタイプを使用すると良いでしょう。

さまざまな演算子によって変換が発生しますが、結果の型も同様の方法で生成されます。結果的に、オペランドが共通の型に変換され、これが結果の型になります。このパターンは、通常の算術変換と呼ばれます。

整数の昇格は、両方のオペランドで実行されます。このとき、どちらかのオペランドがunsigned longになっている場合、もう一方もunsigned longに変換されます。

また、一方のオペランドがlongでもう一方がunsigned intになっている場合、結果は、longによってunsigned intのすべての値を表現できるかどうかで変わってきます。表現できる場合、unsigned intオペランドがlongに変換されます。表現できない場合は、両方がunsigned longに変換されます。

それ以外で、一方のオペランドがlongの場合、もう一方もlongに変換されます。

またそれ以外で、一方のオペランドがunsigned intの場合、もう一方もunsigned intに変換されます。

上記以外の場合、両方のオペランドがint型に変換されます。

"1.2.3.2 char型演算"も参照してください。

たとえば、unsigned charがintに昇格するような状況など、場合によっては、予期しなかった結果が生じることもあります。必要な型が得られるようにするため、常にキャスト変換を使用することもできます。次の例では、この状況を示しています。

```
static unsigned char a=0xFF, b, c;
void f()
{
      b=~a;
```

```
        if ( b == ~a )
        {
               /* This code is never reached because,
                * 0x0000 is compared to 0xFF00.
                * The compiler converts character 'a' to
                * an int before applying the ~ operator
                */
                ...
        }

        c=a+1;
        while( c != a+1 )
        {
               /* This loop never stops because,
                * 0x0000 is compared to 0x0100.
                * The compiler evaluates 'a+1' as an
                * integer expression. As a side effect,
                * the comparison will also be an integer
                * operation
                */
                ...
        }
  }
```

この"望ましくない"動作が発生しないようにするため、明示的なキャスト変換を使用します。

```
  static unsigned char a=0xFF, b, c;
  void f()
  {
        b=~a;
        if ( b == (unsigned char)~a )
        {
               /* This code is always reached */
                ...
        }

        c=a+1;
        while( c != (unsigned char)(a+1) )
        {
               /* This code is never reached */
               ...
        }
  }
```

ただし、算術変換は乗算にも適用されます。

```
  static int     h, i, j;
  static long    k, l, m;
  /* In C the following rules apply:
   *              int     * int   result: int
   *              long    * long  result: long
   *
   *      and Not int     * int   result: long
   */
  void f()
  {
          h = i * j;              /* int  * int  = int   */
          k = l * m;              /* long * long = long  */
          l = i * j;              /* int * int = int
                                   * afterwards promoted (sign
                                   * or zero extended) to long
                                   */
          l = (long) i * j;       /* long * long = long */
          l = (long)(i * j);      /* int * int = int,
                                   * afterwards casted to long
                                   */
  }
```

1.2.3.2 char型演算

c88は、式の結果がまったく同じになる場合、8ビット(char型)の演算を使用してコードを生成します。これは、int演算で評価される場合とまったく同じ方法になります。これは、ANSIがchar型演算とchar型定数を定義していないため、必ずこのようにする必要があります。S1C88プラットフォームは、16ビット演算を8ビット演算と同じ程度の処理速度で実行しますが、整数の昇格によるオーバーヘッドは発生しません。
そのため、式ではchar型変数を使用するのが望まれます。というのは、その場合、割り当てのためにデータ空間が予約され、結果的にコード密度が高くなるためです。char型演算を使用するとき、常にchar型キャストを使用するような設定にすることもできます。
次の例は、int演算が使用される場合とchar型演算の場合を明示しています。

```
char    a, b, c, d;
int     i;
void
main()
{
    c = a + b;                  /* character arithmetic */
    i = a + b;                  /* integer arithmetic   */
    i = (char)(a + b);          /* character arithmetic */
    c = a / d;                  /* character arithmetic */
    i = (a + b) / d;            /* integer arithmetic   */
    i = ((char)(a + b)) / d;    /* character arithmetic */
    c = a >> d;                 /* character arithmetic */
    i = (a + b) >> d;           /* integer arithmetic   */

    if ( a > b )                /* character arithmetic */
       c = d;
    if ( (a + b) > c )          /* integer arithmetic   */
       c = d;
}
```

1.2.3.3 特殊機能レジスタ

_sfrbyteキーワードと_sfrwordキーワードを使用することで、Cの変数と同じように、すべての特殊機能レジスタに直接アクセスできるようになります(ビットとバイトの両方で)。これらの特殊機能レジスタは、すべての自動変換を含む、他のintデータ型と同じ方法で使用することができます。

_sfrbyteは、volatile unsighned char変数と同じ方法で処理されます。_sfrwordは、volatile unsighned int変数と同じ方法で処理されます。

_sfrbyteまたは_sfrwordのデータ型を使用することにより、Cソース内でsfrレジスタを宣言することもできます。表記は次のようになります。

```
_sfrbyte    name _at( address )   ;
_sfrword    name _at( address )   ;
```

ただし、*name*は、指定したいsfrレジスタの名前で置き換えます。また*address*は、sfrレジスタのビット、バイト、ワードアドレスになります。offsetは、_sfrbyteのビットオフセットになります。これらのレジスタがプロセッサのsfr領域に置かれた場合、コンパイラは記憶域を割り当てません。

"sfrbyte"、"sfrword"という単語は**c88**の予約語ではありません。そのため、これらの単語を識別子として使用することができます。特殊機能レジスタは、デバッガによってすでに認識されているため、**c88**は、特殊機能レジスタ用のシンボリックデバッグ情報を生成しません。

特殊機能レジスタは入出力を処理しているため、レジスタへのアクセスを最適化してしまうのはよいことではありません。そのため、**c88**が特殊機能レジスタを処理するとき、volatile修飾子が付けられて宣言されたかのように扱います。

例：

```
_sfrbyte        _SPP  _at(0xFF01);
int             i;
volatile int    v;
main()
{
   i;           /* optimized away                  */
   _SPP=1;      /* access _SPP (implicit volatile) */
   v;           /* volatile: access variable       */
}
```

1.2.4 関数パラメータ

c88は、関数パラメータのプロトタイプ宣言(ANSI)をサポートしています。そのため、**c88**では、char型のパラメータをint型に変換せずに、これらのパラメータを渡すことができます。その結果、コード密度が高くなり、実行速度が高速になると同時に、パラメータの受け渡しに必要なRAMデータ空間も少なくてすみます。

たとえば、次のCコードがあります。

```
void  func( char number, long value );
int   printf( char *format, ... );
void
main(void)
{
      int   i;
      char  c;
      func( c, i );
      printf( "c=%d, i=%d\n", c, i );
}
```

コードジェネレータは、func()のプロトタイプを使用して、次の作業を実行します。

- cをbyteとして渡します。
- iをlongとして渡す前に、longに昇格させます。

ただし、printf()は変数の引数リストで宣言されているため、コードジェネレータは、この関数の引数を認識しません。(カーニハン&リッチーの古いスタイルの関数のように)プロトタイプがない場合、char型変数をint型変数に割り当てるときに自動変換が実行されるのと同じ方法で、char型のパラメータをint型に昇格させます。そのため、printf()呼び出しがある場合、コードジェネレータは、次の作業を実行します。

- cをintとして渡す前に、intに昇格させます。
- iをintとして渡します。

1.2.5 パラメータの受け渡し

デフォルトの場合、パラメータはレジスタ経由で渡されます。十分なレジスタが使用できない場合、一部のパラメータのみがレジスタ経由で渡され、他のパラメータはスタック経由で渡されます。

変数引数リスト関数のすべてのパラメータは、常にスタック経由で渡されます。パラメータは逆順でプッシュされるため、stdarg.hで定義されているすべてのANSI Cマクロが適用できます。

変数引数関数の例として、次のようなものがあります。

```
_printf( char *format, ... )
```

この場合、(formatを含む)すべてのパラメータがスタック経由で渡されます。

1.2.6 auto変数

再入不能な関数の場合、再帰は不可能です。これらの関数では、auto変数がスタック上に割り当てられず、静的領域に割り当てられます。再入可能な関数の場合、オートマチック変数は従来と同じ方法で扱われます。つまり、関数でスタック上に出し入れされます。静的関数の場合、記憶タイプ指定子を使用して、指定されたメモリにオートマチックを強制的に割り当てることができます。オートマチックは、他の関数のオートマチックで重ね書きできるようになっています。

auto変数は、再入不能な関数で静的領域に割り当てられますが、staticキーワードで静的と宣言された(関数内の)ローカル変数と同じではありません。

次のような相違点があります。

- ("通常の"方法では)auto変数の場合、別のモジュールの別のauto変数によって重ね書きされている可能性があるため、前回関数が返されたときと同じ値になっているとは限りません。
- ("通常の"方法では)スタティック変数の値は常に、前回関数が返されたときと同じになります。スタティック変数は、重ね書きされません。

1.2.7 レジスタ変数

Cでは、register型修飾子により、変数が頻繁に使用されることが伝えられます。そのため、コードジェネレータは、この変数のためにレジスタを予約して、このauto変数のスタックロケーションの代わりにこのレジスタを使用しようとする必要があります。一方コンパイラは、可能なときはいつでも、auto変数オブジェクトとパラメータオブジェクトをレジスタ内に割り当てます。そのため、**c88**はregisterキーワードを無視します。

レジスタに入れられないすべてのオブジェクトでは、次の規則が当てはまります。

再入可能な関数の場合

これらの関数では、スタックにオートマチック変数が割り当てられ、スタックまたはインデックスによるアドレッシングモードによりアドレッシングされます。

c88のコードジェネレータでは、"呼び出し側による保存"の方式が使用されます。つまり、関数呼び出しで1つまたは複数のレジスタの内容が必要になる場合、その関数は、これらの"レジスタ"の内容を保存して、呼び出しのあとにその内容を復元しなければなりません。このアプローチの長所は、呼び出しのあとで実際に使用されるレジスタのみが保存されるという点です。

結論：コードの密度や速度を改善するうえで、registerキーワードは必ずしも必要ではありません。

注: 配列、構造体にはregister修飾子は使用できません。

1.2.8 初期化変数

初期化された非auto変数では、ROMとRAM（すべてのRAMメモリ空間で）で同じ量の空間が使用されます。これは、初期化データがROMに格納されており、スタートアップ時にRAMにコピーされるためです。これは、ユーザから完全に透過的になっています。唯一の例外は、_rom記憶タイプ指定子によって初期化された、ROMに常駐する変数です。

例（スモールメモリモデル）：

```
int        i = 100;             /* 2 bytes in ROM and
                                   2 bytes in RAM            */
_rom  int  j = 3;               /* 2 bytes in ROM            */
char      *p = "TEXT";          /* 2 bytes for p in RAM
                                   5 bytes for "TEXT" in ROM */
_rom  char a[] = "HELP";        /* 5 bytes in ROM            */
_near char c = 'a';             /* 1 byte in ROM
                                   1 byte in near RAM        */
```

1.2.9 volatile型修飾子

通常の方法で使用しているとき、オブジェクトの変更によって、望ましくない副作用が発生する場合、volatile型修飾子を使用することができます。通常の場合、コンパイラは、その最適化機能により値をレジスタに保持してメモリ書き込みを保存しようとするため、メモリのロケーションは更新されません。しかし変数にvolatile修飾子を付けて宣言すると、コンパイラは、このような最適化を行いません。このようなvolatile変数は、NOCLEAR属性が設定されているセクションにあります。

ANSIのレポートでは、volatileオブジェクトの更新は抽象機械（ターゲットプロセッサ）の規則に従うため、volatileオブジェクトへのアクセスは、定義された仕様の方法になると記述されています。

例：

```
const volatile _near int real_time_clock _at(0x1234);
/*    define the real time clock register;
      it is read-only (const);
      read operations must access the real memory
      location (volatile)
*/
```

1.2.10 文字列

ここでは、"文字列"という単語は、Cプログラムに現れる文字列の1つ1つの断片を意味します。そのため、文字列で初期化された配列変数は、初期化されたchar型変数(任意のメモリタイプに割り当てることができる)になり、"文字列"と見なされなくなります。詳細については、"1.2.8 初期化変数"を参照してください。

Cソースプログラムの文字列とリテラルは、配列を初期化するときに使用されませんが、静的記憶持続期間を持っています。ANSI X3.159-1989標準では、文字列リテラルをROMに入れることができると規定しています。**c88**は文字列を常にROMに割り当てます。初期化された配列はそのままRAMに置かれます。

```
char ramhelp [] = "help";
/* allocation of 5 bytes in RAM and 5 bytes in ROM */
```

ROM上にしかない文字列のアドレスで初期化された、ROM上の配列の例を示します。

```
char * _rom message [] = {"hello","alarm","exit"};
```

ANSI文字列の連結もサポートされています。隣接する文字列(一次式として表記されている場合のみ)を1つの文字列に連結することができます。ただし結果は、文字列の最大長を超すことはできません(ANSIの制限では509文字だが実際のコンパイラの制限は1500文字)。

ANSI標準では、同一の文字列リテラルを区別する必要がないと明記されています。つまり、同じメモリを共有することができます。マイクロコントローラアプリケーションではメモリが非常に少なくなることがあるため、**c88**コンパイラは、同一の文字列を同じモジュール内に重ね書きします。

ANSI標準の3.1.4節には、このような動作は、プログラムが文字列リテラルを変更しようとする場合、未定義であると記述されています。文字列リテラルを変更できるというのは、ANSIと共通の拡張機能である(A.6.5.5)ため、実行時に文字列を変更するCソースがすでに存在している可能性があります。これは、ポインタで行うか、次のような文で行っている可能性があります。

```
"st ing"[2] = 'r';
```

ただし**c88**の場合、このような文は受け付けることができません。

1.2.11 ポインタ

一部のオブジェクトには、"論理型"と"記憶型"の2つの型があります。たとえば、関数はROM(記憶型)に置かれますが、この関数の戻り値型は、論理型になります。Cの型で、記憶型と論理型が異なる例としてもっともわかりやすいものは、ポインタです。例を示します。

```
_far char *_near p;   /*pointer residing in _near,
pointing to _far */
```

ここでは、pが記憶型_near(直接アドレスできるRAMに割り当てられている)になっていますが、論理型は"ターゲットメモリ空間_farのchar型"です。"*"の左で使用されているメモリ型指定子は、ポインタのターゲットメモリを指定し、"*"の右で使用されているメモリ指定子は、ポインタの記憶域メモリを指定しています。

メモリ型指定子は、他の型指定子と同じように(符号なしのように)扱われます。これは、上記のポインタが、次のような方法でも宣言できるということを示します(まったく同じ)。

```
char _far *_near p;   /*pointer residing in _near,
pointing to _far */
```

ポインタのターゲットメモリと記憶域メモリが明示的に宣言されていない場合、**c88**は、選択されたメモリモデルのデフォルトを使用します。

モデル	ターゲットメモリのデフォルト
's'	_near
'c'	_far
'd'	_near
'l'	_far

ポインタ演算では、**c88**が、それぞれのポインタの型の他、ポインタのターゲットメモリもチェックします。これは同じになる必要があります。たとえば、_farを示すポインタを、_nearを示すポインタに割り当てるのは無効になります(使用されない)。もちろん、キャストを適切に行うことによりエラーを防ぐことはできますが、不正であることには変わりありません。

1.2.12 関数ポインタ

再入可能な関数では、スタックを使用してパラメータとオートマチック変数割り当てを渡します。再入可能なメモリモデルを使用しているとき、すべての関数は、実際には暗黙的に再入可能になります。

そのため、関数ポインタは、再入可能としてコンパイルされた関数をポイントすることのみが許されます。パラメータはスタック経由でこれらの関数に渡されます。関数ポインタは、アプリケーション内にある、再入可能な任意の関数をポイントすることができます。

1.2.13 インラインC関数

関数の呼び出しの代わりに関数の本体をインラインすることをコンパイラに伝える場合、_inlineキーワードが使用されます。インライン関数は、同じソースファイルで、その関数を呼び出す前に定義する必要があります。インライン関数を複数のソースファイルで呼び出す必要がある場合、それぞれのファイルでインライン関数の定義をインクルードする必要があります。通常、これはインライン関数をヘッダファイルで定義することにより実行します。

_inline関数として定義されている関数を使用しない場合、コードが生成されません。

例(t.c)：

```
int   w,x,y,z;
_inline int
add( int a, int b )
{
      return( a + b );
}
void
main( void )
{
      w = add( 1, 2 );
      z = add( x, y );
}
```

インライン関数に関する特定のデバッグ情報は生成されません。デバッガは、インライン関数の中身をステップ実行することができません。インライン関数は、HLLソース行と見なされます。

インライン関数では、プラグマasmとendasmを使用することができます。これらを使用すると、アセンブラの記述関数を定義することができます。"1.2.14 アセンブラの記述"も参照してください。

1.2.14 アセンブラの記述

c88では、次のプラグマを使用してアセンブラの記述をサポートします。

#pragma asm	このプラグマの後にアセンブラテキストを挿入します。
#pragma asm_noflush	#pragma asmと同様ですが、ピープホールオプティマイザがコードバッファをフラッシュしません。
#pragma endasm	C言語に戻します。

アセンブラの記述を含むCモジュールは、移植性がなく、他の環境でプロトタイプを作るのは非常に難しくなります。

コンパイラのピープホールオプティマイザは、出力ファイルに書き込みを行うまでは、アセンブラ命令の最適化シーケンスでコードバッファを維持します。コンパイラは、アセンブラの記述のテキストを解釈せず、アセンブラの記述の行を直接出力ファイルに渡します。ピープホールバッファの命令(アセンブラの記述行の前のCコードにある)が、アセンブラの記述のテキストの後に出力ファイルに書き出されないようにするため、コンパイラは、ピープホールオプティマイザで命令バッファをフラッシュします。バッファにあるすべての命令は、出力ファイルに書き出されます。この動作が適していない場合、プラグマ**asm_noflush**により、アセンブラの記述がコードバッファをフラッシュしないで開始します。

1.2.15 アセンブラ関数の呼び出し

S1C88 Cコンパイラが、パラメータを関数に渡すときのレジスタの使用方法は固定です("1.3.2 レジスタの使用法"参照)。そのため、この使用方法に則れば、Cプログラムからアセンブラの関数を呼び出すことができます。ただし、レジスタには限りがありますので全パラメータがレジスタに割り当てられるようにしてください。(レジスタに割り当てられないパラメータが存在してもエラーやワーニングは発生しませんので注意してください。)

また、アセンブラ関数をコールするC関数部分のコンパイル結果(アセンブラソース)にて引数がどのレジスタに割り当てられているか、必ず確認してください。

以下はプログラム例です。

1. プログラムソース

```
int sub_asm(int ia, char ca, int ib, char cb, int _near *pic);

int main(void)
{
        int ia, ib, ic, id;
        int _near *pic;
        char ca, cb;

        ia = 1;
        ib = 2;
        ca = '3';
        cb = '4';
        ic = 5;
        pic = &ic;

        id = sub_asm(ia, ca, ib, cb, pic);

        id +=1;
              :
}

#pragma asm
_sub_asm:
              :
ret
#pragma endasm
```

2. コンパイル結果アセンブラソース(オプションなしの場合)

```
        LD      iy,#05h
        LD      [sp],iy                 ; [sp] <- ic
        LD      iy,sp                   ; iy <- &ic
        LD      ba,#01h                 ; ba <- ia
        LD      l,#033h                 ; l <- ca
        LD      ix,#02h                 ; ix <- ib
        LD      yp,#034h                ; yp <- cb
        CARL    _sub_asm
        INC     ba                      ; id <- ba
              :
_sub_asm:
              :
ret
```

3. パラメータ割り当て法則("1.3.2 レジスタの使用法"参照)

引数を渡す場合のレジスタ順序は次のとおりです。

	優先順位 高い 低い
char	A L YP XP H B
int	BA HL IX IY
long	HLBA IYIX
near ポインタ	IY IX HL BA
far ポインタ	IYP IXP HLP

したがって、上記のサンプルの、

```
int sub_asm(int ia, char ca, int ib, char cb, int *pic);
```

の呼び出しで使われるレジスタは

```
BA = int  ia
L  = char ca
IX = int  ib
YP = char cb
IY = int  *pic
```

となります。

(char用で最も優先度が高いAレジスタは`int ia`で使われているため、`char ca`には次の優先度のLが使われます。以下同様)

また、関数`sub_asm`からのint型戻り値は、int用で最も優先度が高いBAレジスタに割り当てられます。

1.2.16 組み込み関数

(標準)Cに同等のものがない、S1C88特有の関数を使用したい場合、これらの作業を実行するアセンブラルーチンを記述する他ありません。しかし、**c88**では、これをCで扱う方法が提供されています。**c88**には、さまざまなビルトイン関数があり、組み込み関数として実装されています。

プログラマから見ると、組み込み関数は通常のC関数と同じに見えますが、コードジェネレータがこれを解釈するという点で異なります。このような特性のため、生成されるコードは効率的になります。組み込み関数呼び出し時にインラインアセンブラコードを生成するとき、一部の宣言済みの関数が使用できます。このような関数を使用すると、呼び出される関数を実行する前に、パラメータの受け渡しとコンテキストの保存で通常発生するオーバーヘッドを回避することができます。

すべての組み込み関数の名前には、最初にアンダースコアが付いています。これは、一般的に定義済みの仕様の場合、Cの名前の最初にアンダースコアを付けるということが、ANSI仕様で定義されているためです。

アセンブラの記述(プラグマasm/endasm)と比較した場合の組み込み関数の利点には、次のようなものがあります。

- **c88**によって生成されたアセンブラの記述コードを置換するとき、ホストコンパイラがシミュレーションルーチンまたはスタブ関数を使用できます。
- Cレベルの変数にアクセスすることができます。
- コンパイラが、もっとも効率的にC変数にアクセスできるコードを生成します。
- `_nop()`を除き、組み込みコードが最適化されます。

次の組み込み関数が実装されています。

関数	説　明
_bcd()	式評価で"D"フラグをセット
_halt()	HALT命令
_int()	ソフトウェア割り込み
_jrsf()	条件が真であれば相対位置にジャンプ
_nop()	最適化されないNOP命令
_pack()	整数を文字値にパック
_rlc()	左にローテート
_rrc()	右にローテート
_slp()	SLP命令
_swap()	上位ニブルと下位ニブルをスワップ
_ubcd()	式評価で"U"フラグと"D"フラグをセット
_unpack()	式評価で"U"フラグをセット
_upck()	文字を整数値にアンパック

組み込み関数のプロトタイプは、`c88.h`にあります。以下、実装されている組み込み関数の詳細を示しています。

_bcd

void _bcd();

引数式を評価するとき、"D"フラグがセットされます。つまり、加算/減算命令および否定命令が二進化十進数として実行されます。ただし、式評価を通じて"D"フラグがリセットされないとき、たとえば式が浮動小数点を使用する場合などに問題が起こる可能性もあります。

引数は任意の型の式にすることができます(char/int/long)。そのため、引数リストは、古いスタイル(K&Rスタイル)の関数定義として(引数の型を定義しないで)実装されます。

戻り値　なし

_halt

void _halt (vold);

HALT命令を生成します。

戻り値　なし

```
_halt();
```

...Code...

```
        HALT
```

_int

void _int (ICE *vector*);

実行ソフトウェア割り込み命令(INT)を挿入します。引数はICE型の値で、ジャンプ先の割り込みベクタアドレスを決定します。ICEは、オペランドが、任意の型の整数式でなく、整数定数式でなければならないことを示します。

戻り値　なし

_jrsf

char _jrsf(ICE *number*);

_lab命令、JRS F*number*を使用します。この命令は、if()条件テストで使用するのに適しています。*number*の値は、0から3の定数値でなければなりません。ICEは、オペランドが、任意の型の整数式でなく、整数定数式でなければならないことを示します。コードジェネレータは、F*number*とこの命令の変異体であるNF*number*のいずれかを選択します。

戻り値　結果

```
if ( _jrsf(2) )
{
        ...
}
```

...Code...

```
       JRS NF2, _l0001
_L0001:
```

_nop

void _nop(void);

NOP命令を生成します。

戻り値　なし

```
_nop();
```

...Code...

```
        NOP
```

_pack

char _pack(int *operand*);

PACK命令を使用して、整数*operand*を文字値にパックします。

戻り値　文字値

_rlc

char _rlc(char *operand*);

RLC命令を使用して、バイト*operand*を左にローテートします。命令は結果にのみ影響し、*operand*には影響を与えません。

戻り値　結果

```
char c;
int i;
/* rotate left */
c= _rlc( c ) ;
```

_rrc

char _rrc(char *operand*);

RRC命令を使用して、バイト*operand*を右にローテートします。命令は結果にのみ影響し、*operand*には影響を与えません。

戻り値　結果

```
char C;
int i;
/* rotate right */
c = _rrc( c );
```

_slp

void _slp(void);

SLP命令を生成します。

戻り値　なし

```
_slp();
...Code...
          SLP
```

_swap

char _swap (char *operand*);

SWAP命令を使用して、文字*operand*の上位ニブルと下位ニブルをスワップします。

戻り値　結果

_ubcd

void _ubcd();

引数式を評価するとき、"U"フラグと"D"フラグがセットされます。つまり、計算時にバイトの下位ニブルのみが使用され、同時に加算/減算命令および否定命令がBCD演算として実行されます。ただし、式評価を通じて"D"フラグがリセットされないとき、たとえば式が浮動小数点を使用する場合などに問題が起こる可能性もあります。

戻り値　なし

_unpack

void _unpack();

引数式を評価するとき、"U"フラグがセットされます。つまり、計算時にバイトの下位ニブルのみが使用されます。引数は任意の型の式にすることができます(char/int/long)。そのため、引数リストは、古いスタイルの関数定義として実装されます。

戻り値 なし

_upck

int _upck(char *operand*);

UPACK命令を使用して、文字*operand*を整数値にアンパックします。

戻り値 整数値

1.2.17 割り込み

S1C88 C言語では、新しい予約語_interruptが導入されています。この予約語は、特殊な型の修飾子として使用され、関数宣言でのみ使用することができます。割り込みサービスルーチンとして使用するために、関数を宣言することができます。割り込み関数は、何も返すことができないため、**void**の引数型リストをとる必要があります。例を示します。

```
void _interrupt(vector)
_isr(void)
{
      ...
};
```

コンパイラは、割り込みのために割り込みサービスフレームを生成します。_interrupt関数修飾子は、2バイトの割り込みベクタ領域の割り込みベクタアドレスを定義する引数*vector*を1つだけとります。

一部の割り込みは予約されており、コンパイラ(ランタイムライブラリ)によって次のような処理に使用されます。

- ハードウェアリセット

_interruptの例：

ソフトウェア割り込みで割り込み関数を使用したい状況で、ベクタアドレスが0x30であると仮定します。

```
int c;
void
_interrupt( 0x30 )
transmit(void)
{
      c = 1;
}
```

これによって、次のようなアセンブラが生成されます。

```
          DEFSECT ".abs_48", CODE AT 48
          SECT    ".abs_48"
          DW      _transmit
          DEFSECT ".short_code", CODE, SHORT
          SECT    ".short_code"
_transmit:
          LD A,#1
          LD [_c],A
          RETE
```

1.2.18 構造体タグ

タグ宣言は、構造体や共用体のレイアウトを指定するために使用されます。メモリタイプが指定された場合は、それも宣言の一部と見なされます。"〜に対するポインタ"の型を持つメンバで必要であっても、タグ名自体にもその番号にも記憶域をバインドすることはできません。タグ宣言した後、その型のオブジェクトを宣言するときにタグを使用することができ、それを別のメモリに割り当てることができます(その宣言が同じ領域内にある場合)。次の例は、この制約を示しています。

```
struct S {
    _near int i;        /* referring to storage: not correct */
    _far char *p;       /* used to specify target memory: correct */
    };
```

上記の例の場合、**c88**はエラーのある_near記憶指定子を無視します(警告メッセージも表示しない)。

1.2.19 typedef

typedef宣言は、宣言されたオブジェクトと同じ領域の規則に従います。typedefの名前は、ブロック内で(再)宣言できますが、パラメータレベルでは宣言できません。ただし、typedef宣言では、メモリ指定子を使用できます。typedef宣言には、最低でも1つのタイプ指定子が必要です。

例：

```
typedef _near int NEARINT;      /* storage type _near: OK */
typedef int _far *PTR;          /* logical type _far
                                   storage type 'default' */
```

1.2.20 言語拡張機能

S1C88 Cコンパイラには以下の言語拡張機能が設定されています。これらは、ANSI-Cに準拠してはいません。

char型演算

char型演算を実行します。**c88**は、式の結果がまったく同じになる場合、int型演算で評価されるのとまったく同じように、8ビットのchar型演算を使用してコードを生成します。"1.2.3.2 char型演算"を参照してください。

初期化されていない定数の処理

暗黙的にゼロ初期化を実行する代わりに、初期化されていない定数のROMデータに対して記憶域を定義します。コンパイラは、"`const char i[1];`"に対して"`DS 1`"を生成します。

キーワード言語拡張

_near、_far、_sfrbyteなどのキーワード言語拡張機能をオンにします。

有効文字数

識別子で、ANSI-C変換制限で定められている31の有効文字の代わりに、500の有効文字を許可します。ただし、それ以上の有効文字は、予告なく切り捨てられます。

C++スタイルのコメント

CソースコードでC++スタイルのコメントを許可します。

例：

```
// e.g. this is a C++ comment line.
```

__STDC__定義

__STDC__が"0"と定義されます。10進定数"0"は、ANSI非準拠のインプリメンテーションを意味します。

古いスタイルの関数の引数昇格

プロトタイプチェックのときに、古いスタイルの関数パラメータを昇進させません。

unsigned char型の使用

0x80から0xffでは、unsigned char型を使用します。接尾語のない8進定数または16進定数の型は、値を表現する対応リストの最初に記述されます。

```
char, unsigned char, int, unsigned int, long, unsigned long
```

lvalueキャスト

不完全型voidを持つlvalueオブジェクトの型キャスト、およびlvalueオブジェクトの型とメモリを変更しないlvalueキャストを許可します。

例：

```
void *p; ((int*)p)++;   /* allowed */
int i; (char)i=2;       /* NOT allowed */
```

非定数文字列ポインタへの定数文字列の割り当てチェック

非定数文字列ポインタへの定数文字列の割り当てをチェックしません。このオプションでは、次の例でも、警告が出されません。

```
char *p;
void main( void ) { p = "hello"; }
```

1.2.21 移植性の高いCコード

c88を使用して、S1C88用のCコードを開発している場合、開発に使用しているホストでCコンパイラを使用して、そのホスト上で一部のコードをテストしたいと考えるかも知れません。この目的のため、インクルードファイルc88.hが用意されています。このヘッダファイルは、_C88が定義されているかどうかチェックして(**c88**のみ)、記憶タイプ指定子が定義されていない場合、再定義します。

このインクルードファイルを使用すると、(必要に応じて)記憶タイプ指定子を使用し、"移植性の高いCコード"を記述することができます。

さらに、S1C88 C組み込み関数は、他のANSIコンパイラによって認識されないため、これらの関数に対する適応済みのプロトタイプも用意されています。これらの関数を使用する場合、シミュレーションのために、S1C88と同じジョブを実行するコードをCで記述して、これらの関数をユーザのアプリケーションとリンクする必要があります。

1.2.22 上手なプログラミングの方法

c88で最上のコードを得たい場合、次のガイドラインを守ってください。

1. 常に関数プロトタイプを使用します。こうすると、char変数をintに昇格させずに、charとして渡すことができます。
2. (スモールモデルを使用できないために)ラージモデルを使用している場合、もっとも頻繁に使用される変数(静的)を、記憶タイプ_nearで宣言してみてください。コードの移植性を維持しておきたい場合は、registerキーワードを使用することもできます。
3. できる限り、unsigned修飾子を使用してください(たとえばfor (i = 0; i < 500; i++))。符号なしの比較の場合、符号付きの比較の場合より、コードが小さくてすみます。
4. できる限り、小さなデータ型を使用するようにしてください。たとえば、小さなループの場合は、char型を使用します。"1.2.3.2 char型演算"も参照してください。

1.3 ランタイム環境

1.3.1 スタートアップコード

Cモジュールをライブラリとリンクするとき、自動的に、Cスタートアップコードを含むオブジェクトモジュールをリンクすることになります。このモジュールは、cstart.objという名前で、すべてのCライブラリに存在します(すべてのモデルおよび実行モードに1つずつ)。

このモジュールは、S1C88 Cアプリケーションのランタイム環境を指定するものであるため、必要に合わせて編集する必要があります。そのため、このモジュールは、libディレクトリのlibサブディレクトリにあるsrcのcstart.cファイル内のソースに含まれています。通常の場合、テンプレートスタートアップファイルを自身のディレクトリにコピーしてそれを編集します。スタートアップコードには、スタートアップコードを調整するためのマクロプリプロセッサシンボルが含まれています。起動書式は、次のようになります(**cc88**コントロールプログラムを使用する場合)。

```
cc88 -Ms -c cstart.c
```

Cスタートアップコードでは、リセットベクタおよびS1C88 C環境のセットアップ用として絶対コードセクションが定義されています。リセットベクタには、_STARTラベルへのジャンプが含まれています。このグローバルラベルは、Cコンパイラが参照するため削除できません。これはまた、アプリケーションのデフォルト開始アドレスとしても使用されます(ロケータ記述言語DELFEEのstartキーワードを参照)。未使用の割り込みベクタ用コード空間は、**lc88**のロケータ記述ファイルで予約してユーザコードセクションには使用されないようにしてあります。必要によっては、この空間をユーザコードセクションとして使用するように設定することもできます。

ロケータ記述ファイル(ディレクトリetcの.dsc)では、キーワードstackでスタックが定義されています。そのため、このセクションにはstackという名前がついています。スタックの詳細については、"1.3.4 スタック"を参照してください。

ヒープは、この記述ファイルで、キーワードheapにより定義されています。そのため、このセクションにはheapという名前がついています。ヒープの詳細については、"1.3.5 ヒープ"を参照してください。

スタートアップコードは、さまざまなRAM領域にある初期化済みのC変数にも注意を払います。それぞれのメモリタイプには、ROMセクションおよびRAMセクションの両方で固有の名前が付けられています。スタートアップコードは、これらの特殊なセクションとランタイムライブラリ関数を使用して、初期化されたC変数の初期値をROMからRAMへコピーします。C変数の初期化が必要ない場合、-**DNOCOPY**によりstartup.asmファイルを変換することができます。ロケータ記述言語DELFEEのtableキーワードも参照してください。

上記で説明されていることがすべて実行された後、グローバルラベル_main(**cc88**によりC関数main()で生成されたもの)を使用して、Cアプリケーションが呼び出されます。

Cアプリケーションが"戻る"(これは通常、組み込み環境では発生しません)とき、プログラムは、アセンブララベル__exitを使用してSLP命令で終了します。デバッガを使用すると、このラベルにブレークポイントをセットし、プログラムが最後まで到達したこと、またはライブラリ関数exit()が呼び出されたことを示すことができます。

S1C88マイクロコントローラのスタートアップコードには、新しい機能がもう1つあります。それがウォッチドッグタイマの操作です。ほとんどのアプリケーションに共通する問題として、NMI/ウォッチドッグ割り込み処理が忘れられていることがあります。そのため、ウォッチドッグをオフにできないことから、予期できない結果が引き起こされます。これに対応するため、このスタートアップコードでは、デフォルトでウォッチドッグの操作が行われるようになっています。アプリケーションでNMI/ウォッチドッグ自体の操作が必要な場合は、スタートアップコードをそのアプリケーション用にコンパイルし直さなければなりません。

次のマクロは、cstart.cの機能をコントロールするときに使用することができます。

NOCOPY - BSSセクションをクリアしDATAセクションを初期化するコードを生成しません。

1.3.2 レジスタの使用法

c88は、使用可能なレジスタをできる限り効率的に使おうとします。コンパイラは、柔軟なレジスタ割り当てスキームを使用します。そのため、Cコードを変更すると、結果的にレジスタの使用法が変わってくる可能性があります。

c88は、パラメータを関数**c88**に渡すときには、固定スキームを使用します。

- 引数は、A、B、L、H、YP、XP、BA、HL、IX、IYのレジスタ経由で渡されます。char引数は、バイトレジスタA、L、YP、XP、H、B経由で渡されます。intは、ワードレジスタBA、HL、IX、IY経由で渡されます。longの引数は、32ビットレジスタセット、HLBAおよびIYIX経由で渡されます。
- 構造体と共用体は、スタック経由で渡されます。
- ニアポインタは、レジスタIY、IX、HL、BA経由で渡されます。ファーポインタは、レジスタセット、IYP、IXP、HLP経由で渡されます(ただしIYP=IY+YP、IXP=IX+XP、HLP=HL+A)。
- レジスタに渡される引数が多すぎる場合、引数はスタック経由で渡されます。

C関数の戻り型については、次のレジスタが使用されます。

戻り型	レジスタ	説　明
char	A	アキュムレータ
short/int	BA	
long	HLBA	(HL high word, BA low word)
ポインタ	HLP	HL+A

- 構造体と共用体は、スタック上に返されます。

1.3.3 セクションの使用法

c88は、さまざまなセクションを使用します。コンパイラは、出力で、使用するセクションごとにDEFSECT指示文を生成します。次のリストは、使用されるセクション名の概要を示しています。

セクション名	コメント
.text .text_function .comm	model s and c: コード model d and l: コード code with _common qualifier _interrupt code
.nbss .fbss	cleared _near data cleared _far data
.nbssnc .fbssnc	non-cleared _near data non-cleared _far data
.ndata .fdata	initialized _near data initialized _far data
.nrdata .frdata	const _near data const _far data

1.3.4 スタック

S1C88プロセッサには、最大64Kバイトの"システムスタック"があります。このシステムスタックは、関数の呼び出し、割り込み、関数のパラメータ、オートマチックなどに使用されます。静的な関数は、この目的のため、重ね書き可能なセクションを使用します。

次の図は、再入可能な(デフォルト)関数を使用するときのシステムスタックの構造を示しています。

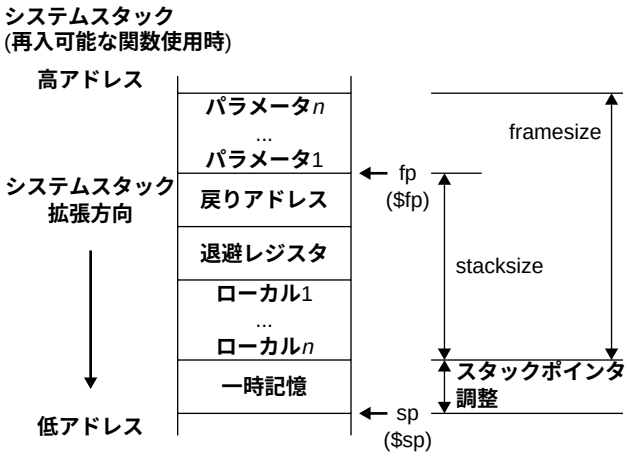


図1.3.4.1　スタックの構造

スタックは、ロケータ記述ファイル(ディレクトリetcの.dsc)で、キーワードstackにより定義されています。そのため、このセクションにはstackという名前が付いています。記述ファイルは、**lc88**に対して、スタックを割り当てる場所を知らせます。

記述ファイルでは、スタックサイズを、キーワードlength=*size*によりコントロールできます。スタックサイズを指定しない場合、ロケータは、スタートアップコードと同じようにして、残りの使用可能RAMをスタックに割り当てます。アプリケーション内で、ロケータ定義ラベル__lc_bsおよび__lc_esを使用することで、スタックの開始アドレスと終了アドレスを取得することができます。ただし、ロケータは、アプリケーションがロケータ定義シンボル__lc_bsおよび__lc_esのいずれかを参照する場合、スタックセクションの割り当てのみを行います。スタックが大きくなっても対応できるように、スタックに割り当てるメモリ空間が十分なければなりません。

再入不可能な関数の場合、(非レジスタ型の)オートマチックおよび(非レジスタ型の)パラメータが静的な領域に割り当てられるため、スタック空間は使用できません。

1.3.5 ヒープ

ヒープは、動的メモリ管理ライブラリ関数、malloc()、calloc()、free()、realloc()を使用する場合に限り必要になります。ヒープは、メモリ上に予約されている領域です。上記のメモリ割り当て関数を使用する場合に限り、ロケータは、ロケータ記述ファイルのキーワードheapの指定に従って、自動的にヒープを割り当てます。

heapという名前の特殊なセクションは、ヒープ領域の割り当てに使用されます。ヒープセクションは、ロケータ記述ファイルを使用することで、メモリ上の任意の場所に置くことができます。ヒープのサイズは、ロケータ記述ファイルのキーワードlength=*size*を使用して指定できます。ヒープサイズを指定しないときに(たとえばmalloc()の呼び出しなどで)それを参照すると、ロケータは、使用可能なメモリの残りをヒープに割り当てます。ロケータ定義ラベル__lc_bhと__lc_eh(ヒープの最初と最後)は、ライブラリ関数sbrk()で使用されます。この関数は、ヒープでメモリが必要になったときにmalloc()によって呼び出されます。

ロケータ記述ファイル内の、ヒープサイズとその位置を定義する箇所の例を示します。

```
amode data
{
        section selection=w;
        heap length=1000;  // heap (only when used)
```

特殊なheapセグメントは、そのロケータラベルがプログラムで使用されている場合に限り割り当てられます。

コンパイラのスモールモデルまたはコンパクトデータモデルで構築されたアプリケーションでヒープが必要な場合は、ロケータ記述ファイルを変更しなければなりません。このファイルを変更しない場合、ロケータはエラーをレポートします。またヒープの宣言についても、"amode data"から"amode data_short"に移行しなければなりません。

1.3.6 割り込み関数

割り込み関数は、_interrupt(*n*)関数修飾子を使用することにより、Cに直接実装されています。この修飾子で宣言された関数は、次のような点で通常の関数定義と異なります。

1. 割り込み関数の実行時に破壊される可能性があるすべてのレジスタは、関数のエントリポイントで保存され、関数の終了時にリストアされます。通常、割り込み関数によって直接使用されるレジスタのみが保存されます。
2. この関数は、RET命令ではなくRETE命令で終了します。

例：

```
; S1C88 C compiler v99.9 r9  SN00000000-000 (c) year TASKING, Inc.
; options: -n -s
$CASE ON

        NAME    INTERPT
;        interpt.c
; 1     |int flag;
; 2     |
; 3     |_interrupt( 0x30)
; 4     |void handler ( void )
; 5     |{
        GLOBAL  _handler
        DEFSECT " .code48", CODE AT 030H
        SECT    ".code48"
        DW      _handler
        DEFSECT ".comm", CODE, SHORT
        SECT    ".comm"
_handler:
        PUSH    ale
; 6     |    flag=1
        LD      iy,#01h
        LD      [_flag], iy
; 7     |}
        POP      ale
        RETE
        DEFSECT ".bss", DATA, SHORT, CLEAR
        SECT    ".bss"
        GLOBAL  _flag
_flag:  DS      2
        EXTERN  (DATA) __lc_es
        END
```

1.4 コンパイラの使用

1.4.1 コントロールプログラム

コントロールプログラム**cc88**には、1行のコマンド行から、S1C88ツールチェーンのいくつかの異なるコンポーネントを起動する機能があります。コントロールプログラムは、コマンド行に記述されたソースファイルとオプションを任意の順序で受け付けます。

コントロールプログラムの起動書式は、次のようになります。

cc88 [[*option*]...[*control*]...[*file*]...] ...

オプションの前には、"-"(マイナス記号)を付けます。入力ファイル*file*には、以下で説明する拡張子を付けることができます。

コントロールプログラムは、次の引数タイプを認識します。

- "-"文字で始まる引数はオプションです。オプションによっては、コントロールプログラム自身によって解釈されるものもありますが、多くのオプションは、そのオプションを受け付けるツールチェーンのプログラムに渡されます。
- .c接尾辞が付いた引数は、Cソースプログラムと見なされ、コンパイラに渡されます。
- .asm接尾辞が付いた引数は、アセンブラソースファイルと見なされ、前処理されてアセンブラに渡されます。
- .src接尾辞が付いた引数は、コンパイルされたアセンブラソースファイルと見なされ、直接アセンブラに渡されます。
- .a接尾辞が付いた引数は、ライブラリファイルと見なされ、リンカに渡されます。
- .obj接尾辞が付いた引数は、オブジェクトファイルと見なされ、リンカに渡されます。
- .out接尾辞が付いた引数は、リンクされるオブジェクトファイルと見なされ、ロケータに渡されます。ロケータは、起動時に1つの.outファイルだけを受け付けます。
- .dsc接尾辞が付いた引数は、ロケータコマンドファイルとして処理されます。コマンド行に拡張子.dscを持つファイルがある場合、コントロールプログラムは、ロケートフェーズを追加する必要があると見なします。.dsc接尾辞を持つファイルがない場合、コントロールプログラムは、リンクが終わったら停止します(ただし前のフェーズで停止するよう指示されていない場合)。
- 他のファイルは、すべてオブジェクトファイルと見なされ、リンカに渡されます。

通常、コントロールプログラムは、リンクロケートフェーズで絶対出力ファイルを生成した後、すべてのソースファイルをコンパイルおよびアセンブルしてオブジェクトファイルを生成します。ただし、アセンブラ、リンカ、ロケータのそれぞれの段階を抑制するためのオプションもあります。コントロールプログラムは、コンパイルプロセスの中間段階として固有のファイル名を生成します(後で削除される)。コンパイラとアセンブラが連続して呼び出される場合、コントロールプログラムはコンパイラの前処理によってアセンブラファイルが生成されないようにします。通常アセンブラ入力ファイルが最初に前処理されます。

次のオプションが、コントロールプログラムによって解釈されます。

コントロールプログラムのオプション

オプション	説　明
-Mc	コンパクトコードメモリモデル
-Md	コンパクトデータメモリモデル
-Ml	ラージメモリモデル
-Ms	スモールメモリモデル
-Ta *arg*	引数を直接アセンブラに渡します。
-Tc *arg*	引数を直接Cコンパイラに渡します。
-Tlk *arg*	引数を直接リンカに渡します。
-Tlc *arg*	引数を直接ロケータに渡します。
-V	バージョンヘッダのみを表示します。
-al	絶対リストファイルを生成します。
-c	リンクしません。.objの段階で停止します。
-cl	ロケートしません。.outの段階で停止します。
-cs	アセンブルしません。Cファイルをコンパイルし.srcファイルを出力して停止します。
-f *file*	*file*から引数を読み込みます("-"は標準入力を表す)。
-ieee	ロケータの出力ファイルフォーマットをIEEE-695に設定します(デフォルト)。
-nolib	標準ライブラリとリンクしません。
-o *file*	出力ファイルを指定します。
-srec	ロケータの出力ファイルフォーマットをMotorola Sレコードに設定します。
-tmp	中間ファイルを削除しません。
-v	コマンドの起動状況を表示します。
-v0	コマンドの起動状況を表示しますが、実際には起動しません。

1.4.1.1 コントロールプログラムのオプションについての詳細な説明

-M{s|c|d|l}　使用するメモリモデルを指定します。
スモール　(**s**)
コンパクトデータ　(**d**)
コンパクトコード　(**c**)
ラージ　(**l**)

-Ta*arg*
-Tc*arg*
-Tlk*arg*
-Tlc*arg*
これらのオプションを使用すると、コマンド行引数を直接アセンブラ(**-Ta**)、Cコンパイラ(**-Tc**)、リンカ(**-Tlk**)、ロケータ(**-Tlc**)に渡すことができます。これらのオプションは、コントロールプログラムが認識できないオプションを、適切なプログラムに渡すときに使用することができます。引数は、これらのオプションに直接追加するか、コントロールプログラムの独立した引数としてオプションの後に続けます。

-V　コントロールプログラムが終了する直前に、バージョン番号を含む著作権ヘッダが表示されます。

-al　アプリケーションのそれぞれのモジュールで絶対リストファイルを生成します。

-c
-cl
-cs
通常、コントロールプログラムは、すべての段階を起動して、指定された入力ファイルから絶対ファイルを構築します。これらのオプションを付けると、Cコンパイラ、アセンブラ、リンカ、ロケータのそれぞれの段階をスキップすることができます。**-cs**オプションを付けると、コントロールプログラムは、Cソースファイル(.c)のコンパイルの後、およびアセンブラソースファイル(.asm)の前処理の後停止して、生成される.srcファイルをそのまま保持します。**-c**オプションを付けると、コントロールプログラムは、アセンブラの後停止し、1つまたは複数のオブジェクトファイル(.obj)を出力します。**-cl**オプションを付けると、コントロールプログラムは、リンク段階の後停止し、1つまたは複数のリンカオブジェクトファイル(.out)を出力します。

-f *file*

コマンド行引数を*file*から読み込みます。標準出力を示すときは、ファイル名に"-"を使用することができます。コマンド行のサイズの制限を回避するために、コマンドファイルを使用することができます。これらのコマンドファイルには、実際のコマンド行で使用できないオプションを入れます。コマンドファイルは、makeユーティリティなどを使用して簡単に作成できます。コマンドファイルには、次のような簡単な規則があります。

1. コマンドファイルの同じ行に複数の引数を指定することができます。
2. 引数にスペースを入れるときは、その引数を一重引用符または二重引用符で囲みます。
3. 引用符の付いた引数の中で一重引用符または二重引用符を使用する場合、次の規則に従います。
   a. 中に入っている引用符が、一重引用符のみまたは二重引用符のみの場合、引数を囲むときもう一方の引用符を使用します。つまり、引数に二重引用符が含まれる場合、引数を一重引用符で囲みます。
   b. 両方の引用符が使用されている場合、それぞれの引用符がもう一方の引用符で囲まれるような形で、引数を分割する必要があります。

   例：
   ```
   "This has a single quote ' embedded"
   ```
   または
   ```
   'This has a double quote " embedded'
   ```
   または
   ```
   'This has a double quote " and a single quote '"' embedded"
   ```
4. オペレーティングシステムによっては、テキストファイル内の行の長さに制限がある場合があります。この制限を回避するため、継続行を使用することができます。これらの行は、最後にバックスラッシュと改行が付きます。引用符の付いた引数の場合、継続行は、次の行のスペースを取らないでそのままつなげられます。引用符が付いていない引数の場合、次の行にあるすべてのスペースは削除されます。

   例：
   ```
   "This is a continuation ¥
   line"
           → "This is a continuation line"
   control(file1(mode,type),¥
           file2(type))
           → control(file1(mode,type),file2(type))
   ```
5. コマンド行ファイルは、最高で25レベルまでネストすることができます。

-ieee
-srec

これらのオプションを使用すると、絶対ファイルのロケータ出力フォーマットを指定することができます。出力ファイルは、IEEE-695ファイル(`.abs`)、Motorola Sレコードファイル(`.sre`)のいずれかにすることができます。デフォルト出力は、IEEE-695ファイル(`.abs`)になります。

-nolib

このオプションを使用すると、コントロールプログラムが標準ライブラリをリンカに提供しなくなります。通常の場合、コントロールプログラムは、リンカにデフォルトCライブラリとランタイムライブラリを提供します。必要なライブラリは、コンパイラのオプションによって決まります。

-o *file*

通常、このオプションは、出力ファイル名を指定するためにロケータに渡されます。**-cl**オプションを使用してロケートフェーズを抑制する場合、**-o**オプションがリンカに渡されます。**-c**オプションを使用してリンクフェーズを抑制すると、ソースファイルが1つだけ指定されている場合に限り、**-o**オプションがアセンブラに渡されます。**-cs**オプションを使用してアセンブラフェーズを抑制すると、**-o**オプションがコンパイラに渡されます。引数は、これらのオプションに直接追加するか、コントロールプログラムの独立した引数としてオプションの後に続けます。

-tmp このオプションを使用すると、コントロールプログラムがカレントディレクトリに中間ファイルを作成するようになります。これらのファイルは自動的に削除されません。通常、コントロールプログラムは、中間変換結果として一時ファイルを生成します。これには、コンパイラが生成するアセンブラファイル、オブジェクトファイル、リンカ出力ファイルなどのようなものがあります。変換プロセスの次のフェーズが無事に完了したら、これらの中間ファイルは削除されます。

-v **-v**オプションを使用すると、それぞれのプログラムの起動状況が標準出力に表示されます。このとき、"+"が行の最初に付けられます。

-v0 このオプションは、**-v**オプションと同じ働きをしますが、起動状況を表示してもプログラムが実際に起動しないという点で異なります。

1.4.1.2 環境変数

コントロールプログラムは、次の環境変数を使用します。

TMPDIR この変数は、コントロールプログラムが一時ファイルの作成時に使用するディレクトリを指定するときに使用できます。この環境変数が設定されていない場合、一時ファイルはカレントディレクトリに作成されます。

CC88OPT コントロールプログラム**cc88**の起動時に余分のオプションや引数を渡すとき、この環境変数を使用できます。コントロールプログラムは、コマンド行引数の前にこの変数で指定された引数を処理します。

CC88BIN この変数が設定されているとき、コントロールプログラムは、起動するツールの名前の前に、この変数で指定されているディレクトリを追加します。

1.4.2 コンパイラ

Cコンパイラの起動書式は、次のようになります。

c88 [[*option*]...[*file*]...] ...

Cコンパイラは、Cソースファイル名とコマンド行オプションをランダムな順序で受け付けます。ソースファイルは、コマンド行に現れるのと同じ順序(左から右)で処理されます。オプションには、最初に"-"が付けられます。それぞれのCソースファイルは、別々にコンパイルされます。コンパイラは、Cソースモジュールごとに、接尾辞.srcの付いた出力ファイル(アセンブラソースコードが含まれる)を生成します。

オプションの優先順位は左から右になります。2つのオプションが競合する場合、最初(もっとも左)のオプションが有効になります。**-o**オプションを付けることで、デフォルト出力ファイル名を無効にすることができます。コンパイラは、それぞれの**-o**オプションを一度しか使用しないため、複数のソースファイルで複数の**-o**オプションを指定することができます。

オプションの要約を以下にまとめます。次の節では、それぞれのオプションについて詳細に説明します。

コンパイラのオプション

オプション	説　明
-D*macro*[=*def*]	プリプロセッサ*macro*を定義します。
-H *file*	コンパイルの前に*file*をインクルードします。
-I*directory*	*directory*でインクルードファイルを探します。
-M{s|c|d|l}	メモリモデルを選択します。それぞれ、スモール、コンパクトコード、コンパクトデータ、ラージに対応します。
-O{0|1}	最適化を制御します。
-V	バージョンヘッダのみを表示します。
-e	コンパイラエラーが発生した場合、出力ファイルを削除します。
-err	診断をエラーリストファイル(.err)に送信します。
-f *file*	オプションを*file*から読み込みます。
-g	シンボリックデバッグ情報を有効にします。
-o *file*	出力ファイルの名前を*file*で指定します。
-s	Cソースコードをアセンブラ出力とマージします。
-w[*num*]	1つまたはすべての警告メッセージを抑制します。

コンパイラオプション(機能別)

説　明	オプション
インクルードオプション	
オプションを*file*から読み込みます。	**-f** *file*
コンパイルの前に*file*をインクルードします。	**-H** *file*
*directory*でインクルードファイルを探します。	**-I***directory*
前処理オプション	
プリプロセッサ*macro*を定義します。	**-D***macro*[=*def*]
コード生成オプション	
それぞれ、スモール、コンパクトコード、コンパクトデータ、ラージに対応します。	**-M{s|c|d|l}**
最適化を制御します。	**-O{0|1}**
出力ファイルオプション	
コンパイラエラーが発生した場合、出力ファイルを削除します。	**-e**
出力ファイルの名前を*file*で指定します。	**-o** *file*
Cソースコードをアセンブラ出力とマージします。	**-s**
診断オプション	
バージョンヘッダのみを表示します。	**-V**
診断をエラーリストファイル(.err)に送信します。	**-err**
シンボリックデバッグ情報を有効にします。	**-g**
1つまたはすべての警告メッセージを抑制します。	**-w**[*num*]

1.4.2.1 コンパイラオプションの詳細な説明

以下に、オプションの文字をリストしています。それぞれのオプション(**-o**を除く、**-o**オプションの説明を参照)は、すべてのソースファイルに適用されます。同じオプションが複数回使用された場合、最初に登場したオプション(もっとも左)が使用されます。コマンド行オプションの配置は、**-I**オプションと**-o**オプションを除いて重要になります。**-o**オプションの場合、ファイル名は、オプションのすぐ後に続けることができません。間にタブまたはスペースを入れる必要があります。他のオプション引数は、オプションの直後に続ける必要があります。ソースファイルは、コマンド行に現れるのと同じ順序(左から右)で処理されます。

一部のオプションには、対応するプラグマがあります。

-D

オプション：

-D*macro*[=*def*]

引数：

定義したいマクロおよびその定義(オプション)。

説明：

*macro*には、#defineと同じようにプリプロセッサを定義します。*def*が指定されていない場合("="がない場合)、"1"が指定されていると見なされます。シンボルはいくつでも定義できます。定義は、条件コンパイルで、#if、#ifdef、#ifndefを持つプリプロセッサによりテストすることができます。コマンド行が、使用しているオペレーティングシステムの制限より長くなる場合、**-f**オプションが必要になります。

なお、ANSIでは、以下のシンボルが定義済みです。

__FILE__	"現在のソースファイル"
__LINE__	現在のソース行番号(int型)
__TIME__	"hh:mm:ss"
__DATE__	"Mmm dd yyyy"
__STDC__	ANSI標準のレベル。このマクロは常に0となります。

また、**c88**を起動すると、以下のシンボルが生成されます。

_C88	このコンパイラを示す、定義済みのシンボル。このシンボルは、**c88**コンパイラのみが認識するソースの部分にフラグを立てるときに使用することができます。コンパイラのバージョン番号が追加されます。
_MODEL	コンパイルするモジュールのメモリモデルを示します。プリプロセッサがテストできる1文字または2文字(内部RAMバンク0の場合"0"、内部RAMバンク1の場合"1")が追加されます。詳細については、"1.2.2.2 メモリモデル"を参照してください。

例：

次のコマンドは、シンボルNORAMを1と定義し、シンボルPIを3.1416と定義します。

```
c88 -DNORM -DPI=3.1416 test.c
```

-e

オプション：

-e

説明：

コンパイラエラーが発生した場合、出力ファイルを削除します。このオプションを使用すると、makeユーティリティが常に正しく動作するようになります。

例：

```
c88 -e test.c
```

-err

オプション：

-err

説明：

エラーをstderrの代わりにファイル*source*.errに書き込みます。

例：

エラーをstderrの代わりにtest.errに書き込む場合、次のコマンドを実行します。

```
c88 -err test.c
```

-f

オプション：

-f *file*

引数：

コマンド行処理に使用するファイル名。標準入力を示すときは、ファイル名"-"を使用できます。

説明：

コマンド行処理のために*file*を使用します。コマンド行のサイズ制限を回避するために、コマンドファイルを使用することができます。これらのコマンドファイルに入れるオプションは、実際のコマンド行では使用できないものです。コマンドファイルは、makeユーティリティなどを使用して簡単に作成できます。

-fオプションは複数使用することができます。

コマンドファイルには、次のような簡単な規則があります。

1. コマンドファイルの同じ行に複数の引数を指定することができます。
2. 引数にスペースを入れるときは、その引数を一重引用符または二重引用符で囲みます。
3. 引用符の付いた引数の中で一重引用符または二重引用符を使用する場合、次の規則に従います。
   a. 中に入っている引用符が、一重引用符のみまたは二重引用符のみの場合、引数を囲むときもう一方の引用符を使用します。つまり、引数に二重引用符が含まれる場合、引数を一重引用符で囲みます。
   b. 両方の引用符が使用されている場合、それぞれの引用符がもう一方の引用符で囲まれるような形で、引数を分割する必要があります。

   例：
   ```
   "This has a single quote ' embedded"
   ```
   または
   ```
   'This has a double quote " embedded'
   ```
   または
   ```
   'This has a double quote " and a single quote '"' embedded"
   ```
4. オペレーティングシステムによっては、テキストファイル内の行の長さに制限がある場合があります。この制限を回避するため、継続行を使用することができます。これらの行は、最後にバックスラッシュと改行が付きます。引用符の付いた引数の場合、継続行は、次の行のスペースを取らないでそのままつなげられます。引用符が付いていない引数の場合、次の行にあるすべてのスペースは削除されます。

   例：
   ```
   "This is a continuation ¥
   line"
           → "This is a continuation line"
   control(file1(mode,type),¥
            file2(type))
           → control(file1(mode,type),file2(type))
   ```
5. コマンド行ファイルは、最高で25レベルまでネストすることができます。

例：

ファイルmycmdsに次の行があると仮定します。

```
-err
test.c
```

コマンド行は、次のようになります。

```
c88 -f mycmds
```

-g

オプション：

-g

説明：

出力ファイルに指示文を追加し、シンボリック情報を組み込んで、高レベルデバッグを可能にします。コンパイラが高レベルの最適化に設定されている場合、デバッグに手間がかかるようになることもあります。

例：

シンボリックデバッグ情報を出力ファイルに追加するときは、次のコマンドを実行します。

```
c88 -g test.c
```

参照：

-O

-H

オプション：

-H*file*

引数：

インクルードファイルの名前。

説明：

Cソースのコンパイルの前に*file*をインクルードします。これは、Cソースの最初の行で#include "*file*"を指定するのと同じです。

例：

```
c88 -Hstdio.h test.c
```

参 照 ：

-I

-I

オプション：

-I*directory*

引数：

インクルードファイルを探すディレクトリの名前。

説明：

名前に絶対パス名がない#includeファイルを検索するときのアルゴリズムを変更し、*directory*を探します。そのため、#includeファイルの名前が""で囲まれている場合は、#include行を含むファイルのディレクトリが最初に探され、次に**-I**オプションで指定されているディレクトリが左から右の順に探されます。それでもインクルードファイルが見つからない場合、コンパイラは、環境変数C88INCで指定されているディレクトリを探します。C88INCには、複数のディレクトリが含まれる可能性があります。

最後に、コンパイラバイナリがあるディレクトリの相対パス`../include`が探されます。これは、コンパイラパッケージに標準で付属するインクルードディレクトリです。

#includeファイルの名前を<>で囲んで指定する場合、#include行を含むファイルのディレクトリは検索されません。ただし、**-I**オプションで指定されたディレクトリ(その他C88INCのディレクトリおよび相対パスも)は検索されます。

例：

```
c88 -I/proj/include test.c
```

参照：

"1.4.3 インクルードファイル"

-M

オプション：

-M*model*

引数：

使用するメモリモデル。*model*は次のいずれかになります。

s スモール
c コンパクトコード
d コンパクトデータ
l ラージ

デフォルト：

-Ms

説明：

使用するデータメモリモデルを指定します。

例：

```
c88 -M1 test.c
```

参照：

"1.2.2.2 メモリモデル"

-O

オプション：

-O*flag*

引数：

0または1

デフォルト：

-O1

説明：

最適化を制御します。最適化のオン、オフを切り換えるために、0または1の数字を指定します。

-O0 最適化はオフになります。

-O1 デフォルト。**c88**が最小のコードを生成するよう最適化を設定します。

以下に、最適化項目と**-O**オプション指定による実行内容を説明します。

エイリアスチェックの緩和

-O1を使用するとエイリアスチェックが緩和されます。書き込み操作が間接(計算)アドレス経由で行われる場合、レジスタに記憶されているユーザ変数の内容が消去されません。

-O0を使用すると、厳密なエイリアスチェックが実行されます。このオプションを指定すると、書き込み操作が間接(計算)アドレス経由で行われる場合、コンパイラは、レジスタに記憶されているユーザ変数のすべての内容を消去します。

初期化されていないスタティック変数およびグローバル変数のクリア

-O1、**-O0**のどちらを指定した場合でも、コンパイラは、初期化されていないスタティック変数とグローバル変数を"クリア"します。

共通項のまとめ

-O1を使用すると、CSE(共通項のまとめ)がオンになります。このオプションを指定すると、コンパイラはCコード内で共通式を探します。共通式は1回だけ評価され、その結果が一時的にレジスタ内に保持されます。

-O0を使用すると、CSE(共通項のまとめ)がオフになります。このオプションを指定すると、コンパイラは共通式を探しません。また、エイリアスチェック、式の置換、不変コードのループ外への移動がすべて無効になります。

例：

```
/*
 * Compile with -O0,
 * Compile with -O1, common subexpressions are found
 *     and temporarily saved.
 */
char x, y, a, b, c, d;
void
main( void )
{
       x = (a * b) - (c * d);
       y = (a * b) + (c * d);   /*(a*b) and (c*d) are common */
}
```

データフロー、明示的代入文の定数式化

-O1を使用すると、定数およびコピーの伝搬がオンになります。このオプションを指定すると、コンパイラは定数値が変数へ割り当てられている場合を探し、それ以降その変数が他の変数に割り当てられていれば定数値で置き換えます。

-O0を使用すると、定数とコピーの伝搬がオフになります。

例：

```
/*
 * Compile with -O0, 'i' is actually assigned to 'j'
 * Compile with -O1, 15 is assigned to 'j', 'i' was propagated
 */
int i;
int j;
void
main( void )
{
       i = 10;
       j = i + 5;
}
```

式の置換

-O1を使用すると、式の置換がオンになります。このオプションを指定すると、コンパイラは式が変数へ割り当てられている場合を探し、それ以降その変数が他の変数に割り当てられていればその式で置き換えます。

-O0を使用すると、式の置換がオフになります。

例：

```
/*
 * Compile with -O0, normal cse is done
 * Compile with -O1, 'i+j' is propagated.
 */
unsigned i, j;
int
main( void )
{
       static int a;
       a = i + j;
       return (a);
}
```

コードフロー、順序の変更

-O1を使用すると、中間ファイルで制御フローの最適化とコード順序の変更がオンになります。たとえば、ジャンプの変更や条件ジャンプの反転などがこれに該当します。

-O0を使用すると、制御フローの最適化がオフになります。

例：

次の例は、制御の最適化の状況を示しています。

```
/*
 * Compile with -O0
 * Compile with -O1, compiler finds first time 'i' is
 * always < 10, the unconditional jump is removed.
 */
int i;
void
main( void )
{
       for( i=0; i<10; i++ )
       {
              do_something( );
       }
}
```

次の例は、条件ジャンプの反転の状況を示しています。

```
/*
 * Compile with -O0, code as written sequential
 * Compile with -O1, code is rearranged
 * Code rearranging enables other optimizations to optimize better, e.g. CSE
 */
int     i;
extern void   dummy( void );
void main( )
{
       do
       {
              if    ( i )
              {
                    i--;
              }
              else
              {
                    i++;
                    break;
              }
              dummy( );
       } while ( i );
}
```

ピープホール最適化

-O1を使用すると、ピープホール最適化がオンになります。これにより、冗長なコードが削除されます。ピープホールオプティマイザは、冗長な命令、または命令の数を少なくするために結合できる命令シーケンスを探します。

-O0を使用すると、ピープホール最適化がオフになります。

不変コードのループ外への移動

-O1を使用すると、不変コードがループ外へ移されます。

-O0を使用すると、不変コードのループ外への移動がオフになります。

例：

```
/*
 * Compile with -O0, normal cse is done
 * Compile with -O1, invariant code is found in
 *      the loop, code is moved outside the loop.
 */
void
main( void )
{
       char x, y, a, b;
       int i;
       for( i=0; i<20; i++ )
       {
              x = a + b;
              y = a + b;
       }
}
```

高速ループ（コードサイズの増加）

-O1、**-O0**のどちらを指定した場合でも、高速ループがオフになります。

コードサイズの縮小

-O1を使用すると、サイズの小さなコードが作成されます。可能であれば、使用される命令の数が削減されます。ただし、このオプションを使用すると、命令サイクルは多くなります。

-O0を使用すると、最小コード最適化がオフになります。

ループの展開

-O1、**-O0**のどちらを指定した場合でも、ループの展開がオフになります。

添字の効率的な最適化

-O1を使用すると、添字の効率的な最適化がオンになります。このオプションを指定すると、コンパイラは、インデックス変数に関連する式の効率的な最適化をしようとします。

-O0を使用すると、添字の効率的な最適化がオフになります。

例：

```
/*
 * Compile with -O0, disable subscript strength reduction
 * Compile with -O1, begin and end address of 'a' are
 * determined before the loop and temporarily put in registers
 * instead of determining the address each time inside the loop
 */
int i;
int a[4];
void
main( void )
{
       for( i=0; i<4; i++ )
       {
              a[i] = i;
       }
}
```

-O

オプション：

-o *file*

引数：

出力ファイルの名前。ファイル名はオプションの直後に続けられません。オプションの後にタブまたはスペースを入れる必要があります。

デフォルト：

.src接尾辞を持つモジュール名。

説明：

.src接尾辞を持つモジュール名の代わりに、*file*を出力ファイル名として使用します。最初に見つかった**-o**オプションが最初にコンパイルするファイルに適用され、2番目の**-o**オプションが2番目のコンパイルファイルに適用されるという具合になるため、このオプションを使用するときは、特に注意が必要です。

例：

次のコマンドが指定された場合、

```
c88 file1.c file2.c -o file3.src -o file2.src
```

file1.cがコンパイルされてfile3.srcが作成され、file2.cがコンパイルされてfile2.srcが作成されて、2つのファイルが作成されます。

-S

オプション：

-s

プラグマ：

source

説明：

Cソースコードを、生成されたアセンブラコードとマージして、ファイルに出力します。

例：

```
c88 -s test.c
```

```
;             test.c:
; 1    |int i;
; 2    |
; 3    |int
; 4    |main( void )
; 5    |{
         extern __START
         global _main
```

参照：

"1.4.4 プラグマ"

-V

オプション：

-V

説明：

バージョン情報を表示します。

例：

```
c88 -V
S1C88 C Compiler vx.y rz        SN00000000-015 (c) year TASKING, Inc.
```

-W

オプション：

-w[*num*]

引数：

オプションで、抑制する警告番号。

説明：

-wを使用すると、すべての警告メッセージが抑制されます。**-w***num*を使用すると、指定された警告メッセージが抑制されます。

例：

警告135を抑制するときは、次のコマンドを実行します。

```
c88 file1.c -w135
```

1.4.3 インクルードファイル

インクルードファイルは、<>で囲む方法と""で囲む方法の2つの方法で指定することができます。#include指示文がある場合、**c88**は次のアルゴリズムを使用して、インクルードファイルをオープンしようとします。

1. ファイル名が""で囲まれていて、そのファイル名が絶対パス名でない("\"が最初に付かない)場合、インクルードファイルは、#include行を含むファイルがあるディレクトリで検索されます。例を示します。

   ```
   c88 ..¥..¥source¥test.c
   ```

   c88は、最初にディレクトリ..¥..¥sourceでインクルードファイルを探します。

   このファイル(**c88 test.c**)があるディレクトリでソースファイルをコンパイルする場合、コンパイラは、インクルードファイルをカレントディレクトリで探します。

   ただし、この最初の手順は、<>で囲まれたインクルードファイルでは実行されません。

2. **-I**オプションで指定されたディレクトリを、左から右の順序で使用します。例を示します。

   ```
   c88 -I..¥..¥include demo.c
   ```

3. 環境変数C88INCがあるかどうかチェックします。この環境変数がある場合、その内容を、インクルードファイルのディレクトリ指定子として使用します。環境変数C88INCでは、区切り文字を使用して複数のディレクトリを指定することができます。上記の例で**-I**オプションを使用する代わりに、次のようにC88INCを使用して、同じディレクトリを指定することができます。

   ```
   set C88INC=..¥..¥include
   c88 demo.c
   ```

4. 上記の方法で探してもインクルードファイルが見つからない場合、コンパイラはサブディレクトリinclude(**c88**バイナリが含まれるディレクトリの1つ上のディレクトリ)を探します。例を示します。

   c88.exeがディレクトリC:¥C88¥BINにインストールされている場合、インクルードファイルが検索されるディレクトリは、C:¥C88¥INCLUDEになります。

   コンパイラは、このincludeディレクトリを探すため、実行時にバイナリがどのディレクトリから実行されているかを判断します。

c88は、必要な場合、ディレクトリ区切り文字を適宜挿入するため、**-I**オプションまたはC88INCで指定されているディレクトリ名の最後には、この区切り文字を付けても付けなくてもかまいません。

環境変数C88INCで複数のディレクトリを指定する場合、次の区切り文字を使用する必要があります。

; , スペース

例： `set C88INC=..¥..¥include;¥proj¥include`

1.4.4 プラグマ

ANSI(3.8.6)によると、次の形式で記述された前処理の指示文がある場合、

#pragma *pragma-token-list new-line*

コンパイラは、インプリメンテーションで定義されている方法で動作します。コンパイラは、以下のリストにないプラグマについては無視します。プラグマは、コンパイラのコードジェネレータに指示を与えます。プラグマの他に、コード生成プロセスを進行させる方法として、Cアプリケーションで使用するコマンド行オプションとキーワード(たとえば_near型変数)があります。コンパイラは、次の規則を使用して、これらの3つのグループを認識します。

コマンド行オプションより、キーワードとプラグマが優先されます。キーワードより、プラグマが優先されます。つまりプラグマの優先度がもっとも高いということになります。

このアプローチを利用すると、ソースモジュールのデフォルト最適化レベルを設定して、ソース内でプラグマにより一時的に無効とすることもできます。

Cコンパイラ**c88**は、次のプラグマをサポートしています。

asm その次の行(プリプロセッサ行以外)をアセンブラ言語ソースコードとして出力ファイルに挿入します。挿入された行については、構文がチェックされません。また、ピープホールオプティマイザのコードバッファがフラッシュされます。そのため、コンパイラはピープホールパターン置換のような最適化を停止して、**endasm**プラグマの後、関数の最初で開始するかのようにこれらの最適化を再開します。高度なアセンブラインラインについては、組み込み関数を使用することができます。定義された組み込み関数のセットは、S1C88特有の機能のほとんどをカバーしており、それ以外の方法ではC言語でアクセスすることはできません。組み込み関数の詳細については、"1.2.16 組み込み関数"を参照してください。

asm_noflush プラグマ**asm**とほとんど同じですが、ピープホールオプティマイザがコードバッファをフラッシュせず、レジスタの内容を有効と見なす点が異なります。

endasm C言語に戻します。"1.2.14 アセンブラの記述"で詳細に説明しています。

source **-s**オプションと同じです。Cソースとアセンブラソースを混在できるようにします。

nosource デフォルト。アセンブラコード内のCソースの生成をオフにします。

1.4.5 コンパイラの制限

ANSI C標準[1-2.2.4]では、さまざまな変換の制限を定義しているため、CコンパイラがANSI C標準に準拠するためには、この制限をサポートする必要があります。ANSI C標準では、以下の制限が少なくとも1つずつ含まれるプログラムを、コンパイラが変換して実行できるようにする必要があると規定しています。**c88**の実際の制限は、かっこで示しています。

実際のコンパイラの制限は、そのほとんどが、ホストシステムの空きメモリの量によって決まります。このような場合、"D"(ダイナミック)という文字がかっこ内に示されています。一部の制限は、内部コンパイラパーサスタックのサイズで決まります。これらの制限には、"P"というマークが付いています。このスタックのサイズは200ですが、実際の制限は、もっと低い値になることがあります。また変換されるプログラムの構造によっても影響を受けます。

- 複合文、繰り返し制御構造、選択制御構造のネスティングレベルは最高で15です(P>15)。
- 条件の包含のネスティングレベルは最高で8です(50)。
- 演算、構造体、共用体、宣言での不完全な型を変更する、ポインタ、配列、関数の宣言子(任意の組み合わせが可能)の数は最高で12です(15)。
- 完全な宣言子の中で使用されているかっこ付きの宣言子のネスティングレベルは最高で31です(P>31)。
- 完全な式の中で使用されているかっこ付きの式のネスティングレベルは最高で32です(P>32)。
- 外部識別子の有効文字は最高で31文字です(完全なANSI Cモード)。
  外部識別子の有効文字は最高で500文字です(非ANSI Cモード)。
- 1つの変換ユニットの外部識別子の数は最高で511です(D)。
- 1つのブロックで宣言されたブロック範囲の識別子の数は最高で127です(D)。
- 1つの変換ユニットで同時に定義できるマクロ識別子の数は最高で1024です(D)。
- 1つの関数宣言で使用できるパラメータの数は最高で31です(D)。
- 1つの関数呼び出しで使用できる引数の数は最高で31です(D)。
- 1つのマクロ宣言で使用できるパラメータの数は最高で31です(D)。
- 1つのマクロ呼び出しで使用できる引数の数は最高で31です(D)。
- 論理ソース行の文字数は最高で509です(1500)。
- 文字列リテラルまたは(連結後の)長文字列リテラルの文字数は最高で509です(1500)。
- **#include**ファイルのネスティングレベルは最高で8です(50)。
- switch文のcaseラベルの数は、ネストされているswitch文を除き、最高で257です(D)。
- 文字列構造体または共用体のメンバの数は最高で127です(D)。
- 1つの列挙内の列挙定数の数は最高で127です(D)。
- 単一のstruct宣言リスト内でネストできる構造体または共用体定義のレベル数は最高で15です(D)。

1.4.6 コンパイラのメッセージ

c88のメッセージには、ユーザエラー、ワーニング、システムエラーの3つのクラスがあります。

ユーザエラーメッセージの中には、通常のメッセージの後、コンパイラによって表示される余分な情報があります。このような余分な情報を含むメッセージには、Appendixのメッセージ一覧で"I"マークが付いています。これらのメッセージには、その前のエラーメッセージとエラー番号も付いています。情報メッセージの番号は、重要ではないため、この番号は表示されません。ユーザエラーは、致命的になるもの(Appendixのメッセージ一覧で"F"マークが付いている)もあります。このようなエラーが発生した場合、コンパイラは、エラーメッセージを表示した後、コンパイルを破棄しますが、場合によっては"完全でない"出力ファイルが生成されます。

エラー番号とワーニング番号は2つのグループに分けられます。コンパイラのフロントエンド部分は0から499の番号を使用し、コンパイラのバックエンド部分(コードジェネレータ)は、500以上の番号を使用します。ほとんどのエラーメッセージとワーニングメッセージは、フロントエンドが生成します。

プログラムに致命的でないエラーがある場合、**c88**は、エラーを含むCソース行を表示し、エラー番号とエラーメッセージを画面に表示します。フロントエンドが関数の終わりに到達した時点でコード生成が始まるため、コードジェネレータによってエラーが生成される場合、表示されるCソース行は、現在のC関数の最終行になります。ただしその場合、**c88**は、エラーメッセージの前に、エラーを引き起こした行の番号を表示します。**c88**は常に、エラーが発生したアセンブラ出力ファイルでエラー番号を生成します。

つまり、コンパイルが成功しなかった場合は、生成された出力ファイルがアセンブラによって受け入れられなかったため、破損したアプリケーションが生成されたことになります(**-e**オプションの説明も参照)。

ワーニングメッセージが表示された場合、それによってアセンブラ出力ファイルにエラーが生じるわけではありません。これらのメッセージは、コンパイラが必ずしも正しい状況になっていないことを知らせるために表示されます。ワーニングメッセージは、**-w**[*num*]オプションを付けることにより、コントロールすることができます。

メッセージの3つ目のクラスは、システムコンパイラエラーです。このエラーでは、次の書式が使用されます。

```
S number: internal error - please report
```

これらのエラーは、内部の整合性チェックの結果発生するもので、絶対に発生してはならないものです。それでも、このような"システム"エラーが発生した場合は、セイコーエプソン(株)に報告してください。その場合、エラーの原因になったCソースが含まれるフロッピーディスクまたはテープを同封してください。

1.4.7 戻り値

c88は、テストのため、終了ステータスをオペレーティングシステム環境に返します。

例：

MS-DOSのバッチファイルで、ERRORLEVELを使用して、実行したプログラムの終了ステータスを調べます。

```
c88 -s %1.c
IF ERRORLEVEL 1 GOTO STOP_BATCH
```

c88の終了ステータスは、次のリストのいずれかの番号になります。

終了ステータス：

0 コンパイルが成功しました。エラーはありません。

1 ユーザエラーが発生しましたが正常に終了しました。

2 致命的なエラーまたはシステムエラーが発生して、異常終了しました。

3 ユーザのアボートにより停止しました。

1.5 ライブラリ

この節では、コンパイラに付属するライブラリ関数について説明します。一部の関数(たとえば`printf()`、`scanf()`)は、必要性に応じて編集することができます。**c88**には、それぞれのメモリモデルに対応するオブジェクト形式のライブラリの他、ライブラリ関数の適切なプロトタイプを持つヘッダファイルが付属します。ライブラリ関数には、ソースコード(Cまたはアセンブラ)も付属します。

Cのさまざまな標準演算用に、インラインコードを生成するのは複雑すぎます(たとえば32ビット符号付き除算)。これらの演算は、ランタイムライブラリとして実装されています。

注: ライブラリの動作は保証されませんので、十分な評価を行った上、お客さまの責任においてご使用ください。

1.5.1 ヘッダファイル

Cコンパイラには、次のヘッダファイルが付属します。

<assert.h> 表明。

<c88.h> **c88**定義を持つ特殊ファイル。C関数はありません。標準Cコンパイラを使用するホストで、アプリケーションのプロトタイプを作成するときに使用できます。

<ctype.h> isalnum、isalpha、isascii、iscntrl、isdigit、isgraph、islower、isprint、ispunct、isspace、isupper、isxdigit、toascii、_tolower、tolower、_toupper、toupper。

<errno.h> エラー番号。C関数はありません。

<float.h> 浮動小数点演算に関連する定数。

<limits.h> int型の制限とサイズを定義しています。C関数はありません。

<locale.h> localeconv、setlocale。骨組みとして提供されています。

<math.h> acos、asin、atan、atan2、ceil、cos、cosh、exp、fabs、floor、fmod、frexp、ldexp、log、log10、modf、pow、sin、sinh、sqrt、tan、tanh。

<setjmp.h> longjmp、setjmp。

<signal.h> raise、signal。これらの関数は骨組みとして提供されています。

<stdarg.h> va_arg、va_end、va_start。

<stddef.h> offsetof、特殊型の定義。

<stdio.h> clearerr、fclose、_fclose、feof、ferror、fflush、fgetc、fgetpos、fgets、fopen、_fopen、fprintf、fputc、fputs、fread、freopen、fscanf、fseek、fsetpos、ftell、fwrite、getc、getchar、gets、_ioread、_iowrite、_lseek、perror、printf、putc、putchar、puts、_read、remove、rename、rewind、scanf、setbuf、setvbuf、sprintf、sscanf、tmpfile、tmpnam、ungetc、vfprintf、vprintf、vsprintf、_write。

<stdlib.h> abort、abs、atexit、atof、atoi、atol、bsearch、calloc、div、exit、free、getenv、labs、ldiv、malloc、mblen、mbstowcs、mbtowc、qsort、rand、realloc、srand、strtod、strtol、stroul、system、wcstombs、wctomb。

<string.h> memchr、memcmp、memcpy、memmove、memset、strcat、strchr、strcmp、strcol、strcpy、strcspn、strerror、strlen、strncat、strncmp、strncpy、strpbrk、strrchr、strspn、strstr、strtok、strxfrm。

<time.h> asctime、clock、ctime、difftime、gmtime、localtime、mktime、strftime、time。これらの関数はすべて骨組みとして提供されています。

1.5.2 Cライブラリ

Cライブラリには、Cライブラリ関数が含まれます。この節では、すべてのCライブラリ関数について説明しています。これらの関数は、アプリケーションプログラムで明示的に関数呼び出しを実行する場合のみ呼び出されます。

libディレクトリには、さまざまなプロセッサタイプに対応するサブディレクトリが含まれます。Cライブラリは、次のような名前の構文を使用します。

表1.5.2.1 Cライブラリ名の構文

コンパイラモデル	リンクするライブラリ
スモール(デフォルト)	libcs.a (デフォルト)
コンパクトコード	libcc.a
コンパクトデータ	libcd.a
ラージ	libcl.a

lk88リンカで**-l**オプションを指定すると、この命名規則が使用されます。たとえば、**-lcd**を付けると、リンカは、システムのlibディレクトリでlibcd.aを探します。ライブラリの指定は、コントロールプログラムの役割になります。

1.5.2.1 Cライブラリの実装の詳細

付属するCライブラリについて、以下のリストで詳細に説明しています。

Cライブラリルーチンによっては、プログラムから呼び出して使用する前に再コンパイルが必要なものもあります。このようなライブラリ関数は、余分なメモリを使用するため、デフォルトの状態ではアクティブになっていません。

説明：

Y - 完全に実装されています。

I - 実装されていますが、ユーザが低レベルのルーチンを記述しなければなりません。

R - 実装されていますが、再コンパイルが必要です。

L - 骨組みとして提供されています。

ファイル	実装の状態	ルーチン名	解説 / 理由
assert.h	Y	'assert()' macro	マクロ定義
ctype.h	Y		ルーチンのほとんどはマクロおよび関数になっています。 (ANSIの規定に準拠)
	Y	isalnum	
	Y	isalpha	
	Y	iscntrl	
	Y	isdigit	
	Y	isgraph	
	Y	islower	
	Y	isprint	
	Y	ispunct	
	Y	isspace	
	Y	isupper	
	Y	isxdigit	
	Y	tolower	
	Y	toupper	
	Y	_tolower	ANSIで定義されていません。
	Y	_toupper	ANSIで定義されていません。
	Y	isascii	ANSIで定義されていません。
	Y	toascii	ANSIで定義されていません。
errno.h	Y		マクロのみ
limits.h	Y		マクロのみ
locale.h	Y		
	L	localeconv	OSは規定されていません。
	L	setlocale	OSは規定されていません。

ファイル	実装の状態	ルーチン名	解説 / 理由
setjmp.h	Y		
	Y	longjmp	
	Y	setjmp	
signal.h	Y		
	L	raise	OSは規定されていません。
	L	signal	OSは規定されていません。
stdarg.h	Y		
	Y	va_arg	
	Y	va_end	
	Y	va_start	
stddef.h	Y		マクロのみ
stdio.h	Y		
	Y	clearerr	
	I	fclose	_fcloseが必要です。
	Y	feof	
	Y	ferror	
	I	fflush	
	I	fgetc	_write/_iowriteが必要です。
	I	fgetpos	_read/_ioreadが必要です。
	I	fgets	_lseekが必要です。
	I	fopen	_read/_ioreadが必要です。
	I	fprintf	_fopenが必要です。
	I	fputc	_write/_iowriteが必要です。
	I	fputs	_write/_iowriteが必要です。
	I	fread	_write/_iowriteが必要です。
	I	freopen	_read/_ioreadが必要です。
	I	fscanf	_fclose/_fopenが必要です。
	I	fseek	_read/_ioreadが必要です。
	I	fsetpos	_lseekが必要です。
	I	ftell	_lseekが必要です。
	I	fwrite	_lseekが必要です。
	I	getc	_write/_iowriteが必要です。
	I	getchar	_read/_ioreadが必要です。
	I	gets	_read/_ioreadが必要です。
	I	perror	_write/_iowriteが必要です。
	I	printf	_write/_iowriteが必要です。
	I	putc	_write/_iowriteが必要です。
	I	putchar	_write/_iowriteが必要です。
	I	puts	_write/_iowriteが必要です。
	L	remove	
	L	rename	
	I	rewind	_lseekが必要です。
	I	scanf	_read/_ioreadが必要です。
	Y	setbuf	
	Y	setvbuf	
	Y	sprintf	
	Y	sscanf	
	L	tmpfile	
	L	tmpnam	ランダムネームジェネレータとして提供されていますが、プロセスIDを使用しなければなりません。
	Y	ungetc	
	I	vfprintf	_write/_iowriteが必要です。
	I	vprintf	_write/_iowriteが必要です。
	Y	vsprintf	
	L	_fclose	低レベルファイルクローズルーチン。
	L	_fopen	低レベルファイルオープンルーチン。
	L	_ioread	低レベル入力ルーチン。
	L	_iowrite	低レベル出力ルーチン。
	L	_lseek	低レベルファイル位置調整ルーチン。
	L	_read	低レベルブロック入力ルーチン。 カスタマイズされていないときは_ioreadを使用します。
	L	_write	低レベルブロック書き込みルーチン。 カスタマイズされていないときは_iowriteを使用します。

ファイル	実装の状態	ルーチン名	解説 / 理由
stdlib.h	Y		
	Y	abort	cstartの_exit()を呼び出します。
	Y	abs	
	R	atexit	_exit()の再コンパイルが必要です。
	Y	atoi	
	Y	atol	
	Y	bsearch	
	Y	calloc	
	Y	div	
	Y	exit	cstartの_exit()を呼び出します。
	Y	free	
	L	getenv	OSは規定されていません。
	Y	labs	
	Y	ldiv	
	Y	malloc	
	Y	qsort	
	Y	strtod	
	Y	strtol	
	Y	strtoul	
	Y	rand	
	Y	realloc	
	Y	srand	
	L	system	OSは規定されていません。
	L	mblen	ワイドcharはサポートされていません。
	L	mbstowcs	ワイドcharはサポートされていません。
	L	mbtowc	ワイドcharはサポートされていません。
	L	wcstombs	ワイドcharはサポートされていません。
	L	wctomb	ワイドcharはサポートされていません。
string.h	Y		
	Y	memchr	
	Y	memcmp	
	Y	memcpy	
	Y	memmove	
	Y	memset	
	Y	strcat	
	Y	strchr	
	Y	strcmp	
	L	strcoll	ワイドcharはサポートされていません。
	Y	strcpy	
	Y	strcspn	
	Y	strerror	
	Y	strlen	
	Y	strncat	
	Y	strncmp	
	Y	strncpy	
	Y	strpbrk	
	Y	strrchr	
	Y	strspn	
	Y	strstr	
	Y	strtok	
	L	strxfrm	ワイドcharはサポートされていません。
time.h	Y		リアルタイムクロックはサポートされていません。
	L	asctime	
	L	clock	
	L	ctime	
	L	gmtime	
	L	localtime	
	L	mktime	
	L	strftime	
	L	time	

1.5.2.2 Cライブラリのインタフェースについての説明

_fclose

```
#include <stdio.h>
int _fclose( FILE *file );
```

低レベルファイルクローズ関数。_fcloseは、関数fcloseによって使用されます。指定されたストリームが正しくクローズされ、バッファがあればすべてフラッシュされます。

_flopen

```
#include <stdio.h>
int _fopen( const char, *file, FILE *iop );
```

低レベルファイルオープン関数。_fopenは、関数fopenおよびfreopenによって使用されます。指定されたストリームが正しくオープンされます。

_ioread

```
#include <stdio.h>
int _ioread( FILE *fp );
```

低レベル入力関数。提供されているライブラリには、"空の"関数が含まれています。実際の入出力を実行するためには、この関数をカスタマイズしなければなりません。_ioreadは、すべての入力関数(scanf、getc、getsなど)によって使用されます。

_iowrite

```
#include <stdio.h>
int _iowrite( int c, FILE *fp );
```

低レベル出力関数。提供されているライブラリには、"空の"関数が含まれています。実際の入出力を実行するためには、この関数をカスタマイズしなければなりません。_iowriteは、すべての出力関数(pritf、putc、putsなど)によって使用されます。

_lseek

```
#include <stdio.h>
long _lseek( FILE *iop, long offset, int origin );
```

低レベルファイル位置調整関数。_lseekは、すべてのファイル位置調整関数(fgetpos、fseek、fsetpos、ftell、rewindなど)によって使用されます。

_read

```
#include <stdio.h>
size_t _read( FILE *fin, char *base, size_t size );
```

低レベルブロック入力関数。この関数は、使用する前にカスタマイズしなければなりません。カスタマイズされていない場合、この関数は_ioreadを使用します。この関数は、すべての入力関数によって使用されるもので、指定されたストリームから文字のブロックを読み込みます。

戻り値　読み込まれた文字の数。

_tolower

```
#include <ctype.h>
int _tolower( int c );
```

cを小文字に変換します。ただし、cが実際に大文字になっているかどうかはチェックしません。これは非ANSI関数です。

戻り値　変換された文字。

_toupper

```
#include <ctype.h>
int _toupper( int c );
```

cを大文字に変換します。ただし、cが実際に小文字になっているかどうかはチェックしません。これは非ANSI関数です。

戻り値 変換された文字。

_write

```
#include <stdio.h>
size_t _write( FILE *iop, char *base, size_t size );
```

低レベルブロック出力関数。この関数は、使用する前にカスタマイズしなければなりません。カスタマイズされていない場合、この関数は_iowriteを使用します。この関数は、すべての出力関数によって使用されるもので、指定されたストリームに文字のブロックを書き込みます。

戻り値 正常に書き込まれた文字の数。

abort

```
#include <stdlib.h>
void abort( void );
```

プログラムを異常終了させます。スタートアップモジュールで定義されている関数_exitを呼び出します。

戻り値 なし。

abs

```
#include <stdlib.h>
int abs( int n );
```

戻り値 signed int引数の絶対値。

asctime

```
#include <time.h>
char *asctime ( const struct tm *tp );
```

構造体*tpの時間を、以下の形式の文字列に変換します。

```
Mon Jan 21 16:15:14 1989¥n¥0
```

戻り値 文字列形式の時間。

assert

```
#include <assert.h>
void assert( int expr );
```

NDEBUGを付けてコンパイルした場合、これは空のマクロになります。NDEBUGを定義しないでコンパイルした場合、exprが真かどうかチェックします。真の場合、次のような行がプリントされます。

"Assertion failed: *expression*, file *filename*, line *num*"

戻り値 なし。

atexit

```
#include <stdlib.h>
int atexit( void (*fcn)( void) );
```

プログラムが正常に終了するときに、関数fcnが呼び出されるよう登録します。

戻り値 プログラムが正常に終了したとき、0。
登録ができない場合、0以外の値。

atoi

```
#include <stdlib.h>
int atoi( const char *s );
```

指定された文字列をint値に変換します。スペースは無視されます。変換は、認識できない文字が見つかった段階で停止します。

戻り値　int値。

atol

```
#include <stdlib.h>
long atol( const char *s );
```

指定された文字列をlong値に変換します。スペースは無視されます。変換は、認識できない文字が見つかった段階で停止します。

戻り値　long値。

bsearch

```
#include <stdlib.h>
void *bsearch( const void *key, const void *base, size_t n, size_t size,
  int (* cmp) (const void *, const void *) );
```

この関数は、nで個のメンバーの配列内で、ptrでポイントされているオブジェクトを検索します。配列の最初の基底は、baseで指定されます。それぞれのメンバーのサイズは、sizeで指定されます。指定する配列は、cmpがポイントする関数の結果に従って、昇順にソートされなければなりません。

戻り値　配列内で一致するメンバーを示すポインタ。見つからなかった場合はNULL。

calloc

```
#include <stdlib.h>
void *calloc( size_t nobj, size_t size );
```

割り当てられた空間が0で埋められます。割り当てられる最大空間は、ヒープサイズをカスタマイズすることで変更できます("1.3.5 ヒープ"を参照)。デフォルトの場合、ヒープは割り当てられません。ヒープが定義されていないときに"calloc()"が使用されると、ロケータがエラーを出します。

戻り値　sizeバイトの長さのnobj個の項目に割り当てた外部メモリの空間を示すポインタ。十分な空間が残っていない場合はNULL。

clearerr

```
#include <stdio.h>
void clearerr( FILE *stream );
```

ファイルの終了とストリームのエラーインジケータをクリアします。

戻り値　なし。

clock

```
#include <time.h>
clock_t clock( void );
```

使用されたプロセッサ時間を判別します。

戻り値　1

ctime

```
#include <time.h>
char *ctime( const time_t :tp );
```

カレンダー時間*tpを、ローカル時刻に文字列形式で変換します。この関数は、次の関数と同じです。

```
asctime( localtime( tp ) );
```

戻り値　文字列形式のローカル時刻。

div

```
#include <stdlib.h>
div_t div( int num, int denom );
```

両方の引数が整数値になります。返される商と余りも整数値になります。

戻り値　numをdenomで割ったときの商と余りを含む構造体。

exit

```
#include <stdlib.h>
void exit( int status );
```

プログラムを正常終了させます。"main( )"が戻り値としてstatusを返す状況と同様の動作をします。

戻り値　正常終了した場合、0。

fclose

```
#include <stdio.h>
int fclose( FILE *stream );
```

streamに書き込まれていないデータがあればフラッシュして、読み込まれていないバッファ入力を廃棄し、自動的に割り当てられていないバッファがあればそれを解放して、最後にstreamをクローズします。

戻り値　streamが正しくクローズできれば0、エラーが発生すればEOF。

feof

```
#include <stdio.h>
int feof( FILE *stream );
```

戻り値　streamのファイル終了(EOF)インジケータがセットされていれば、0以外の値。

ferror

```
#include <stdio.h>
int ferror( FILE *stream );
```

戻り値　streamのエラーインジケータがセットされていれば、0以外の値。

fflush

```
#include <stdio.h>
int fflush( FILE *stream );
```

streamが出力ストリームの場合、バッファされていて記述されていないデータがあれば、それを書き込みます。streamが入力ストリームの場合、その影響は定義されていません。

戻り値　正常に完了すれば0、書き込みエラーの場合はEOF。

fgetc

```
#include <stdio.h>
int fgetc( FILE *stream );
```

指定されたストリームから1文字読み込みます。

戻り値　読み込んだ文字。エラーの場合はEOF。

fgetpos

```
#include <stdio.h>
int fgetpos( FILE *stream, fpos_t *ptr );
```

streamによってポイントされているストリームのファイル位置インジケータの現在の値を、ptrによってポイントされているオブジェクトに格納します。このような値を記録するには、fpos_t型が適しています。

戻り値　正常に完了すれば0、エラーの場合は0以外の値。

fgets

```
#include <stdio.h>
char *fgets( char *s, int n, FILE *stream );
```

指定されたストリームから配列sへ、それ以降の文字を最大n-1個読み込みます。改行文字が現れた時点で読み込みを終了します。

戻り値　s。EOFまたはエラーの場合はNULL。

fopen

```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
```

指定されたmodeでファイルをオープンします。

戻り値　ストリーム。ファイルがオープンできない場合、NULLが返されます。

modeには、以下の値を指定することができます。

"r"　読み込み。テキストファイルを読み込みモードでオープンします。

"w"　書き込み。テキストファイルを書き込みモードで作成します。指定されたファイルがすでに存在している場合、その内容が破棄されます。

"a"　追加。既存のテキストファイルをオープンするか、新しいテキストファイルを作成し、ファイル終了地点から書き込みを始めます。

"r+"　テキストファイルをオープンし更新します。つまり読み込みと書き込みを行います。

"w+"　テキストファイルを作成し更新します。同じ名前のファイルがすでにあれば破棄されます。

"a+"　追加。テキストファイルをオープンまたは作成し更新します。ファイル終了地点から書き込みを始めます。

更新モード("+")では、同じファイルに対して読み込みと書き込みを行います。このモードでは、読み込みしてから書き込みするまでの間または書き込みしてから読み込みするまでの間に、関数fflushが呼び出されます。最初の文字のあとに文字bを入れることで、ファイルがバイナリファイルであることを指定できます。たとえば、"rb"とするとバイナリの読み込みになり、"w+b"とするとバイナリファイルを作成して更新することになります。ファイル名は、FILENAME_MAX個の文字数に制限されます。また、最大FOPEN_MAX個のファイルを同時にオープンすることができます。

fprintf

```
#include <stdio.h>
int fprintf( FILE *stream, const char *format, ... );
```

指定されたストリームに、一定の書式で書き込みを行います。"printf()"、"_iowrite()"の説明の他、"1.5.2.3 printfおよびscanfの書式化ルーチン"も参照してください。

fputc

```
#include <stdio.h>
int fputc( int c, FILE *stream );
```

指定されたストリームに、1文字追加します。"_iowrite()"の説明も参照してください。

戻り値　エラーの場合はEOF。

fputs

```
#include <stdio.h>
int fputs( const char *s, FILE *stream );
```

指定されたストリームに、文字列を書き込みます。最後のNULL文字は書き込まれません。"_iowrite()"の説明も参照してください。

戻り値　正常に完了すればNULL、エラーの場合はEOF。

fread

```
#include <stdio.h>
size_t fread( void *ptr, size_t size, size_t nobj, FILE *stream );
```

指定されたストリームから、sizeバイトのnobjメンバーを読み込み、ptrでポイントされている配列に入れます。"_ioread()"の説明も参照してください。

戻り値　正常に読み込まれたオブジェクトの数。

free

```
#include <stdlib.h>
void free( void *p );
```

pでポイントされた空間の割り当てを解除します。ただしpは、"calloc()"、"malloc()"、"realloc()"を呼び出して割り当てている空間をポイントしなければなりません。正しくポイントされていない場合、動作は予期できないものになります。"calloc()"、"malloc()"、"realloc()"の説明も参照してください。

戻り値　なし。

freopen

```
#include <stdio.h>
FILE *freopen( const char *filename, const char *mode, FILE *stream );
```

指定されたmodeでファイルをオープンし、streamをそれに対応させます。この関数は、通常、stdin、stdout、stderrに対応するファイルを変更するときに使用されます。"fopen()"の説明も参照してください。

戻り値　stream。エラーの場合はNULL。

fscanf

```
#include <stdio.h>
int fscanf( FILE *stream, const char *format, ... );
```

指定されたストリームから、一定の書式で読み込みを行います。"scanf()"、"_ioread()"の説明の他、"1.5.2.3 printfおよびscanfの書式化ルーチン"も参照してください。

戻り値　正常に変換された項目の数。

fseek

```
#include <stdio.h>
int fseek( FILE *stream, long offset, int origin );
```

streamでファイル位置インジケータをセットします。それ以降の読み込みまたは書き込みでは、新しい位置で開始するデータにアクセスします。バイナリファイルの場合、位置はoriginからoffset文字の位置にセットされます。originは、ファイルの最初の場合SEEK_SET、ファイルの現在位置の場合SEEK_CUR、ファイル終了の場合SEEK_ENDになります。テキストストリームの場合、offsetは0、またはftellで返される値になります。その場合、originはSEEK_SETになります。

戻り値　正常に完了すれば0、エラーの場合は0以外の値。

fsetpos

```
#include <stdio.h>
int fsetpos( FILE *stream, const fpos_t *ptr );
```

streamを、*ptr内のfgetposで記録されている位置に配置します。

戻り値　正常に完了すれば0、エラーの場合は0以外の値。

ftell

```
#include <stdio.h>
long ftell( FILE *stream );
```

戻り値　streamの現在のファイル位置。エラーの場合は-1L。

fwrite

```
#include <stdio.h>
size_t fwrite( const void *ptr, size_t size, size_t nobj, FILE *stream );
```

ptrでポイントされている配列から、指定されたストリームにsizeバイトのnobjメンバーを書き込みます。

戻り値　正常に書き込まれたオブジェクトの数。

getc

```
#include <stdio.h>
int getc( FILE *stream );
```

指定されたストリームから1文字読み込みます。現在、FILE I/Oがサポートされていないため、getchar()として#defineされています。"_ioread()"の説明も参照してください。

戻り値　読み込まれた文字。エラーの場合はEOF。

getchar

```
#include <stdio.h>
int getchar( void );
```

標準入力から1文字読み込みます。"_ioread()"の説明も参照してください。

戻り値　読み込まれた文字。エラーの場合はEOF。

getenv

```
#include <stdlib.h>
char *getenv( const char *name );
```

戻り値　nameに対応する環境文字列。文字列が存在しない場合はNULL。

gets

```
#include <stdio.h>
char *gets( char *s );
```

改行文字が現れるまで、標準入力からすべての文字読み込みます。改行はNULL文字で置き換えられます。"_ioread()"の説明も参照してください。

戻り値　読み込まれた文字列に対するポインタ。エラーの場合はNULL。

gmtime

```
#include <time.h>
struct tm *gmtime( const time_t *tp );
```

カレンダー時間*tpを協定世界時(UTC)に変換します。

戻り値　UTCを表す構造体。UTCが使用できない場合はNULL。

isalnum

```
#include <ctype.h>
int isalnum( int c );
```

戻り値　cが英数字([A–Z][a–z][0–9])の場合、0以外の値。

isalpha

```
#include <ctype.h>
int isalpha( int c );
```

戻り値　cが英字([A–Z][a–z])の場合、0以外の値。

isascii

```
#include <ctype.h>
int isascii( int c );
```

戻り値 cが0から127の範囲の場合、0以外の値。これは非ANSI関数です。

iscntrl

```
#include <ctype.h>
int iscntrl( int c );
```

戻り値 cがコントロール文字の場合、0以外の値。

isdigit

```
#include <ctype.h>
int isdigit( int c );
```

戻り値 cが数字([0–9])の場合、0以外の値。

isgraph

```
#include <ctype.h>
int isgraph( int c );
```

戻り値 cがスペース以外のプリント可能な文字の場合、0以外の値。

islower

```
#include <ctype.h>
int islower( int c );
```

戻り値 cが小文字([a–z])の場合、0以外の値。

isprint

```
#include <ctype.h>
int isprint( int c );
```

戻り値 cがスペースを含むプリント可能な文字の場合、0以外の値。

ispunct

```
#include <ctype.h>
int ispunct( int c );
```

戻り値 cが句読文字("."、","、"!"など)の場合、0以外の値。

isspace

```
#include <ctype.h>
int isspace( int c );
```

戻り値 cが、スペース類の文字(スペース、タブ、垂直タブ、改ページ、改行文字、復帰文字)の場合、0以外の値。

isupper

```
#include <ctype.h>
int isupper( int c );
```

戻り値 cが大文字([A–Z])の場合、0以外の値。

isxdigit

```
#include <ctype.h>
int isxdigit( int c );
```

戻り値 cが16進数([0–9][A–F][a–f])の場合、0以外の値。

labs

```
#include <stdlib.h>
long labs( long n );
```

戻り値　signed long引数の絶対値。

ldiv

```
#include <stdlib.h>
ldiv_t ldiv( long num, long denom );
```

両方の引数がlong整数値になります。返される商と余りもlong整数値になります。

戻り値　numをdenomで割ったときの商と余りを含む構造体。

localeconv

```
#include <locale.h>
struct lconv *localeconv( void );
```

struct lconv型を持つオブジェクトの構成要素に、現在のロケールの規則に従って数量を書式化するうえで適した値をセットします。

戻り値　埋め込まれたオブジェクトを示すポインタ。

localtime

```
#include <time.h>
struct tm *localtime( const time_t *tp );
```

カレンダー時間*tpをローカル時刻に変換します。

戻り値　ローカル時刻を表す構造体。

longjmp

```
#include <setjmp.h>
void longjmp( jmp_buf env. int val );
```

setjmp( )を呼び出して前に保存した環境をリストアします。対応するsetjmp( )呼び出しを呼び出した関数が、まだ終了していない可能性があります。またvalの値は、0にすることができません。

malloc

```
#include <stdlib.h>
void *malloc( size_t size );
```

割り当てられた空間が初期化されません。割り当てられる最大空間は、ヒープサイズをカスタマイズすることで変更できます("1.3.5 ヒープ"を参照)。デフォルトの場合、ヒープは割り当てられません。ヒープが定義されていないときに"malloc()"が使用されると、ロケータがエラーを出します。スモールメモリモデルまたはコンパクトデータメモリモデルで"malloc()"を使用する場合は、ロケータ記述ファイルでヒープをアドレッシングモード"データ"からアドレッシングモード"データショート"に移す必要があります。これを怠ると、アプリケーションの結果をロケートするときに、ロケートエラーが出ます。

戻り値　外部メモリにあるsizeバイトの長さの空間を示すポインタ。十分な空間がない場合はNULLが返されます。

mblen

```
#include <stdlib.h>
int mblen( const char *s, size_t n );
```

sがNULLポインタでない場合、sがポイントする2バイトコード文字の構成バイト数を判別します。ただしシフト状態は影響を受けません。sがポイントする文字から最大でn文字をチェックします。

戻り値　バイト数。ただしsがNULL文字をポイントしている場合0になり、各バイトが正しい2バイトコード文字を構成していない場合1になります。

mbstowcs

```
#include <stdlib.h>
size_t mbstowcs( wchar_t *pwcs, const char *s, size_t n );
```

sがポイントする配列から初期シフト状態が始まる一連の2バイトコード文字を、対応するコードに変換し、これらのコードをpwcsがポイントする配列に格納します。n個のコードを格納した時点、または0の値を持つコードを格納した時点で停止します。

戻り値　変更された配列要素の数(終了0コードがある場合でも、これは含まれない)。ただし無効な2バイトコード文字が見つかった場合は(size_t)-1。

mbtowc

```
#include <stdlib.h>
int mbtowc( wchar_t *pwc, const char *s, size_t n );
```

sがポイントする2バイトコード文字の構成バイト数を判別します。その後、その2バイトコード文字に対応する、型wchar_tの値に対するコードを判別します。2バイトコード文字が有効でpwcがNULLポインタでない場合、mbtowc関数は、pwcがポイントするオブジェクトにコードを格納します。sがポイントする文字から最大でn文字をチェックします。

戻り値　バイト数。ただしsがNULL文字をポイントしている場合0になり、各バイトが正しい2バイトコード文字を構成していない場合1になります。

memchr

```
#include <string.h>
void *memchr( const void *cs, int c, size_t n );
```

文字cが見つかったときにcsの最初のnバイトをチェックします。

戻り値　見つからない場合はNULLが、見つかった場合は見つかった文字を示すポインタが返されます。

memcmp

```
#include <string.h>
int memcmp( const void *cs, const char *ct, size_t n );
```

csの最初のnバイトをctの内容と比較します。

戻り値　cs<ctの場合負の値、
cs==ctの場合0、
cs>ctの場合正の値。

memcpy

```
#include <string.h>
void *memcpy( void *s, const void *ct, size_t n );
```

ctからsにnバイトの文字をコピーします。ただし、2つのオブジェクトが重ね書きされることに対しては何の対策もとられません。

戻り値　s

memmove

```
#include <string.h>
void *memmove( void *s, const void *ct, size_t n );
```

ctからsにn個の文字をコピーします。ただし、2つのオブジェクトが重ね書きされることに対しては何の対策もとられません。

戻り値　s

memset

```
#include <string.h>
void *memset( void *s, int c, size_t n );
```

sの最初のnバイトを文字cで埋めます。

戻り値　s

mktime

```
#include <time.h>
time_t mktime( struct tm *tp );
```

構造体*tpのローカル時刻をカレンダー時間に変換します。

戻り値　カレンダー時間。カレンダー時間を表すことができない場合-1。

offsetof

```
#include <stddef.h>
int offsetof( type, member );
```

戻り値　typeのオブジェクトにあるmemberのオフセット。

perror

```
#include <stdio.h>
void perror( const char *s );
```

sの他、整数errnoに対応する実装定義のエラーメッセージをプリントします。つまり、次の関数と同じ結果になります。

fprintf(stderr, %s: %s\n, s, "*error message*")

エラーメッセージの内容は、strerror関数にerrno引数を付けた場合に返されるものと同じです。"strerror()"関数の説明も参照してください。

戻り値　なし。

printf

```
#include <stdio.h>
int printf( const char *format,...);
```

標準出力ストリームに、一定の書式で書き込みを行います。"_iowrite()"の説明および"1.5.2.3 printfおよびscanfの書式化ルーチン"も参照してください。

戻り値　出力ストリームに書き込まれた文字の数。

format文字列には、変換指定子とプレーンテキストを組み合わせたものが入ります。それぞれの変換指定子の前には"%"を付けます。変換指定子は以下の順序で構成されます。

- フラグ(任意の順序)：

-	変換された引数を左詰めで出力します。
+	符号文字の前には常に数値が付きます。+はスペースよりも優先度が高くなります。
スペース	負の数の前には符号が付き、正の数の前にはスペースが付きます。
0	フィールド幅に0を詰め込むよう指定します(数値の場合のみ)。
#	別の出力形式を指定します。oの場合、最初の桁が0になります。xまたはXの場合、"0x"と"0X"が数値の前に付きます。e、E、f、g、Gの場合、出力に常に小数点が付くようになり、最後の0も削除されなくなります。

- 最小フィールド長を指定する数値。変換された引数は、ここで指定された以上の長さを持つフィールドにプリントされます。変換された引数が、指定されている文字数より短い場合、その左側(フラグ"-"が指定されているときは右側)がスペースで埋められます。フラグに"0"が指定されている場合も、数値フィールドは0で埋められます(左側に埋め込まれる場合のみ)。数値の代わりに"*"を指定することもできます。その場合、値はその次の引数から取られます。ただしこの引数はint型と見なされます。

- ピリオド。最小フィールド幅と精度を区切ります。
- プリントする文字列の最大長を指定する数値。浮動小数点変換の場合は、小数点のあとにプリントされる桁数を指定します。整数変換の場合は、プリントされる最小の桁数を指定します。ここでも数値の代わりに"*"を指定することもできます。その場合、値はその次の引数から取られます。この引数もint型と見なされます。
- 長さの変更子、"h"、"l"、"L"。"h"を指定すると、引数がショートまたは符号なしショートの数値と見なされます。"l"は、引数がロングの整数の場合に使用します。"L"は、引数がロングの倍精度であることを指定します。

フラグ、長さの指定子、ピリオド、精度と長さの変更子はどれもオプションですが、変換文字はオプションではありません。変換文字は、以下のいずれかでなければなりません。このリストにない文字が"%"のあとに付けられた場合、動作は予期できないものになります。

文字	プリントされる文字の属性
d, i	int型の符号付き10進数。
o	int型の符号なし8進数。
x, X	小文字、大文字の場合、それぞれint型の符号なし16進数。
u	int型の符号なし10進数。
c	int型の単一の文字(unsigned charに変換)。
s	char *。この文字列の文字は、NULL文字が見つかるまでプリントされます。精度で指定されていれば、プリントも停止します。
f	double。
e, E	double。
g, G	double。
n	int *。書き込まれる文字の数が、引数に入れられます。これは、デフォルトメモリ内の整数を示すポインタになります。値はプリントされません。
p	ポインタ(24ビットの16進値)。
%	引数が変換されず、"%"のみがプリントされます。

putc

```
#include <stdio.h>
int putc( int c, FILE *stream );
```

指定されたストリームに1文字だけ書き込みます。"_iowrite()"の説明も参照してください。

戻り値　エラーの場合はEOF。

putchar

```
#include <stdio.h>
int putchar( int c );
```

標準出力に1文字だけ書き込みます。"_iowrite()"の説明も参照してください。

戻り値　書き込まれた文字。エラーの場合はEOF。

puts

```
#include <stdio.h>
int puts( const char *s );
```

文字列をstdoutに書き込みます。文字列は改行で終了します。"_iowrite()"の説明も参照してください。

戻り値　正常に完了すればNULL、エラーの場合はEOF。

qsort

```
#include <stdlib.h>
void qsort( const void *base, size_t n, size_t size,
  int (* cmp)(const void *, const void *) );
```

この関数は、n個のメンバーの配列をソートします。配列の最初の基底は、baseで指定されます。それぞれのメンバーのサイズは、sizeで指定されます。指定された配列は、cmpがポイントする関数の結果に従って、昇順にソートされます。

raise

```
#include <signal.h>
int raise( int sig );
```

シグナルsigをプログラムに送信します。"signal()"の説明も参照してください。

戻り値 正常に完了すれば0、異常終了の場合は0以外の値。

rand

```
#include <stdlib.h>
int rand( void );
```

戻り値 0からRAND_MAXの範囲内にある、疑似乱数(整数)のシーケンス。

realloc

```
#include <stdlib.h>
void *realloc( void *p, size_t size );
```

pがポイントするオブジェクトに空間を割り当て直します。オブジェクトの内容は、realloc()を呼び出す前と同じになります。割り当てられる最大空間は、ヒープサイズをカスタマイズすることで変更できます("1.3.5 ヒープ"を参照)。デフォルトの場合、ヒープは割り当てられません。ヒープが定義されていないときに"realloc()"が使用されると、リンカがエラーを出します。"malloc()"の説明も参照してください。

戻り値 新しい割り当てに対して十分な空間がない場合、NULLが返され、*pは変更されません。それ以外の場合、そのオブジェクトに新しく割り当てられた空間を示すポインタが返されます。

remove

```
#include <stdio.h>
int remove( const char *filename );
```

指定されたファイルを削除します。その結果、それ以降そのファイルをオープンしようとしてもエラーが発生します。

戻り値 ファイルが正常に削除できれば0、失敗すれば0以外の値が返されます。

rename

```
#include <stdio.h>
int rename( const char *oldname, const char *newname );
```

ファイルの名前を変更します。

戻り値 ファイルの名前が正常に変更できれば0、失敗すれば0以外の値が返されます。

rewind

```
#include <stdio.h>
void rewind( FILE *stream );
```

streamでポイントされたストリームについて、ファイル位置インジケータをファイルの最初にセットします。この関数は、次のスクリプトと同じ結果になります。

```
(void) fseek( stream, 0L, SEEK_SET );
clearerr( stream );
```

戻り値 なし。

scanf

```
#include <stdio.h>
int scanf( const char *format, ...);
```

標準入力ストリームから、一定の書式で読み込みを行います。"_ioread()"の説明および"1.5.2.3 printfおよびscanfの書式化ルーチン"も参照してください。

戻り値 正常に変換された項目の数。

この関数のすべての引数は、フォーマット文字列で指定されている型の(デフォルトメモリ内の)変数に対するポインタでなければなりません。

フォーマット文字列には、以下のものが入ります。

- ブランクまたはタブ。どちらもスキップされます。
- 通常の文字("%"を除く)。この文字列は、入力ストリームと正確に一致しなければなりません。
- 変換指定子。最初に"%"が付きます。

変換指定子は、以下の順序で構成されます。

- "*"。このフィールドに割り当てが行われないことを示します。
- 最大フィールド幅を示す数値。
- 引数が、intでなくshortを示すポインタの場合、変換文字d、i、n、o、u、xの前に"h"が付き、引数が、longを示すポインタの場合、"l"(エル)が付きます。ポインタfloatではなくポインタdoubleが引数リストに入っている場合、変換文字e、f、gの前に"l"が付き、ポインタがlong doubleを示す場合、"L"が付きます。
- 変換指定子。"*"、最大フィールド幅、長さの変更子はどれもオプションですが、変換文字はオプションではありません。変換文字は、以下のいずれかでなければなりません。このリストにない文字が"%"のあとに付けられた場合、動作は予期できないものになります。

長さの指定子と長さの変更子はどちらもオプションですが、変換文字はオプションではありません。変換文字は、以下のいずれかでなければなりません。このリストにない文字が"%"のあとに付けられた場合、動作は予期できないものになります。

文字	読み込まれる文字の属性
d	int型の符号付き10進数。
i	int型。整数は8進数(最初に0が入ります)、16進数(最初に"0x"または"0X"が入ります)、10進数のいずれでも指定できます。
o	int型の符号なし8進数。
u	int型の符号なし10進数。
x	小文字でも大文字でもint型の符号なし16進数。
c	単一の文字(unsigned charに変換)。
s	char *。スペース以外の文字による文字列。引数は、文字列と最後のNULL文字が入るだけの大きさを持つ、文字の配列をポイントしなければなりません。
f	float。
e, E	float。
g, G	float。
n	int *。書き込まれる文字の数が、引数に入れられます。スキャンは行われません。
p	ポインタ(24ビットの16進値)。
[...]	かっこ内の入力文字で構成される文字列と一致します。この文字列の最後には、NULL文字が追加されます。[]...]のように指定すると、"]"文字がスキャン文字のセットに入れられます。
[^...]	かっこ内の入力文字セット以外で構成される文字列と一致します。この文字列の最後には、NULL文字が追加されます。[^]...]のように指定すると、"]"文字がスキャン文字除外のセットに入れられます。
%	リテラル"%"。割り当てが行われません。

setbuf

```
#include <stdio.h>
void setbuf ( FILE *stream, char *buf );
```

bufがNULLの場合、streamでバッファリングがオフになります。それ以外の場合、setbuf関数は、次の関数と同じ結果になります。

```
(void) setvbuf( stream, buf, _IOFBF, BUFSIZ );
```

"setvbuf()"の説明も参照してください。

setjmp

```
#include <setjmp.h>
int setjmp( jmp_buf env );
```

これ以降longjmpを呼び出すときのために、現在の環境を保存します。

戻り値 setjmp( )を直接呼び出した後では、値0。保存されたenvを使用して関数"longjmp( )"を呼び出した場合、現在の環境がリストアされて、元の場所にジャンプします。その場合、0以外の戻り値になります。

"longjmp( )"の説明も参照してください。

setlocale

```
#include <local.h>
char *setlocale( int category, const char *locale );
```

category引数とlocale引数で指定されたプログラムのロケールの適切な部分を選択します。

戻り値 正しく選択できた場合、新しいロケールの指定のcategoryに対応する文字列。正しく選択できない場合、NULLポインタ。

setvbuf

```
#include <stdio.h>
int setvbuf( FILE *stream, char *buf, int mode, size_t size );
```

streamのバッファリングを制御します。この関数は、読み込みや書き込みの前に呼び出さなければなりません。modeには、以下の値が入ります。

_IOFBF	完全バッファリングを実行します。
_IOLBF	テキストファイルの行バッファリングを実行します。
_IONBF	バッファリングを実行しません。

bufがNULLでない場合、そのままバッファとして使用されます。NULLの場合、バッファが割り当てられます。sizeはバッファサイズを指定します。

戻り値 正常に完了すれば0、エラーの場合は0以外の値。

"setbuf()"の説明も参照してください。

signal

```
#include <signal.h>
void (*signal( int sig, void (*handler)(int)))(int);
```

以降のシグナルをどのように処理するか決定します。handlerがSIG_DFLの場合、デフォルトの動作が使用されます。handlerがSIG_IGNの場合、シグナルは無視されます。それ以外の場合、handlerがポイントする関数に、シグナルの型の引数が付けられて呼び出されます。有効なシグナルには、次のようなものがあります。

SIGABRT	abortなどによる異常終了。
SIGFPE	0による除算やオーバーフローなどによる算術エラー。
SIGILL	不正な命令などによる不正な関数イメージ。
SIGINT	割り込みなどの、対話的な注意。
SIGSEGV	メモリ制限外へのアクセスなどの、不正な格納アクセス。
SIGTERM	このプログラムに送られる終了要求。

それ以降にシグナルｓｉｇが現れると、シグナルはデフォルトの動作に戻されます。そのあと、(*handler)(sig)を実行した場合と同じようにシグナルハンドラ関数が呼び出されます。ハンドラが返されると、シグナルが発生した時点まで実行が戻されます。

戻り値 特定のシグナルに対するhandlerの前の値。エラーの場合はSIG_ERR。

sprintf

```
#include <stdio.h>
int sprintf( char *s, const char *format,... );
```

文字列に、一定の書式で書き込みを行います。"printf()"の説明および"1.5.2.3 printfおよびscanfの書式化ルーチン"も参照してください。

srand

```
#include <stdlib.h>
void srand( unsigned int seed );
```

この関数は、それ以降srand( )を呼び出すときに返される疑似乱数の新しい順序の開始としてseedを使用します。同じシード値でsrandが呼び出されるとき、rand( )が生成する疑似乱数の順序が繰り返されます。

戻り値 なし。

sscanf

```
#include <stdio.h>
int sscanf( char *s, const char *format, ... );
```

文字列から、一定の書式で読み込みを行います。"scanf()"の説明および"1.5.2.3 printfおよびscanfの書式化ルーチン"も参照してください。

strcat

```
#include <string.h>
char *strcat( char *s, const char *ct );
```

文字列ctを文字列sに連結します。この場合、最後のNULL文字も入れられます。

戻り値 s

strchr

```
#include <string.h>
char *strchr( const char *cs, int c );
```

戻り値 文字列csにある文字cが現れたときの、その文字を示すポインタ。文字が見つからない場合、NULLが返されます。

strcmp

```
#include <string.h>
int strcmp( const char *cs, const char *ct );
```

文字列csを文字列ctと比較します。

戻り値 cs<ctの場合負の値、
cs==ctの場合0、
cs>ctの場合正の値。

strcoll

```
#include <string.h>
int strcoll( const char *cs, const char *ct );
```

文字列csを文字列ctと比較します。比較は、プログラムのロケールに適していると見なされる文字列に基づいて行われます。

戻り値 cs<ctの場合負の値、
cs==ctの場合0、
cs>ctの場合正の値。

strcpy

```
#include <string.h>
char *strcpy( char *s, const char *ct );
```

文字列ctを文字列sにコピーします。この場合、最後のNULL文字もそのまま入れられます。

戻り値　s

strcspn

```
#include <string.h>
size_t strcspn( const char *cs, const char *ct );
```

戻り値　文字列ctにない文字で構成される、文字列csの接頭辞の長さ。

strerror

```
#include <string.h>
char *strerror( size_t n );
```

戻り値　エラーnに対応する実装定義文字列を示すポインタ。

strftime

```
#include <time.h>
size_t strftime( char *s, size_t smax, const char *fmt, const struct tm *tp );
```

指定された書式fmtに従って構造体*tpの日付と時刻の情報を書式化し、sに入れます。fmtはprintfの書式と似ています。それぞれの%*c*は、以下のように置き換えられます。

%a　曜日の省略形。
%A　曜日の完全名。
%b　月の省略形。
%B　月の完全名。
%c　地域ごとの日付と時刻の表現。
%d　日(01～31)。
%H　時(00～23)。
%I　時(01～12)。
%j　その年の1月1日からの日数(001～366)。
%m　月(01～12)。
%M　分(00～59)。
%p　AMまたはPMの地域ごとの表現。
%S　秒(00～59)。
%U　その年の1月1日からの週数(00～53)。日曜日が週の始めとして計算されます。
%w　曜日(0～6)。
%W　その年の1月1日からの週数(00～53)。月曜日が週の始めとして計算されます。
%x　地域ごとの日の表現。
%X　地域ごとの時刻の表現。
%y　世紀を取った西暦(00～99)。
%Y　世紀が付いた西暦。
%Z　時間帯がある場合、その名前。
%%　%

通常の文字(最後に付く"\0"を含む)はsにコピーされます。smax以上の文字はsに入りません。

戻り値　文字数("\0"は含まない)。smax個以上の文字が生成された場合0。

strlen

```
#include <string.h>
size_t strlen( const char *cs );
```

戻り値　csに入っている文字列の長さ。ただしNULL文字は除外されます。

strncat

```
#include <string.h>
char *strncat( char *s, const char *ct, size_t n );
```

文字列ctを文字列sに連結します。ただしコピーされるのは最大でn文字です。この場合、最後のNULL文字も追加されます。

戻り値　s

strncmp

```
#include <string.h>
int strncmp( const char *cs, const char *ct, size_t n );
```

文字列csのうち最大nバイトが文字列ctと比較されます。

戻り値　cs<ctの場合負の値、
cs==ctの場合0、
cs>ctの場合正の値。

strncpy

```
#include <string.h>
char *strncpy( char *s, const char *st, size_t n );
```

文字列ctを文字列sにコピーします。ただしコピーされるのは最大でn文字です。文字列がn文字より短い場合、最後のNULL文字も追加されます。

戻り値　s

strpbrk

```
#include <string.h>
char *strpbrk( const char *cs, const char *ct );
```

戻り値　文字列ctにあるcsの文字が最初に現れたときの、その文字を示すポインタ。文字が見つからない場合、NULLが返されます。

strrchr

```
#include <string.h>
char *strrchr( const char *cs, int c );
```

戻り値　文字列csにある文字cが最後に現れたときの、その文字を示すポインタ。文字が見つからない場合、NULLが返されます。

strspn

```
#include <string.h>
size_t strspn( const char *cs, const char *ct );
```

戻り値　文字列ctにある文字で構成される、文字列csの接頭辞の長さ。

strstr

```
#include <string.h>
char *strstr( const char *cs, const char *ct );
```

戻り値　文字列csにある文字列ctが最初に現れたときの、その文字を示すポインタ。文字が見つからない場合、NULLが返されます。

strtod

```
#include <stdlib.h>
double strtod( const char *s, char **endp );
```

sでポイントされる文字列の最初の部分をdoubleの値に変換します。最初にスペースがある場合はスキップされます。endpがNULLポインタでない場合にこの関数が呼び出されると、*endpは、変換で使用されない最初の文字をポイントします。

戻り値　読み込まれた値。

strtok

```
#include <string.h>
char *strtok( char *s, const shar *ct );
```

文字列ctの文字で区切られたトークンで文字列sを検索します。トークンの最後にはNULL文字が付きます。

戻り値　トークンを示すポインタ。それ以降s==NULLで呼び出しを行うと、文字列の次のトークンが返されます。

strtol

```
#include <stdlib.h>
long strtol( const char *s, char **endp, int base );
```

sによってポイントされる文字列の最初の部分をlong int型に変換します。最初にスペースがある場合、そのスペースは無視されます。値は、指定されたbaseを使用して読み込まれます。baseが0の場合、そのbaseは整数定数として定義されているものと見なされます。つまり、"0"で始まる数値は8進数で、"0x"または"0X"で始まる数値は16進数と見なされます。その他の数値は10進数と見なされます。endpがNULLポインタでない場合、この関数が呼び出された後、*endpは、変換によって使用されていない最初の文字をポイントします。

戻り値　読み込み値。

strtoul

```
#include <stdlib.h>
unsigned long strtoul( const char *s, char **endp, int base );
```

sによってポイントされる文字列の最初の部分を符号なしのlong int型に変換します。最初にスペースがある場合、そのスペースは無視されます。値は、指定されたbaseを使用して読み込まれます。baseが0の場合、そのbaseは整数定数として定義されているものと見なされます。つまり、"0"で始まる数値は8進数で、"0x"または"0X"で始まる数値は16進数と見なされます。その他の数値は10進数と見なされます。endpがNULLポインタでない場合、この関数が呼び出された後、*endpは、変換によって使用されていない最初の文字をポイントします。

戻り値　読み込み値。

strxfrm

```
#include <string.h>
size_t strncmp( char *ct, const char *cs, size_t n );
```

csでポイントされている文字列を変形して、生成される文字列をctでポイントされる配列に入れます。ctでポイントされる文字列には、NULL文字を含みn個以上の文字を入れることはできません。

戻り値　変形された文字列の長さ。

system

```
#include <stdlib.h>
int system( const char *s );
```

文字列sを、実行のための環境に渡します。

戻り値 sがNULLでコマンドプロセッサがある場合、0以外の値。sがNULLでない場合、実装定義の値。

time

```
#include <time.h>
time_t time( time_t *tp );
```

tpがNULLの場合、戻り値が*tpに割り当てられます。

戻り値 現在のカレンダー時間。カレンダー時間が使用できない場合は-1。

tmpfile

```
#include <stdio.h>
FILE *tmpfile( void );
```

クローズ時またはプログラムが正常終了するときに自動的に削除される、モード"wb+"の一時ファイルを作成します。

戻り値 正常に完了すればストリーム、ファイルが作成できない場合はNULL。

tmpnam

```
#include <stdio.h>
char *tmpnam( char s[L_tmpnam] );
```

一時的な名前(一時ファイルではない)を作成します。tmpnamが呼び出されるたびに、異なる名前が作成されます。

tmpnam(NULL)を指定すると、既存のファイル名ではない文字列が作成され、内部の静的配列を示すポインタが返されます。tmpnam(s)を指定すると、文字列が作成されそれがsに格納されたうえ、それが関数値として返されます。sには、L_tmpnam個以上の文字を入れられなければなりません。プログラムの実行時には最高でTMP_MAX個の異なる名前が保証されます。

戻り値 上記で説明した一時的な名前を示すポインタ。

toascii

```
#include <ctype.h>
int toascii( int c );
```

cをASCII値に変換します(最上位ビットを除去)。これは非ANSI関数です。

戻り値 変換された値。

tolower

```
#include <ctype.h>
int tolower( int c );
```

戻り値 cが大文字の場合小文字に変換します。それ以外の場合は、cをそのまま返します。

toupper

```
#include <ctype.h>
int toupper( int c );
```

戻り値 cが小文字の場合大文字に変換します。それ以外の場合は、cをそのまま返します。

ungetc

```
#include <stdio.h>
int ungetc( int c, FILE *fin );
```

最大1文字を入力バッファに戻してプッシュします。

戻り値　エラーの場合EOF。

va_arg

```
#include <stdarg.h>
va_arg( va_list ap, type );
```

戻り値　変数引数リストの次の引数の値。この戻り型は、引数typeの型になります。このマクロを次に呼び出すと、その次の引数の値が返されます。

va_end

```
#include <stdarg.h>
va_end( va_list ap );
```

このマクロは、引数が処理された後に呼び出す必要があります。また、マクロ"va_start"を使用する関数が終了した後呼び出します(ANSI仕様)。

va_start

```
#include <stdarg.h>
va_start( va_list ap, lastarg );
```

このマクロは、apを初期化します。このマクロの呼び出しの後、va_arg( )を呼び出すたびに、その次の引数の値が返されます。この製品の場合、va_listには、ビット型の変数を入れることができません。また指定された引数lastargは、リスト内にある最後の非ビット型引数になります。

vfprintf

```
#include <stdio.h>
int vfprintf( FILE *stream, const char *format, va_list arg );
```

vprintfと同様ですが、指定されたストリームに書き込む点で異なっています。"vprintf()"、"_iowrite()"の説明の他、"1.5.2.3 printfおよびscanfの書式化ルーチン"も参照してください。

vprintf

```
#include <stdio.h>
int vprintf( const char *format, va_list arg );
```

標準出力に、一定の書式で書き込みを行います。printf()の場合は変数引数リストが使われますが、この関数では、そのリストを示すポインタを使用します。"printf()"、"_iowrite()"の説明の他、"1.5.2.3 printfおよびscanfの書式化ルーチン"も参照してください。

vsprintf

```
#include <stdio.h>
int vsprintf( char *s, const char *format, va_list arg );
```

文字列に、一定の書式で書き込みを行います。printf()の場合は変数引数リストが使われますが、この関数では、そのリストを示すポインタを使用します。"printf()"、"_iowrite()"の説明の他、"1.5.2.3 printfおよびscanfの書式化ルーチン"の節も参照してください。

wcstombs

```
#include <stdlib.h>
size_t wcstombs( char *s, const wchar_t *pwcs, size_t n );
```

pwcsがポイントする配列にある2バイトコード文字に対応する一連のコードを、初期シフト状態で始まる一連の2バイトコード文字に変換し、これらの2バイトコード文字を、sがポイントする配列に格納します。2バイトコード文字の全バイト数が制限nを越えた時点、またはNULL文字が格納された時点で停止します。

戻り値 変更されたバイト数(終了NULL文字がある場合でも、これは含まれない)。ただし有効な2バイトコード文字に対応しないコードが見つかった場合は(size_t)-1。

wctomb

```
#include <stdlib.h>
int wctomb( char *s, wchart_t wchar );
```

wcharの値のコードに対応する2バイトコード文字を表現するのに必要なバイト数を判別します(シフト状態の変更は含まれない)。その場合、2バイトコード文字表現を、sがポイントする配列に格納します(ただしsがNULLポインタでない場合)。最大でMB_CUR_MAXの文字数が格納されます。wcharの値が0の場合、wctomb関数は、初期シフト状態のままになります。

戻り値 バイト数。ただしwcharが有効な2バイトコード文字に対応していない場合-1になります。

1.5.2.3 printfおよびscanfの書式化ルーチン

printf()、fprintf()、vfprintf()、vsprintf()などの関数は、文字列と引数を書式付きで扱う単一の関数を呼び出します。この関数が_doprint()です。この関数は、書式文字列で使用できる書式指定子が非常に多岐に渡るため、多様で大きな関数と言うことができます。このような多様な書式指定子をすべて利用しない場合、より小さいバージョンの_doprint()関数を使用することもできます。_doprint()関数には、次のような3つの異なるバージョンがあります。

LARGE　完全なフォーマッタで、制限がありません。

MEDIUM　浮動小数点プリントがサポートされていません。

SMALL　MEDIUMとほぼ同じですが、精度指定子"."も使用できません。

すべてのscanf型の関数にも同じことが当てはまります。この場合も、すべての関数が_doscan()関数を呼び出します。

ライブラリに含まれるフォーマッタはLARGEです。他のフォーマッタを指定する場合は、独立した_doscan()および_doprint()のオブジェクトをアプリケーションにリンクします。次のようなオブジェクトがあります。

lib¥libcs

_doprnts.obj	_doprint()、スモールモデル、SMALLフォーマッタ
_doprntm.obj	_doprint()、スモールモデル、MEDIUMフォーマッタ
_doprntl.obj	_doprint()、スモールモデル、LARGEフォーマッタ
_doscans.obj	_doscan()、スモールモデル、SMALLフォーマッタ
_doscanm.obj	_doscan()、スモールモデル、MEDIUMフォーマッタ
_doscanl.obj	_doscan()、スモールモデル、LARGEフォーマッタ

lib¥libcc

_doprnts.obj	_doprint()、コンパクトコード、SMALLフォーマッタ
_doprntm.obj	_doprint()、コンパクトコード、MEDIUMフォーマッタ
_doprntl.obj	_doprint()、コンパクトコード、LARGEフォーマッタ
_doscans.obj	_doscan()、コンパクトコード、SMALLフォーマッタ
_doscanm.obj	_doscan()、コンパクトコード、MEDIUMフォーマッタ
_doscanl.obj	_doscan()、コンパクトコード、LARGEフォーマッタ

lib¥libcd

_doprnts.obj	_doprint()、コンパクトデータ、SMALLフォーマッタ
_doprntm.obj	_doprint()、コンパクトデータ、MEDIUMフォーマッタ
_doprntl.obj	_doprint()、コンパクトデータ、LARGEフォーマッタ
_doscans.obj	_doscan()、コンパクトデータ、SMALLフォーマッタ
_doscanm.obj	_doscan()、コンパクトデータ、MEDIUMフォーマッタ
_doscanl.obj	_doscan()、コンパクトデータ、LARGEフォーマッタ

lib¥libcl

_doprnts.obj	_doprint()、ラージモデル、SMALLフォーマッタ
_doprntm.obj	_doprint()、ラージモデル、MEDIUMフォーマッタ
_doprntl.obj	_doprint()、ラージモデル、LARGEフォーマッタ
_doscans.obj	_doscan()、ラージモデル、SMALLフォーマッタ
_doscanm.obj	_doscan()、ラージモデル、MEDIUMフォーマッタ
_doscanl.obj	_doscan()、ラージモデル、LARGEフォーマッタ

例

```
cc88 -Ms hello.obj c:¥c88¥lib¥libcs¥_doprntm.obj
```

このコマンドを実行すると、スモールモデル用でMEDIUMの_doprint()フォーマッタが使用されます。

1.5.3 ランタイムライブラリ

コンパイラが生成するコードには、ランタイムライブラリ関数を呼び出すものもあります。ランタイムライブラリ関数は、インラインコードとして生成した場合、使用されるコードの量が非常に多くなります。ランタイムライブラリ関数の名前の最初には、常にアンダースコアが2つ付いています。ランタイムライブラリ関数の例として、32ビット除算を実行するものがあります。

c88は(オブジェクトコードではなく)アセンブラコードを生成するため、(パブリック)C変数の名前の前にアンダースコア"_"を付けて、これらのシンボルをS1C88レジスタと区別します。そのため、アンダースコアが最初に付いている関数を使用する場合、この関数のアセンブララベルの最初に2つのアンダースコアが付けられます。結果的に、この関数名は、ランタイムライブラリ関数と競合する可能性があります(二重定義)。ただしANSIでは、パブリックC変数と関数で最初にアンダースコアが付いた名前を使用するのは、結果的に定義済みのインプリメンテーションになるため、移植性が低いと定義しています。

表1.5.3.1 ランタイムライブラリ名の構文

コンパイラモデル	リンクするライブラリ
スモール(デフォルト)	librts.a (デフォルト)
コンパクトコード	librtc.a
コンパクトデータ	librtd.a
ラージ	librtl.a

1.6 浮動小数点演算

c88では浮動小数点演算がサポートされていますが、この機能は独立したライブラリのセットとしてソフトウェアに実装されています。リンク時に、必要な浮動小数点ライブラリをCライブラリのあとに指定しなければなりません。このライブラリは、再入可能で、一時プログラムスタックメモリのみを使用します。

浮動小数点演算で移植性を確保するため、**c88**では、IEEE-754の浮動小数点演算標準に準拠した浮動小数点演算が実装されています。これらの浮動小数点演算の定義の詳細については、"IEEE Standard for Binary Floating-Point Arithmetic"(1985年、IEEE Computer Society刊)を参照してください。このマニュアルでは、この文書をIEEE-754と呼びます。

c88では、単精度の浮動小数点のみをサポートしており、ANSI Cの`float`型と`double`型を介して使用できるようになっています。速度が必要な場合に備えて、それぞれのメモリモデルごとにトラップなしのライブラリも用意されています。ライブラリ名の構文については、"1.6.6 浮動小数点ライブラリ"を参照してください。

必要であれば、浮動小数点の例外が発生したときにそれを遮断し、アプリケーションで定義した例外ハンドラでそれを処理するようにすることもできます。浮動小数点の例外の遮断を"トラップ"と呼びます。トラップハンドラをインストールする方法の例については、あとで紹介します。

1.6.1 データサイズとレジスタ割り付け

c88は`float`型と`double`型をどちらも同じ4バイトのデータとして扱います。指定可能な値の範囲は次のとおりです。

+/-1,176E-38～+/-3,402E+38

レジスタにより引数や戻り値の受け渡しを行う場合、`float`型と`double`型はHLBAレジスタ(HLが上位ワード、BAが下位ワード)を使用します。

1.6.2 コンパイラオプション

c88には起動時に指定する浮動小数点演算用のオプションとして、以下に示す**-F**および**-Fc**が用意されています。**c88**の起動書式やその他のオプションについては、"1.4.2 コンパイラ"を参照してくだい。

-F/-Fc

オプション：

-F[**c**]

説明：

-Fオプションを指定すると、`double`と`long double`が使用されている場合でも、単精度の浮動小数点のみが使用されるようになります。実際、`double`と`long double`が`float`として扱われ、デフォルトで行われる`float`から`double`への引数の昇格は抑制されます。このオプションを使用するとき、単精度バージョンのCライブラリを使用しなければなりません。標準ライブラリの命名規則については、"1.6.6 浮動小数点ライブラリ"を参照してください。

-Fcオプションを指定すると、`float`定数が使用できるようになります。ANSI Cでは、浮動小数定数は、定数に接尾辞"**f**"が付いている場合を除き、`double`型を持つものとして扱われます。つまり"3.0"は倍精度定数で"3.0f"は単精度定数です。このオプションは、コンパイラに対して、すべての浮動小数点定数を単精度の`float`型として扱うよう指示します("**l**"接尾辞で明示的に指定されている場合を除く)。

例：

`double`を`float`として扱うよう指定する場合、以下のコマンドを入力します。

```
c88 -F test.c
```

1.6.3 特殊な浮動小数点値

以下のリストは、実行時に発生する特殊な浮動小数点値を示しています。ただしこれは、IEEE-754で定義されているものです。

表1.6.3.1 特殊な浮動小数点値

特殊な値	符号	指数	仮数
+0.0 (正の0)	0	すべて0	すべて0
-0.0 (負の0)	1	すべて0	すべて0
+INF (正の無限大)	0	すべて1	すべて0
-INF (負の無限大)	1	すべて0	すべて0
NaN (数値以外)	0	すべて1	すべて1

1.6.4 浮動小数点例外のトラップ

それぞれのメモリモデルごとに、以下の2つの浮動小数点ランタイムライブラリが用意されています。

浮動小数点トラップ処理機能のあるもの(`libfp`*`m`*`t.a`)

トラップ機能のないもの(`libfp`*`m`*`.a`)

トラップ処理機能を持つライブラリの名前の最後に"t"を付けることで両者を区別しています。*m*は、Cメモリモデルのいずれかを示す記号("s"スモール、"c"コンパクトコード、"d"コンパクトデータ、"l"ラージ)で置き換えられます。コントロールプログラム**cc88**に**-fptrap**オプションを指定すると、トラップタイプの浮動小数点ライブラリがアプリケーションにリンクされます。このオプションが指定されていない場合、リンク時に、トラップ機能のない浮動小数点ライブラリが使用されます。

トラップ機能のない浮動小数点ライブラリは処理速度は高速ですが、オペランドや結果が範囲外になった場合、浮動小数点演算の結果が予期できないものになります。

IEEE-754トラップハンドラ

IEEE-754標準の定義によると、トラップハンドラは、(指定された)例外イベントが発生したときに起動され、イベントに関する多くの情報と一緒に渡されることになっています。独自のトラップハンドラをインストールするときは、ライブラリ呼び出し`_fp_install_trap_handler`を使用します。また、独自の例外ハンドラをインストールするときは、関数呼び出し`_fp_set_exception_mask`を使用して、どのタイプの例外のときにハンドラを起動するか選択しなければなりません。浮動小数点ライブラリの例外処理関数のインタフェースの詳細については、以下を参照してください。

SIGFPEシグナルハンドラ

ANSI Cでは、浮動小数点例外を扱う場合、通常、ANSI Cライブラリ呼び出し`signal`を使い、いわゆるシグナルハンドラをインストールすることによって実行します。このようなハンドラがインストールされると、浮動小数点例外が発生したときにこのハンドラが起動するようになります。**SIGFPE**シグナルに対応するシグナルハンドラが、提供されている浮動小数点ライブラリで実際に動作するためには、ANSI Cライブラリの関数呼び出し`raise`によって必要なシグナルを発生させる、(非常に)基本的なバージョンのIEEE-754例外ハンドラをインストールしなければなりません(以下を参照)。そのために、関数呼び出し`_fp_install_trap_handler`が用意されています。独自の例外ハンドラをインストールするときは、関数呼び出し`_fp_set_exception_mask`を使用して、どのタイプの例外のときにシグナルを受信するか選択しなければなりません。浮動小数点ライブラリの例外処理関数のインタフェースの詳細については、以下を参照してください。

シグナルハンドラに対して、例外の文脈や性質に関する情報を指定する方法はありません。発生した浮動小数点例外を検出することだけが可能です。そのため、浮動小数点の結果についてさらに細かく制御したい場合は、上記のIEEE-754トラップハンドラの説明を参照してください。

例：
```
#include <float.h>
#include <signal.h>

static void pass_fp_exception_to_signal(_fp_exception_info_t *info)
{
   info;  /* suppress parameter not used warning */

   /* cause SIGFPE signal to be raised */

   raise( SIGFPE);

   /*
    * now continue the program
    * with the unaltered result
    */
}
```

1.6.5 浮動小数点トラップ処理API

浮動小数点演算の例外を処理するために、以下のライブラリ呼び出しが用意されています。

```
#include <float.h>

int     _fp_get_exception_mask( void );
void    _fp_set_exception_mask( int );
```

この2つの関数は、どの浮動小数点演算例外が無視され、どの例外がトラップハンドラに渡されるか制御するためのマスクを取得またはセットします。使用できる例外フラグビットのタイプとして、以下のものが定義されています。

EFINVOP
EFDIVZ
EFOVFL
EFUNFL
EFINEXCT

また、

EFALL

は、すべてのフラグの論理和になります。それぞれのフラグの詳細については、以下を参照してください。

```
#include <float.h>

int     _fp_get_exception_status( void );
void    _fp_set_exception_status( int );
```

この2つの関数は、これまでに発生したすべての浮動小数点例外タイプを累積したステータスワードを検査またはプリセットします。上記の使用可能な例外タイプフラグを参照してください。

```
#include <float.h>

void    _fp_install_trap_handler( void (*)(_fp_exception_info_t * ) );
```

この関数呼び出しは、タイプ_fp_exception_info_tの構造体を示すポインタをポイントする関数を、さらにポイントするポインタを指定します。_fp_exception_info_tのメンバーは次のようなものになります。

exception

このメンバーには、以下のいずれかの(数)値が含まれます。

EFINVOP
EFDIVZ
EFOVFL
EFUNFL
EFINEXCT

operation

このメンバーには、以下のいずれかの数値が含まれます。

_OP_ADDITION
_OP_SUBTRACTION
_OP_CMPARISON
_OP_EQUALITY
_OP_LESS_THAN
_OP_LARGER_THAN
_OP_MULTIPLICATION
_OP_DIVISION
_OP_CONVERSION

source_format
dextination_format

この2つのメンバーの数値は、以下のようなものになります。

_TYPE_SIGNED_CHARACTER
_TYPE_UNSIGNED_CHARACTER
_TYPE_SIGNED_SHORT_INTEGER
_TYPE_UNSIGNED_SHORTINTEGER
_TYPE_SIGNED_INTEGER
_TYPE_UNSIGNED_INTEGER
_TYPE_SIGNED_LONG_INTEGER
_TYPE_UNSIGNED_LONG_INTEGER
_TYPE_FLOAT
_TYPE_DOUBLE

operand1 `/* left side of binary or right side of unary */`
operand2 `/* right side for binary */`
result

この3つは、次のタイプになり、任意のタイプの値を受け取り返します。

```
typedef union _fp_value_union_t
{
        char            c;
        unsigned char   uc;
        short           s;
        unsigned short  us;
        int             i;
        unsigned int    ui;
        long            l;
        unsigned long   u1;
        float           f;
#if ! _SINGLE_FP
        double          d;
#endif
}
_fp_value_union_t;
```

次の表では、すべての例外コードフラグをリストし、それに対応するエラーの説明と結果も表記しています。

表1.6.5.1 例外タイプのフラグコード

エラーの説明	例外フラグ	トラップでのデフォルトの結果
不正な処理	EFINVOP	NaN
0による除算	EFDIVZ	+INFまたは-INF
オーバーフロー	EFOVFL	+INFまたは-INF
アンダーフロー	EFUNFL	0
不正確	EFINEXT	未定義
INF 最大の絶対浮動小数が無限大で、常に次の式が成り立ちます。 -INF < すべての有限の数値 < +INF NAN 数値以外。浮動小数点フォーマットでコード化されるシンボリックエンティティ。		

すべての例外タイプが指定されるようにするため、関数に**EFALL**を指定することができます。**EFALL**は、上記にリストされているすべてのフラグの二進論理和になります。

1.6.6 浮動小数点ライブラリ

浮動小数点を使用する場合、常に浮動小数点ライブラリをCライブラリとランタイムライブラリの間でリンクしなければなりません。これらのライブラリには、sin()、cos()などの算術ルーチンはなく(これらのルーチンはCライブラリにあります)、基本的な浮動小数点演算のみが実行できます。

表1.6.6.1 コンパイラモデルと浮動小数点ライブラリの対応

コンパイラモデル	リンクするライブラリ	
	トラップなし	トラップ
スモール(デフォルト)	libfps.a (デフォルト)	libfpst.a
コンパクトコード	libfpc.a	libfpct.a
コンパクトデータ	libfpd.a	libfpdt.a
ラージ	libfpl.a	libfplt.a

浮動小数点演算に関するヘッダファイルおよび実装されているルーチンは以下のとおりです。

<float.h> 浮動小数点演算に関連する定数。

<math.h> acos、asin、atan、atan2、ceil、cos、cosh、exp、fabs、floor、fmod、frexp、ldexp、log、log10、modf、pow、sin、sinh、sqrt、tan、tanh

<time.h> difftime(骨組みとして提供されています。)

1.6.6.1 浮動小数点演算ルーチン

acos

```
#include <math.h>
double acos( double x );
```

戻り値 [0, π]、x∈[-1, 1]の範囲のxの逆余弦$\cos^{-1}(x)$。

asin

```
#include <math.h>
double asin( double x );
```

戻り値 [-π/2, π/2]、x∈[-1, 1]の範囲のxの逆正弦$\sin^{-1}(x)$。

atan

```
#include <math.h>
double atan( double x );
```

戻り値 [-π/2, π/2]、x∈[-1, 1]の範囲のxの逆正接$\tan^{-1}(x)$。

atan2

```
#include <math.h>
double atan2( double y, double x );
```

戻り値 [-π, π]の範囲の$\tan^{-1}(y/x)$の結果。

atof

```
#include <stdlib.h>
double atof( const char *s );
```

指定された文字列をdouble値に変換します。空白は無視され、認識されない文字が最初に現れた時点で変換が終了します。

戻り値 変換されたdouble値。

ceil

```
#include <math.h>
double ceil( double x);
```

戻り値　xより大きい最小の整数をdoubleで返します。

cos

```
#include <math.h>
double cos( double x );
```

戻り値　xの余弦。

cosh

```
#include <math.h>
double cosh( double x );
```

戻り値　xの双曲余弦。

difftime

```
#include <time.h>
double difftime( time_t time2, time_t time1 );
```

戻り値　time2 - time1の結果(秒単位)。

exp

```
#include <math.h>
double exp( double x );
```

戻り値　指数関数e^xの結果。

fabs

```
#include <math.h>
double fabs( double x );
```

戻り値　値xの絶対値。|x|。

floor

```
#include <math.h>
double floor( double x );
```

戻り値　xより小さい最大の整数をdoubleで返します。

fmod

```
#include <math.h>
double fmod( double x, double y );
```

戻り値　x/yの浮動小数点剰余で、xと同じ符号が付きます。yが0の場合、結果は、実装定義されたものになります。

frexp

```
#include <math.h>
double frexp( double x, int *exp );
```

xを[1/2, 1> //C-51互換の間隔で正規化小数に分割し、その値を返して、2のべき乗を*expに格納します。xが0の場合、結果の両方の部分も0になります。たとえば、frexp(4.0, &var)の場合、$0.5 \cdot 2^3$になります。この関数は、0.5を返し、3がvarに格納されます。

戻り値　正規化小数。

ldexp

```
#include <math.h>
double ldexp( double x, int n );
```

戻り値　結果は$x \cdot 2^n$。

log

```
#include <math.h>
double log( double x );
```

戻り値　自然対数ln(x)、x>0。

log10

```
#include <math.h>
double log10( double x );
```

戻り値　底10の対数log10(x)、x>0。

modf

```
#include <math.h>
double modf( double x, double *ip );
```

xを整数と小数に分割し、それぞれにxと同じ符号を付けます。整数部分を*ipに格納します。

戻り値　小数部分。

pow

```
#include <math.h>
double pow( double x, double y );
```

x=0かつy≥0の場合、またはx<0でyが整数でない場合、ドメインエラーが発生します。

戻り値　xのy乗の結果、つまりx^y。

sin

```
#include <math.h>
double sin( double x );
```

戻り値　xの正弦。

sinh

```
#include <math.h>
double sing( double x );
```

戻り値　xの双曲正弦。

sqrt

```
#include <math.h>
double sqrt( double x );
```

戻り値　xの平方根$\sqrt{x}$。ただしx≥0。

tan

```
#include <math.h>
duble tan( double x );
```

戻り値　xの正接。

tanh

```
#include <math.h>
double tanh( couble x );
```

戻り値　xの双曲正接。

2 アセンブラ

2.1 概要

S1C88アセンブラ**as88**は、Cコンパイラ**c88**が生成したアセンブリソースファイルをアセンブルし、**lk88**でリンク可能なリロケータブルオブジェクトファイルを生成します。

アセンブラには、次のフェーズがあります。

1. プリプロセス処理
2. すべての命令の正当性チェック
3. アドレスの計算
4. オブジェクトおよび(要求した場合)リストファイルの生成

アセンブラは、IEEE-695オブジェクトフォーマットを使用して、リロケータブルオブジェクトファイルを生成します。このファイルフォーマットは、コード部分とシンボル部分を、シンボリックデバッグ情報部分と同じように指定します。

ファイルのインクルードおよびマクロ機能は、アセンブラに統合されています。詳細については、"2.5 マクロ動作"を参照してください。

2.1.1 起動

コンパイラコントロールプログラム**cc88**では、このアセンブラを自動的に呼び出すことができます。**cc88**は、コマンド行オプションの一部を**as88**のオプションに変換します。ただし、このアセンブラは、独立したプログラムとして起動することもできます。

as88を起動する場合、次のような構文になります。

as88 [*option*]...*source-file* [*map-file*]

as88 -V

-Vのみを付けて起動すると、バージョンヘッダが表示されます。

*source-file*は、アセンブリソースファイルにします。このファイルは、アセンブラの入力ソースになります。このファイルには、ユーザが記述したアセンブラコード、または**c88**が生成したアセンブラコードが含まれます。このファイルには、どのような名前を付けてもかまいません。このファイル名に拡張子がない場合、`.asm`拡張子が付いているものと見なされ、それでもファイルが見つからない場合、`.src`拡張子が付いているものと見なされます。

絶対リストファイルを生成するとき、オプションの*map-file*がアセンブラに渡されます。このマップファイルは、ロケータによって生成されます。絶対リストファイルを生成する方法については、"2.1.4.1 絶対リストファイルの生成"を参照してください。

デフォルトの場合、`.obj`拡張子を持つオブジェクトファイルが生成されます。**-l**オプションを付けた場合、`.lst`拡張子を持つリストファイルが生成されます。

オプションの前には、"-"(マイナス記号)を付けます。1つの"-"の後に、複数のオプションを組み合わせて追加することはできません。すべてが成功すると、アセンブラは、オブジェクトコードを含むリロケータブルオブジェクトモジュールを生成します。デフォルトでは、このファイルに`.obj`拡張子が付けられます。**-o**オプションを使用すれば、他の出力ファイル名を指定することもできます。**-err**アセンブラオプションを付けて、エラーリストファイルに出力するよう指示した場合を除き、エラーメッセージは端末に出力されます。

次のリストは、アセンブラのオプションを簡単にまとめたものです。次の節では、それぞれのオプションについて詳細に説明します。

オプションの要約

オプション	説　明
-C *file*	*file*をソースの前にインクルードします。
-D*macro*[=*def*]	プリプロセッサ*macro*を定義します。
-L[*flag*...]	指定されたソース行をリストファイルから削除します。
-M[**s**|**c**|**d**|**l**]	メモリモデルを指定します。
-V	バージョンヘッダのみを表示します。
-c	大文字と小文字を区別しないモードに切り換えます(デフォルトでは区別する)。
-e	アセンブリエラーの場合、オブジェクトファイルを削除します。
-err	エラーメッセージをエラーファイルにリダイレクトします。
-f *file*	オプションを*file*から読み込みます。
-i[**l**|**g**]	デフォルトのラベルスタイルをローカルまたはグローバルとして指定します。
-l	リストファイルを生成します。
-o *filename*	出力ファイルの名前を指定します。
-t	セクションの要約を表示します。
-v	冗長モード。進行中にファイル名とパスの回数をプリントします。
-w[*num*]	1つまたはすべての警告メッセージを抑制します。

2.1.2 アセンブラオプションの詳細な説明

-C

オプション：

-C *file*

引数：

インクルードファイルの名前。

説明：

ソースのアセンブルの前に*file*をインクルードします。

例：

他のインクルードファイルの前にs1c88.incをインクルードする場合は、次のコマンドを入力します。

```
as88 -C s1c88.inc test.src
```

-c

オプション：

-c

デフォルト：

大文字と小文字を区別。

説明：

大文字と小文字を区別しないモードに切り換えます。デフォルトの場合アセンブラは大文字と小文字を区別するモードで動作します。

例：

大文字と小文字を区別しないモードに切り換える場合は、次のコマンドを入力します。

```
as88 -c test.src
```

-D

オプション：

-D*macro*[=*def*]

引数：

定義したいマクロ。オプションでその定義。

説明：

*macro*を"define"のように定義します。*def*が指定されていない("="がない)場合、"1"が指定されたものと見なされます。シンボルはいくつでも定義できます。

例：

```
as88 -DTWO=2 test.src
```

-e

オプション：

-e

説明：

アセンブラでエラーが発生したとき、オブジェクトファイルが生成されないようにする場合、このオプションを使用します。このオプションを使用すると、"make"ユーティリティが常に正しいファイルを作成するようになります。

例：

```
as88 -e test.src
```

-err

オプション：

-err

説明：

アセンブラが、出力ファイルと同じベース名に拡張子.ersを付けたファイルに、エラーメッセージをリダイレクトします。アセンブラは、入力ファイルではなく、出力ファイルのベース名を使用します。

例：

エラーをstderrの代わりにtest.ersに書き込む場合、次のコマンドを実行します。

```
as88 -err test.src
```

-f

オプション：

-f *file*

引数：

コマンド行処理に使用するファイル名。標準入力を示すときは、ファイル名"-"を使用できます。

説明：

コマンド行処理のために*file*を使用します。コマンド行のサイズ制限を回避するために、コマンドファイルを使用することができます。これらのコマンドファイルに入れるオプションは、実際のコマンド行では使用できないものです。コマンドファイルは、makeユーティリティなどを使用して簡単に作成できます。

-fオプションは複数使用することができます。

コマンドファイルの書式には、次のような簡単な規則があります。

1. コマンドファイルの同じ行に複数の引数を指定することができます。
2. 引数にスペースを入れるときは、その引数を一重引用符または二重引用符で囲みます。
3. 引用符の付いた引数の中で一重引用符または二重引用符を使用する場合、次の規則に従います。
   a. 中に入っている引用符が、一重引用符のみまたは二重引用符のみの場合、引数を囲むときもう一方の引用符を使用します。つまり、引数に二重引用符が含まれる場合、引数を一重引用符で囲みます。
   b. 両方の引用符が使用されている場合、それぞれの引用符がもう一方の引用符で囲まれるような形で、引数を分割する必要があります。

   例：
   ```
   "This has a single quote ' embedded"
   ```
   または
   ```
   'This has a double quote " embedded'
   ```
   または
   ```
   'This has a double quote " and a single quote '"' embedded"
   ```
4. オペレーティングシステムによっては、テキストファイル内の行の長さに制限がある場合があります。この制限を回避するため、継続行を使用することができます。これらの行は、最後にバックスラッシュと改行が付きます。引用符の付いた引数の場合、継続行は、次の行のスペースを取らないでそのままつなげられます。引用符が付いていない引数の場合、次の行にあるすべてのスペースが削除されます。

   例：
   ```
   "This is a continuation ¥
   line"
           → "This is a continuation line"
   control(file1(mode,type),¥
           file2(type))
           → control(file1(mode,type),file2(type))
   ```
5. コマンド行ファイルは、最高で25レベルまでネストすることができます。

例：

ファイルmycmdsに次の行があると仮定します。

```
-err
test.src
```

コマンド行は、次のようになります。

```
as88 -f mycmds
```

-i

オプション：

-i[**l**|**g**]

デフォルト：

-il（ローカルラベル）

説明：

ラベル識別子のデフォルト処理を選択します。**-il**の場合、データおよびコードアセンブリのラベルは、GLOBAL擬似命令で指定を上書きしていない限り、デフォルトでLOCALラベルとして扱われます。**-ig**の場合、データおよびコードアセンブリのラベルは、LOCAL擬似命令で指定を上書きしていない限り、デフォルトでGLOBALラベルとして扱われます。

例：

アセンブリラベル識別子が、デフォルトでGLOBALラベルとして扱われるよう指定する場合、次のコマンドを入力します。

```
as88 -ig test.src
```

-L

オプション：

-L[*flag...*]

引数：

オプションで、リストファイルから削除されるソース行を指定するフラグ(複数指定可)。

デフォルト：

-LcDEGlMnPQsWXy

説明：

リストファイルから削除されるソース行を指定します。リストファイルは、**-l**オプションが指定されたときに生成されます。**-L**オプションを指定しない場合、アセンブラは、#line擬似命令またはシンボリックデバッグ情報が含まれるソース行および空のソース行を削除し、折り返されたソース行を1行にします。フラグを付けないで**-L**を指定した場合、**-Lcdeglmnpqswxy**と同等になり、指定されたすべてのソース行がリストファイルから削除されます。

フラグは、小文字のときオンになり、大文字のときオフになります。次のようなフラグを使用することができます。

c デフォルト。アセンブラコントロール(OPTIMIZE擬似命令)を含むソース行が削除されます。
C アセンブラコントロールを含むソース行がそのまま保持されます。
d セクション擬似命令(DEFSECT、SECT擬似命令)を含むソース行が削除されます。
D デフォルト。セクション擬似命令を含むソース行がそのまま保持されます。
e EXTERN、GLOBAL、LOCALなどのシンボル定義擬似命令を含むソース行が削除されます。
E デフォルト。シンボル定義擬似命令を含むソース行がそのまま保持されます。
g 包括的命令展開が削除されます。
G デフォルト。包括的命令展開がそのまま保持されます。
l デフォルト。Cプリプロセッサ行情報(#lineの行)を含むソース行が削除されます。
L Cプリプロセッサ行情報を含むソース行がそのまま保持されます。
m マクロ/繰り返し擬似命令を含むソース行(MACRO、DUPがある行)が削除されます。
M デフォルト。マクロ/繰り返し擬似命令を含むソース行がそのまま保持されます。
n デフォルト。空のソース行(改行のみの行)が削除されます。
N 空のソース行がそのまま保持されます。
p 条件アセンブラを含むソース行(IF、ELSE、ENDIFがある行)が削除されます。有効な条件のみが残されます。
P デフォルト。条件アセンブラを含むソース行がそのまま保持されます。
q アセンブラ定数定義を含むソース行(EQUがある行)が削除されます。
Q デフォルト。アセンブラ定数定義を含むソース行がそのまま保持されます。
s デフォルト。高レベル言語のシンボリックデバッグ情報を含むソース行(SYMBがある行)が削除されます。
S HLLのシンボリックデバッグ情報を含むソース行がそのまま保持されます。
w ソース行の折り返し部分が削除されます。
W デフォルト。ソース行の折り返し部分がそのまま保持されます。
x MACRO/DUP展開を含むソース行が削除されます。
X デフォルト。MACRO/DUP展開を含むソース行がそのまま保持されます。
y デフォルト。サイクルカウントを非表示にします。
Y サイクルカウントを表示します。

例：

生成されるリストファイルから、アセンブラコントロールが含まれるソース行を削除して、折り返されたソース行を削除する場合、次のコマンドを入力します。

```
as88 -l -Lcw test.src
```

-l

オプション：

-l

説明：

リストファイルを生成します。リストファイルには、出力ファイルと同じベース名が付けられます。拡張子は、.lstになります。

例：

test.lstという名前のリストファイルを生成する場合、次のコマンドを入力します。

```
as88 -l test.src
```

参照：

-L

-M

オプション：

-M*model*

引数：

使用するメモリモデル。*model*は次のいずれかになります。

s	スモール	最大64Kのコードとデータ。
c	コンパクトコード	最大64Kのコードと16Mのデータ。
d	コンパクトデータ	最大8Mのコードと64Kのデータ。
l	ラージ	最大8Mのコードと16Mのデータ。

デフォルト：

-Ml

説明：

ソースファイルのアセンブルに使用されるメモリモデルを指定します。異なるモデルのアセンブラモジュールは、一緒にリンクすることができません。

例：

バンク1モデルを使用してアセンブルする場合、次のコマンドを入力します。

```
as88 -Ms test.src
```

-o

オプション：

-o *filename*

引数：

出力ファイルの名前。ファイル名はオプションの直後に続けられません。オプションの後にタブまたはスペースを入れる必要があります。

デフォルト：

アセンブラファイルのベース名に.obj接尾辞を付けたもの。

説明：

アセンブラファイルのベース名＋.obj拡張子の代わりに、*filename*をアセンブラの出力ファイル名として使用します。

例：

ジャンプおよび分岐の最適化を有効にする場合、次のコマンドを入力します。

```
as88 test.src -o myfile.obj
```

-t

オプション：

-t

説明：

トータル(セクションサイズの要約)を生成します。stdoutで、セクションごとにメモリアドレス、サイズ、サイクル数、名前がリストされます。

例：

as88 -t test.src

```
Section summary:
NR ADDR    SIZE CYCLE NAME
 1         0007     5 .text
 2 021234  000e     0 .data
 3         0001     0 .tiny
```

-V

オプション：

-V

説明：

このオプションを使用すると、アセンブラのバージョンヘッダのみが表示されます。このオプションは、**as88**の唯一の引数にする必要があります。他のオプションは無視されます。アセンブラは、バージョンヘッダを表示した後終了します。

例：

as88 -V

```
S1C88 assembler va.b rc      SN000000-015 (c) year TASKING, Inc.
```

-v

オプション：

-v

説明：

冗長モード。このオプションを指定すると、アセンブラは、処理中にファイル名とアセンブラパスの回数をプリントします。そのため、アセンブラの現在のステータスがわかります。

例：

as88 -v test.src

```
Parsing "test.src"
   30 lines (total now 31)
Optimizing
Evaluating absolute ORG addresses
Parsing symbolic debug information
Creating object file "test.obj"
Closing object file
```

-W

オプション：

-w[*num*]

引数：

オプションで、抑制する警告番号。

説明：

-wを使用すると、すべての警告メッセージが抑制されます。**-w***num*を使用すると、番号*num*の警告メッセージが抑制されます。**-w***num*オプションは複数指定することもできます。

例：

警告113および114を抑制するときは、次のコマンドを実行します。

as88 -w113 -w114 file.src

2.1.3 as88で使用される環境変数

AS88INC　この環境変数は、**as88**アセンブラがインクルードファイルを検索するディレクトリを指定するときに使用します。複数のパス名はセミコロンで区切ることができます。

インクルードファイルの名前が""で囲まれている場合は、#include行を含むファイルのディレクトリが最初に探され、次にカレントディレクトリの順に探されます。それでもインクルードファイルが見つからない場合、アセンブラは、AS88INCで指定されているディレクトリを探します。最後に、**as88.exe**があるディレクトリの相対パス`..¥include`が探されます。

インクルードファイルの名前を<>で囲んで指定する場合、#include行を含むファイルのディレクトリとカレントディレクトリは検索されません。AS88INCのディレクトリおよび相対パスは検索されます。

TMPDIR　TMPDIR環境シンボルを使用すると、アセンブラが一時ファイルを生成するディレクトリを指定することができます。アセンブラが正常に終了した場合、一時ファイルは自動的に削除されます。TMPDIRが設定されていない場合、一時ファイルは、現在の作業ディレクトリに作成されます。

2.1.4 リストファイル

リストファイルはアセンブラの出力ファイルで、生成されたコードについての情報が含まれています。情報の量と形式は、**-L**オプションを使用したかどうかにより異なってきます。このファイルの名前は、出力ファイルのベース名に拡張子`.lst`を付けたものになります。**-l**オプションを指定した場合、リストファイルは、唯一の生成ファイルになります。**-l**を指定した場合、アセンブラエラー/警告が発生したときにもリストファイルが生成されます。この場合、エラー/警告は、エラー/警告を発生させたソース行のすぐ下に出力されます。

2.1.4.1 絶対リストファイルの生成

アプリケーション全体をロケートした後、アセンブラのアセンブラソース入力ファイルとして絶対リストファイルを生成することができます。アセンブラソースから絶対リストファイルを生成するとき、そのアセンブラソースが所属するアプリケーションのロケータマップファイルを使用して、アセンブルし直す必要があります。ロケータマップファイルを生成する方法については、"4.5 ロケータの出力"を参照してください。

リストファイルオプション**-l**でリストファイルの生成を有効にした場合、マップファイルがアセンブラの入力として指定されていれば、絶対リストファイルが生成されます。絶対リストファイルの生成を有効にしている場合、すでに生成されているオブジェクトファイルは上書きされません。絶対リストファイルの生成は、入力としてファイル名拡張子`.map`を持つマップファイルが指定されている場合に限り有効になります。

注： 絶対リストファイルを生成したい場合、オブジェクトファイルを生成したいときと同じオプションを指定する必要があります。オプションが同じでない場合、不正な絶対リストファイルが生成されます。

例：

最初の起動コマンドが、

```
as88 -Ms test.src
```

だった場合、絶対リストファイルを生成するとき、同じオプション(**-Ms**)と**-l**オプションを指定する必要があります。

```
as88 -Ms -l test.src test.map
```

このコマンドを使用すると、絶対リストファイル"test.lst"が作成されます。

2.1.4.2 ページヘッダ

ページヘッダは、4行で構成されます。

最初の1行には、次のような情報が含まれます。

- アセンブラ名についての情報
- バージョンとシリアル番号
- 著作についての表記

2行目には、TITLE(最初のページ)またはSTITLE(以降のページ)のコントロールで指定されたタイトルの他、ページ番号が含まれます。

3行目には、ファイルの名前(最初のページ)が含まれるか、または空になります(以降のページ)。

4行目には、次の節で説明するようなソースリストのヘッダが含まれます。

例：

```
S1C88 assembler va.b rc      SNzzzzzz-zzz (c) year TASKING, Inc.
Title for demo use only                                 page    1
/tmp/hello.asm
ADDR   CODE      CYCLES LINE SOURCELINE
```

2.1.4.3 ソースリスト

ページヘッダには、次のような行が記述されます。

```
ADDR   CODE     CYCLES LINE SOURCELINE
```

次にそれぞれのカラムについて説明します。

ADDR

このカラムには、メモリアドレスが入ります。このアドレスは(6桁の)16進数になり、リロケータブルセクションの最初からのオフセット、または絶対セクションの絶対アドレスを示します。

オブジェクトコードを生成する行の場合、この値は行の最初に記述されます。他の行の場合、何も記述されません。

例：

```
ADDR   CODE     LINE SOURCELINE
000000             1         defsect ".text", code
000000             2         sect    ".text"
000000 CEC6rr      4         ld      xp,#@dpag(data_label)
000003 CEC4rr      7         ld      nb,#@cpag(label)
000006 F101        8         jr      label
                   .
                   .
021234            13         defsect ".data", data at 21234h
                  14 data_label:
021234            16         ds      49
   |   RESERVED
021264
```

CODE

このカラムには、アセンブラによってこのソース行のため生成された16進法のオブジェクトコードが入ります。ここに記述されるコードは、オブジェクトモジュール内に生成されたコードと必ずしも同じではありません。このコードは、リロケータブルコードの他、リロケータブル部分や外部部分になることもあります。この場合、リストのリロケータブルコード部分に対して文字"r"がプリントされます。

空間を割り当てる行(DS)の場合、コードフィールドには"RESERVED"というテキストが入れられます。

例：

```
ADDR   CODE     LINE SOURCELINE
                   .
                   .
000000 CEC6rr      4         ld      xp,#@dpag(data_label)
000003 CEC4rr      7         ld      nb,#@cpag(label)
000006 F101        8         jr      label
                   .
                   .
021234            13         defsect ".data", data at 21234h
                  14 data_label:
021234            16         ds      49
   |   RESERVED
021264
```

この例では、"RESERVED"という単語が、**ds**擬似命令で予約された空間を示しています。

CYCLES

アセンブラでオプション**-LY**を指定した場合、リストファイルにCYCLESカラムも表示されます。最初の値は、命令のサイクルカウントを示し、2番目の値は、累積サイクルカウントを示します。

例：

```
ADDR   CODE      CYCLES LINE SOURCELINE
                            .
                            .
000000 CEC6rr    3    3    4         ld      xp,#@dpag(data_label)
000003 CEC4rr    4    7    7         ld      nb,#@cpag(label)
000006 F101      2    9    8         jr      label
                            .
                            .
```

LINE

このカラムには、行番号が入れられます。行番号は、それぞれの入力行を示す10進数で、それぞれのソース行に対応する番号が、1から順に入れられます。($LIST OFFなどで)行のリストを抑制した場合、この値は1ずつ増えていきます。

例：

次のソース部分、

```
                ;Line 12
        $LIST OFF
                ;Line 14
        $LIST ON
                ;Line 16
```

により、次のリストファイル部分が生成されます。

```
ADDR   CODE    CYCLES LINE SOURCELINE
                          .
                          .
                         12           ;Line 12
                         16           ;Line 16
```

SOURCELINE

このカラムには、ソーステキストが入ります。これは、ソースモジュールのソース行のコピーになります。リストファイルを読みやすくするため、タブがスペースに展開されています。

リストのソースカラムが狭すぎて、すべてのソース行を表示できない場合、そのソース行は次のリスト行に続けられます。

エラーや警告は、リストファイルで、エラーや警告の原因になったソース行のすぐ下に出力されます。

例：

```
ADDR   CODE     CYCLES LINE SOURCELINE
                           .
                           .
021271 FF8F               29              dw      @coff(@caddr(300,8fffh))
as88 W172: /tmp/t.src line 29 : page number must be between 0 and FF
```

2.1.5 デバッグ情報

Cコンパイラが生成したデバッグ情報がソースファイルにある場合、**as88**はその情報を含めてオブジェクトファイルを生成します。これにより、Cソースのシンボリックデバッグが可能になります。**as88**自体は新たなデバッグ情報を生成しません。

2.1.6 命令セット

as88は、S1C88用に定義されたすべてのアセンブリ言語命令ニーモニックを受け付けます。

ニーモニック、オペランド、命令コードの書式と状態を記述したすべての命令のリストについては、"S1C88コアCPUマニュアル"を参照してください。

以下に、注意事項を示します。

RETS命令の処理

プログラムに**rets**命令が含まれる場合、特に注意が必要になります。**rets**は戻りアドレス+2に復帰します。この命令は戻るアドレスに影響を及ぼすため、以下の状況では動作しません。

```
        carl  _label
        carl  _function ;  3バイト命令
        ...
_label:
        ...
        rets            ;  --> 結果的に3バイトの"carl  _function"命令の
                        ;  真ん中(戻りアドレス+2)にジャンプすることになります。
```

アセンブラは、この種の矛盾を検出できません。

2.2 ソフトウェアの概念

2.2.1 はじめに

複雑なソフトウェアプロジェクトは、多くの場合小さなプログラムユニットに分けられます。これらのサブプログラムは、プログラマのチームが同時進行で記述することができる他、開発の過程で再利用する目的で記述することもできます。**as88**アセンブラでは、プログラムを小さな部品、つまりモジュールに分割するための擬似命令をサポートしています。シンボルをあるモジュールに対してローカルに定義することにより、他のモジュールでもそのシンボルに関係なくシンボル名を使用できるようにすることができます。またコードとデータを、別のセクションに入れることもできます。これらのセクションは、異なるモジュールが、これらのセクションの異なる部分を実装できるような方法で、命名することができます。これらのセクションは、ロケータがメモリ内にロケートできるため、メモリの配置の問題は、アセンブリプロセス以降に先送りされます。独立したモジュールを使用すれば、他のモジュールをアセンブルし直さずにモジュールを変更できるようになります。こうすることによって、開発プロセスの所要時間を短縮することができます。

2.2.2 モジュール

モジュールとは、プロジェクトの個別の実装部分を示します。それぞれのモジュールは、個別のファイルで定義されます。各モジュールは、それぞれ個別にアセンブルされます。INCLUDE擬似命令を使用することで、共通の定義およびマクロをそれぞれのモジュールにインクルードすることができます。**mk88**ユーティリティを使用すると、モジュールファイルとインクルードファイルの依存関係を指定できるため、モジュールが依存するファイルを変更した後、適切なモジュールのみをアセンブルし直すことができます。

2.2.2.1 モジュールおよびシンボル

モジュールでは、他のモジュールまたはそのモジュールで定義されているシンボルを使用することができます。モジュールで定義されているシンボルは、ローカル(他のモジュールがアクセスできない)にすることもグローバル(他のモジュールがアクセスできる)にすることもできます。モジュールの外部のシンボルは、EXTERN擬似命令で定義します。シンボルをLOCAL擬似命令で定義するか、SET擬似命令またはEQU擬似命令で定義すると、ローカルシンボルになります。グローバルシンボルは、ラベルになるか、GLOBAL擬似命令で明示的にグローバルと定義したシンボルになります。

2.2.3 セクション

セクションとは、コードとデータの再配置可能なブロックを示します。セクションは、DEFSECT擬似命令で定義され、名前が付けられます。セクションにはさまざまな属性があり、この属性によって、近くまたは遠くのメモリにある定義済みの開始アドレスに配置するようロケータに指示することができます。また、このセクションを他のセクションで重ね書きできることを指示することも可能です。使用できる属性の詳細については、DEFSECT擬似命令の説明を参照してください。セクションは、一度定義した後、SECT擬似命令で起動します。リンカは、異なるモジュール間をチェックし、セクションの属性が適切でない場合、エラーメッセージを出します。またリンカは、一致するすべてのセクション定義を1つのセクションに連結します。そのため、コンパイラが生成したすべての".text"セクションは、1つの".text"チャンクにリンクされ、1つの断片としてロケートされます。この命名方式を使用することにより、リンクフェーズで、すべてのコードまたはデータの断片を集めて1つの大きなセクションにすることができます。すでに定義されているセクションを参照するSECT擬似命令は、継続と呼ばれます。この場合、名前のみを指定することができます。

2.2.3.1 セクション名

アセンブラは、オブジェクトファイルをリロケータブルIEEE-695オブジェクトフォーマットで生成します。アセンブラは、オブジェクトファイルで、セクションを使用してコードおよびデータのユニットをグループ化します。すべてのリロケータブル情報は、セクションの開始アドレスとの相対値になります。ロケータは、セクションに絶対アドレスを割り当てます。セクションは、コードまたはデータの最小単位で、ソースファイルをアセンブルした後、メモリ内の特定のアドレスに移動させることができます。コンパイラによる必要性のため、アセンブラは、特定の特性を割り当てるために適切な属性を持つ複数の異なるセクション(読み取り専用データを持つセクション、コードを持つセクションなど)をサポートする必要があります。

DEFSECT *sect_name, sect_type* [*, attrib*] ... [**AT** *address*]

セクションは、宣言してからでないと使用できません。DEFSECT擬似命令では、属性を付けてセクションを宣言します。セクション名は、どのような識別子でもかまいません。ただし通常のセクション名には"@"文字を使用することはできません。この文字は、重ね書き可能なセクションを作成するときに、アセンブラとリンカによって使用されます。これについては、後で説明します。

セクションの型には、次のようなものを使用することができます。

sect_type: CODE|DATA　これは、セクションがロケートされるメモリ(CODE、DATA)を定義するものです。

セクションの属性には、次のようなものを使用することができます。

attrib:	SHORT	コードメモリの最初の32K内に収まるか、データメモリの最初の64K内に収まります。
	FIT 100H	セクションを256Bページに収める必要があります。
	FIT 8000H	セクションを32KBページに収める必要があります。
	FIT 10000H	セクションを64KBページに収める必要があります。
	CLEAR	プログラムのスタートアップ時にセクションをクリアします。
	NOCLEAR	セクションがプログラムのスタートアップ時にクリアされません。
	INIT	スタートアップ時に初期化データがROMからRAMへコピーされます。
	OVERLAY	セクションが、重ね書きの名前を持っている必要があります。
	ROMDATA	セクションが、実行可能コードの代わりに、データを含んでいます。
	JOIN	セクションをグループ化します。

オフになっていない限り、ツールチェーンのスタートアップコードは、**CLEAR**属性を持つデータセクションをクリアする必要があります。これらのセクションには、イニシャライザが指定されていないデータ空間割り当てが含まれます。CLEARセクションは、プログラムのスタートアップ時にゼロにセット(クリア)されます。**NOCLEAR**属性を付けることによって、セクションから、この初期化を免除することができます。これは、すべてのセクションのデフォルトの状態にもなっています。

SHORT属性が付いたセクションは、(CODEセクションの場合)S1C88のコードメモリの最初の32Kバイト、(DATAセクションの場合)データメモリの最初の64Kバイト内に割り当てなければなりません。SHORT属性の付いたセクションがこの領域に割り当てられない場合、ロケータが警告を出します。

JOIN属性を使用すると複数のセクションをグループ化することができます。たとえば、同じデータページにもっと多くのセクションをロケートしたい場合、この属性を使用することができます。

OVERRAY属性を指定すると、セクションを重ね書き可能にすることができます。重ね書き可能なのは、DATAセクションのみです。この属性が他の型のセクションと組み合わされている場合、アセンブラがエラーを報告します。重ね書きされたセクションをプログラムのスタートアップ時に初期化するのは無意味であるため(重ね書きされたデータを使用するコードの場合、最初の使用時にそのデータが定義どおりの状態になっていると見なすことはできない)、OVERRAYが指定された場合、NOCLEAR属性が暗黙的に定義されます。重ね書き可能なセクション名は、次のように構成されます。

```
DEFSECT    "OVLN@nfunc", DATA, OVERLAY, SHORT
              ↑     ↑
           プール名  関数名
```

リンカは、セクションを同じプール名で重ね書きします。リンカは、DATAセクションとXDATAセクションのどちらを重ね書きできるか決定するために、呼び出しグラフを構築します。互いに呼び出しあう関数の場合、それに属するセクションのデータは重ね書きすることができません。コンパイラは、リンカがこの呼び出しグラフを構築するための情報を持つ擬似命令(CALLS)を生成します。CALLS擬似命令は、(簡略化すると)次の構文になります。

CALLS '*caller_name*', '*callee_name*'[, '*callee_name*']...

関数`main()`で、近いメモリおよび呼び出し`nfunc()`に重ね書き可能なデータ割り当てがある場合、次のセクションと呼び出し情報が生成されます。

```
DEFSECT "OVLN@nfunc", DATA, OVERLAY, SHORT
DEFSECT "OVLN@main", DATA, OVERLAY, SHORT

CALLS 'main', 'nfunc'
```

ATキーワードを使用する宣言でアドレスが指定されている場合、セクションは絶対セクションになります。アセンブラは、セクションの内容を指定アドレスに置くようロケータに命令する情報をオブジェクトファイルに生成します。重ね書き可能なセクションを絶対セクションにすることはできません。ATキーワードがOVERLAYセクション属性と組み合わされて使用されている場合、エラーが報告されます。

セクションは、宣言された後、SECT擬似命令によって起動および再起動することができます。

```
          DEFSECT ".STRING", CODE, ROMDATA
          SECT    ".STRING"
_l101:    ASCII   "hello world"
```

データまたはコードを生成するすべての命令と疑似命令は、アクティブセクション内になければなりません。セクション定義や起動を行わずにコードやデータを開始すると、警告が出されます。

2.2.3.2 絶対セクション

絶対セクション(つまり開始アドレスを持つDEFSECT擬似命令)は、定義しているモジュールのみで続行することができます(継続)。このようなセクションが、他のモジュールで同じ方法によって定義されている場合、ロケータは、2つのセクションを同じアドレスに置こうとします。そのため、ロケータエラーが発生します。絶対セクションが複数のモジュールで定義されている場合、セクションをリロケータブルとして定義し、その開始アドレスをロケータ記述(`.dsc`)ファイルで定義する必要があります。また、オーバーレイセクションは、絶対セクションとして定義することができません。

2.2.3.3 グループ化セクション

複数のセクションを1ページ内にグループ化しなければならない場合、**JOIN**セクション属性を使用します。**JOIN**属性は、ページサイズを定義する**FIT**属性と一緒に使用しなければなりません。ある特定のグループに対するページサイズは、そのグループ内のすべてのセクションで同じになる必要があります。たとえば、2つのデータセクションを同じ64Kページに置かなければならない場合、以下のように記述することができます。

```
DEFSECT ".data1@group", DATA, JOIN, FIT 10000H
SECT    ".data1@group"
```

また2番目のセクションでは、次のようにします。

```
DEFSECT ".data2@group", DATA, JOIN, FIT 10000H
SECT    ".data2@group"
```

セクションは、セクション名に使用されている拡張子ごとにグループ化されます。そのため、定義は次のようになります。

```
DEFSECT    "sect@group", DATA, JOIN, FIT 10000H
                ↑     ↑
          セクション名   ジョイングループ名
```

2.2.3.4 セクションの例

次にDEFSECT擬似命令とSECT擬似命令の例を示します。

```
DEFSECT ".CONST", CODE AT 1000H
SECT    ".CONST"
```

アドレス1000Hで始まる.CONSTという名前のセクションを定義して起動します。同じモジュールにある、同じセクションの他の部分は、次のように定義する必要があります。

```
SECT    ".CONST"
```

```
DEFSECT ".text", CODE
SECT    ".text"
```

CODEメモリにリロケータブルセクションを定義して起動します。このセクションで同じ名前を持つ他の部分は、同じモジュールで定義できる他、他の任意のモジュールでも定義することができます。他のモジュールでは、同じDEFSECT文を使用する必要があります。必要であれば、ロケータ記述ファイルを使用して、セクションに絶対開始アドレスを指定することもできます。

```
DEFSECT ".fardata", DATA, CLEAR
SECT    ".fardata"
```

DATAメモリにリロケータブルの名前付きセクションを定義します。CLEAR属性は、ロケータに対して、このセクションにロケートされたメモリをクリアするよう命令します。このセクションが他のモジュールで使用されている場合、それぞれ個別に定義する必要があります。同じモジュール内にあるこのセクションの継続は、次のようになります。

```
SECT     ".fardata"
```

```
DEFSECT ".ovlf@f", DATA, OVERLAY
SECT    ".ovlf@f"
```

DATAメモリにリロケータブルセクションを定義します。このセクションは、他の重ね書き可能なDATAセクションで重ね書きされます。この重ね書き可能な部分に対応する関数は"f"です。これは、リンカが正しい呼び出しグラフを作成する上でどの関数呼び出しが必要か指定するため、CALLS擬似命令で使用される名前になります。"2.2.3.1 セクション名"も参照してください。

2.3 アセンブラ言語

2.3.1 入力の指定

アセンブラプログラムは、0個以上の文で構成され、1文を1行に記述します。文には、オプションでコメントを付けることができます。コメントはセミコロン(;)の後に付けて、入力行の最後まで続けます。ソースの文は、次の行に続く継続行の最後の文字として継続文字(\)を行の最後に付けることで、複数行に拡張することができます。ソースの文(最初の行と継続行)の長さは、使用可能メモリの制限のみを受けます。アセンブラのニーモニックと擬似命令の場合、大文字と小文字は同一視されますが、ラベル、シンボル、擬似命令の引数、リテラル文字列などでは区別されます。

文は、次の書式で定義します。

[*label*:] [*instruction*|*directive*|*macro_call*] [;*comment*]

ただし各項目は、次のようになります。

label 識別子です。ラベルは行の最初に始める必要はありませんが、ラベルの後には、常にコロンを付ける必要があります。

識別子には、文字や数字の他、アンダースコア(_)を使用することができます。ただし最初の文字には、数字を使用できません。識別子のサイズは、使用可能メモリの制限のみを受けます。

例：

```
LAB1:   ;This is a label
```

instruction ニーモニックやオペランドで構成される有効なS1C88アセンブラ言語命令です。オペランドについては、"2.4 オペランドと式"で説明しています。命令については、"S1C88コアCPUマニュアル"を参照してください。

例：

```
RET             ; No operand
PUSH  A         ; One operand
ADD   BA,HL     ; Two operands
```

directive アセンブラ擬似命令。"2.6 アセンブラ擬似命令"で説明しています。

macro_call 定義済みのマクロの呼び出しです。"2.5 マクロ動作"を参照してください。

文は空にすることもできます。

2.3.2 アセンブラで有効な文字

アセンブラで有効な1文字のシーケンスがいくつかあります。文字の中には、使用されているコンテキストによって異なる意味を持つものもあります。式の評価に関連する特殊文字については、"2.4 オペランドと式"で説明しています。その他のアセンブラ有効文字には、次のようなものがあります。

; - コメント区切り文字

¥ - 行連結文字
　またはマクロのダミー引数連結演算子

? - マクロの値置換演算子

% - マクロの16進値置換演算子

^ - マクロのローカルラベル演算子

" - マクロの文字列区切り文字
　または引用符が付いた文字列のDEFINE展開文字

@ - 関数区切り文字

***** - ロケーションカウンタの置換演算子

[] - ロケーションアドレッシングモード演算子

- 即値アドレッシングモード演算子

それぞれのアセンブラ特殊文字については、この後説明します。それぞれの説明では、使用法のガイドライン、機能の説明、例などを紹介します。

;

コメント区切り文字

セミコロン(;)の後にある数字や文字で、リテラル文字列の一部でないものは、コメントと見なされます。コメントは、アセンブラにとっては無意味なものですが、ソースプログラムの記録を残すときに使用することができます。コメントは、アセンブラ出力リストでは再生成され、マクロ定義では保持されます。

コメントを記述するとき、1行全部を使用することもできますが、最後の有意なアセンブラフィールドの後に置くこともできます。コメントは、リストファイルでも、元のように再生されます。

例：

```
; This comment begins in column 1 of the source file
Loop:   CALL    [COMPUTE]     ; This is a trailing comment
        ; These two comments are preceded
        ; by a tab in the source file
```

¥ (\)

行連結文字またはマクロのダミー引数連結演算子

行連結

円記号(¥)またはバックスラッシュ(\)は、行の最後に使用された場合、そのソース文が次の行に継続することをアセンブラに指示します。連結行は、ソース文の前の行に連結されて、結果的に単一のソース文であるかのように処理されます。ソース文の最大長(最初の行と継続行)は512文字です。

例：

```
; THIS COMMENT ¥
EXTENDS OVER ¥
THREE LINES
```

マクロのダミー引数連結演算子

円記号(¥)またはバックスラッシュ(\)は、マクロのダミー引数を隣接する他の英数字と連結するときに使用することもできます。ダミー引数を認識するマクロプロセッサの場合、通常、シンボル以外の文字を使用して、ダミー引数を他の英数字と区別する必要があります。ただし、場合によっては、引数文字を他の文字と連結する方が良い場合もあります。引数を他のシンボル文字の前または後に連結する場合、バックスラッシュをそれぞれ後または前に置く必要があります。

"2.5.5.1 ダミー引数連結演算子 - \"も参照してください。

例：

ソース入力ファイルに次のマクロ定義があると仮定します。

```
SWAP_MEM MACRO  REG1,REG2       ;swap memory contents
         LD     A,[I¥REG1]      ;using A as temp
         LD     B,[I¥REG2]      ;using B as temp
         LD     [I¥REG1],B
         LD     [I¥REG2],A
         ENDM
```

連結演算子(¥)は、ダミー引数の置換文字を、いずれの場合も文字@に連結するよう、マクロプロセッサに対して指示します。このマクロが次の文で呼び出された場合、

```
         SWAP_MEM    X,Y
```

結果は次のように展開されます。

```
         LD      A,[IX]
         LD      B,[IY]
         LD      [IX],B
         LD      [IY],A
```

?

シンボル文字の戻り値

?*symbol*シーケンスがマクロ定義で使用された場合、*symbol*の値を示すASCII文字列で置換されます。この演算子は、連結演算子(\)と同等の方法で使用できます。*symbol*の値は整数でなければなりません。

"2.5.5.2 戻り値演算子 - ?"も参照してください。

例：

次のマクロ定義があると仮定します。

```
SWAP_MEM MACRO  REG1,REG2       ;swap memory contents
         LD     A,[_lab¥?REG1]  ;using A as temp
         LD     B,[_lab¥?REG2]  ;using B as temp
         LD     [_lab¥?REG1],B
         LD     [_lab¥?REG2],A
         ENDM
```

ソースファイルに次のSET文とマクロ呼び出しがある場合、

```
AREG     SET      1
BREG     SET      2
         SWAP_MEM AREG,BREG
```

ソースリストに書き出される展開結果は次のようになります。

```
         LD      A,[_lab1]
         LD      B,[_lab2]
         LD      [_lab1],B
         LD      [_lab2],A
```

%

シンボル文字の16進戻り値

%*symbol*シーケンスがマクロ定義で使用された場合、*symbol*の16進値を示すASCII文字列で置換されます。この演算子は、連結演算子(\)と同等の方法で使用できます。*symbol*の値は整数でなければなりません。

"2.5.5.3 戻り16進値演算子 - %"も参照してください。

例：

次のマクロ定義があると仮定します。

```
GEN_LAB    MACRO LAB,VAL,STMT
LAB¥%VAL:  STMT
           ENDM
```

このマクロが次のように呼び出された場合、

```
NUM      SET      10
         GEN_LAB  HEX,NUM,'NOP'
```

リストファイルに書き出される展開結果は次のようになります。

```
HEXA:    NOP
```

^

マクロのローカルラベル演算子

アクセント記号(^)が、マクロ定義で単一の演算子として使用された場合、対応するローカルラベルの名前が無効になります。通常、マクロプリプロセッサは、マクロ展開にローカルラベルがある場合、現在のモジュールでは標準のラベルにしておきます。ローカルラベル文字(^)を使用すると、そのラベルは固有のラベルになります。これは、先行するアンダースコアを削除して固有の文字列"__M_Lxxxxxx"(ただし"xxxxxx"は固有の順序番号)を追加することで実行します。^演算子は、マクロ展開外では何の影響も与えません。^演算子は、マクロでローカルラベル名として使用されるマクロ引数としてラベル名を渡すときに使用すると便利です。アクセント記号は、2進数の排他的論理和演算子としても使用されます。

"2.5.5.5 マクロのローカルラベル演算子 - ^"も参照してください。

例：

次のマクロ定義があると仮定します。

```
LOAD     MACRO    ADDR
         ADDR:
                  LD       A,[ADDR]
         ^ADDR:
                  LD       A,[^ADDR]
         ENDM
```

このマクロが次のように呼び出された場合、

```
         LOAD     _LOCAL
```

リストファイルに書き出される展開結果は次のようになります。

```
_LOCAL:
         LD       A,[_LOCAL]
_LOCAL__M_L000001:
         LD       A,[_LOCAL__M_L000001]
```

"

マクロの文字列区切り文字または引用符が付いた文字列のDEFINE展開文字

マクロ文字列

二重引用符(")は、マクロ定義で使用された場合、マクロプロセッサによって文字列の区切り文字、一重引用符(')に変換されます。マクロプロセッサは、マクロ引数の二重引用符の間にある文字を調べます。このメカニズムにより、マクロ引数をリテラル文字列として使用することができます。

"2.5.5.4 ダミー引数文字列演算子 - ""も参照してください。

例：

次のマクロ定義と

```
CSTR    MACRO   STRING
        ASCII   "STRING"
        ENDM
```

次のマクロ呼び出しを使用すると、

```
        CSTR    ABCD
```

展開結果は次のようになります。

```
        ASCII   'ABCD'
```

引用符が付いた文字列のDEFINE展開

文字シーケンスが、引用符の付いた文字列に含まれる場合、**DEFINE**擬似命令で作成されたシンボルと一致する一連の文字は展開されません。アセンブラの文字列は一般的に一重引用符(')で囲みます。文字列が二重引用符(")で囲まれている場合、**DEFINE**シンボルはその文字列内で展開されます。他の点では、二重引用符は、一重引用符と同じです。

例：

次のようなソースがあると仮定します。

```
        DEFINE  LONG  'short'
STR_MAC MACRO   STRING
        MSG     'This is a LONG STRING'
        MSG     "This is a LONG STRING"
        ENDM
```

このマクロが次のように呼び出された場合、

```
        STR_MAC sentence
```

展開結果は次のようになります。

```
        MSG     'This is a LONG STRING'
        MSG     'This is a short sentence'
```

@

関数区切り文字

すべてのアセンブラ組み込み関数は、最初に**@**シンボルが付いています。これらの関数の詳細については、"2.4.4 関数"を参照してください。

例：

```
SVAL      EQU   @ABS(VAL)     ; Obtain absolute value
```

*****

ロケーションカウンタの置換

アスタリスクは、式のオペランドとして使用された場合、ランタイムロケーションカウンタの現在の整数値を示します。

例：

```
          DEFSECT ".CODE", CODE  AT 100H
          SECT    ".CODE"
  XBASE   EQU     *+20H   ; XBASE = 120H
```

[]

ロケーションアドレッシングモード演算子

角かっこは、ロケーションアドレッシングモードを使用するようアセンブラに指示するときに使用します。

例：

```
  LD      A,[_Value]
```

#

即値アドレッシングモード

ポンド記号(#)は、即値アドレッシングモードを使用するようアセンブラに指示するときに使用します。

例：

```
  CNST    EQU     5H
          LD      A,#CNST   ;Load A with the value 5H
```

2.3.3 レジスタ

次のS1C88レジスタ名は、大文字小文字を問わず、アセンブラ言語ソースファイルでシンボル名として使用することはできません。

A
B
BA
H
L
HL
NB
EP
XP
YP
BR
IX
IY
SC
PC
SP

2.3.4 その他の特殊な名前

S1C88命令セットで使用されている以下の名前は、大文字でも小文字でも同様ですが、アセンブラ言語のソースファイルでシンボル名として使用することはできません。

C
T
LT
LE
GT
GE
V
NV
NC
NT
P
M
F0
F1
F2
F3
NF0
NF1
NF2
NF3

2.4 オペランドと式

2.4.1 オペランド

オペランドは、命令コードの後に付けられる命令の一部分です。オペランドは、1つの命令で1つまたは2つ使用することができますが、まったく使用しないこともあります。アセンブラ命令のオペランドは、次のいずれかの型になります。

オペランド	説明
expr	"式"の節で説明している有効な式。
reg	"レジスタ"の節で説明している有効なレジスタ。
symbol	等式で作成されたシンボル名。シンボルは式にすることもできます。
address	expr、reg、symbolの組み合わせ。

式がアセンブル時に完全に評価できる場合、その式は絶対式と呼ばれます。完全に評価できない場合は、再配置可能式と呼ばれます。詳細については、"2.4.2 式"を参照してください。

2.4.1.1 オペランドおよびアドレッシングモード

S1C88アセンブラ言語には、いくつかのアドレッシングモードがあります。以下では、これらのモードについて簡単に説明しています。詳細については、"S1C88コアCPUマニュアル"を参照してください。

レジスタ直接

命令が、オペランドアドレスを含むレジスタを指定します。

構文： *mnemonic register*

レジスタ間接

この命令は、オペランドアドレスを入れるレジスタを指定します。いくつかの形式があります。

構文： *mnemonic* [RR]
mnemonic [RR + *off*]
mnemonic [RR + L]

即値

即値オペランドは、1バイトの数値か1ワードの数値になり、命令の一部としてコード化されます。即値オペランドは、オペランドの値を定義する式の前に置かれる#記号で示されます。

構文： *mnemonic* #*number*

絶対

この命令には、オペランドアドレスが含まれます。アドレスは8ビットまたは16ビットにすることができます。

構文： *mnemonic* [*direct_address*]

PC相対

命令に、現在のPC値との相対オフセット(8ビットまたは16ビット)が含まれます。

構文： *mnemonic offset*

暗示

命令が、使用されているレジスタを暗黙的に定義します。

構文： *mnemonic*

2.4.2 式

アセンブラ命令または擬似命令のオペランドは、アセンブラシンボル、レジスタ名、式のいずれかになります。式は、シンボルのシーケンスで、特定のメモリ空間のアドレスまたは数値を指定します。

アセンブル時に評価できる式は、"絶対式"と呼ばれます。すべてのセクションが組み合わされてロケートされるまで結果が未知の式は、"再配置可能式"と呼ばれます。式のオペランドが再配置可能な場合、式全体が再配置可能になります。再配置可能式は、オブジェクトファイルに書き出され、リンカまたはロケータによって評価されます。再配置可能式は、整関数のみを含むことができます。オブジェクトの作成時に非IEEEの再配置可能式が見つかった場合、エラーが発生します。

式の型は、式の識別子の型に従属します。詳細については、"2.4.2.4 式の型"を参照してください。

アセンブラは、2の補数で64ビット精度の式を評価します。

式の構文は、次のいずれかになります。

- *number*
- *expression_string*
- *symbol*
- *expression binary_operator expression*
- *unary_operator expression*
- (*expression*)
- *function*

すべての式の型については、以下の節で説明します。

( ) 演算子の評価の順序を制御するときかっこを使用することができます。かっこ内の式が最初に評価されます。

例：

```
(3+4)*5 ; Result is 35. 3 + 4 is evaluated first.
3+(4*5) ; Result is 23. 4 * 5 is evaluated first.
        ; parentheses are superfluous here
```

2.4.2.1 数値

式では数値定数を使用することができます。接尾辞がない場合、アセンブラは、数値がデフォルトのRADIX形式になっていると見なします。その場合、デフォルトのRADIXは10進数です。

number 次のいずれかになります。
- *bin_num*B
- *dec_num*（または*dec_num*D）
- *oct_num*O（または*oct_num*Q）
- *hex_num*H

小文字のb、d、o、q、hを使用することもできます。

bin_num "0"と"1"で構成される2進値で、最後に"B"または"b"が付けられます。

例： `1001B; 1011B; 01100100b;`

dec_num "0"から"9"で構成される10進値で、オプションとして最後に"D"または"d"を付けることができます。

例： `12; 5978D;`

oct_num "0"から"7"で構成される8進値で、最後に"O"、"o"、"Q"、"q"のいずれかが付けられます。

例： `11O; 447o; 30146q`

hex_num "0"から"9"および"a"から"f"または"A"から"F"で構成される16進値で、最後に"H"または"h"が付けられます。最初の文字は数値でなければならないため、"a"から"f"または"A"から"F"で始まる16進数は、先頭に"0"を置く必要があります。

例： `45H; 0FFD4h; 9abcH`

RADIX擬似命令により入力基数が変更されている場合、数値は、後に基数標識を付けないで記述されます。たとえば、（最初の基数が10と仮定して）入力基数が16に設定されている場合、16進数は接尾辞**H**が付けられないで記述されます。デフォルトの基数は10になります。

2.4.2.2 式の文字列

*expression_string*は、数値に評価される任意の長さの文字列(*string*)です。文字列の長さは、0で埋められた左側の最初の4文字をとって計算されます。

string 一重引用符(')または二重引用符(")で囲まれたASCII文字です。最初の引用符と最後の引用符は同じでなければなりません。使用している引用符を文字列に入れるときは、それを繰り返して使用します。たとえば、両方の引用符を持つ文字列の場合、"'""または'''"'のように表記します。一重引用符が付いた文字列と二重引用符が付いた文字列の違いについては、"2.5 マクロ動作"を参照してください。

例：

```
'A'+1   ; a 1-character ASCII string, result 42H
"9C"+1  ; a 2-character ASCII string, result 3944H
```

2.4.2.3 シンボル

*symbol*は"識別子"です。*symbol*は、すでに定義されている識別子、または、ラベル宣言や等式の擬似命令により現在のソースモジュールで定義される識別子の値を示しています。

例：

```
CON1 EQU 3H     ; The variable CON1 represents the value of 3
LD A,[CON1+20H] ; Load A with contents of address 23H
```

アセンブラを起動した時点で、次の定義済みのシンボルが存在します。

_AS88 アセンブラの名前を持つ文字列("as88")が含まれます。

_MODEL 選択したMODEL(小文字)のASCII値を持つ整数が含まれます。

2.4.2.4 式の型

式の型は、数値(整数)またはアドレスになります。式の結果の型は、演算子およびオペランドによって異なります。以下の表は、すべての演算子をまとめたものです。

次の点に注意してください。

1. ラベルは"アドレス"型になります。等式のシンボルは、等式の型になります。
2. 型が宣言されていないシンボルの型は、コンテキストによってアドレスか数値になります。演算の結果は、以下の表に基づいて判断することができます。
3. 2値論理および関係演算子(||、&&、==、!=、<、<=、>、>=)は、任意の組み合わせのオペランドを受け付け、結果は常に整数値0か1になります。
4. 2進数のシフトおよびビット単位の演算(<<、>>、|、&、^)は、整数オペランドのみを受け付けます。

次の表は、単項演算子を持つ式(unary_operator expression)の結果の型を示します(*は不正な組み合わせを示す)。

表2.4.2.4.1 式の型、単項演算子

演算子	整数	アドレス
~	整数	*
!	整数	*
-	整数	*
+	整数	整数

次の表は、2項演算子を持つ式の結果の型を示します。

表2.4.2.4.2 式の型、2項演算子

演算子	整数, 整数	アドレス, 整数	整数, アドレス	アドレス, アドレス
-	整数	アドレス	*	整数
+	整数	アドレス	アドレス	*
*	整数	*	*	*
/	整数	*	*	*
%	整数	*	*	*

注： 文字列オペランドは、整数値に変換されます。

次の表は、関数(function)の結果の型を示します。"オペランド"の列にある"-"は、その関数にオペランドがないことを示しています。

表2.4.2.4.3 式の型、関数

関数	オペランド	結果
@ABS()	整数	整数
@ARG()	シンボル 整数	整数 整数
@AS88()	―	文字列
@CADDR()	整数, アドレス	アドレス
@CAT()	文字列, 文字列	文字列
@CNT()	―	整数
@COFF()	アドレス	アドレス
@CPAG()	アドレス	整数
@DADDR()	整数, アドレス	アドレス
@DEF()	シンボル	整数
@DOFF()	アドレス	アドレス
@DPAG()	アドレス	整数
@HIGH()	アドレス	整数
@LEN()	文字列	整数
@LOW()	アドレス	整数
@LST()	―	整数
@MAC()	シンボル	整数
@MAX()	整数, 整数, ...	整数
@MIN()	整数, 整数, ...	整数
@MODEL()	―	整数
@MXP()	―	整数
@POS()	文字列, 文字列 文字列, 文字列, 整数	整数 整数
@SCP()	文字列, 文字列	整数
@SGN()	整数	整数
@SUB()	文字列, 整数, 整数	文字列

2.4.3 演算子

演算子には、次の2つの種類があります。

- 単項演算子
- 2項演算子

演算子は、算術演算子、シフト演算子、関係演算子、ビット単位演算子、論理演算子にすることができます。すべての演算子について、以降の節で説明します。

演算子のグループ化がかっこで指定されていない場合、評価の順序を決定するために演算子の順位が使用されます。すべての演算子には、対応する順位レベルがあります。次の表では、演算子とその順位をリストしています(降順)。

表2.4.3.1 演算子の順位リスト

演算子	型
+, -, ~, !	単項
*, /, %	2項
+, -	2項
<<, >>	2項
<, <=, >, >=	単項
==, !=	2項
&	2項
^	2項
|	2項
&&	2項
||	2項

単項演算子以外の場合、同じ順位レベルの演算子を持つ式は左から右方向に評価されます。単項演算子の場合は、右から左方向に評価されます。そのため、`-4+3*2`は、`(-4)+(3*2)`のように評価されます。

2.4.3.1 加算と減算

構文：

加算： *operand* **+** *operand*

減算： *operand* **-** *operand*

+演算子は、2つのオペランドを加え、-演算子は引きます。オペランドには、無名数またはリロケータブルオペランドに評価される任意の式を使用することができます。ただし表2.4.2.4.2の制限はあります。

例：

```
0A342H + 23H    ; addition of absolute numbers
0FF1AH - AVAR   ; subtraction with the value of symbol AVAR
```

2.4.3.2 記号演算子

構文：

プラス： **+***operand*

マイナス： **-***operand*

+演算子は、そのオペランドを変更しません。-演算子は、そのオペランドを0から引きます。表2.4.2.4.1の制限も参照してください。

例：

```
5+-3  ; result is 2
```

2.4.3.3 乗算と除算

構文：

乗算： *operand* ***** operand
除算： *operand* **/** operand
モジュロ： *operand* **%** operand

*演算子は、2つのオペランドをかけます。/演算子は、整数の除算を実行し、余りを破棄します。%演算子も整数の除算を実行しますが、商を破棄して余りを返します。オペランドには、無名数またはリロケータブルオペランドに評価される任意の式を使用することができます。ただし表2.4.2.4.2の制限はあります。また、/演算子および%演算子の右側のオペランドは0にすることができません。

例：

```
AVAR*2          ; multiplication
0FF3CH/COUNT    ; division
23%4            ; modulo, result is 3
```

2.4.3.4 シフト演算子

構文：

左へシフト： *operand* **<<** *count*
右へシフト： *operand* **>>** *count*

これらの演算子は、左のオペランド(*operand*)を、右オペランド(*count*)で指定されたビット数(無名数)分左(<<)または右(>>)にシフトします。オペランドには、(整)数に評価される任意の式を使用することができます。

例：

```
AVAR>>4 ; shift right variable AVAR, 4 times
```

2.4.3.5 関係演算子

構文：

等号： *operand* **==** *operand*
不等号： *operand* **!=** *operand*
より小： *operand* **<** *operand*
以下： *operand* **<=** *operand*
より大： *operand* **>** *operand*
以上： *operand* **>=** *operand*

これらの演算子は、そのオペランドを比較して、"真"の場合1、"偽"の場合0の無名数(整数)を返します。オペランドには、無名数またはリロケータブルオペランドに評価される任意の式を使用することができます。

例：

```
3>=4            ; result is 0 (false)
4==COUNT        ; 1 (true), if COUNT is 4.
                ; 0 otherwise.
9<0AH           ; result is 1 (true)
```

2.4.3.6 ビット単位演算子

構文：

ビット単位AND： *operand* **&** *operand*
ビット単位OR： *operand* **|** *operand*
ビット単位XOR： *operand* **^** *operand*
1の補数： **~** *operand*

AND、OR、XORの各演算子は、左と右のオペランドにビット単位のAND、OR、XORをそれぞれとります。1の補数(ビット単位NOT)演算子は、そのオペランドについてビット単位の補数をとります。オペランドには、(整)数に評価される任意の式を使用することができます。

例：

```
0BH&3   ; result is 3
                1011B
                0011B  &
                0011B

~0AH    ; result is 0FFF5H
                ~ 00000000 00001010B
                = 11111111 11110101B
```

2.4.3.7 論理演算子

構文：

論理AND： *operand* **&&** *operand*
論理OR： *operand* **||** *operand*
論理NOT： **!** *operand*

論理AND演算子は、両方のオペランドが0でない場合整数1を返し、それ以外の場合整数0を返します。論理OR演算子は、オペランドのどちらかが0でない場合整数1を返し、それ以外の場合整数0を返します。!演算子は、オペランドについて論理否定を実行します。!は、オペランドが0の場合整数1("真")を返し、それ以外の場合整数0("偽")を返します。それぞれのオペランドには、整数に評価される任意の式を使用することができます。

例：

```
0BH&&3  ; result is 1 (true)

!0AH    ; result is 0 (false)
!(4<3)  ; result is 1 (true)
        ; 4 < 3 result is 0 (false)
```

2.4.4 関数

アセンブラには、データ変換、文字列比較、数値計算をサポートする組み込み関数が用意されています。関数は、任意の式で項として使用することができます。関数は次の構文になります。

@function_name(*argument*[,*argument*]...)

関数の最初には"@"記号が付き、後にかっこが続けられます。このとき、引数を付けることもあります(複数指定する場合もある)。関数名と左かっこの間、および引数と引数の間(コンマ区切り)には、スペースを入れません。

アセンブラの関数は、次の5つの種類に分類することができます。

1. 数学関数
2. 文字列関数
3. マクロ関数
4. アセンブラモード関数
5. アドレス操作関数

2.4.4.1 数学関数

数学関数には、MIN/MAX関数などがあります。

ABS - 絶対値。
MAX - 最大値。
MIN - 最小値。
SGN - 記号を返します。

2.4.4.2 文字列関数

文字列関数は、文字列を比較する他、文字列の長さを返したり、文字列内の副文字列の位置を返したりします。

CAT - 文字列を連結します。
LEN - 文字列の長さ。
POS - 文字列内の副文字列の位置。
SCP - 文字列を比較します。
SUB - 文字列から副文字列を抽出します。

2.4.4.3 マクロ関数

マクロ関数は、マクロに関する情報を返します。

ARG - マクロ引数関数。
CNT - マクロ引数の数。
MAC - マクロ定義関数。
MXP - マクロ展開関数。

2.4.4.4 アセンブラモード関数

アセンブラの動作に関連するさまざまな関数です。

AS88 - アセンブラの実行可能ファイル名。
DEF - シンボル定義関数。
LST - LIST擬似命令のフラグ値。
MODEL - 選択したアセンブラのモデル。

2.4.4.5 アドレス操作関数

特定のアドレス演算を処理する関数です。

CADDR - コードアドレス
COFF - コードページオフセット
CPAG - コードページ番号
DADDR - データアドレス
DOFF - データページオフセット
DPAG - データページ番号
HIGH - 256バイトページ番号
LOW - 256バイトページオフセット

2.4.4.6 詳細な説明

次に、アセンブラ関数のそれぞれについて詳細に説明します。それぞれの説明では、使用法のガイドライン、機能の説明、例などを紹介します。

@ABS (*expression*)

*expression*の無名数を整数値として返します。

例：
```
LD    A,#@ABS(VAL)      ;load absolute value
```

@ARG (*symbol* | *expression*)

*symbol*または*expression*で示されているマクロ引数がある場合整数1を返し、それ以外の場合0を返します。引数がシンボルの場合、一重引用符で囲んで、ダミー引数名を参照するようにする必要があります。引数が式の場合、マクロダミー引数リスト内の引数の順序の位置を参照するようにします。マクロ展開がアクティブでない場合にこの関数が使用されると、警告が出されます。

例：
```
IF    @ARG('TWIDDLE')   ;twiddle factor provided?
```

@AS88 ()

アセンブラの実行可能ファイルの名前を返します。S1C88 Familyの場合、これは**as88**になります。

例：
```
ANAME:  DB  @AS88()   ;ANAME = 'as88'
```

@CADDR (*code-page*,*code-offset*)

コードページ(32Kバンク)およびページオフセット(32Kオフセット)で指定されたコードアドレスを返します。*code-offset*が再配置可能な場合、生成される値も再配置可能な式になります。*code-offset*が絶対値の場合、結果は定数値になります。

例：
```
CAZERO  SET     @CADDR(3,8004h) ;CAZERO = 18004h
CAONE   SET     @CADDR(3,5000h) ;CAONE  =  5000h
```

@CAT (*str1*,*str2*)

2つの文字列を1つの文字列に連結します。2つの文字列は、一重引用符または二重引用符で囲む必要があります。

例：
```
DEFINE  ID  "@CAT('S1C','88')"     ;ID = 'S1C88'
```

@CNT ()

現在のマクロ展開引数の数を整数で返します。マクロ展開がアクティブでない場合にこの関数が使用されると、警告が出されます。

例：

```
ARGCNT  SET  @CNT()    ;squirrel away arg count
```

@COFF (*address*)

指定されたアドレスのコードページオフセット(32Kオフセット)を返します。*address*が再配置可能な場合、生成される値も再配置可能な式になります。*address*が絶対値の場合、結果は定数値になります。アドレスが最初のコードページ(最初の32K)にある場合、生成される結果のビット16(MSB)が0になります。その他のケースでは、結果のビット16(MSB)は1になります。

例：

```
PAGEZERO SET    @COFF(07FFFH)   ; PAGEZERO = 07FFFH
PAGEONE  SET    @COFF(0CFFFH)   ; PAGEONE  = 0CFFFH
PAGETWO  SET    @COFF(014FFFH)  ; PAGETWO  = 0CFFFH
```

@CPAG (*address*)

指定されたアドレスのコードページ(32Kバンク)を返します。*address*が再配置可能な場合、生成される値も再配置可能な式になります。*address*が絶対値の場合、結果は定数値になります。

例：

```
ZEROPAGE SET    @CPAG(07FFFH)   ;ZEROPAGE = 0H
ONEPAGE  SET    @CPAG(0CFFFH)   ;ONEPAGE  = 1H
TWOPAGE  SET    @CPAG(014FFFH)  ;TWOPAGE  = 2H
```

@DADDR (*data-page*,*data-offset*)

データページ(64Kバンク)およびページオフセット(64Kオフセット)で指定されたデータアドレスを返します。*data-offset*が再配置可能な場合、生成される値も再配置可能な式になります。*data-offset*が絶対値の場合、結果は定数値になります。

例：

```
DATHREE SET     @DADDR(3,1234h) ;DATHREE = 31234h
```

@DEF (*symbol*)

*symbol*が定義されている場合整数1を返し、それ以外の場合0を返します。*symbol*は、**MACRO**擬似命令に対応しない任意のラベルにすることができます。*symbol*に引用符が付いている場合、**DEFINE**シンボルと見なされ、引用符が付いていない場合は、通常のラベルと見なされます。

例：

```
IF    @DEF('ANGLE')     ;assemble if ANGLE defined
```

@DOFF (*address*)

指定されたアドレスのデータページオフセット(64Kオフセット)を返します。*address*が再配置可能な場合、生成される値も再配置可能な式になります。*address*が絶対値の場合、結果は定数値になります。

例：

```
PAGEZERO SET    @DOFF(07FFFH)   ;PAGEZERO = 07FFFH
PAGEONE  SET    @DOFF(0CFFFH)   ;PAGEONE  = 0CFFFH
PAGETWO  SET    @DOFF(014FFFH)  ;PAGETWO  = 04FFFH
```

@DPAG (*address*)

指定されたアドレスのデータページ(64Kバンク)を返します。*address*が再配置可能な場合、生成される値も再配置可能な式になります。*address*が絶対値の場合、結果は定数値になります。

例：

```
ZEROPAGE SET    @DPAG(07FFFH)   ;ZEROPAGE = 0H
ONEPAGE  SET    @DPAG(0CFFFH)   ;ONEPAGE  = 0H
TWOPAGE  SET    @DPAG(014FFFH)  ;TWOPAGE  = 1H
```

@HIGH (*address*)

指定されたアドレスの256バイトページ番号を返します。*address*が再配置可能な場合、結果の値は再配置可能式になります。*address*が絶対アドレスの場合、結果は定数値になります。

例：

```
HPAGE   SET     @HIGH(07FFFH)   ;HPAGE = 07FH
```

@LEN (*string*)

*string*の長さを整数で返します。

例：

```
SLEN    SET     @LEN('string')  ;SLEN = 6
```

@LOW (*address*)

指定されたアドレスの256バイトページオフセットを返します。*address*が再配置可能な場合、結果の値は再配置可能式になります。*address*が絶対アドレスの場合、結果は定数値になります。

例：

```
LPAGE   SET     @LOW(07FFFH)    ;LPAGE = 0FFH
```

@LST ()

LIST擬似命令のフラグの値を整数として返します。アセンブラソースに**LIST ON**擬似命令が見つかった場合、フラグがインクリメントされます。**LIST OFF**擬似命令が見つかった場合、フラグはデクリメントされます。

例：

```
DUP     @ABS(@LST())            ;list unconditionally
```

@MAC (*symbol*)

*symbol*がマクロ名として定義されている場合整数1を返し、それ以外の場合0を返します。

例：

```
IF      @MAC(DOMUL)             ;expand macro
```

@MAX (*expr1*[,*exprN*]...)

*expr1,...,exprN*のうち最大値を整数で返します。

例：

```
MAX:    DB      @MAX(1,5,-3)    ;MAX = 5
```

@MIN (*expr1*[,*exprN*]...)

*expr1,...,exprN*のうち最小値を整数で返します。

例：

```
MIN:    DB      @MIN(1,5,-3)    ;Min = -3
```

@MODEL ()

選択したアセンブラのモデルを返します(コマンド行またはMODELコントロールで指定したもの)。戻り値は、選択したMODEL(常に小文字)のASCII文字値になります。

例(**-Ms**オプションの場合)**:**

```
MDL     SET     @MODEL()        ;MDL = 73h (ASCII value of 's')
```

@MXP ()

アセンブラがマクロを展開している場合整数1を返し、それ以外の場合0を返します。

例:

```
IF      @MXP()                  ;macro expansion active?
```

@POS (*str1*,*str2*[,*start*])

*start*の位置から数えた、*str1*内の*str2*の位置を整数値で返します。*start*が指定されていない場合、*str1*の最初の位置から数えられます。*start*引数が指定されている場合、その値は正の整数でなければならず、またソース文字列の長さを越えることもできません。

例:

```
ID      EQU     @POS('S1C88','88')      ;ID = 3
```

@SCP (*str1*,*str2*)

2つの文字列を比較して同じ値になっている場合整数1を返し、それ以外の場合0を返します。2つの文字列はコンマで区切る必要があります。

例:

```
IF      @SCP(STR,'MAIN')        ;does STR equal MAIN?
```

@SGN (*expression*)

*expression*の記号を整数値で返します。引数が負の場合-1、0の場合0が返され、正の場合1が返されます。*expression*は相対式にすることも絶対式にすることもできます。

例:

```
IF      @SGN(INPUT) == 1        ;is sign positive?
```

@SUB (*str*,*expr1*,*expr2*)

*str*から副文字列を返します。*expr1*が、*str*内の開始位置、*expr2*が、返される文字列の長さになります。*expr1*または*expr2*のいずれかが*str*の長さを越えた場合、エラーが発生します。

例:

```
DEFINE  ID      "@SUB('S1C88',3,2)"     ;ID = '88'
```

2.5 マクロ動作

2.5.1 はじめに

この節では、マクロ動作と条件アセンブラについて説明します。
このアセンブラには、マクロプリプロセッサが実装されています。

2.5.2 マクロ動作

アプリケーションを頻繁にプログラミングする場合、一定の命令のパターンやグループを繰り返しコーディングすることになります。これらのパターンの中には、変数エントリがあり、パターンを使用するたびに変動するものがあります。また、命令グループが発生するときの条件によって、条件アセンブラされるように設定されたものもあります。どちらの場合も、マクロを使用すると、これらの命令パターンを操作するときに簡単な表記を使用できるようになります。繰り返しパターンを決定したら、プログラマはマクロを使用して、変数として選択した文のフィールドを指定します。その後プログラマは、マクロを起動することにより、パターン全体を必要に応じて何度も使用し、文で指定した変数の位置に異なるパラメータを使用することができます。

パターンを定義するとき、名前を指定します。この名前は、その後マクロが起動される(呼び出される)ときに使用されるニーモニックになります。マクロの名前が、既存のアセンブラ擬似命令やニーモニック命令コードと同じ場合、マクロが擬似命令やニーモニック命令コードと置き換えられ、警告が出されます。

マクロを呼び出すと、ソース文が生成されます。生成される文には、置換可能な引数を入れることができます。マクロ呼び出しで生成される文は、種類という点ではあまり制限を受けません。任意のプロセッサ命令、ほとんどすべてのアセンブラ擬似命令、すでに定義されているマクロなどを使用することができます。マクロ呼び出しによって生成されたソース文は、プログラマが記述した文に当てはまるものと同じ条件および制限を受けることになります。

マクロを起動するとき、マクロ名を、ソース文の命令コードフィールドに記述します。引数はオペランドフィールドに置きます。マクロ定義で示されている使用法に対応する引数を適切に選択することにより、マクロ定義のインラインコードの変化形が生成されます。

マクロを呼び出すと、結果的に、定義済みの関数を実行するインラインコードが生成されます。これらのコードは通常のプログラムの流れの中に挿入されるため、マクロが呼び出されるたびに、生成された命令はプログラムの他の部分と一緒に実行されます。

マクロを定義する上で重要な機能として、マクロ定義でマクロ呼び出しを使用するというものがあります。アセンブラはこのような"ネスト"されたマクロ呼び出しを展開時にのみ処理します。他の定義内にネストできるマクロ定義は1つだけに限られます。ただしネストされたマクロ定義は、主マクロが展開されるまで処理されません。またマクロは、ソース文の命令フィールドに現れる前に定義しておく必要があります。

2.5.3 マクロ定義

マクロの定義は、ヘッダ(マクロに名前を割り当てダミー引数を定義)、本体(プロトタイプまたはスケルトンソース文で構成される)、終止符の3つの部分で構成されます。ヘッダは、**MACRO**擬似命令、その名前、ダミー引数リストで構成されます。本体には、標準ソース文のパターンが含まれます。終止符は**ENDM**擬似命令になります。

マクロ定義のヘッダは次のような書式になります。

macro_name **MACRO** [*dummy argument list*] [*comment*]

名前は必須で、マクロが呼び出されるときに使用されるシンボルにします。ダミー引数リストは次のような書式になります。

[*dumarg*[,*dumarg*]...]

ダミー引数は、マクロが展開される(呼び出される)ときにマクロプロセッサが引数と置換するシンボル名になります。それぞれのダミー引数は、グローバルシンボル名と同じ規則に従う必要があります。ダミー引数は、コンマで区切ります。

マクロ呼び出しが実行されるとき、マクロ定義内のダミー引数(以下の例ではNMUL、AVEC、BVEC、OFFSET、RESULT)は、マクロ呼び出しで定義されている対応する引数と置換されます。

ローカルラベル演算子を使用する、マクロ内のすべてのローカルラベル定義は、このマクロ呼び出しで固有のものになります。これは、すべてのローカルラベルに固有の接尾辞を追加して、ラベルの範囲をモジュールでローカルにすることにより実行されます。このメカニズムにより、プログラマは、マクロが展開される回数を気にせずに、マクロ定義内で自由にローカルラベルを使用できるようになります。ローカルラベル演算子が付いていないラベルは、通常のラベルと見なされるため、**SET**擬似命令と一緒に使用されていない限り、複数回使用することはできません("2.6 アセンブラ擬似命令"を参照)。

例:

次のマクロは、

```
N_R_MUL MACRO   NMUL,AVEC,BVEC,OFFSET,RESULT    ;header
        LD      B,#NMUL                         ;body
        LD      IX,#AVEC
        LD      IY,#BVEC
^again: LD      L,[IX+OFFSET]
        LD      A,[IY+OFFSET]
        MLT
        ADD     A,[RESULT]
        LD      [RESULT],A
        INC     IX
        INC     IY
        DJR     NZ,^again
        ENDM                                    ;terminator
        N_R_MUL 10H,_obj1,_obj2,10H,_RESULT
```

次のように展開されます(againおよび_RESULTの処理が異なる)。

```
        LD      B,#10H
        LD      IX,#_obj1
        LD      IY,#_obj2
again__M_L000001:
        LD      L,[IX+10H]
        LD      A,[IY+10H]
        MLT
        ADD     A,[_RESULT]
        LD      [_RESULT],A
        INC     IX
        INC     IY
        DJR     NZ,again__M_L000001
```

2.5.4 マクロ呼び出し

マクロが起動されるとき、アクションを起こす文は、マクロ呼び出しと呼ばれます。マクロ呼び出しの構文は、次のフィールドで構成されます。

[*label*:] *macro_name* [*arguments*] [*comment*]

引数(*arguments*)フィールドは、次のような書式になります。

[*arg*[,*arg*]...]

マクロ呼び出し文は、*label*フィールド、命令フィールド、オペランドフィールドの3つのフィールドとコメントフィールドで構成されます。*label*フィールドがある場合、このフィールドには、マクロ展開の開始時点でのロケーションカウンタ値が入ります。命令フィールドには、マクロ名が入り、オペランドフィールドには、置換用の引数が入ります。オペランドフィールドでは、マクロ呼び出しのそれぞれの呼び出し引数が、マクロ定義のダミー引数と1対1で対応します。たとえば、前の節で定義したN_R_MULマクロが、次の文で展開(呼び出し)されるとします。

```
N_R_MUL     CNT+1,VEC1,VEC2,OFFS,OUT
```

ただし、オペランドフィールド引数はコンマで区切られ、左から右の順で、NMULからRESULTまでのダミー引数にそれぞれ対応します。これらの引数は、対応する定義の位置で置換され、命令のシーケンスが生成されます。

マクロ引数は、コンマで区切られた文字のシーケンスで構成されます。マクロ引数文字列は、引用符の付いた文字列として指定することもできますが、必ずしも一重引用符を付ける必要はありません。これは、コーディングを簡単にするための配慮によるものです。ただし、引数にコンマやスペースが組み込まれている場合、その引数は一重引用符(')で囲む必要があります。マクロを呼び出すとき、引数を付けずに宣言することもできます。ただし、その場合明示的にNULLであることを宣言する必要があります。NULL引数は、次の4つの方法で指定することができます。

- 間にスペースを入れないで、区切りコンマを続けて記述する。
- 引数リストの最後にコンマを入れ、他の引数リストを入れない。
- 引数をNULL文字列で宣言する。
- 一部またはすべての引数を省略する。

NULL引数を使用すると、引数を参照する文が生成されるときも、文字が置換されません。マクロ呼び出しで指定された引数の数が、マクロ定義よりも多い場合は、警告が出されます。

2.5.5 ダミー引数演算子

アセンブラのマクロプロセッサでは、マクロ展開のときにテキスト置換を行います。引数置換機能に柔軟性を持たせるため、アセンブラでは、引数テキストの変換を行うマクロ定義内で特定のテキスト演算子を認識するようになっています。これらの演算子は、テキストの連結、数値変換、文字列操作で使用することができます。

2.5.5.1 ダミー引数連結演算子 - \

他の文字と連結するためのダミー引数の場合、その前に連結演算子"**¥**"(または"\")を付けて、他の文字と区別する必要があります。引数は隣接するテキストの前または後ろにつなげることができますが、連結演算子と他の文字との間にスペースを入れないようにします。2つの英数字の間に引数を置くとき、引数名の前後に"**¥**"(または"\")を置きます。たとえば、次のマクロ定義があると仮定します。

```
SWAP_MEM MACRO  REG1,REG2       ;swap memory contents
         LD     A,[I¥REG1]      ;using A as temp
         LD     B,[I¥REG2]      ;using B as temp
         LD     [I¥REG1],B
         LD     [I¥REG2],A
         ENDM
```

このマクロが次の文で呼び出された場合、

```
         SWAP_MEM X,Y
```

マクロ展開のときに、マクロプロセッサがダミー引数REG1と文字"X"を置換し、ダミー引数REG2と文字"Y"を置換します。連結演算子(**¥**)は、マクロプロセッサに対して、ダミー引数に対応する置換文字を、文字Iが付いている両方の場合について連結するよう伝えます。このマクロの展開結果は、次のようになります。

```
         LD    A,[IX]
         LD    B,[IY]
         LD    [IX],B
         LD    [IY],A
```

2.5.5.2 戻り値演算子 - ?

もう1つのマクロ定義演算子に、シンボルの値を返す疑問符(**?**)があります。マクロプロセッサがこの演算子を見つけると、**?***symbol*シーケンスを、*symbol*の10進値を示す文字列に変換します。たとえば、上記のSWAP_MEMマクロを次のように変更したと仮定します。

```
SWAP_MEM MACRO  REG1,REG2       ;swap memory contents
         LD     A,[_lab¥?REG1]  ;using A as temp
         LD     B,[_lab¥?REG2]  ;using B as temp
         LD     [_lab¥?REG1],B
         LD     [_lab¥?REG2],A
         ENDM
```

ソースファイルが、次のSET文とマクロ呼び出しを持っている場合、

```
AREG     SET      1
BREG     SET      2
         SWAP_MEM AREG,BREG
```

イベントは次のように発生します。マクロプロセッサは最初にREG1を文字AREGに変換し、REG2を文字BREGに変換します。わかりやすくするため、中間的なマクロ展開を次に示します(ソースリストにはこのような形で書き出されることはない)。

```
         LD    A,[_lab¥?AREG]
         LD    B,[_lab¥?BREG]
         LD    [_lab¥?AREG],B
         LD    [_lab¥?BREG],A
```

マクロプロセッサは、その次に**?**AREGを文字Xで、**?**BREGを文字Yで置換します。これは、XがシンボルAREGの値で、YがシンボルBREGの値であるためです。これによって生成される中間的な展開は次のようになります。

```
        LD    A,[_lab¥1]
        LD    B,[_lab¥2]
        LD    [_lab¥1],B
        LD    [_lab¥2],A
```

次に、マクロプロセッサは、連結演算子(\)を適用します。ソースリストに書き出される展開結果は、次のようになります。

```
        LD    A,[_lab1]
        LD    B,[_lab2]
        LD    [_lab1],B
        LD    [_lab2],A
```

2.5.5.3 戻り16進値演算子 - %

パーセント記号(**%**)は、標準的な戻り値演算子とほとんど同じですが、シンボルの16進値を返すという点で異なっています。マクロプロセッサがこの演算子を見つけると、**%***symbol*シーケンスを、*symbol*の16進値を示す文字列に変換します。次のマクロ定義があると仮定します。

```
GEN_LAB    MACRO   LAB,VAL,STMT
LAB¥%VAL:  STMT
           ENDM
```

このマクロは、ラベルの接頭辞引数と、16進値に変換される値を連結して、ラベルを生成します。このマクロが、次のように呼び出されたと仮定します。

```
NUM        SET     10
           GEN_LAB HEX,NUM,'NOP'
```

マクロプロセッサは、最初にLABを文字HEXで置換し、次に文字Aを**%**VALで置換します。これは、Aが10進整数10の16進表現であるためです。次にマクロプロセッサは、連結演算子(\)を適用します。最後にSTMTが文字列'NOP'で置換されます。リストファイルに書き出される展開結果は、次のようになります。

```
HEXA:   NOP
```

パーセント記号は、2進定数を示すときに使用される文字にもなります。マクロ内で2進定数が必要な場合、定数をかっこで囲むか、パーセント記号の後に"\"を付けて定数をエスケープします。

2.5.5.4 ダミー引数文字列演算子 - "

もう1つのダミー引数演算子に、二重引用符(")があります。この文字は、マクロプロセッサによって一重引用符で置き換えられますが、その後の文字は、ダミー引数名としてチェックされます。マクロを呼び出すと、結果的に、引用符で囲まれたダミー引数がリテラル文字列に変換されます。たとえば、次のマクロ定義があると仮定します。

```
STR_MAC MACRO   STRING
        ASCII   "STRING"
        ENDM
```

このマクロが次のマクロ展開行で呼び出された場合、

```
        STR_MAC ABCD
```

マクロ展開の結果は次のようになります。

```
        ASCII   'ABCD'
```

二重引用符を使用すると、引用符の付いた文字列内で、**DEFINE**擬似命令を展開できるようにもなります。二重引用符にはこのような性質があるため、マクロ定義で不適切な展開が発生しないよう注意する必要があります。**DEFINE**展開はマクロ置換の前に出現するため、**DEFINE**シンボルは、最初にマクロのダミー引数文字列内で置換されます。

```
          DEFINE  LONG  'short'
STR_MAC   MACRO   STRING
          MSG     'This is a LONG STRING'
          MSG     "This is a LONG STRING"
          ENDM
```

このマクロを次のように起動した場合、

```
          STR_MAC sentence
```

展開結果は、次のようになります。

```
          MSG     'This is a LONG STRING'
          MSG     'This is a short sentence'
```

2.5.5.5 マクロのローカルラベル演算子 - ^

マクロ本体内でローカルアドレスを参照するために使用されるマクロ引数として、名前を渡したいような場合があります。アクセント記号(^)が識別子の前にある場合、マクロプリプロセッサは、マクロ展開結果でラベルがそのまま使用されるようにするため、そのラベルの名前を変更します。例を示します。

```
LOAD      MACRO   ADDR
          LD      A,[^ADDR]
          ENDM
```

マクロの^演算子は、ADDR引数で名前の変更を実行します。次のマクロ呼び出しがあると仮定します。

```
_LOCAL: LOAD      _LOCAL
```

マクロ定義のローカルラベルでは、マクロLOADが次のように展開されます。

```
_LOCAL:
          LD      A,[_LOCAL__M_L000001]
```

ラベル_LOCAL__M_L000001はどこにも定義されていないため、これにより、アセンブラエラーが発生します。(上記のような)ローカルラベル演算子がマクロ定義にない場合、マクロLOADは次のように展開されます。

```
_LOCAL:
          LD      A,[_LOCAL]
```

この場合、正しくアセンブルされます。

2.5.6 DUP、DUPA、DUPC、DUPF擬似命令

DUP、**DUPA**、**DUPC**、**DUPF**擬似命令は、専門化したマクロ形式です。これらは、命名されていないマクロの同時定義および呼び出しと考えることができます。**DUP**、**DUPA**、**DUPC**、**DUPF**擬似命令と**ENDM**擬似命令の間にあるソース文は、マクロ定義と同じ規則に従います。上記のダミー演算子の文字についても同様です(**DUPA**、**DUPC**、**DUPF**擬似命令の場合)。これらの擬似命令の詳細については、"2.6 アセンブラ擬似命令"を参照してください。

2.5.7 条件アセンブラ

条件アセンブラは、さまざまな条件をカバーする包括的なソースプログラムを記述できるようにするものです。アセンブラの条件は、マクロの場合引数を使用し、その他の場合**DEFINE**、**SET**、**EQU**擬似命令でシンボルの定義によって指定することができます。パラメータの変動により、特定の条件で必要な部分のみアセンブルすることができます。また、アセンブラの組み込み関数を使用すれば、アセンブル環境のさまざまな条件を柔軟な方法でテストすることができます(アセンブラの組み込み関数の詳細については"2.4.4 関数"を参照)。

マクロ定義内では、展開時に引数が有効範囲内に収まるようにするために、条件擬似命令を使用することもできます。この方法を使用すれば、マクロはセルフチェックするようになり、一定レベルになった時点でエラーメッセージが表示されるようにすることもできます。

条件アセンブラ擬似命令**IF**は、次のような書式になります。

```
IF      expression
 .
 .
[ELSE]  ;(the ELSE directive is optional)
 .
 .
ENDIF
```

条件に従ってアセンブルされるプログラムのセクションは、**IF-ENDIF**擬似命令で囲まれています。オプションの**ELSE**擬似命令がない場合、**IF**擬似命令から次の**ENDIIF**擬似命令までのソース文は、*expression*が0以外の場合に限り、アセンブルされるソースファイルの一部としてインクルードされます。*expression*の値が0の場合、ソースファイルは、**IF**擬似命令と**ENDIF**擬似命令の間の文が存在しないかのようにアセンブルされます。**ELSE**擬似命令が存在し、*expression*が0以外になる場合、**IF**擬似命令と**ELSE**擬似命令の間の文がアセンブルされ、**ELSE**擬似命令と**ENDIF**擬似命令の間の文がスキップされます。また*expression*が0になる場合、**IF**擬似命令と**ELSE**擬似命令の間の文がスキップされ、**ELSE**擬似命令と**ENDIF**擬似命令の間の文がアセンブルされます。

2.6 アセンブラ擬似命令

2.6.1 概要

アセンブラ擬似命令は、アセンブラのプロセスをコントロールするために使用されます。アセンブラ擬似命令は、S1C88マシン命令に変換されるのではなく、アセンブラによって解釈されます。擬似命令は、アセンブラコントロール、リストコントロールなどのアクションを実行する他、シンボルの定義やロケーションカウンタの変更を行います。アセンブラ擬似命令の場合、大文字と小文字は同一視されます。

アセンブラ擬似命令は、機能によって次の5つの種類に分けることができます。

1. デバッグ
2. アセンブラコントロール
3. シンボル定義
4. データ定義/記憶域の割り当て
5. マクロおよび条件アセンブラ

2.6.1.1 デバッグ

コンパイラは、アセンブラ経由で高レベル言語のシンボリックデバッグ情報をオブジェクトファイルに渡すとき、次の擬似命令を生成します。

CALLS - 呼び出し情報をオブジェクトファイルに渡します。オーバーレイセクションを重ね書きするためにリンク時に呼び出しツリーを構築するときに使用されます。

SYMB - シンボリックデバッグ情報を渡します。

2.6.1.2 アセンブラコントロール

アセンブラコントロールに使用される擬似命令には、次のようなものがあります。

ALIGN - 調整を指定します。

COMMENT - コメント行を開始させます。この擬似命令は、IF/ELIF/ELSE/ENDIF構成体およびMACRO/DUP定義では使用できません。

DEFINE - 置換文字列を定義します。

DEFSECT - セクション名と属性を定義します。

END - ソースプログラムの終了を示します。

FAIL - プログラマが生成するエラーメッセージです。

INCLUDE - 二次ファイルをインクルードします。

MSG - プログラマが生成するメッセージです。

RADIX - 定数に対する入力基数を変更します。

SECT - セクションを起動します。

UNDEF - **DEFINE**シンボルの定義を解除します。

WARN - プログラマが生成する警告です。

2.6.1.3 シンボル定義

シンボル定義をコントロールするときに使用される擬似命令には、次のようなものがあります。

EQU - シンボルと値を等しいものとして定義します。順方向の参照を受け付けます。
EXTERN - 外部シンボル宣言です。モジュールの本体でも使用できます。
GLOBAL - グローバルシンボル宣言です。モジュールの本体でも使用できます。
LOCAL - ローカルシンボル宣言です。
NAME - オブジェクトファイルを識別します。
SET - シンボルに値を設定します。順方向の参照を受け付けます。

2.6.1.4 データ定義/記憶域の割り当て

定数データ定義と記憶域の割り当てをコントロールするときに使用される擬似命令には、次のようなものがあります。

ASCII - ASCII文字列を定義します。
ASCIZ - NULLを埋め込んだASCII文字列を定義します。
DB - 定数バイトを定義します。
DS - 記憶域を定義します。
DW - 定数ワードを定義します。

2.6.1.5 マクロおよび条件アセンブラ

マクロおよび条件アセンブラで使用される擬似命令には、次のようなものがあります。

DUP - ソース行のシーケンスを複製します。
DUPA - 引数を付けてシーケンスを複製します。
DUPC - 文字を付けてシーケンスを複製します。
DUPF - ループでシーケンスを複製します。
ENDIF - 条件アセンブラの終了。
ENDM - マクロ定義の終了。
EXITM - マクロを終了させます。
IF - 条件アセンブラ擬似命令。
MACRO - マクロ定義。
PMACRO - マクロ定義を消去します。

2.6.2 ALIGN擬似命令

構文：

ALIGN *expression*

説明：

ロケーションカウンタを調整します。*expression*は、2^kの値で表現する必要があります。デフォルトの調整は、バイト単位で行われます。*expression*は、正の値でなければなりません。*expression*が2^kの値でない場合、警告が出され、次の2^k値で揃えられます。調整は、**align**擬似命令を記述している場所で一度だけ実行されます。セクションの最初では、自動的に、そのセクションで使用される最大の調整値に調整されます。

セクションの種類により、この擬似命令の動作は、2つの状況に分けられます。

- リロケータブルセクション

  このセクションの場合、計算された調整境界で調整されます。このセクションの現在の相対ロケーションカウンタによっては、ギャップが生成されます。

- 絶対セクション

  セクションのロケーションは変更されません。現在の絶対アドレスに従ってギャップが生成されます。

例：

```
ALIGN 4          ;align at 4 bytes
lab1: ALIGN 6    ;not a 2^k value.
                 ;a warning is issued
                 ;lab1 is aligned on 8 bytes
```

2.6.3 ASCII擬似命令

構文：

[*label*:] **ASCII** *string* [,*string*]...

説明：

ASCII文字のリストを定義します。**ASCII**擬似命令は、それぞれの*string*引数に対してメモリの配列を割り当て、初期化します。配列の最後にNULLバイトが追加されることはありません。そのため動作は、文字列引数を持つDB擬似命令と同じになります。

参照：

ASCIZ、**DB**

例：

```
HELLO:  ASCII   "Hello world"   ;Is the same as DB "Hello world"
```

2.6.4 ASCIZ擬似命令

構文：

[*label*:] **ASCIZ** *string* [,*string*]...

説明：

ASCII文字のリストを定義します。**ASCIZ**擬似命令は、それぞれの*string*引数に対してメモリの配列を割り当て、初期化します。配列の最後にはNULLバイトが追加されます。

参照：

ASCII、**DB**

例：

```
HELLO:  ASCIZ   "Hello world"   ;Is the same as DB "Hello world",0
```

2.6.5 CALLS擬似命令

構文：

CALLS '*caller*', '*callee*'[, *nr*]...[, '*callee*'[, *nr*]...]...

説明：

*caller*と*callee*の間にフローグラフ参照を作成します。リンカで、オーバーレイアルゴリズムを動作させるフローグラフを作成するときにこの情報が必要になります。*caller*と*callee*は関数名です。

スタック情報も指定されます。*callee*の名前の後に、スタック単位(システムスタック、ユーザスタックなど)で、使用回数を指定することができます。指定された値(*nr*)が、呼び出し時のスタック使用回数(S1C88の場合バイト単位)になります(現在の関数の"RET"アドレスを含む)。現在、S1C88ツールでは、システムスタックのみを使用します。

この情報は、リンカが、アプリケーション内で使用されるスタックを計算するときに使用します。この情報は、呼び出しグラフで生成されるリンカマップファイル(**-M**オプション)にあります。

*callee*の名前を指定しない場合、関数自身のスタック使用回数を定義することになります。

"2.2.3.1 セクション名"、および"3.4 リンカの出力"も参照してください。

例：

```
DEFSECT "OVLN@nfunc", DATA, OVERLAY, SHORT
DEFSECT "OVLN@main", DATA, OVERLAY, SHORT
CALLS   'main', 'nfunc', 5
```

2.6.6 COMMENT擬似命令

構文：

COMMENT *delimiter*

:

delimiter

説明：

コメント行を開始させます。**COMMENT**擬似命令は、1行または複数行をコメントとして定義するときに使用します。**COMMENT**擬似命令の後に付く最初のブランク以外の文字が、コメント区切り文字になります。コメントテキストを定義するときは、2つの区切り文字が使用されます。2番目のコメント区切り文字を含む行が、コメントの最終行と見なされます。コメントテキストには、任意のプリント可能文字を入れることができ、ソースファイルに出力されるときに、ソースリストで生成されるようにすることもできます。

この擬似命令では、ラベルを使用することができません。

この擬似命令は、IF/ELSE/ENDIF構成体およびMACRO/DUP定義では使用できません。

例：

```
COMMENT + This is a one line comment +
COMMENT *     This is a multiple line comment. Any number of lines
              can be placed between the two delimiters
        *
```

2.6.7 DB擬似命令

構文：

[*label*:] **DB** *arg*[, *arg*]...

説明：

定数バイトを定義します。**DB**擬似命令は、それぞれの*arg*引数に対して1バイトのメモリを割り当て、初期化します。*arg*には、数値定数、単一または複数の文字列定数、シンボル、式のいずれでも指定できます。**DB**擬似命令は、1つ以上の引数をとることができますが、複数の引数を付ける場合、これらをコンマで区切ります。複数の引数は、連続するアドレスロケーションに格納されます。複数の引数がある場合、それらのどれについてもNULLにすることができます(2つのコンマを連続して使用)。その場合、対応するアドレスロケーションに0が入れられます。評価された引数値が、1バイトで表現できない長さになる場合、エラーが発生します。
*label*が指定されている場合、擬似命令の処理を開始する時点でランタイムロケーションカウンタの値が割り当てられます。
整数引数は、そのまま格納されますが、バイト値でなければなりません(たとえば0～255の範囲内)。単一文字または複数文字による文字列は、次のような方法で処理されます。

1. 単一文字の文字列は、1バイトに格納され、その各ビットが文字のASCII値を示します。
   例： `'R' = 52H`
2. 複数文字の文字列の場合、その文字列を構成する文字のASCII表現によるバイトの集まりになります。
   例： `'ABCD' = 41H, 42H, 43H, 44H`

参照：

DS、**DW**

例：

```
TABLE:   DB  14,253,62H,'ABCD'
CHARS:   DB  'A','B','C','D'
```

2.6.8 DEFINE擬似命令

構文：

DEFINE *symbol* *string*

説明：

置換文字列を定義します。**DEFINE**擬似命令は、それ以降のすべてのソース行で使用される置換文字列を定義するときに使用します。それ以降のすべての行で*symbol*が検索され、*string*で置換されます。この擬似命令は、ソースプログラムをわかりやすく記述するときに使用すると便利です。*symbol*は、ラベルの制限に従う必要があります。つまり、最初の文字は英字またはアンダースコア(_)で、それ以降の文字が英数字またはアンダースコア(_)でなければなりません。すでに定義されているシンボルを新しく定義しようとすると、警告が発生します。マクロの場合は、特殊な状況になります。**DEFINE**擬似命令による変換は、マクロ定義が見つかった時点で適用されます。また、マクロが展開されるとき、その時点でアクティブな**DEFINE**擬似命令変換が再度適用されます。
この擬似命令では、ラベルを使用することができません。

参照：

UNDEF

例：

次の**DEFINE**擬似命令が、ソースプログラムの最初の部分で見つかった場合、

```
        DEFINE    ARRAYSIZ    '10 * SAMPLSIZ'
```

次のソース行は、

```
        DS        ARRAYSIZ
```

アセンブラによって次のように変換されます。

```
        DS        10 * SAMPLSIZ
```

2.6.9 DEFSECT擬似命令

構文：

DEFSECT *section, type* [, *attr*]...[**AT** *address*]

説明：

セクションの名前と宣言属性を定義するとき、この擬似命令を使用します。コードやデータをセクションに置く前に、SECT擬似命令を使用して、そのセクションを起動しておく必要があります。定義には、宣言属性を入れることができ、セクションの型(*type*)を指定する必要があります。

セクションの型は、次のいずれかになります。

type： **DATA**｜**CODE**

セクションの宣言属性は、次のいずれかになります。

attr： グループ1： **SHORT**｜**TINY**
グループ2： **FIT 100H**｜**FIT 8000H**｜**FIT 10000H**
グループ3： **OVERLAY**｜**ROMDATA**｜**NOCLEAR**｜**CLEAR**｜**INIT**｜**MAX**
グループ4： **JOIN**

各グループはせいぜい1つの属性が指定されます。**CLEAR**セクションは、スタートアップ時に0になります。この属性は、DATA型のセクションでのみ使用することができます。

NOCLEAR属性を持つセクションは、スタートアップ時に0に設定されません。これは、DATAセクションのデフォルトです。この属性は、DATAセクションのみ使用することができます。

INIT属性は、DATAセクションに初期化データが含まれることを定義します。この初期化データは、プログラムのスタートアップ時にROMからRAMにコピーされます。

OVERLAY属性を指定すると、セクションが重ね書き可能になります。重ね書き可能なのは、DATAセクションのみです。

(DATA、CODEセクションで使用できる)**ROMDATA**セクションには、ROM内で置換するデータが含まれます。このROM領域は実行可能ではありません。

同じ名前を持つDATAセクションが、**MAX**属性を持ち、さまざまなオブジェクトモジュールにある場合、リンカは、それぞれのオブジェクトモジュールのうち最大のサイズを、生成するセクションのサイズとして採用します。**MAX**属性は、DATAセクションのみ適用されます。

SHORT属性は、セクションを最初の32K(共通の非バンク領域)内に置かなければならないCODEセクションを指定します。**SHORT**属性がDATAセクションで使用される場合、そのセクションが最初の64Kページ内になければならないことを指定します。

TINY属性は、セクションがデータメモリの最初の64K内の最大256バイトページ、1ページ内になければならないことを指定します。

FIT属性は、セクションが一定の境界を越えないことを指定します。そのため、ここで指定したサイズが、そのセクションで使用可能な最大サイズになります(ただしリンカは、同じ名前を持つすべてのセクションを一緒にリンクしてから、生成されたセクションについてサイズをチェックする)。そのため、たとえば`FIT 8000H`セクションは、`0`から`7FFFH`の範囲、または`8000H`と`0FFFFH`の範囲内にロケートすることができます。アドレス`8000H`や`10000H`を越えて配置することはできません。

JOIN属性を使用すると、複数のセクションを1ページ内にグループ化することができます。**JOIN**属性は常に、**FIT**属性と一緒に使用しなければなりません。たとえば、複数のセクションを同じデータページ内にロケートする必要がある場合に、この属性を使用します。"2.2.3.3 グループ化セクション"も参照してください。

セクション、セクションの型、セクション属性などの詳細については、"2.2.3.1 セクション名"を参照してください。

参照：

SECT

例：

```
DEFSECT ".text", DATA   ;declare section .text
SECT    ".text"         ;switch to section .text
```

2.6.10 DS擬似命令

構文：

[*label*:] **DS** *expression*

説明：

記憶域を定義します。**DS**擬似命令は、*expression*の値と同じバイト数分、メモリのブロックを予約します。この擬似命令を使用すると、オペランドフィールドにある絶対整数式の値の分だけランタイムロケーションカウンタが進められます。予約されたメモリのブロックは、どの値にも初期化されることはありません。式は正の整数にならなければならず、アドレスラベルに対する順方向の参照(まだ定義されていないラベル)も入れることができません。

*label*が指定されている場合、擬似命令の処理を開始する時点でランタイムロケーションカウンタの値が割り当てられます。

参照：

DB、**DW**

例：

```
S_BUF:  DS      12      ; Sample buffer
```

2.6.11 DUP擬似命令

構文：

[*label*:] **DUP** *expression*

:

ENDM

説明：

ソース行のシーケンスを複製します。**DUP**擬似命令と**ENDM**擬似命令の間にあるソース行のシーケンスは、整数*expression*で指定された数だけ複製されます。式が0以下の値に評価された場合、その行のシーケンスは、アセンブラ出力に入れられません。式の結果は、絶対整数にならなければならず、アドレスラベルに対する順方向の参照(まだ定義されていないラベル)も入れることができません。**DUP**擬似命令は、いくらでもネストすることができます。

*label*が指定されている場合、**DUP**擬似命令の処理を開始する時点でランタイムロケーションカウンタの値が割り当てられます。

参照：

DUPA、**DUPC**、**DUPF**、**ENDM**、**MACRO**

例：

次のようなソース入力文のシーケンスがあると仮定します。

```
COUNT   SET     3
        DUP     COUNT   ; SRA BY COUNT
        SRA     A
        ENDM
```

その場合、次のソースリストが生成されます。

```
COUNT   SET     3
        DUP     COUNT   ; SRA BY COUNT
        SRA     A
        ENDM
;       SRA     A
;       SRA     A
;       SRA     A
```

2.6.12 DUPA擬似命令

構文：

[*label*:] **DUPA** *dummy*, *arg*[, *arg*]...

:

ENDM

説明：

引数を付けてシーケンスを複製します。**DUPA**擬似命令と**ENDM**擬似命令で定義されたソース文のブロックは、それぞれの引数で繰り返されます。繰り返されるたびに、ブロック内のダミーパラメータが、その後の引数文字列で置換されます。引数文字列がNULLの場合、ダミーパラメータが削除されてブロックが繰り返されます。引数に、組み込みブランクやアセンブラの特殊文字が含まれる場合、一重引用符で囲む必要があります。

*label*が指定されている場合、**DUPA**擬似命令の処理を開始する時点でランタイムロケーションカウンタの値が割り当てられます。

参照：

DUP、**DUPC**、**DUPF**、**ENDM**、**MACRO**

例：

次のような文を持つ入力ソースファイルがあると仮定します。

```
        DUPA    VALUE,12,32,34
        DB      VALUE
        ENDM
```

その場合、アセンブラソースリストは次のようになります。

```
        DUPA    VALUE,12,32,34
        DB      VALUE
        ENDM
;       DB      12
;       DB      32
;       DB      34
```

2.6.13 DUPC擬似命令

構文：

[*label*:] **DUPC** *dummy*, *string*

:

ENDM

説明：

文字を付けてシーケンスを複製します。**DUPC**擬似命令と**ENDM**擬似命令で定義されたソース文のブロックは、*string*のそれぞれの文字で繰り返されます。繰り返されるたびに、ブロック内のダミーパラメータが、その後にある文字列の文字で置換されます。文字列がNULLの場合、ブロックがスキップされます。

*label*が指定されている場合、**DUPC**擬似命令の処理を開始する時点でランタイムロケーションカウンタの値が割り当てられます。

参照：

DUP、**DUPA**、**DUPF**、**ENDM**、**MACRO**

例：

次のような文を持つ入力ソースファイルがあると仮定します。

```
        DUPC    VALUE,'123'
        DB      VALUE
        ENDM
```

その場合、アセンブラソースリストは次のようになります。

```
        DUPC    VALUE,'123'
        DB      VALUE
        ENDM

;       DB      1
;       DB      2
;       DB      3
```

2.6.14 DUPF擬似命令

構文：

[*label*:] **DUPF** *dummy*, [*start*], *end*[, *increment*]

:

ENDM

説明：

ループでシーケンスを複製します。**DUPF**擬似命令と**ENDM**擬似命令で定義されたソース文のブロックは、*increment*が1の場合、一般的に(*end* - *start*) + 1回繰り返されます。ただし*start*はループインデックスの開始値で、*end*はループインデックスの終了値になります。*increment*は、ループインデックスの増分で、省略された場合、デフォルトで1になります(*start*値も同様)。*dummy*パラメータは、ループインデックス値を保持し、命令の本文内で使用されます。

*label*が指定されている場合、**DUPF**擬似命令の処理を開始する時点でランタイムロケーションカウンタの値が割り当てられます。

参照：

DUP、**DUPA**、**DUPC**、**ENDM**、**MACRO**

例：

次のような文を持つ入力ソースファイルがあると仮定します。

```
        DUPF    NUM,0,3
        LD      [NUM],A
        ENDM
```

その場合、アセンブラソースリストは次のようになります。

```
        DUPF    NUM,0,3
        LD      [NUM],A
        ENDM

;       LD      [0],A
;       LD      [1],A
;       LD      [2],A
;       LD      [3],A
```

2.6.15 DW擬似命令

構文：

[*label*:] **DW** *arg*[, *arg*]...

説明：

定数ワードを定義します。**DW**擬似命令は、それぞれの*arg*引数に対して1ワードのメモリを割り当て、初期化します。*arg*には、数値定数、単一または複数の文字列定数、シンボル、式のいずれでも指定できます。**DW**擬似命令は、1つ以上の引数をとることができますが、複数の引数を付ける場合、これらをコンマで区切ります。複数の引数は、連続するアドレスロケーションに格納されます。複数の引数がある場合、それらのどれについてもNULLにすることができます(2つのコンマを連続して使用)。その場合、対応するアドレスロケーションに0が入れられます。評価された引数値が、1ワードで表現できない長さになる場合、エラーが発生します。

*label*が指定されている場合、擬似命令の処理を開始する時点でランタイムロケーションカウンタの値が割り当てられます。

ワード値は、最下位のアドレスに下位ビットとともにメモリに格納されます。

整数の引数はそのまま格納されます。単一文字または複数文字による文字列は、次のような方法で処理されます。

1. 単一文字の文字列は、1ワードに格納され、その下位7ビットが文字のASCII値を示します。
   例： `'R' = 52H`
2. 3文字以上で構成される複数文字の文字列は使用できません。2文字の文字列は、最初の文字のASCII値がワードの上位バイト値であるかのように格納されます。2番目の文字は、下位バイトとして使用されます。
   例： `'AB' = 4142H`

参照：

DB、**DS**

例：

```
TABLE:  DW      14,1635,2662H,'AB'
```

上記の文は、次の文と同じです。

```
TABLE:  DB      14,0,1635%256,6,62H,26H,'B','A'
```

2.6.16 END擬似命令

構文：

END

説明：

ソースプログラムの終了を示します。オプションの**END**擬似命令は、ソースプログラムの論理的な終了を示します。**END**擬似命令は、マクロ展開では使用できません。

この擬似命令では、ラベルを使用することができません。

例：

```
END     ;End of source program
```

2.6.17 ENDIF擬似命令

構文：

ENDIF

説明：

条件アセンブラの終了を示します。**ENDIF**擬似命令は、条件アセンブラの現在のレベルが終了することを示します。条件アセンブラ擬似命令はいくらでもネストできますが、**ENDIF**擬似命令が参照するのは、常に直前の**IF**擬似命令になります。この擬似命令では、ラベルを使用することができません。

参照：

IF

例：

```
IF      DEB             ;Report building of the debug version
MSG     'Debug Version'
ENDIF
```

2.6.18 ENDM擬似命令

構文：

ENDM

説明：

マクロ定義の終了を示します。**MACRO**、**DUP**、**DUPA**、**DUPC**の各擬似命令の最後には、**ENDM**擬似命令を付ける必要があります。この擬似命令では、ラベルを使用することができません。

参照：

DUP、**DUPA**、**DUPC**、**MACRO**

例：

```
SWAP_MEM MACRO  REG1,REG2       ;swap memory contents
         LD     A,[I¥REG1]      ;using A as temp
         LD     B,[I¥REG2]      ;using B as temp
         LD     [I¥REG1],B
         LD     [I¥REG2],A
         ENDM
```

2.6.19 EQU擬似命令

構文：

name **EQU** *expression*

説明：

シンボルと値を等しいものとして定義します。**EQU**擬似命令は、*expression*の値をシンボル*name*に割り当てます。**EQU**擬似命令は、プログラムカウンタ以外の値を名前に割り当てる擬似命令です。このシンボル名は、プログラムの他の場所で再定義することはできません。*expression*は相対でも絶対でも可能で、順方向の参照を受け付けます。

EQUシンボルはグローバルにすることもできます。

参照：

SET

例：

```
A_D_PORT  EQU   4000H
```

この文は、値4000Hをシンボル A_D_PORTに割り当てます。

2.6.20 EXITM擬似命令

構文：

EXITM

説明：

マクロを終了させます。**EXITM**擬似命令は、マクロ展開をすぐに終了させます。この擬似命令は、条件アセンブラ擬似命令**IF**と一緒に使用して、エラーの条件が検出された時点でマクロ展開を終了させる場合に使用すると便利です。

この擬似命令では、ラベルを使用することができません。

参照：

DUP、**DUPA**、**DUPC**、**MACRO**

例：

```
CALC    MACRO   XVAL,YVAL
        IF      XVAL<0
        MSG     'Macro parameter value out of range'
        EXITM   ;Exit macro
        ENDIF
         :
        ENDM
```

2.6.21 EXTERN擬似命令

構文：

EXTERN [(*attrib*[, *attrib*]...)] *symbol*[, *symbol*]...

説明：

外部シンボルの宣言です。**EXTERN**擬似命令は、現在のモジュールにあるシンボルのリストが参照されることを指定するときに使用しますが、現在のモジュール内では定義されません。これらのシンボルは、モジュールの外部で定義されているか、別のモジュール内で**GLOBAL**擬似命令を使用し、グローバルにアクセスできるものとして宣言されている必要があります。

オプションの引数*attrib*には、次のシンボル属性のいずれかを指定することができます。

CODE	シンボルがROM内にあります。
DATA	シンボルがRAM内にあります。
SHORT	シンボルがメモリの最初のページ内に置かれます。 CODEの場合最初の32Kになり、 DATAの場合最初の64Kになります。
TINY	シンボルがDATAの最初の64K内の最大256バイトページ、1ページ内に置かれます。

シンボルが外部で定義され、シンボルが現在のモジュール内で定義されないことを指定する場合に**EXTERN**擬似命令が使用されないとき、警告が出され、**EXTERN**シンボルが挿入されます。

この擬似命令では、ラベルを使用することができません。

参照：

GLOBAL

例：

```
EXTERN  AA,CC,DD         ;defined elsewhere
EXTERN  (CODE,SHORT) EE ;within first 32K of code memory
```

2.6.22 FAIL擬似命令

構文：

FAIL [{*str* | *exp*}[, {*str* | *exp*}]...]

説明：

プログラマが生成するエラーです。**FAIL**擬似命令は、アセンブラによってエラーメッセージが出されるようにするときに使用します。このとき、他のエラーの場合と同じように、エラーの回数の合計がインクリメントされます。**FAIL**擬似命令は通常、例外条件チェックのために、条件アセンブラ擬似命令と組み合わせて使用します。アセンブラは通常、エラーが出力された後も進行します。またオプションで、生成されるエラーの性質を記述するため、任意の数の文字列と式を、コンマ区切りで任意の順序で指定することができます。

この擬似命令では、ラベルを使用することができません。

参照：

MSG、**WARN**

例：

```
FAIL    'Parameter out of range'
```

2.6.23 GLOBAL擬似命令

構文：

GLOBAL *symbol*[, *symbol*]...

説明：

グローバルシンボルの宣言です。**GLOBAL**擬似命令は、シンボルのリストが現在のセクションまたはモジュールで定義されており、これらの定義がすべてのモジュールからアクセスできるようにすることを指定する場合に使用します。オペランドフィールドに入れるシンボルがそのモジュールで定義されていない場合、エラーが発生します。"グローバル"として定義されたシンボルは、他のモジュールから**EXTERN**擬似命令を使用してアクセスすることができます。

この擬似命令では、ラベルを使用することができません。

グローバルにできるのは、プログラムラベルと**EQU**ラベルのみです。

参照：

EXTERN、**LOCAL**

例：

```
GLOBAL  LOOPA   ;LOOPA will be globally
                ;accessible by other modules
```

2.6.24 IF擬似命令

構文：

IF *expression*

:

[**ELSE**]（**ELSE**擬似命令はオプション）

:

ENDIF

説明：

条件アセンブラ擬似命令。条件に応じてアセンブルされるプログラム部分は、**IF**擬似命令と**ENDIF**擬似命令で囲む必要があります。オプションの**ELSE**擬似命令がない場合、**IF**擬似命令から次の**ENDIF**擬似命令までのソース文が、*expression*が0以外の結果になった場合のみアセンブルされるソースファイル部分になります。*expression*の結果が0になる場合、**IF**擬似命令と**ENDIF**擬似命令の間にあるこれらの文がないかのように、ソースファイルがアセンブルされます。**ELSE**擬似命令があり、*expression*の結果が0以外の場合、**IF**擬似命令と**ELSE**擬似命令の間の文がアセンブルされ、**ELSE**擬似命令と**ENDIF**擬似命令の間の文はスキップされます。また、*expression*の結果が0の場合、**IF**擬似命令と**ELSE**擬似命令の間の文はスキップされ、**ELSE**擬似命令と**ENDIF**擬似命令の間の文がアセンブルされます。

*expression*の結果は、絶対整数になる必要があります。結果が0以外の場合に真と見なされます。*expression*は、結果が0の場合のみ偽と見なされます。擬似命令の性質のため、*expression*は1回のパスで認識される必要があります(順方向の参照は使用できない)。**IF**擬似命令はいくらでもネストすることができます。**ELSE**擬似命令は、**ENDIF**擬似命令と同じように、常に直前の**IF**擬似命令を参照します。

この擬似命令では、ラベルを使用することができません。

参照：

ENDIF

例：

```
IF      XVAL<0
MSG     'Please select larger value for XVAL'
ENDIF
```

2.6.25 INCLUDE擬似命令

構文：

INCLUDE *string* | <*string*>

説明：

二次ファイルをインクルードします。この擬似命令は、ソース入力ストリームで二次ファイルをインクルードしたい場所でソースプログラムに挿入します。*string*は、二次ファイルのファイル名を指定します。ファイル名は、オペレーティングシステムと互換性のあるものにし、ディレクトリ指定を入れることもできます。

<*string*>構文が使用されていない場合、ファイルは最初にカレントディレクトリまたは*string*で指定されたディレクトリで検索されます。ファイルが見つからない場合、アセンブラは、環境変数AS88INCで指定されているディレクトリを探します。最後に、**as88.exe**があるディレクトリの相対パス`..¥include`が探されます。

<string>構文が指定されている場合、*string*で指定されたディレクトリとカレントディレクトリは検索されません。AS88INCのディレクトリおよび相対パスは検索されます。

この擬似命令では、ラベルを使用することができません。

例：

```
INCLUDE 'storage¥mem.asm'
INCLUDE <data.asm>              ; Do not look in current directory
```

2.6.26 LOCAL擬似命令

構文：

LOCAL *symbol*[, *symbol*]...

説明：

ローカルセクションのシンボル宣言です。**LOCAL**擬似命令は、シンボルのリストが現在のモジュール内で定義されており、これらの定義が明示的にそのセクションまたはモジュールに対してローカルになっていることを指定するときに使用します。シンボルがモジュールの外部にエクスポートできないような状況で使用すると便利です(モジュール内のラベルはデフォルトで"グローバル"と定義されるため)。オペランドフィールドに入れるシンボルがそのモジュールで定義されていない場合、エラーが発生します。

この擬似命令では、ラベルを使用することができません。

参照：

GLOBAL

例：

```
LOCAL   LOOPA   ;LOOPA local to this module
```

2.6.27 MACRO擬似命令

構文：

name **MACRO** [*dummy argument list*]

:

macro definition statements

:

ENDM

説明：

マクロ定義です。ダミー引数リストは次のような書式になります。

[*dumarg*[, *dumarg*]...]

名前は必須で、マクロを呼び出すときのシンボルになります。

マクロの定義は、ヘッダ(マクロに名前を割り当てダミー引数を定義)、本体(プロトタイプまたは骨組ソース文で構成される)、終止符の3つの部分で構成されます。ヘッダは、**MACRO**擬似命令、その名前、ダミー引数リストで構成されます。本体には、標準ソース文のパターンが含まれます。終止符は**ENDM**擬似命令になります。

ダミー引数は、マクロが展開される(呼び出される)ときにマクロプロセッサが引数と置換するシンボル名になります。それぞれのダミー引数は、グローバルシンボル名と同じ規則に従う必要があります。3つのダミー引数フィールドのそれぞれで、ダミー引数をコンマで区切ります。ダミー引数フィールドは、1つ以上のスペースで区切ります。

マクロ定義はネストできますが、ネストされたマクロは、主マクロが展開されるまで定義されません。

"2.5 マクロ動作"で、マクロについて詳細に説明しています。

参照：

DUP、**DUPA**、**DUPC**、**DUPF**、**ENDM**

例：

```
SWAP_MEM MACRO  REG1,REG2       ;swap memory contents
         LD     A,[I¥REG1]      ;using A as temp
         LD     B,[I¥REG2]      ;using B as temp
         LD     [I¥REG1],B
         LD     [I¥REG2],A
         ENDM
```

2.6.28 MSG擬似命令

構文：

MSG [{*str* | *exp*}[, {*str* | *exp*}]...]

説明：

プログラマが生成するメッセージです。**MSG**擬似命令を使用すると、アセンブラによってメッセージが出されます。エラーと警告の回数は影響を受けません。**MSG**擬似命令は通常、通知の目的で、条件アセンブラ擬似命令と組み合わせて使用します。アセンブラは通常、メッセージが出力された後も進行します。またオプションで、メッセージの性質を記述するため、任意の数の文字列と式を、コンマ区切りで任意の順序で指定することができます。

この擬似命令では、ラベルを使用することができません。

参照：

FAIL、**WARN**

例：

```
MSG      'Generating tables'
```

2.6.29 NAME擬似命令

構文：

NAME "*str*"

説明：

NAME擬似命令は、生成されるオブジェクトファイルを識別するためにアセンブラが使用します。リンカとロケータは、この情報を使用して、マップファイル内のソースを識別します。またデバッガは、この値を"モジュール"名として表示することもできます。

この擬似命令が省略された場合、アセンブラはモジュールのソース名を識別子として使用します。コントロールプログラムを使用している場合、この名前は"ランダムな"名前になります。

例：

```
NAME    "strcat"        ;object is identified by the name "strcat"
```

2.6.30 PMACRO擬似命令

構文：

PMACRO *symbol*[, *symbol*]...

説明：

マクロ定義を消去します。指定されたマクロ定義がマクロテーブルから消去され、マクロテーブルの空間を再利用できるようになります。

この擬似命令では、ラベルを使用することができません。

参照：

MACRO

例：

```
PMACRO  MAC1,MAC2
```

この文は、MAC1およびMAC2という名前のマクロを消去します。

2.6.31 RADIX擬似命令

構文：

RADIX *expression*

説明：

定数に対する入力基数を変更します。定数の入力基数を*expression*の結果に変更します。絶対整数式は、有効な定数基数(2、8、10、16)に評価される必要があります。デフォルトの基数は10です。**RADIX**擬似命令を使用すると、プログラマが、基数標識を使用せずに定数を好みの基数で指定できるようになります。10の基数を持つ数値の基数の接尾辞は文字"D"です。基数を変更するために定数が使用されている場合、その定数は、**RADIX**擬似命令が見つかった時点で、適切な入力基数になっている必要があります。

この擬似命令では、ラベルを使用することができません。

例：

```
_RAD10: DB      10      ; Evaluates to hex A
        RADIX   2
_RAD2:  DB      10      ; Evaluates to hex 2
        RADIX   16D
_RAD16: DB      10      ; Evaluates to hex 10
        RADIX   3       ; Bad radix expression
```

2.6.32 SECT擬似命令

構文：

SECT "*str*"[, **RESET**]

説明：

SECT擬似命令は、*str*という名前を持つ他のセクションがアクティブになったことをアセンブラに通知します。セクションを初めて起動するときは、その前に**DEFSECT**擬似命令でそのセクションを定義しておく必要があります。それ以降は、**SECT**擬似命令のみで起動することができます。

セクション属性MAXを持つDATAセクションの記憶域割り当てのカウントをリセットするときは、セクション属性RESETを使用することができます。

セクションの詳細については、"2.2.3.1 セクション名"を参照してください。

参照：

DEFSECT

例：

```
DEFSECT ".text",DATA    ;declare section .text
SECT    ".text"         ;switch to section .text
```

2.6.33 SET擬似命令

構文：

name **SET** *expression*

説明：

シンボルに値を設定します。**SET**擬似命令は、オペランドフィールドの式の値をシンボル名に割り当てるときに使用します。**SET**擬似命令は、**EQU**擬似命令と同様の動作をします。ただし、**SET**擬似命令を使用して定義されたシンボルは、プログラムの別の部分で値を再定義することができます(ただし他の**SET**擬似命令を使用する場合に限られる)。**SET**擬似命令は、マクロ内で一時カウンタまたは再使用可能なカウンタを作成するときに使用すると便利です。**SET**のオペランドフィールドにある*expression*には、順方向の参照を入れることができます。

SETシンボルはグローバルにすることができません。

参照：

EQU

例：

```
COUNT   SET     0       ; Initialize COUNT
```

2.6.34 SYMB擬似命令

構文：

SYMB *string, expression*[*, abs_expr*] [*, abs_expr*]

説明：

SYMB擬似命令は、アセンブラ(およびリンカ/ロケータ)経由で、高レベル言語のシンボリックデバッグ情報をデバッガに渡すときに使用します。*expression*には、任意の式を置くことができます。*abs_expr*は、絶対値を生成する任意の式にすることができます。

SYMB擬似命令は、"手書き"のアセンブラファイルの作成を目的にしたものではありません。補完のみの目的で使用されるもので、ツールチェーンの"内部"で使用することが前提になっています。

2.6.35 UNDEF擬似命令

構文：

UNDEF *symbol*

説明：

DEFINEシンボルの定義を解除します。**UNDEF**擬似命令は、*symbol*に対応する置換文字列を解除します。これ以降*symbol*は、有効な**DEFINE**置換を示さなくなります。詳細については、**DEFINE**擬似命令を参照してください。

この擬似命令では、ラベルを使用することができません。

参照：

DEFINE

例：

```
UNDEF   DEBUG   ;Undefines the DEBUG substitution string
```

2.6.36 WARN擬似命令

構文：

WARN [{*str* | *exp*}[, {*str* | *exp*}]...]

説明：

プログラマが生成する警告です。**WARN**擬似命令は、アセンブラによって警告メッセージが出されるようにするときに使用します。このとき、他の警告の場合と同じように、警告の回数の合計がインクリメントされます。**WARN**擬似命令は通常、例外条件チェックのために、条件アセンブラ擬似命令と組み合わせて使用します。アセンブラは通常、警告が出力された後も進行します。またオプションで、生成される警告の性質を記述するため、任意の数の文字列と式を、コンマ区切りで任意の順序で指定することができます。

この擬似命令では、ラベルを使用することができません。

参照：

FAIL、**MSG**

例：

```
WARN    'parameter too large'
```

2.7 アセンブラのコントロール

2.7.1 はじめに

アセンブラのコントロールは、アセンブラのデフォルトの動作を変更するときに使用します。これらのコントロールは、ソースファイルの"コントロール行"で指定することができます。コントロール行は、ドル記号(**$**)が最初に付く行です。このような行は、通常のアセンブラソース行と同じように処理されず、アセンブラコントロール行として処理されます。コントロールは、1行のソース行につき1つずつ指定することができます。アセンブラのコントロール行には、コメントを入れることもできます。アセンブラの擬似命令では、大文字と小文字が同一視されます。

コントロールは、基本コントロールと一般コントロールに分けることができます。

"基本コントロール"は、アセンブラの動作全体に影響し、アセンブルの最中もずっと有効になっています。そのため、基本コントロールは、アセンブルが始まる前の、ソースファイルの最初のポイントでのみ使用することができます。基本コントロールを複数回指定すると、警告メッセージが出され、最後の定義のみが使用されます。このような性質のため、基本コントロールをコマンド行オプションで上書きすることができます。

"一般コントロール"は、アセンブルの間アセンブラをコントロールするときに使用します。一般コントロールを含むコントロール行は、ソースファイルの任意の場所に記述することができます。コマンド行で一般コントロールを指定すると、ソースファイル内の対応する一般コントロールが無視されます。

次のページでは、使用可能なアセンブラのコントロールをアルファベット順にリストします。コントロールには、デフォルトでセットされるものや、デフォルト値を持つものもあります。

2.7.2 アセンブラのコントロールの概要

表2.7.2.1 アセンブラのコントロール

コントロール	タイプ	デフォルト	説 明
$CASE ON $CASE OFF	基本	ON	すべてのユーザ名で大文字と小文字が区別されます。 ユーザ名で大文字と小文字が区別されません。
$IDENT LOCAL $IDENT GLOBAL	基本	LOCAL	デフォルトがローカルラベルになります。 デフォルトがグローバルラベルになります。
$LIST ON $LIST OFF	一般		リストを再開します。 リストを停止します。
$LIST "*flags*"	基本	cDEGlMnPQsWXy	リストファイルに入れるものと排除するものを定義します。
$MODEL [S|C|D|L]	基本	L	メモリモデルを選択します。異なるメモリモデルのオブジェクトファイルは、リンクできません。
$STITLE "*title*"	一般		次のページのリストページヘッダタイトルを設定します。
$TITLE "*title*"	基本	スペース	最初のページのリストページヘッダタイトルを設定します。
$WARNING OFF $WARNING OFF *num*	基本		すべての警告を抑制します。 1つの警告を抑制します。
タイプ: コントロールのタイプ。"基本"は基本コントロール、"一般"は一般コントロールを示します。			

2.7.3 アセンブラのコントロールの説明

2.7.3.1 CASE

コントロール：

$CASE ON

$CASE OFF

関連オプション：

-c 大文字と小文字の区別をオフにし、コントロールの設定を上書きします。

クラス：

基本

デフォルト：

$CASE ON

説明：

アセンブラが、大文字小文字を区別するモードで動作するかどうかを選択します。区別しないモードの場合、アセンブラは、入力時の文字をすべて大文字に変換します(リテラル文字列は除く)。

例：

```
;Begin of source
$case off       ;assembler in case insensitive mode
```

2.7.3.2 IDENT

コントロール：

$IDENT LOCAL

$IDENT GLOBAL

関連オプション：

-i[l | g] デフォルトラベルが、ローカルまたはグローバルになります。

クラス：

基本

デフォルト：

$IDENT LOCAL

説明：

$IDENTコントロールは、アセンブラがラベルを処理する方法について指定するときに使用します。対象になるのは、コードラベルとデータラベルのみです。$IDENT LOCALの場合、ラベルがデフォルトでローカルになり、$IDENT GLOBALの場合、ラベルがデフォルトでグローバルになります。

SET識別子は、常にローカルシンボルとして扱われます。

特定のラベルでLOCAL擬似命令またはGLOBAL擬似命令を使用することで、いつでもデフォルト設定を上書きすることができます。

例：

```
;Begin of source
$ident global   ; assembly labels are global by default
```

2.7.3.3 LIST ON/OFF

コントロール：

$LIST ON

$LIST OFF

関連オプション：

-l アセンブラリストファイルを生成します。

クラス：

一般

デフォルト：

$LIST ON

説明：

リスト生成のオンとオフを切り換えます。これらのコントロールは、次の行から有効になります。実際のリストファイルの生成は、コマンド行で選択されます。コマンド行オプション**-l**が指定されていない場合、リストファイルは生成されません。

例：

```
$list off       ; Turn listing off. These lines are
                ; not present in the list file
    :
$list on        ; Turn listing back on. These lines
                ; are present in the list file
    :
```

2.7.3.4 LIST

コントロール：

$LIST "*flags*"

関連オプション：

-L[*flag...*] 指定されたソース行をリストファイルから削除します。

クラス：

基本

デフォルト：

$LIST "cDEGIMnPQsWXy"

説明：

リストファイルから削除するソース行を指定します。文字列内で定義するフラグは、**-L**コマンド行オプションのフラグと同じものです。使用できるそれぞれのフラグについては、**-L**オプションを参照してください。

例：

```
;Begin of source
$list "cw"      ; Remove source lines with assembler controls from the
                ; resulting list file and remove wrapped source lines
    :
```

2.7.3.5 MODEL

コントロール：

$MODEL [S ¦ D ¦ C ¦ L]

関連オプション：

-M*model*　メモリモデルを指定します。

クラス：

基本

デフォルト：

$MODEL L

説明：

$MODELコントロールで、ソースがプロセッサを使用する方法を指定します。次のようなモデルを指定することができます。

モデル	説　明
S	スモールモデル。最大64Kのコードとデータ。
C	コンパクトコードモデル。最大64Kのコードと16Mのデータ。
D	コンパクトデータモデル。最大8Mのコードと64Kのデータ。
L	ラージモデル。最大8Mのコードと16Mのデータ。

つまり、たとえば、スモールモデルの場合、CB/NBレジスタの値をソースで変更することができません。またEP/XP/YPの各レジスタも固定しなければなりません。

異なるモデルでアセンブルされたオブジェクトファイルは、リンクすることができません。このコントロールは、異なるモデルでも、同じアプローチでページレジスタを使用できるようにするためのものです。$MODELコントロールは、このような方法でCコンパイラによって使用されますが、アセンブラのプログラマは、依然としてソース内で"不正な"モデルを選択することができます。そのような場合、動作しないプログラムが生成されます。

例：

```
;Begin of source
$model s
; assemble using the small model
```

2.7.3.6 STITLE

コントロール：

$STITLE "*title*"

関連オプション：

-l アセンブラリストファイルを生成します。

クラス：

一般

デフォルト：

$STITLE ""

説明：

プログラムのサブタイトルを初期化します。$STITLEコントロールは、プログラムのサブタイトルを、オペランドフィールドのtitleに初期化します。サブタイトルは、他の$STITLEコントロールが指定されるまで、それ以降のすべてのページの上部にプリントされます。サブタイトルは最初ブランクになっています。$STITLEコントロールは、ソースリストにはプリントされません。$STITLEコントロールに文字列引数を付けないで指定した場合、現在のサブタイトルがブランクになります。

ページ幅が小さすぎてタイトルがヘッダに入りきらない場合、タイトルは切り捨てられます。

参照：

TITLE

例：

```
$stitle "Demo title"

; title in page header on succeeding pages
; is Demo title
```

2.7.3.7 TITLE

コントロール：

$TITLE "*title*"

関連オプション：

-l アセンブラリストファイルを生成します。

クラス：

基本

デフォルト：

スペース

説明：

このコントロールは、リストファイルの最初のページにあるページヘッダのタイトルを*title*で指定します。

ページ幅が小さすぎてタイトルがヘッダに入りきらない場合、切り捨てられます。

参照：

STITLE

例：

```
;Begin of source
$title "NEWTITLE"

; title in page header on first page is NEWTITLE
```

2.7.3.8 WARNING

コントロール：

$WARNING OFF

$WARNING OFF *num*

関連オプション：

-w[*num*]　1つまたはすべての警告を抑制します。

クラス：

基本

デフォルト：

-（すべての警告が有効）

説明：

$WARNINGは、すべての警告を抑制します。これは**-w**と同じです。$WARNING OFF *num*は、1つの警告メッセージを抑制します。この場合、*num*は警告メッセージ番号になります（**-w** *num*オプションと同じ）。

例：

```
;Begin of source
$warning off       ; switch all warning off
```

3 リンカ

3.1 概要

ここでは、S1C88およびその派生製品用のプログラムをリンクするプロセスについて説明します。S1C88のリンカ実行可能ファイルの名前は、**lk88**です。

リンカは、アセンブラが生成した複数のリロケータブルオブジェクトファイルを結合して、1つの新しいリロケータブルオブジェクトファイルを生成します(推奨拡張子.out)。このファイルは、その後のリンカ呼び出しでも入力として使用することができます。つまりリンクプロセスは、インクリメンタルにすることができます。通常リンカは、解決されない外部参照があると、それについて警告を出します。インクリメンタルリンクの場合、解決されない参照が出力ファイルに書き出されるのが普通です。そのため、インクリメンタルリンクは、個別に選択する必要があります。

リンカは、通常のオブジェクトファイルとオブジェクトモジュールのライブラリを読み込むことができます。ライブラリのモジュールは、参照される場合に限りインクルードされます。リンクプロセスの最終段階では、未解決の参照がないファイルが生成されます。これがロードモジュールと呼ばれます。

S1C88リンカは、オーバーレイリンカです。コンパイラは、重ね書き可能なセクションを生成します。重ね書き可能なセクションには、変数(Cレベルでは、ある関数に対してローカル)用に予約されたスペースが含まれます。2つの関数がそれぞれを互いに呼び出さない場合、そこで使用されるローカル変数はメモリ内で重ね書きすることができます。関数呼び出し情報を結合して呼び出しグラフにする作業と、この呼び出しグラフの構造に基づいて重ね書きできるセクションを判定する作業は、リンカが担当します(このとき最小限のRAMを使用)。

インクリメンタルリンクの場合は重ね書きが無効になるため、インクリメンタルフェーズがすべての外部参照を解決する場合でも、最後のリンクフェーズはインクリメンタルにしないようにします。

次の図は、リンカの入力ファイルと出力ファイルを示しています。

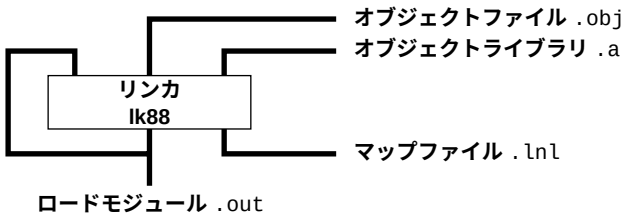


図3.1.1 S1C88リンカ

3.2 リンカの起動

S1C88リンカの起動構文は次のようになります。

lk88 [*option*]... *file* ...

オプションには、"-"が最初に付けられます。**-lx**オプションを除けば、位置は問題になりません。オプションは組み合わせることができます。そのため**-rM**は**-r -M**と同じ意味になります。ファイル名や文字列をとるオプションの場合、間にスペースを入れることができますが、入れなくてもかまいません。つまり**-o***name*は**-o** *name*と同じです。

*file*は、オブジェクトファイル(`.obj`)、オブジェクトライブラリ(`.a`)、インクリメンタルリンカ(`.out`)ファイルのいずれかになります。ファイルは、コマンド行に記述されるのと同じ順序でリンクされます。

リンカは、次のオプションを認識します。

オプションの要約

オプション	説　明
-C	大文字と小文字を区別しないでリンクします(デフォルトでは区別)。
-L *directory*	システムライブラリを探すための検索パスを追加します。
-L	システムライブラリの検索をスキップします。
-M	リンクマップ(`.lnl`)を生成します。
-N	重ね書きをオフにします。
-O *name*	生成されるマップファイルのベース名を指定します。
-V	バージョンヘッダのみを表示します。
-c	独立した呼び出しグラフファイル(`.cal`)を生成します。
-e	結果にエラーがある場合、その結果を消去します。
-err	エラーメッセージをエラーファイル(`.elk`)にリダイレクトします。
-f *file*	コマンド行の情報を*file*から読み込みます。"-"の場合`stdin`を示します。
-l *x*	システムライブラリ`lib`*x*`.a`内も検索します。
-o *filename*	出力ファイルの名前を指定します。
-r	定義されていないシンボルの診断を抑制します。
-u *symbol*	*symbol*を、シンボルテーブルで定義されていないものとして入力します。
-v or **-t**	冗長オプション。進行中にそれぞれのファイル名をプリントします。
-w *n*	警告レベルが*n*より上のメッセージを抑制します。

3.2.1 リンカオプションの詳細な説明

-C

このオプションを指定すると、**lk88**は大文字と小文字を区別しないでリンクします。デフォルトの場合、大文字と小文字を区別してリンクします。

-L [*directory*]

システムライブラリを検索するためのディレクトリのリストに、*directory*を追加します。**-L**で指定したディレクトリは、環境変数C88LIBで指定した標準ディレクトリの前に検索されます。ディレクトリ名を付けないで**-L**を指定した場合、環境変数C88LIBのディレクトリでシステムライブラリが検索されません。システムライブラリの検索パスに複数のディレクトリを追加する場合、**-L**オプションを複数回使用することができます。検索パスは、コマンド行で指定されたディレクトリの順序で作成されます。

注： *directory*に大文字の"O"を含むディレクトリ名を指定することはできません。

-M

リンクマップ(`.lnl`)を生成します。

-N

重ね書きをオフにします。デバッグのときに使用します。

-O *name*

*name*を、生成されるマップファイルのデフォルトベース名として指定します。

-V

このオプションを使用すると、リンカのバージョンヘッダのみを表示することができます。このオプションは、**lk88**の唯一の引数にする必要があります。他のオプションは無視されます。リンカは、バージョンヘッダを表示した後終了します。

-c

独立した呼び出しグラフファイル(.cal)を生成します。

-e

リンカでエラーが発生した場合、一時ファイル、生成出力ファイル、マップファイルなど、すべてのリンク時生成ファイルを削除します。

-err

リンカが、出力ファイルと同じベース名に拡張子.elkを付けたファイルに、エラーメッセージをリダイレクトします。デフォルトファイル名はa.elkです。

-f *file*

コマンド行の情報を*file*から読み込みます。ファイル名が"-"になっている場合、標準入力から情報が読み込まれます。stdinを閉じる時EOFコードが必要です。

コマンド行処理のために*file*を使用します。コマンド行のサイズ制限を回避するために、コマンドファイルを使用することができます。これらのコマンドファイルに入れるオプションは、実際のコマンド行では使用できないものです。コマンドファイルは、makeユーティリティなどを使用して簡単に作成できます。

-fオプションは複数使用することができます。

コマンドファイルの書式には、次のような簡単な規則があります。

1. コマンドファイルの同じ行に複数の引数を指定することができます。
2. 引数にスペースを入れるときは、その引数を一重引用符または二重引用符で囲みます。
3. 引用符の付いた引数の中で一重引用符または二重引用符を使用する場合、次の規則に従います。
   a. 中に入っている引用符が、一重引用符のみまたは二重引用符のみの場合、引数を囲むときもう一方の引用符を使用します。つまり、引数に二重引用符が含まれる場合、引数を一重引用符で囲みます。
   b. 両方の引用符が使用されている場合、それぞれの引用符がもう一方の引用符で囲まれるような形で、引数を分割する必要があります。

   例：
   ```
   "This has a single quote ' embedded"
   ```
   または
   ```
   'This has a double quote " embedded'
   ```
   または
   ```
   'This has a double quote " and a single quote '"' embedded"
   ```
4. オペレーティングシステムによっては、テキストファイル内の行の長さに制限がある場合があります。この制限を回避するため、継続行を使用することができます。これらの行は、最後に"\"(または"\")と改行が付きます。引用符の付いた引数の場合、継続行は、次の行のスペースを取らないでそのままつなげられます。引用符が付いていない引数の場合、次の行にあるすべてのスペースが削除されます。

   例：
   ```
   "This is a continuation ¥
   line"
           → "This is a continuation line"

   control(file1(mode,type),¥
           file2(type))
           → control(file1(mode,type),file2(type))
   ```
5. コマンド行ファイルは、最高で25レベルまでネストすることができます。

-l *x*

システムライブラリlib*x*.a(ただし*x*は文字列)内も検索します。リンカは最初に、**-L***directory*で指定されたディレクトリでシステムライブラリを検索し、次に環境変数C88LIBで指定された標準ディレクトリで検索します。ただしディレクトリ名を付けないで**-L**オプションを指定した場合はこの限りではありません。このオプションでは、位置が重要になります("3.3.2 ライブラリとのリンク"を参照)。

-o *filename*
リンカの出力ファイルの名前として*filename*を使用します。このオプションが省略された場合、デフォルトファイル名は`a.out`になります。

-r
解決されていないシンボルでレポートを作成しないようにします。インクリメンタルリンクの場合はこのオプションを使用しないでください。

-u *symbol*
*symbol*を、シンボルテーブルで定義されていないものとして入力します。これは、ライブラリからリンクするときに使用します。

-vまたは**-t**
冗長オプション。進行中にそれぞれのファイル名をプリントします。

-w *n*
警告レベルに0から9までの値を指定します。警告レベルが*n*より上のすべてのメッセージが抑制されます。メッセージのレベルは、このメッセージの最後のカラムにプリントされます。**-w**オプションを使用しない場合、デフォルトの警告レベルは8になります。

3.3 ライブラリ

ライブラリには2種類あります。そのうちの1つはユーザライブラリです。オブジェクトモジュールのライブラリを自身で作成する場合、このライブラリを通常のファイル名で指定する必要があります。リンカが、検索パスを使用してこのようなライブラリを検索することはありません。ファイルには拡張子`.a`を付けます。例を示します。

```
lk88 start.obj -fobj.lnk mylib.a
```

ライブラリがサブディレクトリにある場合は、次のようにします。

```
lk88 start.obj -fobj.lnk libs¥mylib.a
```

もう1つのライブラリは、システムライブラリです。システムライブラリは、**-l**オプションで定義する必要があります。オプション**-lcs**を使用すると、システムライブラリ`libcs.a`が指定されることになります。

3.3.1 ライブラリ検索パス

リンカは、次のアルゴリズムを使用して、システムライブラリファイルを検索します。

1. **-L***directory*オプションで指定されているディレクトリを、左から右の順番に使用します。例を示します。

   ```
   lk88 -L..¥lib -L¥usr¥local¥lib start.obj -fobj.lnk -lcs
   ```

2. **-L**オプションにディレクトリが指定されていない場合、環境変数C88LIBが存在するかどうかチェックします。存在する場合、その内容をライブラリファイルのディレクトリ指定子として使用します。ディレクトリをディレクトリ区切り文字で区切ることにより、環境変数C88LIBには複数のディレクトリを指定することができます。有効なディレクトリ区切り文字は、次のようになっています。

   上記の例では**-L**を使用していますが、C88LIBを使用しても同じディレクトリを指定することができます。

   ```
   set C88LIB=..¥lib;¥usr¥local¥lib
   lk88 start.obj -fobj.lnk -lcs
   ```

3. **lk88**インストールディレクトリの相対ディレクトリ`lib`で、ライブラリファイルを検索します。

   lk88.exeは`C:¥C88¥BIN`にインストールされます。そのためライブラリファイルが検索されるディレクトリは`C:¥C88¥LIB`になります。

   リンカは実行時に、どのディレクトリからバイナリが実行されるか判断するため、この`lib`ディレクトリを検索します。

4. それでもライブラリが見つからない場合、**lk88**のインストールディレクトリの相対ディレクトリ`lib`にあるプロセッサとモデル固有のサブディレクトリ(次の例を参照)でライブラリファイルを検索します。

   `C:¥C88¥LIB¥S1C88s`

アプリケーションがスモールメモリモデルで構築される場合、S1C88sディレクトリが検索されます。一般的に以下のようなディレクトリが検索されます。

ディレクトリ	構築されるアプリケーション
S1C88s	スモールモデル
S1C88d	コンパクトデータモデル
S1C88c	コンパクトコードモデル
S1C88l	ラージモデル

メモリモデルの詳細については、"1 Cコンパイラ"および"2.7.3.5 MODEL"を参照してください。

lk88は、必要な場合、ディレクトリ区切り文字を適宜挿入するため、**-L***directory*オプションまたはC88LIBで指定されているディレクトリ名の最後には、ディレクトリ区切り文字を付けても付けなくてもかまいません。

3.3.2 ライブラリとのリンク

ライブラリからリンクする場合、必要なオブジェクトのみがライブラリから抽出されます。つまり、次のようにしてリンカを起動した場合、

```
lk88 mylib.a
```

リンカが`mylib.a`内を検索するときに未解決のシンボルがないため、何もリンクされず、出力ファイルも生成されません。

オプション**-u**を付けることにより、強制的にシンボルを未定義のままにすることもできます。

```
lk88 -u main mylib.a
```
（-uとmainの間のスペースはオプション）

この場合、ライブラリ内でシンボル`main`が検索され、`main`を含むオブジェクトが抽出されます。このモジュールに新しい未解決のシンボルが含まれる場合、リンカは`mylib.a`内を再度調べます。このプロセスは、新しい未解決シンボルが見つからなくなるまで繰り返されます。"3.3.3 ライブラリメンバの検索アルゴリズム"も参照してください。

次のように指定した場合、ライブラリの位置が重要になります。

```
lk88 -lcs myobj.obj mylib.a
```

未解決のシンボルがないため、リンカはまずシステムライブラリ`libcs.a`を検索します。そのため、モジュールは抽出されません。その後、ユーザオブジェクトおよびユーザライブラリがリンクされます。これが終わった時点で、Cライブラリのすべてのシンボルが未解決になっています。そのため、正しい起動構文は次のようになります。

```
lk88 myobj.obj mylib.a -lcs
```

`myobj.obj`および`mylib.a`をリンクした後未解決になっているすべてのシンボルが、システムライブラリ`libcs.a`で検索されます。オブジェクト、ユーザライブラリ、システムライブラリの検索順は、コマンド行に記述されている順序になります。オブジェクトは常にリンクされ、ライブラリのオブジェクトモジュールは、必要な場合に限ってリンクされます。

3.3.3 ライブラリメンバの検索アルゴリズム

ar88で構築したライブラリの場合、そのライブラリの最初の部分に常にインデックス部分が含まれます。リンカは、未解決の外部参照を検索するときこのインデックスをスキャンします。ただし、インデックスをできる限り小さくするため、ライブラリメンバの定義済みのシンボルのみがこの領域に記録されます。

リンカが、未解決の外部参照と一致するシンボルを見つけると、対応するオブジェクトファイルがライブラリから抽出されて処理されます。オブジェクトファイルの処理が終わると、残りのライブラリインデックスが検索されます。検索が終わった後、未解決の外部参照が見つかった場合、ライブラリが再度スキャンされます。

-vオプションを使用すると、ライブラリに関するリンカアクションをたどることができます。

3.4 リンカの出力

リンカは、IEEE-695オブジェクト出力ファイルを生成し、必要であればマップファイルや呼び出しグラフファイルを生成します。

リンカの出力オブジェクトは、依然として再配置可能な状態になっています。セクションの絶対アドレスを決定するのは、ロケータです。リンカは、同じ名前を持つセクション同士を結合して、1つの(大きな)出力セクションにします。

リンカは、**-M**オプションが指定されている場合、マップファイルを生成します。マップファイルの名前は、出力ファイルの名前と同じで、拡張子が`.lnl`になります。出力ファイル名が指定されていない場合、デフォルト名として`a.lnl`が使用されます。マップファイルは、リンクされるオブジェクトごとに編成されます。それぞれのオブジェクトは、セクションに分割されており、各セクションごとにシンボルに分割されています。マップファイルには、リンクされたそれぞれのオブジェクトの、セクションの最初からの相対位置が示されます。

生成される呼び出しグラフも、マップファイルにプリントされます。呼び出しグラフには、どの関数呼び出しが存在するかについて、その概要が含まれています。また、呼び出しグラフのスタックの使用状況に関する情報も含まれています。関数が呼び出されるとき、関数に入る前のスタック使用状況が関数名の前に書き出されます。また、その関数のスタック使用回数の合計(その呼び出し自体も含まれる)が関数の後に書き出されます。関数自体の最大スタック使用回数は、関数の下に書き出されます。この数値は、使用スタックのサイズを示します(S1C88の場合バイト単位)。例を参照してください。

呼び出しグラフでは、その他に2種類のメッセージが表示されます。

- 1つは、再帰関数呼び出しが検出されたことを示すもので、次のように表示されます。

```
Call graph(s)
=============

Call graph 1:

 function
  |
  +-- function1  !! RECURSIVE !!
```

- もう1つは、静的関数がさまざまな呼び出しグラフから参照されているまたは呼び出されていることを検出した場合です。

  静的関数がさまざまな呼び出しグラフから呼び出されている場合、その関数は独立したグラフとして扱われ、参照されているさまざまな呼び出しグラフで重ね書きされることはありません。

```
Call graph(s)
=============

Call graph 1:

 root1
  |
  +-- shared  !! NOT OVERLAYED !! (referenced by different call graphs)

Call graph 2:

 root2
  |
  +-- shared  !! NOT OVERLAYED !! (referenced by different call graphs)
  |
  +-- sub2
```

コマンド行オプション**-c**を付けると、リンカは強制的に、圧縮した呼び出しグラフを持つ独立した呼び出しグラフファイルを生成するようになります。このファイルのファイル名拡張子は、.calになります。

リンカをインクリメンタルリンクに使用する場合、**-r**オプションを使用します。このオプションを使用すると、未解決シンボルの診断が生成されなくなり、重ね書きも行われなくなります("3.5 オーバーレイセクション"を参照)。この場合、リンカの出力を入力オブジェクトとして再使用することができます。呼び出しグラフは常に生成されます。

マップファイルサンプル(.lnl):

```
Call graph(s)
=============

Call graph 1:

 _start ( 14 )
  |
  +-( 4 )- _exit ( 2 )
  |    |
  |    +-( 2 )
  |
  +-( 2 )- main ( 12 )
  |    |
  |    +-( 2 )- puts ( 10 )
  |    |    |
  |    |    +-( 2 )- fputc ( 8 )
  |    |    |    |
  |    |    |    +-( 2 )- _flsbuf ( 6 )
  |    |    |    |    |
  |    |    |    |    +-( 2 )- _iowrite ( 2 )
  |    |    |    |    |    |
  |    |    |    |    |    +-( 2 )
  |    |    |    |    |
  |    |    |    |    +-( 2 )- _write ( 4 )
  |    |    |    |    |    |
  |    |    |    |    |    +-( 2 )- _iowrite ( 2 )
  |    |    |    |    |    |    |
  |    |    |    |    |    |    +-( 2 )
  |    |    |    |    |    |
  |    |    |    |    |    +-( 2 )
  |    |    |    |    |
  |    |    |    |    +-( 2 )
  |    |    |    |
  |    |    |    +-( 2 )
  |    |    |
  |    |    +-( 2 )
  |    |
  |    +-( 2 )
  |
  +-( 4 )

Maximum stack usage: 14
Pool offsets
============

Pool #1: zp_ovln  (Total of  39 bytes)

                Pool: zp_ovln
                        off     siz
       puts()           0       6
       fputc()          6       7
       _flsbuf()        13      12
       _write()         25      10
       _iowrite()       35      4
Object: cstart.obj
==================
```

```
Section:abs_65534 ( Start = 0x0 )


Section:.text ( Start = 0x0 )
0x0000001c E __exit
0x00000000 E __START
Object: hello.obj
=================

Section:.text ( Start = 0x1f )
0x0000001f E _main


Section:.string ( Start = 0x0 )
Object: _puts.obj
=================

Section:.text ( Start = 0x28 )
0x00000028 E _puts
Object: _fputc.obj
==================

Section:.text ( Start = 0x78 )
0x00000078 E _fputc
Object: _iob.obj
================

Section:.near_data ( Start = 0x0 )
0x00000000 E __iob


Section:.near_bss ( Start = 0x0 )
0x00000000 E __ungetc
Object: _flsbuf.obj
===================

Section:.text ( Start = 0x0102 )
0x00000102 E __flsbuf
Object: _iowrite.obj
====================

Section:.text ( Start = 0x0314 )
0x00000314 E __iowrite
Object: _write.obj
==================

Section:.text ( Start = 0x0318 )
0x00000318 E __write
```

マップファイル内のアドレスは、出力ファイルのセクションの最初との相対オフセットになります。たとえば、オブジェクトモジュール`hello.obj`のセクション`.text`は、出力の`.text`セクションからのオフセット0x1fで始まります。関数`main`も、生成される`.text`セクションの開始点からのオフセット0x1fで始まります。アドレスの後の`E`は、ラベルが外部であることを示しています。

呼び出しグラフの次の箇所を見ると、

```
+----- _write ( 4 )
      |
      +-( 2 )- _iowrite ( 2 )
      |    |
      |    +-( 2 )
      |
      +-( 2 )
```

ツリー構造のインデントから、関数_writeが関数_iowriteを呼び出すことがわかります。関数_writeのスタック使用回数の合計(その呼び出し自体も含まれる)は、次のように関数名の後に示されています。

```
_write ( 4 )
```

スタック使用回数の合計を判断するとき、次の最大値を参照します。

1. 関数を呼び出す前のローカル使用回数(最初の値)を、その関数の使用回数(最後の値)の合計に加えたもの。
   ```
   +-( 2 )- _iowrite ( 2 )
   ```
2. 関数自体の使用回数。
   ```
      |
   +-( 2 )
   ```

3.5 オーバーレイセクション

静的メモリモデルのメモリをさらに効率的に使用するため、コンパイラは、オーバーレイ属性を付けて特殊なセクションを生成(リンカは必ず重ね書きする)します。それぞれのC関数には、ローカル変数や一時データなどを持つ独自のセクションがあります。リンカは、呼び出しグラフを構築し、お互いに呼び出さない関数のセクションについて、重ね書きが正しいかどうか判定します。

例：

```
#include <stdio.h>

void foo( int );

void
main(void)
{
        int j;
        printf( "hello¥n" );
        j = 2;
        foo(j);
}
void
foo( int j )
{
        int i;
        i = j;
}
```

リンカは、fooがprintfを呼び出さず、printfもfooを呼び出さないことを検出します。コンパイラは、ローカル変数iに対して重ね書き可能なデータセクションを生成します。printfにもローカル変数がありますが、printfは、独自の重ね書き可能データセクションを生成します。リンカは、これらの2つの関数のオーバーレイセクションを同じメモリ領域に置きます。その結果、ターゲットメモリが効率的に使用できるようになります。

3.6 型のチェック

3.6.1 はじめに

デフォルトの場合、コンパイラとアセンブラは、高レベルの型情報を生成します。(**-g0**で)型情報の生成をオフにしていない限り、それぞれのオブジェクトには、高レベルの型情報が含まれることになります。リンカは、この型情報を比較し、矛盾があればそれについて警告を出します。リンカは、次の4種類の型矛盾を区別します。

1. 型が正しく指定されていない場合(W109)。配列の深さを指定していない場合、または関数プロトタイプで引数を指定していない場合に発生します。警告レベル9を指定していない限り(**-w9**)、リンカはこの種の矛盾を報告しません。デフォルトの警告レベルは8です。
2. 型に互換性があるにもかかわらず、定義が異なる場合(W110)。たとえばshort型にint型をリンクした場合に発生します。S1C88は、どちらも16ビットと見なすため、問題にはなりません。しかし、このようなコードには移植性がなくなります。また、異なる名前の構造体や型を使用しても、この警告が発生します。このメッセージの警告レベルは8であるため、この種のメッセージをオフにするためには、警告レベルを7以下に設定します(**-w7**)。
3. 符号付き/符号なしの矛盾(W111)。signed intをunsigned intとリンクすると、このメッセージが出されます。多くの場合問題はありませんが、符号なしバージョンでは、整数値が大きくなります。このメッセージの警告レベルは6であるため、この種のメッセージを抑制するためには、警告レベルを5以下に設定します(**-w5**)。
4. その他の矛盾(W112)。警告112が出された場合、もっと重大な型矛盾があると考えられます。関数の戻り型の矛盾、2つの組み込み型の長さの矛盾(short/long)、型が完全に異なる場合などが考えられます。このメッセージの警告レベルは4であるため、この種のメッセージをオフにするためには、警告レベルを3以下に設定します(**-w3**)。

3.6.2 再帰的な型チェック

リンカは、型を再帰的に比較します。たとえば、`foo`の型の場合を考えます。

```
struct s1 {
     struct s2 *s2_ptr;
};

struct s2 {
     int  count;
} sample;

struct s1 foo = { &sample };
```

このソースをコンパイルして、次のような、`struct s2`が異なるだけの他のコンパイル済みソースとリンクした場合、

```
struct s1 {
     struct s2 *s2_ptr;
};

struct s2 {
     short count;
};

extern struct s1 foo;
```

メッセージW112(型の矛盾)が生成されます。`struct s1`はどちらの場合も同じですが、実数型の矛盾になります。たとえばコード"`foo.s2_ptr->count++`"の場合、両方のオブジェクトで異なるコードが生成されます。

1つのシンボルに複数の矛盾がある場合、リンカは低い方の警告レベルの矛盾(もっとも重大なもの)を報告します。

3.6.3 関数間の型チェック

K&Rスタイルの関数を使用する場合、引数の型および引数の数値をチェックすることはできません。特に指定されていない場合、戻り型は"int"になります。プロトタイプは、関数の戻り型が整数型以外の場合のみ必要になります。

```
test2( par )
int par;
{
        test1( par );
        return test3( 1, 2 );
}
```

この場合、test1(他のソースで定義されたもの)の戻り型はvoidで、test3の戻り型はデフォルトのintになっています。デフォルトの警告レベルの場合、リンカは矛盾を報告しません。警告レベル9を指定した場合(**-w9**)、リンカが引数をチェックしないため、リンカは"型が正しく指定されていない場合"を報告します。戻り型の矛盾は、警告レベル4の実数型の矛盾になります。

ソースがANSIスタイルの場合(推奨)、リンカは、すべてのパラメータの型およびパラメータの数値をチェックします。この場合、上記の例のソースは次のようになります。

```
void test1( int );      /* ANSI style prototypes */
int  test3( int, int );

test2( int par )        /* ANSI style function definition*/
{
        test1( par );
        return test3( 1, 2 );
}
```

test1およびtest3の定義を含むソースは、次のようになっています。

```
void test1( int one )
{
        /*
        **  code for function test1
        */
        .
        .
        .
}
int test3( int one, int two )
{
        /*
        **  Code for function test3
        */
        .
        .
        .
}
```

プロトタイプは、ある関数がソース内で定義される前に参照される場合のみ必要になります。しかし、1つのファイルですべての関数についてプロトタイプを記述しておき、そのプロトタイプファイルをインクルードすることもできます。こうすると、関数の型チェックがコンパイラによって行われるようになります。それにもかかわらず、プロトタイプファイルを変更した後すべてのソースをコンパイルしなかった場合、リンカによって型の矛盾が報告されます。

ANSIスタイルのプロトタイプをK&RスタイルのCコードに追加することができます。この場合、関数について完全な型チェックが可能になります。これを実行するためには、作成するアプリケーションで、すべての関数についてのすべてのプロトタイプを記述した新しいヘッダファイルを作成します。このファイルをそれぞれのソースにインクルードするか、**-H**オプションを指定することにより、コンパイラに対してこのファイルをインクルードするよう命令します。

```
cc88 -c -Hproto.h *.c
```

3.6.4 指定されていない型

Cでは、指定されていないオブジェクトに対するポインタを定義することができます。リンカは、このような型をチェックすることができません。例を示します。

```
struct s1 {
        struct s2 *s2_ptr;
};

struct s1 foo;
```

構造体s2は指定されていません。リンカは、struct s2がすべてのソースで同じかどうかチェックすることができないため、次のようなレベル9の警告が出されます。

```
lk88 W102(9) <name>: Incomplete type specification, type index = T101
```

struct s2が他のソースで認識されている可能性もあります。このソースが変数fooを使用する場合、レベル9の型の矛盾を報告する2番目のメッセージが生成されます。

```
lk88 W109(9) <fl>: Type not completely specified for symbol <foo> in <f2>
```

最初の警告は、型の定義が完全でないため、リンカが型をチェックできないことを報告しています。ただし、これはCで認められています。このメッセージは、不完全な型があれば、オブジェクトごとに一度ずつ出されます。2番目のメッセージは、不完全な型と完全な型とで、型が異なることを報告しています。これらのすべての警告は、警告レベル9を指定した場合(**-w9**)にのみ生成されます。

3.7 リンカのメッセージ

メッセージには、致命的なメッセージ、エラーメッセージ、警告メッセージ、冗長メッセージの4種類があります。致命的なメッセージは、重大なエラーのためリンカがタスクを継続できない場合に出されます。このような場合、終了コードは2になります。エラーメッセージは、リンカにとって致命的でないエラーが発生した場合に報告されます。ただしこの場合、リンカの出力は使用できなくなります。エラーメッセージが1つ以上ある場合終了コードは1になります。警告メッセージは、リンカがエラーの可能性を検出したにもかかわらず、これらのエラーを判断できない場合に生成されます。この場合の終了コードは0になります。この場合、.outファイルが使用できます。もちろん、リンカがメッセージをまったく報告しない場合も、終了コードは0になります。

それぞれのリンカメッセージには組み込みの警告レベルがあります。オプション**-w***x*を付けると、警告レベルが*x*より上のすべてのメッセージが抑制されます。

冗長メッセージは、冗長オプション(**-v**)がオンの場合のみ生成されます。リンクプロセスの進行が報告されます。

リンカメッセージは、次のようなレイアウトになります。

```
S1C88 object linker v x.y rz        SN000000-000 (C)year Tasking Software BV
lk88 W112 a.obj: Type conflict for symbol <f> in b.ob                  (4)
```

1行目は、S1C88リンカのバナーです。2行目は、ファイルa.objの型の矛盾を報告しています。ここでは、モジュールb.objの関数fの型定義が矛盾していることが示されています。行の最後の数字"(4)"は警告レベルを示しています。

次のような4つのメッセージグループがあります。

1. 致命的(常にレベル0)：
   - 書き込みエラー。
   - メモリ不足。
   - 不正な入力オブジェクト。

2. エラー(常にレベル0)：
   - シンボルが未解決(しかもインクリメンタルリンクではない)。
   - 入力ファイルがオープンできない。
   - 再入不能な関数の不正な再帰的使用。

3. 警告(レベル1から9)：
   - 2つのシンボルの型の矛盾。
   - 不正なオプション(無視)。
   - システムライブラリ検索パスが指定されていないときに、システムライブラリが要求された。

4. 冗長(レベルと関係なし。オプション**-v**が指定された場合のみ)：
   - ライブラリからファイルが抽出された。
   - 現在のファイル/ライブラリ名。
   - 1回目のパスと2回目のパス。
   - ライブラリで新しい未解決シンボルを再スキャン。
   - 一時ファイルの消去。
   - 警告レベル。

4 ロケータ

4.1 概要

この章では、S1C88ロケータについて説明します。

ロケータのタスクは、**lk88**が作成した.outファイルを絶対アドレスにロケートすることです。組み込み環境では、使用可能なメモリの正確な記述や、その動作のコントロールに関する情報が、アプリケーションの成功にとって重要になります。たとえば、アプリケーションを、異なるメモリ構成を持つプロセッサに移植する場合にもこれが必要になる他、セクションのロケーションを調節して高速メモリチップを最大限に活用する場合にも必要になります。ロケータがタスクを実行するとき、使用するS1C88派生製品の記述が必要になります。ロケータは、この記述に特殊な言語、DELFEE(DEscriptive Language For Embedded Environments)を使用します。この言語は、記述ファイルと呼ばれる特殊なファイルで使用されます。詳細については、"5 DEscriptive Language for Embedded Environment"を参照してください。

記述ファイルは、ロケータの起動時にオプションパラメータで指定します。コマンド行で記述ファイル名を指定しない場合、または**-d**オプションを使用しない場合、ロケータはカレントディレクトリ、またはS1C88製品ツリーのディレクトリetcでファイルs1c88.dscを検索します。

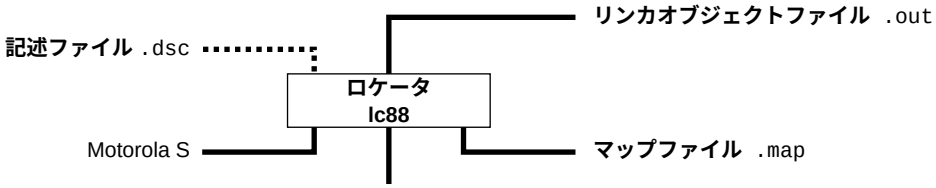


図4.1.1 ロケータ

4.2 起動

ロケータの起動構文は次のようになります。

lc88 [*option*]... [*file*] ...

オプションには、"-"が最初に付けられます。オプションは組み合わせることができます。そのため、**-eM**は**-e -M**と同じ意味になります。ファイル名や文字列をとるオプションの場合、間にスペースを入れることができますが、入れなくてもかまいません。つまり**-o***name*は**-o** *name*と同じです。*file*は、.out拡張子または.dsc拡張子を持つ任意のファイルになります。

ロケータは、次のオプションを認識します。

オプションの要約

オプション	説　明
-M	ロケートマップファイル(.map)を生成します。
-S *space*	特定の*space*を生成します。
-V	バージョンヘッダのみを表示します。
-d *file*	記述ファイルの情報を*file*から読み込みます。"-"の場合stdinを示します。
-e	結果にエラーがある場合、その結果を消去します。
-err	エラーメッセージをリダイレクトします。(.elc)
-f *file*	コマンド行の情報を*file*から読み込みます。"-"の場合stdinを示します。
-f *format*	出力フォーマットを指定します。
-o *filename*	出力ファイルの名前を指定します。
-p	stdoutにソフトウェア部分のプロポーザルを作成します。
-v	冗長オプション。進行中にそれぞれのファイル名をプリントします。
-w *n*	警告レベルが*n*より上のメッセージを抑制します。

4.2.1 ロケータオプションの詳細な説明

-M

ロケートマップ(.map)を生成します。

-S *space*

このオプションを使用すると、すべての空間を含む出力ファイルではなく、指定された空間(*space*)に対応する特定の出力ファイルを生成することができます。*space*は、.dscファイルの空間の名前になります。

-V

このオプションを使用すると、ロケータのバージョンヘッダのみを表示することができます。このオプションは、**lc88**の唯一の引数にする必要があります。他のオプションは無視されます。ロケータは、バージョンヘッダを表示した後終了します。

-d *file*

記述ファイルの情報を、.dscファイルからではなく*file*から読み込みます。*file*が"-"の場合、情報は標準入力から読み込まれます。

-e

エラーが発生した場合、一時ファイル、生成出力ファイル、マップファイルなど、すべてのロケート時生成ファイルを削除します。

-err

拡張子.elcを持つエラーファイルに、エラーメッセージをリダイレクトします。

-f *file*

コマンド行の情報を*file*から読み込みます。*file*が"-"の場合、標準入力から情報が読み込まれます。stdinをクローズするためにはEOFコードを入れなければなりません。*file*には、0～3の範囲の数字を使用することはできません。これらの数字は、出力フォーマットを指定するときに使用されます。
コマンド行処理のために*file*を使用します。コマンド行のサイズ制限を回避するために、コマンドファイルを使用することができます。これらのコマンドファイルに入れるオプションは、実際のコマンド行では使用できないものです。コマンドファイルは、makeユーティリティなどを使用して簡単に作成できます。

-fオプションは複数使用することができます。

コマンドファイルの書式には、次のような簡単な規則があります。

1. コマンドファイルの同じ行に複数の引数を指定することができます。
2. 引数にスペースを入れるときは、その引数を一重引用符または二重引用符で囲みます。
3. 引用符の付いた引数の中で一重引用符または二重引用符を使用する場合、次の規則に従います。
   a. 中に入っている引用符が、一重引用符のみまたは二重引用符のみの場合、引数を囲むときもう一方の引用符を使用します。つまり、引数に二重引用符が含まれる場合、引数を一重引用符で囲みます。
   b. 両方の引用符が使用されている場合、それぞれの引用符がもう一方の引用符で囲まれるような形で、引数を分割する必要があります。

   例：
   ```
   "This has a single quote ' embedded"
   ```
   または
   ```
   'This has a double quote " embedded'
   ```
   または
   ```
   'This has a double quote " and a single quote '"' embedded"
   ```
4. オペレーティングシステムによっては、テキストファイル内の行の長さに制限がある場合があります。この制限を回避するため、継続行を使用することができます。これらの行は、最後に"\"(または"\")と改行が付きます。引用符の付いた引数の場合、継続行は、次の行のスペースを取らないでそのままつなげられます。引用符が付いていない引数の場合、次の行にあるすべてのスペースが削除されます。

例：

```
"This is a continuation ¥
line"
        → "This is a continuation line"

control(file1(mode,type),¥
        file2(type))
        → control(file1(mode,type),file2(type))
```

5. コマンド行ファイルは、最高で25レベルまでネストすることができます。

-f *format*

出力フォーマットを指定します。*format*は、以下のいずれかの出力フォーマットになります。

1 = IEEE標準695（デフォルト）
2 = Motorola Sレコード

デフォルトの出力フォーマットはIEEE標準695（**-f1**）で、このフォーマットの場合デバッガで直接使用できます。他の出力フォーマットは、PROMプログラマにロードするときに使用することができます。

-o *filename*

ロケータの出力ファイルの名前として*filename*を使用します。このオプションが省略された場合、デフォルトファイル名は、指定された出力フォーマットによって異なってきます。

フォーマット	デフォルト出力名
1	a.abs
2	a.sre

-p

記述ファイルのソフトウェア部分のプロポーザルを標準出力に作成します。

-v

冗長オプション。進行中にそれぞれのファイル名をプリントします。

-w *n*

警告レベルに0から9までの値を指定します。警告レベルが*n*より上のすべてのメッセージが抑制されます。メッセージのレベルは、このメッセージの最後のカラムにプリントされます。**-w**オプションが省略された場合、デフォルトの警告レベルは8になります。

4.3 入門

ロケータの起動は、通常、コントロールプログラムで行います。このコントロールプログラムは、ロケータフェーズを完全に覆い隠します。この節では、ロケータを独立したツールとして起動することで、オプションや記述ファイルの使用について理解を深めます。

Descriptive Language for Embedded Environments(DELFEE)の詳細については、"5 DEscriptive Language for Embedded Environment"を参照してください。

calcデモをロケートする場合は、リロケータブルデモファイルcalc.outをロケータの入力として使用する必要があります。このファイルは、ディレクトリexamles¥asmの内容を作業ディレクトリにコピーして、次のようにコントロールプログラムを起動することにより生成します。

```
cc88 -cl -M -Ms -nolib startup.asm _copytbl.asm watchdog.asm
     calc.asm -o calc.out
```

ただし、S1C88ツールのbinディレクトリが検索パスに入っている必要があります。ここで使用しているオプション**-cl**は、リンクが終わった時点で停止しロケートフェーズを抑制するよう、コントロールプログラムに命令するためのものです。このコマンドで作成したファイルは、完全なデモですが、再配置可能(リロケータブル)な形式になっています。ここで、次のように入力することにより、このリロケータブルファイルcalc.outを絶対アドレスにロケートできます。

```
lc88 -M calc.out -ds1c88316.dsc
```

-Mオプションは、**lc88**にマップファイルを作成するよう命令するためのものです。デフォルトの出力ファイル形式は、IEEE-695(**-f1**オプション)になっています。ここでは出力ファイルの名前を指定していないため、デフォルトの出力ファイルa.absが生成されます(デフォルトは、**-f1**の場合a.abs、**-f2**の場合a.sreになる)。これを実行すると、ロケータが次の2つのファイルを生成します。

- a.abs　IEEE-695出力ファイル
- a.map　ロケートマップファイル

出力ファイルに特定の名前を指定したい場合、次のように**-o** *file*オプションを使用します。

```
lc88 -M calc.out -o calc.abs -ds1c88316.dsc
```

場合によっては、記述ファイルを調整する必要があります。記述ファイルでは、ロケータのロケートアルゴリズムを変更することができます。記述ファイル(**-d**オプションの引数)、を指定していない場合、ロケータは(S1C88製品ツリーの)etcサブディレクトリにあるファイルs1c88.dscを使用します。上記では、**-d**オプションを付けて、s1c88316.dsc記述ファイルを指定しています。この元の記述ファイルを変更したくない場合、ファイルs1c88316.dscをユーザの作業ディレクトリにコピーします(この方法を推奨)。

次に、この記述ファイルのコピーを変更します。コメント(//)の後にあるものは、行末まですべて無視されます。例として、次の行がある箇所を、

```
amode   code {
        section selection=x;
        section selection=r;
        copy;
        table;
}
```

以下のように変更します。

```
amode   code {
        section .text;
        section .ptext;
        copy;
        table;
}
```

この結果、.textと.ptextのセクションのロケーションの順序が強制的に変更されます。

影響を見るために再度ロケートします。ユーザの作業ディレクトリにある修正済みの記述ファイルs1c88316.dscが、etcディレクトリにある元のバージョンの前に検出されます。マップファイルを比較したいため、他の出力名を選択します。

```
lc88 -M calc.out -ocalc_o.abs -ds1c88316.dsc
```

これで、calc.mapとcalc_o.mapを比較できます。

変更前の記述ファイルと変更後の記述ファイルを状況に応じて切り替えたい場合、作業ディレクトリにあるs1c88316.dscの名前をorder.dscなどに変更します。変更したバージョンの記述ファイルが必要な場合、ロケータを次のように起動します。

```
lc88 -M -d order calc.out -ocalc_o.abs
```

-dと**order**の間のスペースはオプションです。etcサブディレクトリにorder.dscをインストールすれば、任意の作業ディレクトリからオプション**-dorder**を使用できるようになります。

ロケート言語DELFEEの詳細については、第5章を参照してください。

4.4 コントロールプログラムからのロケータの呼び出し

ロケータはコントロールプログラム**cc88**から呼び出すことをお勧めします。コントロールプログラムは、特定のオプションをロケータ用に変換します(たとえば**-srec**を**-f2**へ)。(**-M**などの)他のオプションは直接ロケータに渡されます。通常、コントロールプログラムを使用すると、.c、.src、.asm、.objの各ファイルから.absファイルを直接生成することができます。

```
cc88 -M -Ms -g -nolib startup.asm _copytbl.asm watchdog.asm calc.asm
     -o calc.abs s1c88316.dsc
```

上記の起動構文は、デバッガ経由で実行できるcalc.absという名前の絶対デモファイルを構築します。

4.5 ロケータの出力

ロケータは絶対ファイルを生成します。また必要に応じて、マップファイルやエラーファイルも生成します。この出力ファイルは絶対ファイルで、**-f**オプションの使用法に従ってMotorola Sレコードフォーマット、IEEE-695フォーマットのいずれかになります。この場合デフォルト出力名は、それぞれa.sre、a.absになります。

マップファイル(**-M**オプション)は常に、出力オブジェクトファイルと同じベース名に拡張子.mapを付けたファイルになります。マップファイルには、各セクションの絶対位置が記述されています。リストされている外部シンボルには、絶対アドレスが付いており、アドレスおよびシンボルの両方でソートされています。

エラー出力ファイル(**-err**オプション)には、オブジェクト出力ファイルと同じ名前が付けられますが、同時に拡張子.elcが付きます。**-err**オプションが評価される前にエラーが発生した場合、stderrにプリントされます。

4.6 ロケータのメッセージ

メッセージには、致命的なメッセージ、エラーメッセージ、警告メッセージ、冗長メッセージの4種類があります。致命的なメッセージは、重大なエラーのためロケータがタスクを継続できない場合に出されます。このような場合、終了コードは2になります。エラーメッセージは、ロケータにとって致命的でないエラーが発生した場合に報告されます。ただしこの場合、ロケータの出力は使用できなくなります。エラーメッセージが1つ以上ある場合終了コードは1になります。警告メッセージは、ロケータがエラーの可能性を検出したにもかかわらず、これらのエラーを判断できない場合に生成されます。この場合の終了コードは0になります。この場合、.absファイルが使用できます。もちろん、ロケータがメッセージをまったく報告しない場合も、終了コードは0になります。

それぞれのロケータメッセージには組み込みの警告レベルがあります。オプション**-w***x*を付けると、警告レベルが*x*より上のすべてのメッセージが抑制されます。

冗長メッセージは、冗長オプション(**-v**)がオンの場合のみ生成されます。冗長メッセージでは、ロケートプロセスの進行が報告されます。

ロケータメッセージは、次のようなレイアウトになります。

```
S1C88 locator vx.y rz        SN000000-127 (C) year TASKING Software BV
lc88 W112 (3) calc.out: Copy table not referenced, initial data is not copied
```

1行目は、ロケータのバナーです(ロケータがコントロールプログラムから起動された場合表示されない)。
2行目は、警告です。警告番号の後にある数字は警告レベルを示しています。

4.7 アドレス空間

図4.7.1、4.7.2は、S1C88のさまざまなアドレス空間マッピングを示しています。

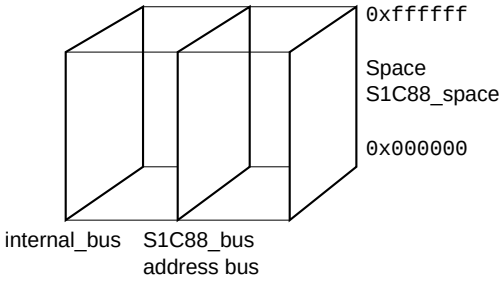


図4.7.1 S1C88の物理アドレス空間マッピング

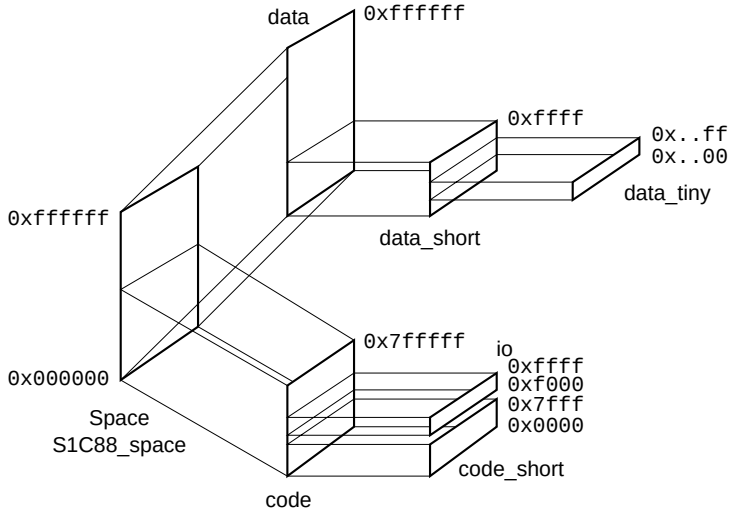


図4.7.2 S1C88の仮想アドレス空間マッピング

4.8 コピーテーブル

プロセスの初期化時のアクションの1つに、ROMからRAMへデータをコピーして、メモリをCLEAR属性で初期化するというものがあります。ロケータは、それぞれのプロセスごとにコピーテーブルを生成します。コピーテーブルは、ラベル__lc_cpで参照することができます。コピーテーブルのエントリは、次のようなレイアウトになっています(Cコンパイラに付属するlocate.hを参照)。

```
typedef struct cp_entry {
   char                  cp_actions;     /* 1 byte */
   _huge unsigned char   *cp_destin;     /* 3 byte address */
   _huge unsigned char   *cp_source;     /* 3 byte address */
   usingned long         cp_length;      /* 4 byte length */
} cp_entry_t;
```

最初のメンバcp_actionsは、現在のエントリでどのようなアクションを実行するか定義します。それぞれのアクションは、アクションごとに1ビットの値で編成されます。

値0　テーブルの最後に到達しました。

CP_COPY(値1)　cp_lengthバイト分、cp_sourceからcp_destinへコピーします。

CP_BSS(値2)　cp_lengthバイト分、cp_destinのメモリをクリアします。

テーブルエントリは、次のように生成されます。

- CLEAR属性を持つセクションにはそれぞれ1つのエントリ
- INIT属性を持つセクションにはそれぞれ1つのエントリ
- テーブルの最後を示す場合"0"エントリ

何もすることがない場合(クリアするセクションもコピーするデータもない場合)、コピーテーブルには、値0を持つアクションエントリが1つだけあります。

Cのレベルでは、コピーテーブルを次のように宣言することができます。

```
cpt_t _lc_cp;
```

エントリxのメンバにアクセスする場合、次のように記述します。

```
_lc_cp[ x ].cp_actions;
```

ラベル__lc_cpを使用しない場合、テーブルは生成されません。

4.9 ロケータのラベル

ロケータは、次のラベルを参照するとき、それぞれのラベルにアドレスを割り当てます。

__lc_cp: コピーテーブルの開始。コピーテーブルは、コピーするセクションのソースアドレスとデスティネーションアドレスを指定します。このテーブルは、このラベルが使用される場合に限り、ロケータによって生成されます。

__lc_bs: スタック空間の開始(キーワードstackを使用)。

__lc_es: スタック空間の終了。スタックポインタの初期化。

__lc_b_*name*: セクション*name*の開始。

__lc_e_*name*: セクション*name*の終了。

__lc_u_*name*: ユーザ定義ラベル。このラベルは記述ファイルで定義します。例を示します。

```
label mylab;
```

__lc_ub_*name*: ユーザ定義ラベルの開始。このラベルは記述ファイルで定義します。例を示します。

```
label mybuffer length=100;
```

__lc_ue_*name*: ユーザ定義ラベルの終了。

4.9.1 ロケータラベルのリファレンス

この節では、すべてのロケータラベルについて説明します。ロケータラベルは、**__lc_**が最初に付きます。これらのラベルは、リンカでは無視されて、ロケート時に解決されます。これらのラベルには、セクションの最初と最後に実際に置かれるラベルになるものもあります。他のラベルについては、ロケータ生成データをアドレスするときにも使用できるようになっています。これらのデータは、ラベルが使用されるときに限り生成されます。

__lc_が最初に付くラベルは、リンカとロケータで扱い方が変わるため、この種のラベルは、定義用ではなく参照用として使用します。

注： Cのレベルでは、すべてのロケータラベルの最初にアンダースコア(_)が1つ付いています(コンパイラがアンダースコアをもう1つ追加する)。

__lc_b_*section*, __lc_e_*section*

構文：

```
extern unsigned char _lc_b_section[ ];
extern unsigned char _lc_e_section[ ];
```

説明：

プログラム内のセクション*section*のアドレスを取得するときに、一般ロケータラベル**__lc_b_***section*と**__lc_e_***section*を使用することができます。**b**バージョンは、セクションの最初をポイントし、**e**バージョンは、セクションの最後をポイントします。

セクション名の前のドットをアンダースコア(_)で置換することにより、これらのラベルに"C"からアクセスできるようになります。ただしこの変換により、名前の競合が発生する可能性があります。たとえば、セクション.textとセクション_textの両方が存在する場合、一般ラベル__lc_b_textは、_textの最初に設定されます。そのためセクション.textのラベルは、アセンブリレベルでのみ実際の名前によって使用することができます。このような競合は、もちろん回避する必要があるため、アンダースコアが最初に付くセクション名は使用しないようにします。

例：

```
Printf ("Text size is 0x%x¥n", _lc_e__text-_lc_b__text);
```

__lc_bh, __lc_eh

構文：

```
extern unsigned char _lc_bh[ ];
extern unsigned char _lc_eh[ ];
```

説明：

すべてのロケータ**h**ラベルは、ヒープに関連しています。ヒープは、クラスタの記述で定義することによって割り当てることができます。DELFEEキーワード**heap**も参照してください。

__lc_bhは、ヒープの最初に置かれるラベルです。"C"レベルの場合、**_lc_bh**はヒープを表します。このラベルはchar配列として定義されますが、任意の基本型の配列として定義することができます。**__lc_eh**は、ヒープの最後を表します。

例：

ヒープの定義

```
block total_range {
        .
        .
        cluster ram {
          amode data {
                heap length = 200;
                .
          }
        }
        .
}
```

sbrkコード

```
extern unsigned char _lc_bh[];
extern unsigned char _lc_eh[];

static char *
sbrk(   long length ) {
        .
        .
if ( (lastmem + length) > _lc_eh ) {
        return (char *) -1; /* overflow */
}
```

__lc_bs, __lc_es

構文：

```
extern unsigned char _lc_bs[ ];
extern unsigned char _lc_es[ ];
```

説明：

すべてのロケータ**s**ラベルは、スタックに関連しています。スタックは、クラスタの記述で定義することによって割り当てることができます。DELFEEキーワード**stack**も参照してください。

__lc_bsは、スタックの最下位アドレスのラベルです。"C"レベルの場合、**_lc_bs**はスタックの下位アドレスを表します。このラベルはchar配列として定義されますが、任意の基本型の配列として定義することができます。**__lc_es**は、スタックの最上位アドレスを表します。**__lc_es**は、**__lc_bs**より高いアドレスになります。S1C88のスタックはもっと低いアドレスになるため、実際にはスタックはラベル**__lc_es**で開始し、**__lc_bs**で終了します。

例：

スタックの定義

```
block total_range {
       cluster ram {
         amode data {
            stack length = 100;
             .
         }
      }
}
```

スタックの初期化

```
__START:
       LD SP, #__lc_es  ; set stack pointer to begin of stack space
```

__lc_cp

構文：

```
extern char* _lc_cp;
```

説明：

コピーテーブルはプロセスごとに生成されます。このテーブルの各エントリは、コピーまたは消去のアクションを表しています。テーブルのエントリは、次のセクションで、ロケータによって自動的に生成されます。

- 属性bを持つすべてのセクション。スタートアップ時にクリアされます(クリアアクション)。
- 属性iを持つすべてのセクション。プログラムのスタートアップ時にROMからRAMへコピーされます(コピーアクション)。

コピーテーブルのレイアウトは、"4.8 コピーテーブル"に記述されています。タイプ`cpt-t`は`locate.h`を定義しています。

__lc_u_*identifier*

構文：

```
extern int _lc_u_identifier[ ];
```

説明：

このロケータラベルは、ユーザがDELFEEキーワード**label**を使用して定義することができます。このラベルは、DELFEEファイルで、接頭辞**__lc_u_**を付けずに定義する必要があります。アセンブリからは接頭辞**__lc_u_**を付けてラベルを参照することができ、Cからは接頭辞**_lc_u_**(最初のアンダースコアが1つ)を付けて参照できます。

例：

記述ファイルの内容

```
block   total_range {
        cluster ram {
          amode data {
                label bstart;
                section text;
                label bend;
          }
        }
        .
        .
}
```

Cからの呼び出し

```
#include <stdio.h>
extern int _lc_u_bstart[];
extern int _lc_u_bend[];
int main()
{
  printf( "Size of cluster ram is %d¥n",
              (long)_lc_u_bend -
              (long)_lc_u_bstart );
}
```

__lc_ub_*identifier*, __lc_ue_*identifier*

構文：

```
extern int _lc_ub_identifier[ ];
extern int _lc_ue_identifier[ ];
```

説明：

これらのロケータラベルは、ユーザがDELFEEキーワード**reserved label=**を使用して定義することができます。このロケータラベルは、予約領域の最初と最後を指定します。*identifier*は予約領域の名前であり、DELFEEファイルで、接頭辞**__lc_ub_**または**__lc_ue_**を付けずに定義する必要があります。アセンブラからは接頭辞**__lc_ub_**および**__lc_ue_**を付けてラベルを参照することができ、Cからは接頭辞**_lc_ub_**および**_lc_ue_**（最初のアンダースコアが1つ）を付けて参照できます。

例：

記述ファイルの内容

```
block address_range {
        cluster ram {
                attribute w;
                amode data {
                        section selection=w;
                        reserved label=xvwbuffer length=0x10;
                        // Start address of reserved area is
                        // label __lc_ub_xvwbuffer
                        // End address of reserved area is
                        // label__lc_ue_xvwbuffer
                }
        }
}
```

Cからの呼び出し

```
#include <stdio.h>
extern int _lc_ub_xvwbuffer[];
extern int _lc_ue_xvwbuffer[];
int main()
{
  printf( "Size of reserved area xvwbuffer is %d¥n",
               (long)_lc_ue_xvwbuffer -
               (long)_lc_ub_xvwbuffer );
}
```

5 DEscriptive Language for Embedded Environment

5.1 はじめに

組み込み環境では、使用可能なメモリの正確な記述や、ロケータの動作のコントロールに関する情報が、アプリケーションの成功にとって重要になります。たとえば、アプリケーションを、異なるメモリコンフィグレーションを持つプロセッサに移植する場合にもこれが必要になる他、セクションのロケーションを調節して高速メモリチップを最大限に活用する場合にも必要になります。

このような目的のため、DELFEE(DEscriptive Language For Embedded Environments)言語が設計されました。

5.2 入門

5.2.1 はじめに

この節では、DELFEE記述言語について簡単に紹介します。この節の最終目的は、DELFEEの概要を説明し、DELFEE記述言語についての基本事項および基本的な記述ファイルの内容を紹介することです。詳細な説明と例については、以降の節で紹介します。

5.2.2 基本構造

DELFEE言語は、コードセクションやデータセクションが実際のメモリチップのどこに配置されるかを記述するものです。この言語は、仮想的ワールド(ソフトウェア)と物理的ワールド(ハードウェアコンフィグレーション)との間のインタフェースを定義します。

仮想ワールドには、アセンブラ言語で記述されたコードセクションとデータセクションが存在します。セクションには名前の他、書き込み可能や読み取り専用などの属性、アドレッシング空間内のアドレス、アドレッシングモードなどを指定することができ、これらによって、ロケートされるアドレス空間を記述します。

物理ワールドには、命令をメモリチップから読み込んで解釈する実際のプロセッサが存在します。DELFEE言語を使用することにより、メモリチップの種類(ROM/RAM、高速/低速)、使用可能なメモリなどを考慮した上で、正しいアドレスにコードセクションとデータセクションを配置するようロケータに命令することができます。また、DELFEE言語を使用すると、同じアプリケーションをさまざまなハードウェア構成に対応するよう調整することもできます。

DELFEE言語では、仮想ワールドと物理ワールドの間のインタフェースは、次の3つの部分で記述します。

1. software部分(`*.dsc`)

   software部分は仮想ワールドに属しており、データセクションとコードセクションの順序を記述します。software部分は、アプリケーションごとに異なり、空にすることもできます。

2. cpu部分(`*.cpu`)

   cpu部分は仮想ワールドと物理ワールドの間のインタフェースです。この部分には、仮想ワールドのアプリケーション依存部分(アドレッシングモードからアドレッシング空間へのアドレッシング変換)および物理ワールドのコンフィグレーション非依存部分(オンチップメモリ、アドレスバス)が含まれます。cpu部分は、アプリケーションと構成に依存しません。

3. memory部分(`*.mem`)

   メモリは、物理ワールドに属します。この部分には、外部メモリの記述が含まれます。memory部分は、さまざまなコンフィグレーションごとに異なり、空にすることもできます(外部メモリがない場合)。

software部分とmemory部分は空にすることができますが、cpu部分は常に定義する必要があります。

DELFEE言語は、記述ファイルと呼ばれる特殊ファイルで使用されます。DELFEE記述言語では、さまざまな部分を次の構文で定義します。

```
software {
        layout {
                // ordering of sections
        }
}

cpu {
        // mapping of addressing modes to address space
        // defining address space
        // mapping of address space to actual busses
        // defining on-chip memory
}

memory {
        // description of external memory
}
```

便宜上、cpu部分とmemory部分は異なるファイルに入れます。こうすることによって、アプリケーションごとに異なるlayout部分を使用できる他、コンフィグレーションごとに異なるmemory部分を使用できるようになります。これらのファイルは次の構文を使用してインクルードすることができます。

```
cpu filename   // include cpu part defined in file filename
mem filename   // include memory part defined in file filename
```

5.3 cpu部分

5.3.1 はじめに

cpu部分には、記述ファイルのうち、アプリケーションとコンフィグレーションに依存しない部分が含まれます。この部分は、アセンブラ言語(仮想アドレス)からチップ(物理アドレス)へのアドレス変換を定義します。変換を記述するため、DELFEEは次の4つの主要なレベルを識別します。

1. アドレッシングモードの定義。アドレッシングモードは、アドレス空間のサブセットです。アドレス空間内のアドレス範囲を定義します。
2. アドレス空間の定義。アドレス空間は、使用可能なアドレス範囲全体を示します。
3. バスの定義。
4. (オンチップ)メモリチップの定義。

アドレス変換は、アドレッシングモードから、アドレス空間とバスを経由してチップまでの範囲で定義されます。アドレッシングモードとバスはネストできますが、アドレス空間とチップはネストできません。

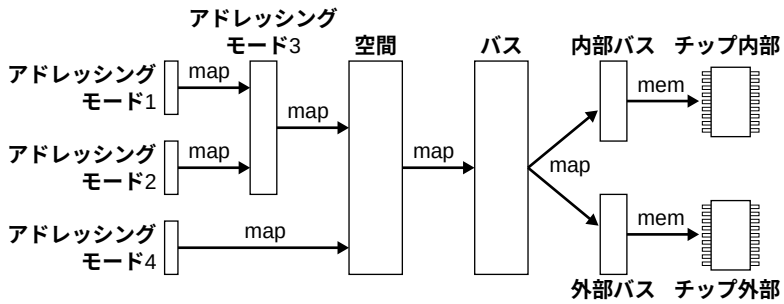


図5.3.1.1 アドレス変換

アドレッシングモードとアドレッシング空間は仮想部分に属し、バスとチップは物理部分に属します。次の節では、アドレス空間、およびアドレス空間のサブセットであるアドレッシングモードについて説明します。その後、物理サイド(ハードウェア構成)について説明し、さらに、使用できるバスとチップについて説明します。

次の例は、cpu部分を示しています。これは、定義を紹介するために作成した架空の例です。この例を見ると、アドレッシングモード定義、アドレス空間定義、バス定義、オンチップメモリ定義がそれぞれ識別できます。

```
  cpu {
          //
          // addressing mode definitions
          //
          amode near_code {
                  attribute Y1;
                  mau 8;
                  map src=0 size=1k dst=0 amode = far_code;
          }
          amode far_code {
                  attribute Y2;
                  mau 8;
                  map src=0 size=32k dst=0 space = address_space;
          }
          amode near_data {
                  attribute Y3;
                  mau 8;
                  map src=0 size=1k dst=0 amode = far_data;
          }
          amode far_data {
                  attribute Y4;
                  mau 8;
                  map src=0 size=32k dst=32k space = address_space;
          }
          //
          // space definitions
          //
          space address_space {
                  mau 8;
                  map src=0   size=32k dst=0   bus=address_bus label=rom;
                  map src=32k size=32k dst=32k bus=address_bus label=ram;
          }
          //
          // bus definitions
          //
          bus address_bus {
                  mau 8;
                  mem addr=0   chips=rom_chip;
                  map src=0x100 size=0x7f00  dst=0x100 bus=external_rom_bus;
                  mem addr=32k chips=ram_chip;
                  map src=0x8100 size=0x7f00 dst=0x100 bus=external_ram_bus;
          }
          //
          // internal memory definitions
          //
          chips rom_chip  attr=r mau=8 size=0x100;  // internal rom
          chips ram_chip  attr=w mau=8 size=0x100;  // internal ram
  }
```

5.3.2 アドレス変換：mapとmem

DELFEEでは、2つのレベル(ソースレベルとデスティネーションレベル)間のメモリ変換を、次の2種類の方法で記述します。

1. **map**キーワード。これは、アドレッシングモード、アドレス空間、バスの間(チップは該当しない)でアドレス変換するためのものです。
2. **mem**キーワード。これは、バスとチップの間のアドレス変換を記述するためのものです。**mem**は**map**を簡単にしたものです。

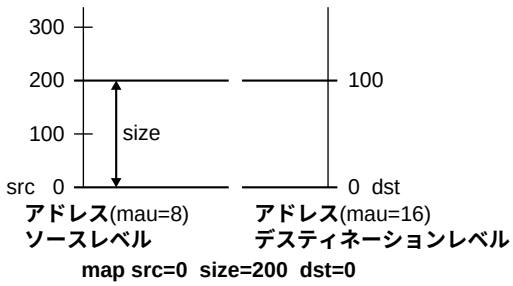


図5.3.2.1 マップアドレス変換

マップ定義の一般的な構文は、次のようになります(図5.3.2.1参照)。

map **src=***number* **size=***number* **dst=***number destination_type*=*destination_name optional_specifiers***;**

src ソースレベルの開始アドレス。アドレッシングモードとアドレス空間との間でアドレス変換を行う場合、ソースレベルがアドレッシングモードで、デスティネーションレベルがアドレス空間になります。

size ソースレベルの長さ。

dst デスティネーションレベルの開始アドレス。

destination_type

デスティネーションタイプはマッピングが使用されるコンテキストに依存し、次の3種類のいずれかのタイプになります。

1. **amode** アドレッシングモードのコンテキストで使用できます。
2. **space** アドレッシングモードのコンテキストで使用できます。
3. **bus** アドレス空間、バスのコンテキストで使用できます。

optional_specifiers

オプションの識別子も、使用されるコンテキストに依存します。

1. **label** アドレス空間のコンテキストでのみ使用でき、software部分のブロック定義で参照値として必要になります(5.4.5節を参照)。

   label = *name***;**

2. **align** これは、すべてのセクションが指定された値に揃えられることを示します。

   align = *number***;**

3. **page** これは、すべてのセクションが指定されたページサイズ内に入れられることを示します。

   page = *number***;**

ソースレベルとデスティネーションレベルの両方に、最小アドレス可能単位(MAU、アドレスを使用してアクセスできる記憶域の最小量をビット単位で示したもの)の数値で表現されるアドレス範囲があります。マッピングの場合のみ、アドレスマッピングの範囲とデスティネーションを記述することができます。実際の変換も、アドレスがアクセスできるメモリ単位に依存します。最小アドレス可能単位8ビット(mau=8)のソースレベルが、最小アドレス可能単位16ビット(mau=16)のデスティネーションレベルにマッピングされる場合、デスティネーションレベルのサイズ(アドレス範囲で表現)は、元のサイズの半分になります。そのため、図5.3.2.1の例では、デスティネーションレベルのサイズが100になります。

*level1*から*level2*方向にmapが存在する場合、マップ定義は次のようになります。

end_address of level2 = *dst* + (*size* * *mau of level1* / *mau of level2*)

memの記述は、実際のところ、**map**の記述を簡単にしたものです。アドレス変換の長さは、チップサイズが元になり、デスティネーションアドレスは常に0になります。**mem**は、バスをチップにマッピングするときに使用されます。

構文は次のようになります。

mem **addr=***number* **chips=***name***;**

addr チップの開始アドレスロケーション。

chips アドレスnumberにロケートされるチップの名前。

5.3.3 アドレス空間

仮想ワールドと物理ワールドとのリンクは、アドレス空間、およびそれが内部アドレスバスにマッピングされる方法を記述することで行います。

アドレス空間は、命令セットがアクセスできるアドレスの完全な範囲で定義します。一部の命令セットは、複数のアドレス空間をサポートします(たとえばデータ空間とコード空間)。

アドレス空間は、次の構文で記述します。

```
space name {
        mau number;
        map src=number  size=number  dst=number  bus=bus_name  label=name;
                // :
                // more maps
}
```

space 記述ファイルでアドレス空間を参照するときに使用される名前を定義します。

mau 最小アドレス可能単位。アドレスを使用してアクセスできる記憶域の最小量を示したものです(ビット単位)。

map このアドレス空間のアドレスの範囲から*bus_name*で定義されたバスへのマッピングを指定します。アドレスの範囲は、**src**および**length**で定義し、バスのオフセットは**dst**で定義されます(アドレス空間をマッピングするバスは異なるMAU値になっていることがあるが、その場合バスの範囲の長さが変わる)。アドレス空間は、バスのみにマッピングできます。

通常、アドレス空間のアドレスは、バス上の同じアドレスに対応しています。その場合、**src**と**dst**が同じ値になります。

前の例では、空間定義が1つだけありました。

```
space address_space {
        mau 8;
        map src=0   size=32k dst=0   bus=address_bus label=rom;
        map src=32k size=32k dst=32k bus=address_bus label=ram;
}
```

この例では、空間の名前が`address_space`になっています。**amod**定義はこれをマッピング先の名前として使用します。最小アドレス可能単位(MAU)は8ビットに設定されています。ラベル`rom`と`ram`は、software部分の**block**定義によって使用されます。これについては、5.4.5節で説明します。

5.3.4 アドレッシングモード

アドレッシングモードは、アドレス空間のアドレス範囲を定義します。アドレッシングモードには通常、メモリのビットアドレス可能部分、コードセクション専用の部分、ゼロページなど、特殊な特性があります。アドレッシングモードは、命令セットによって定義されます。アドレッシングモードをDELFEE言語で定義するときの構文は次のようになります。

アドレス空間は、次の構文で記述します。

amode *name* **{**
　　mau *number***;**
　　attr Y *number***;**
　　map src=*number* **size=***number* **dst=***number* **amode** | **space=***name***;**
}

amode アドレッシングモードを参照するときに使用される名前。オブジェクトファイルの場合、セクションのアドレッシングモードは**Y***number*でコード化されます。これは、アドレッシングモードに対して指定された*name*が、セクションではなく、記述ファイル内だけで有効であることを示しています。

mau 最小アドレス可能単位。アドレスを使用してアクセスできる記憶域の最小量を示したものです(ビット単位)。

attr Y アドレッシングモード番号。(アセンブラが生成する)コードセクションまたはデータセクションにはすべて、それが属するアドレッシングモードを指定するための数値があります。DELFEE記述ファイルでは、アドレッシングモードを識別するためにこの数値が使用されています。セクションの変換と混同される可能性があるため、この数値は変更することができません。

map このアドレッシングモードから他のアドレッシングモード(**amode**)またはアドレス空間(**space**)に対するマッピングを定義します。

以下は、2つのアドレッシングモード定義の例です。

```
amode near_data {
        attribute Y3;
        mau 8;
        map src=0 size=1k dst=0 amode=far_data;
}
amode far_data {
        attribute Y4;
        mau 8;
        map src=0 size=32k dst=32k space=address_space;
}
```

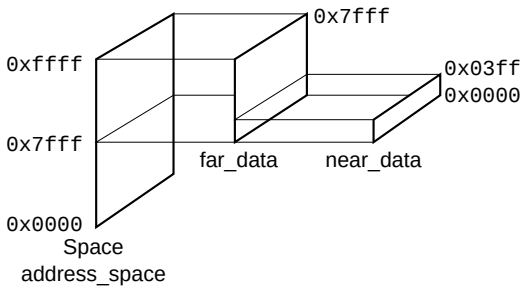


図5.3.4.1　アドレッシングモードのマッピング

この例では、アドレッシングモードに`near_data`および`far_data`という名前が付いています。これらは、それぞれアドレッシングモード番号`Y3`および`Y4`で識別されます。最小アドレス可能単位(MAU)は8ビットに設定されています。アドレッシングモード`near_data`は、アドレッシングモード`far_data`にマッピングされ、`far_data`は、アドレス空間`address_space`にマッピングされています。`address_space`は、前の節で紹介したアドレス空間です。

5.3.5 バス

busキーワードはCPUのバス構成を記述します。このキーワードは基本的に、アドレス空間からチップへのアドレス変換を記述します。構文は次のようになります。

bus *name* **{**
　　mau *number***;**
　　map src=*number* **size=***number* **dst=***number* **bus=***name***;**
　　mem addr=*number* **chips=***name***;**
}

bus　バスを参照するときに使用される名前。

mau　最小アドレス可能単位。アドレスを使用してアクセスできる記憶域の最小量を示したものです(ビット単位)。

map　他のバスへのマッピング。

mem　メモリチップへのマッピング。

以下は、バス定義の例です。

```
bus address_bus {
        mau 8;
        mem addr=0   chips=rom_chip;
        map src=0x100 size=0x7f00  dst=0x100 bus=external_rom_bus;
        mem addr=32k chips=ram_chip;
        map src=0x8100 size=0x7f00 dst=0x100 bus=external_ram_bus;
}
```

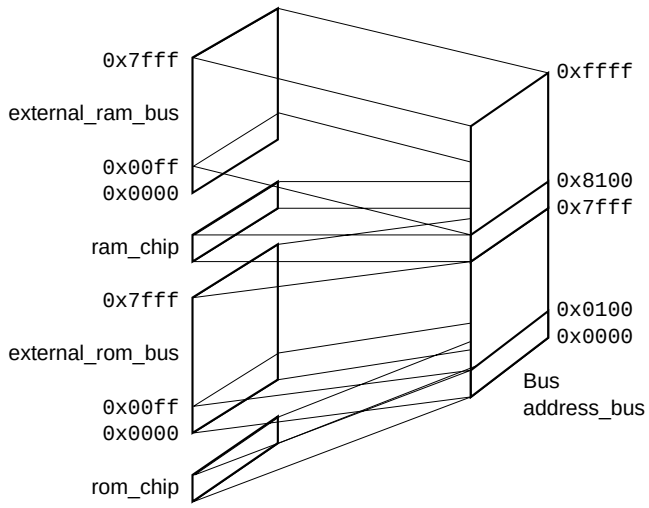


図5.3.5.1　バスのマッピング

この例では、アドレスバスに`address_bus`という名前が付いています。最小アドレス可能単位(MAU)は8ビットに設定されています。内部メモリチップ`rom_chip`は、バスのアドレス0にロケートされ、チップ`ram_chip`は、アドレス`32k`にロケートされます。

他のバスに対するアドレスマッピングには、`external_rom_bus`に対するものと`external_ram_bus`に対するものの2つがあります。

最初のマッピングでは、`address_bus`(`src=0x100 size=0x7f00`)のアドレス`0x100-0x7ff`が、`0x100`(`dst=0x100`)で始まる`external_rom_bus`のアドレスに変換されます。

2番目のマッピングでは、`address_bus`(`src=0x8100 size=0x7f00`)のアドレス`0x8100-0xffff`が、`0x100`(`dst=0x100`)で始まる`external_ram_bus`のアドレスに変換されます。2番目のマッピングは、ROMではなくRAMにマッピングされます。そのため、両方のデスティネーションアドレスが同じになっています。

5.3.6 チップ

chipsキーワードは、メモリチップを記述します。構文は次のようになります。

chips *name* **attr=***letter_code* **mau=***number* **size=***number***;**

chips チップを参照するときに使用される名前。

attr 文字コード(letter_code)でチップの属性を定義します。

letter_code 次のいずれかの属性になります。
r 読み取り専用メモリ。
w 書き込み可能メモリ。
s 特殊メモリ(ロケートできない)。

mau 最小アドレス可能単位。アドレスを使用してアクセスできる記憶域の最小量です(ビット単位)。

size チップのサイズ(0～*size*までのアドレス範囲)。

以下は、2つのチップ定義の例です。

```
chips rom_chip  attr=r mau=8 size=0x100;   // internal rom
chips ram_chip  attr=w mau=8 size=0x100;   // internal ram
```

この例では、チップに`rom_chip`および`ram_chip`という名前がついています。最小アドレス可能単位(MAU)は8ビットに設定されています。両方のチップのサイズは、`0x100` MAU(=256バイト)です。ROM、RAMということばから推測できるように、チップ`rom_chip`は読み取り専用で、チップ`ram_chip`は書き込み可能です。

5.3.7 外部メモリ

前の節で紹介した構文を使用すると、アドレス空間から外部メモリチップへのマッピングを定義することができます(メモリがオンチップであるかどうかについては、実際にはDELFEEが認識することはない)。ただしこれはあまりお勧めできません。保守性と柔軟性の理由から、内部(静的)メモリ部分を外部(可変)メモリ部分と分けておくのが良い方法です。外部メモリの操作方法については、"5.5 memory部分"で説明します。

cpu部分では、外部バスへのマッピングのみ定義する必要があります。これは、後でmemory部分で定義することができます。次の例には、2つの外部バス`external_ram_bus`と`external_rom_bus`への参照が含まれています。

```
bus address_bus {
        mau 8;
        mem addr=0   chips=rom_chip;
        map src=0x100 size=0x7f00  dst=0x100 bus=external_rom_bus;
        mem addr=32k chips=ram_chip;
        map src=0x8100 size=0x7f00 dst=0x100 bus=external_ram_bus;
}
```

5.4 software部分

5.4.1 はじめに

software部分には、次の2つの主要な部分があります。

1. load_mod
2. layout記述

```
software {
        load_mod start = start_label;

        layout {
                // ordering of sections
        }
}
```

5.4.2 ロードモジュール

キーワード**load_mod**は、プログラムの開始ラベルを定義します。プログラムの開始ラベルはコードの開始点を示し、リセットベクタがこのラベルをポイントします。このラベルが参照されない場合、ロケータは警告を生成します。

```
load_mod start = start_label;
```

5.4.3 layout記述

layout定義は、省略することができます。layout定義を省略した場合、ロケータは、cpu部分(5.3節を参照)にあるamode(アドレッシングモード)のDELFEE記述に基づいてlayout定義を生成します。ただしこれは、(スタックやヒープのような)セクションをロケートするときの順序を制御するときに使用できません。layout部分を定義する場合、ロケータはこの記述を使用します。

layout部分は、DELFEE言語でもっとも難しい部分と言えるでしょう。この部分は、セクションを正しくロケートするときに必要な情報を、ロケートアルゴリズムに提供する目的で設計されています。いくつかの例を使用することで、DELFEE言語を使用してロケートアルゴリズムに影響を与える方法を紹介します。

全体を把握しやすくするため、まず、layout部分の例を示します。

```
layout {
        space address_space {
                block rom {
                        cluster first_code_clstr {
                                attribute i;
                                amode near_code;
                                amode far_code;
                        }
                        cluster code_clstr {
                                attribute r;
                                amode near_code {
                                        section selection=x;
                                        section selection=r;
                                }
                                amode far_code {
                                        table;
                                        section selection=x;
                                        section selection=r;
                                        copy;    // locate rom copies here
                                }
                        }
                }
```

```
                block ram {
                        cluster data_clstr {
                                attribute w;
                                amode near_data {
                                        section selection=w;
                                }
                                amode far_data {
                                        section selection=w;
                                        heap;
                                        stack;
                                }
                        }
                }
        }
}
```

layout定義は、次の構文で定義します。

layout {
    // space definitions
}

最初に、layout定義の中で指定されるさまざまなレベルについて説明します。

space このレベルは、layoutレベルの中でのみ使用できます。cpu部分にあるspace定義と同じspaceレベルがあります。

block このレベルは、spaceレベルの中でのみ使用できます。cpu部分にあるspace定義で定義されているマッピングと同じblockレベルがあります。

cluster このレベルは、blockレベルの中でのみ使用できます。blockには複数のclusterがある場合もあります。主な目的は、(コード/データ)セクションをグループ化することです。ロケータは、指定された順序でそれぞれのクラスタをロケートします。

amode このレベルは、clusterレベルの中でのみ使用できます。**amode**はcpu部分の**amode**定義に対応します。**amode**内では、データ/コードセクションがロケートされる順序を指定することができます。

4つのレベルは、大きく2つのグループに分けることができます。**space**および**block**定義はアドレス範囲に対応し、**cluster**および**amode**定義はセクション(のグループ)に対応します。

次の節では、まず**space**および**block**定義について説明します。その次の節では、セクションの一定のグループを選択する方法と、**cluster**および**amode**定義でこれを使用する方法について説明します。

5.4.4 space定義

5.3.3節で、cpu部分のspaceのアドレス変換についてすでに定義しています。5.3.3節の例では、次のspaceが定義されています。

```
space address_space {
        mau 8;
        map src=0   size=32k dst=0   bus=address_bus label=rom;
        map src=32k size=32k dst=32k bus=address_bus label=ram;
}
```

layout定義では、cpu部分で定義されているすべてのspaceについて記述する必要があります。

spaceレベルは、必ずlayout定義の中に入れ、spaceレベルの中には、blockレベルのみ(複数指定可)を入れることができます。

spaceの名前は、cpu部分のspace定義に対応している必要があります。

構文は次のようになります。

space *name* **{**
    // block definitions
}

以下は、software部分のspace定義の例です。

```
space address_space {
        block rom {
                ....
        }
        block ram {
                ....
        }
}
```

この例では、space `address_space`で、block `rom`とblock `ram`の2つのブロックが定義されています。

5.4.5 block定義

block定義を使用すると、チップサイズに基づいてセクションに境界を設定できるようになります。

ブロックは、メモリの物理領域を参照します。block記述の範囲内では、選択されるセクションのみを使用することができます。結果的に、ブロックにより、セクションをロケートできる範囲が決まります。

ブロックの物理アドレス範囲は、実際にはラベルの付いたマッピングによってcpu部分で定義されています。

```
space address_space {
        mau 8;
        map src=0   size=32k dst=0   bus = address_bus label = rom; //<--
                                                    // --> block name; rom
        map src=32k size=32k dst=32k bus = address_bus label = ram; //<--
                                                    // --> block name; ram
}
```

block記述の名前は、cpu部分の**space**定義にある**map**定義のラベルに対応させる必要があります。**block**定義は、必ず**space**定義の中に入れ、**block**定義の中には、clusterレベルのみ(複数指定可)を入れることができます。

構文は次のようになります。

block *name* **{**
    // cluster definitions
}

以下は、software部分のbus定義の例です。

```
block rom {
        cluster first_code_clstr {
                ...
        }
        cluster code_clstr {
                ...
        }
}
```

この例では、block `rom`で、cluster `first_code_clstr`とcluster `code_clstr`の2つのクラスタが定義されています。

5.4.6 セクションの選択

前の節では、block定義によってアドレス範囲を定義する方法について説明しました。次に、これらのブロックに入れるセクションを選択します。DELFEEには、ロケートの順序を定義するレベルとして次の2つがあります。

1. cluster
2. amode

ロケート順を定義するとき、セクションやセクションのグループを指定するために、ある種のハンドルが必要になります。DELFEEでは、次のようなセクションの特性が認識されます。

セクションの名前

特定のセクションを示す固有の名前です。

セクションの属性

アセンブラまたはコンパイラによってセクションの属性が指定されます。指定可能な属性には、表5.4.6.1のようなものがあります。属性を選択することにより、セクションのグループを選択します。属性は、**by1w**などのような属性文字列でグループ化することができます。

アドレッシングモード

すべてのセクションには、(cpu部分で定義されたものと同じ)アドレッシングモードがあります。

表5.4.6.1 セクションの属性

attr	意 味	説 明
W	書き込み可能	RAMにロケートする必要があります。
R	読み取り専用	ROMにロケートすることができます。
X	実行専用	ROMにロケートすることができます。
Z	ゼロページ	ゼロページにロケートする必要があります。
Y*num*	アドレッシングモード	アドレッシングモード*num*でロケートする必要があります。
A	絶対	すでにアセンブラによってロケートされています。
B	ブランク	セクションを"0"で初期化する必要があります(クリア)。
F	未充てん	セクションが充てんされておらずクリアもされていません(スクラッチ)。
I	初期化	セクションをROMで初期化する必要があります。
N	今	セクションが、(NもPもない)通常のセクションの前にロケートされます。
P	延期	セクションが、(NもPもない)通常のセクションの後にロケートされます。

DELFEEでは、セクション(のグループ)を指定するとき、次の構文を使用します。

1. セクション属性でグループを選択する場合。

   section selection = *attr***;**

2. 名前でセクションを選択する場合。

   section *name***;**

3. 特殊なセクションを選択する場合。

   heap; / /locate heap here
   stack; / /locate stack here
   table; / /locate copy table here
   copy; / /locate all initial data here
   copy *name***;** / /locate initial data of the named section here

4. セクションを作成する場合。

   reserved label=*name* **length=***number***;**

DELFEEでは、属性でセクションを選択する代わりに、属性によって、あるセクションを除外することもできます。属性を除外するときは"-"(マイナス記号)をattrの前に置きます。

そのため、次の例は、

```
section selection=attr1-attr2
```

属性*attr1*を持ち属性*attr2*を持たないセクションのグループを選択します。

5.4.7 クラスタ定義

指定したセクションをグループに置くときは、クラスタが使用されます。ロケータは指定された順序でクラスタを処理します。クラスタを使用すると、選択したセクションのグループを作成し、ロケートの優先順位を上げることができます。

セクションを、あるクラスタの部分として指定する方法は、いくつかあります。正確な規則とその優先順位については、"5.4.10 セクション配置アルゴリズム"で説明しています。主な方法は、次の3つです。

1. attribute
2. section selection=
3. amode定義

次の例を参照してください。

```
layout {
        space address_space {
                block rom {
                        cluster first_code_clstr {
                                attribute i;
                                amode near_code;
                                amode far_code;
                        }
                        cluster code_clstr {
                                attribute r;
                                amode near_code {
                                        section selection=x;
                                        section selection=r;
                                }
                                amode far_code {
                                        table;
                                        section selection=x;
                                        section selection=r;
                                        copy;   // locate rom copies here
                                }
                        }
                }
        }
}
```

この例では、特別なクラスタ`first_code_clstr`が作成されています。配置アルゴリズム(5.4.10節)を使用することで、属性"i"を持つセクションがクラスタ`first_code_clstr`に置かれるため、`code_clstr`のセクションより優先順位が高くなります。

構文は次のようになります。

cluster *name* **{**
        // section selections
}

クラスタ内では自由度の小さいセクションが最初にロケートされます。自由度は、セクションをロケートできるアドレスによって定義されます。

5.4.8 amode定義

クラスタ内では、アドレッシングモード(amode)を指定することができます。cpu部分(5.3.4節)では、すべてのアドレッシングモードにアドレス範囲が割り当てられており、layout部分では、セクションのグループを識別するためにアドレッシングモードが使用されています。

構文は次のようになります。

```
        :
amode name {
        section selection = attr;
                :
}
        :
```

ロケートの順序は、指定された順序によって決定されます。

たとえば、すべての書き込み可能なセクションを最初にロケートして、その後にヒープ、スタックを順にロケートしたい場合を考えます。DELFEE言語では、これを次のようにして指定します。

```
        :
section selection = w;   // 'w' means writable sections
heap;
stack;
        :
```

5.4.9 amodeでのセクションの処理

前の節では、**amode**定義内でセクションの順序を設定する方法について説明しました。DELFEEでは、コードセクションとデータセクションのロケートをさらに詳細に調節するため、その他にもキーワードを認識するようになっています。

amode定義には、次のようなキーワードを入れることができます。

キーワード	説　明
section	セクションまたはセクションのグループを選択します。
selection	セクションをグループ化するための属性を指定します。
attribute	属性を割り当てます(クラスタにペースト)。
copy	セクションのROMコピーを名前で選択します。または、一般的にすべてのROMコピーを選択します。
fixed	セクションを固定アドレスのまわりにロケートするよう強制します。
gap	セクションがロケートされていないアドレス範囲にギャップを生成します。
reserved	ロケータラベルを使用して参照できるメモリ領域を予約します。
heap	ヒープの配置と属性を定義します。
stack	スタックの配置と属性を定義します。
table	コピーテーブルの配置と属性を定義します。
assert	ユーザ定義の表明です。
length	スタック、ヒープ、物理ブロック、予約空間を指定します。

すべてのキーワードについては、"5.6 DELFEEキーワードリファレンス"で説明します。

5.4.10 セクション配置アルゴリズム

セクションを参照する方法には、いくつかの方法があります。セクションは、特定の属性に基づくグループとして参照することもできますし、名前で個別に参照することもできます。セクションがlayout部分のどこに配置されているか確認するとき、DELFEEでは、次のアルゴリズムが使用されます。

1. 最初にセクション名でselectionを探します。
2. 見つからない場合、一致するamodeブロック内の"section selection="を検索します。
3. 見つからない場合、一致するamodeブロック外の"section selection="を検索します。
4. 見つからない場合、正しい"amode=...,...,..;"と正しい属性を持つクラスタを検索します。
5. 見つからない場合、正しい属性を持つクラスタを検索します。
6. 見つからない場合、属性チェックを緩和して、最初からやり直します。

属性を緩和するとき、次の規則に従います。

1. stack、heap、reservedの場合、指示をオフにして再度実行します。
2. 属性が"f"(未充てん)の場合、"f"をオフにして再度実行します。
3. 属性が"b"(クリア)の場合、"b"をオフにして再度実行します。
4. 属性が"i"(初期化)の場合、"i"をオフにして再度実行します。
5. 属性が"x"(実行可能コード)の場合、"x"をオフにし"r"(読み取り専用)をオンにして、再度実行します(実行可能セクションを読み取り専用メモリに配置しようとする)。
6. 属性が"r"(読み取り専用)の場合、"r"をオフにし"w"(書き込み可能)をオンにして、再度実行します(読み取り専用セクションを書き込み可能メモリに配置しようとする)。

5.5 memory部分

5.5.1 はじめに

memory部分は、メモリコンフィグレーションの可変部分を定義します。この部分は異なるファイルに入れることができます。こうすることで、さまざまなメモリコンフィグレーションを簡単に切り換えられるようになります。マッピングに使用する構文は、cpu部分の場合と同じです。

"5.3 cpu部分"の例では、外部バスに対する参照が2つありました。

```
bus address_bus {
        mau 8;
        mem addr=0   chips=rom_chip;
        map src=0x100 size=0x7f00  dst=0x100 bus=external_rom_bus;
        mem addr=32k chips=ram_chip;
        map src=0x8100 size=0x7f00 dst=0x100 bus=external_ram_bus;
}
```

memory部分では、external_rom_busおよびexternal_ram_busの2つのバスに対する記述を定義する必要があります。5.3.5節および5.3.6節の記述を使用してバスとチップを指定すると、memory部分は次のようになります。

```
memory {
        bus external_rom_bus {
                mau 8;
                mem addr=0 chips=xrom;
        }

        chips xrom attr=r mau =8 size=0x8000;

        bus external_ram_bus {
                mau 8;
                mem addr=0 chips=xram;
        }
        chips xram attr=w mau =8 size=0x8000;
}
```

5.6 DELFEEキーワードリファレンス

この節では、記述ファイルで使用できるすべてのキーワードについてアルファベット順に紹介します。キーワードによっては、最小4文字に省略できるものもあります。

.addr

構文：

.addr （software部分）

説明：

定義済みのラベル**.addr**には、現在のアドレスが含まれます。

例：

```
block ram {
        cluster data_clstr {
                attribute w;
                amode near_data {
                        section selection=w;
                        assert ( .addr < 256, "page overflow");
                                // if the condition is false,
                                // the locator generates an error with
                                // the text as message
                }
                ...
        }
}
```

address

構文：

address = *address* （すべての部分）
addr = *address* （省略形）

説明：

メモリ内で絶対アドレスを指定します。

例：

cpu部分またはmemory部分

```
bus address_bus {
        mau 8;
        mem addr=0   chips=rom_chip;
        ...
        mem addr=32k chips=ram_chip;
        ...
}
```

software部分

```
block rom {
        ...
        cluster code_clstr {
                attribute r;
                amode near_code {
                        section selection=x;
                        section selection=r;
                        section .string address = 0x0100;
                }
                ...
        }
}
```

上記の例にある**amode**定義内のロケート順は固定されています。属性セクション"x"や"r"を持つセクションは、強制的にセクション.stringの前にロケートされます。

この固定順が気に入らない場合、他の**amode**定義で絶対アドレス指定を行うことができます。

例：

```
amode near_code {
        section .string address = 0x0100;
}

amode near_code {
        section selection=x;
        section selection=r;
}
```

amode

構文：

amode *identifier*[**,** *identifier*]...**{** *amod_description* **}** （定義） （cpu部分またはmemory部分）
amode **=** *identifier* （参照）

amode *identifier*[**,** *identifier*]...; （software部分）
amode *identifier*[**,** *identifier*]...**{** *section_blocks* **}**

説明：

キーワード**amode**はすべての部分で使用することができます。cpu部分またはmemory部分では、**amode**を使用して、特定のアドレス空間にアドレッシングモードやレジスタバンクをマッピングすることができます(定義)。**amode=**を指定した場合、特定のアドレッシングモードを、以前定義したアドレッシングモードにマッピングすることができます(参照)。*amod_description*(cpu部分)で使用できるキーワードは、**attribute**、**map**、**mau**のみです。キーワード**attribute Y***num*は、個別のアドレッシングモードを識別します。

software部分では、セクションのロケート順を変更するために、**amode**をクラスタ定義の一部として使用することができます。"5.4.10 セクション配置アルゴリズム"も参照してください。

例：

cpu部分またはmemory部分から

```
cpu {
  amode near_data {
        attribute Y3;
        mau 8;
        map src=0 size=1k dst=0 amode = far_data;
                // reference
  }
  amode far_data { // definition
        attribute Y4;
        mau 8;
        map src=0 size=32k dst=32k space = address_space;
  }
```

software部分から

```
block ram {
  cluster data_clstr {
        attribute w;
        amode near_data {
          // Sections with addressing mode
          // near_data are located here
          section selection=w;
        }
        amode far_data {
          // Sections with addressing mode
          // far_data and the stack and heap
          // are located here
          section selection=w;
          heap;
          stack;
        }
  }
}
```

assert

構文：

assert (*condition* , *text* **);** （software部分）
asse (*condition* , *text* **);** （省略形）

説明：

メモリ内にある仮想アドレスの条件をテストします。表明が失敗した場合エラーが出され、"*text*"にメッセージが書き込まれます。*condition*は次のいずれかで指定します。

expr1 > *expr2*
expr1 < *expr2*
expr1 == *expr2*
expr1 != *expr2*

*expr1*および*expr2*は、任意の式またはラベルにすることができます。定義済みのラベル**.addr**には、現在のアドレスが含まれます。

例：

```
block ram {
        cluster data_clstr {
                attribute w;
                amode near_data {
                        section selection=w;
                        assert ( .addr < 256, "page overflow");
                                // if the condition is false,
                                // the locator generates an error with
                                // the text as message
                }
                ...
        }
}
```

attribute

構文：

attribute *attribute_string***;** （software部分）
attr *attribute_string***;** （省略形）
attribute = *attribute_string* （software部分）
attr = *attribute_string* （省略形）

説明：

attributeを使用すると、属性をsection、cluster、memoryの各ブロックに割り当てることができます。キーワード**selection**も参照してください。

sectionの場合、これらの属性は標準セクション属性を純粋に補うものになります。ゼロページ(Y1)、ブランク(B)、実行可能(X)などの標準セクション属性は、コンパイラ(またはアセンブラプログラムの場合アセンブラによって)によって設定されます。

sectionの後にアクション属性を追加する(**attr=**)と、セクション属性を設定したり、セクション属性を無効(-記号を付けた場合)にしたりすることができます。

属性には、次の意味があります。

num (sectionのみ)セクションを2^{num} MAUに揃えます。

Y*num* (amodeとsectionのみ)アドレッシングモードを識別します。この属性を持つセクションが、このクラスタに割り当てられます。

r (memoryとcluster)これが読み取り専用クラスタまたは読み取り専用メモリであることを示します。

w (memoryとcluster)これが書き込み可能クラスタまたは書き込み可能メモリであることを示します。

s (memoryのみ)これが特殊メモリであることを示します。この場合ロケートできません。

x (cluster/sectionのみ)クラスタ/セクションが実行可能であることを示します。

g (cluster/sectionのみ)クラスタ/セクションがグローバルであることを示します(マルチモジュール環境であることが前提)。

b (cluster/sectionのみ)クラスタ/セクションをロケートの前にクリアしなければならないことを示します。

i (sectionのみ)クラスタ/セクションをROMからRAMへコピーしなければならないことを示します。

f (cluster/sectionのみ)クラスタ/セクションが充てんもクリアもされないことを示します。これはスクラッチクラスタ/セクションと呼ばれます。

属性キーワードが省略された場合のデフォルト属性は、次のようになります。

section： アセンブラ/コンパイラから生成されたものと同じ属性。

cluster： 基本となるメモリによって示された属性。つまりROMの場合**r**、RAMの場合**w**になります。

memory： 属性が定義されていない場合、デフォルトは書き込み可能(**w**)になります。

例：

software部分から

```
layout {
   space address_space {
        block rom {
          cluster first_code_clstr {
                attribute i; // set cluster attribute
                amode near_code;
                amode far_code;
          }
        }
        block ram
          cluster ram {
                amode near_data {
                  // Default attribute of cluster
                  // data is 'w', because the
                  // memory is RAM.

                 section selection=w;
                 section selection=b attr=-b;
                  // Sections with attribute b are
                  // are located here, and
                  // attribute 'b' is switched off
                }
                .
          }
          .
        }
   }
}
```

cpu部分から

```
 amode near_data {
        attribute Y3;  //identify code with Y3
        mau 8;
        map src=0 size=1k dst=0 amode = far_data;
 }
 ...

  chips rom_chip  attr=r mau=8 size=0x100;
  chips ram_chip  attr=w mau=8 size=0x100;
        ...
        // memory attributes
```

block

構文：

block *identifier* **{** *block_description* **}**　　（software部分）

説明：

blockを使用して、メモリの物理領域の内容を定義します。使用するチップごとに**block**の記述を作成します。それぞれのブロックには、キーワード**chips**ですでに定義されているシンボル名が付いています。また、アドレス範囲全体が、ギャップを除いて線形になる限り、複数のメモリチップを1つのブロックに結合することもできます。*identifier*は、結合されているチップの数がいくつであっても、指定されたチップでメモリブロックが開始することを示します。

例：

```
layout {
        space address_space {
                block ram
                 // Memory block starting at chip ram_chip
                        cluster ram {
                         ...
                        }
                }
        }
}
```

bus

構文：

bus *identifier*[**,** *identifier*]...**{** *bus_description* **}**　（定義）　（cpu部分またはmemory部分）
bus = *identifier*　（参照）

説明：

busを使用すると、cpuにロケートされるチップに対して物理メモリアドレスを定義することができます（定義）。**bus=**を指定した場合、特定のアドレス範囲を、以前定義したアドレスバスにマッピングすることができます（参照）。**bus**の記述で使用できるキーワードは、**mem**、**map**、**mau**のみです。

例：

```
cpu {
        space address_space {
                // Specify space 'address_space' for the address_bus
                // address bus.
                mau 8;
                map src=0   size=32k dst=0   bus = address_bus label = rom;
                map src=32k size=32k dst=32k bus = address_bus label = ram;
                                                              // ref
        }

        bus address_bus {  // definition
                mau 8;
                mem addr=0   chips=rom_chip;
                map src=0x100 size=0x7f00  dst=0x100 bus = external_rom_bus;
                mem addr=32k chips=ram_chip;
                map src=0x8100 size=0x7f00 dst=0x100 bus = external_ram_bus;
        }
        ...
}
```

chips

構文：

chips *identifier*[**,** *identifier*]...*chips_description*　（定義）　（cpu部分またはmemory部分）
chips **=** *identifier*[| *identifier*]...[**,** *identifier*[| *identifier*]...]...　（参照）

説明：

chipsを使用すると、CPU上またはターゲットボード上のチップを定義することができます(定義)。チップごとに、**size**と最小アドレス可能単位(**mau**)を指定します。キーワード**attr**を使用することで、メモリが読み取り専用かどうかを定義することができます。使用できるキーワードは、**r**、**w**、**s**のみで、それぞれ読み取り専用、書き込み可能、特殊を表します。省略された場合、**w**がデフォルトになります。

キーワード**mem**の後に**chips=**を指定すると、チップがロケートされる場所を指定することができます(参照)。それぞれのチップを垂直バー"|"で区切ることにより、チップを組み合わせることができます。

例：

```
cpu {
        bus address_bus {
                mau 8;
                mem addr=0 chips=rom_chip; // ref
                ...
        }
        chips rom_chip  attr=r mau=8 size=0x100;  // def
        chips ram_chip  attr=w mau=8 size=0x100;
        ...
}
```

cluster

構文：

cluster *cluster_name* **{** *cluster_description* **}**　（software部分）

cluster *cluster_name* [**,** *cluster_name*]...**;**

説明：

softwareのlayout部分では、クラスタ名とクラスタのロケーション順を定義することができます。クラスタで有効な属性(**attribute**を参照)は、最初の構文で指定することができます。属性を指定しなかった場合、デフォルト属性**r**または**w**が自動的に設定されます。

クラスタの記述では、指定されたクラスタ内にあるセクションのロケート順を決定するだけでなく、スタックサイズとヒープサイズや予備プロセスメモリを指定したり、そのプロセスに対するラベルを定義したりすることもできます。

例：

```
space address_space {
        block rom {
                cluster first_code_clstr {
                 // The default attribute 'r' of cluster
                 // text is overruled to  'i'. All sections with attribute
                 // 'i' are located here by default.
                       attribute i;
                       amode near_code;
                       amode far_code;
                        // Sections with addressing mode
                        // near_code or far_cdoe are located here
                }
        block ram {
                cluster data_clstr {
                 // default attribute 'w' because the memory is RAM.
                 // All writable sections are located here by default.
                       attribute w; // can be omitted
                       amode near_data {
                              section selection=w;
                       }
        }
}
```

copy

構文：

copy *section_name* [**attr** = *attribute*]**;** （software部分）
copy selection = *attribute* [**attr** = *attribute*]**;**
copy ;

説明：

プログラムのスタートアップ時に、属性**i**を持つデータセクションのROMコピーが、ROMからRAMへコピーされます。**copy**を使用することで、これらのROMコピーのメモリの配置を定義することができます。セクションの名前を指定して特定のセクションを指定することができる他、特定の属性を持つセクションを選択することができます。引数を指定しない場合、ロケータはすべてのROMコピーを、指定されたロケーションに配置します。**attr=**を指定した場合、セクションの属性を変更することができます。

キーワード**copy**をまったく指定しなかった場合、ロケータはROMコピーに適した場所を探します。

キーワード**attribute**と**selection**の説明も参照してください。

例：

```
space address_space {
  block rom {
        ...
        cluster code_clstr {
                attribute r; //cluster attribute
                amode far_code {
                  table;
                  section selection=x;
                  section selection=r;
                  copy; // all ROM copies are located here
                }
        }
}
```

cpu

構文：

cpu { *cpu_description* **}** （cpu部分）
cpu *filename*

説明：

キーワード**cpu**は、記述ファイルの一番高い位置に**software**および**memory**と一緒に記述します。実際のcpu記述は、中かっこ**{ }**の間に入れます。通常、それぞれのcpu部分は製品で用意されており派生製品についても完全に記述されているため、cpu部分を変更する必要はありません。

2番目の構文は、いわゆるインクルード構文です。ロケータはファイル*filename*をオープンして、このファイルから実際のcpu記述を読み込みます。インクルードされたファイルは、再度**cpu**で開始する必要があります。*filename*には、ドライブ名を含む完全なパスを入れることができます。*filename*の一部または完全な*filename*は、環境変数に入れることもできます。指定されたファイルは、最初カレントディレクトリで検索され、次にインストールディレクトリの相対ディレクトリ`etc`で検索されます。

例：

記述ファイルの内容

```
software {
        ...
}

cpu target.cpu  //cpu part in separate file
memory target.mem
```

`.cpu`ファイルの内容のサンプルについては、5.3節を参照してください。

dst

構文：

dst = *address*　　（cpu部分またはmemory部分）

説明：

amode、**space**、**bus**記述のキーワード**map**の一部として、デスティネーションアドレスを指定します。*address*については、10進数、16進数、8進数を使用することができます。またキロ(2^{10})やメガ(2^{20})を表すとき、(標準)DELFEE接尾辞の**k**や**M**をそれぞれ使用することができます。値の単位は、デスティネーションメモリ空間のMAU(最小アドレス可能単位)によって変動します。

例：

```
cpu {
        ...
        amode near_code {
                attribute Y1;
                mau 8; // 8-bit addressable
                map src=0 size=1k dst=0 amode=far_code;
        }
}
```

fixed

構文：

fixed address = *address*;　　（software部分）
fixed addr = *address*;　　（省略形）

説明：

メモリマップの固定ポイントを定義します。ロケータは、fixed定義の前にあるセクション/クラスタ、およびfixed定義の後にあるセクション/クラスタを、できる限り固定ポイントの近くに割り当てます。

例：

```
block ram {
        cluster near_data_clstr {
          amode near_data {
                section selection=w;
                fixed  addr = 0x2000;
          }
        }
        cluster far_data_clstr;
}
```

クラスタ`far_data_clstr`は、アドレス0x2000が上限の境界になるようにロケートされ、クラスタ`near_data_clstr`はこのアドレスで開始します。同じことがセクションの場合にも当てはまります。

gap

構文：

gap;　　（software部分）
gap length = *value*;

説明：

動的なサイズを持つギャップを予約します。ロケータはメモリ空間をできる限り大きくしようとします。このキーワードは、クラスタ間にギャップを生成するときにblock記述で使用できる他、セクション間にギャップを生成するときにcluster記述で使用することができます。また**gap**キーワードは、**fixed**キーワードと組み合わせて使用することもできます。

2番目の書式は、固定長のギャップを指定するときに使用します。この書式は、block記述でのみ使用できます。

例：

```
space address_space {
        block ram {
                cluster data_clstr {
                        attr w;
                        amode near_data;
                } // low side mapping

                gap;    // balloon
                cluster stck; // high side mapping
        }
}
```

heap

構文：

heap *heap_description*; （software部分）
heap;

説明：

tableや**stack**と同じように、**heap**も特殊セクションです。このセクションは、.outファイルから作成されないで、ロケート時に生成されます。この特殊セクションのサイズをコントロールするため、heap記述の中でキーワード**length**を使用することができます。プロセスで動的メモリを入れるときも、**heap**を使用することができます。

heapは、malloc()関数が実装されている場合に限り使用することができます。

ヒープ領域の最初と最後を指定するために、2つのロケータラベルが使用されます。つまり、ヒープの最初を指定する場合__lc_bh、ヒープの最後を指定する場合__lc_ehを使用します。

記述ファイルで**heap**キーワードが指定されている場合、そのままヒープ領域が常に生成されるということにはなりません。ヒープ領域は、セクションラベル(ヒープの最初を指定する__lc_bhとヒープの最後を指定する__lc_eh)がプログラムで使用されている場合に限り割り当てられます。

heap記述では、長さの指定や属性の指定も可能です。例を参照してください。

例：

```
layout {
        space address_space {
                block ram {
                        cluster data_clstr {
                                amode far_data {
                                        section selection=w;
                                        heap length=100;
                                                // Heap of 100 MAUs
                                }
                        }
                }
        }
}
```

label

構文：

label *identifier*; （software部分）
label = *identifier*; （すべての部分）

説明：

最初の書式は、メモリ内の仮想アドレスをラベルで指定するときにスタンドアローンで使用されます。仮想アドレスは、label **__lc_u_***identifier*になります。Cレベルでは、すべてのロケータラベルの最初にアンダースコアが1つ付けられます(コンパイラがアンダースコア"_"をもう1つ追加)。

2番目の書式は、他のキーワードの一部として使用することができます。キーワード**reserved**の一部として使用することで、ラベルをアドレス範囲に割り当てることができます。アドレス範囲の最初は、label **__lc_ub_***identifier*で識別し、アドレス範囲の最後は、label **__lc_ue_***identifier*で識別します。
キーワード**label**は、**map**キーワードの一部として使用することにより、space定義で名前をメモリのブロックに割り当てることもできます。

例：

software部分から

```
block ram {
        cluster data_clstr {
                attribute w;
                amode far_data {
                        section selection=w;
                        heap;
                        stack;
                        reserved label=xvwbuffer length=0x10;
                        // Start address of reserved area is
                        // label __lc_ub_xvwbuffer
                        // End address of reserved area is
                        // label __lc_ue_xvwbuffer
                }
        }
}
```

cpu部分から

```
space address_space {
        mau 8;
        map src=0   size=32k dst=0   bus = address_bus label=rom;
        map src=32k size=32k dst=32k bus = address_bus label=ram;
}
```

layout

構文：

layout { *layout_description* **}** （software部分）
layout *filename*

説明：

layout部分では、メモリ内のセクションのレイアウトを記述します。**layout**部分では、複数のセクションをクラスタにグループ化して、これにクラスタの名前、数、順序を定義することができます。**layout**部分では、これらのクラスタが物理的なRAMおよびROMブロックに割り当てられる方法を記述します。
layout部分で使用されるspace名とblock名は、memory部分またはcpu部分に存在している必要があります。クラスタ定義には、セクション間のギャップの他、固定アドレスを入れることができます。

例：

```
software {
  layout {
    space address_space {
        block rom {
          cluster first_code_clstr {
            attribute i;
            amode near_code;
          }
        ....
```

length

構文：

length = *length*　（cpu部分、memory部分、software部分）
leng = *length*　（省略形）

説明：

キーワード**length**は、特定のメモリ領域のMAU(最小アドレス可能単位)の長さを定義するときに使用することができます。*length*は数値でなければならず、16進数、8進数、10進数のいずれかで指定することができます。通常16進数には最初に"0x"を付け、8進数には"0"を付けます。接尾辞として、キロを示す**k**やメガを示す**M**を使用することもできます。

予約メモリの長さやスタック、ヒープ、ギャップの長さを指定するときも、**length**を使用することができます。詳細については、キーワード**reserved**、**stack**、**heap**、**gap**の説明を参照してください。

例：

```
space address_space {
        block ram {
                cluster data_clstr {
                        amode far_data {
                                stack leng = 2k;
                        }
                }
        }
}
```

load_mod

構文：

load_mod *identifier* **start** = *label*;　（software部分）

load_mod start = *label*;

説明：

ロードモジュール記述を開始するとき、**load_mod**を使用します。このキーワードには、オプションのidentifierとしてロードモジュール名を続けます。この場合、.out拡張子は付けても付けなくてもかまいません。ロードモジュール自体は、起動構文でパラメータとして指定する必要があります。identifierが省略された場合、ロードモジュールはコマンド行から取得されます。

例：

```
software {
        load_mod start = __START;
}
```

または

```
software {
        load_mod hello start = __USER_start;
}
```

map

構文：

map *map_description*　（cpu部分またはmemory部分）

説明：

ソースアドレスおよびサイズとして指定されたmemory部分を、**amode**、**space**、**bus**のデスティネーションアドレスにマッピングします。値の単位は、メモリ空間のMAUによって変動します。

例：

```
cpu {
  .
  amode far_data {
         attribute Y4;
         mau 8;
         map src=0 size=32k dst=32k space=address_space;
  }
  space address_space {
         mau 8;
         map src=0   size=32k dst=0   bus = address_bus label=rom;
         map src=32k size=32k dst=32k bus = address_bus label=ram;
  }
  bus address_bus {
         mau 8;
         mem addr=0   chips=rom_chip;
         map src=0x100 size=0x7f00  dst=0x100 bus=external_rom_bus;
         mem addr=32k chips=ram_chip;
         map src=0x8100 size=0x7f00 dst=0x100 bus=external_ram_bus;
  }
  .
}
```

mau

構文：

mau *number*; （cpu部分またはmemory部分）
mau = *number*

説明：

最小アドレス可能単位を、特定メモリ領域のビット単位で指定するとき、キーワード**mau**を使用することができます。最初の書式は、**amode**、**space**、**bus**の記述でのみ使用することができます。2番目の書式は、チップの最小アドレス可能単位を指定するときに使用することができます。**mau**は、他のキーワードの値の単位にまで影響します。**mau**が指定されていない場合、デフォルト値は8(バイトアドレス可能)になります。

例：

```
cpu {
  amode near_code {
     attribute Y1;
     mau 8;     // byte addressable
     map src=0 size=1k dst=0 amode=far_code;
               // src is at address 0,
               // size is 1k byte units
               // dst is at address 0
  }
}
```

mem

構文：

mem *mem_description*; （cpu部分またはmemory部分）

説明：

メモリ内のチップの開始アドレスを定義します。memの記述で使用できるキーワードは、**address**と**chips**のみです。

例：

```
cpu {
  ...
  bus internal_bus {
    mau 8;
    mem addr=0    chips=rom_chip;
        // chip 'rom_chip' is located at memory
        // address 0
        ...
    mem addr=32k  chips=ram_chip;
        // chip 'ram_chip' is located at memory
        // address 0x8000
        ...

  }
  chips rom_chip  attr=r mau=8 size=0x100;
  chips ram_chip  attr=w mau=8 size=0x100;

}
```

memory

構文：

memory { *memory_description* **}** (memory部分)
memory *filename*

説明：

software、**cpu**と共に使用することで、**memory**は記述ファイルの主要な部分を構成します。実際のmemory部分は、中かっこ**{ }**の間に入れます。

追加メモリやCPUに統合されていない周辺機器のアドレスについて記述する場合、memory部分を使用することができます。

2番目の構文は、インクルード構文です。この場合、memory部分は個別のファイルで定義されます。インクルードされたファイルは、**memory**で開始する必要があります。*filename*には、ドライブ名を含む完全なパスを入れることができます。*filename*の一部または完全な*filename*は、環境変数れることもできます。指定されたファイルは、最初カレントディレクトリで検索され、次にインストールディレクトリの相対ディレクトリetcで検索されます。

例：

```
software {
        ...
}

cpu target.cpu
memory target.mem       //mem part in separate file
```

.memファイルの内容のサンプルについては、5.5節を参照してください。

regsfr

構文：

regsfr *filename* （cpu部分またはmemory部分）

説明：

レジスタマネージャが生成したレジスタファイルを指定し、デバッガが使用できるようにします。

例：

```
cpu {
        .
        .
        regsfr regfile.dat
        /*
         * Use file regfile.dat generated by register manager
         */
}
```

reserved

構文：

reserved *reserved_description***;** （software部分）
reserved;

説明：

メモリ空間の一定量を予約するか、メモリ空間のできる限り大きな部分を予約します。長さが指定されていない場合、メモリ割り当てのサイズは、メモリ空間のサイズ、または**reserved**割り当ての後の固定ポイント定義で制限されるサイズによって変動します。

reserved記述では、キーワード**address**、**attribute**、**label**、**length**のみを使用することができます。amode記述では、キーワード**reserved**を使用することができます。

例：

```
space address_space {
        block rom {
                cluster code_clstr {
                        amode near_code {
                          // system reserved
                          // (exception vector)
                          reserved length=0x2 addr=0x24;
                        }
        }
}
```

section

構文：

section *identifier* [**addr** = *address*] [**attr** = *attribute*]**;** （software部分）
section selection = *attribute* [**addr** = *address*] [**attr** = *attribute*]**;**

説明：

sectionは、クラスタ内のロケーションの順序を指定するため、layout部分で使用することができます。**layout**の説明も参照してください。

*identifier*は、セクションの名前です。

addr=を使用すると、セクションを絶対セクションにすることができます。

attr=を使用すると、新しい属性をセクションに割り当てたり、属性を無効にすることができます。

キーワード**address**、**attribute**、**selection**の説明も参照してください。

例：

```
space address_space {
        block ram {
                cluster data_clstr {
                        amode near_data {
                          // locate section .data here and set
                          // attribute 'w'
                          section .data attr=w;
                          section selection=b attr=-b;
                        }
                }
        }
}
```

selection

構文：

selection = *attribute*

説明：

selectionは、指定した属性を持つすべてのセクションを選択するため、キーワード**section**または**copy**の後に使用することができます。

複数の属性が指定されている場合、そのすべての属性を持つセクションのみが選択されます。マイナス記号"-"を属性の前に付けると、その属性を持っていないセクションのみが選択されます。

キーワード**attribute**、**copy**、**section**の説明も参照してください。

例：

```
space address_space {
        block ram {
          cluster data_clstr {
                amode near_data {
                 // select sections with w on and not i.
                 // (select all writable sections which
                 // are not copied from ROM)
                 section selection=-iw;
                }
          }
        }
        .
}
...
```

size

構文：

size = *size*　　（cpu部分またはmemory部分）

説明：

特定のメモリ空間について、最小アドレス可能単位(MAU)でサイズを定義するとき、キーワード**size**を使用することができます。*size*は数値でなければならず、16進数、8進数、10進数のいずれかで指定することができます。通常16進数には最初に"0x"を付け、8進数には"0"を付けます。接尾辞として、キロを示す**k**やメガを示す**M**を使用することもできます。

他のメモリ部分にマッピングするメモリ部分のサイズを指定するとき、またはチップのサイズを指定するとき、**size**を指定することができます。詳細については、キーワード**map**および**chips**の説明を参照してください。

例：

```
cpu {
  amode near_code {
        attribute Y1;  //identify near_code with Y1
        map src=0 size=1k dst=0 amode=far_code;
  }
  space address_space {
        mau 8;
        map src=0   size=32k dst=0   bus=address_bus label=rom;
        map src=32k size=32k dst=32k bus=address_bus label=ram;
  }
  chips rom_chip attr=r mau=8 size=0x100;
  chips ram_chip attr=w mau=8 size=0x100;
        // size of chips
}
```

software

構文：

software { *software_description* **}** （software部分）
software *filename*

説明：

キーワード**software**は、記述ファイルの最高レベルに記述します。実際のsoftware記述は、中かっこ**{ }**の間に入れます。
2番目の構文は、いわゆるインクルード構文です。ロケータはファイル*filename*をオープンして、このファイルから実際のsoftware記述を読み込みます。*filename*にある最初のキーワードも、**software**でなければなりません。*filename*には、ドライブ名を含む完全なパスを入れることができます。*filename*の一部または完全な*filename*は、環境変数に入れることもできます。指定されたファイルは、最初カレントディレクトリで検索され、次にインストールディレクトリの相対ディレクトリ`etc`で検索されます。

例：

記述ファイルの内容

```
software $(MY_OWN_DESCRIPTION)
```

```
cpu target.cpu
memory target.mem
```

環境変数`MY_OWN_DESCRIPTION`には、次のような内容を持つファイルの名前が含まれます。

```
software {
        load_mod start = __START;
        layout {
        .
        }
}
```

space

構文：

space *identifier* **{** *space_description* **}** （software部分）
space *identifier* [**,** *identifier*]**...{** *space_description* **}** （cpu部分またはmemory部分）
space = *identifier*

説明：

キーワード**space**は、cpu部分、memory部分、software部分で使用することができます。cpu部分やmemory部分では、**space**を使用して物理メモリアドレス空間を記述することができます。cpu部分やmemory部分のspace記述で使用できるキーワードは、**mau**と**map**のみです。software部分では、**space**を使用して1つまたは複数のメモリブロックを記述することができます。それぞれのspaceには、cpu部分やmemory部分にあるキーワード**space**ですでに定義されているシンボル名を付けます。

例：

cpu部分から

```
cpu {
cpu {
  amode far_data {
        attribute Y4;
        mau 8;
        map src=0 size=32k dst=32k space=address_space;
  }
        ...
  space address_space {
        // Specify space 'address_space' for the
        // address_bus address bus.
        mau 8;
        map src=0   size=32k dst=0   bus=address_bus label=rom;
        map src=32k size=32k dst=32k bus=address_bus label=ram;
  }
  .
}
```

software部分から

```
layout {
        // define the preferred locating order of sections
        // in the memory space
        // (the range is defined in the .cpu file)
        space address_space {
        ...
                // define for each sub-area in the space
                // the locating order of sections
                block rom {
                  // Memory block starting at chip rom_chip

                  // define a cluster for read-only sections
                  cluster code_clstr {
                        ....
                  }
                }
        .
        }
}
```

src

構文：

src = *address*　　（cpu部分またはmemory部分）

説明：

amode、space、bus記述にあるキーワード**map**の一部として、ソースアドレスを指定します。*address*については、10進数、16進数、8進数を使用することができます。またキロ(2^{10})やメガ(2^{20})を表すとき、(標準)DELFEE接尾辞の**k**や**M**をそれぞれ使用することができます。アドレスは、アドレッシングモードのローカルMAU(最小アドレス可能単位)サイズ(デフォルトでは8ビット)で指定します。

例：

```
cpu {
  ...
  amode near_code {
         attribute Y1;
         mau 8; // 8-bit addressable
         map src=0 size=1k dst=0 amode=far_code;
  }
}
```

stack

構文：

stack *stack_description*;　　（software部分）

stack;

説明：

stackは、section記述の特殊な形式です。スタックはロケート時に割り当てられます。ロケータは、必要な場合のみスタックを割り当てます。キーワード**stack**でロケートされたスタック空間には、2つの特殊なロケータラベルが対応します。スタック領域の最初はロケータラベル__lc_bsで取得され、スタック領域の最後はロケータラベル__lc_esでアクセスできます。

スタックが下方向に拡大してきた場合、スタックの開始がもっとも高いアドレスになる必要があります。そのためには、長さを正の値に保ち、*end_of_stack*をポイントするようにスタックポインタを設定することができます。その場合、

end_of_stack = *begin_of_stack* + length

上記の式が常に真になります。

stack記述では、キーワード**attribute**と**length**のみを使用することができます。記述を付けずに**stack**を指定した場合、ロケータはスタックをできる限り大きくしようとします。キーワード**stack**をまったく指定しない場合、ロケータは、スタックをできる限り大きくしようとしますが、その場合最低値は100(MAU)になります。

例：

```
space address_space {
        block ram {
                cluster data_clstr {
                        amode far_data {
                                section selection=w;
                                stack leng=150;
                                // stack of 150 MAUs
                                ...
                        }
                }
        }
}
```

start

構文：

start = *label*; （software部分）

説明：

プロセスの開始ラベルを定義します。

startは、ロードモジュール記述の中でのみ使用することができます。

例：

```
software {
        load_mod start = system_start;

        layout {
        .
        .
        }
}
```

table

構文：

table attr = *attribute*; （software部分）
table;

説明：

stackや**heap**と同じように、**table**も特殊なセクションです。通常のセクションは、コンパイル時に生成されて、アセンブラおよびリンカ経由でロケータに渡されます。stackセクションとheapセクションは、ユーザが要求したサイズでロケート時に生成されます。

tableの場合は異なります。ロケータはコピーテーブルを生成することができます。通常、このテーブルは、読み取り専用メモリに置かれます。テーブルロケーションを操作したい場合、**table**キーワードを使用することができます。**table**では、**attribute**のみを使用することができます。長さはロケート時に計算されます。**table**は、cluster記述でのみ指定することができます。

例：

```
space address_space {
  block rom {
        ...
        cluster code_clstr {
                attribute r; // cluster attribute
                amode far_code {
                  table; // locate copy table here
                  section selection=x;
                  section selection=r;
                  copy; // all ROM copies are located here
                }
        }
}
```

5.6.1 DELFEEキーワードの省略語

次のDELFEEキーワードは、固有の4文字の単語に省略することができます。

表5.6.1.1 DELFEEキーワードの省略語

キーワード	省略語
address	addr
assert	asse
attribute	attr
length	leng

5.6.2 DELFEEキーワードの要約

表5.6.2.1 DELFEEキーワードの要約

キーワード	説　明
address	絶対アドレスを指定します。
amode	アドレッシングモードを指定します。
assert	表明が失敗した場合エラーを出します。
attribute	属性をクラスタ、セクション、スタック、ヒープに割り当てます。
block	物理メモリ領域を定義します。
bus	アドレスバスを定義します。
chips	CPUチップを定義します。
cluster	クラスタの順序と配置を指定します。
copy	データセクションのROMコピーの配置を定義します。
cpu	cpu部分を定義します。
dst	デスティネーションアドレスを指定します。
fixed	メモリマップの固定ポイントを定義します。
gap	動的なメモリギャップを予約します。
heap	ヒープを定義します。
label	仮想アドレスラベルを定義します。
layout	layout記述の最初です。
length	スタック、ヒープ、物理ブロック、予約空間の長さを指定します。
load_mod	ロードモジュールを定義します(プロセス)。
map	ソースアドレスをデスティネーションアドレスにマッピングします。
mau	最小アドレス可能単位を(ビット単位で)定義します。
mem	チップの物理開始アドレスを定義します。
memory	memory部分を定義します。
regsfr	デバッガが使用するレジスタファイルを指定します。
reserved	メモリを予約します。
section	セクションがロケートされる方法を定義します。
selection	セクションをクラスタにグループ化するときの属性を指定します。
size	アドレス空間またはメモリのサイズを指定します。
software	software部分を定義します。
space	アドレス空間を定義するか、メモリブロックを指定します。
src	ソースアドレスを指定します。
stack	stackセクションを定義します。
start	他の開始ラベルを定義します。
table	tableセクションを定義します。

6 ユーティリティ

6.1 概要

S1C88プロセッサファミリ用のクロスアセンブラには、次のユーティリティが付属します。これらのユーティリティは、プログラム開発のさまざまな段階で利用できます。

ar88 IEEEアーカイバ。これはライブラリ機能で、オブジェクトライブラリを作成するとき、および保守するときに使用することができます。

cc88 S1C88ツールチェーンのコントロールプログラム。

mk88 プログラムのグループを保守、更新、再構築するためのユーティリティプログラム。

pr88 S1C88ツールチェーンのツールで作成したファイルの内容を表示するためのIEEEオブジェクトリーダー。

以降のページでは、それぞれのユーティリティについて説明します。

6.2 ar88

名前

ar88　IEEEアーカイバ兼ライブラリメンテナ。

構文

ar88　*key_option* [*option*]... *library* [*object_file*]...
ar88　**-V**
ar88　**-?**

説明

ar88を使用すると、別々のオブジェクトモジュールを1つのライブラリファイルに結合することができます。リンカは、特定のモジュールが、すでに読み込まれているモジュールの外部シンボル定義を解決するとき、オプションとしてライブラリのモジュールをインクルードします。ライブラリメンテナ**ar88**は、ライブラリファイルを構築するためのプログラムで、これを使用することで、既存のライブラリ内のモジュールの置換、抽出、削除が可能になります。

key_option　主要なオプションの1つで、**ar88**が実行するアクションを示します。このオプションは、任意の順序で任意の場所に記述することができます。

option　次のページで説明するようなサブオプション。これはオプションです。

library　ライブラリファイルです。

object_file　ライブラリとの間で追加、抽出、置換、削除するオブジェクトモジュールです。

オプション

オプションを指定するとき、最初に"-"を付けることができますが、付けなくてもかまいません。オプションは任意の順序で記述できます。また、オプションは組み合わせることができます。そのため**-xv**のように使用することもできます。ただし**-V**と**-?**については、コマンド行で独立したオプションとして使用する必要があります。

キーオプション：

-d　指定されたオブジェクトモジュールをライブラリから削除します。

-m　指定されたオブジェクトモジュールをライブラリの最後、または配置オプションで指定された位置に移動します。

-p　ライブラリ内にある指定オブジェクトモジュールを標準出力にプリントします。

注：　オブジェクトはバイナリ形式になっています。このオプションは通常、次のようにリダイレクトと併用します。

```
ar88 -p lib.a object.obj > t.obj
```

-r　ライブラリ内に、指定されたオブジェクトモジュールがある場合、それを置き換えます。そのオブジェクトモジュールがライブラリにない場合は、追加します。名前が指定されていない場合は、カレントディレクトリに同じ名前のファイルがあるオブジェクトモジュールのみが置換されます。新しいモジュールは最後に追加されます。

-t　ライブラリの一覧をプリントします。名前が指定されていない場合、ライブラリ内のすべてのオブジェクトモジュールがプリントされます。名前が指定された場合は、そのオブジェクトモジュールだけがリストされます。

-x　指定されたオブジェクトモジュールをライブラリから抽出します。名前が指定されていない場合、すべてのモジュールがライブラリから抽出されます。どちらの場合もライブラリ自体は変更されません。

その他のオプション：

-?　`stdout`でオプションについての説明を表示します。

-V　`stderr`でバージョン情報を表示します。

-a *posname*　新しいオブジェクトモジュールを既存のモジュール*posname*の後に追加または移動します。このオプションは、**m**オプションまたは**r**オプションと組み合わせた場合のみ使用できます。

-b *posname*　新しいオブジェクトモジュールを既存のモジュール*posname*の前に追加または移動します。このオプションは、**m**オプションまたは**r**オプションと組み合わせた場合のみ使用できます。

-c　ライブラリが存在しない場合、ユーザに通知しないまま、ライブラリファイルを作成します。

-f *file*　オプションを*file*から読み込みます。"-"の場合`stdin`を示します。`stdin`をクローズするときはEOFコードを指定する必要があります(通常Ctrl-Z)。

-o　最新修正日を、ライブラリに記録された日付にリセットします。このオプションは、**x**オプションと組み合わせた場合のみ使用できます。

-s　シンボルのリストをプリントします。このオプションは**-t**オプションと組み合わせて使用します。

-s1　シンボルのリストをプリントします。それぞれのシンボルの前に、ライブラリ名とオブジェクトファイルの名前が入れられます。このオプションは**-t**オプションと組み合わせて使用します。

-u　ライブラリファイルより新しい最新修正日を持つオブジェクトモジュールのみを置換します。**r**オプションと組み合わせた場合のみ使用できます。

-v　冗長オプション。冗長オプションを指定した場合、古いライブラリおよびその構成要素のモジュールから新しいライブラリファイルが作成されるとき、**ar88**はモジュール単位でそれについて報告します。このオプションは、**d**、**m**、**r**、**x**の各オプションと組み合わせた場合のみ使用できます。

-w*n*　警告レベルを*n*に設定します。

例

1. モジュール**startup.obj**および**calc.obj**で構成されるライブラリ**clib.a**を作成します。

```
ar88 cr clib.a startup.obj calc.obj
```

2. ライブラリ**clib.a**からすべてのモジュールを抽出します。

```
ar88 x clib.a
```

3. ライブラリ**clib.a**のシンボルのリストをプリントします。

```
ar88 ts clib.a
```

```
startup.obj
        symbols:
                _start
                _copytable
calc.obj
        symbols:
                _entry
```

4. ライブラリ**clib.a**のシンボルのリストを、異なる書式でプリントします。

```
ar88 ts1 clib.a
```

```
clib.a:startup.obj:_start
clib.a:startup.obj:_copytable
clib.a:calc.obj:_entry
```

5. ライブラリ**clib.a**からモジュール**calc.obj**を削除します。

```
ar88 d clib.a calc.obj
```

6.3 cc88

名前

cc88　S1Cツールチェーンのコントロールプログラム。

構文

cc88　[[*option*]... [*control*] ... [*file*]...]...
cc88　**-V**
cc88　**-?**

説明

コントロールプログラム**cc88**は、1行のコマンド行から、S1Cツールチェーンのさまざまなコンポーネントを起動する機能があります。コントロールプログラムは、コマンド行に記述されたソースファイルとオプションを任意の順序で受け付けます。

オプションの前には、"-"(マイナス記号)を付けます。入力ファイル*file*には、以下で説明する拡張子を付けることができます。

コントロールプログラムは、次の引数タイプを認識します。

- "-"文字で始まる引数はオプションです。オプションによっては、コントロールプログラム自身によって解釈されるものもありますが、多くのオプションは、そのオプションを受け付けるツールチェーンのプログラムに渡されます。
- .c拡張子が付いた引数は、Cソースプログラムと見なされ、コンパイラに渡されます。
- .asm拡張子が付いた引数は、アセンブラソースファイルと見なされ、前処理されてアセンブラに渡されます。
- .src拡張子が付いた引数は、コンパイルされたアセンブラソースファイルと見なされ、直接アセンブラに渡されます。
- .a拡張子が付いた引数は、ライブラリファイルと見なされ、リンカに渡されます。
- .obj拡張子が付いた引数は、オブジェクトファイルと見なされ、リンカに渡されます。
- .out拡張子が付いた引数は、リンクされるオブジェクトファイルと見なされ、ロケータに渡されます。ロケータは、起動時に1つの.outファイルだけを受け付けます。
- .dsc拡張子が付いた引数は、ロケータコマンドファイルとして処理されます。コマンド行に拡張子.dscを持つファイルがある場合、コントロールプログラムは、ロケートフェーズを追加する必要があると見なします。.dsc拡張子を持つファイルがない場合、コントロールプログラムは、リンクが終わったら停止します(ただし前のフェーズで停止するよう指示されていない場合)。
- 他のファイルが見つかった場合、エラーメッセージが表示されます。

通常、コントロールプログラムは、リンクロケートフェーズで絶対出力ファイルを生成した後、すべてのソースファイルをコンパイルおよびアセンブルしてオブジェクトファイルを生成します。ただし、アセンブラ、リンカ、ロケータのそれぞれの段階を抑制するためのオプションもあります。コントロールプログラムは、コンパイルプロセスの中間段階として固有のファイル名を生成します(後で削除される)。コンパイラとアセンブラが連続して呼び出される場合、コントロールプログラムは、コンパイラの生成したアセンブラファイルが前処理されないようにします。通常アセンブラ入力ファイルが最初に前処理されます。

オプション

-?　stdoutでオプションについての簡単な説明を表示します。

-M{s|c|d|l}　使用するメモリモデルを指定します。

スモール	(**s**)
コンパクトデータ	(**d**)
コンパクトコード	(**c**)
ラージ	(**l**)

-V コントロールプログラムが終了する直前に、バージョン番号を含む著作権ヘッダが表示されます。

-Ta*arg*
-Tc*arg*
-Tlk*arg*
-Tlc*arg*
これらのオプションを使用すると、コマンド行引数を直接アセンブラ(**-Ta**)、Cコンパイラ(**-Tc**)、リンカ(**-Tlk**)、ロケータ(**-Tlc**)に渡すことができます。これらのオプションは、コントロールプログラムが認識できないオプションを、適切なプログラムに渡すときに使用することができます。引数は、これらのオプションに直接追加するか、コントロールプログラムの独立した引数としてオプションの後に続けます。

-al アプリケーションのそれぞれのモジュールで絶対リストファイルを生成します。

-c
-cl
-cs
通常、コントロールプログラムは、すべての段階を起動して、指定された入力ファイルから絶対ファイルを構築します。これらのオプションを付けると、Cコンパイラ、アセンブラ、リンカ、ロケータのそれぞれの段階をスキップすることができます。**-cs**オプションを付けると、コントロールプログラムは、Cソースファイル(`.c`)のコンパイルの後、およびアセンブラソースファイル(`.asm`)の前処理の後停止して、生成される`.src`ファイルをそのまま保持します。**-c**オプションを付けると、コントロールプログラムは、アセンブラの後停止し、1つまたは複数のオブジェクトファイル(`.obj`)を出力します。**-cl**オプションを付けると、コントロールプログラムは、リンク段階の後停止し、リンカオブジェクトファイル(`.out`)を出力します。

-f*file* コマンド行引数を*file*から読み込みます。標準入力を示すときは、ファイル名に"-"を使用することができます。コマンド行のサイズの制限を回避するために、コマンドファイルを使用することができます。これらのコマンドファイルには、実際のコマンド行で使用できないオプションを入れます。コマンドファイルは、makeユーティリティなどを使用して簡単に作成できます。

コマンドファイルには、次のような簡単な規則があります。

1. コマンドファイルの同じ行に複数の引数を指定することができます。
2. 引数にスペースを入れるときは、その引数を一重引用符または二重引用符で囲みます。
3. 引用符の付いた引数の中で一重引用符または二重引用符を使用する場合、次の規則に従います。
   a. 中に入っている引用符が、一重引用符のみまたは二重引用符のみの場合、引数を囲むときもう一方の引用符を使用します。つまり、引数に二重引用符が含まれる場合、引数を一重引用符で囲みます。
   b. 両方の引用符が使用されている場合、それぞれの引用符がもう一方の引用符で囲まれるような形で、引数を分割する必要があります。

   例：
   ```
   "This has a single quote ' embedded"
   ```
   または
   ```
   'This has a double quote " embedded'
   ```
   または
   ```
   'This has a double quote " and a single quote '"' embedded"
   ```
4. オペレーティングシステムによっては、テキストファイル内の行の長さに制限がある場合があります。この制限を回避するため、継続行を使用することができます。これらの行は、最後にバックスラッシュと改行が付きます。引用符の付いた引数の場合、継続行は、次の行のスペースを取らないでそのままつなげられます。引用符が付いていない引数の場合、次の行にあるすべてのスペースは削除されます。

   例：
   ```
   "This is a continuation ¥
   line"
           → "This is a continuation line"

   control(file1(mode,type),¥
           file2(type))
           → control(file1(mode,type),file2(type))
   ```
5. コマンド行ファイルは、最高で25レベルまでネストすることができます。

-ieee
-srec
これらのオプションを使用すると、絶対ファイルのロケータ出力フォーマットを指定することができます。出力ファイルは、IEEE-695ファイル(.abs)、Motorola Sレコードファイル(.sre)のいずれかにすることができます。デフォルト出力は、IEEE-695ファイル(.abs)になります。

-nolib
このオプションを使用すると、コントロールプログラムが標準ライブラリをリンカに提供しなくなります。通常の場合、コントロールプログラムは、リンカにデフォルトCライブラリとランタイムライブラリを提供します。必要なライブラリは、コンパイラのオプションによって決まります。

-o *file*
通常、このオプションは、出力ファイル名を指定するためにロケータに渡されます。**-cl**オプションを使用してロケートフェーズを抑制する場合、**-o**オプションがリンカに渡されます。**-c**オプションを使用してリンクフェーズを抑制すると、ソースファイルが1つだけ指定されている場合に限り、**-o**オプションがアセンブラに渡されます。**-cs**オプションを使用してアセンブラフェーズを抑制すると、**-o**オプションがコンパイラに渡されます。引数は、これらのオプションに直接追加するか、コントロールプログラムの独立した引数としてオプションの後に続けます。

-tmp
このオプションを使用すると、コントロールプログラムがカレントディレクトリに中間ファイルを作成するようになります。これらのファイルは自動的に削除されません。通常、コントロールプログラムは、中間変換結果として一時ファイルを生成します。これには、コンパイラが生成するアセンブラファイル、オブジェクト・ファイル、リンカ出力ファイルなどのようなものがあります。変換プロセスの次のフェーズが無事に完了したら、これらの中間ファイルは削除されます。

-v
-vオプションを使用すると、それぞれのプログラムの起動状況が標準出力に表示されます。このとき、"+"が行の最初に付けられます。

-v0
このオプションは、**-v**オプションと同じ働きをしますが、起動状況を表示してもプログラムは実際に起動されないという点で異なります。

cc88が使用する環境変数

コントロールプログラムは、次の環境変数を使用します。

TMPDIR
この変数は、コントロールプログラムが一時ファイルの作成時に使用するディレクトリを指定するときに使用できます。この環境変数が設定されていない場合、一時ファイルはカレントディレクトリに作成されます。

CC88OPT
コントロールプログラムの起動時に余分のオプションや引数を渡すとき、この環境変数を使用できます。コントロールプログラムは、コマンド行引数の前にこの変数で指定された引数を処理します。

CC88BIN
この変数が設定されているとき、コントロールプログラムは、起動するツールの名前の前に、この変数で指定されているディレクトリを追加します。

6.4 mk88

名前

mk88　プログラムのグループの保守、更新、再構築を行います。

構文

mk88　[*option*]... [*target*] ... [*macro*=*value*]...
mk88　**-V**
mk88　**-?**

説明

mk88は、依存関係のファイル("makefile")を調べ、ファイルを最新の状態にするためにどのようなコマンドを実行したら良いか判断します。これらのコマンドは、**mk88**から直接実行することができる他、実行しないで標準出力に書き出すこともできます。

コマンド行でターゲット(target)が指定されていない場合、**mk88**は、最初のmakeファイルで定義されている最初のターゲットを使用します。

オプション

-?　起動構文を表示します。

-D　makeファイルのテキストを読み込みながら表示します。

-DD　makeファイルおよび"mk88.mk"のテキストを表示します。

-G *dirname*　makeファイルを読み込む前に、*dirname*で指定されたディレクトリに変更します。これにより、現在の作業ディレクトリと別のディレクトリに、アプリケーションを構築できるようになります。

-S　**-k**オプションの効果を無効にします。コマンドが0以外の終了ステータスを返した場合、処理が停止します。

-V　`stderr`でバージョン情報を表示します。

-W *target*　この*target*の修正時が"現在"になっているものとして実行します。これは、"状況を想定した場合"のオプションです。

-d　**mk88**がターゲットを再構築する理由を表示します。比較的新しいすべての依存関係が表示されます。

-dd　依存関係チェックを詳細に表示します。比較的古い依存関係も、新しいものと一緒に表示されます。

-e　環境変数がmakeファイルのマクロ定義を上書きするよう指定します。通常の場合、makeファイルのマクロが環境変数を上書きします。コマンド行のマクロ定義は常に、環境変数とmakeファイルマクロ定義の両方を上書きます。

-f *file*　"makefile"の代わりに、指定されたファイルを使用します。makeファイルの引数として使用される"-"は、標準入力を示します。

-i　コマンドが返すエラーコードを無視します。これは、特殊ターゲット.IGNORE:と同等です。

-k　コマンドが0以外の終了ステータスを返した場合、現在のターゲットで実行している作業は破棄しますが、このターゲットに依存しない他の分岐については継続します。

-n　乾実行を行います。つまりコマンドをプリントしますが、実際に実行することはありません。行の最初に@記号が付いている場合でもプリントします。ただしコマンド行で指定されたコマンドが、**mk88**を起動するものであれば、その行は常に実行されます。

-q　クエスチョンモード。**mk88**は、ターゲットファイルが最新であるかどうかによって、0または0以外のステータスコードを返します。

-r　デフォルトファイル"mk88.mk"を読み込みません。

-s サイレントモード。コマンドを実行する前に、コマンド行をプリントしません。これは、特殊ターゲット.SILENT:と同等です。

-t ターゲットファイルを再構築するという規則を実行せずに、ターゲットファイルを変更して最新の状態にします。

-w 警告とエラーを標準出力にリダイレクトします。このオプションを指定しない場合、**mk88**、および**mk88**が実行するコマンドは、標準エラーを出力します。

macro=value マクロ定義。この定義は、**mk88**の起動時に固定されたままになります。指定されたマクロに対応するmakeファイル内のマクロ、および環境変数のマクロの通常の定義が、これによって上書きされます。この定義は、下位の**mk88**にも継承されますが、その場合は、環境変数として動作します。つまり、これは、**-e**設定によっては、makeファイルの定義で上書きされることもあるということになります。

使用法

makeファイル

最初に読み込まれるmakeファイルは"mk88.mk"で、このファイルは次のロケーションで(この順序で)検索されます。

- 現在の作業ディレクトリ
- HOME環境変数によってポイントされるディレクトリ
- **mk88**が存在するディレクトリの相対ディレクトリ`etc`

例：

mk88が`¥C88¥BIN`にインストールされている場合、ディレクトリ`¥C88¥ETC`でmakeファイルが検索されます。

通常このディレクトリには、定義済みのマクロと暗黙的な規則が含まれます。

makeファイルのデフォルト名は、カレントディレクトリの"makefile"です。他のmakeファイルを指定する場合は、コマンド行で**-f**オプションを使用します(複数指定可)。**-f**オプションを複数指定した場合、すべてのmakeファイルが左から右の順に連結されたかのように扱われます。

makeファイルでは、コメント行、マクロ定義、インクルード行、ターゲット行が混在しています。行は、改行を"\"(または"\")でエスケープすることにより、入力行をまたいで続けることができます。行の最後に"\"(または"\")が付いている場合、空のマクロが追加されます。"#"の後の文字はすべてコメントと見なされ、"#"の直前にあるスペースと一緒に行から除去されます。"#"が引用符付きの文字列の中にある場合は、コメントとして扱われません。また、完全なブランク行は無視されます。

*include*行は、他のmakeファイルのテキストをインクルードするために使用します。この行では、"include"という単語の後に、スペースが続き、さらにこの行にインクルードするファイルの名前が続けられます。インクルードされるファイル名にあるマクロは、ファイルがインクルードされる前に展開されます。インクルードファイルは、ネストすることもできます。

*export*行は、**mk88**が実行するコマンドの環境にマクロ定義をエクスポートするために使用します。このような行では、"export"という単語の後に、スペースが続き、さらにエクスポートするファイルの名前が続けられます。マクロは、export行が読み込まれるときにエクスポートされます。つまり、順方向のマクロ定義に対する参照は、定義されていないマクロと同等になります。

条件処理

`ifdef`、`ifndef`、`else`、`endif`が含まれる行は、makeファイルの条件処理で使用されます。これらは、次のようにして使用します。

```
ifdef macroname
if-lines
else
else-lines
endif
```

*if-lines*と*else-lines*には、任意の数の行と任意のテキストを入れることができます。`ifdef`、`ifndef`、`else`、`endif`の行も入れることができる他、1行も入れないことも可能です。

また、elseの行は、その後の*else-lines*と一緒に省略することができます。

最初に、ifコマンドの後にある*macroname*が定義されているかどうかチェックされます。マクロが定義されていれば、*if-lines*が解釈され、*else-lines*が(存在する場合)破棄されます。定義されていない場合、*if-lines*が破棄されます。その場合、*else*行があれば*else-lines*が解釈されますが、*else*行がなければ行は解釈されません。

ifdefの代わりにifndef行を使用すると、マクロが定義されていないことがチェックされます。これらの条件行は、最高で6レベルまでネストすることができます。

マクロ

マクロは"WORD = text and more text"の書式になります。WORDは大文字にする必要はありませんが、一般的には大文字にすることになっています。等号の左右のスペースは重要ではありません。この後の行に$(WORD)や${WORD}が出てくると、"text and more text"で置換されます。マクロ名が単一の文字の場合、このかっこはオプションになります。展開は再帰的に実行されるため、マクロの本体にマクロの起動コマンドを入れることもできます。マクロ定義の右辺は、定義のときではなく、マクロが実際に使用されるときに展開されます。

例：

```
FOOD = $(EAT) and $(DRINK)
EAT = meat and/or vegetables
DRINK = water
export FOOD
```

$(FOOD)は、"meat and/or vegetables and water"になり、環境変数FOODは、それに従ってexport行で設定されます。ただしマクロ定義に、定義されているマクロへの直接的な参照がある場合、これらのマクロ定義は、定義の段階で展開されます。これは、マクロ定義の右辺が(部分的ではあるが)展開される唯一の事例になります。例を示します。

```
DRINK = $(DRINK) or beer
```

export行の後に上記の行があれば、次の行とまったく同じ扱いで$(FOOD)に影響します。

```
DRINK = water or beer
```

ただし環境変数FOODは、再度エクスポートされない限り更新されません。

特殊マクロ

MAKE

このマクロは通常、値**mk88**になります。MAKEを一時的に起動する行は、1行の期間だけ、**-n**オプションを上書きします。そのため、MAKEをネストして起動する場合に**-n**オプションを付けてテストすることができます。

MAKEFLAGS

このマクロは、**mk88**に対して指定されたオプションのセットを、その値として持ちます。これが環境変数として設定されている場合、そのオプションのセットが、コマンド行オプションの前に処理されます。このマクロは、ネストされた**mk88**に明示的に渡すことができますが、これらの起動構文で環境変数として使用することもできます。**-f**フラグと**-d**フラグは、このマクロには記録されません。

PRODDIR

このマクロは、**mk88**がインストールされたディレクトリの名前について、最後のパス構成要素を付けずに展開します。生成されるディレクトリ名は、インストールされたS1Cパッケージのルートディレクトリになります(**mk88**を別のディレクトリにインストールしている場合はこの限りではない)。このマクロは、製品に付属するファイル(たとえばライブラリソースファイルなど)を参照するときに使用することができます。

例：

```
DOPRINT = $(PRODDIR)¥lib¥src¥_doprint.c
```

mk88が¥c88¥binディレクトリにインストールされている場合、この行は次のように展開されます。

```
DOPRINT = ¥c88¥lib¥src¥_doprint.c
```

SHELLCMD

このマクロには、SHELLに対してローカルなコマンドのデフォルトリストが含まれます。これらのコマンドのいずれかを起動するような規則になっている場合、これを処理するためにSHELLが自動的に起動します。

TMP_CCPROG

このマクロには、コントロールプログラムの名前が含まれます。このマクロとTMP_CCOPTマクロが設定されていて、コントロールプログラムのコマンド行引数リストが127文字を越えている場合、**mk88**は、これらのコマンド行引数を書き込んだ一時ファイルを作成します。**mk88**は、コマンド入力ファイルとしてこの一時ファイルを指定した状態でコントロールプログラムを呼び出します。

TMP_CCOPT

このマクロにはコントロールプログラムのオプションが含まれ、ファイルをコマンド引数として読み込むようコントロールプログラムに通知します。

例：

```
TMP_CCPROG = cc88
TMP_CCOPT  = -f
```

$ このマクロは、ドル記号に変換されます。そのためmakeファイルで単一の"$"を表現するときは"$$"を使用することができます。

また、動的に保持されるマクロもいくつかあり、規則内で省略語として使用することができます。これらのマクロは、明示的に定義しないで使用します。

$* カレントターゲットのベース名。

$< 現在の依存関係ファイルの名前。

$@ カレントターゲットの名前。

$? ターゲットより新しい依存関係の名前。

$! 依存関係の名前。

$<マクロと$*マクロは、通常暗黙的な規則で使用されます。これらのマクロを明示的なターゲットコマンド行で使用すると、信頼できない結果になることがあります。すべてのマクロでは、F接尾辞を付けてファイル名構成要素を指定することができます(たとえば${*F}、${@F})。同様にマクロ$*、$<、$@では、D接尾辞を付けてディレクトリ構成要素を指定することができます。

関数

関数は、展開するだけでなく、特定の動作も実行します。関数は構文上マクロと同様ですが、"$(match arg1 arg2 arg3)"のようにマクロ名にスペースが埋め込まれています。すべての関数は、組み込み関数で、現在、`match`、`separate`、`protect`、`exist`、`nexist`の5つがあります。

`match`関数は、特定の接尾辞と一致するすべての引数を生成します。

```
$(match .obj prog.obj sub.obj mylib.a)
```

たとえば上記の関数により、次の文が生成されます。

```
prog.obj sub.obj
```

`separate`関数は、最初の引数を区切り文字として使用し、引数を連結します。最初の引数が二重引用符で囲まれている場合、"\n"は改行文字、"\t"はタブ、"*ooo*"は8進値(*ooo*は3桁までの8進数)にそれぞれ変換され、スペースはそのまま使用されます。例を示します。

```
$(separate "¥n" prog.obj sub.obj)
```

この関数は、次のような結果を生成します。

```
prog.obj
sub.obj
```

関数の引数には、マクロまたは関数自体を入れることができます。そのため、

```
$(separate "¥n" $(match .obj $!))
```

は、カレントターゲットが依存するすべてのオブジェクトファイルを改行文字列で区切って生成します。

protect関数は、1レベルの引用を追加します。この関数は引数を1つとり、引数にはスペースを入れることができます。引数にスペース、一重引用符、二重引用符、円記号(またはバックスラッシュ)が含まれている場合は、その引数を二重引用符で囲みます。また二重引用符や円記号(またはバックスラッシュ)は、円記号(またはバックスラッシュ)でエスケープします。

例：

```
echo $(protect I'll show you the "protect" function)
```

この関数は、次の文を生成します。

```
echo "I'll show you the ¥"protect¥" function"
```

exist関数は、最初の引数が既存のファイルまたはディレクトリの場合、2番目の引数を展開します。

例：

```
$(exist test.c cc88 test.c)
```

ファイルtest.cが存在する場合、次の文が生成されます。

```
cc88 test.c
```

ファイルtest.cが存在しない場合、何も展開されません。

nexist関数は、exist関数の反対です。最初の引数が既存のファイルまたはディレクトリでない場合、2番目の引数を展開します。

例：

```
$(nexist test.src cc88 test.c)
```

ターゲット

makeファイルのターゲットエントリは、次の書式になります。

```
target ... : [dependency ...] [; rule]
       [rule]
        ...
```

スペースが最初に付いていない行は、(マクロ定義を除き)すべて"ターゲット"行になります。ターゲット行は、ターゲットと呼ばれる1つまたは複数のファイル名(または同じものに展開されるマクロ)にコロン(:)を付けて指定します。":"の後には、依存ファイルのリストを続けます。依存関係のリストは、最後にセミコロン(;)を付け、その後に規則またはシェルコマンドを続けます。

MS-DOSでは、他のドライブを指定するときにコロンを指定する必要があるため、そのような場合にコロンを使用できるようになっています。そのため、たとえば次の文は、意図した通りに動作するようになっています。

```
c:foo.obj : a:foo.c
```

ターゲットが複数のターゲット行で指定されている場合、それぞれの依存関係が追加され、ターゲットの完全な依存関係リストが形成されます。

依存関係は、ターゲットが構築されるときの元になるものです。ある依存関係は、他の依存関係のターゲットになります。一般的に特定のターゲットファイルに対して、依存関係ファイルが作成され、依存関係の点で最新の状態になるよう配慮されています。

ターゲットの変更時刻は、それぞれの依存関係ファイルの修正時刻と比較されます。ターゲットの方が古い場合、1つまたは複数の依存関係が変更されていることになるため、ターゲットを構築する必要があります。もちろん、このチェックは再帰的に実行されるようになっており、すべての依存関係の依存関係の依存関係の……を最新の状態に保ちます。

ターゲットを再構築するとき、**mk88**は、マクロと関数を展開し、最初のスペースを取り去った後で、規則を直接実行するか、実行のためそれぞれをシェルまたはCOMMAND.COMに渡します。

ターゲット行では、入力時にマクロと関数が展開されます。他のすべての行では、絶対的に必要になるまで拡張が行われません(つまり規則のマクロと関数が動的になっている)。

特殊なターゲット

.DEFAULT:

このターゲットの規則は、そのターゲットに対するエントリ、およびそれを構築するための暗黙的な規則が他にない場合に、ターゲットを処理するために使用されます。**mk88**は、このターゲットに対するすべての依存関係を無視します。

.DONE:

このターゲットおよびその依存関係は、他のすべてのターゲットが構築された後、処理されます。

.IGNORE:

0以外のエラーコードがコマンドから返されても無視します。makeファイルにこのターゲットがある場合、コマンド行で**-i**を指定した場合と同じ結果になります。

.INIT:

このターゲットおよびその依存関係は、他のすべてのターゲットが処理される前に処理されます。

.SILENT:

コマンドを実行する前に、そのコマンドを出力しません。makeファイルにこのターゲットがある場合、コマンド行で**-s**を指定した場合と同じ結果になります。

.SUFFIXES:

暗黙的な規則を選択するときの接尾辞リストです。依存関係を付けてこのターゲットを指定すると、その依存関係が接尾辞リストの最後に追加されます。依存関係を付けないでこのターゲットを指定すると、リストがクリアされます。

.PRECIOUS:

このターゲットで指定された依存関係ファイルは削除されません。通常**mk88**は、構築ルールのコマンドがエラーを返した場合、またはターゲットの構築が割り込まれた場合、ターゲットファイルを削除します。

規則

makeファイルで、最初にタブやスペースが付いている行は、シェル行または規則です。この行は、直前の依存関係行に対応しています。これらのシーケンスは、単一の依存関係行と対応させることができます。ターゲットが、依存関係の点で最新の状態でなくなった場合、コマンドのシーケンスが実行されます。シェル行では、コマンドの左側に次の文字の組み合わせを入れることができます。

@ **-n**が使用された場合を除き、コマンド行に出力しません。

- **mk88**は、コマンドの終了コード、つまりMS-DOSのERRORLEVELを無視します。これを指定しない場合、0以外の終了コードが返されたときに**mk88**が終了します。

+ **mk88**が、シェルまたはCOMMAND.COMを使用して、コマンドを実行します。

シェル行に"+"が付いていないにもかかわらず、コマンドがDOSコマンドである場合、またはリダイレクトが使用されている場合(<、|、>)、シェル行はCOMMAND.COMに渡されます。

mk88は、インラインの一時ファイルを生成することができます。行に"<<WORD"が含まれている場合、行頭にWORDが指定されている行までの、すべての行が一時ファイルに入れられます。

例：

```
lk88 -o $@ -f <<EOF
       $(separate "¥n" $(match .obj $))
       $(separate "¥n" $(match .a $))
       $(LKFLAGS)
EOF
```

タグ(EOF)の間の3行が一時ファイル(たとえば"\tmp\mk2")に書き込まれ、コマンド行が"lk88 -o $@ -f \tmp\mk2"のように書き換えられます。

暗黙的な規則

暗黙的な規則は、.SUFFIXES:特殊ターゲットと密接に関連しています。.SUFFIXES:リストのそれぞれのエントリは、他のファイルを構築するときに使用されるファイル名の拡張子を定義します。暗黙的な規則は、あるファイルを他のファイルから実際に構築する方法を定義します。これらのファイルは、共通のベース名を共有していますが、拡張子はそれぞれで異なっています。

作成されるファイルに明示的なターゲット行がない場合、暗黙的な規則が探されます。暗黙的なターゲットの名前を取得するために、.SUFFIXES:リストの各エントリが、ターゲットの拡張子と照合されます。このようなターゲットが存在する場合、その依存関係拡張子を持つファイルの変換に使用される規則が、そのターゲットファイルに適用されます。暗黙的なターゲットの依存関係は無視されます。

作成されるファイルに明示的なターゲット行があって規則がない場合、同様の方法で暗黙的な規則が探されます。暗黙的なターゲットの名前を取得するために、.SUFFIXES:リストの各エントリが、ターゲットの拡張子と照合されます。このようなターゲットが存在する場合、依存関係のリストで正しい拡張子を持つファイルが検索され、ターゲットを作成するために暗黙的な規則が起動されます。

例

このmakeファイルでは、`prog.out`が`prog.obj`と`sub.obj`の2つのファイルに依存していて、この2つのファイルがそれぞれの対応するソースファイル(`prog.c`と`sub.c`)および共通ファイル`inc.h`に依存していることが指定されています。

```
LIB=          -ls

prog.out:     prog.obj sub.obj
     lk88     prog.obj sub.obj $(LIB) -o prog.out

prog.obj:     prog.c inc.h
    c88       prog.c
    as88      prog.src

sub.obj:      sub.c inc.h
    c88       sub.c
    as88      sub.src
```

次のmakeファイルは、(`mk88.mk`の)暗黙的な規則を使用して同じジョブを実行します。

```
LDFLAGS       = -ls
prog.out:     prog.obj sub.obj
prog.obj:     prog.c inc.h
sub.obj:      sub.c inc.h
```

ファイル

makefile　依存性と規則の記述。

mk88.mk　デフォルトの依存性と規則。

診断

mk88は、結果がエラーで終わった場合、終了ステータス1を返します。それ以外の場合、ステータス0を返します。

6.5 pr88

名前

pr88　IEEEオブジェクトリーダー。
リロケータブルオブジェクトファイルまたは絶対ファイルの内容を表示します。

構文

pr88　[*option*]... *file*
pr88　**-V**
pr88　**-?**

説明

pr88は、S1Cツールチェーンのツールによって作成されたオブジェクトファイルを高レベルで表示します。ただし、**pr88**は逆アセンブラではありません。

オプション

オプションの最初には"-"記号を付けます。また、単一の"**-**"の後に複数のオプションを組み合わせることもできます。オブジェクトファイルの特定の部分をプリントするオプションもあります。たとえばオプション**-h**を使用すると、ヘッダ部分、環境部分、AD/拡張部分を全体的に表示することができます。これらの部分は小さいもので、それぞれの部分を別々に表示することもできます。部分を指定しない場合、デフォルトは**-hscegd0i0**になります(すべての部分、デバッグ部分、イメージ部分が一覧形式で表示される)。

他に、出力をコントロールするためのオプションもあります。

入力コントロールオプション

-f *file*　コマンド行情報を*file*から読み込みます。"**-**"の場合、情報は標準入力から読み込まれます。

コマンド行処理のために*file*を使用します。コマンド行のサイズ制限を回避するために、コマンドファイルを使用することができます。これらのコマンドファイルに入れるオプションは、実際のコマンド行では使用できないものです。コマンドファイルは、makeユーティリティなどを使用して簡単に作成できます。

-fオプションは複数使用することができます。

コマンドファイルには、次のような簡単な規則があります。

1. コマンドファイルの同じ行に複数の引数を指定することができます。
2. 引数にスペースを入れるときは、その引数を一重引用符または二重引用符で囲みます。
3. 引用符の付いた引数の中で一重引用符または二重引用符を使用する場合、次の規則に従います。
   a. 中に入っている引用符が、一重引用符のみまたは二重引用符のみの場合、引数を囲むときもう一方の引用符を使用します。つまり、引数に二重引用符が含まれる場合、引数を一重引用符で囲みます。
   b. 両方の引用符が使用されている場合、それぞれの引用符がもう一方の引用符で囲まれるような形で、引数を分割する必要があります。

   例：
   ```
   "This has a single quote ' embedded"
   ```
   または
   ```
   'This has a double quote " embedded'
   ```
   または
   ```
   'This has a double quote " and a single quote '"' embedded"
   ```
4. オペレーティングシステムによっては、テキストファイル内の行の長さに制限がある場合があります。この制限を回避するため、継続行を使用することができます。これらの行は、最後にバックスラッシュと改行が付きます。引用符の付いた引数の場合、継続行は、次の行のスペースを取らないでそのままつなげられます。引用符が付いていない引数の場合、次の行にあるすべてのスペースは削除されます。

例：

```
"This is a continuation ¥
line"
      → "This is a continuation line"

control(file1(mode,type),¥
      file2(type))
      → control(file1(mode,type),file2(type))
```

5. コマンド行ファイルは、最高で25レベルまでネストすることができます。

出力コントロールオプション

-Hまたは**-?** 起動構文の説明を`stdout`に表示します。

-V `stderr`でバージョン情報を表示します。

-W*n* 出力幅を*n*桁に設定します。デフォルトは128桁で、最小値は78桁です。

-l*n* レベルをコントロールします。6.5.3節を参照してください。

-o*file* 出力ファイル名を指定します。デフォルトは`stdout`です。

-v 選択した部分を冗長形式でプリントします。

-v*n* レベル*n*の冗長形式でプリントします。6.5.3節を参照してください。

-w*n* 警告レベルが*n*より上のメッセージを抑制します。

表示オプション

-c 呼び出しグラフをプリントします。

-d グローバルの型を除く、すべてのデバッグ情報をプリントします。

-d0 デバッグ部分を一覧形式でプリントします。

-d*n* ファイル番号*n*以降のデバッグ情報をプリントします。

-e 変数を、外部の有効範囲と一緒にプリントします。

-e1 変数を、外部の有効範囲と一緒にプリントし、オブジェクトファイルの名前が付いたシンボル名を前に付けます。

-g グローバルの型をプリントします。

-h 一般ファイル情報をプリントします。

-i すべてのセクションイメージをプリントします。

-i0 イメージ部分を一覧形式でプリントします。

-i*n* セクション*n*のイメージをプリントします。

-s セクション情報をプリントします。

6.5.1 デモファイルの準備

この節で、**pr88**の使用法を示すために使用するファイルは、次の3つです。

calc.obj
calc.out
calc.abs

デモを実行する場合、これらのcalcサンプルファイルを作業ディレクトリにコピーすることにより、これらのファイルを準備します。S1Cの各ツールは、検索パス経由で検索できるようにしておく必要があります。次のコマンドを実行してファイルを作成します。

```
cc88 -Ms -nolib startup.asm _copytbl.asm calc.asm -o calc.abs
     s1c88316.dsc -tmp
```

6.5.2 オブジェクトファイルの部分の表示

6.5.2.1 オプション-h：一般ファイル情報の表示

-hオプションを指定すると、ファイルの一般情報が表示されます。起動構文は次のようになります。

```
pr88 -h calc.out
```

次の情報が表示されます。

```
File name     = calc.out:
Format        = Relocatable
Produced by   = S1C object linker
Date          = jan 23, 1997 16:35:40h
```

この出力の意味は自明でしょう。**-h**スイッチは冗長オプションと組み合わせることもできます。

```
pr88 -hv calc.out
```

出力は、重要性の低い、より一般的な情報にまで拡大されます。

```
File name    = calc.out:
Format       = Relocatable
Produced by  = S1C object linker
Date         = jan 23, 1997 16:35:40h
Obj version  = 1.1
Processor    = S1Cs
Address size = 24 bits
Byte order   = Least significant byte at lowest address
Host         = Sun

Part                    File offset    Length
-------------------------------------------------
Header part             0x00000000     0x00000055
AD Extension part       0x00000055     0x00000033
Environment part        0x00000088     0x0000002b
Section part            0x000000b3     0x0000009b
External part           0x0000014e     0x00000098
Debug/type part         0x000001e6     0x000002b8
Data part               0x0000049e     0x000002b8
Module end              0x00000756
```

この表では、主要なオブジェクト部分のファイルオフセットと長さが表示されています。

6.5.2.2 オプション-s：セクション情報の表示

-sオプションを指定すると、オブジェクトモジュールからセクション情報を取得することができます。セクションの内容については、**-i**オプションを付けることで表示できます。6.5.2.7節を参照してください。

```
pr88 -s calc.out
```

```
Section           Size
-----------------------------
.startup_vector   0x000002
.startup          0x000063
.watchdog_vector  0x000002
.watchdog         0x000001
.text             0x00002d
.data             0x000003
.zdata            0x000001
```

ロケートされたファイルの場合、これ以上セクション情報を表示することはできません。一度ロケートすると、独立したセクションは新しいクラスタに結合されます。絶対ファイルについては、"**pr88 -s**"により、クラスタ情報を表示することができます。

```
pr88 -s calc.abs
```

```
Section   Size
----------------------
rom       0x0000b9
ram       0x00f800
```

ロケートマップを表示すると、どのセクションがどのクラスタにロケートされるかがわかります。もちろん、冗長オプションを使用して、すべてのセクション情報を表示することもできます。

```
pr88 -sv calc.out
```

```
Section          Size     Address  Align PageSize Mau Attributes
-------------------------------------------------------------------------------------------
.startup_vector  0x000002 0x000000 0x001 -        -   ReadOnly Execute ZeroPage Space 1 Abs Separate
.startup         0x000063 -        0x001 -        -   ReadOnly Execute ZeroPage Space 1 Cumulate
.watchdog_vector 0x000002 0x000004 0x001 -        -   ReadOnly Execute ZeroPage Space 1 Abs Separate
.watchdog        0x000001 -        0x001 -        -   ReadOnly Execute ZeroPage Space 1 Cumulate
.text            0x00002d -        0x001 -        -   ReadOnly Execute ZeroPage Space 1 Cumulate
.data            0x000003 -        0x001 -        -   Write Space 2 Initialized Cumulate
.zdata           0x000001 -        0x001 -        -   Write Space 2 Cleared Cumulate
```

最初の2つのカラムには、セクション名とセクションサイズが表示されます。"Address"のカラムにはセクションアドレスが表示されますが、セクションがリロケータブルな状態になっている場合は"-"が表示されます。S1Cの場合、セクションの揃え(Align)は常に1になります。ページサイズ(PageSize)は、短いセクションの場合のみ有効です。MAUは、アドレス空間の最小アドレス可能単位を示しています(ビット単位)。セクション属性(Attributes)には大きく2つのグループがあります。1つはロケータによって使用される割り当て属性で、もう1つはリンカによって使用されるオーバーラップ属性です。

割り当て属性

Write	RAMにロケートする必要があります。
ReadOnly	ROMにロケートすることができます。
Execute	ROMにロケートすることができます。
Space *num*	アドレッシングモード*num*でロケートする必要があります。
Abs	すでにアセンブラによってロケートされています。
Cleared	セクションを"0"で初期化する必要があります。
Initialized	セクションをRAMからROMにコピーする必要があります。
Scratch	セクションが充てんされておらずクリアもされていません。

オーバーラップ属性

MaxSize	検出されたサイズのうち最大のサイズが使用されます。
Unique	1つのセクションでのみこの名前を使用できます。
Cumulate	同じ名前を持つセクションを連結して、1つの大きなセクションを生成します。
Overlay	名前*name@func*を持つセクションを、呼び出しグラフから取得された*func*の規則に従ってセクション*name*に結合します。
Separate	セクションがリンクされません。

6.5.2.3 オプション-c：呼び出しグラフの表示

呼び出しグラフは、リンカの重ね書きアルゴリズムによって使用されます。ファイルがリンクされて重ね書きが実行されると、呼び出しグラフの情報はオブジェクトファイルから削除されます。calc.outの呼び出しグラフの情報を参照しようとすると、"No call graph found"というメッセージが出されます。

この時点では、ファイルcalc.objはまだリンクされていません。そのため、呼び出しグラフを参照したい場合、このファイルを使用することができます。

```
pr88 -c calc.obj
```

calcのサンプルファイルには、重ね書きする必要のあるセクションがないため、再度"No call graph found"というメッセージが出されます。次の例は、呼び出しグラフがどのように表示されるかを示す一例です。

```
Call graph(s)
=============

Call graph 0:

main()
  ->See call graph 1
  ->See call graph 4
  ->See call graph 2
  _exit()
  print_str()
  clear_screen()

Call graph 1:

queens?find_legal_row()
  ->See call graph 1
  ->See call graph 2
  abs()
  ->See call graph 3
```

それぞれの呼び出しグラフでは、関数の後に、関数や他のグラフのリストが続けられます。これらは最初の関数によって呼び出されます。この関数によって呼び出される関数と呼び出しグラフは、2つのスペース分インデントされます。ある関数が他の関数を呼び出す場合、その関数は、さらに2スペース分インデントされてリストされます。

例を見るとわかるように、1つの呼び出しグラフから他の呼び出しグラフに対する参照があります。しかも呼び出しグラフ1(Call graph 1)は、それ自身を呼び出しています。これは、関数find_legal_row()が再帰関数であることを示しています。冗長スイッチを使用すると、出力は次の図のように見やすくなります。

```
main()
  |
  +--->See call graph 1
  |
  +--->See call graph 4
  |
  +--->See call graph 2
  |
  +--exit()
  |
  +--print_str()
  |
  +--clear_screen()
```

呼び出しグラフ1の関数find_legal_rowは、静的関数です。名前の競合を回避するため、ソース名はこの関数名に追加されます。

呼び出しグラフの参照が解決された呼び出しグラフが必要な場合、次のようにリンカを使用することで生成することができます。

```
lk88 -o call.out -Mcr calc.obj
```

オプション**-M**は、リンカに.lnlファイルを生成するよう命令します。このファイルには、呼び出しグラフが冗長レイアウトで記述されています。またオプション**-c**の指定によって、リンカは.calファイルを生成します。このファイルにも、(同じ)呼び出しグラフが含まれますが、簡単な(冗長でない)レイアウトになっています。オプション**-r**は、これがインクリメンタルリンクであることをリンカに伝えます。

6.5.2.4 オプション-e：外部部分の表示

他のファイルの外部部分には、リンク時に使用されたすべてのシンボルが記述されています。これらのシンボルには、外部の有効範囲が設定されています。**pr88**で**-e**オプション(または**-e0**)を使用すると、次のように外部シンボルが表示されます。

```
pr88 -e calc.out
```

```
Variable     S  Address/Size
-----------------------------
__start_cpt  I  .startup + 0x00
__START      I  .startup + 0x00
__exit       I  .startup + 0x20
__copytable  I  .startup + 0x22
_main        I  .text + 0x20
__lc_es      X  -
__lc_cp      X  -
```

オプション**-e1**を使用すると、出力オブジェクトファイルの名前も表示されます。

```
pr88 -e1 calc.out
```

```
Variable              S  Address/Size
-------------------------------------
calc.out:__start_cpt  I  .startup + 0x00
calc.out:__START      I  .startup + 0x00
calc.out:__exit       I  .startup + 0x20
calc.out:__copytable  I  .startup + 0x22
calc.out:_main        I  .text + 0x20
calc.out:__lc_es      X  -
calc.out:__lc_cp      X  -
```

最初のカラムには、シンボルの名前が入ります。一般的にこのシンボルは、最初に"F"が付けられた高レベルシンボルです。次のカラムには、シンボルのステータスが入ります。定義済みのシンボルの場合"**I**"、参照されるがまだ定義されていないシンボルの場合"**X**"が入ります。最後のカラムには、シンボルのアドレスが入ります。このアドレスがリロケータブルな状態になっていれば、セクションのオフセットが"*section*+offset"という書式でプリントされます。シンボルが絶対アドレスを受け取っている場合、このアドレスはプリントされません。まだ定義されていないシンボル(**X**マークが付いているもの)の場合、アドレスのカラムにダッシュがプリントされます。これは"未知"であることを示しています。

他の場合と同じように冗長オプションを追加することができます。冗長オプションを使用すると、プリントされる情報が多くなります。

```
pr88 -ev calc.out
```

```
Variable        S       Type    Attrib  MAU     Amod    Address/Size
-------------------------------------------------------------------------
__start_cpt     I       -       -       8       1       .startup + 0x00
__START         I       -       -       8       1       .startup + 0x00
__exit          I       -       -       8       1       .startup + 0x20
__copytable     I       -       -       8       1       .startup + 0x22
_main           I       -       -       8       1       .text + 0x20
__lc_es         X       -       -       8       2       -
__lc_cp         X       -       -       8       2       -
```

カラムが新たに4つ追加されています。"Type"カラムには、シンボルの型が入ります(ある場合)。型の意味については、6.5.2.5節のグローバルの型の節を参照してください。グローバルの型は、リンクのとき、シンボルの型チェックに使用されます。"Attrib"カラムでは、シンボルの属性が指定されます(ある場合)。たとえば、属性値0x0020は、シンボルがアセンブラによって生成されていることを示します。"MAU"カラムには、最小アドレス可能単位がビット単位で表示されます。そのためMAU 8は、シンボルが8ビットアドレス可能になっていることを示します。"Amod"カラムには、シンボルのアドレッシングモードが入ります。

6.5.2.5 オプション-g：グローバルの型の情報の表示

リンカは、グローバルの型の情報を使用して、外部部分にあるシンボルの型の不一致をチェックします。コンパイル時にオプション**-gn**を付けてこれらの型の生成を明示的に抑制しない限り、この情報は常に参照できるようになっています。もちろん型チェックは、型がある場合のみ実行できます。`calc.out`のグローバルの型は、次のようにして表示します。

```
pr88 -g calc.out
```

ただし、この例では"No global types available"というメッセージが出されます。次の例は、グローバルの型がどのように表示されるかを示す一例です。

```
Tp#     Mnem    Name    Entry
-------------------------------
101     X       -       0, T10, 0, 0
102     X       -       0, T1, 0, 0
103     X       -       0, T1, 0, 1, T104
104     P       -       T105
105     n       -       T2, 1
106     X       -       0, T1, 0, 1, T10
107     X       -       0, T10, 0, 1, T10
108     X       -       0, T1, 0, 2, T109, T109
109     T       Byte    T3
10a     X       -       0, T1, 0, 1, T109
...
10f     X       -       0, T1, 0, 3, T12, T110, T12
110     O       -       T111
111     n       -       T2, 0
112     Z       -       T2, 13
113     Z       -       T2, 7
```

最初のカラムには、型のインデックスが入れられます。これはその型が参照されるときの番号になります。この番号は常に16進値になります。0x100より下のインデックスは、いわゆる"基本型"のために予約されているため、番号は0x101から始まります。2番目のカラムには、型のニーモニックが入ります。このニーモニックは、新しい"高レベル"の型を定義します。"Name"カラムには、その型の名前が入ります(ある場合)。

最後のカラムには、型のパラメータが入ります。これらは、高レベルの型の元になる(基本)型がどれであるかを示すもので、モードやサイズなどの他のパラメータが記述されます。型の場合、上記の例のように前に"**T**"が付けられます。型105は型2(パラメータリストの"T2")に基づき、型103は型1と型104に基づいていることがわかります。

次の表では、基本型の概要を示しています。

型のインデックス	型	意味
1	void	-
2	char	8ビット符号付き
3	unsigned char	8ビット符号なし
4	short	16ビット符号付き
5	unsigned short	16ビット符号なし
6	long	32ビット符号付き
7	unsigned long	32ビット符号なし
10	float	32ビット浮動小数点
11	double	64ビット浮動小数点
16	int	16ビット符号付き
17	unsigned int	16ビット符号なし

型のニーモニックは、新しく作成される型のクラスを定義します。次の表では、型のニーモニックを示しています。

ニーモニック	説明	パラメータ
G	一般的な構造体	size, [member, T*index*, offset, size]...
N	列挙型	[name, value]...
n	ポインタ修飾子	T*index*, memspace
O	小ポインタ	T*index*
P	大ポインタ	T*index*
Q	型修飾子	q-bits, T*index*
S	構造体	size, [member, T*index*, offset]...
T	型定義	T*index*
t	コンパイラ生成の型	T*index*
U	共用体	size, [member, T*index*, offset]...
X	関数	x-bits, T*index*, 0, nbr-arg, [T*index*]...
Z	配列	T*index*, upper-bound
g	ビット型	sign, nbr-of-bits

ニーモニックn、O、P、Q、T、t、ZのT*index*は、新しい型を構築するときの元になる型共用体です。共用体および構造体のT*index*は、メンバの型インデックスになります。関数型では、最初のT*index*がその関数の戻り型になります。2番目のT*index*は、それぞれのパラメータごとに繰り返され、それぞれのパラメータの型を示します。値-1(0xffffffff)は常に"未知"を意味します。これは、パラメータの番号が未知の場合に関数型で使用されるか、または上位の境界が未知の場合に配列で使用されます。一般的な構造体の場合、サイズ(size)とオフセット(offset)はビット単位で表示されます。最初のsizeは構造体のサイズ、2番目のサイズはメンバのサイズになります。

-gスイッチで取得される型の情報には、冗長オプションに相当するものはありません。

6.5.2.6 オプション-d：デバッグ情報の表示

-dスイッチには2つの形式があります。**-d0**を使用すると、内容が一覧形式で表示されます。

```
pr88 -d0 calc.out

Choose option -d with the number of the file:
 1 - startup
 2 - _copytbl
 3 - calc
```

この時点で、単一の(リンク)ファイルを調べるときは**-d***n*を使用することができます。たとえば、**-d3**を指定すると、calc.objのデバッグ情報のみが表示されます。数値を付けないでオプション**-d**を指定すると、すべてのデバッグ情報を表示することができます。

冗長オプション**-v**を付けずに**-d**スイッチを指定すると、ローカル変数とプロシージャ情報のみが表示されます。**-d**スイッチと冗長スイッチ**-v**を組み合わせると、ローカルの型の情報、行番号、スタックの更新情報、その他のプロシージャ情報も一緒に表示されます。

次の例では、冗長スイッチを使用しています。必要な場合、"Only with verbose on"と表示されます。

```
pr88 -d3v calc.out
```

オブジェクトリーダーでは、最初にヘッダ、次にローカルの型の情報が表示されます。

```
                        ******************************
                        *   O b j e c t   c a l c    *
                        ******************************

                            M o d u l e   i n f o
                            =====================

Type info calc:
===============

No local types available
```

この型の情報は、冗長オプション**-v**を使用した場合に限りプリントされます。この表で表示される情報は、グローバルの型の情報で説明した情報とまったく同じものです。6.5.2.5節を参照してください。

ローカルの型の後に、ローカルのシンボルが表示されます。

```
Symbols calc:
=============

Variable        S       Type    Attrib  MAU     Amod    Address/Size
-----------------------------------------------------------------------
_MODEL          N       -       0x0010  0       0       -
_MODEL          N       -       0x0010  0       0       -
_factorial      N       -       0x0020  8       1       -
_compute        N       -       0x0020  8       1       -
_val            N       -       0x0020  8       2       -
_zero           N       -       0x0020  8       2       -
_cll            N       -       0x0020  8       2       -
```

外部部分のシンボルステータスの値は"**I**"または"**X**"になっていました。ここでは新しい文字"**N**"が入っています。"**N**"はローカルシンボルを表しています。他のエントリの場合、"**G**"や"**S**"が入ることもあります。これらはシンボルではなく、プロシージャです。これらのプロシージャは、その相対位置を定義するためにこの位置にプリントされます。実際のプロシージャ情報は、次の情報のブロックで定義されます。他のプロシージャ情報を、次に示します。プロシージャブロックは、冗長スイッチを使用した場合に限りプリントされます。

```
Procedures calc:
================

No procedures
```

次は、プロシージャの例です。

```
Name            S  Additional information
-----------------------------------------
main            G  0x00, 0x00, T101, QUEENS_PR + 0x00,
                   ( QUEENS_PR + 0x49 ) - 0x01
find_legal_row  S  0x00, 0x00, T120, QUEENS_PR + 0x49,
                   ( QUEENS_PR + 0x156 ) - 0x01
display_board   S  0x00, 0x00, T10a, QUEENS_PR + 0x156,
                   ( QUEENS_PR + 0x2a4 ) - 0x01
display_field   S  0x00, 0x00, T121, QUEENS_PR + 0x2a4,
                   ( QUEENS_PR + 0x302 ) - 0x01
display_status  S  0x00, 0x00, T103, QUEENS_PR + 0x302,
                   ( QUEENS_PR + 0x31d ) - 0x01
```

最初の2つのカラムは、ローカル変数の表の場合と同じです。**G**は外部(グローバル)関数を示し、**S**は静的(ローカル)関数を示します。

それぞれの関数には、5つのパラメータがあり、それぞれ次の意味を持っています。

パラメータ#1　フレームの型。使用されません。

パラメータ#2　フレームのサイズ。スタックの位置に対する関数呼び出しの前で、なおかつローカル変数の直後にあるスタックポインタからの距離になります。

パラメータ#3　関数のタイプ。

パラメータ#4　関数の開始アドレス。リロケータブルオブジェクトの場合、構文"*section* + offset"が使用されます。

パラメータ#5　関数の最終アドレス。param #4を参照してください。

次のデバッグ情報には、行番号情報とスタック情報があります。冗長スイッチをオンにした場合に限り、両方の情報がプリントされます。

```
Lines include/stdarg.h:
=======================
No line info available

Lines include/stdio.h:
======================
No line info available
```

```
Lines queens.c:
===============

Address              | Line  Address              | Line  Address     ...
-------------------------  -------------------------  -----------------
QUEENS_PR + 0x000000 | 52    QUEENS_PR + 0x0000c2 | 90    QUEENS_PR + ...
QUEENS_PR + 0x000000 | 53    QUEENS_PR + 0x0000d9 | 101   QUEENS_PR + ...
QUEENS_PR + 0x000006 | 55    QUEENS_PR + 0x0000d9 | 103   QUEENS_PR + ...
        .                          .                          .
        .                          .                          .
        .                          .                          .
QUEENS_PR + 0x0000bd | 98    QUEENS_PR + 0x00018e | 133   QUEENS_PR + ...
QUEENS_PR + 0x0000c0 | 99    QUEENS_PR + 0x000190 | 136   QUEENS_PR + ...
QUEENS_PR + 0x0000c2 | 100   QUEENS_PR + 0x00019f | 137

Stack info include/stdarg.h:
============================
No stack info available

Stack info include/stdio.h:
===========================
No stack info available

Stack info queens.c:
====================
No stack info available
```

スタック情報では、それぞれの実行可能アドレスに対する実際のスタック位置が表示されます。この値は、関数のローカル変数の直後にある開始位置から実際のスタック位置までの値になります。スタックに1バイト分プッシュした場合、デルタが1増加します。

デバッグ情報はモジュール単位で示され、その最後に、それぞれの関数に対するブロックが付けられます。このブロックでは、関数ごとにローカル変数が示されます。

```
                        P r o c e d u r e   i n f o
                        ===========================

Procedure find_legal_row:
=========================

Symbols find_legal_row:
=======================

Variable  S  Type    Attrib  Mau  Amod  Address/Size
-----------------------------------------------------
accepted  N  0x0109  0x0004   0     0   QUEENS_DA + 0x09
row       N  0x0109  0x0805   0     0   0x02
col       N  0x0109  0x0805   0     0   0x03
chk_row   N  0x0109  0x0005   0     0   0x01
chk_col   N  0x0109  0x0005   0     0   0x00

                E n d   o f   p r o c e d u r e   i n f o
                =========================================
```

6.5.2.7 オプション-i：セクションイメージの表示

-dオプションの場合と同じように、オプション**-i0**を指定すると、使用可能なセクションイメージを表示するよう要求することができます。

```
pr88 -i0 calc.out

Choose option -i with the number of the section:
 1 - .startup_vector
 2 - .startup
 3 - .watchdog_vector
 4 - .watchdog
 5 - .text
 6 - .data
 7 - .zdata
```

イメージの番号を指定することにより、表示するイメージを選択することができます。

```
pr88 -i5 calc.out

Section .text:
==============

02 32 05 e3 ce 00 01 c4 f8 b0 cf 88 f3 f0 50 b4
cf d8 ce a1 51 d8 ce e1 cf 00 b1 cc a9 01 cf f8
rr rr rr rr rr rr rr rr rr rr rr rr rr
```

冗長フラグを指定することで、セクションのオフセットや絶対アドレスを表示することもできます。

```
pr88 -i5v calc.out

Section .text:
==============

000000  02 32 05 e3 ce 00 01 c4 f8 b0 cf 88 f3 f0 50 b4 .2............P.
000010  cf d8 ce a1 51 d8 ce e1 cf 00 b1 cc a9 01 cf f8 ....Q...........
000020  rr rr rr rr rr rr rr rr rr rr rr rr rr           .............
```

ダンプでは、常にアドレス単位で16進数のバイト値が表示されます。ただし場合によっては、ダンプすることができません。たとえば、特定のバイトがまだリロケートされていないような状況では、それを決定することができないため、バイトは**rr**で表現されます。

また、セクションイメージがない場合もあります。これにはたとえば、スタートアップ時にクリアされるセクションや複数の関数によって共有されるセクションなどが該当します。そのような場合、起動構文(冗長モードをオン)の後、リーダーは次のようにプリントします。

```
pr88 -i7v calc.out

Section .zdata:
===============

No image allowed, cleared during startup
```

絶対ファイルを読み込むこともできます。絶対ファイルでは、さまざまなセクションを組み合わせて新しいクラスタを生成することができます。これらのクラスタには、セクションと同じ属性がないため、リーダーはオーバーレイ領域が配置される場所を認識できなくなります。

```
pr88 -v -i1 calc.abs

Section rom:
============

000000  00 53 f9 ss 00 02 02 00 f0 00 00 00 00 00 00 00 .S..............
000010  01 01 00 f0 01 00 00 b6 00 00 00 03 00 ss ss ss .......2........
000020  ss ss ss ss ss ss 02 32 05 e3 ce 00 01 c4 f8 b0 ....P.....Q.....
000030  cf 88 f3 f0 50 b4 cf d8 ce a1 51 d8 ce e1 cf 00 ................
....
```

これを見るとわかるように、リーダーは、実際にオブジェクトファイルから読み込めるバイトのみをプリントします。ダンプの**ss**は、スクラッチメモリを示しています。これはスタートアップコードで初期化されている可能性も初期化されていない可能性もあります。このような情報は、リーダーからは認識できません。スタートアップコードは、ロケータが生成したテーブルを使用して、情報を取得することができます。"4 ロケータ"を参照してください。

6.5.3 低レベルでのオブジェクトの表示

6.5.3.1 オブジェクト層

良く知られたOSIの通信層モデルの場合、オブジェクトファイル内の層も区別することができます。オブジェクトファイルは、コンパイラがデバッガやターゲットボードと通信するときに使用される媒体です。もっとも低いレベルは、大容量記憶装置(多くの場合ディスク)として区別することができます。リーダーが参照できるもっとも低いレベルは、生バイトです。

pr88では、この層を"*レベル0*"として扱います。

もちろん、レベル0のバイトには意味があります。オブジェクトフォーマットは、IEEE 695に準拠したフォーマットになっているため、オブジェクトファイルはMUFOMコマンドの集まりになります。一般的には、オブジェクト生成ツールは、コマンドをオブジェクト消費ツールに送信します。これらのコマンドについては、公式のIEEE標準[1]で詳細に記述されています。レベル0の生バイトは、コード化されたMUFOMコマンドになります。MUFOMコマンドは、生バイト層のすぐ上の層で解釈されます。

pr88では、この層を"*レベル1*"として扱います。

次の層はMUFOM環境です。この層では、MUFOMコマンドの実行によって、型およびセクションのテーブルが構築され、値が割り当てられて、属性が設定されます。IEEE文書では、MUFOM変数の有効範囲、セクション属性命名規則について、一部、定義済みの意味を記述しています。このことは、もっとも高いMUFOM層からも参照できます。

pr88では、この層を"*レベル2*"として扱います。

これらの第1の層については、コンパイラやデバッガ/ターゲットボードに完全な通信チャネルがあります。次の層(現時点ではリーダーによってサポートされていない)では、ターゲットおよび言語固有の情報について、コンパイラとデバッガ間のプロトコルを定義しています。

次の節では、リーダーをもっとも低いレベルで使用する方法について例を示します。ここまでは、リーダーのデフォルトレベル、レベル2を使用してきました。

6.5.3.2 レベルオプション -ln

レベル1

他のレベルに切り換えるのは簡単です。参照したいレベルを**-l**オプションに付けて使用します。たとえば、`calc.out`のセクション部分をレベル1で参照したい場合、次のように入力します。

```
pr88 -l1 -s calc.out
```

```
ST:     1, RXAZS, .startup_vector
AS:     L1, 0x0
AS:     S1, 0x2
ST:     2, RXZC, .startup
AS:     S2, 0x63
ST:     3, RXAZS, .watchdog_vector
AS:     L3, 0x4
AS:     S3, 0x2
ST:     4, RXZC, .watchdog
AS:     S4, 0x1
ST:     5, RXZC, .text
AS:     S5, 0x2d
ST:     6, WIY2C, .data
AS:     S6, 0x3
ST:     7, WBY2C, .zdata
AS:     S7, 0x1
```

[1] IEEE Trial Use Standard for Microprocessor Universal Format for Object Modules (IEEE std. 695), IEEE Technical Committee on Microcomputers and Microprocessors of the IEEE Computer Society, 1990.

MUFOMコマンドについてあまり知識がない場合、冗長スイッチを使用することができます。AS、SA、STなどの省略コマンドが、それぞれ*Assignment*、*Section alignment*、*Section type*に展開されます。

```
pr88 -v -l1 -s calc.out
```

```
ST:     Section type:
        Nbr = 1, type = RXAZS, name = .startup_vector
AS:     Assignment:
        Variable = L1, expression = 0x0
AS:     Assignment:
        Variable = S1, expression = 0x2
.
.
ST:     Section type:
        Nbr = 7, type = WBY2C, name = .zdata
AS:     Assignment:
        Variable = S7, expression = 0x1
```

MUFOM変数L*n*およびS*n*は、セクション*n*のアドレスおよびサイズとして定義されています。レベル2では(6.5.2.2節を参照)L変数とS変数の意味がすでにわかっていたため、レベル2でL変数とS変数は表示されていません。

レベル0

レベル0への切り替えは、(推察通り)**-l0**を使用して実行します。

```
pr88 -l0s calc.out
```

```
e6 01 d2 d8 c1 da d3 0f 2e 73 74 61 72 74 75 70 5f
76 65 63 74 6f 72
e2 cc 01 81 00
e2 d3 01 02
...
e6 07 d7 c2 d9 02 c3 06 2e 7a 64 61 74 61
e2 d3 07 01
```

各バイトは、MUFOMコマンド構造でプリントされます。使用されているMUFOMコマンドのコードを見つけるのは簡単です。ファイルオフセットを参照したい場合、冗長スイッチを使用することができます。

```
pr88 -l0vs calc.out
```

```
0000b3   e6 01 d2 d8 c1 da d3 0f 2e 73 74 61 72 74 75 70 5f .........startup_
         76 65 63 74 6f 72                                  vector
0000ca   e2 cc 01 81 00                                     .....
0000cf   e2 d3 01 02                                        ....
....
00013c   e6 07 d7 c2 d9 02 c3 06 2e 7a 64 61 74 61         .........zdata
00014a   e2 d3 07 01                                        ....
```

レベルの混合表示

複数のレベルを混合して表示することもできます。たとえば、オプション**-l01**(**-l10**または**-l0 -l1**と同等)を指定すると、レベル0とレベル1を一緒に表示することができます。

```
pr88 -sl01 calc.out
```

```
ST:     1, RXAZS, .startup_vector
        e6 01 d2 d8 c1 da d3 0f 2e 73 74 61 72 74 75 70 5f
        76 65 63 74 6f 72
AS:     L1, 0x0
        e2 cc 01 81 00
AS:     S1, 0x2
        e2 d3 01 02
.
.
.
ST:     7, WBY2C, .zdata
        e6 07 d7 c2 d9 02 c3 06 2e 7a 64 61 74 61
AS:     S7, 0x1
        e2 d3 07 01
```

もちろん、冗長スイッチをオンにすることができます。レベル0とレベル1を切り換えるときは、MUFOMコマンドを使用します。これは、レベル1ではMUFOMコマンドが最小単位であるためです。

レベル1とレベル2を表示する場合、オブジェクト部分ごとに切り替えが行われます。これは、レベル2ではオブジェクト部分が最小単位であるためです。そのため、すべてのセクション関連コマンドが実行されるまでは、これらのすべてのコマンドの結果を表示することはできません。

```
pr88 -s -l1 -l2 calc.out
```

```
ST:     1, RXAZS, .startup_vector
AS:     L1, 0x0
AS:     S1, 0x2
.
.
.
ST:     7, WBY2C, .zdata
AS:     S7, 0x1

Section           Size
------------------------------
.startup_vector   0x000002
.startup          0x000063
.watchdog_vector  0x000002
.watchdog         0x000001
.text             0x00002d
.data             0x000003
.zdata            0x000001
```

6.5.3.3 冗長オプション -vn

6.5.3.2節で見てきたように、レベルスイッチ-**l***n*を使用することにより、低レベルに切り換えることができます。冗長プリントアウトが必要な場合、-**v**オプションを使用することができます。

レベル0の冗長出力を表示したい場合、-**v0**を指定することもできます。この場合オプション-**v***n*は、-**v** -**l***n*(または-**vl***n*)の省略形です。この新しい表記の場合、レベルの混合出力が必要なときにも、レベルごとに冗長オプションを選択できるという利点があります。たとえば、-**l0** -**v1**と指定すると、レベル0が非冗長モード、レベル1が冗長モードになります。

```
pr88 -sl0v1 calc.out
ST:     Section type:
        Nbr = 1, type = RXAZS, name = .startup_vector
        e6 01 d2 d8 c1 da d3 0f 2e 73 74 61 72 74 75 70 5f
        76 65 63 74 6f 72
AS:     Assignment:
        Variable = L1, expression = 0x0
        e2 cc 01 81 00
AS:     Assignment:
        Variable = S1, expression = 0x2
        e2 d3 01 02
.
.
.
ST:     Section type:
        Nbr = 7, type = WBY2C, name = .zdata
        e6 07 d7 c2 d9 02 c3 06 2e 7a 64 61 74 61
AS:     Assignment:
        Variable = S7, expression = 0x1
        e2 d3 07 01
```

一般的に冗長スイッチ-**v**(数値を付けない)を使用すると、選択したすべてのレベルが冗長モードになります。冗長スイッチ-**v***n*を指定すると、レベル*n*が選択され、レベル*n*のみが冗長モードになります。

Appendix A　Cコンパイラのエラーメッセージ

以下のエラーリストは、最初にエラータイプ、次にエラー番号とメッセージを示しています。エラータイプは、以下の文字で示されます。

I　情報
E　エラー
F　致命的なエラー
S　システムエラー
W　ワーニング

フロントエンド

F 1:　evaluation expired
評価版の有効期限が切れています。

W 2:　unrecognized option: '*option*'
このオプションが存在しません。正しいオプションを使用したかどうか、起動書式をチェックしてください。

E 4:　expected *number* more '#endif'
コンパイラのプリプロセッサ部分に、"#if"、"#ifdef"、"#ifndef"指示文がありますが、同じソースファイルに、対応する"#endif"がありません。"#if"、"#ifdef"、"#ifndef"のそれぞれに、対応する"#endif"があるかどうか、ソースファイルをチェックしてください。

E 5:　no source modules
コンパイルするソースファイルを少なくとも1つ指定する必要があります。

F 6:　cannot create "*file*"
出力ファイルまたは一時ファイルが作成できませんでした。ディスク容量が十分あるかどうか、および指定ディレクトリに書き込み許可があるかどうかチェックしてください。

F 7:　cannot open "*file*"
このファイルが実際に存在するかどうかチェックしてください。名前をミスタイプしたか、このファイルが別のディレクトリに存在している可能性があります。

F 8:　attempt to overwrite input file "*file*"
出力ファイルの名前は、入力ファイルと異なる名前にする必要があります。

E 9:　unterminated constant character or string
このエラーは、文字列を二重引用符(")で閉じなかった場合や文字定数を一重引用符(')で閉じなかった場合に発生します。このエラーメッセージは、多くの場合、E 19エラーメッセージの後に付きます。

F 11:　file stack overflow
このエラーは、ファイルインクルードのネストの深さが最大数(50)を越えた場合に発生します。他の#includeファイルを含む#includeファイルをチェックしてください。ネストされているファイルを、単純なファイルに分割してみてください。

F 12:　memory allocation error
すべての空きメモリ空間が使用されています。常駐プログラムを除去してメモリを解放する、ファイルを小さなソースファイルに分割する、式を小さな部分式に分ける、メモリを増やすなどの手段を実行してください。

W 13:　prototype after forward call or old style declaration - ignored
それぞれの関数のプロトタイプが実際の呼び出しの前にあることをチェックしてください。

E 14:　';' inserted
式の文にセミコロンが必要です。たとえば、`{ int i; ++i }`の`++i`の後に必要になります。

E 15:　missing filename after -o option
-oオプションの後に、出力ファイル名を付ける必要があります。

E 16: bad numeric constant
定数は、その構文に準拠する必要があります。たとえば、`08`は8進数の構文に違反しています。また定数は、割り当てられた型を表す上で大きくなりすぎないようにします。たとえば、`int i = 0x1234567890;`とした場合、int型としては値が大きすぎます。

E 17: string too long
このエラーは、最大文字列サイズ(1500)を越えた場合に発生します。文字列のサイズを小さくしてください。

E 18: illegal character (0x*hexnumber*)
16進ASCII値0x*hexnumber*の文字は、ここでは使用できません。たとえば16進値0x23を付けた"#"文字をプリプロセッサコマンドで使用する場合、スペース以外の文字を前に入れることはできません。次の例では、エラーが発生します。

```
char *s = #S ;    // error
```

E 19: newline character in constant
文字定数または文字列定数で改行を使用できるのは、円記号(\)またはバックスラッシュ(\)が前に付いている場合のみです。ソースファイルで2行にわたる文字列を分割するときは、次のいずれかの方法を使用します。

- 最初の行の最後に文字継続文字、円記号(\)またはバックスラッシュ(\)を付けます。
- 最初の行の文字列の最後に二重引用符を付け、次の行の文字列の最初にも引用符をもう1つ付けます。

E 20: empty character constant
文字定数に入れる文字は1文字だけです。空の文字定数('')は使用できません。

E 21: character constant overflow
文字定数に入れる文字は1文字だけです。エスケープシーケンス(たとえばタブを示す**¥t**など)は1文字に変換されます。

E 22: '#define' without valid identifier
"#define"の後には識別子を指定する必要があります。

E 23: '#else' without '#if'
"#else"は、対応する"#if"、"#ifdef"、"#ifndef"構文の中でのみ使用することができます。この文の前で、"#if"、"#ifdef"、"#ifndef"文が有効になっていることを確認してください。

E 24: '#endif' without matching '#if'
"#endif"がありますが、それに対応するプリプロセッサ指示文"#if"、"#ifdef"、"#ifndef"がありません。それぞれの"#if"、"#ifdef"、"#ifndef"文に対応する"#endif"があるかどうか確認してください。

E 25: missing or zero line number
"#line"では、0以外の数値を指定する必要があります。

E 26: undefined control
コントロール行("#*identifier*"を含む行)には、既知のプリプロセッサ指示文が入っていなければなりません。

W 27: unexpected text after control
"#ifdef"および"#ifndef"には、識別子が1つだけ必要です。また、"#else"および"#endif"には、改行以外の文字は必要ありません。"#undef"では、識別子が1つだけ必要です。

W 28: empty program
ソースファイルには、外部定義が最低でも1つ必要になります。コメント以外何もないソースファイルは、空のプログラムと見なされます。

E 29: bad '#include' syntax
"#include"の後には、有効なヘッダ名構文を続ける必要があります。たとえば、`#include <stdio.h`となっている場合、">"が必要です。

E 30: include file "*file*" not found
"#include"指示文の後に、必ず既存のインクルードファイルを指定する必要があります。また、インクルードファイルには、正しいパスを指定していなければなりません。

E 31: end-of-file encountered inside comment
コンパイラがコメントを読み込んでいるときに、エンドオブファイルが見つかりました。コメントが正しく終了していない可能性があります。ANSI-Cスタイルのコメントを使用するときは、コメントの最後に"*/"を付ける必要があります。

E 32: argument mismatch for macro "*name*"
関数形式のマクロを起動する場合、引数の数は、定義のパラメータの数と一致させる必要があります。また、関数形式のマクロを起動するときは、")"を最後に付ける必要があります。次に示すのは、このエラーの例です。

```
#define A(a) 1
int i = A(1,2); /* error */
#define B(b) 1
int j = B(1;    /* error */
```

E 33: "*name*" redefined
この識別子が、複数回定義されているか、後の宣言が前の宣言と異なっているかのいずれかです。次の例の場合、エラーが発生します。

```
int  i;
char i;         /* error */
main( )
{
}
main( )
{
      int j;
      int j;    /* error */
}
```

W 34: illegal redefinition of macro "*name*"
マクロは、再定義されるマクロの本体が、最初に定義されたマクロの本体とまったく同じ場合に限り、再定義することができます。
このワーニングは、マクロをコマンド行およびソースで"#define"指示文を付けて定義するときに発生します。また、インクルードファイルからマクロをインポートした場合にも発生することがあります。ワーニングを出さないようにするためには、いずれかの定義を削除するか、2番目の定義の前に"#undef"指示文を使用します。

E 35: bad filename in '#line'
#lineの文字列リテラルは、(ある場合)"wide-char"文字列でなければなりません。そのため、`#line 9999 L"t45.c"`などは使用できません。

W 36: 'debug' facility not installed
"#pragma debug"は、コンパイラのデバッグバージョンでのみ使用できます。

W 37: attempt to divide by zero
0による除算または剰余算が見つかりました。式を調整して、除算または剰余算の2番目のオペランドが0になっていないかテストしてください。

E 38: non integral switch expression
`switch`条件式は、整数値を評価する必要があります。そのため、`char *p = 0; switch (p)`は使用できません。

F 39: unknown error number: *number*
このエラーは、発生してはならないものです。万一このエラーが発生した場合は、セイコーエプソン(株)に連絡して、正確なエラーメッセージを知らせてください。

W 40: non-standard escape sequence
エスケープシーケンス("\"または"\"の後に数値か文字)のスペリングをチェックしてください。不正なエスケープ文字が含まれています。たとえば、`¥c`を指定した場合にこのワーニングが発生します。

E 41: '#elif' without '#if'
"#elif"指示文が、"#if"、"#ifdef"、"#ifndef"構文内にありませんでした。この文の前で、対応する"#if"、"#ifdef"、"#ifndef"文が有効になっていることを確認してください。

E 42: syntax error, expecting parameter type/declaration/statement
パラメータリスト、宣言、文のいずれかに、構文エラーがあります。原因としては、数値構文のエラー、予約語の使用、演算子のエラー、パラメータ型の不足、トークンの不足など、さまざまなものが考えられます。

E 43: unrecoverable syntax error, skipping to end of file
コンパイラが、回復不可能なエラーを検出しました。このエラーは、ほとんどの場合、他のエラーの後に現れます。通常エラー42の後になります。

I 44: in initializer "*name*"
定数イニシャライザが正しいかどうかチェックするときの情報メッセージです。

E 46: cannot hold that many operands
値スタックには、20を越えるオペランドを入れることができません。

E 47: missing operator
式の中に演算子が必要です。

E 48: missing right parenthesis
")"が必要です。

W 49: attempt to divide by zero - potential run-time error
0による除算または剰余算を含む式が見つかりました。式を調整して、除算または剰余算の2番目のオペランドが0になっていないかテストしてください。

E 50: missing left parenthesis
"("が必要です。

E 51: cannot hold that many operators
状態スタックには、20を越えるオペレーターを入れることができません。

E 52: missing operand
オペランドが必要です。

E 53: missing identifier after 'defined' operator
`#if defined`(*identifier*)には識別子が必要です。

E 54: non scalar controlling expression
反復条件および"if"条件は、スカラ型になっている必要があります(`struct`、`union`、ポインタは不可)。たとえば、`static struct {int i;} si = {0};`の後に、`while (si) ++si.i;`を指定することはできません。

E 55: operand has not integer type
"#if"指示文のオペランドは、int型定数を評価する必要があります。そのため、`#if 1.`は使用できません。

W 56: '<*debugoption*><*level*>' no associated action
このワーニングは、コンパイラのデバッグバージョンでのみ現れます。このデバッグオプションおよびレベルには、対応するデバッグアクションがありません。

W 58: invalid warning number: *number*
-wオプションで指定されたワーニング番号が存在しません。正しい番号を指定してください。

F 59: sorry, more than number errors
40以上のエラーがあるため、コンパイルが停止しました。

E 60: label "*label*" multiple defined
同じ関数内では、1つのラベルを複数回定義することができません。次に示すのは、このエラーの例です。

```
f( )
{
lab1;
lab1;              /* error */
}
```

E 61: type clash
型の競合が見つかりました。たとえばintまたはdoubleではlongのみが使用でき、struct、union、enumでは指定子を使用することはできません。次に示すのは、このエラーの例です。

```
unsigned signed int i;  /* error */
```

E 62: bad storage class for "*name*"
記憶クラス指定子autoおよびregisterは、外部定義の宣言指定子で使用することはできません。また、パラメータ宣言で使用できる記憶クラス指定子は、registerのみです。

E 63: "*name*" redeclared
この識別子は、すでに宣言されています。コンパイラは2番目の宣言を使用します。次に示すのは、このエラーの例です。

```
struct T { int i; };
struct T { long j; };   /* error */
```

E 64: incompatible redeclaration of "*name*"
この識別子は、すでに宣言されています。同じ関数またはモジュール内にあって、同じオブジェクトまたは関数を参照するすべての宣言では、互換性のある型を指定する必要があります。次に示すのは、このエラーの例です。

```
f( )
{
     int i;
     char i;      /* error */
}
```

W 66: function "*name*": variable "*name*" not used
使用されていない変数が宣言されています。このワーニングを出さないようにするためには、この未使用の変数を削除するか、**-w66**オプションを使用します。

W 67: illegal suboption: *option*
サブオプションが、このオプションで有効ではありません。起動書式をチェックして、使用可能なすべてのサブオプションを調べてください。

W 68: function "*name*": parameter "*name*" not used
使用されていない関数パラメータが宣言されています。このワーニングを出さないようにするためには、この未使用のパラメータを削除するか、**-w68**オプションを使用します。

E 69: declaration contains more than one basic type specifier
型指定子を繰り返すことはできません。次に示すのは、このエラーの例です。

```
int char i;  /* error */
```

E 70: 'break' outside loop or switch
breakは、switchまたはループ(do、for、while)内でのみ使用できます。そのため、if (0) break;は使用できません。

E 71: illegal type specified
指定した型は、このコンテキストでは使用できません。たとえば、変数を宣言するときにvoid型は使用できません。次に示すのは、このエラーの例です。

```
void i;      /* error */
```

W 72: duplicate type modifier
指定子リストまたは修飾子リストで型修飾子を繰り返すことはできません。次に示すのは、このエラーの例です。

```
{ long long i; }  /* error */
```

E 73: object cannot be bound to multiple memories
1つのオブジェクトではメモリ属性を1つだけ使用してください。たとえば、ROMとRAMの両方を同じオブジェクトに指定することはできません。

E 74: declaration contains more than one class specifier
1つの宣言に入れることができる記憶クラス指定子は1つだけです。そのため、register auto i;は使用できません。

E 75: 'continue' outside a loop
`continue`は、ループ本体(`do`、`for`、`while`)の中でのみ使用できます。そのため、`switch (i) {default: continue;}`は使用できません。

E 76: duplicate macro parameter "*name*"
この識別子が、マクロ定義のformat1パラメータリストで複数回使用されています。それぞれのマクロパラメータは一度だけ宣言する必要があります。

E 77: parameter list should be empty
関数定義の一部でない識別子リストは、空でなければなりません。たとえば`int f (i, j, k);`は宣言レベルで使用できません。

E 78: 'void' should be the only parameter
引数をとらない関数の関数プロトタイプでは、`void`が唯一のパラメータになります。そのため、`int f(void, int)`は使用できません。

E 79: constant expression expected
定数式には、カンマを入れることができません。また、ビットフィールド幅、`enum`を定義する式、配列にバインドされた定数、`switch` case式は、すべて`int`型定数式でなければなりません。

E 80: '#' operator shall be followed by macro parameter
"#"演算子の後にはマクロ引数を続ける必要があります。

E 81: '##' operator shall not occur at beginning or end of a macro
"##"(トークン連結)演算子は、隣接するプリプロセッサトークンを一緒にペーストするときに使用されます。そのため、マクロ本体の最初や最後で使用することはできません。

W 86: escape character truncated to 8 bit value
16進エスケープシーケンス("\"または"\"の後に"x"と数値)の値は、8ビット記憶域に収まる必要があります。1文字は、8ビット以下にしなければなりません。次に示すのは、このエラーの例です。

```
char c = '¥xabc'; /* error */
```

E 87: concatenated string too long
生成された文字列が、1500文字の制限を越えています。

W 88: "*name*" redeclared with different linkage
この識別子は、すでに宣言されています。このワーニングは、異なる基本記憶クラスでオブジェクトを宣言し直すとき、両方のオブジェクトがexternやstaticで宣言されていない場合に出されます。次に示すのは、このエラーの例です。

```
int i;
int i( );    /* error E 64 and warning */
```

E 89: illegal bitfield declarator
ビットフィールドは、int型としてのみ宣言することができ、ポインタや関数として宣言することはできません。そのため、`struct {int *a:1;} s;`は使用できません。

E 90: #error *message*
*message*は、"#error"プリプロセッサ指示文で指定する説明テキストです。

W 91: no prototype for function "*name*"
それぞれの関数には、正しい関数プロトタイプが必要です。

W 92: no prototype for indirect function call
それぞれの関数には、正しい関数プロトタイプが必要です。

I 94: hiding earlier one
エラーE 63の後に追加されるメッセージです。2番目の宣言が使用されます。

F 95: protection error: *message*
プロテクションキーの初期化時に問題が発生しました。メッセージは、"Key is not present or printer is not correct"、"Can't read key"、"Can't initialize key"、"Can't set key-model"などになることもあります。

E 96: syntax error in #define
#define id(には、右のかっこ")"が必要です。

E 97: "..." incompatible with old-style prototype
2つの関数が同じ名前のパラメータ型リストを持っている場合、これは古いスタイルの宣言であるため、パラメータリストに省略記号を入れることはできません。次に示すのは、このエラーの例です。

```
int f(int, ...);
int f( );               /* error, old-style */
```

E 98: function type cannot be inherited from a typedef
typedefは関数定義で使用することはできません。次に示すのは、このエラーの例です。

```
typedef int INTFN( );
INTFN f {return (0);}   /* error */
```

F 99: conditional directives nested too deep
"#if"、"#ifdef"、"#ifndef"指示文は、50レベルより深くネストすることができません。

E 100: case or default label not inside switch
case:ラベルまたはdefault:ラベルは、switch内でのみ使用することができます。

E 101: vacuous declaration
宣言の中に足りないものがあります。宣言が空になっているか、中に不完全な文があります。配列宣言子またはstruct、union、enumメンバを宣言する必要があります。次に示すのは、このエラーの例です。

```
int ;                   /* error */
static int a[2] = { };  /* error */
```

E 102: duplicate case or default label
switch caseでは、評価の後値がそれぞれ固有でなければならず、switch内にdefault:ラベルが少なくとも1つ必要になります。

E 103: may not subtract pointer from scalar
ポインタの減算で使用できるオペランドは、ポインタ - ポインタ、またはポインタ - スカラのみです。そのため、スカラ - ポインタは使用できません。次に示すのは、このエラーの例です。

```
int i;
int *pi = &1;
ff(1 - pi);             /* error */
```

E 104: left operand of *operator* has not struct/union type
"."または"->"の最初のオペランドは、struct型またはunion型にならなければなりません。

E 105: zero or negative array size - ignored
配列にバインドされる定数は、0より大きくなければなりません。そのためchar a[0];は使用できません。

E 106: different construction
パラメータ型リストを持つ互換関数型は、パラメータ数と省略記号の使用の点で共通でなければなりません。また対応するパラメータは、互換性のある型でなければなりません。通常このエラーの後に、情報メッセージI 111が来ます。次に示すのは、このエラーの例です。

```
int f(int);
int f(int, int);        /* error different parameter list */
```

E 107: different array sizes
互換関数型のそれぞれの配列パラメータは同じサイズでなければなりません。通常このエラーの後に、情報メッセージI 111が来ます。次に示すのは、このエラーの例です。

```
int f(int [][2]);
int f(int [][3]);       /* error */
```

E 108: different types
互換関数型のそれぞれのパラメータ、およびそれぞれのプロトタイプパラメータの型は、公開されている定義パラメータを持つ互換性のある型でなければなりません。

通常このエラーの後に、情報メッセージI 111が来ます。次に示すのは、このエラーの例です。

```
int f(int);
int f(long);            /* error different type in parameter list */
```

E 109: floating point constant out of valid range

浮動小数点定数には、割り当てられている型に収まる値がなければなりません。浮動小数点定数の有効範囲については、"1.2.3 データ型"を参照してください。次に示すのは、このエラーの例です。

```
float d = 10E9999;     /* error, too big */
```

E 110: function cannot return arrays or functions

関数では、配列型または関数型の戻り型を使用できません。関数に対するポインタは使用できます。次に示すのは、このエラーの例です。

```
typedef int F( ); F f( );  /* error */
typedef int A[2]; A g( );  /* error */
```

I 111: parameter list does not match earlier prototype

パラメータリストをチェックするか、プロトタイプを調整してください。パラメータの数と型は一致する必要があります。このメッセージは、エラーE 106、E 107、E 108の後に表示されます。

E 112: parameter declaration must include identifier

宣言がプロトタイプの場合、それぞれのパラメータの宣言に識別子がなければなりません。また、typedef名として宣言された識別子は、パラメータ名として使用できません。次に示すのは、このエラーの例です。

```
int f(int g, int) {return (g);} /* error */
```

```
typedef int int_type;
int h(int_type) {return (0);}   /* error */
```

E 114: incomplete struct/union type

structまたはunionを使用する前に、その型を知らせる必要があります。次に示すのは、このエラーの例です。

```
extern struct unknown sa, sb;
sa = sb;                /* 'unknown' does not have a defined type */
```

割り当ての左側(lvalue)は、変更可能でなければなりません。

E 115: label "*name*" undefined

goto文が見つかりましたが、同じ関数またはモジュール内にこのラベルがありませんでした。次に示すのは、このエラーの例です。

```
f1( ) { a: ; }     /* W 116 */
f2( ) { goto a; }  /* error, label 'a:' is not defined in f2( ) */
```

W 116: label "*name*" not referenced

このラベルが定義されていましたが、参照されませんでした。ラベルの参照は、同じ関数内またはモジュール内になければなりません。次に示すのは、このエラーの例です。

```
f1( ) { a: ; }  /* 'a' is not referenced */
```

E 117: "*name*" undefined

この識別子は定義されていませんでした。変数の型は、使用する前に宣言で指定する必要があります。このエラーは、その前のエラーの結果として発生することもあります。次に示すのは、このエラーの例です。

```
unknown i;      /* error, 'unknown' undefined */
i = 1;          /* as a result, 'i' is also undefined */
```

W 118: constant expression out of valid range

caseラベルで使用されている定数式が長すぎる可能性があります。また、浮動小数点値をint型に変換するとき、浮動小数点定数が長すぎることがあります。このワーニングは、通常、エラーE 16またはE 109の後に表示されます。次に示すのは、このエラーの例です。

```
int i = 10E88;  /* error and warning */
```

E 119: cannot take 'sizeof' bitfield or void type
ビットフィールドまたはvoid型のサイズが知らされていません。そのためそのサイズを使用することができません。

E 120: cannot take 'sizeof' function
関数のサイズが知らされていません。そのためそのサイズを使用することができません。

E 121: not a function declarator
これは有効な関数ではありません。その前のエラーが原因になっている可能性があります。次に示すのは、このエラーの例です。

```
int f( ) return 0;      /* missing '{}' */
int g( ) { }            /* error, 'g' is not a formal parameter and
                           therefore, this is not a valid function
                           declaration */
```

E 122: unnamed formal parameter
パラメータには、正しい名前を指定する必要があります。

W 123: function should return something
非void関数の戻り値には式がなければなりません。

E 124: array cannot hold functions
関数の配列は使用できません。

E 125: function cannot return anything
式を持つreturnが、void関数にありません。

W 126: missing return (function "*name*")
空でない関数を持つ非void関数には、return文が必要になります。

E 129: cannot initialize "*name*"
宣言子リスト内の宣言子では、初期化文を入れないでください。またextern宣言では、イニシャライザを使用できません。次に示すのは、このエラーの例です。

```
{ extern int i = 0; }   /* error */
int f( i ) int i=0;     /* error */
```

W 130: operands of *operator* are pointers to different types
演算子または割り当て("=")のポインタオペランドは同じ型でなければなりません。たとえば、次のコードの場合、このワーニングが表示されます。

```
long *pl;
int  *pi = 0;
pl = pi;        /* warning */
```

E 131: bad operand type(s) of *operator*
この演算子には、他の型のオペランドが必要です。次に示すのは、このエラーの例です。

```
int  *pi;
pi += 1.;       /* error, pointer on left; needs
                                  integral value on right */
```

W 132: value of variable "*name*" is undefined
変数が定義される前に使用されている場合、このワーニングが発生します。たとえば、次のコードの場合、このワーニングが表示されます。

```
int a,b;
a = b;          /* warning, value of b unknown */
```

E 133: illegal struct/union member type
関数は、structまたはunionのメンバにすることができません。また、ビットフィールドはint型またはunsigned型のみをとることができます。

E 134: bitfield size out of range - set to 1
ビットフィールドの幅は、この型のビット数より多くしたり、負の値にしたりすることはできません。次の例の場合、このエラーが生成されます。

```
struct i { unsigned i : 999; }; /* error */
```

W 135:　statement not reached
指定された文が、実行されません。たとえば、case文でreturnの後に文がある場合などがこれに当たります。

E 138:　illegal function call
関数以外のオブジェクトで関数呼び出しを実行することはできません。次の例の場合、このエラーが生成されます。

```
int i, j;
j = i( );       /* error, i is not a function */
```

E 139:　*operator* cannot have aggregate type
(キャスト)の型名と(キャスト)のオペランドは、スカラでなければなりません(struct、union、ポインタは不可)。次に示すのは、このエラーの例です。

```
static union ui {int a;} ui ;
ui = (union ui)9; /* cannot cast to union */
ff( (int)ui );    /* cannot cast a union to something else */
```

E 140:　*type* cannot be applied to a register/bit/bitfield object or builtin/inline function
たとえば"&"演算子(アドレス)は、レジスタやビットフィールドでは使用できません。そのため、func(&r6);およびfunc(&bitf.a);は無効です。

E 141:　*operator* requires modifiable lvalue
"++"演算子や"--"演算子のオペランド、および割り当てや複合割り当ての左の演算子(lvalue)は、変更可能でなければなりません。次に示すのは、このエラーの例です。

```
const int i = 1;
i = 3;          /* error, const cannot be modified */
```

E 143:　too many initializers
イニシャライザの数をオブジェクトの数より多くすることはできません。次に示すのは、このエラーの例です。

```
static int a[1] = {1, 2}; /* error only one object
                                  can be initialized */
```

W 144:　enumerator "*name*" value out of range
enum定数がintの制限を越えています。次に示すのは、このエラーの例です。

```
enum { A = INT_MAX, B };  /* warning, B does not fit
                                  in an int anymore */
```

E 145:　requires enclosing curly braces
複合イニシャライザは、中かっこで閉じる必要があります。たとえば、int a[ ] = 2;は無効で、int a[ ] = {2};は有効です。

E 146:　argument #*number*: memory spaces do not match
プロトタイプでは、引数のメモリ空間が一致する必要があります。

W 147:　argument #*number*: different levels of indirection
プロトタイプでは、引数と割り当ての型に互換性がなければなりません。次のコードの場合、このワーニングが表示されます。

```
int i; void func(int,int);
func( 1, &i );  /* warning, argument 2 */
```

W 148:　argument #*number*: struct/union type does not match
プロトタイプでは、プロトタイプ化した関数の引数と実際の引数の両方がstructまたはunionでしたが、異なるタグが付いています。タグの型は一致しなければなりません。次の例の場合、このワーニングが表示されます。

```
f(struct s);    /* prototype */
main( )
{
   struct { int i; } t;
   f( t );      /* t has other type than S */
}
```

E 149: object "*name*" has zero size
structまたはunionに、不完全な型のメンバがあります。次に示すのは、このエラーの例です。

```
struct { struct unknown m; } s; /* error */
```

W 150: argument #*number*: pointers to different types
プロトタイプでは、引数のポインタ型に互換性がなければなりません。次の例の場合、このワーニングが表示されます。

```
int f(int*);
long *l;
f(l);           /* warning */
```

W 151: ignoring memory specifier
struct、union、enumのメモリ指定子が無視されます。

E 152: operands of *operator* are not pointing to the same memory space
オペランドが同じメモリ空間をポイントしていることを確認してください。このエラーは、たとえばポインタを、異なるメモリ空間のポインタに割り当てようとすると発生します。

E 153: 'sizeof' zero sized object
暗黙的または明示的なsizeof演算で、未知のサイズのオブジェクトが参照されています。このエラーは、通常エラーE 119またはE 120の後に表示され、"sizeof"をとることができなくなります。

E 154: argument #*number*: struct/union mismatch
プロトタイプでは、プロトタイプ化した関数の引数と実際の引数のうち、いずれかだけがstructまたはunionでした。型は一致しなければなりません。次の例の場合、このワーニングが表示されます。

```
f(struct s);    /* prototype */
main( )
{
   int i;
   f( i );      /* i is not a struct */
}
```

E 155: casting lvalue '*type*' to '*type*' is not allowed
"++"演算子や"--"演算子のオペランド、または割り当てや複合割り当ての左の演算子(lvalue)は、別の型にキャストすることができません。次に示すのは、このエラーの例です。

```
int i = 3;
++(unsigned)i;  /* error, cast expression is not an lvalue */
```

E 157: "*name*" is not a formal parameter
宣言子に識別子リストがある場合、宣言子リストにはその識別子のみを入れることができます。次に示すのは、このエラーの例です。

```
int f( i ) int a;       /* error */
```

E 158: right side of *operator* is not a member of the designated struct/union
"."または"->"の2番目のオペランドは、指定されたstructまたはunionのメンバでなければなりません。

E 160: pointer mismatch at *operator*
*operator*の両方のオペランドが、有効なポインタでなければなりません。次の例の場合、このエラーが生成されます。

```
int *pi =44;            /* right side not a pointer */
```

E 161: aggregates around *operator* do not match
*operator*の両側にある構造体、共用体、配列の内容は同じでなければなりません。次の例の場合、このエラーが生成されます。

```
struct {int a; int b;} s;
struct {int c; int d; int e;} t;
s = t;                  /* error */
```

E 162: *operator* requires an lvalue or function designator
"&"演算子(アドレス)には、lvalueや関数指名子が必要になります。次に示すのは、このエラーの例です。

```
int i;
i = &( i = 0);
```

W 163: operands of *operator* have different level of indirection
演算子のポインタまたはアドレスの型は、割り当てと互換性がなければなりません。次に示すのは、このワーニングの例です。

```
char **a;
char *b;
a = b;                  /* warning */
```

E 164: operands of *operator* may not have type 'pointer to void'
*operator*のオペランドにオペランド(`void *`)がありません。

W 165: operands of *operator* are incompatible: pointer vs. pointer to array
ポインタの型または演算子のアドレスは、割り当てと互換性がなければなりません。ポインタは、配列へのポインタに割り当てることができません。次に示すのは、このワーニングの例です。

```
main( )
{
      typedef int array[10];
      array a;
      array *ap = a;    /* warning */
}
```

E 166: *operator* cannot make something out of nothing
型`void`を他の型にキャストすることはできません。次の例の場合、このエラーが生成されます。

```
void f(void);
main( )
{
      int i;
      i = (int)f( );    /* error */
}
```

E 170: recursive expansion of inline function "*name*"
`_inline`関数は再帰的に使用することができません。次の例の場合、このエラーが生成されます。

```
_inline int a (int i)
{
      a(i);     /* recursive call */
      return i;
}
main( )
{
      a(1);     /* error */
}
```

E 171: too much tail-recursion in inline function "*name*"
関数レベルが40以上の場合、このエラーが現れます。次の例の場合、このエラーが生成されます。

```
_inline void a ( )
{
      a( );
}
main( )
{
      a( );
}
```

W 172: adjacent string have different types
2つの文字列を連結するとき、両方の文字列が同じ型でなければなりません。次の例の場合、このワーニングが生成されます。

```
char b[] = L"abc""def"; /* strings have different types */
```

E 173: 'void' function argument
関数は、void型の引数をとることができません。

E 174: not address constant
定数アドレスが使用される予定になっていました。スタティック変数と異なり、auto変数には、固定されたメモリロケーションがないため、auto変数のアドレスは定数になりません。次に示すのは、このエラーの例です。

```
int *a;
static int *b = a;      /* error */
```

E 175: not an arithmetic constant
定数式では、割り当て演算子、"++"演算子、"--"演算子、関数を使用できません。次に示すのは、このエラーの例です。

```
int *a;
static int b = a++;     /* error */
```

E 176: address of automatic is not a constant
スタティック変数と異なり、auto変数には、固定されたメモリロケーションがないため、auto変数のアドレスは定数になりません。次に示すのは、このエラーの例です。

```
int *a;                 /* automatic */
static int *b = &a;     /* error */
```

W 177: static variable "*name*" not used
使用されていないスタティック変数が宣言されています。このワーニングを出さないようにするためには、未使用のスタティック変数を削除します。

W 178: static function "*name*" not used
呼び出されていない静的関数が宣言されています。このワーニングを出さないようにするためには、未使用の関数を削除します。

E 179: inline function "*name*" is not defined
インライン関数のプロトタイプのみがあり、実際のインライン関数が存在しないことが原因です。次に示すのは、このエラーの例です。

```
_inline int a(void);    /* prototype */
main( )
{
      int b;
      b = a( );         /* error */
}
```

E 180: illegal target memory (*memory*) for pointer
ポインタが*memory*をポイントしていません。たとえば、ビットアドレス可能なメモリを示すポインタは使用できません。

W 182: argument #*number*: different types
プロトタイプで、引数のタイプに互換性がなければなりません。

I 185: (prototype synthesized at line *number* in "*name*")
古いスタイルのプロトタイプが含まれているソースファイルの位置を通知する情報メッセージです。このメッセージの前に、E 146、W 147、W 148、W 150、E 154、W 182、E 203が表示されます。

E 186: array of type bit is not allowed
配列には、ビット型変数を入れることができません。

E 187: illegal structure definition
構造体は、メンバが知らされている場合に限り、定義(初期化)することができます。そのため`struct unknown s = { 0 };`は使用できません。

E 188: structure containing bit-type field is forced into bitaddressable area
このエラーは、ビット型メンバを含む構造体でビットアドレス可能な記憶タイプを使用するときに発生します。

E 189: pointer is forced to bitaddressable, pointer to bitaddressable is illegal
ビットアドレス可能なメモリを示すポインタは使用できません。

W 190: "long float" changed to "float"
ANSI Cの浮動小数点定数は、定数に接尾辞"f"が付いている場合を除き、`double`型を持つものとして扱われます。浮動小数点定数を使用するオプションを指定した場合、`123.12fl`などの`long`浮動小数点定数は、`float`に変更されます。

E 191: recursive struct/union definition
`struct`または`union`には、それ自体を入れることができません。次の例の場合、このエラーが生成されます。

```
struct s { struct s a; } b;     /* error */
```

E 192: missing filename after -f option
-fオプションには、ファイル名引数を付ける必要があります。

E 194: cannot initialize typedef
typedef変数に値を割り当てることはできません。そのため、`typedef i=2;`は使用できません。

F 199: demonstaration package limits exceeded
デモパッケージには、製品版にはない一定の制限があります。製品版については、セイコーエプソン(株)にお問い合わせください。

W 200: unknown pragma - ignored
コンパイラは、未知のプラグマについては無視します。たとえば`#pragma unknown`などがこれに当たります。

W 201: "*name*" cannot have storage type - ignored
`register`変数またはオートマチック/パラメータには、記憶タイプを使用することができません。このワーニングを出さないようにするためには、記憶タイプを削除するか、関数の外で変数を使用します。

E 202: "*name*" is declared with 'void' parameter list
関数が何も受け付けない場合(`void`パラメータリスト)、関数に引数を付けて呼び出すことはできません。次に示すのは、このエラーの例です。

```
int f(void);            /* void parameter list */
main( )
{
      int i;
      i = f(i);         /* error */
      i = f( );         /* OK */
}
```

E 203: too many/few actual parameters
プロトタイプでは、関数の引数の数が、その関数のプロトタイプと一致している必要があります。次に示すのは、このエラーの例です。

```
int f(int);             /* one parameter */
main( )
{
      int i;
      i = f(i,i);       /* error, one too many */
      i = f(i);         /* OK */
}
```

W 204: U suffix not allowed on floating constant - ignored
浮動小数点定数には、"U"接尾辞または"u"接尾辞を付けることができません。

W 205: F suffix not allowed on integer constant - ignored
int型定数には、"F"接尾辞または"f"接尾辞を付けることができません。

E 206: '*name*' named bit-field cannot have 0 width
ビットフィールドは、0より大きい値を持つint型定数式でなければなりません。

E 212: "*name*": missing static function definition
staticプロトタイプを持つ関数に、その定義がありません。

W 303: variable '*name*' possibly uninitialized
関数が何かを返すことになっているにもかかわらず、初期化文が、到達しない箇所にあります。次に示すのは、このワーニングの例です。

```
int a;
int f(void)
{
   int i;
   if ( a )
   {
      i = 0;            /* statement not reached */
   }
   return i;            /* warning */
}
```

E 327: too many arguments to pass in registers for _asmfunc '*name*'
_asmfunc関数は、Cとアセンブラの間で固定レジスタベースのインタフェースを使用しますが、このときに受け渡しする引数の数は、使用可能なレジスタの数によって制限を受けます。関数*name*で、この制限を越えてしまいました。

バックエンド

W 501: function qualifier used on non-function
関数修飾子は、関数でのみ使用できます。

E 502: Intrinsic function '_int()' needs an immediate value as parameter
_int()組み込み関数の引数は、任意のタイプの整数式ではなく、整数定数式でなければなりません。

E 503: Intrinsic function '_jrsf()' needs an immediate value 0..3
指定する値は、0から3までの定数値でなければなりません。

W 508: function qualifier duplicated
使用できる関数修飾子は1つだけです。関数修飾子が重複する場合、いずれかを削除してください。

E 511: interrupt function must have void result and void parameter list
_interrupt(*n*)で宣言された関数は、引数を受け付けないため、何も返すことができません。

W 512: '*number*' illegal interrupt number (0, or 3 to 251) - ignored
割り込みベクタ番号は、0、または3から251の範囲内になければなりません。他の数値の場合は不正になります。

E 513: calling an interrupt routine, use '_swi()'
割り込み関数は、直接呼び出すことができません。組み込み関数_swi()を使用しなければなりません。

E 514: conflict in '_interrupt'/'_asmfunc' attribute
現在の関数修飾子宣言および前回の関数修飾子宣言の属性が同じになっていません。

E 515: different '_interrupt' number
現在の関数修飾子宣言および前回の関数修飾子宣言の番号が同じになっていません。

E 516: '*memory_type*' is illegal memory for function
記憶タイプがこの関数で有効ではありません。

E 517: conversion of long address to short address
このエラーは、ポインタ変換が必要な場合に出されます。たとえば、_hugeポインタを_nearポインタに割り当てる場合などがこれに該当します。

F 524: illegal memory model
コンパイラの使用法を参照して、**-M**オプションの正しい引数を調べてください。

E 526: function qualifier '_asmfunc' not allowed in function definition
`_asmfunc`は、関数プロトタイプでのみ使用できます。

E 528: _at( ) requires a numerical address
数値アドレスに評価される式のみを使用することができます。

E 529: _at( ) address out of range for this type of object
指定されたメモリ空間に、絶対アドレスがありません。

E 530: _at( ) only valid for global variables
絶対アドレスには、グローバル変数のみを置くことができます。

E 531: _at( ) only allowed for uninitialized variables
絶対変数は初期化することができません。

E 532: _at( ) has no effect on external declaration
`extern`と宣言した場合、その変数はコンパイラによって割り当てられません。

W 533: c88 language extension keyword used as identifier
言語拡張機能キーワードは予約語になっており、予約語は識別子として使用することができません。

E 536: illegal syntax used for default section name '*name*' in -R option
-Rオプションの説明を参照し、正しい構文を調べてください。

E 537: default section name '*name*' not allowed
-Rオプションの説明を参照し、正しい構文を調べてください。

W 538: default section name '*name*' already renamed to '*new_name*'
-Rオプションのいずれか、**renamesect**プラグマ、他の名前のみを使用してください。

W 542: optimization stack underflow, no optimization options are saved with #pragma optimize
このワーニングは、前の`#pragma endoptimize`でオプションが保存されていないときに`#pragma endoptimize`を使用すると発生します。

W 555: current optimization level could reduce debugging comfort (-g);
これらの最適化設定では、HLLデバッグ競合が発生します。

E 560: Float/Double: not yet implemented
浮動小数点は、以下のバージョンでサポートされています。

Appendix B アセンブラのエラーメッセージ

アセンブラでは、標準エラー出力にエラーメッセージが出されます。アセンブラのリストオプションがオンになっている場合、アセンブラがリストファイルの生成を始めるときにエラーメッセージがリストファイルにも出力されます。エラーメッセージは次のようなレイアウトになります。

[E | F | W] *error_number*:　*filename* line *number*: *error_message*

例：

as88 E214: ¥tmp¥tst.src line 17: illegal addressing move

この例では、エラーの重大性(E：エラー、F：致命的、W：警告)の後にエラー番号、さらにその後にソースのファイル名と行番号が報告されています。行の最後の部分は、エラーメッセージのテキストです。

次に、**as88**のすべての警告(W)、エラー(E)、致命的エラー(F)について説明します。

警告(W)

アセンブラは、以下の警告を生成します。

W 101:　use *option* at the start of the source; ignored
プライマリオプションは、ソースの最初に使用しなければなりません。

W 102:　duplicate attribute "*attribute*" found
EXTERN擬似命令の属性が複数回使用されています。重複している属性を削除してください。

W 104:　expected an attribute but got *attribute*; ignored

W 105:　section activation expected, use *name* directive
SECT擬似命令を使用してセクションをアクティブにしてください。

W 106:　conflicting attributes specified "*attributes*"
EXTERN擬似命令で競合する属性が2回使用されています。たとえばEXTERNとINTERNなどがこれに当たります。どちらを使用するか決めて、使用しない方を削除してください。

W 107:　memory conflict on object "*name*"
ラベルまたは他のオブジェクトが明示的または暗黙的に定義されていますが、使用されているメモリタイプ同士に互換性がありません。オブジェクト*name*のすべての使用と定義をチェックして、この矛盾を取り去ってください。

W 108:　object attributes redefinition "*attributes*"
ラベルまたは他のオブジェクトが明示的または暗黙的に定義されていますが、使用されている属性同士に互換性がありません。たとえばEXTERNとINTERNなどがこれに当たります。オブジェクトのすべての使用と定義をチェックして、この矛盾を取り去ってください。

W 109:　label "*label*" not used
ラベル*label*が、GLOBAL擬似命令で定義されていますが、定義も参照もされていません。またはラベルが、LOCAL擬似命令で定義されていますが、参照されていません。(LOCALラベルの場合)このラベルと定義を削除してください。

W 110:　extern label "*label*" defined in module, made global
ラベル*label*が、EXTERN擬似命令で定義されていますが、ソース内のラベルとして定義されています。このラベルはグローバルラベルとして扱われます。EXTERN定義をGLOBALまたはいずれかの識別子に変更してください。

W 111:　unknown $LIST flag "*flag*"
$LISTコントロールに未知の*flag*を指定しています。使用できる引数については、$LISTコントロールの説明を参照してください。

W 112:　text found after END; ignored
END擬似命令がソースファイルの終わりを指定しています。END擬似命令の後にあるすべてのテキストは無視されます。テキストを削除してください。

W 113: unknown $MODEL specifier; ignored
未知のモデルを指定しています。使用できるモデルについては、$MODELコントロールの説明を参照してください。

W 114: $MODEL may only be specified once, it remains "*model*"; ignored
複数のモデルを指定しています。使用できるモデルについては、$MODELコントロールの説明を参照してください。

W 115: use ON or OFF after control name
指定したコントロールには、コントロール名の後にONかOFFを指定する必要があります。詳細については、コントロールの説明を参照してください。

W 116: unknown parameter "*parameter*" for *control-name* control
使用できるパラメータについては、コントロールの説明を参照してください。

W 118: inserted "extern *name*"
シンボル*name*が式の中で使用されていますが、EXTERN擬似命令で定義されていません。アセンブラは、問題のあるシンボルのEXTERN定義を挿入しません。EXTERN定義を追加してください。

W 119: "*name*" section has not the MAX attribute; ignoring RESET

W 120: assembler debug information: cannot emit non-tiof expression for *label*
IEEE-695オブジェクトフォーマットによってサポートされない式が、SYMBレコードにあります。このSYMBレコードがCコンパイラによって生成される場合、エラー報告書に記入してセイコーエプソン(株)に送付してください。

W 121: changed alignment size to *size*

W 123: expression: *type-error*
式が、アドレスで不正な動作を実行しているか、互換性のないメモリ空間を結合しています。式をチェックして、変更してください。

W 124: cannot purge macro during its own definition

W 125: "*symbol*" is not a DEFINE symbol
DEFINEされていないシンボル、またはすでに定義解除されたシンボルをUNDEFしようとしました。問題のあるシンボルに対して、すべてのDEFINE/UNDEFの組み合わせをチェックしてください。

W 126: redefinition of "*define-symbol*"
現在の有効範囲ではシンボルがすでにDEFINEされています。シンボルはこのDEFINEに従って再定義されます。シンボルは、再定義の前にUNDEFしてください。

W 127: redefinition of macro "*macro*"
マクロがすでに定義されています。マクロはこのマクロ定義に従って再定義されます。マクロは、再定義の前にPMACROを使用して消去してください。

W 128: number of macro arguments is less than definition
マクロを定義するとき、引数が少なすぎました。このマクロ呼び出しを持つマクロ定義をチェックしてください。マクロ引数を定義解除するときは、(`DEFINE def ''`のように)空にしておきます。

W 129: number of macro arguments is greater than definition
マクロを定義するとき、引数が多すぎました。このマクロ呼び出しを持つマクロ定義をチェックしてください。余分なマクロ引数は無視されます。

W 130: DUPA needs at least one value argument
DUPA擬似命令には、ダミーパラメータと値パラメータの2つ以上の引数が必要です。値パラメータを1つ以上追加してください。

W 131: DUPF increment value gives empty macro
DUPFマクロに指定したステップ値がDUPFマクロ本体をスキップしています。ステップ値をチェックしてください。

W 132:　IF started in previous file "*file*", line *line*
ENDIFまたはELSEプリプロセッサ擬似命令が、他のファイルのIF擬似命令と対応しています。このファイルに、抜けているENDIF擬似命令またはELSE擬似命令がないかチェックしてください。

W 133:　currently no macro expansion active
@CNT()@関数と@ARG()関数は、マクロ展開の中でのみ使用できます。マクロ定義または式をチェックしてください。

W 134:　"*directive*" is not supported, skipped
指定された擬似命令は、本アセンブラではサポートされていません。この擬似命令はすべて削除してください。

W 135:　define symbol of "*define-symbol*" is not an identifier; skipped definition
コマンド行で**-D**オプションに不正な識別子を指定しています。識別子は、最初に文字を使用して、その後に任意の数の文字、数字、アンダースコアを続ける必要があります。

W 137:　label "*label*" defined *attribute* and *attribute*
ラベルはEXTERN擬似命令およびGLOBAL擬似命令で定義されています。EXTERN擬似命令を削除して、ラベルをグローバルにします。

W 138:　warning: *WARN-directive-arguments*
WARN擬似命令からの出力です。

W 139:　expression must be between *hex-value* and *hex-value*

W 140:　expression must be between *value* and *value*

W 141:　*global/local* label "*name*" not defined in this module; made extern
ラベルが宣言されて使用されていますが、このソースファイルでは定義されていません。ラベルとその使用について現在の有効範囲をチェックし、宣言をEXTERNに変更するか、ラベル定義を追加します。

W 170:　code address maps to zero page
@CPAG関数で指定したコードオフセットが0ページにあります。

W 171:　address offset must be between 0 and FFFF
@CADDR関数または@DADDR関数で指定したオフセットが大きすぎました。オフセットは0と0FFFFhの間でなければなりません。

W 172:　page number must be between 0 and FF
@CADDR関数または@DADDR関数で指定したページ番号が大きすぎました。ページ番号は0と0FFhの間でなければなりません。

エラー(E)

ユーザエラーの状況が発生したときに、アセンブラは次のエラーメッセージを生成します。これらのエラーが発生しても、アセンブリはすぐには停止しません。これらのエラーが発生した場合、アセンブラはアクティブなパスの最後で停止します。

E 200:　*message*; halting assembly
アセンブラが、ソースファイルの処理を停止します。これは情報を伝えるためのメッセージです。すでに報告されているすべてのエラーを取り除いて再度実行してください。

E 201:　unexpected newline or line delimiter
構文チェッカーが、アセンブラの文法に準拠していない改行文字または行区切り文字を見つけました。その行に構文エラーがないかチェックして、問題のある改行文字または行区切り文字を削除してください。

E 202:　unexpected character: '*character*'
構文チェッカーが、アセンブラの文法に準拠していない文字を見つけました。その行に構文エラーがないかチェックして、問題のある文字を削除してください。

E 203:　illegal escape character in string constant
構文チェッカーが、文字列定数の中に、アセンブラの文法に準拠していない不正なエスケープ文字を見つけました。その行に構文エラーがないかチェックして、問題のあるエスケープ文字を削除してください。

E 204:　I/O error: open intermediate file failed (*file*)
アセンブラは、中間ファイルをオープンして、語彙スキャンフェーズを最適化しますが、アセンブラがこのファイルをオープンできません。アセンブラは、環境シンボルTMPDIRが設定されているかどうかチェックします。設定されている場合、このディレクトリで当該ファイルをオープンします。それ以外の場合、ファイルはカレントディレクトリでオープンされます。

E 205:　syntax error: expected *token* at *token*
構文チェッカーが、あるトークンを見つけようとしましたが、別のトークンが見つかりました。予定のトークンを、見つかったトークンの代わりに挿入します。その行に構文エラーがないかチェックしてください。

E 206:　syntax error: *token* unexpected
構文チェッカーが、予定外のトークンを見つけました。問題のトークンを入力から除去すると、アセンブリが継続します。その行に構文エラーがないかチェックしてください。

E 207:　syntax error: missing ':'
構文チェッカーが、ラベル定義またはメモリ空間変更子を見つけましたが、セミコロンが追加されていませんでした。その行に構文エラーがないかチェックして、ニーモニックのスペルミスなどを調べてください。

E 208:　syntax error: missing ')'
構文チェッカーが、閉じかっこを見つけられません。式の構文をチェックして、演算子の不足やかっこのネストなどを調べてください。

E 209:　invalid radix value, should be 2, 8, 10 or 16
RADIX擬似命令は、2、8、10、16のみを受け付けます。その行に構文エラーがないかチェックしてしてください。

E 210:　syntax error
構文チェッカーがエラーを見つけました。

E 211:　unknown model
正しいモデルを置き換えてください。

E 212:　syntax error: expected *token*
構文チェッカーがトークンを見つけようとしましたが、見つかりませんでした。予定のトークンを挿入します。その行に構文エラーがないかチェックしてしてください。

E 213:　label "*label*" defined *attribute* and *attribute*
ラベルが、LOCAL擬似命令、GLOBAL擬似命令、EXTERN擬似命令のいずれかで定義されています。ラベルの有効範囲をチェックして、ラベルの宣言を変更してください。

E 214:　illegal addressing mode
ニーモニックが不正なアドレッシングモードを使用しています。レジスタによるアドレス構成体の使用をチェックしてください。

E 215:　not enough operands
ニーモニックに指定されているオペランドが少なすぎます。ソース行をチェックして、命令を変更してください。

E 216:　too many operands
ニーモニックに指定されているオペランドが多すぎます。ソース行をチェックして、命令を変更してください。

E 217:　*description*
ニーモニックのアセンブル中にエラーが見つかりました。命令をチェックしてください。

E 218: unknown mnemonic: "*name*"
アセンブラが、未知のニーモニックを見つけました。命令をチェックしてください。
ラベルを指定して":"を付け忘れている可能性もあります。

E 219: this is not a hardware instruction (use $OPTIMIZE OFF "H")
アセンブラが汎用命令を見つけましたが、**-Oh**(ハードウェアのみ)オプションまたは$OPTIMIZE ON "H"コントロールが指定されていました。

E 223: unknown section "*name*"
SECT指示文で指定したセクション名が、DEFSECT指示文で定義されていません。SECT名と、それに対応するDEFSECT名をチェックしてください。

E 224: unknown label "*name*"
定義されていないラベルが使用されています。ラベルをチェックして、その定義が同じ名前になっていることを確認してください。

E 225: invalid memory type
不正なメモリ変更子が指定されました。

E 226: unknown symbol attribute: *attribute*

E 227: invalid memory attribute
アセンブラが、未知のロケーションカウンタまたはメモリマッピング属性を見つけました。

E 228: *attr* attribute needs a number
属性*attr*には、1つのパラメータが必要です。たとえばFIT属性などを付けなければなりません。

E 229: only one of the *name* attributes may be specified

E 230: invalid section attribute: *name*
アセンブラが、未知のセクション属性を見つけました。

E 231: absolute section, expected "AT" expression
絶対セクションは、"AT *address*"式を使用して指定する必要があります。

E 232: MAX/OVERLAY sections need to be named sections
MAX属性またはOVERLAY属性を持つセクションには、名前を付ける必要があります。
名前がない場合、ロケータがセクションを重ね書きできません。

E 233: *type* section cannot have *attribute* attribute
コードセクションには、CLEARまたはOVERLAY属性を指定することができません。

E 234: section attributes do not match earlier declaration
同じセクションの前の定義で、他の属性が使用されています。同じ名前を持つすべてのセクション定義をチェックしてください。

E 235: redefinition of section
同じ名前の絶対セクションは一度しかロケートできません。

E 236: cannot evaluate expression of *descriptor*
一部の関数および擬似命令は、アセンブル中にその引数を評価する必要があります。引数を評価できるように、式を変更してください。
シンボルロケーションに循環した従属性がある可能性もあります。

E 237: *descriptor* directive must have positive value
一部の擬似命令は、正の引数をとる必要があります。正の引数に評価されるよう、式を変更してください。

E 238: Floating point numbers not allowed with DB directive
DB擬似命令は浮動小数点数をとることができません。式を変換するか、DW擬似命令を代わりに使用してください。

E 239: byte constant out of range
DB擬似命令は式をバイト単位で格納します。1バイトには、0から255までの数値を入れることができます。

E 240: word constant out of range
DW擬似命令は式をワード単位で格納します。1ワードには、16ビットの数値を入れることができます。式の範囲をチェックしてください。

E 241: Cannot emit non tiof functions, replaced with integral value '0'
浮動小数点式および一部の関数は、IEEE-695オブジェクトフォーマットで表現することができません。式に未知のシンボルが含まれている場合、評価できないため、オブジェクトファイルに出力されません。これらの式を整数式に変更するか、またはアセンブル中に評価されるようにしてください。

E 242: the *name* attribute must be specified
セクションには、CODE属性またはDATA属性がなければなりません。

E 243: use $OBJECT OFF or $OBJECT "*object-file*"

E 244: unknown control "*name*"
指定されたコントロールが存在しません。使用可能なすべてのコントロールの詳細については、"2.7 アセンブラのコントロール"を参照してください。

E 246: ENDM within IF/ENDIF
アセンブラが、IF/ENDIFの間にENDM擬似命令を見つけました。マクロおよびDUP定義をチェックするか、この擬似命令を削除してください。

E 247: illegal condition code
アセンブラが、命令の中に不正な条件コードを見つけました。入力行をチェックしてください。

E 248: cannot evaluate origin expression of org "*name*: *address*"
絶対セクションのすべての起点は、オブジェクトファイルの作成前に評価されなければなりません。定義されていないシンボルまたはロケーションに依存するシンボルが使用されている箇所の、アドレス式をチェックしてください。

E 249: incorrect argument types for function "*function*"
指定された引数が、予定外の型に評価されました。引数式を正しい型に変更してください。

E 250: tiof function not yet implemented: "*function*"
指定されたtiof関数は、まだ実装されていません。

E 251: @POS(,,*start*) start argument past end of string
*start*引数が、最初のパラメータの文字列の長さより長くなっています。

E 252: second definition of label "*label*"
ラベルが、同じ有効範囲内で二度定義されています。ラベル定義をチェックして、重複する定義の名前を変更するか、削除してください。

E 253: recursive definition of symbol "*symbol*"
シンボルの評価が、それ自身の値に依存しています。シンボル値を変更して、この循環定義を排除してください。

E 254: missing closing '>' in include directive
構文チェッカーが、INCLUDE擬似命令で閉じかっこ">"を見つけられません。閉じかっこ">"を追加してください。

E 255: could not open include file *include-file*
アセンブラが、このinclude-fileをオープンできませんでした。現在の検索パスをチェックして、このインクルードファイルが存在し読み込まれるようになっているかどうか調べてください。

E 256: integral divide by zero
式に、0による除算が含まれています。これは定義されません。式を変更して、0による除算がなくなるようにしてください。

E 257: unterminated string
すべての文字列は、開始した行と同じ行で終了しなければなりません。閉じ引用符があるかどうかチェックしてください。

E 258: unexpected characters after macro parameters, possible illegal white space
マクロパラメータの間にはスペースを入れることはできません。マクロ呼び出しの構文をチェックしてください。

E 259: COMMENT directive not permitted within a macro definition and conditional assembly
本アセンブラでは、MACRO/DUP定義またはIF/ELSE/ENDIF構成体でCOMMENT擬似命令を使用できません。問題のCOMMENTを、セミコロンで始まるコメントで置き換えてください。

E 260: definition of "*macro*" unterminated, missing "endm"
マクロ定義がENDM擬似命令で終わっていません。マクロ定義をチェックしてください。

E 261: macro argument name may not start with an '_'
MACRO引数とDUP引数の最初にアンダースコアを付けることはできません。問題のパラメータ名を変更し、アンダースコア以外の文字を付けてください。

E 262: cannot find "*symbol*"
"%"演算子または"?"演算子の引数の定義をマクロ展開内で見つけることができませんでした。問題のシンボルの定義をチェックしてください。

E 263: cannot evaluate: "*symbol*", value is unknown at this point
マクロ展開内で"%"演算子または"?"演算子が使用されているシンボルは、まだ定義されていません。問題の識別子の定義を挿入してください。

E 264: cannot evaluate: "*symbol*", value depends on an unknown symbol
"%"演算子または"?"演算子の引数を、マクロ展開内で評価することができません。問題のシンボルの定義をチェックしてください。

E 265: cannot evaluate argument of *dup* (unknown or location dependant symbols)
DUP擬似命令の引数が評価できませんでした。引数の式をチェックして、順方向の参照または未知のシンボルがないか調べてください。

E 266: *dup* argument must be integral
DUP擬似命令の引数が整数ではありません。式を変更して、整数値に評価されるようにしてください。

E 267: *dup* needs a parameter
DUP擬似命令の構文をチェックしてください。

E 268: ENDM without a corresponding MACRO or DUP definition
アセンブラが、対応するMACRO定義またはDUP定義がないENDM擬似命令を見つけました。MACRO定義およびDUP定義をチェックするか、この擬似命令を削除してください。

E 269: ELSE without a corresponding IF
アセンブラが、対応するIF擬似命令のないELSE擬似命令を見つけました。IF/ELSE/ENDIFのネストをチェックするか、この擬似命令を削除してください。

E 270: ENDIF without a corresponding IF
アセンブラが、対応するIF擬似命令のないENDIF擬似命令を見つけました。IF/ELSE/ENDIFのネストをチェックするか、この擬似命令を削除してください。

E 271: missing corresponding ENDIF
アセンブラが、対応するENDIF擬似命令のないIF擬似命令またはELSE擬似命令を見つけました。IF/ELSE/ENDIFのネストをチェックするか、この擬似命令を削除してください。

E 272: label not permitted with this directive
一部の擬似命令はラベルを受け付けません。ラベルをこの行の前または後に移動してください。

E 273: wrong number of arguments for *function*
この関数には、もっと多い引数またはもっと少ない引数を指定する必要があります。関数定義をチェックして、引数を追加または削除してください。

E 274: illegal argument for *function*
引数に不正な型があります。関数定義をチェックし、それに従って引数を変更してください。

E 275:　expression not properly aligned

E 276:　immediate value must be between *value* and *value*
命令の即値オペランドは、この範囲の値のみを受け付けます。"&"演算子を使用して値がこの範囲内におさまるようにするか、"#>"を使用して強制的にロング即値オペランドにします。

E 277:　address must be between $*address* and $*address*
アドレスオペランドが、この範囲内にありません。アドレス式を変更してください。

E 278:　operand must be an address
オペランドはアドレスでなければなりませんが、アドレス属性がありません。アドレス変更子を使用するか、アドレス式を変更します。

E 279:　address must be short

E 280:　address must be short
オペランドは、短い範囲のアドレスでなければなりません。式は、ロングアドレス、または未知の範囲のアドレスに評価されました。

E 281:　illegal option "*option*"
アセンブラが、未知またはスペルミスのあるコマンド行オプションを見つけました。このオプションは無視されます。

E 282:　"Symbols:" part not found in map file "*name*"
マップファイルが不完全な可能性があります。ロケータが正しくマップファイルを作成しているかどうかチェックしてください。

E 283:　"Sections:" part not found in map file "*name*"
マップファイルが不完全な可能性があります。ロケータが正しくマップファイルを作成しているかどうかチェックしてください。

E 284:　module "*name*" not found in map file "*name*"
マップファイルが不完全な可能性があります。ロケータが正しくマップファイルを作成しているかどうかチェックしてください。

E 285:　*file-kind* file will overwrite *file-kind* file
出力ファイルの1つが、コマンド行で指定したソースファイルや他の出力ファイルを上書きするとき、アセンブラは警告を出します。ソースファイルの名前を変更する、**-o**オプションを使用して出力ファイルの名前を変更する、**-err**オプションを削除してエラーファイルの生成を抑制する、などの手段を実行します。

E 286:　$CASE options must be given before any symbol definition
$CASEオプションは、シンボルが定義される前にのみ指定することができます。このオプションを、最初のソースファイルの開始点に移動します。

E 287:　symbolic debug error: *message*
アセンブラが、シンボリックデバッグ(SYMB)命令でエラーを見つけました。このSYMB命令がCコンパイラによって生成される場合、エラー報告書に記入してセイコーエプソン(株)に送付してください。対策として、このモジュールのシンボリックデバッグ情報をオフにすることができます(**-g**オプションを削除)。

E 288:　error in PAGE directive: *message*
PAGE擬似命令に指定された引数は、制限に従っていません。マニュアルでPAGE擬似命令の制限をチェックして、それに従って引数を変更ください。

E 290:　fail: *message*
FAIL擬似命令の出力。これはユーザ生成エラーです。ソースコードをチェックして、FAIL擬似命令が実行された理由を探してください。

E 291:　generated check: *message*
Cコンパイラとアセンブラとの間の整合性チェックです。このエラーメッセージは、("#pragma asm"構成体を使用して)ユーザが挿入したアセンブリにエラーがない限り出されることはありません。

E 293:　expression out of range
命令オペランドは、指定されたアドレス範囲内になければなりません。アドレス式をチェックして、変更してください。

E 294:　expression must be between *hexvalue* and *hexvalue*

E 295:　expression must be between *value* and *value*

E 296:　optimizer error: *message*
オプティマイザがエラーを見つけました。命令を変更するか、オプティマイザをオフにしてください。

E 297:　jump address must be a code address
ジャンプおよびジャンプサブルーチンには、コードメモリのターゲットアドレスを指定しなければなりません。アドレス展開をチェックするか、メモリ変更子を使用して、式をコードメモリに移します。

E 298:　size depends on location, cannot evaluate
一部の構成体(特にalign擬似命令)のサイズは、メモリアドレスに依存します。問題の構成体を変更してください。

致命的エラー(F)

致命的エラーが発生すると、アセンブラはすぐに停止します。致命的エラーは通常、ユーザエラーが原因になっています。

F 401:　memory allocation error
空きメモリに対する要求がシステムによって拒絶されました。すべてのメモリが使用されています。プログラムを小さな部分に分ける必要があります。

F 402:　duplicate input filename "*file*" and "*file*"
アセンブラでは、コマンド行で入力ファイル名を1つだけ指定する必要があります。ファイル名が2つ以上指定された場合、エラーになります。

F 403:　error opening *file-kind* file : "*file-name*"
アセンブラがこのファイルをオープンできませんでした。これがソースファイルの場合、コマンド行で指定したファイルが存在するかどうか、および、このファイルが読み取り可能になっているかどうかチェックしてください。このファイルが一時ファイルの場合、環境シンボルTMPDIRが正しく設定されているかどうかチェックしてください。

F 404:　protection error: *message*
プロテクションキーがないか、またはIBM互換PCではありません。

F 405:　I/O error
アセンブラが、出力をファイルに書き込むことができません。空きディスク容量が十分あるかどうかチェックしてください。

F 406:　parser stack overflow

F 407:　symbolic debug output error
シンボリックデバッグ情報が、オブジェクトファイルに不正に書き込まれています。エラー報告書に記入してセイコーエプソン(株)に送付してください。

F 408:　illegal operator precedence
演算子の優先順テーブルが壊れています。エラー報告書に記入してセイコーエプソン(株)に送付してください。

F 409:　Assembler internal error
アセンブラが、内部の不整合を見つけました。エラー報告書に記入してセイコーエプソン(株)に送付してください。

F 410:　Assembler internal error: duplicate mufom "*symbol*" during rename
アセンブラは、有効範囲でローカルなシンボルの名前をすべて固有のシンボル名に変更します。ここでは、アセンブラが、固有の名前を生成できませんでした。エラー報告書に記入してセイコーエプソン(株)に送付してください。

F 411: symbolic debug error: "*message*"
SYMB擬似命令の解析時にエラーが発生しました。このSYMB擬似命令がCコンパイラによって生成される場合、エラー報告書に記入してセイコーエプソン(株)に送付してください。

F 412: macro calls nested too deep (possible endless recursive call)
ネストするマクロ展開の数には制限があります。現在この制限は1000になっています。この制限が表示されたら、再帰定義をチェックするか、ソースを書き換えて簡単なものにします。

F 413: cannot evaluate "*function*"
すでに処理されているはずの関数呼び出しが見つかりました。対策として、問題の関数呼び出しを探し出し、ソースから削除してください。また、エラー報告書に記入してセイコーエプソン(株)に送付してください。

F 414: cannot recover from previous errors, stopped
前に発生したエラーにより、アセンブラの内部状態が破壊され、プログラムのアセンブルが停止しました。前に報告されたエラーを除去して再実行してください。

F 415: error opening temporary file
アセンブラは、デバッグ情報およびリストファイルを生成するとき、一時ファイルを使用します。これらの一時ファイルがオープンまたは作成できませんでした。環境シンボルTMPDIRが正しく設定されているかどうかチェックしてください。

F 416: internal error in optimizer
オプティマイザがデッドロックの状態を検出しました。最適化オプションを付けないでアセンブルしてください。また、エラー報告書に記入してセイコーエプソン(株)に送付してください。

Appendix C リンカのエラーメッセージ

リンカのエラーメッセージおよび警告メッセージは、文字、数字、情報テキストの順に表示されます。エラーの文字は、次のようなエラータイプを示しています。

W 警告
E エラー
F 致命的エラー
V 冗長メッセージ

警告(W)

W 100: Cannot create map file *filename*, turned off -M option
このファイルは作成できませんでした。

W 101: Illegal filename (*filename*) detected
不正な拡張子の付いたファイル名が検出されました。

W 102: Incomplete type specification, type index = T*hexnumber*
未知の型が参照されました。指定されていない構造体に対するポインタが定義されている場合に発生します。

W 103: Object name (*name*) differs from filename
オブジェクトファイルの内部名がファイル名と同じではありません。ファイル名が変更されている可能性があります。

W 104: '-o *filename*' option overwrites previous '-o *filename*'
2番目の**-o**オプションが見つかったため、最初の名前が破棄されます。

W 105: No object files found
起動文でファイルが指定されていません。

W 106: No search path for system libraries. Use -L or env "*variable*"
システムライブラリファイル(**-l**オプションで指定されたもの)に対して、環境変数またはオプション**-L**で定義された検索パスが指定されていなければなりません。

W 108: Illegal option: *option* (-H or -¥? for help)
不正なオプションが検出されました。

W 109: Type not completely specified for symbol <*symbol*> in *file*
現在のファイルまたはこのファイルに、不完全な型指定があります。未知の深さを持つ配列、未知のパラメータを持つ関数などがこれに当たります。

W 110: Compatible types, different definitions for symbol <*symbol*> in *file*
互換性のある型の間に名前の競合があります。メンバ名や構造体のタグ名の競合が考えられる他、同じサイズの基本型(int、long)に対して異なる型名が割り当てられている場合などが考えられます。基本型の競合がある場合、構造が移植性のないものになります。

W 111: Signed/unsigned conflict for symbol <*symbol*> in *file*
両方の型のサイズは正しいですが、一方の型が符号なしで、もう一方の型が符号付きになっています。

W 112: Type conflict for symbol <*symbol*> in *file*
実数型の競合があります。

W 113: Table of contents of *file* out of date, not searched. (Use ar ts <*name*>)
arライブラリに、最新でないシンボルテーブルがあります。"**ar ts**"を使用して、新しいテーブルを生成してください。

W 114: No table of contents in *file*, not searched. (Use ar ts <*name*>)
arライブラリにシンボルテーブルがありません。"**ar ts**"を使用してテーブルを生成してください。

W 115: Library *library* contains ucode which is not supported
ucodeは、このリンカではサポートされていません。

W 116: Not all modules are translated with the same threashold (-G value)
ライブラリファイルに未知のフォーマットがあるか、ファイルが壊れています。

W 117: No type found for <*symbol*>. No type check performed
このシンボルに対する型が生成されていません。

W 118: Variable <*name*>, has incompatible external addressing modes with file <*filename*>
変数はまだ割り当てられていませんが、2つの外部参照が、オーバーラップしないアドレッシングモードで作成されています。 これは常にエラーになります。

W 119: error from the Embedded Environment: *message*, switched off relaxed addressing mode check
リンカで組み込み環境が読み込み可能な場合、アドレッシングモードのチェックが解放されます。そのため、たとえば、データとして定義された変数が、HUGEとしてアクセスされる可能性があります。組み込み環境のエラーメッセージの概要については、Appendix Fの"組み込み環境のエラーメッセージ"を参照してください。

エラー(E)

E 200: Illegal object, assignment of non existing var *var*
MUFOM変数が存在しません。オブジェクトファイルが壊れています。

E 201: Bad magic number
指定されたライブラリファイルのマジックナンバーに問題があります。

E 202: Section *name* does not have the same attributes as already linked files
指定されたセクションに、異なる属性が指定されています。**-t***flag*を使用して、どのファイルがすでにリンクされているか確認します。すでにリンクされているファイルの場合、.outセクションを間違った属性で開始している可能性があります。

E 203: Cannot open *filename*
このファイルが見つかりません。

E 204: Illegal reference in address of *name*
変数の式で不正なMUFOM変数が使用されています。オブジェクトファイルが壊れています。

E 205: Symbol '*name*' already defined in <*name*>
シンボルが二度定義されています。このメッセージでは、関連するファイルを示しています。

E 206: Illegal object, multi assignment on *var*
前に発生したE 205の"already defined"エラーにより、MUFOM変数が、複数回割り当てられています。

E 207: Object for different processor characteristics
MAU単位のビット、アドレス単位のMAU、このオブジェクトのエンディアンなどが、最初にリンクしたオブジェクトと異なります。

E 208: Found unresolved external(s):
見つからないシンボルがあります。**-r**が設定されていない場合、エラーになります。

E 209: Object format in *file* not supported
オブジェクトファイルに未知のフォーマットがあります。またはオブジェクトファイルが壊れています。

E 210: Library format in *file* not supported
ライブラリファイルに未知のフォーマットがあります。またはライブラリファイルが壊れています。

E 211: Function <*function*> cannot be added to the already built overlay pool <*name*>
オーバーレイプールが前のリンカアクションで構築されています。このエラーを防止するために、オプション**-r**を使用してください。

E 212: Duplicate absolute section name <*name*>
絶対セクションが固定アドレスで始まっています。そのためリンクできません。

E 213: Section <*name*> does not have the same size as the already linked one
EQUAL属性を持つセクションが、他のリンク済みのセクションと同じサイズになっていません。

E 214: Missing section address for absolute section <*name*>
それぞれの絶対セクションには、オブジェクト内にセクションアドレスコマンドがなければなりません。オブジェクトファイルが壊れています。

E 215: Section <*name*> has a different address from the already linked one
2つの絶対セクションを同じ条件でリンク(重ね書き)することはできません。それぞれに同じアドレスがなければなりません。

E 216: Variable <*name*>, *name* <*name*> has incompatible external addressing modes
変数が、参照アドレッシング空間の外側に割り当てられています。たとえば、変数がゼロページに割り当てられていないのに、この変数がゼロページアドレッシングモードで参照されている場合などがこれに該当します。これは常にエラーになります。

E 217: Variable <*name*>, has incompatible external addressing modes with file <*filename*>
変数はまだ割り当てられていませんが、2つの外部参照が、オーバーラップしないアドレッシングモードで行われています。これは常にエラーになります。

E 218: Variable <*name*>, also referenced in <*name*> has an incompatible address format
アドレスは通常バイト単位で表現されます。ただし場合によっては、アドレスはビット単位で表現されます。ビット変数の場合その必要があります。現在のファイルとこのファイルとの間で、異なるアドレス形式のリンクが試みられました。

E 219: Not supported/illegal *feature* in object format *format*
このオブジェクトフォーマットでオプション/機能がサポートされていないか、または不正です。

E 220: page size (0x*hexvalue*) overflow for section <*name*> with size 0x*hexvalue*
セクションが大きすぎてページに収まりません。

E 221: *message*
オブジェクトが生成したエラーです。これらのエラーは実際にはアセンブラが生成したものです。範囲外のジャンプ命令によって引き起こされたものです。

E 222: Address of <*name*> not defined
変数にアドレスが割り当てられていません。オブジェクトファイルが壊れています。

致命的エラー(F)

F 400: Cannot create file *filename*
このファイルを作成できませんでした。

F 401: Illegal object: Unknown command at offset *offset*
オブジェクトファイルで未知のコマンドが検出されました。オブジェクトファイルが壊れています。

F 402: Illegal object: Corrupted hex number at offset *offset*
16進数のバイトカウントが不正です。オブジェクトファイルが壊れています。

F 403: Illegal section index
範囲外のセクションインデックスが検出されました。オブジェクトファイルが壊れています。

F 404: Illegal object: Unknown hex value at offset *offset*
オブジェクトファイルで未知の変数が検出されました。オブジェクトファイルが壊れています。

F 405: Internal error *number*
内部の致命的エラーです。渡された番号が詳細を示しています。

F 406: *message*
キーがないか、またはIBM互換PCではありません。

F 407:　Missing section size for section <*name*>
それぞれのセクションでは、オブジェクトにセクションサイズコマンドがなければなりません。オブジェクトファイルが壊れています。

F 408:　Out of memory
より多くのメモリを割り当てようとして失敗しました。

F 409:　Illegal object, offset *offset*
オブジェクトモジュールに不整合が見つかりました。

F 410:　Illegal object
オブジェクトモジュールの未知のオフセットに不整合が見つかりました。

F 413:　Only *name* object can be linked
他のプロセッサ用のオブジェクトは、リンクすることができません。

F 414:　Input file *file* same as output file
入力ファイルと出力ファイルは同じにすることができません。

F 415:　Demonstration package limits exceeded
このデモバージョンの制限を越えました。

冗長(V)

V 000:　Abort!
プログラムがユーザによってアボートされました。

V 001:　Extracting files
ライブラリからファイルを抽出することを示す冗長メッセージ。

V 002:　File currently in progress:
ファイルを現在処理していることを示す冗長メッセージ。

V 003:　Starting pass *number*
このパスの開始を示す冗長メッセージ。

V 004:　Rescanning...
ライブラリの再スキャンを示す冗長メッセージ。前回のスキャンで、解決されていない新しい外部参照があった場合、再スキャンされます。

V 005:　Removing file *file*
削除を示す冗長メッセージ。一時ファイルは常に削除され、マップファイルと.outファイルは、スイッチ**-e**がオンで終了コードが0以外の場合に削除されます。

V 006:　Object file *file* format *format*
指定されたオブジェクトファイルに標準ツールチェーンオブジェクトフォーマットTIOF-695がありません。

V 007:　Library *file* format *format*
指定されたライブラリに、標準ツールチェーン**ar88**フォーマットがありません。

V 008:　Embedded environment *name* read, relaxed addressing mode check enabled
組み込み環境が読み込まれました。

Appendix D ロケータのエラーメッセージ

ロケータのエラーメッセージおよび警告メッセージは、文字、数字、情報テキストの順に表示されます。エラーの文字は、次のようなエラータイプを示しています。

W 警告
E エラー
F 致命的エラー
V 冗長メッセージ

警告(W)

W 100: Maximum buffer size for *name* is *size* (Adjusted)
このフォーマットの場合、最大バッファサイズが定義されています。

W 101: Cannot create map file *filename*, turned off -M option
このファイルが作成できませんでした。

W 102: Only one -g switch allowed, ignored -g before *name*
デバッグできる.outファイルは1つだけです。

W 104: Found a negative length for section *name*, made it positive
負の値の長さをとることができるのは、スタックセクションのみです。

W 107: Inserted '*name*' keyword at line *line*
記述ファイルにないキーワードが挿入されました。

W 108: Object name (*name*) differs from filename
オブジェクトファイルの内部名がファイル名と同じではありません。ファイル名が変更されている可能性があります。

W 110: Redefinition of system start point
通常、システムテーブル(__lc_pm)にアクセスできるロードモジュールは1つだけです。

W 111: Two -o options, output name will be *name*
2番目の**-o**オプションが見つかりました。メッセージには、有効な名前が示されます。

W 112: Copy table not referenced, initial data is not copied
layout部分でcopy文を使用すると、初期データがROMにロケートされます。スタートアップコードはこのデータをRAMロケーションにコピーします。

W 113: No .out files found to locate
起動構文で指定されたファイルがありません。

W 114: Cannot find start label *label*
開始点が見つかりませんでした。

W 116: Redefinition of *name* at line *line*
識別子が二度定義されています。

W 119: File *filename* not found in the argument list
ロケートする必要があるすべてのファイルは引数として指定しなければなりません。

W 120: unrecognized name option <*name*> at line *line* (inserted '*name*')
オプションの割り当てが不正です。使用できるオプションについては、マニュアルを参照してください。

W 121: Ignored illegal sub-option '*name*' for *name*
不正なフォーマットサブオプションが検出されました。このフォーマットについては、マニュアルのフォーマットの説明を参照してください。

W 122: Illegal option: *option* (-H or -¥? for help)
不正なオプションが検出されました。

W 123:　Inserted *character* at line *line*
この文字が、記述ファイルにありませんでした。

W 124:　Attribute *attribute* at line *line* unknown
記述ファイルで、未知の属性が指定されています。

W 125:　Copy table not referenced, blank sections are not cleared
属性がブランクになっているセクションが検出されましたが、コピーテーブルは参照されていません。ロケータは、コピーテーブルにスタートアップモジュールについての情報を生成して、スタートアップ時にブランクセクションを消去します。マニュアルの"__lc_cp"の説明を参照してください。

W 127:　Layout *name* not found
指定されているファイルで使用するレイアウトが、layout部分で定義されていなければなりません。

W 130:　Physical block *name* assigned for the second time to a layout
ブロックをlayoutブロックに複数回割り当てることはできません。

W 136:　Removed *character* at line *line*
ここでは、この文字は必要ありません。

W 137:　Cluster *name* declared twice (layout part)
指定されたクラスタが二度宣言されています。layout部分でクラスタ名の重複が認められているのは、条件がある場合のみです。この場合、クラスタの参照のみが行われるためです。layout部分では、クラスタは一度だけ宣言できます。

W 138:　Absolute section *name* at non-existing memory address 0x*hexnumber*
絶対セクションのアドレスが物理メモリの外に指定されています。アドレスが不正であるか、ターゲットのメモリ記述に整合性がないかのいずれかです。

W 139:　*message*
組み込み環境からの警告メッセージです。組み込み環境のエラーメッセージの概要については、Appendix Fの"組み込み環境のエラーメッセージ"を参照してください。

W 140:　File *filename* not found as a parameter
ロケータ記述ファイル(software部分)で定義されたすべてのプロセスは、起動行で指定しなければなりません。

W 141:　Unknown space <*name*> in -S option
-Sオプションで未知の空間名が指定されました。

W 142:　No room for section *name* in read-only memory, trying writable memory ...
読み取り専用属性を持つセクションは、読み取り専用メモリに配置することができません。このセクションは、書き込み可能メモリに配置されます。

エラー(E)

E 200:　Absolute address 0x*hexnumber* occupied
絶対アドレスが要求されましたが、そのアドレスはすでに他のセクションによって占有されています。

E 201:　No physical memory available for section *name*
絶対アドレスが要求されましたが、そのアドレスには物理メモリがありません。

E 202:　Section *name* with mau size *size* cannot be located in an addressing mode with mau size *size*
ビットセクションは、バイト専用アドレッシングモードでロケートすることができません。

E 203:　Illegal object, assignment of non existing var *var*
MUFOM変数が存在しません。一部の変数では、これはエラーになります。

E 204:　Cannot duplicate section '*name*' due to hardware limitations
プロセスを複数回ロケートする必要がありますが、セクションが、メモリ管理機能のない仮想空間にマッピングされています。

E 205: Cannot find section for *name*
セクションのない変数が見つかりました。これは認められない状況です。

E 206: Size limit for the section group containing section *name* exceeded by 0x*hexnumber* bytes
小さなセクションが、これ以上ページに収まりません。

E 207: Cannot open *filename*
このファイルが見つかりません。

E 208: Cannot find a cluster for section *name*
書き込み可能なメモリがないか、アドレッシングモードが未知です。多くの場合、このエラーは、記述ファイルのエラーが原因になっています。

E 210: Unrecognized keyword <*name*> at line *line*
記述ファイルで未知のキーワードが使用されています。

E 211: Cannot find 0x*hexnumber* bytes for section *name* (fixed mapping)
仮想メモリまたは物理メモリのいずれかが占有されています。また物理メモリがまったくない可能性もあります。

E 213: The physical memory of *name* cannot be addressing in space *name*
マッピングが失敗しました。仮想アドレス空間が残っていません。

E 214: Cannot map section *name*, virtual memory address occupied
絶対マッピングが失敗しました。仮想ターゲットアドレスがすでに占有されていません。

E 215: Available space within *name* exceeded by *number* bytes for section *name*
アドレッシングモードで利用可能なアドレッシング空間がなくなりました。

E 217: No room for section *name* in cluster *name*
.dscファイルで定義されているクラスタのサイズが小さすぎます。

E 218: Missing *identifier* at line *line*
識別子を指定しなければなりません。

E 219: Missing ')' at line *line*
閉じかっこが足りません。

E 220: Symbol '*symbol*' already defined in <*name*>
シンボルが二度定義されています。

E 221: Illegal object, multi assignment on *var*
MUFOM変数が複数回割り当てられています。オブジェクト生成エラーが原因と考えられます。

E 223: No software description found
それぞれの入力ファイルは、.dscファイルのソフトウェア記述に記述しなければなりません。

E 224: Missing <length> keyword in block '*name*' at line *line*
ハードウェア記述に長さの定義がありません。

E 225: Missing <*keyword*> keyword in space '*name*' at line *line*
このマッピングに対して、キーワードを指定しなければなりません。

E 227: Missing <start> keyword in block '*name*' at line *line*
ハードウェア記述に開始の定義がありません。

E 230: Cannot locate section *name*, requested address occupied
絶対アドレスが要求されましたが、そのアドレスはすでに他のプロセスまたはセクションによって占有されています。

E 232: Found file *filename* not defined in the description file
ロケートするすべてのファイルには、それに対する定義レコードが記述ファイルに必要になります。

E 233: Environment variable too long in line *line*
記述ファイルにある環境変数の文字数が多すぎます。

E 235:　Unknown section size for section *name*
この.outファイルにセクションサイズが見つかりませんでした。実際には.outファイルが壊れています。

E 236:　Unrecoverable specification at line *line*
記述ファイルに回復不能なエラーがあります。

E 238:　Found unresolved external(s):
ロケート時に、すべての外部参照が解決されなければなりません。

E 239:　Absolute address *addr.addr* not found
この空間に、絶対アドレスが見つかりませんでした。

E 240:　Virtual memory space *name* not found
記述ファイルにある、このファイルのsoftware部分に、存在しないメモリ空間が記述されています。

E 241:　Object for different processor characteristics
MAU単位のビット、アドレス単位のMAU、このオブジェクトのエンディアンなどが、最初にリンクしたオブジェクトと異なります。

E 242:　*message*
オブジェクトが生成したエラーです。これらのエラーは実際にはアセンブラが生成したものです。範囲外のジャンプ命令によって引き起こされたものです。

E 244:　Missing *name* part
記述ファイルに、この部分が見つかりませんでした。前に発生したエラーが原因と考えられます。

E 245:　Illegal *name*value at line *line*
記述ファイルに、有効でない値が見つかりました。

E 246:　Identifier cannot be a number at line *line*
記述ファイルに、有効でない識別子が見つかりました。

E 247:　Incomplete type specification, type index = T*hexnumber*
このファイルにより、未知の型が参照されました。オブジェクトファイルが壊れています。

E 250:　Address conflict between block *block1* and *block2* (memory part)
記述ファイルのmemory部分にオーバーラップするアドレスがあります。

E 251:　Cannot find 0x*hexnumber* bytes for section *section* in block *block*
セクションをロケートしなければならない物理ブロックに空きがありません。

E 255:　Section '*name*' defined more than once at line *line*
1つのlayout/loadmod部分でセクションを複数回宣言することはできません。

E 258:　Cannot allocate reserved space for process *number*
空間の予約部分のメモリが占有されています。

E 261:　User assert: *message*
ユーザがプログラムした表明が失敗しました。これらの表明は、記述ファイルのlayout部分でプログラムすることができます。

E 262:　Label '*name*' defined more than once in the software part
記述ファイルで定義されているラベルは固有のものでなければなりません。

E 264:　*message*
組み込み環境からのエラーメッセージです。組み込み環境のエラーメッセージの概要については、Appendix Fの"組み込み環境のエラーメッセージ"を参照してください。

E 265:　Unknown section address for absolute section *name*
この.outファイルにセクションアドレスが見つかりませんでした。実際には.outファイルが壊れています。

E 266:　%s %s not (yet) supported
要求された機能は、このリリースでは(まだ)サポートされていません。

致命的エラー(F)

F 400:　Cannot create file *filename*
このファイルを作成できませんでした。

F 401:　Cannot open *filename*
このファイルが見つかりません。

F 402:　Illegal object: Unknown command at offset *offset*
オブジェクトファイルで未知のコマンドが検出されました。オブジェクトファイルが壊れています。

F 403:　Illegal filename (*name*) detected
不正な拡張子の付いたファイル名がコマンド行で検出されました。

F 404:　Illegal object: Corrupted hex number at offset *offset*
16進数のバイトカウントが不正です。オブジェクトファイルが壊れています。

F 405:　Illegal section index
範囲外のセクションインデックスが検出されました。オブジェクトファイルが壊れている可能性がありますが、E 231(セクションがない)のような前に発生したエラーも原因になっています。

F 406:　Illegal object: Unknown hex value at offset *offset*
オブジェクトファイルで未知の変数が検出されました。オブジェクトファイルが壊れています。

F 407:　No description file found
ロケータには、システムのハードウェアとソフトウェアを記述している記述ファイルが必要になります。

F 408:　*message*
プロテクションキーがないか、またはIBM互換PCではありません。

F 410:　Only one description file allowed
ロケータは、記述ファイルを1つだけ受け付けます。

F 411:　Out of memory
より多くのメモリを割り当てようとして失敗しました。

F 412:　Illegal object, offset *offset*
オブジェクトモジュールに不整合が見つかりました。

F 413:　Illegal object
オブジェクトモジュールの未知のオフセットに不整合が見つかりました。

F 415:　Only *name* .out files can be located
他のプロセッサ用のオブジェクトは、ロケートすることができません。

F 416:　Unrecoverable error at line *line*, *name*
記述ファイルのこの部分に回復不能なエラーがあります。

F 417:　Overlaying not yet done
この.outファイルでは、重ね書きがまだ実行されていません。最初に**-r**フラグを付けないでリンクしてください。

F 418:　No layout found, or layout not consistent
レイアウトに構文エラーがある場合、そのレイアウトがロケータで使用できない可能性があります。記述ファイルの構文エラーは解決されなければなりません。

F 419:　*message*
組み込み環境からの致命的エラーメッセージです。組み込み環境のエラーメッセージの概要については、Appendix Fの"組み込み環境のエラーメッセージ"を参照してください。

F 420:　Demonstration package limits exceeded
このデモバージョンの制限を越えました。

冗長(V)

V 000:　File currently in progress:
冗長メッセージ。次の行に、単一のファイル名が、処理される順にプリントされます。

V 001:　Output format: *name*
出力フォーマットの生成を示す冗長メッセージ。

V 002:　Starting pass *number*
このパスの開始を示す冗長メッセージ。

V 003:　Abort!
プログラムがユーザによってアボートされました。

V 004:　Warning level *number*
使用されている警告レベルを報告する冗長メッセージ。

V 005:　Removing file *file*
削除を示す冗長メッセージ。一時ファイルは常に削除され、マップファイルと.outファイルは、スイッチ**-e**がオンで終了コードが0以外の場合に削除されます。

V 006:　Found file <*filename*> via path *pathname*
記述(インクルード)ファイルが標準ディレクトリにありませんでした。ロケータは、インストールディレクトリも検索し、その結果ファイルが見つかりました。

V 007:　*message*
組み込み環境からの冗長メッセージです。組み込み環境のエラーメッセージの概要については、Appendix Fの"組み込み環境のエラーメッセージ"を参照してください。

Appendix E アーカイバのエラーメッセージ

この付録では、アーカイバ**ar88**が生成するすべての警告(W)、エラー(E)、致命的エラー(F)について説明します。

警告(W)

W 100: Illegal warning level: *level*
警告レベルは1桁の数字です。

W 101: Member *name* not found
ライブラリメンバが見つかりません。これは警告です。

W 102: Can't modify modification time for *name*
アーカイバが、ファイル*name*にアクセスして変更時刻を修正することができません。

W 103: creating archive *name*
アーカイブファイルが存在しないときに、**q**オプションが使用されました(**r**オプションの方が適切)。

W 104: Option -a or -b only allowed with key option 'r' or 'm'. Ignored!
置換アクションまたは移動アクションでは、オプション**a**または**b**(アーカイブの位置を指定)のみを適用できます。

W 105: Only one position specification allowed, ignored '-a or -b *file_offset*'
アーカイブでは複数の位置を指定することはできません。オプション**-a**および**-b**は、どちらも位置を指定するために使用できます。

W 106: Option -o only allowed with key option 'x'. Ignored!
ライブラリの日付は、ライブラリメンバの抽出の場合のみ保持できます。

W 107: Option -u only allowed with key option 'r'. Ignored!
アーカイブより新しいオブジェクトは、キーオプション**r**でのみ置換されます。

W 108: Option -z only allowed with key option 'r'. Ignored!
アーカイブに移動されるオブジェクトのみをチェックすることができます。

W 109: Option -v has no meaning with key option 'p' or 't'. Ignored!
オプション**p**および**t**の場合、冗長スイッチは意味を持ちません。

エラー(E)

E 200: filename too long
ファイル名が長すぎて、内部バッファに収まりません。

E 201: Member *name* not found
ライブラリメンバが見つかりません。

E 204: Can't obtain file-status information *filename*
ファイルステータス情報を取得するために*filename*にアクセスすることができません。

E 207: illegal option: *option*
不正なオプションが検出されました。

致命的エラー(F)

F 300: user abort
ライブラリマネージャがユーザによってアボートされています。

F 301: too much error
エラーの最大数を越えました。

F 302: protection error: *error*
ky_initからエラーメッセージが届きました。

F 303: can't create "*filename*"
この名前のファイルを作成することができません。

F 304: can't open "*filename*"
この名前のファイルをオープンすることができません。

F 305: can't reopen "*filename*"
ファイル*filename*は再オープンすることができませんでした。

F 306: read error while reading "*filename*"
このファイルを読み込むときに読み込みエラーが発生しました。

F 307: write error
このファイルを書き込むときに書き込みエラーが発生しました。このエラーは、DOSでも、**-p**を使用して(2進)出力を画面にプリントする場合に発生します。

F 308: out of memory
メモリを割り当てようとしましたが失敗しました。

F 309: illegal character
使用できない文字が見つかりました。

F 310: *filename* not in archive format
このアーカイブファイルは適切なフォーマットになっていません。

F 311: specification of more than one key {rxdmpt} is not permitted
複数のキーが指定されました。

F 312: no one of the keys {rxdmpt} was specified
キーが指定されていません。

F 313: error in the invocation. Use option -? or -H to get help.
使用法を表示してください。さらに詳細なヘルプが必要な場合は、オプション**-?**を使用してください。

F 314: *name* does not exist
ライブラリは、**r**キーオプションが指定された場合にのみ作成されます。

F 315: IEEE violation for object module *name* at address *address*
IEEE違反が検出されました(**z**オプションがオン)。

F 316: corrupted object module *name*
オブジェクトモジュール名が、IEEEオブジェクト仕様に準拠していません。

F 317: *name*: illegal byte count in hex number, offset = *offset*
16進数のバイトカウントが不正です(IEEE違反)。

F 318: evaluation date expired!!

Appendix F　組み込み環境のエラーメッセージ

組み込み環境からのエラーメッセージは、リンカやロケータのエラーメッセージの一部になっています。以下のエラー番号は、メッセージの一部にはなりません。

E　エラー
W　警告

エラー(E)

E 1:　Conflicting attributes *attributes* at line *number*
属性の競合があります。

E 2:　Unknown attribute '*character*' at line *number*
未知の属性があります。

E 3:　Unknown keyword '*name*' at line *number*
未知のキーワードがあります。

E 4:　Illegal character '*character*' at line *number*
不正な文字があります。

E 5:　Page size only allowed in a space definition at line *number*
ページサイズは空間定義でのみ可能です。

E 6:　Page size must be a power of 2 at line *number*
ページサイズは、2の累乗でなければなりません。

E 7:　Mau size must be a power of 2 at line *name*
MAUサイズは、2の累乗でなければなりません。

E 8:　Cannot synchronize any more line *number*
これ以上同調させることはできません。

E 9:　Illegal value '*value*' at line *number*
不正な値があります。

E 10:　Illegal hex value '*value*' at line *number*
不正な16進値があります。

E 11:　Illegal octal value '*value*' at line *number*
不正な8進値があります。

E 12:　Missing value at line *number*
足りない値があります。

E 13:　Illegal identifier at line *number*
不正な識別子があります。

E 14:　Wrong attribute '*attribute*' at line *number*
認められていない属性があります。

E 15:　Unknown identifier '*name*' at line *number*
未知の識別子があります。

E 16:　Inserted '*character*' at line *number*
挿入された文字があります。

E 17:　Cannot find bus/space '*name*' in definition for space '*name*'
空間からのマッピング先にエラーがあります。

E 18:　Cannot find space/amode '*name*' in definition for amode '*name*'
マップエラーです。

E 19:　Cannot find chip '*name*' in definition for bus '*name*'
マップエラーです。

E 20:　Cannot find space/amode '*name*' in layout definition for segment '*name*'
マップエラーです。

E 21:　Cannot find bus '*name*' in definition for mapping '*name*'
マップエラーです。

警告(W)

W 100:　Cannot find mapping '*name*' in segment definition for space '*name*'
セグメントマッピングの警告です。

Appendix G DELFEE

この付録では、DELFEE記述言語について説明します。

全体

description
    partition
    description partition

partition
    memory_partition
    cpu_partition
    software_partition

ident_list
    ident_list , identifier
    identifier

identifier
    STRING

file_name
    STRING

CPU

cpu_partition
    cpu { *static_specs_list* **}**
    cpu { }
    cpu *file_name*

Memory

memory_partition
    memory { *static_specs_list* **}**
    memory { }
    memory *file_name*

static_specs_list
    static_specs_list static_specs
    static_specs

static_specs
    amod_specs
    spce_specs
    bus_specs
    chips_specs

amod_specs
    amode *ident_list* **{** *amod_list* **}**

spce_specs
    space *ident_list* **{** *spce_list* **}**

bus_specs
    bus *ident_list* **{** *bus_list* **}**

chips_specs
    chips *ident_list chips_list* **;**

amod_list
    amod_list amod_def
    amod_def

spce_list
    spce_list spce_def
    spce_def

bus_list
    bus_list bus_def
    bus_def

chips_list
    chips_list chips_def
    chips_def

amod_def
    mau_spec
    attribute_spec
    map_spec

spce_def
    mau_spec
    map_spec

bus_def
    mau_spec
    mem_spec
    map_spec

chips_def
    mau_equ_spec
    attribute_equ_spec
    size_spec

mau_spec
    mau *NUMBER* **;**

mau_equ_spec
    mau = *NUMBER*

attribute_spec
    attribute *STRING* **;**
    attribute *NUMBER* **;**
    attr *STRING* **;**
    attr *NUMBER* **;**

attribute_equ_spec
    attribute = *STRING*
    attribute = *NUMBER*
    attr = *STRING*
    attr = *NUMBER*

map_spec
    map *map_list* **;**

map_list
    map_list map_def
    map_def

map_def
    src_spec
    size_spec
    dst_spec
    align_spec
    page_spec
    amode_spec
    space_spec
    bus_spec

mem_spec
    mem *mem_list* **;**

mem_list
    mem_list mem_def
    mem_def

mem_def
    addr_spec
    chips_spec

src_spec
    src = *NUMBER*

size_spec
    size = *NUMBER*

dst_spec
    dst = *NUMBER*

align_spec
    align = *NUMBER*

page_spec
    page = *NUMBER*

amode_spec
    amode = *identifier*

space_spec
    space = *identifier*

bus_spec
    bus = *identifier*

addr_spec
    address = *NUMBER*
    addr = *NUMBER*

chips_spec
    chips = *low_chip_list*

low_chip_list
    low_chip_list , low_chip_pair
    low_chip_pair

low_chip_pair
    low_chip_pair | low_chip
    low_chip

low_chip
    identifier

Software

software_partition
    software { *layout_blocks* **}**
    software { }
    software *file_name*

layout_blocks
    layout_blocks layout_block
    layout_block

layout_block
    layout
    loadmod

loadmod
    load_mod *software_specs* **;**
    load_mod *identifier software_specs* **;**

software_specs
    software_specs software_spec
    software_spec

software_spec
    start
    process

start
    start = *identifier* **;**

process
    process = *pids*

pids
    NUMBER
    pids , NUMBER

layout
    layout { *space_blocks* **}**
    layout { }
    layout *file_name*

space_blocks
    space_blocks space_block
    space_block

space_block
    space *identifier* **{** *block_blocks* **}**

block_blocks
    block_blocks block_block
    block_block

block_block
    block *identifier* **{** *cluster_blocks* **}**

cluster_blocks
    cluster_blocks cluster_block
    cluster_block

cluster_block
    cluster_spec
    p_gap_spec
    p_fixed_spec
    p_pool_spec
    p_skip_spec
    p_label_spec

cluster_spec
    cluster *identifier* **{** *amod_blocks* **}**
    cluster *ident_list* **;**

amode_blocks
    amode_blocks amode_block
    amode_block

amode_block
    amode *ident_list* **{** *section_blocks* **}**
    amode *ident_list* **;**
    section_block

p_gap_spec
    gap *length* **;**
    gap ;

p_fixed_spec
    fixed *address* **;**

p_pool_spec
    pool *length* **;**
    pool ;

p_label_spec
    label *identifier* **;**

p_skip_spec
    skip ;

attribute
    attribute_equ_spec

length
    length = *NUMBER*
    leng = *NUMBER*

address
    address = *NUMBER*
    addr = *NUMBER*

section_blocks
    section_blocks section_block
    section_block

section_block
    section_spec
    copy_spec
    v_fixed_spec
    v_gap_spec
    v_reserved_spec
    stack_spec
    heap_spec
    table_spec
    others
    v_label_spec
    v_assert_spec
    attribute_spec

section_spec
    section *selection modifiers* **;**
    section *selection* **;**

modifiers
    modifiers modifier
    modifier

modifier
    attribute
    address

copy_spec
    copy *selection attribute* **;**
    copy *selection* **;**
    copy ;

selection
    selection = *STRING*
    identifier

v_fixed_spec
    fixed *address* **;**

v_gap_spec
    gap ;

v_reserved_spec
    reserved *reserved_options* **;**
    reserved ;

reserved_options
    reserved_options reserved_option
    reserved_option

reserved_option
    attribute
    address
    length
    v_label_equ_spec

stack_spec
    stack *stack_options* **;**
    stack ;

heap_spec
heap *stack_options* **;**
heap ;

stack_options
stack_options stack_option
stack_option

stack_option
attribute
length

table_spec
table *attribute* **;**
table ;

v_label_spec
label *identifier* **;**

v_label_equ_spec
label = *identifier*

v_assert_spec
assert (*bool_expression , STRING* **) ;**
asse (*bool_expression , STRING* **) ;**

others
others ;

bool_expression
termp bool_op termp

termp
term + *termp*
term - *termp*
term

term
(*term* **)**
identifier
NUMBER

bool_op
<
>
==
!=

NUMBERは、一連の(16進)数値で、オプションで接尾辞"k"、"M"、"G"が付けられます(それぞれ"キロ"、"メガ"、"ギガ"を意味する)。この数値は16進数、8進数、10進数で指定することができ、通常の接頭辞で表現します。適切な値であれば、マイナス記号を数値の前に付けることもできます。

STRINGは、数値以外の一連の文字で(089は有効な8進値でないため**STRING**になる)、英数字で構成されます。またその他に、"_"、"."、"-"や、ディレクトリ区切り文字も使用することができます("\"、"/"、":")。

トークン(の一部)には環境変数を入れることができます。環境変数Aに、テキスト"foo"が設定されている場合、次のシーケンスは

```
$A/proto.dsc
```

次のように変換されます。

```
foo/proto.dsc
```

複数文字の変数の場合、かっこを付ける必要があります。

```
window = $(MODE) ;
```

DELFEEスクリプトにコメントを記述する方法は3つあります。1つ目の方法は、"C"のスタイルを使用して"/*"と"*/"でコメントを囲む方法です。2番目の方法は、最初のカラムに"#"を入れる方法です。この方法の場合、Cプリプロセッサの前処理も可能になります。ロケータは#*line*擬似命令および#*file*擬似命令を無視します。3番目の方法は、"C++"スタイルでコメントを記述する方法です。つまり、2つのスラッシュ"//"を行の任意の場所に入れ、それ以降の文字を行末までコメントにする方法です。

Appendix H　IEEE-695オブジェクトフォーマット

H.1　IEEE-695

IEEE-695標準では、MUFOM（Microprocessor Universal Format for Object Modules）について記述しています。これは、ターゲットに依存する、オブジェクトファイルの記録標準です。ただし、この標準では、この標準に従ってシンボリックデバッグ情報をコード化する方法については記述していません。シンボリックデバッグ情報はオブジェクトファイルの一部にすることもできます。デバッガは、オブジェクトファイルを読み込んで、シンボリックデバッグ情報を使用し、実行可能コードと最初の高レベル言語ソースファイルとの間の関係を認識します。IEEE-695標準では、デバッグ情報の表現について記述していないため、この領域については、この標準の実装の段階で、ベンダー固有およびマイクロプロセッサ固有の方法が提供されています。

H.2　コマンド言語の概念

ほとんどのオブジェクトフォーマットは、レコード指向になっています。つまり、ファイル内の決まった位置に、セクションの数を記述する1つまたは複数のセクションヘッダがあります。セクションヘッダには、開始アドレス、ファイルオフセットなどの情報が含まれます。セクションの内容は、いくつかのデータ部分に置かれており、ヘッダが読み込まれた後に限り処理できるようになっています。そのため、このようなオブジェクトを読み込むツールでは、このようなファイルを処理する方法が暗黙的に決まっています。つまり、ファイル内を検索して関連するレコードを取得するというのが、通常の方法になります。

一方MUFOM(IEEE-695)では、異なる方法を使用します。MUFOMは、デバッガ内でリンカ、ロケータ、オブジェクトリーダーを操作するためのコマンド言語として設計されています。

アセンブラまたはコンパイラは、そこに含まれるほとんどのデータがリロケータブルになっているオブジェクトモジュールを作成することができます。変換プロセスの次のフェーズでは、いくつかのオブジェクトモジュールをリンクして、1つの新しいオブジェクトモジュールを生成します。リロケータブルオブジェクトは、絶対アドレス値がまだわかっていない場所で再配置式を使用しています。ロケータ内の式評価機能が、この再配置式を絶対アドレス値に変換します。

最終的に、オブジェクトはメモリにロードできる状態になります。オブジェクトファイルは複数のプロセスで変換されるため、MUFOMでは、オブジェクトファイルを、この変換プロセスを操作するためのコマンドのシーケンスとして実装します。

これらのコマンドは、次の5つのプロセスのいずれかで作成、実行、コピーされます。これらのプロセスでは、MUFOMオブジェクトファイルを処理します。

1. 作成プロセス

   アセンブラまたはコンパイラによるオブジェクトファイルの作成。アセンブラまたはコンパイラは、アセンブリソーステキストまたは高レベル言語から生成されたコマンドを送ることによって、他のMUFOMプロセスに何をすべきか指令します。

2. リンクプロセス

   いくつかのオブジェクトモジュールを1つのモジュールにリンクし、X変数の名前をI変数に変更して新しいコマンドを生成することにより、外部参照を解決します(R変数の割り当て)。

3. リロケーションプロセス

   リロケーション。L変数を割り当てることにより、すべてのセクションに絶対アドレスを指定します。

4. 式評価プロセス

   上記の3つのMUFOMプロセスのいずれかで生成されたローダ式の評価。

5. ローダプロセス

   絶対メモリイメージのロード。

最後の4つのプロセスは、実際にはコマンドインタープリタになっています。つまりアセンブラが記述するオブジェクトファイルは、基本的に、リンカ用の大きな命令の集まりになっています。たとえばIEEE-695では、ファイル内の特定の位置で、セクションの内容をバイト単位のシーケンスとして記述する代わりに、ロードコマンドLRを定義します。このコマンドは、リンカに複数のバイトをロードするよう命令するものです。LRコマンドは、リロケートされるMAU(最小アドレス可能単位)の数を指定して、その後実際のデータを指定します。このデータは、絶対バイトの集まりにすることもできますし、リンカが評価する式にすることもできます。

リロケーション式を新しい式または絶対データに変換して、セクションを結合するのが、実際のリンクプロセスです。

上記のMUFOMプロセスのいくつかは、1つのツールで組み合わせることができます。たとえば、ロケータは上記のプロセス3とプロセス4を実行します。

H.3　表記の規則

この付録では、次の規則が使用されています。

| | "|"の間にある項目の場合、そのリストの中から1つを選択します。
"" | リテラル文字は""で囲まれています。
[]+ | オプション項目を1回以上繰り返します。
[]? | オプション項目を0回または1回繰り返します。
[]* | オプション項目を0回以上繰り返します。
::= | "次のように定義されている"と解釈します。

H.4　式

IEEE-695ファイルの式は、変数、演算子、絶対データの組み合わせになります。

変数名は、常に最初が16進数以外の文字(G...Z)になり、その後にオプションで16進数が続きます。最初の非16進文字は、変数のクラスを指定します。オブジェクトファイルを読み込むとき、次の変数が見つかります。

G-　プログラムの開始アドレス。このアドレスが割り当てられていない場合、デフォルトで低レベルシンボル**_start**のアドレスが使用されます。

I-　I変数は、オブジェクトモジュールでは、グローバルシンボルを表現します。
I変数には、他のモジュールでリンケージ編集のために使用できる式が割り当てられます。I変数の名前は常に文字"I"で始まり、その後に16進数が続きます。I変数はNIコマンドでのみ作成されます。

L-　セクションの開始アドレス。この変数は、絶対セクションでのみ使用されます。"L"の後に、16進数のセクションインデックスが続きます。L変数は、割り当て(AS)コマンドによって作成されますが、その時点でセクションインデックスがSTコマンドで定義されていなければなりません。

N-　内部シンボルの名前。この変数は、ローカルシンボルの値を割り当てるときに使用する他、高レベル言語デバッガが使用するため、またはリンク時のモジュール間の型チェックのために複合型を構築するときにも使用します。N変数は、NNコマンドで作成されます。

P-　セクション単位のプログラムポインタ。この変数には常に、ターゲットメモリロケーションの現在のアドレスが入れられます。P変数の後には、16進数のセクションインデックスが続きます。セクションインデックスは、STコマンド(セクションタイプコマンド)で定義されていなければなりません。この変数は、最初の割り当ての後に作成されます。

R-　Rタイプの変数は、特定のセクションが使用するリロケーション参照です。このセクションのアドレスに対するすべての参照値は、R変数との相対値になります。リンクは、新しい値をRに割り当てることによって実行されます。R変数は、文字"R"で始まり、その後に16進数のセクションインデックスが続きます。このセクションインデックスは、STコマンドで定義されていなければなりません。(割り当てられていない)Rのデフォルト値は0です。

S-　Sタイプの変数は、あるセクションのセクションサイズ(MAU単位)になります。セクションごとに1つのS変数があります。"S"の後には、セクションインデックスが続きます。S変数は、最初の割り当て時に作成されます。

W-　作業変数。このタイプの変数は、次のMUFOMコマンドで使用できる値を割り当てるときに使用することができます。このコマンドは、作業空間の値に、余計な意味を追加しないで保持するためのものです。作業変数は、文字"W"で始まり、その後に16進数のセクションインデックスが続きます。W変数は、最初の割り当て時に作成されます。

X-　Xタイプの変数は、外部参照を示します。X-変数には、値を割り当てることができません。X変数は、文字"X"で始まり、その後に16進数が続きます。

MUFOM言語では、次のデータタイプを使用して、式を構築します。

```
digit             ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
hex_letter        ::=  "A" | "B" | "C" | "D" | "E" | "F"
hex_digit         ::=  digit | hex_letter
hex_number        ::=  [ hex_digit ]+
nonhex_letter     ::=  "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" |
                       "X" | "Y" | "Z"
letter            ::=  hex_letter | nonhex_letter
alpha_num         ::=  letter | digit
identifier        ::=  letter [ alpha_num ]*
character         ::=  'value valid within chosen character set'
char_string_length ::=  hex_digit hex_digit
char_string       ::=  char_string_length [ character ]*
```

"char_string_length"で数値を指定する場合、その後に同じ数の文字を続ける必要があります。

式は、即値とMUFOM変数で構成されます。MUFOMは、2から4のプロセス(リンカとロケータ)を処理します。このプロセスには、式の値を解析して計算する式評価の過程も含まれます。変数の値が未知であるため、MUFOMプロセスが式の絶対アドレス値を計算できない場合、その式(修正付き)が出力ファイルにコピーされます。

式は、逆ポーランド表記法(演算子がオペランドの後に置かれる)で記述されます。

```
expression ::=  boolean_function | one_operand_function | two_operand_function | three_operand_function |
                four_operand_function | conditional_expr | hex_number | MUFOM_variable
```

H.4.1 オペランドが付かない関数

@F　偽関数

@T　真関数

```
boolean_function ::= "@F" | "@T"
```

偽関数と真関数は、ブール値の結果(偽または真)を生成します。これは論理式で使用することができます。これらの関数には、両方ともオペランドが付きません。

H.4.2 単項関数

単項関数の場合、関数の前にオペランドが1つ付きます。

```
one_operand_function ::= operand "," monop
operand ::= expression
monop ::= "@ABS" | "@NEG" | "@NOT" | "@ISDEF"
```

@ABS　整数オペランドの絶対値を返します。

@NEG　整数オペランドの負の値を返します。

@NOT　ブールオペランドの否定を返します。オペランドが整数の場合、1の補数を返します。

@ISDEF　式のすべての変数が定義されている場合、論理的な真の値を返し、その他の場合、偽の値を返します。

H.4.3 2項関数および演算子

2項関数および演算子の場合、関数または演算子の前に2つのオペランドが付きます。

```
two_operand_function ::= operand1 "," operand2 "," dyadop
operand1 ::= expression
operand2 ::= expression
dyadop   ::= "@AND" | "@MAX" | "@MIN" | "@MOD" | "@OR" | "@XOR" |
             "+" | "-" | "/" | "*" | "<" | ">" | "=" | "#"
```

@AND　両方のオペランドが論理値の場合、オペランドの論理AND演算の結果をブール値の真/偽で返します。両方のオペランドが論理値でない場合、ビット単位ANDが実行されます。

@MAX　両方のオペランドを算術的に比較し、最大の値を返します。

@MIN　両方のオペランドを算術的に比較し、最小の値を返します。

@MOD　operand1をoperand2で割ったときのモジュロの結果を返します。どちらかのオペランドが負の場合、またはoperand2が0の場合、結果は定義されません。

@OR　両方のオペランドが論理値の場合、オペランドの論理OR演算の結果をブール値の真/偽で返します。両方のオペランドが論理値でない場合、ビット単位ANDが実行されます。

+、-、*、/　これらは、加算、減算、乗算、除算を示す算術演算子です。結果は整数になります。除算の場合、operand2が0であれば、結果は定義されません。除算の結果は四捨五入されます。

<、>、=、#　これらは、"より小"、"より大"、"等しい"、"等しくない"の各論理関係式を示す演算子です。結果は、真または偽になります。

H.4.4 MUFOM変数

MUFOM変数の意味については、H.4節で説明しています。次の構文の規則はMUFOM変数に適用されます。

```
MUFOM_variable      ::= MUFOM_var |
                        MUFOM_var_num
                        MUFOM_var_optinum
MUFOM_var           ::= "G"
MUFOM_var_num       ::= "I" | "N" | "W" | "X"
                        hex_number
MUFOM_var_optinum ::= "L" | "P" | "R" | "S"
                        [ hex_number ]?
```

H.4.5 @INS演算子および@EXT演算子

@INS演算子は、ビット文字列を挿入します。

```
four_operand_function ::= operand1 "," operand2 "," operand3 "," operand4 "," @INS
```

operand2は、operand1に挿入されます。このときoperand3の位置からoperand4の位置まで挿入が行われます。

@EXT演算子は、ビット文字列を挿入します。

```
three_operand_function ::= operand1 "," operand2 "," operand3 "," @EXT
```

ビット文字列は、operand1から抽出されます。このときoperand2の位置からoperand3の位置まで抽出されます。

H.4.6 条件式

```
conditional_expr ::= err_expr | if_else_expr
err_expr         ::= value "," condition "," err_num "," "@ERR"
value            ::= expression
condition        ::= expression
err_num          ::= expression
if_else_expr     ::= condition "," "@IF" "," expression "," "@ELSE" "," expression "," "@END"
```

H.5　MUFOMコマンド

H.5.1 モジュールレベルのコマンド

モジュールレベルのコマンドには、モジュールを開始させるコマンド、モジュールを終了させるコマンド、モジュールの作成日時を設定するコマンド、アドレスの書式を指定するコマンドの4つがあります。

H.5.1.1 MBコマンド

MBコマンドは、モジュールに出てくる最初のコマンドです。モジュール名と一緒に、ターゲットマシンの構成とオプションコマンドを指定します。

MB_command ::= "MB" machine_identifier ["," module_name]? "."

例：　`MB S1C88`

H.5.1.2 MEコマンド

モジュール終了(ME)コマンドは、オブジェクトファイルの最後のコマンドになります。オブジェクトモジュールの最後を定義します。

ME_command ::= "ME."

H.5.1.3 DTコマンド

DTコマンドは、オブジェクトモジュールの作成日時を設定します。

DT_command ::= "DT" [digit]* "."

例：　`DT19930120120432`

日時の表示形式は、"YYYYMMDDHHMMSS"のようになります。

年が4桁、月が2桁、日が2桁、時間が2桁、分が2桁、秒が2桁です。

H.5.1.4 ADコマンド

ADコマンドは、ターゲット実行環境のアドレスの書式を指定します。

```
AD_command      ::=  "AD" bits_per_MAU [ ","  MAU_per_address [ "," order ]? ]?
MAU_per_address ::=  hex_number
bits_per_MAU    ::=  hex_number
order           ::=  "L" | "M"
```

MAUは、最小アドレス可能単位を略したものです。これは、ターゲットプロセッサに依存します。

L　最低アドレスの最下位バイト(リトルエンディアン)

M　最低アドレスの最上位バイト(ビッグエンディアン)

例：　`AD8,3,L.`　3バイトアドレス可能な8ビットプロセッサがリトルエンディアンモードで動作することを指定します。

H.5.2 コメントコマンドおよびチェックサムコマンド

コメント(CO)コマンドは、オブジェクトモジュールおよびそれを作成した翻訳プログラムについての情報をオブジェクトモジュールに格納するためのものです。コメントは、オブジェクトモジュールのソースファイルの名前や、それを作成した翻訳プログラムのバージョン番号などを記録するときに使用できます。この標準では、それぞれ独自のリビジョン番号を持つ複数の層をサポートしているため、オブジェクトモジュールには、モジュールの作成にどのリビジョンの標準が使用されているかを指定する複数のコメントコマンドを入れることができます。コメントの内容は標準では規定されていないため、MUFOMプロセスがコメントコマンドを処理する方法は、実装ごとに定義されます。

```
CO_command  ::=  "CO" [comment_level]? "," comment_text  "."
comment_level ::=  hex_number
comment_text  ::=  char_string
```

この標準では、コメントレベル0～6が、層のリビジョン番号に関する情報の受け渡しのために予約されています。チェックサム(CS)コマンドが開始すると、オブジェクトモジュールのチェックサム計算がチェックされます。

H.5.3 セクション

セクションは、コードまたはデータの最小単位で、個別にコントロールすることができます。それぞれのセクションには固有の番号が付いており、最初のセクション開始(SB)コマンドの部分で指定されています。セクションの内容は、その後に続けられます。セクションは、現在の番号と異なる番号を持つ次のSBコマンドの部分で終了します。セクションは、同じ番号を持つSBコマンドが指定された箇所から再開します。

H.5.3.1 SBコマンド

```
SB_command ::= "SB" hex_number "."
```

1つのオブジェクトモジュールに入れることのできるセクションの最大数は、実装ごとに定義されます。

H.5.3.2 STコマンド

STコマンドは、セクションのタイプを指定します。

```
ST_command  ::=  "ST" section_number [ "," section_type ]* [ "," sectoin_name ]? "."
section_type    ::=  letter
sectoin_name  ::=  char_string
```

セクション名は指定することもできますし、指定しないこともできます。sectoin_nameが省略されると、セクションは指定されません。セクションはリロケータブルまたは絶対になります。セクションの開始アドレスが絶対アドレスの場合、セクションは絶対セクションと呼ばれます。セクションの開始アドレスが未知の場合、セクションはリロケータブルセクションと呼ばれます。リロケータブルセクションでは、すべてのアドレスが、そのセクションのリロケーションベースとの相対値で指定されます。リンカまたはロケータのリロケーションフェーズでは、セクションのリロケーションベースが固定アドレスにマッピングされます。

リンケージ編集のとき、セクション名とセクション属性が、セクションを識別し、実行するアクションを識別します。セクションが複数のモジュールで定義されている場合、リンケージエディタが、同じ名前を持つセクションをどう処理するか決定しなければなりません。その場合、次のいずれかの方法がとられます。

- 複数のセクションをジョインして1つのセクションにします。
- 複数のセクションをオーバーラップさせます。
- セクションを共存させません。

セクションタイプは、リンケージエディタに対して、セクションに関する情報を提供します。この情報は、メモリ内でセクションをレイアウトするときに使用されます。セクションタイプの情報は、文字でコード化され、1つのSTコマンドに結合させることができます。文字の組み合わせによっては、不正なものや意味をなさないものもあります。

文字	意　味	クラス	説　明
A	絶対	アクセス	セクションに、対応するL-変数に割り当てられた絶対アドレスがあります。
R	読み取り専用	アクセス	このセクションに対する書き込みアクセス権がありません。
W	書き込み可能	アクセス	セクションの読み書きができます。
X	実行可能	アクセス	セクションに実行可能コードがあります。
Z	ゼロページ	アクセス	ターゲットにゼロページまたは短いアドレス可能ページがある場合、Z-セクションがそれにマッピングされます。
Y*num*	アドレッシングモード	アクセス	セクションをアドレッシングモード*num*でロケートする必要があります。
B	ブランク	アクセス	セクションを"0"で初期化する必要があります(クリア)。
F	未充てん	アクセス	セクションが充てんされておらずクリアもされていません(スクラッチ)。
I	初期化	アクセス	セクションをROMで初期化する必要があります。
E	等値	オーバーラップ	別々のモジュールにある2つのセクションの長さが異なる場合、エラーが発生します。
M	最大	オーバーラップ	最大値をセクションサイズとして使用します。
U	固有	オーバーラップ	セクション名は固有のものでなければなりません。
C	累積	オーバーラップ	セクションが複数のモジュールにある場合、それらのセクションを連結します。部分的なセクションの整列は保持する必要があります。
O	重ね書き	オーバーラップ	名前*name@func*を持つセクションを、呼び出しグラフから取得された*func*の規則に従って、セクション*name*に結合します。
S	分離	オーバーラップ	複数のセクションが同じ名前をとることができ、関連のないアドレスにリロケートできます。
N	今	時期	セクションが、(NもPもない)通常のセクションの前にロケートされます。
P	延期	時期	セクションが、(NもPもない)通常のセクションの後にロケートされます。

H.5.3.3 SAコマンド

```
SA_command    ::=  "SA" section_number "," [ MAU_boundary ]? [ "," page_size ]? "."
MAU_boundary ::=  expression
page_size          ::=  expression
```

MAU境界値は、リロケータに対して、セクションを、指定されたMAUの値で揃えるよう強制します。page_sizeが存在する場合、リロケート時にセクションがページ境界制限を越えていないか、リロケータがチェックします。

H.5.4 シンボル名の宣言とタイプ定義

H.5.4.1 NIコマンド

NIコマンドは、内部のシンボルを定義します。内部のシンボルはモジュールの外からでも参照できます。そのため、他のモジュールに、定義解除されている外部参照があってもそれを解決することができます。

```
NI_command ::= "N" I_variable "," char_string "."
```

NI_commandは、モジュール内で、I_variableに対する参照の前に置く必要があります。また同じ名前または数値を持つI_variableを複数使用することはできません。

H.5.4.2 NXコマンド

NXコマンドは、現在のモジュールで定義解除されている外部シンボルを定義します。NXコマンドは、対応するすべてのX変数の前に置く必要があります。

```
NX_command ::= "N" X_variable "," char_string "."
```

NXコマンドに対応する解決されていない参照は、他のモジュールで、内部シンボル定義(NI_command)によって解決することができます。

H.5.4.3 NNコマンド

NNコマンドは、モジュールのローカルシンボルの名前、または型定義の名前を定義するときに使用できるローカル名を定義します。

NNコマンドで定義した名前は、モジュールの有効範囲外からは参照できません。NNコマンドは、対応するすべてのN変数の前に置く必要があります。

```
NN_command ::= "N" N_variable "," char_string "."
```

H.5.4.4 ATコマンド

シンボルタイプ番号のような、シンボルのデバッグ関連情報を定義するとき、属性(AT)コマンドを使用することができます。標準のレベル2では、ATコマンドのオプションフィールドの内容について規定していません。言語依存の層(レベル3)では、ATコマンドでこれらのフィールドを使用して、高レベルシンボル情報を渡す方法について記述しています。

```
AT_command    ::= "AT" variable "," type_table_entry [ "," lex_level [ "," hex_number ]* ]? "."
variable      ::= I_variable | N_variable | X_variable
type_table_entry::= hex_number
lex_level     ::= hex_number
```

type_table_entryは、タイプ(TY)コマンドで使用されるタイプ番号が入ります。ATコマンドでタイプ番号を参照する場合、TYコマンドのタイプの定義の前に行います。

lex_levelフィールドの意味は、第3層以上で定義します。同じことは、オプションのhex_numberフィールドにも当てはまります。

H.5.4.5 TYコマンド

TYコマンドは、新しいタイプテーブルエントリを定義します。タイプコマンドで使用されるタイプ番号は、このタイプに対する参照インデックスと見なすことができます。TYコマンドは、新しく導入されたタイプと、オブジェクトモジュールの他の場所で定義されているその他のタイプの間の関係を定義します。このコマンドは、新しいタイプインデックスとシンボル(N_variable)との関係も確立します。

```
TY_command    ::= "TY" type_table_entry [ "," parameter ]+ "."
type_table_entry::= hex_number
parameter     ::= hex_number | N_variable | "T" type_table_entry
```

レベル2では、パラメータの意味を定義していません。これらはレベル3、言語層で定義されています。リンケージエディタがタイプコマンドのパラメータの意味を認識していない場合でも、タイプの比較を行うことができます。2つのタイプは、次の条件があてはまるとき等しいと見なされます。

- 2つのタイプに同じ数のパラメータがあるとき。
- タイプの数値が等しいとき。
- 2つのタイプのN_variableが同じ名前になっているとき。
- 2つのタイプから参照されるタイプエントリが等しいとき。

変数N0は、他の任意の名前と等しいと見なされます。

タイプテーブルエントリT0は、他の任意のタイプと等しいと見なされます。

H.5.5 値の割り当て

H.5.5.1 ASコマンド

割り当て(AS)コマンドは、値を変数に割り当てます。

```
AS_command ::= "AS" MUFOM_variable "," expression "."
```

H.5.6 ロードコマンド

セクションの内容は、絶対データ(コード)かリロケータブルデータ(コード)のいずれかになっています。絶対データはLDコマンドでロードすることができます。ロードが発生する場所は、そのセクションに属するP-変数の値によって決まります。LDコマンドの実行時に隣接しているデータは、メモリ内に連続してロードされるものと見なされます。

データが絶対でない場合、式評価機能によって評価されなければならない式が含まれています。LRコマンドを使用すると、リロケーション式をロードコマンドの一部として指定することができます。

H.5.6.1 LDコマンド

LD_command ::= "LD" [hex_digit]+ "."

LDコマンドでロードされる定数は、最初に最上位部分へロードされます。

H.5.6.2 IRコマンド

リロケーションベースは、リロケーション文字と関連付けられる式です。このリロケーション文字は、その後のロードリロケート(LR)コマンドで使用することができます。

IR_command　::= "IR" relocation_letter "," relocation_base ["," number_of_bits]? "."
relocation_letter ::= nonhex_letter
relocation_base ::= expression
number_of_bits ::= expression

例：
```
IRV,X20,16.
ITM,R2,40,+,8.
```

number_of_bitsは、アドレス単位のビット数以下でなければなりません。このアドレス単位のビット数は、アドレスあたりのMAUとMAUあたりのビット数との積で、この2つの値はADコマンドで指定されます。number_of_bitsが指定されていない場合、この値はアドレスあたりのビット数と同じになります。

H.5.6.3 LRコマンド

LR_command　::= "LR" [load_item]+ "."
load_item　::= relocation_letter offset "," | load_constant | "(" expression ["," number_of_MAUs]? ")"
load_constant　::= [hex_digit]+
number_of_MAUs ::= expression

例：
```
LR002000400060.
LRT80,0020.
LR(R2,100,+,4).
```

最初の例は、LRコマンドでロードすることができる即値定数を示しています。
2番目の例では、前の節で定義したリロケーションベースを使用しており、その後に定数を指定しています。
3番目の例は、式の値R2 + 100を使用して4 MAUをロードする方法を示しています。
この例の3つのコマンドは、1つのLRコマンドで次のように結合することもできます。

```
LR002000400060T80,0020(R2,100,+,4).
```

H.5.6.4 REコマンド

複製(RE)コマンドは、LRコマンドを繰り返す回数を定義します。

RE_command ::= "RE" expression "."

LRコマンドは、REコマンドの直後に使用します。

例：
```
RE04.
LR(R2,200,+,4).
```

上記のコマンドでは、R2 + 200の16 MAU(4×4 MAU)をロードします。

H.5.7 リンケージコマンド

H.5.7.1 RIコマンド

内部シンボル保持(RI)コマンドは、NIコマンドのシンボル情報が出力ファイルに保持されなければならないことを示します。

RI_command ::= "R" I_variable ["," level_number]? "."
level_number ::= hex_number

H.5.7.2 WXコマンド

弱い外部参照(WX)コマンドは、前に定義された外部参照(NX_command)にフラグを立て、これを弱い外部参照と見なします。この場合、外部参照が解決されないで残ったとき、WXコマンドの式の値がX変数に割り当てられることになります。

```
WX_command ::=  "W" X_variable [ "," default_value ]? "."
default_value    ::=  expression
```

H.5.7.3 LIコマンド

LIコマンドでは、デフォルトライブラリの検索リストを指定します。LI_commandで指定されたライブラリで、解決されていない参照がないか検索されます。

```
LI_command ::= "LI" char_string [ "," char_string ]* "."
```

H.5.7.4 LXコマンド

LXコマンドでは、指定された未解決変数を検索するライブラリを指定します。

```
LX_command ::= "L" X_variable [ "," char_string ]+ "."
```

これまでの節では、コマンドと演算子をASCII文字列で示しました。オブジェクトファイルでは、これらは2進数でコード化されています。以下の表では、2進表現で示しています。

H.6　MUFOM関数

次の表では、MUFOMエレメントの最初のバイトがリストされています。0から255までのそれぞれの値は、それ以降のMUFOM言語エレメントを分類するものであるか、言語エレメント自体になります。たとえば、0から127の範囲外の数値は、長さフィールドが前に置かれるため、0x82は2バイトの整数が続くことを指定します。0xE4は、LRコマンドの関数コードです。

MUFOM言語エレメントの最初のバイトの概要

0x00～0x7F	正規文字列の開始、または0～127の範囲の1バイトの数値
0x80	省略されたオプション数値フィールドを示すコード
0x81～0x88	0～127の範囲外の数値
0x89～0x8F	未使用
0x90～0xA0	ユーザ定義関数コード
0xA0～0xBF	MUFOM関数コード
0xC0	未使用
0xC1～0xDA	MUFOM文字
0xDB～0xDF	未使用
0xE0～0xF9	MUFOMコマンド
0xFA～0xFF	未使用

MUFOM文字と関数コードの2進表現

関数コード		識別子	
関数	コード	文字	コード
@F	0xA0		
@T	0xA1	A	0xC1
@ABS	0xA2	B	0xC2
@NEG	0xA3	C	0xC3
@NOT	0xA4	D	0xC4
+	0xA5	E	0xC5
-	0xA6	F	0xC6
/	0xA7	G	0xC7
*	0xA8	H	0xC8
@MAX	0xA9	I	0xC9
@MIN	0xAA	J	0xCA
@MOD	0xAB	K	0xCB
<	0xAC	L	0xCC
>	0xAD	M	0xCD
=	0xAE	N	0xCE
!= <>	0xAF	O	0xCF
@AND	0xB0	P	0xD0
@OR	0xB1	Q	0xD1
@XOR	0xB2	R	0xD2
@EXT	0xB3	S	0xD3
@INS	0xB4	T	OxD4
@ERR	0xB5	U	0xD5
@IF	0xB6	V	0xD6
@ELSE	0xB7	W	0xD7
@END	0xB8	X	0xD8
@ISDEF	0xB9	Y	0xD9
		Z	0xDA

MUFOMコマンドのコード

コマンド	コード	説　明
MB	0xE0	モジュールの開始
ME	0xE1	モジュールの終了
AS	0xE2	割り当て
IR	0xE3	リロケーションベースの初期化
LR	0xE4	リロケーションしてロード
SB	0xE5	セクションの開始
ST	0xE6	セクションのタイプ
SA	0xE7	セクションの調節
NI	0xE8	内部の名前
NX	0xE9	外部の名前
CO	0xEA	コメント
DT	0xEB	日時
AD	0xEC	アドレスの記述
LD	0xED	ロード
CS (合計値付き)	0xEE	合計値が後に続くチェックサム
CS	0xEF	チェックサム(合計値が0にリセット)
NN	0xF0	名前
AT	0xF1	属性
TY	0xF2	タイプ
RI	0xF3	内部シンボル保持
WX	0xF4	弱い外部参照
LI	0xF5	ライブラリの検索リスト
LX	0xF6	ライブラリの外部参照
RE	0xF7	複製
SC	0xF8	有効範囲の定義
LN	0xF9	行番号
	0xFA	未定義
	0xFB	未定義
	0xFC	未定義
	0xFD	未定義
	0xFE	未定義
	0xFF	未定義

Appendix I　Motorola Sレコード

ロケータは、S0、S2、S8の3つのタイプのSレコードを生成します。これらは、次のようなレイアウトになります。

S0レコード

'S' '0' <*length_byte*> <2 bytes 0> <*comment*> <*checksum_byte*>

ロケータが生成したSレコードファイルは、次の内容を持つS0レコードで始まります。

length_byte : 10H
comment　: E0C88 locator
checksum　: 88H

```
        E 0 C 8 8   l o c a t o r
S0100000534D433838206C6F6361746F7288
```

S0レコードは、コメントレコードであるため、プログラムの実行に関連する情報は含まれていません。

length_byteは、レコード内のバイト数を示しています。この場合、レコードのタイプと長さのバイトは含まれません。

チェックサムを計算するとき、最初に、レコードタイプの後にあるバイト(length_byteで始まる箇所)の2進表現をチェックサムの直前に追加します。次に、この合計値から1の補数が計算されます。この結果の最下位バイトがチェックサムになります。レコードタイプの後にあるすべてのバイトの合計は0FFHになります。

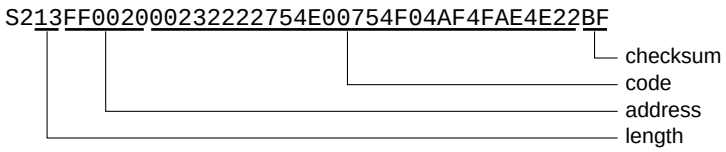
S2レコード

実際のプログラムコードおよびデータは、次のレイアウトでS2レコードに記録されます。

'S' '2' <*length_byte*> <*address*> <*code bytes*> <*checksum_byte*>

S1C88の場合、ロケータは、3バイトのアドレスを生成します。

例：



S2レコードを生成するときの出力バッファの長さは32コードバイトです。

S2レコードのチェックサムの計算方法は、S0の場合と同じです。

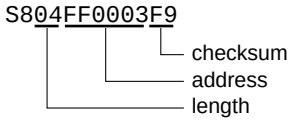
S8レコード

ロケータは、Sレコードファイルの終わりにS8レコードを生成します。これには、プログラムの開始アドレスが含まれます。

レイアウト：

'S' '8' <*length_byte*> <*address*> <*checksum_byte*>

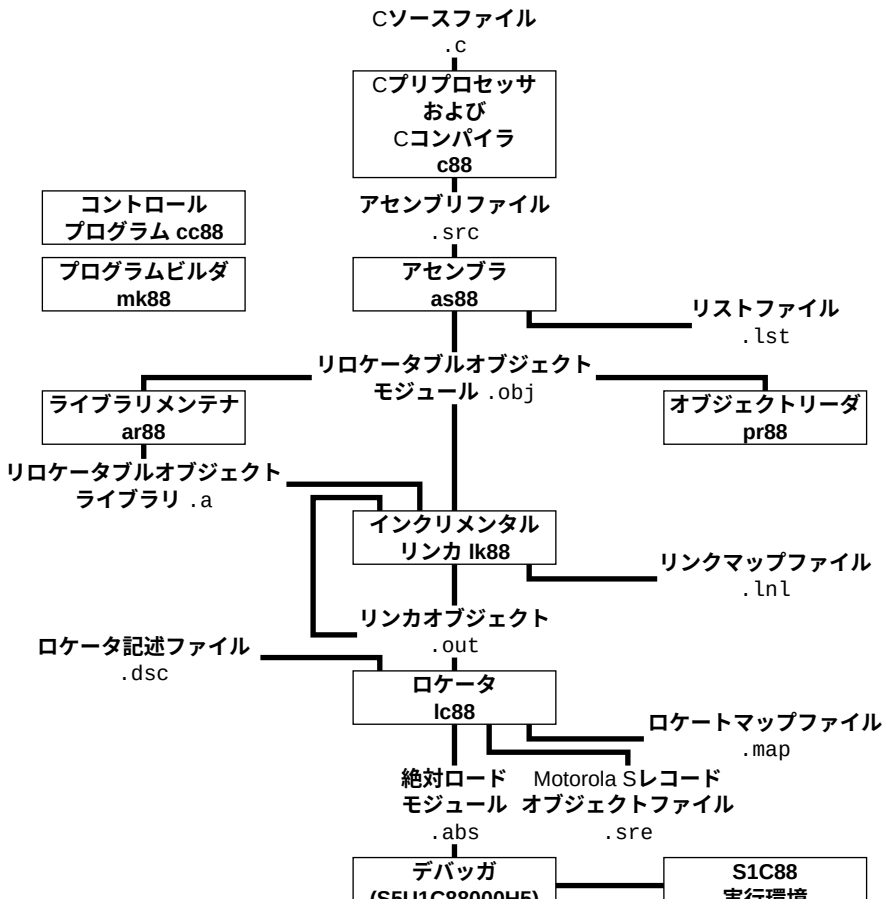
例：



S8レコードのチェックサムの計算方法は、S0の場合と同じです。

S1C88 Family C Compiler

Quick Reference


Cソースファイル
.c
Cプリプロセッサ
および
Cコンパイラ
c88
コントロール
プログラム cc88
アセンブリファイル
.src
プログラムビルダ
mk88
アセンブラ
as88
リストファイル
.lst
リロケータブルオブジェクト
モジュール .obj
ライブラリメンテナ
ar88
オブジェクトリーダ
pr88
リロケータブルオブジェクト
ライブラリ .a
インクリメンタル
リンカ lk88
リンクマップファイル
.lnl
リンカオブジェクト
.out
ロケータ記述ファイル
.dsc
ロケータ
lc88
ロケートマップファイル
.map
絶対ロード
モジュール
.abs
Motorola Sレコード
オブジェクトファイル
.sre
デバッガ
(S5U1C88000H5)
S1C88
実行環境

起動コマンド

c88 [[*option*]...[*file*]...]...

オプション

インクルードオプション

-f *file*	**オプションを***file***から読み込みます。**
-H *file*	**コンパイルの前に***file***をインクルードします。**
-I*directory*	***directory*でインクルードファイルを探します。**

前処理オプション

-D*macro*[=*def*]	**プリプロセッサ***macro***を定義します。**

コード生成オプション

-M{**s**|**c**|**d**|**l**}	**それぞれスモール、コンパクトコード、コンパクトデータ、ラージに対応します。**
-O{**0**|**1**}	**最適化を制御します。**

出力ファイルオプション

-e	**コンパイラエラーが発生した場合、出力ファイルを削除します。**
-o *file*	**出力ファイルの名前を***file***で指定します。**
-s	**Cソースコードをアセンブラ出力とマージします。**

診断オプション

-V	**バージョンヘッダのみを表示します。**
-err	**診断をエラーリストファイル(`.err`)に送信します。**
-g	**シンボリックデバッグ情報を有効にします。**
-w[*num*]	**1つまたはすべての警告メッセージを抑制します。**

エラー/ワーニングメッセージ

I: **情報**　E: **エラー**　F: **致命的なエラー**　S: **システムエラー**　W: **ワーニング**

フロントエンド

F 1:	evaluation expired	**評価版の有効期限が切れています。**
W 2:	unrecognized option: '*option*'	**このオプションが存在しません。**
E 4:	expected *number* more '#endif'	**コンパイラのプリプロセッサ部分に、"#if"、"#ifdef"、"#ifndef"指示文がありますが、同じソースファイルに、対応する"#endif"がありません。**
E 5:	no source modules	**コンパイルするソースファイルを少なくとも1つ指定する必要があります。**
F 6:	cannot create "*file*"	**出力ファイルまたは一時ファイルが作成できませんでした。**
F 7:	cannot open "*file*"	**このファイルが実際に存在するかどうかチェックしてください。**
F 8:	attempt to overwrite input file "*file*"	**出力ファイルの名前は、入力ファイルと異なる名前にする必要があります。**
E 9:	unterminated constant character or string	**このエラーは文字列を二重引用符(")で閉じなかった場合や文字定数を一重引用符(')で閉じなかった場合に発生します。**
F 11:	file stack overflow	**このエラーは、ファイルインクルードのネストの深さが最大数(50)を越えた場合に発生します。**
F 12:	memory allocation error	**すべての空きメモリ空間が使用されています。**
W 13:	prototype after forward call or old style declaration - ignored	**それぞれの関数のプロトタイプが実際の呼び出しの前にあることをチェックしてください。**
E 14:	';' inserted	**式の文にセミコロンが必要です。**
E 15:	missing filename after -o option	**-oオプションの後に、出力ファイル名を付ける必要があります。**
E 16:	bad numeric constant	**定数は、その構文に準拠する必要があります。また定数は、割り当てられた型を表す上で大きくなりすぎないようにします。**
E 17:	string too long	**このエラーは、最大文字列サイズ(1500)を越えた場合に発生します。**
E 18:	illegal character (0x*hexnumber*)	**16進ASCII値0x*hexnumber*の文字は、ここでは使用できません。**
E 19:	newline character in constant	**文字定数または文字列定数で改行を使用できるのは、円記号(\)またはバックスラッシュ(\)が前に付いている場合のみです。**
E 20:	empty character constant	**文字定数に入れる文字は1文字だけです。空の文字定数(')は使用できません。**
E 21:	character constant overflow	**文字定数に入れる文字は1文字だけです。エスケープシーケンスは1文字に変換されます。**
E 22:	'#define' without valid identifier	"**#define**"**の後には識別子を指定する必要があります。**

エラー/ワーニングメッセージ

フロントエンド

E 23:	'#else' without '#if'	"**#else**"は、対応する"**#if**"、"**#ifdef**"、"**#ifndef**"構文の中でのみ使用することができます。
E 24:	'#endif' without matching '#if'	"**#endif**"がありますが、それに対応するプリプロセッサ指示文"**#if**"、"**#ifdef**"、"**#ifndef**"がありません。
E 25:	missing or zero line number	"**#line**"では、0以外の数値を指定する必要があります。
E 26:	undefined control	コントロール行("*#identifier*"を含む行)には、既知のプリプロセッサ指示文が入っていなければなりません。
W 27:	unexpected text after control	"**#ifdef**"および"**#ifndef**"には、識別子が1つだけ必要です。また、"**#else**"および"**#endif**"には、改行以外の文字は必要ありません。"**#undef**"では、識別子が1つだけ必要です。
W 28:	empty program	ソースファイルには、外部定義が最低でも1つ必要になります。コメント以外何もないソースファイルは、空のプログラムと見なされます。
E 29:	bad '#include' syntax	"**#include**"の後には、有効なヘッダ名構文を続ける必要があります。
E 30:	include file "*file*" not found	"**#include**"指示文の後に、必ず既存のインクルードファイルを指定する必要があります。また、インクルードファイルには、正しいパスを指定していなければなりません。
E 31:	end-of-file encountered inside comment	コンパイラがコメントを読み込んでいるときに、エンドオブファイルが見つかりました。コメントが正しく終了していない可能性があります。
E 32:	argument mismatch for macro "*name*"	関数形式のマクロを起動する場合、引数の数は、定義のパラメータの数と一致させる必要があります。また、関数形式のマクロを起動するときは、")"を最後に付ける必要があります。
E 33:	"*name*" redefined	この識別子が、複数回定義されているか、後の宣言が前の宣言と異なっているかのいずれかです。
W 34:	illegal redefinition of macro "*name*"	マクロは、再定義されるマクロの本体が、最初に定義されたマクロの本体とまったく同じ場合に限り、再定義することができます。
E 35:	bad filename in '#line'	**#line**の文字列リテラルは、(ある場合)"**wide-char**"文字列でなければなりません。
W 36:	'debug' facility not installed	"**#pragma debug**"は、コンパイラのデバッグバージョンでのみ使用できます。
W 37:	attempt to divide by zero	0による除算または剰余算が見つかりました。
E 38:	non integral switch expression	`switch`条件式は、整数値を評価する必要があります。
F 39:	unknown error number: *number*	このエラーは、発生してはならないものです。
W 40:	non-standard escape sequence	不正なエスケープ文字が含まれています。

フロントエンド

E 41:	'#elif' without '#if'	"**#elif**"指示文が、"**#if**"、"**#ifdef**"、"**#ifndef**"構文内にありませんでした。
E 42:	syntax error, expecting parameter type/declaration/ statement	パラメータリスト、宣言、文のいずれかに、構文エラーがあります。
E 43:	unrecoverable syntax error, skipping to end of file	コンパイラが、回復不可能なエラーを検出しました。
I 44:	in initializer "*name*"	定数イニシャライザが正しいかどうかチェックするときの情報メッセージです。
E 46:	cannot hold that many operands	値スタックには、20を越えるオペランドを入れることができません。
E 47:	missing operator	式の中に演算子が必要です。
E 48:	missing right parenthesis	")"が必要です。
W 49:	attempt to divide by zero - potential run-time error	0による除算または剰余算を含む式が見つかりました。
E 50:	missing left parenthesis	"("が必要です。
E 51:	cannot hold that many operators	状態スタックには、20を越えるオペレーターを入れることができません。
E 52:	missing operand	オペランドが必要です。
E 53:	missing identifier after 'defined' operator	#if defined(*identifier*)には識別子が必要です。
E 54:	non scalar controlling expression	反復条件および"**if**"条件は、スカラ型になっている必要があります(struct、union、ポインタは不可)。
E 55:	operand has not integer type	"**#if**"指示文のオペランドは、**int**型定数を評価する必要があります。
W 56:	'<*debugoption*><*level*>' no associated action	このデバッグオプションおよびレベルには、対応するデバッグアクションがありません。
W 58:	invalid warning number: *number*	**-w**オプションで指定されたワーニング番号が存在しません。
F 59:	sorry, more than number errors	40以上のエラーがあるため、コンパイルが停止しました。
E 60:	label "*label*" multiple defined	同じ関数内では、1つのラベルを複数回定義することができません。
E 61:	type clash	型の競合が見つかりました。
E 62:	bad storage class for "*name*"	記憶クラス指定子`auto`および`register`は、外部定義の宣言指定子で使用することはできません。また、パラメータ宣言で使用できる記憶クラス指定子は、`register`のみです。
E 63:	"*name*" redeclared	この識別子は、すでに宣言されています。コンパイラは2番目の宣言を使用します。

エラー/ワーニングメッセージ

フロントエンド

E 64:	incompatible redeclaration of "*name*"	この識別子は、すでに宣言されています。
W 66:	function "*name*": variable "*name*" not used	使用されていない変数が宣言されています。
W 67:	illegal suboption: *option*	サブオプションがこのオプションで有効ではありません。
W 68:	function "*name*": parameter "*name*" not used	使用されていない関数パラメータが宣言されています。
E 69:	declaration contains more than one basic type specifier	型指定子を繰り返すことはできません。
E 70:	'break' outside loop or switch	`break`文は、`switch`またはループ(`do`、`for`、`while`)内でのみ使用できます。
E 71:	illegal type specified	指定した型は、このコンテキストでは使用できません。
W 72:	duplicate type modifier	指定子リストまたは修飾子リストで型修飾子を繰り返すことはできません。
E 73:	object cannot be bound to multiple memories	1つのオブジェクトではメモリ属性を1つだけ使用してください。
E 74:	declaration contains more than one class specifier	1つの宣言に入れることができる記憶クラス指定子は1つだけです。
E 75:	'continue' outside a loop	`continue`は、ループ本体(`do`、`for`、`while`)の中でのみ使用できます。
E 76:	duplicate macro parameter "*name*"	この識別子が、マクロ定義のformat1パラメータリストで複数回使用されています。
E 77:	parameter list should be empty	関数定義の一部でない識別子リストは、空でなければなりません。
E 78:	'void' should be the only parameter	引数をとらない関数の関数プロトタイプでは、voidが唯一のパラメータになります。
E 79:	constant expression expected	定数式には、カンマを入れることができません。また、ビットフィールド幅、enumを定義する式、配列にバインドされた定数、switch case式は、すべてint型定数式でなければなりません。
E 80:	'#' operator shall be followed by macro parameter	"#"演算子の後にはマクロ引数を続ける必要があります。
E 81:	'##' operator shall not occur at beginning or end of a macro	"##"(トークン連結)演算子は、隣接するプリプロセッサトークンを一緒にペーストするときに使用されます。そのためマクロ本体の最初や最後で使用することはできません。
W 86:	escape character truncated to 8 bit value	16進エスケープシーケンス("\"または"\"の後に"x"と数値)の値は、8ビット記憶域に収まる必要があります。
E 87:	concatenated string too long	生成された文字列が、1500文字の制限を越えています。
W 88:	"*name*" redeclared with different linkage	この識別子は、すでに宣言されています。

フロントエンド

E 89:	illegal bitfield declarator	ビットフィールドは、int型としてのみ宣言することができ、ポインタや関数として宣言することはできません。
E 90:	#error *message*	*message*は、"**#error**"プリプロセッサ指示文で指定する説明テキストです。
W 91:	no prototype for function "*name*"	それぞれの関数には、正しい関数プロトタイプが必要です。
W 92:	no prototype for indirect function call	それぞれの関数には、正しい関数プロトタイプが必要です。
I 94:	hiding earlier one	エラーE 63の後に追加されるメッセージです。2番目の宣言が使用されます。
F 95:	protection error: *message*	プロテクションキーの初期化時に問題が発生しました。
E 96:	syntax error in #define	`#define id(`には、右のかっこ")"が必要です。
E 97:	"..." incompatible with old-style prototype	2つの関数が同じ名前のパラメータ型リストを持っている場合、これは古いスタイルの宣言であるため、パラメータリストに省略記号を入れることはできません。
E 98:	function type cannot be inherited from a typedef	typedefは関数定義で使用することはできません。
F 99:	conditional directives nested too deep	"**#if**"、"**#ifdef**"、"**#ifndef**"指示文は、50レベルより深くネストすることができません。
E 100:	case or default label not inside switch	`case:`ラベルまたは`default:`ラベルは、`switch`内でのみ使用することができます。
E 101:	vacuous declaration	宣言の中に足りないものがあります。
E 102:	duplicate case or default label	`switch case`では、評価の後値がそれぞれ固有でなければならず、`switch`内に`default:`ラベルが少なくとも1つ必要になります。
E 103:	may not subtract pointer from scalar	ポインタの減算で使用できるオペランドは、ポインタ - ポインタ、またはポインタ - スカラのみです。
E 104:	left operand of operator has not struct/union type	"."または"->"の最初のオペランドは、`struct`型または`union`型にならなければなりません。
E 105:	zero or negative array size - ignored	配列にバインドされる定数は、0より大きくなければなりません。
E 106:	different construction	パラメータ型リストを持つ互換関数型は、パラメータ数と省略記号の使用の点で共通でなければなりません。また対応するパラメータは互換性のある型でなければなりません。
E 107:	different array sizes	互換関数型のそれぞれの配列パラメータは同じサイズでなければなりません。
E 108:	different types	互換関数型のそれぞれのパラメータ、およびそれぞれのプロトタイプパラメータの型は、公開されている定義パラメータを持つ互換性のある型でなければなりません。

エラー/ワーニングメッセージ

フロントエンド

E 109:	floating point constant out of valid range	浮動小数点定数には、割り当てられている型に収まる値がなければなりません。
E 110:	function cannot return arrays or functions	関数では、配列型または関数型の戻り型を使用できません。関数に対するポインタは使用できます。
I 111:	parameter list does not match earlier prototype	パラメータリストをチェックするか、プロトタイプを調整してください。パラメータの数と型は一致する必要があります。
E 112:	parameter declaration must include identifier	宣言がプロトタイプの場合、それぞれのパラメータの宣言に識別子がなければなりません。また、typedef名として宣言された識別子はパラメータ名として使用できません。
E 114:	incomplete struct/union type	structまたはunionを使用する前に、その型を知らせる必要があります。
E 115:	label "*name*" undefined	goto文が見つかりましたが、同じ関数またはモジュール内にこのラベルがありませんでした。
W 116:	label "*name*" not referenced	このラベルが定義されていましたが、参照されませんでした。ラベルの参照は、同じ関数内またはモジュール内になければなりません。
E 117:	"*name*" undefined	この識別子は定義されていませんでした。変数の型は、使用する前に宣言で指定する必要があります。
W 118:	constant expression out of valid range	caseラベルで使用されている定数式が長すぎる可能性があります。また、浮動小数点値をint型に変換するとき、浮動小数点定数が長すぎることがあります。
E 119:	cannot take 'sizeof' bitfield or void type	ビットフィールドまたはvoid型のサイズが知らされていません。そのためそのサイズを使用することができません。
E 120:	cannot take 'sizeof' function	関数のサイズが知らされていません。そのためそのサイズを使用することができません。
E 121:	not a function declarator	これは有効な関数ではありません。
E 122:	unnamed formal parameter	パラメータには、正しい名前を指定する必要があります。
W 123:	function should return something	非void関数の戻り値には式がなければなりません。
E 124:	array cannot hold functions	関数の配列は使用できません。
E 125:	function cannot return anything	式を持つreturnが、void関数にありません。
W 126:	missing return (function "*name*")	空でない関数を持つ非void関数には、return文が必要になります。
E 129:	cannot initialize "*name*"	宣言子リスト内の宣言子では、初期化文を入れないでください。またextern宣言では、イニシャライザを使用できません。
W 130:	operands of operator are pointers to different types	演算子または割り当て("=")のポインタオペランドは同じ型でなければなりません。

フロントエンド

E 131:	bad operand type(s) of *operator*	この演算子には、他の型のオペランドが必要です。
W 132:	value of variable "*name*" is undefined	変数が定義される前に使用されている場合、このワーニングが発生します。
E 133:	illegal struct/union member type	関数は、structまたはunionのメンバにすることができません。また、ビットフィールドはint型またはunsigned型のみをとることができます。
E 134:	bitfield size out of range - set to 1	ビットフィールドの幅は、この型のビット数より多くしたり、負の値にしたりすることはできません。
W 135:	statement not reached	指定された文が、実行されません。
E 138:	illegal function call	関数以外のオブジェクトで関数呼び出しを実行することはできません。
E 139:	*operator* cannot have aggregate type	(キャスト)の型名と(キャスト)のオペランドは、スカラでなければなりません(struct、union、ポインタは不可)。
E 140:	*type* cannot be applied to a register/bit/bitfield object or builtin/inline function	たとえば"&"演算子(アドレス)は、レジスタやビットフィールドでは使用できません。
E 141:	*operator* requires modifiable lvalue	"++"演算子や"--"演算子のオペランド、および割り当てや複合割り当ての左の演算子(lvalue)は、変更可能でなければなりません。
E 143:	too many initializers	イニシャライザの数をオブジェクトの数より多くすることはできません。
W 144:	enumerator "*name*" value out of range	enum定数がintの制限を越えています。
E 145:	requires enclosing curly braces	複合イニシャライザは中かっこで閉じる必要があります。
E 146:	argument #*number*: memory spaces do not match	プロトタイプでは、引数のメモリ空間が一致する必要があります。
W 147:	argument #*number*: different levels of indirection	プロトタイプでは、引数と割り当ての型に互換性がなければなりません。
W 148:	argument #*number*: struct/union type does not match	プロトタイプでは、プロトタイプ化した関数の引数と実際の引数の両方がstructまたはunionでしたが、異なるタグが付いています。タグの型は一致しなければなりません。
E 149:	object "*name*" has zero size	structまたはunionに、不完全な型のメンバがあります。
W 150:	argument #*number*: pointers to different types	プロトタイプでは、引数のポインタ型に互換性がなければなりません。
W 151:	ignoring memory specifier	struct、union、enumのメモリ指定子が無視されます。
E 152:	operands of *operator* are not pointing to the same memory space	オペランドが同じメモリ空間をポイントしていることを確認してください。

エラー/ワーニングメッセージ

フロントエンド

E 153:	'sizeof' zero sized object	暗黙的または明示的な`sizeof`演算で、未知のサイズのオブジェクトが参照されています。
E 154:	argument #*number*: struct/union mismatch	プロトタイプでは、プロトタイプ化した関数の引数と実際の引数のうち、いずれかだけがstructまたはunionでした。型は一致しなければなりません。
E 155:	casting lvalue '*type*' to '*type*' is not allowed	"++"演算子や"--"演算子のオペランド、または割り当てや複合割り当ての左の演算子(lvalue)は、別の型にキャストすることができません。
E 157:	"*name*" is not a formal parameter	宣言子に識別子リストがある場合、宣言子リストにはその識別子のみを入れることができます。
E 158:	right side of *operator* is not a member of the designated struct/union	"."または"->"の2番目のオペランドは、指定されたstructまたはunionのメンバでなければなりません。
E 160:	pointer mismatch at *operator*	*operator*の両方のオペランドが、有効なポインタでなければなりません。
E 161:	aggregates around *operator* do not match	*operator*の両側にある構造体、共用体、配列の内容は同じでなければなりません。
E 162:	*operator* requires an lvalue or function designator	"&"演算子(アドレス)には、lvalueや関数指名子が必要になります。
W 163:	operands of *operator* have different level of indirection	演算子のポインタまたはアドレスの型は、割り当てと互換性がなければなりません。
E 164:	operands of *operator* may not have type 'pointer to void'	*operator*のオペランドにオペランド(`void *`)がありません。
W 165:	operands of *operator* are incompatible: pointer vs. pointer to array	ポインタの型または演算子のアドレスは、割り当てと互換性がなければなりません。ポインタは、配列へのポインタに割り当てることができません。
E 166:	*operator* cannot make something out of nothing	型`void`を他の型にキャストすることはできません。
E 170:	recursive expansion of inline function "*name*"	_inline関数は再帰的に使用することができません。
E 171:	too much tail-recursion in inline function "*name*"	関数レベルが40以上の場合、このエラーが現れます。
W 172:	adjacent string have different types	2つの文字列を連結するとき、両方の文字列が同じ型でなければなりません。
E 173:	'void' function argument	関数は、void型の引数をとることができません。
E 174:	not address constant	定数アドレスが使用される予定になっていました。スタティック変数と異なり、auto変数には、固定されたメモリロケーションがないため、auto変数のアドレスは定数になりません。
E 175:	not an arithmetic constant	定数式では、割り当て演算子、"++"演算子、"--"演算子、関数を使用できません。

フロントエンド

E 176:	address of automatic is not a constant	スタティック変数と異なり、auto変数には、固定されたメモリロケーションがないため、auto変数のアドレスは定数になりません。
W 177:	static variable "*name*" not used	使用されていないスタティック変数が宣言されています。
W 178:	static function "*name*" not used	呼び出されていない静的関数が宣言されています。
E 179:	inline function "*name*" is not defined	インライン関数のプロトタイプのみがあり、実際のインライン関数が存在しないことが原因です。
E 180:	illegal target memory (*memory*) for pointer	ポインタが*memory*をポイントしていません。
W 182:	argument #*number*: different types	プロトタイプで、引数のタイプに互換性がなければなりません。
I 185:	(prototype synthesized at line *number* in "*name*")	古いスタイルのプロトタイプが含まれているソースファイルの位置を通知する情報メッセージです。
E 186:	array of type bit is not allowed	配列には、ビット型変数を入れることができません。
E 187:	illegal structure definition	構造体は、メンバが知らされている場合に限り、定義(初期化)することができます。
E 188:	structure containing bit-type field is forced into bitaddressable area	このエラーは、ビット型メンバを含む構造体でビットアドレス可能な記憶タイプを使用するときに発生します。
E 189:	pointer is forced to bitaddressable, pointer to bitaddressable is illegal	ビットアドレス可能なメモリを示すポインタは使用できません。
W 190:	"long float" changed to "float"	ANSI Cの浮動小数点定数は、定数に接尾辞"f"が付いている場合を除き、`double`型を持つものとして扱われます。
E 191:	recursive struct/union definition	`struct`または`union`には、それ自体を入れることができません。
E 192:	missing filename after -f option	-fオプションにはファイル名引数を付ける必要があります。
E 194:	cannot initialize typedef	typedef変数に値を割り当てることはできません。
F 199:	demonstaration package limits exceeded	デモパッケージには製品版にはない一定の制限があります。
W 200:	unknown pragma - ignored	コンパイラは、未知のプラグマについては無視します。
W 201:	"*name*" cannot have storage type - ignored	register変数またはオートマチック/パラメータには、記憶タイプを使用することができません。
E 202:	"*name*" is declared with 'void' parameter list	関数が何も受け付けない場合(voidパラメータリスト)、関数に引数を付けて呼び出すことはできません。
E 203:	**too many/few actual parameters**	プロトタイプでは、関数の引数の数が、その関数のプロトタイプと一致している必要があります。

エラー/ワーニングメッセージ

フロントエンド

W 204:	U suffix not allowed on floating constant - ignored	浮動小数点定数には、"U"接尾辞または"u"接尾辞を付けることができません。
W 205:	F suffix not allowed on integer constant - ignored	int型定数には、"F"接尾辞または"f"接尾辞を付けることができません。
E 206:	'*name*' named bit-field cannot have 0 width	ビットフィールドは、0より大きい値を持つint型定数式でなければなりません。
E 212:	"*name*": missing static function definition	staticプロトタイプを持つ関数にその定義がありません。
W 303:	variable '*name*' possibly uninitialized	関数が何かを返すことになっているにもかかわらず、初期化文が、到達しない箇所にあります。
E 327:	too many arguments to pass in registers for _asmfunc '*name*'	`_asmfunc`関数は、Cとアセンブラの間で固定レジスタベースのインタフェースを使用しますが、このときに受け渡しする引数の数は、使用可能なレジスタの数によって制限を受けます。関数*name*でこの制限を越えてしまいました。

バックエンド

W 501:	function qualifier used on non-function	関数修飾子は、関数でのみ使用できます。
E 502:	Intrinsic function '_int()' needs an immediate value as parameter	_int()組み込み関数の引数は、任意のタイプの整数式ではなく、整数定数式でなければなりません。
E 503:	Intrinsic function '_jrsf()' needs an immediate value 0..3	指定する値は0から3までの定数値でなければなりません。
W 508:	function qualifier duplicated	使用できる関数修飾子は1つだけです。
E 511:	interrupt function must have void result and void parameter list	`_interrupt(`*n*`)`で宣言された関数は、引数を受け付けないため、何も返すことができません。
W 512:	'*number*' illegal interrupt number (0, or 3 to 251) - ignored	割り込みベクタ番号は、0、または3から251の範囲内になければなりません。他の数値の場合は不正になります。
E 513:	calling an interrupt routine, use '_swi()'	割り込み関数は、直接呼び出すことができません。組み込み関数_swi()を使用しなければなりません。
E 514:	conflict in '_interrupt'/ '_asmfunc' attribute	現在の関数修飾子宣言および前回の関数修飾子宣言の属性が同じになっていません。
E 515:	different '_interrupt' number	現在の関数修飾子宣言および前回の関数修飾子宣言の番号が同じになっていません。
E 516:	'*memory_type*' is illegal memory for function	記憶タイプがこの関数で有効ではありません。
E 517:	conversion of long address to short address	このエラーは、ポインタ変換が必要な場合に出されます。

バックエンド

F 524:	illegal memory model	コンパイラの使用法を参照して、-Mオプションの正しい引数を調べてください。
E 526:	function qualifier '_asmfunc' not allowed in function definition	`_asmfunc`は、関数プロトタイプでのみ使用できます。
E 528:	_at( ) requires a numerical address	数値アドレスに評価される式のみを使用することができます。
E 529:	_at( ) address out of range for this type of object	指定されたメモリ空間に、絶対アドレスがありません。
E 530:	_at( ) only valid for global variables	絶対アドレスには、グローバル変数のみを置くことができます。
E 531:	_at( ) only allowed for uninitialized variables	絶対変数は初期化することができません。
E 532:	_at( ) has no effect on external declaration	`extern`と宣言した場合、その変数はコンパイラによって割り当てられません。
W 533:	c88 language extension keyword used as identifier	言語拡張機能キーワードは予約語になっており、予約語は識別子として使用することができません。
E 536:	illegal syntax used for default section name '*name*' in -R option	-Rオプションの説明を参照し、正しい構文を調べてください。
E 537:	default section name '*name*' not allowed	-Rオプションの説明を参照し、正しい構文を調べてください。
W 538:	default section name '*name*' already renamed to '*new_name*'	-Rオプションのいずれか、renamesectプラグマ、他の名前のみを使用してください。
W 542:	optimization stack underflow, no optimization options are saved with #pragma optimize	このワーニングは、前の`#pragma endoptimize`でオプションが保存されていないときに`#pragma endoptimize`を使用すると発生します。
W 555:	current optimization level could reduce debugging comfort (-g);	これらの最適化設定ではHLLデバッグ競合が発生します。
E 560:	Float/Double: not yet implemented	浮動小数点は以下のバージョンでサポートされています。

ライブラリ

<ctype.h>	isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, toascii, _tolower, tolower, _toupper, toupper
<errno.h>	**エラー番号** C**関数はありません**
<float.h>	**浮動小数点演算に関連する定数**
<limits.h>	int**型の制限とサイズを定義しています** C**関数はありません**
<locale.h>	localeconv, setlocale **骨組みとして提供されています**
<math.h>	acos, asin, atan, atan2, ceil, cos, cosh, exp, fabs, floor, fmod, frexp, ldexp, log, log10, modf, pow, sin, sinh, sqrt, tan, tanh
<setjmp.h>	longjmp, setjmp
<signal.h>	raise, signal **これらの関数は骨組みとして提供されています**
<simio.h>	_simi, _simo
<stdarg.h>	va_arg, va_end, va_start
<stddef.h>	offsetof, **特殊型の定義**
<stdio.h>	clearerr, fclose, _fclose, feof, ferror, fflush, fgetc, fgetpos, fgets, fopen, _fopen, fprintf, fputc, fputs, fread, freopen, fscanf, fseek, fsetpos, ftell, fwrite, getc, getchar, gets, _ioread, _iowrite, _lseek, perror, printf, putc, putchar, puts, _read, remove, rename, rewind, scanf, setbuf, setvbuf, sprintf, sscanf, tmpfile, tmpnam, ungetc, vfprintf, vprintf, vsprintf, _write
<stdlib.h>	abort, abs, atexit, atof, atoi, atol, bsearch, calloc, div, exit, free, getenv, labs, ldiv, malloc, mblen, mbstowcs, mbtowc, qsort, rand, realloc, srand, strtod, strtol, stroul, system, wcstombs, wctomb
<string.h>	memchr, memcmp, memcpy, memmove, memset, strcat, strchr, strcmp, strcol, strcpy, strcspn, strerror, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, strtok, strxfrm
<time.h>	asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, time **これらの関数はすべて骨組みとして提供されています**

起動コマンド

as88 [*option*]...*source-file* [*map-file*]

オプション

-C *file*	*file*をソースの前にインクルードします。
-D*macro*[=*def*]	プリプロセッサ*macro*を定義します。
-L[*flag...*]	指定されたソース行をリストファイルから削除します。
-M[**s**|**c**|**d**|**l**]	メモリモデルを指定します。
-V	バージョンヘッダのみを表示します。
-c	大文字と小文字を区別しないモードに切り換えます(デフォルトでは区別する)。
-e	アセンブリエラーの場合、オブジェクトファイルを削除します。
-err	エラーメッセージをエラーファイルにリダイレクトします。
-f *file*	オプションを*file*から読み込みます。
-i[**l**|**g**]	デフォルトのラベルスタイルをローカルまたはグローバルとして指定します。
-l	リストファイルを生成します。
-o *filename*	出力ファイルの名前を指定します。
-t	セクションの要約を表示します。
-v	冗長モード。進行中にファイル名とパスの回数をプリントします。
-w[*num*]	1つまたはすべての警告メッセージを抑制します。

関数

@function_name(*argument*[,*argument*]...)

数学関数

ABS	絶対値
MAX	最大値
MIN	最小値
SGN	記号を返します

文字列関数

CAT	文字列を連結します
LEN	文字列の長さ
POS	文字列内の副文字列の位置
SCP	文字列を比較します
SUB	文字列から副文字列を抽出します

マクロ関数

ARG	マクロ引数関数
CNT	マクロ引数の数
MAC	マクロ定義関数
MXP	マクロ展開関数

アセンブラモード関数

AS88	アセンブラの実行可能ファイル名
DEF	シンボル定義関数
LST	LIST擬似命令のフラグ値
MODEL	選択したアセンブラのモデル

アドレス操作関数

CADDR	コードアドレス
COFF	コードページオフセット
CPAG	コードページ番号
DADDR	データアドレス
DOFF	データページオフセット
DPAG	データページ番号
HIGH	256バイトページ番号
LOW	256バイトページオフセット

アセンブラ擬似命令

デバッグ

CALLS	呼び出し情報をオブジェクトファイルに渡します。オーバーレイセクションを重ね書きするためにリンク時に呼び出しツリーを構築するときに使用されます。
SYMB	シンボリックデバッグ情報を渡します。

アセンブラコントロール

ALIGN	調整を指定します。
COMMENT	コメント行を開始させます。この擬似命令は、IF/ELIF/ELSE/ENDIF構成体およびMACRO/DUP定義では使用できません。
DEFINE	置換文字列を定義します。
DEFSECT	セクション名と属性を定義します。
END	ソースプログラムの終了を示します。
FAIL	プログラマが生成するエラーメッセージです。
INCLUDE	二次ファイルをインクルードします。
MSG	プログラマが生成するメッセージです。
RADIX	定数に対する入力基数を変更します。
SECT	セクションを起動します。
UNDEF	DEFINEシンボルの定義を解除します。
WARN	プログラマが生成する警告です。

シンボル定義

EQU	シンボルと値を等しいものとして定義します。順方向の参照を受け付けます。
EXTERN	外部シンボル宣言です。モジュールの本体でも使用できます。
GLOBAL	グローバルシンボル宣言です。モジュールの本体でも使用できます。
LOCAL	ローカルシンボル宣言です。
NAME	オブジェクトファイルを識別します。
SET	シンボルに値を設定します。順方向の参照を受け付けます。

データ定義/記憶域の割り当て

ASCII	ASCII文字列を定義します。
ASCIZ	NULLを埋め込んだASCII文字列を定義します。
DB	定数バイトを定義します。
DS	記憶域を定義します。
DW	定数ワードを定義します。

マクロおよび条件アセンブラ

DUP	ソース行のシーケンスを複製します。
DUPA	引数を付けてシーケンスを複製します。
DUPC	文字を付けてシーケンスを複製します。
DUPF	ループでシーケンスを複製します。
ENDIF	条件アセンブラの終了。
ENDM	マクロ定義の終了。
EXITM	マクロを終了させます。
IF	条件アセンブラ擬似命令。
MACRO	マクロ定義。
PMACRO	マクロ定義を消去します。

エラーメッセージ

警告(W)

W 101:	use *option* at the start of the source; ignored	プライマリオプションは、ソースの最初に使用しなければなりません。
W 102:	duplicate attribute "*attribute*" found	EXTERN擬似命令の属性が複数回使用されています。重複している属性を削除してください。
W 104:	expected an attribute but got *attribute*; ignored	
W 105:	section activation expected, use *name* directive	SECT擬似命令を使用してセクションをアクティブにしてください。
W 106:	conflicting attributes specified "*attributes*"	EXTERN擬似命令で競合する属性が2回使用されています。
W 107:	memory conflict on object "*name*"	ラベルまたは他のオブジェクトが明示的または暗黙的に定義されていますが、使用されているメモリタイプ同士に互換性がありません。
W 108:	object attributes redefinition "*attributes*"	ラベルまたは他のオブジェクトが明示的または暗黙的に定義されていますが、使用されている属性同士に互換性がありません。
W 109:	label "*label*" not used	ラベル*label*が、GLOBAL擬似命令で定義されていますが、定義も参照もされていません。またはラベルが、LOCAL擬似命令で定義されていますが、参照されていません(LOCALラベルの場合)。
W 110:	extern label *"label*" defined in module, made global	ラベル*label*が、EXTERN擬似命令で定義されていますが、ソース内のラベルとして定義されています。このラベルはグローバルラベルとして扱われます。
W 111:	unknown $LIST flag "*flag*"	$LISTコントロールに未知の*flag*を指定しています。
W 112:	text found after END; ignored	END擬似命令がソースファイルの終わりを指定しています。END擬似命令の後にあるすべてのテキストは無視されます。
W 113:	unknown $MODEL specifier; ignored	未知のモデルを指定しています。
W 114:	$MODEL may only be specified once, it remains "*model*"; ignored	複数のモデルを指定しています。
W 115:	use ON or OFF after control name	指定したコントロールには、コントロール名の後にONかOFFを指定する必要があります。
W 116:	unknown parameter "*parameter*" for *control-name* control	使用できるパラメータについては、コントロールの説明を参照してください。
W 118:	inserted "extern *name*"	シンボル*name*が式の中で使用されていますが、EXTERN擬似命令で定義されていません。
W 119:	"*name*" section has not the MAX attribute; ignoring RESET	

警告(W)

W 120:	assembler debug information: cannot emit non-tiof expression for *label*	IEEE-695オブジェクトフォーマットによってサポートされない式が、SYMBレコードにあります。
W 121:	changed alignment size to *size*	
W 123:	expression: *type-error*	式が、アドレスで不正な動作を実行しているか、互換性のないメモリ空間を結合しています。
W 124:	cannot purge macro during its own definition	
W 125:	"*symbol*" is not a DEFINE symbol	DEFINEされていないシンボル、またはすでに定義解除されたシンボルをUNDEFしようとしました。
W 126:	redefinition of "*define-symbol*"	現在の有効範囲ではシンボルがすでにDEFINEされています。シンボルはこのDEFINEに従って再定義されます。
W 127:	redefinition of macro "*macro*"	マクロがすでに定義されています。マクロはこのマクロ定義に従って再定義されます。
W 128:	number of macro arguments is less than definition	マクロを定義するとき、引数が少なすぎました。
W 129:	number of macro arguments is greater than definition	マクロを定義するとき、引数が多すぎました。
W 130:	DUPA needs at least one value argument	DUPA擬似命令には、ダミーパラメータと値パラメータの2つ以上の引数が必要です。
W 131:	DUPF increment value gives empty macro	DUPFマクロに指定したステップ値がDUPFマクロ本体をスキップしています。
W 132:	IF started in previous file "*file*", line *line*	ENDIFまたはELSEプリプロセッサ擬似命令が、他のファイルのIF擬似命令と対応しています。
W 133:	currently no macro expansion active	@CNT()@関数と@ARG()関数は、マクロ展開の中でのみ使用できます。
W 134:	"*directive*" is not supported, skipped	指定された擬似命令は、本アセンブラではサポートされていません。
W 135:	define symbol of "*define-symbol*" is not an identifier; skipped definition	コマンド行で-Dオプションに不正な識別子を指定しています。
W 137:	label "*label*" defined *attribute* and *attribute*	ラベルはEXTERN擬似命令およびGLOBAL擬似命令で定義されています。
W 138:	warning: *WARN-directive-arguments*	WARN擬似命令からの出力です。
W 139:	expression must be between *hex-value* and *hex-value*	
W 140:	expression must be between *value* and *value*	

エラーメッセージ

警告(W)

W 141:	*global/local* label "*name*" not defined in this module; made extern	ラベルが宣言されて使用されていますが、このソースファイルでは定義されていません。
W 170:	code address maps to zero page	@CPAG関数で指定したコードオフセットが0ページにあります。
W 171:	address offset must be between 0 and FFFF	@CADDR関数または@DADDR関数で指定したオフセットが大きすぎました。
W 172:	page number must be between 0 and FF	@CADDR関数または@DADDR関数で指定したページ番号が大きすぎました。

エラー(E)

E 200:	*message*; halting assembly	アセンブラが、ソースファイルの処理を停止します。
E 201:	unexpected newline or line delimiter	構文チェッカーが、アセンブラの文法に準拠していない改行文字または行区切り文字を見つけました。
E 202:	unexpected character: '*character*'	構文チェッカーが、アセンブラの文法に準拠していない文字を見つけました。
E 203:	illegal escape character in string constant	構文チェッカーが、文字列定数の中に、アセンブラの文法に準拠していない不正なエスケープ文字を見つけました。
E 204:	I/O error: open intermediate file failed (*file*)	アセンブラは、中間ファイルをオープンして、語彙スキャンフェーズを最適化しますが、アセンブラがこのファイルをオープンできません。
E 205:	syntax error: expected *token* at *token*	構文チェッカーが、あるトークンを見つけようとしましたが、別のトークンが見つかりました。
E 206:	syntax error: *token* unexpected	構文チェッカーが、予定外のトークンを見つけました。
E 207:	syntax error: missing ':'	構文チェッカーが、ラベル定義またはメモリ空間変更子を見つけましたが、セミコロンが追加されていませんでした。
E 208:	syntax error: missing ')'	構文チェッカーが、閉じかっこを見つけられません。
E 209:	invalid radix value, should be 2, 8, 10 or 16	RADIX擬似命令は、2、8、10、16のみを受け付けます。
E 210:	syntax error	構文チェッカーがエラーを見つけました。
E 211:	unknown model	正しいモデルを置き換えてください。
E 212:	syntax error: expected *token*	構文チェッカーがトークンを見つけようとしましたが、見つかりませんでした。
E 213:	label "*label*" defined *attribute* and *attribute*	ラベルが、LOCAL擬似命令、GLOBAL擬似命令、EXTERN擬似命令のいずれかで定義されています。
E 214:	illegal addressing mode	ニーモニックが不正なアドレッシングモードを使用しています。
E 215:	not enough operands	ニーモニックに指定されているオペランドが少なすぎます。
E 216:	too many operands	ニーモニックに指定されているオペランドが多すぎます。

エラー(E)

E 217:	*description*	ニーモニックのアセンブル中にエラーが見つかりました。
E 218:	unknown mnemonic: "*name*"	アセンブラが、未知のニーモニックを見つけました。
E 219:	this is not a hardware instruction (use $OPTIMIZE OFF "H")	アセンブラが汎用命令を見つけましたが、-Oh(ハードウェアのみ)オプションまたは$OPTIMIZE ON "H"コントロールが指定されていました。
E 223:	unknown section "*name*"	SECT指示文で指定したセクション名が、DEFSECT指示文で定義されていません。
E 224:	unknown label "*name*"	定義されていないラベルが使用されています。
E 225:	invalid memory type	不正なメモリ変更子が指定されました。
E 226:	unknown symbol attribute: *attribute*	
E 227:	invalid memory attribute	アセンブラが、未知のロケーションカウンタまたはメモリマッピング属性を見つけました。
E 228:	*attr* attribute needs a number	属性*attr*には、1つのパラメータが必要です。
E 229:	only one of the *name* attributes may be specified	
E 230:	invalid section attribute: *name*	アセンブラが、未知のセクション属性を見つけました。
E 231:	absolute section, expected "AT" expression	絶対セクションは、"AT *address*"式を使用して指定する必要があります。
E 232:	MAX/OVERLAY sections need to be named sections	MAX属性またはOVERLAY属性を持つセクションには、名前を付ける必要があります。
E 233:	*type* section cannot have *attribute* attribute	コードセクションには、CLEARまたはOVERLAY属性を指定することができません。
E 234:	section attributes do not match earlier declaration	同じセクションの前の定義で、他の属性が使用されています。
E 235:	redefinition of section	同じ名前の絶対セクションは一度しかロケートできません。
E 236:	cannot evaluate expression of *descriptor*	一部の関数および擬似命令は、アセンブル中にその引数を評価する必要があります。
E 237:	*descriptor* directive must have positive value	一部の擬似命令は、正の引数をとる必要があります。
E 238:	Floating point numbers not allowed with DB directive	DB擬似命令は浮動小数点数をとることができません。
E 239:	byte constant out of range	DB擬似命令は式をバイト単位で格納します。
E 240:	word constant out of range	DW擬似命令は式をワード単位で格納します。
E 241:	Cannot emit non tiof functions, replaced with integral value '0'	浮動小数点式および一部の関数は、IEEE-695オブジェクトフォーマットで表現することができません。
E 242:	the *name* attribute must be specified	セクションには、CODE属性またはDATA属性がなければなりません。

エラーメッセージ

エラー(E)

E 243: use $OBJECT OFF or $OBJECT "*object-file*"	
E 244: unknown control "*name*"	指定されたコントロールが存在しません。
E 246: ENDM within IF/ENDIF	アセンブラがIF/ENDIFの間にENDM擬似命令を見つけました。
E 247: illegal condition code	アセンブラが命令の中に不正な条件コードを見つけました。
E 248: cannot evaluate origin expression of org "*name*: *address*"	絶対セクションのすべての起点は、オブジェクトファイルの作成前に評価されなければなりません。
E 249: incorrect argument types for function "*function*"	指定された引数が、予定外の型に評価されました。
E 250: tiof function not yet implemented: "*function*"	指定されたtiof関数は、まだ実装されていません。
E 251: @POS(,,*start*) start argument past end of string	*start*引数が、最初のパラメータの文字列の長さより長くなっています。
E 252: second definition of label "*label*"	ラベルが、同じ有効範囲内で二度定義されています。
E 253: recursive definition of symbol "*symbol*"	シンボルの評価が、それ自身の値に依存しています。
E 254: missing closing '>' in include directive	構文チェッカーが、INCLUDE擬似命令で閉じかっこ">"を見つけられません。
E 255: could not open include file *include-file*	アセンブラが、このinclude-fileをオープンできませんでした。
E 256: integral divide by zero	式に、0による除算が含まれています。
E 257: unterminated string	すべての文字列は、開始した行と同じ行で終了しなければなりません。
E 258: unexpected characters after macro parameters, possible illegal white space	マクロパラメータの間にはスペースを入れることはできません。
E 259: COMMENT directive not permitted within a macro definition and conditional assembly	本アセンブラでは、MACRO/DUP定義またはIF/ELSE/ENDIF構成体でCOMMENT擬似命令を使用できません。
E 260: definition of "*macro*" unterminated, missing "endm"	マクロ定義がENDM擬似命令で終わっていません。
E 261: macro argument name may not start with an '_'	MACRO引数とDUP引数の最初にアンダースコアを付けることはできません。
E 262: cannot find "*symbol*"	"%"演算子または"?"演算子の引数の定義をマクロ展開内で見つけることができませんでした。
E 263: cannot evaluate: "*symbol*", value is unknown at this point	マクロ展開内で"%"演算子または"?"演算子が使用されているシンボルは、まだ定義されていません。

エラー(E)

E 264: cannot evaluate: "*symbol*", value depends on an unknown symbol	"%"演算子または"?"演算子の引数を、マクロ展開内で評価することができません。
E 265: cannot evaluate argument of *dup* (unknown or location dependant symbols)	DUP擬似命令の引数が評価できませんでした。
E 266: *dup* argument must be integral	DUP擬似命令の引数が整数ではありません。
E 267: *dup* needs a parameter	DUP擬似命令の構文をチェックしてください。
E 268: ENDM without a corresponding MACRO or DUP definition	アセンブラが、対応するMACRO定義またはDUP定義がないENDM擬似命令を見つけました。
E 269: ELSE without a corresponding IF	アセンブラが、対応するIF擬似命令のないELSE擬似命令を見つけました。
E 270: ENDIF without a corresponding IF	アセンブラが、対応するIF擬似命令のないENDIF擬似命令を見つけました。
E 271: missing corresponding ENDIF	アセンブラが、対応するENDIF擬似命令のないIF擬似命令またはELSE擬似命令を見つけました。
E 272: label not permitted with this directive	一部の擬似命令はラベルを受け付けません。
E 273: wrong number of arguments for *function*	この関数には、もっと多い引数またはもっと少ない引数を指定する必要があります。
E 274: illegal argument for *function*	引数に不正な型があります。
E 275: expression not properly aligned	
E 276: immediate value must be between *value* and *value*	命令の即値オペランドは、この範囲の値のみを受け付けます。
E 277: address must be between $*address* and $*address*	アドレスオペランドが、この範囲内にありません。
E 278: operand must be an address	オペランドはアドレスでなければなりませんが、アドレス属性がありません。
E 279: address must be short	
E 280: address must be short	オペランドは短い範囲のアドレスでなければなりません。
E 281: illegal option "*option*"	アセンブラが、未知またはスペルミスのあるコマンド行オプションを見つけました。
E 282: "Symbols:" part not found in map file "*name*"	マップファイルが不完全な可能性があります。
E 283: "Sections:" part not found in map file "*name*"	マップファイルが不完全な可能性があります。
E 284: module "*name*" not found in map file "*name*"	マップファイルが不完全な可能性があります。

エラーメッセージ

エラー(E)

E 285: *file-kind* file will overwrite *file-kind* file	出力ファイルの1つが、コマンド行で指定したソースファイルや他の出力ファイルを上書きするとき、アセンブラは警告を出します。
E 286: $CASE options must be given before any symbol definition	$CASEオプションは、シンボルが定義される前にのみ指定することができます。
E 287: symbolic debug error: *message*	アセンブラが、シンボリックデバッグ(SYMB)命令でエラーを見つけました。
E 288: error in PAGE directive: *message*	PAGE擬似命令に指定された引数は制限に従っていません。
E 290: fail: *message*	FAIL擬似命令の出力。これはユーザ生成エラーです。
E 291: generated check: *message*	Cコンパイラとアセンブラとの間の整合性チェックです。
E 293: expression out of range	命令オペランドは、指定されたアドレス範囲内になければなりません。
E 294: expression must be between *hexvalue* and *hexvalue*	
E 295: expression must be between *value* and *value*	
E 296: optimizer error: *message*	オプティマイザがエラーを見つけました。
E 297: jump address must be a code address	ジャンプおよびジャンプサブルーチンには、コードメモリのターゲットアドレスを指定しなければなりません。
E 298: size depends on location, cannot evaluate	一部の構成体(特にalign擬似命令)のサイズは、メモリアドレスに依存します。

致命的エラー(F)

F 401: memory allocation error	空きメモリに対する要求がシステムによって拒絶されました。すべてのメモリが使用されています。
F 402: duplicate input filename "*file*" and "*file*"	アセンブラでは、コマンド行で入力ファイル名を1つだけ指定する必要があります。
F 403: error opening *file-kind* file: "*file-name*"	アセンブラがこのファイルをオープンできませんでした。
F 404: protection error: *message*	プロテクションキーがないか、またはIBM互換PCではありません。
F 405: I/O error	アセンブラが出力をファイルに書き込むことができません。
F 406: parser stack overflow	
F 407: symbolic debug output error	シンボリックデバッグ情報が、オブジェクトファイルに不正に書き込まれています。
F 408: illegal operator precedence	演算子の優先順テーブルが壊れています。
F 409: Assembler internal error	アセンブラが、内部の不整合を見つけました。

致命的エラー(F)

F 410: Assembler internal error: duplicate mufom "*symbol*" during rename	アセンブラは、有効範囲でローカルなシンボルの名前をすべて固有のシンボル名に変更します。ここでは、アセンブラが、固有の名前を生成できませんでした。
F 411: symbolic debug error: "*message*"	SYMB擬似命令の解析時にエラーが発生しました。
F 412: macro calls nested too deep (possible endless recursive call)	ネストするマクロ展開の数には制限があります。現在この制限は1000になっています。
F 413: cannot evaluate "*function*"	すでに処理されているはずの関数呼び出しが見つかりました。
F 414: cannot recover from previous errors, stopped	前に発生したエラーにより、アセンブラの内部状態が破壊され、プログラムのアセンブルが停止しました。
F 415: error opening temporary file	アセンブラは、デバッグ情報およびリストファイルを生成するとき、一時ファイルを使用します。これらの一時ファイルがオープンまたは作成できませんでした。
F 416: internal error in optimizer	オプティマイザがデッドロックの状態を検出しました。最適化オプションを付けないでアセンブルしてください。

起動コマンド

lk88 [*option*]...*file*...

オプション

-C	**大文字と小文字を区別しないでリンクします(デフォルトでは区別)。**
-L *directory*	**システムライブラリを探すための検索パスを追加します。**
-L	**システムライブラリの検索をスキップします。**
-M	**リンクマップ**(.lnl)**を生成します。**
-N	**重ね書きをオフにします。**
-O *name*	**生成されるマップファイルのベース名を指定します。**
-V	**バージョンヘッダのみを表示します。**
-c	**独立した呼び出しグラフファイル**(.cal)**を生成します。**
-e	**結果にエラーがある場合、その結果を消去します。**
-err	**エラーメッセージをエラーファイル**(.elk)**にリダイレクトします。**
-f *file*	**コマンド行の情報を***file***から読み込みます。**"-"**の場合**stdin**を示します。**
-l *x*	**システムライブラリ**libx.a**内も検索します。**
-o *filename*	**出力ファイルの名前を指定します。**
-r	**定義されていないシンボルの診断を抑制します。**
-u *symbol*	***symbol*****を、シンボルテーブルで定義されていないものとして入力します。**
-v or **-t**	**冗長オプション。進行中にそれぞれのファイル名をプリントします。**
-w *n*	**警告レベルが***n***より上のメッセージを抑制します。**

エラーメッセージ

警告(W)

W 100:	Cannot create map file *filename*, turned off -M option	**このファイルは作成できませんでした。**
W 101:	Illegal filename (*filename*) detected	**不正な拡張子の付いたファイル名が検出されました。**
W 102:	Incomplete type specification, type index = T*hexnumber*	**未知の型が参照されました。**
W 103:	Object name (*name*) differs from filename	**オブジェクトファイルの内部名がファイル名と同じではありません。**
W 104:	'-o *filename*' option overwrites previous '-o *filename*'	**2番目の-oオプションが見つかったため、最初の名前が破棄されます。**
W 105:	No object files found	**起動文でファイルが指定されていません。**
W 106:	No search path for system libraries. Use -L or env "*variable*"	**システムライブラリファイル(-lオプションで指定されたもの)に対して、環境変数またはオプション-Lで定義された検索パスが指定されていなければなりません。**
W 108:	Illegal option: *option* (-H or -¥? for help)	**不正なオプションが検出されました。**
W 109:	Type not completely specified for symbol <*symbol*> in *file*	**現在のファイルまたはこのファイルに、不完全な型指定があります。**
W 110:	Compatible types, different definitions for symbol <*symbol*> in *file*	**互換性のある型の間に名前の競合があります。**
W 111:	Signed/unsigned conflict for symbol <*symbol*> in *file*	**両方の型のサイズは正しいですが、一方の型が符号なしで、もう一方の型が符号付きになっています。**
W 112:	Type conflict for symbol <*symbol*> in *file*	**実数型の競合があります。**
W 113:	Table of contents of *file* out of date, not searched. (Use ar ts <*name*>)	**arライブラリに、最新でないシンボルテーブルがあります。**
W 114:	No table of contents in *file*, not searched. (Use ar ts <*name*>)	**arライブラリにシンボルテーブルがありません。**
W 115:	Library *library* contains ucode which is not supported	ucode**は、このリンカではサポートされていません。**
W 116:	Not all modules are translated with the same threashold (-G value)	**ライブラリファイルに未知のフォーマットがあるか、ファイルが壊れています。**
W 117:	No type found for <*symbol*>. No type check performed	**このシンボルに対する型が生成されていません。**

エラーメッセージ

警告(W)

W 118: Variable <*name*>, has incompatible external addressing modes with file <*filename*>	変数はまだ割り当てられていませんが、2つの外部参照が、オーバーラップしないアドレッシングモードで作成されています。
W 119: error from the Embedded Environment: *message*, switched off relaxed addressing mode check	リンカで組み込み環境が読み込み可能な場合、アドレッシングモードのチェックが解放されます。そのため、たとえば、データとして定義された変数が、HUGEとしてアクセスされる可能性があります。

エラー(E)

E 200: Illegal object, assignment of non existing var *var*	MUFOM変数が存在しません。オブジェクトファイルが壊れています。
E 201: Bad magic number	指定されたライブラリファイルのマジックナンバーに問題があります。
E 202: Section *name* does not have the same attributes as already linked files	指定されたセクションに、異なる属性が指定されています。
E 203: Cannot open *filename*	このファイルが見つかりません。
E 204: Illegal reference in address of *name*	変数の式で不正なMUFOM変数が使用されています。オブジェクトファイルが壊れています。
E 205: Symbol '*name*' already defined in <*name*>	シンボルが二度定義されています。
E 206: Illegal object, multi assignment on *var*	前に発生したE 205の"already defined"エラーにより、MUFOM変数が、複数回割り当てられています。
E 207: Object for different processor characteristics	MAU単位のビット、アドレス単位のMAU、このオブジェクトのエンディアンなどが、最初にリンクしたオブジェクトと異なります。
E 208: Found unresolved external(s):	見つからないシンボルがあります。
E 209: Object format in *file* not supported	オブジェクトファイルに未知のフォーマットがあります。またはオブジェクトファイルが壊れています。
E 210: Library format in *file* not supported	ライブラリファイルに未知のフォーマットがあります。またはライブラリファイルが壊れています。
E 211: Function <*function*> cannot be added to the already built overlay pool <*name*>	オーバーレイプールが前のリンカアクションで構築されています。
E 212: Duplicate absolute section name <*name*>	絶対セクションが固定アドレスで始まっています。そのためリンクできません。
E 213: Section <*name*> does not have the same size as the already linked one	EQUAL属性を持つセクションが、他のリンク済みのセクションと同じサイズになっていません。

エラー(E)

E 214: Missing section address for absolute section <*name*>	それぞれの絶対セクションには、オブジェクト内にセクションアドレスコマンドがなければなりません。オブジェクトファイルが壊れています。
E 215: Section <*name*> has a different address from the already linked one	2つの絶対セクションを同じ条件でリンク(重ね書き)することはできません。それぞれに同じアドレスがなければなりません。
E 216: Variable <*name*>, name <*name*> has incompatible external addressing modes	変数が、参照アドレッシング空間の外側に割り当てられています。
E 217: Variable <*name*>, has incompatible external addressing modes with file <*filename*>	変数はまだ割り当てられていませんが、2つの外部参照がオーバーラップしないアドレッシングモードで行われています。
E 218: Variable <*name*>, also referenced in <*name*> has an incompatible address format	現在のファイルとこのファイルとの間で、異なるアドレス形式のリンクが試みられました。
E 219: Not supported/illegal *feature* in object format *format*	このオブジェクトフォーマットでオプション/機能がサポートされていないか、または不正です。
E 220: page size (0x*hexvalue*) overflow for section <*name*> with size 0x*hexvalue*	セクションが大きすぎてページに収まりません。
E 221: *message*	オブジェクトが生成したエラーです。
E 222: Address of <*name*> not defined	変数にアドレスが割り当てられていません。オブジェクトファイルが壊れています。

致命的エラー(F)

F 400: Cannot create file *filename*	このファイルを作成できませんでした。
F 401: Illegal object: Unknown command at offset *offset*	オブジェクトファイルで未知のコマンドが検出されました。オブジェクトファイルが壊れています。
F 402: Illegal object: Corrupted hex number at offset *offset*	16進数のバイトカウントが不正です。オブジェクトファイルが壊れています。
F 403: Illegal section index	範囲外のセクションインデックスが検出されました。オブジェクトファイルが壊れています。
F 404: Illegal object: Unknown hex value at offset *offset*	オブジェクトファイルで未知の変数が検出されました。オブジェクトファイルが壊れています。
F 405: Internal error *number*	内部の致命的エラーです。
F 406: *message*	キーがないか、またはIBM互換PCではありません。
F 407: Missing section size for section <*name*>	それぞれのセクションでは、オブジェクトにセクションサイズコマンドがなければなりません。オブジェクトファイルが壊れています。

エラーメッセージ

致命的エラー(F)

F 408: Out of memory	より多くのメモリを割り当てようとして失敗しました。
F 409: Illegal object, offset *offset*	オブジェクトモジュールに不整合が見つかりました。
F 410: Illegal object	オブジェクトモジュールの未知のオフセットに不整合が見つかりました。
F 413: Only *name* object can be linked	他のプロセッサ用のオブジェクトは、リンクすることができません。
F 414: Input file *file* same as output file	入力ファイルと出力ファイルは同じにすることができません。
F 415: Demonstration package limits exceeded	このデモバージョンの制限を越えました。

冗長(V)

V 000: Abort!	プログラムがユーザによってアボートされました。
V 001: Extracting files	ライブラリからファイルを抽出することを示す冗長メッセージ。
V 002: File currently in progress:	ファイルを現在処理していることを示す冗長メッセージ。
V 003: Starting pass *number*	このパスの開始を示す冗長メッセージ。
V 004: Rescanning...	ライブラリの再スキャンを示す冗長メッセージ。
V 005: Removing file *file*	削除を示す冗長メッセージ。
V 006: Object file *file* format *format*	指定されたオブジェクトファイルに標準ツールチェーンオブジェクトフォーマットTIOF-695がありません。
V 007: Library *file* format *format*	指定されたライブラリに、標準ツールチェーンar88フォーマットがありません。
V 008: Embedded environment *name* read, relaxed addressing mode check enabled	組み込み環境が読み込まれました。

起動コマンド

lC88 [*option*]...[*file*]...

オプション

-M	**ロケートマップファイル**(.map)**を生成します。**
-S *space*	**特定の***space***を生成します。**
-V	**バージョンヘッダのみを表示します。**
-d *file*	**記述ファイルの情報を***file***から読み込みます。"-"の場合**stdin**を示します。**
-e	**結果にエラーがある場合、その結果を消去します。**
-err	**エラーメッセージをリダイレクトします。**(.elc)
-f *file*	**コマンド行の情報を***file***から読み込みます。"-"の場合**stdin**を示します。**
-f *format*	**出力フォーマットを指定します。**
-o *filename*	**出力ファイルの名前を指定します。**
-p	**stdout にソフトウェア部分のプロポーザルを作成します。**
-v	**冗長オプション。進行中にそれぞれのファイル名をプリントします。**
-w *n*	**警告レベルが***n***より上のメッセージを抑制します。**

エラーメッセージ

警告(W)

W 100:	Maximum buffer size for *name* is *size* (Adjusted)	**このフォーマットの場合、最大バッファサイズが定義されています。**
W 101:	Cannot create map file *filename*, turned off -M option	**このファイルが作成できませんでした。**
W 102:	Only one -g switch allowed, ignored -g before *name*	**デバッグできる**.out**ファイルは**1**つだけです。**
W 104:	Found a negative length for section *name*, made it positive	**負の値の長さをとることができるのは、スタックセクションのみです。**
W 107:	Inserted '*name*' keyword at line *line*	**記述ファイルにないキーワードが挿入されました。**
W 108:	Object name (*name*) differs from filename	**オブジェクトファイルの内部名がファイル名と同じではありません。**
W 110:	Redefinition of system start point	**通常、システムテーブル**(__lc_pm)**にアクセスできるロードモジュールは**1**つだけです。**
W 111:	Two -o options, output name will be *name*	2**番目の**-o**オプションが見つかりました。メッセージには、有効な名前が示されます。**
W 112:	Copy table not referenced, initial data is not copied	layout**部分で**copy**文を使用すると、初期データが**ROM**にロケートされます。**
W 113:	No .out files found to locate	**起動構文で指定されたファイルがありません。**
W 114:	Cannot find start label *label*	**開始点が見つかりませんでした。**
W 116:	Redefinition of name at line *line*	**識別子が二度定義されています。**
W 119:	File *filename* not found in the argument list	**ロケートする必要があるすべてのファイルは引数として指定しなければなりません。**
W 120:	unrecognized name option <*name*> at line *line* (inserted '*name*')	**オプションの割り当てが不正です。**
W 121:	Ignored illegal sub-option '*name*' for *name*	**不正なフォーマットサブオプションが検出されました。**
W 122:	Illegal option: *option* (-H or -¥? for help)	**不正なオプションが検出されました。**
W 123:	Inserted *character* at line *line*	**この文字が、記述ファイルにありませんでした。**
W 124:	Attribute *attribute* at line *line* unknown	**記述ファイルで、未知の属性が指定されています。**
W 125:	Copy table not referenced, blank sections are not cleared	**属性がブランクになっているセクションが検出されましたが、コピーテーブルは参照されていません。ロケータは、コピーテーブルにスタートアップモジュールについての情報を生成して、スタートアップ時にブランクセクションを消去します。**

エラーメッセージ

警告(W)

W 127: Layout *name* not found	指定されているファイルで使用するレイアウトが、layout部分で定義されていなければなりません。
W 130: Physical block *name* assigned for the second time to a layout	ブロックをlayoutブロックに複数回割り当てることはできません。
W 136: Removed *character* at line *line*	ここでは、この文字は必要ありません。
W 137: Cluster *name* declared twice (layout part)	指定されたクラスタが二度宣言されています。
W 138: Absolute section *name* at non-existing memory address 0x*hexnumber*	絶対セクションのアドレスが物理メモリの外に指定されています。
W 139: *message*	組み込み環境からの警告メッセージです。
W 140: File *filename* not found as a parameter	ロケータ記述ファイル(software部分)で定義されたすべてのプロセスは、起動行で指定しなければなりません。
W 141: Unknown space <*name*> in -S option	-Sオプションで未知の空間名が指定されました。
W 142: No room for section *name* in read-only memory, trying writable memory ...	読み取り専用属性を持つセクションは、読み取り専用メモリに配置することができません。このセクションは、書き込み可能メモリに配置されます。

エラー(E)

E 200: Absolute address 0x*hexnumber* occupied	絶対アドレスが要求されましたが、そのアドレスはすでに他のセクションによって占有されています。
E 201: No physical memory available for section *name*	絶対アドレスが要求されましたが、そのアドレスには物理メモリがありません。
E 202: Section *name* with mau size *size* cannot be located in an addressing mode with mau size *size*	ビットセクションは、バイト専用アドレッシングモードでロケートすることができません。
E 203: Illegal object, assignment of non existing var *var*	MUFOM変数が存在しません。
E 204: Cannot duplicate section '*name*' due to hardware limitations	プロセスを複数回ロケートする必要がありますが、セクションが、メモリ管理機能のない仮想空間にマッピングされています。
E 205: Cannot find section for *name*	セクションのない変数が見つかりました。
E 206: Size limit for the section group containing section *name* exceeded by 0x*hexnumber* bytes	小さなセクションが、これ以上ページに収まりません。
E 207: Cannot open *filename*	このファイルが見つかりません。

エラー(E)

E 208: Cannot find a cluster for section *name*	書き込み可能なメモリがないか、アドレッシングモードが未知です。
E 210: Unrecognized keyword <*name*> at line *line*	記述ファイルで未知のキーワードが使用されています。
E 211: Cannot find 0x*hexnumber* bytes for section *name* (fixed mapping)	仮想メモリまたは物理メモリのいずれかが占有されています。また物理メモリがまったくない可能性もあります。
E 213: The physical memory of *name* cannot be addressing in space *name*	マッピングが失敗しました。仮想アドレス空間が残っていません。
E 214: Cannot map section *name*, virtual memory address occupied	絶対マッピングが失敗しました。
E 215: Available space within *name* exceeded by *number* bytes for section *name*	アドレッシングモードで利用可能なアドレッシング空間がなくなりました。
E 217: No room for section *name* in cluster *name*	.dscファイルで定義されているクラスタのサイズが小さすぎます。
E 218: Missing *identifier* at line *line*	識別子を指定しなければなりません。
E 219: Missing ')' at line *line*	閉じかっこが足りません。
E 220: Symbol '*symbol*' already defined in <*name*>	シンボルが二度定義されています。
E 221: Illegal object, multi assignment on *var*	MUFOM変数が複数回割り当てられています。
E 223: No software description found	それぞれの入力ファイルは、.dscファイルのソフトウェア記述に記述しなければなりません。
E 224: Missing <length> keyword in block '*name*' at line *line*	ハードウェア記述に長さの定義がありません。
E 225: Missing <*keyword*> keyword in space '*name*' at line *line*	このマッピングに対して、キーワードを指定しなければなりません。
E 227: Missing <start> keyword in block '*name*' at line *line*	ハードウェア記述に開始の定義がありません。
E 230: Cannot locate section *name*, requested address occupied	絶対アドレスが要求されましたが、そのアドレスはすでに他のプロセスまたはセクションによって占有されています。
E 232: Found file *filename* not defined in the description file	ロケートするすべてのファイルには、それに対する定義レコードが記述ファイルに必要になります。
E 233: Environment variable too long in line *line*	記述ファイルにある環境変数の文字数が多すぎます。
E 235: Unknown section size for section *name*	この.outファイルにセクションサイズが見つかりませんでした。実際には.outファイルが壊れています。

エラーメッセージ

エラー(E)

E 236: Unrecoverable specification at line *line*	記述ファイルに回復不能なエラーがあります。
E 238: Found unresolved external(s):	ロケート時に、すべての外部参照が解決されなければなりません。
E 239: Absolute address *addr.addr* not found	この空間に、絶対アドレスが見つかりませんでした。
E 240: Virtual memory space *name* not found	記述ファイルにある、このファイルのsoftware部分に、存在しないメモリ空間が記述されています。
E 241: Object for different processor characteristics	MAU単位のビット、アドレス単位のMAU、このオブジェクトのエンディアンなどが、最初にリンクしたオブジェクトと異なります。
E 242: *message*	オブジェクトが生成したエラーです。
E 244: Missing *name* part	記述ファイルに、この部分が見つかりませんでした。
E 245: Illegal *name*value at line *line*	記述ファイルに、有効でない値が見つかりました。
E 246: Identifier cannot be a number at line *line*	記述ファイルに、有効でない識別子が見つかりました。
E 247: Incomplete type specification, type index = T*hexnumber*	このファイルにより、未知の型が参照されました。オブジェクトファイルが壊れています。
E 250: Address conflict between block *block1* and *block2* (memory part)	記述ファイルのmemory部分にオーバーラップするアドレスがあります。
E 251: Cannot find 0x*hexnumber* bytes for section *section* in block *block*	セクションをロケートしなければならない物理ブロックに空きがありません。
E 255: Section '*name*' defined more than once at line *line*	1つのlayout/loadmod部分でセクションを複数回宣言することはできません。
E 258: Cannot allocate reserved space for process *number*	空間の予約部分のメモリが占有されています。
E 261: User assert: *message*	ユーザがプログラムした表明が失敗しました。
E 262: Label '*name*' defined more than once in the software part	記述ファイルで定義されているラベルは固有のものでなければなりません。
E 264: *message*	組み込み環境からのエラーメッセージです。
E 265: Unknown section address for absolute section *name*	この.outファイルにセクションアドレスが見つかりませんでした。実際には.outファイルが壊れています。
E 266: %s %s not (yet) supported	要求された機能は、このリリースでは(まだ)サポートされていません。

致命的エラー(F)

F 400: Cannot create file *filename*	このファイルを作成できませんでした。
F 401: Cannot open *filename*	このファイルが見つかりません。
F 402: Illegal object: Unknown command at offset *offset*	オブジェクトファイルで未知のコマンドが検出されました。オブジェクトファイルが壊れています。
F 403: Illegal filename (*name*) detected	不正な拡張子の付いたファイル名がコマンド行で検出されました。
F 404: Illegal object: Corrupted hex number at offset *offset*	16進数のバイトカウントが不正です。オブジェクトファイルが壊れています。
F 405: Illegal section index	範囲外のセクションインデックスが検出されました。
F 406: Illegal object: Unknown hex value at offset *offset*	オブジェクトファイルで未知の変数が検出されました。オブジェクトファイルが壊れています。
F 407: No description file found	ロケータには、システムのハードウェアとソフトウェアを記述している記述ファイルが必要になります。
F 408: *message*	プロテクションキーがないか、またはIBM互換PCではありません。
F 410: Only one description file allowed	ロケータは、記述ファイルを1つだけ受け付けます。
F 411: Out of memory	より多くのメモリを割り当てようとして失敗しました。
F 412: Illegal object, offset *offset*	オブジェクトモジュールに不整合が見つかりました。
F 413: Illegal object	オブジェクトモジュールの未知のオフセットに不整合が見つかりました。
F 415: Only *name* .out files can be located	他のプロセッサ用のオブジェクトは、ロケートすることができません。
F 416: Unrecoverable error at line *line*, *name*	記述ファイルのこの部分に回復不能なエラーがあります。
F 417: Overlaying not yet done	この.outファイルでは重ね書きがまだ実行されていません。
F 418: No layout found, or layout not consistent	レイアウトに構文エラーがある場合、そのレイアウトがロケータで使用できない可能性があります。
F 419: *message*	組み込み環境からの致命的エラーメッセージです。
F 420: Demonstration package limits exceeded	このデモバージョンの制限を越えました。

エラーメッセージ

冗長(V)

V 000: File currently in progress:	冗長メッセージ。次の行に、単一のファイル名が、処理される順にプリントされます。
V 001: Output format: *name*	出力フォーマットの生成を示す冗長メッセージ。
V 002: Starting pass *number*	このパスの開始を示す冗長メッセージ。
V 003: Abort!	プログラムがユーザによってアボートされました。
V 004: Warning level *number*	使用されている警告レベルを報告する冗長メッセージ。
V 005: Removing file *file*	削除を示す冗長メッセージ。
V 006: Found file <*filename*> via path *pathname*	記述(インクルード)ファイルが標準ディレクトリにありませんでした。
V 007: *message*	組み込み環境からの冗長メッセージです。

キーワード

address	**絶対アドレスを指定します。**
amode	**アドレッシングモードを指定します。**
assert	**表明が失敗した場合エラーを出します。**
attribute	**属性をクラスタ、セクション、スタック、ヒープに割り当てます。**
block	**物理メモリ領域を定義します。**
bus	**アドレスバスを定義します。**
chips	CPU**チップを定義します。**
cluster	**クラスタの順序と配置を指定します。**
copy	**データセクションの**ROM**コピーの配置を定義します。**
cpu	cpu**部分を定義します。**
dst	**デスティネーションアドレスを指定します。**
fixed	**メモリマップの固定ポイントを定義します。**
gap	**動的なメモリギャップを予約します。**
heap	**ヒープを定義します。**
label	**仮想アドレスラベルを定義します。**
layout	layout**記述の最初です。**
length	**スタック、ヒープ、物理ブロック、予約空間の長さを指定します。**
load_mod	**ロードモジュールを定義します(プロセス)。**
map	**ソースアドレスをデスティネーションアドレスにマッピングします。**
mau	**最小アドレス可能単位を(ビット単位で)定義します。**
mem	**チップの物理開始アドレスを定義します。**
memory	memory**部分を定義します。**
regsfr	**デバッガが使用するレジスタファイルを指定します。**
reserved	**メモリを予約します。**
section	**セクションがロケートされる方法を定義します。**
selection	**セクションをクラスタにグループ化するときの属性を指定します。**
size	**アドレス空間またはメモリのサイズを指定します。**
software	software**部分を定義します。**
space	**アドレス空間を定義するか、メモリブロックを指定します。**
src	**ソースアドレスを指定します。**
stack	stack**セクションを定義します。**
start	**他の開始ラベルを定義します。**
table	table**セクションを定義します。**

S5U1C88000C Manual **I**
(S1C88 Family**統合ツールパッケージ**)
Cコンパイラ/アセンブラ/リンカ

セイコーエプソン株式会社
半導体事業部　IC営業部

〈IC国内営業グループ〉
東京　〒191-8501　東京都日野市日野421-8
TEL (042) 587-5313 (直通)　FAX (042) 587-5116

大阪　〒541-0059　大阪市中央区博労町3-5-1　エプソン大阪ビル15F
TEL (06) 6120-6000 (代表)　FAX (06) 6120-6100

インターネットによる電子デバイスのご紹介　http://www.epson.jp/device/semicon/

ドキュメントコード：411391000
2008年 1月改訂
2001年10月作成 ⓁB