お客様各位

カタログ等資料中の旧社名の扱いについて

2010年4月1日を以ってNECエレクトロニクス株式会社及び株式会社ルネサステクノロジが合併し、両社の全ての事業が当社に承継されております。従いまして、本資料中には旧社名での表記が残っておりますが、当社の資料として有効ですので、ご理解の程宜しくお願い申し上げます。

ルネサスエレクトロニクス ホームページ（http://www.renesas.com）

2010年4月1日
ルネサスエレクトロニクス株式会社

【発行】ルネサスエレクトロニクス株式会社（http://www.renesas.com）

【問い合わせ先】http://japan.renesas.com/inquiry

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されていない用途に当社製品を使用したことによりお客様または第三者に生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   標準水準： コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通用信号機器、防災・防犯装置、各種安全装置、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害などを生じさせないようお客様の責任において冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、機器またはシステムとしての出荷保証をお願いいたします。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、詳細につきましては製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注 1. 本資料において使用されている「当社」とは、ルネサスエレクトロニクス株式会社およびルネサスエレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

User's Manual

H8S、H8/300シリーズ C/C++コンパイラ、アセンブラ、最適化リンケージエディタ

HSS008CLCS6S　ユーザーズマニュアル

ルネサスマイクロコンピュータ開発環境システム

ルネサス エレクトロニクス
www.renesas.com

Rev.1.00　2003.09

ご注意

安全設計に関するお願い

1. 弊社は品質、信頼性の向上に努めておりますが、半導体製品は故障が発生したり、誤動作する場合があります。弊社の半導体製品の故障又は誤動作によって結果として、人身事故、火災事故、社会的損害などを生じさせないような安全性を考慮した冗長設計、延焼対策設計、誤動作防止設計などの安全設計に十分ご留意ください。

本資料ご利用に際しての留意事項

1. 本資料は、お客様が用途に応じた適切なルネサス テクノロジ製品をご購入いただくための参考資料であり、本資料中に記載の技術情報についてルネサス テクノロジが所有する知的財産権その他の権利の実施、使用を許諾するものではありません。
2. 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他応用回路例の使用に起因する損害、第三者所有の権利に対する侵害に関し、ルネサス テクノロジは責任を負いません。
3. 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他全ての情報は本資料発行時点のものであり、ルネサス テクノロジは、予告なしに、本資料に記載した製品または仕様を変更することがあります。ルネサス テクノロジ半導体製品のご購入に当たりましては、事前にルネサス テクノロジ、ルネサス販売または特約店へ最新の情報をご確認頂きますとともに、ルネサス テクノロジホームページ(http://www.renesas.com)などを通じて公開される情報に常にご注意ください。
4. 本資料に記載した情報は、正確を期すため、慎重に制作したものですが万一本資料の記述誤りに起因する損害がお客様に生じた場合には、ルネサス テクノロジはその責任を負いません。
5. 本資料に記載の製品データ、図、表に示す技術的な内容、プログラム及びアルゴリズムを流用する場合は、技術内容、プログラム、アルゴリズム単位で評価するだけでなく、システム全体で十分に評価し、お客様の責任において適用可否を判断してください。ルネサス テクノロジは、適用可否に対する責任は負いません。
6. 本資料に記載された製品は、人命にかかわるような状況の下で使用される機器あるいはシステムに用いられることを目的として設計、製造されたものではありません。本資料に記載の製品を運輸、移動体用、医療用、航空宇宙用、原子力制御用、海底中継用機器あるいはシステムなど、特殊用途へのご利用をご検討の際には、ルネサス テクノロジ、ルネサス販売または特約店へご照会ください。
7. 本資料の転載、複製については、文書によるルネサス テクノロジの事前の承諾が必要です。
8. 本資料に関し詳細についてのお問い合わせ、その他お気付きの点がございましたらルネサス テクノロジ、ルネサス販売または特約店までご照会ください。

はじめに

本マニュアルは、**「**H8S,H8/300 Series C/C++**コンパイラ、アセンブラ、最適化リンケージエディタ」**の使用方法を述べたものです。

本製品は C、C++言語およびアセンブリ言語で記述したソースプログラムを、H8SX シリーズ、H8S/2600 シリーズ、H8S/2000 シリーズ、H8/300 シリーズ、H8/300 シリーズ、または H8/300L シリーズ用オブジェクトプログラムおよびロードモジュールに変換するソフトウエアシステムです。

ご使用になる前に、本マニュアルを良く読んで理解してください。

■表記上の注意事項

本マニュアルの説明の中で用いられる記号は、次の意味を示しています。

〈 〉	この記号で囲まれた内容を指定することを示します。
［ ］	省略してもよい項目を示します。
. . .	直前の項目を 1 回以上指定することを示します。
△	1 個以上の空白を示します。
(RET)	キャリッジリターンキー(リターンキーともいいます)を示します。
｜	|で区切られた項目を選択できることを示します。
(CNTL)	次の文字を、コントロールキーを押しながら入力することを示します。

本マニュアルは UNIX*[1]、または PC-9801*[2] シリーズ、IBM PC*[3] およびその互換機上で動作する Microsoft® Windows® 98 operating system、Microsoft® Windows® Millennium Edition operating system、Microsoft® Windows NT® operating system、Microsoft® Windows® 2000 operating system、Microsoft® Windows® XP operating system*[4] に対応するように書かれています。UNIX 上で動作するソフトウエアシステムを以下 UNIX 版と称します。または PC-9801 シリーズ、IBM PC およびその互換機上で動作するソフトウエアシステムを以下 PC 版と称します。

【注】 *1 UNIX **は、**X/Open **カンパニーリミテッドがライセンスしている米国ならびに他の国における登録商標です。**

*2 PC-9801 **は、日本電気株式会社の商標です。**

*3 IBM PC **は、米国** International Business Machines Corporation **の登録商標です。**

*4 Microsoft®, Windows®, Windows NT®**は、米国** Microsoft Corporation **の米国及びその他の国における登録商標または商標です。**

目次

1. 概要

1.1 プログラムの開発手順

プログラムの開発手順を**図** 1.1 に示します。網掛け部分は、**「**H8S,H8/300 **シリーズ** C/C++**コンパイラパッケージ」**として提供するソフトウェアを示します。

本マニュアルでは、C/C++コンパイラ、アセンブラ、最適化リンケージエディタ、標準ライブラリ構築ツール、スタック解析ツール、フォーマットコンバータについて説明します。

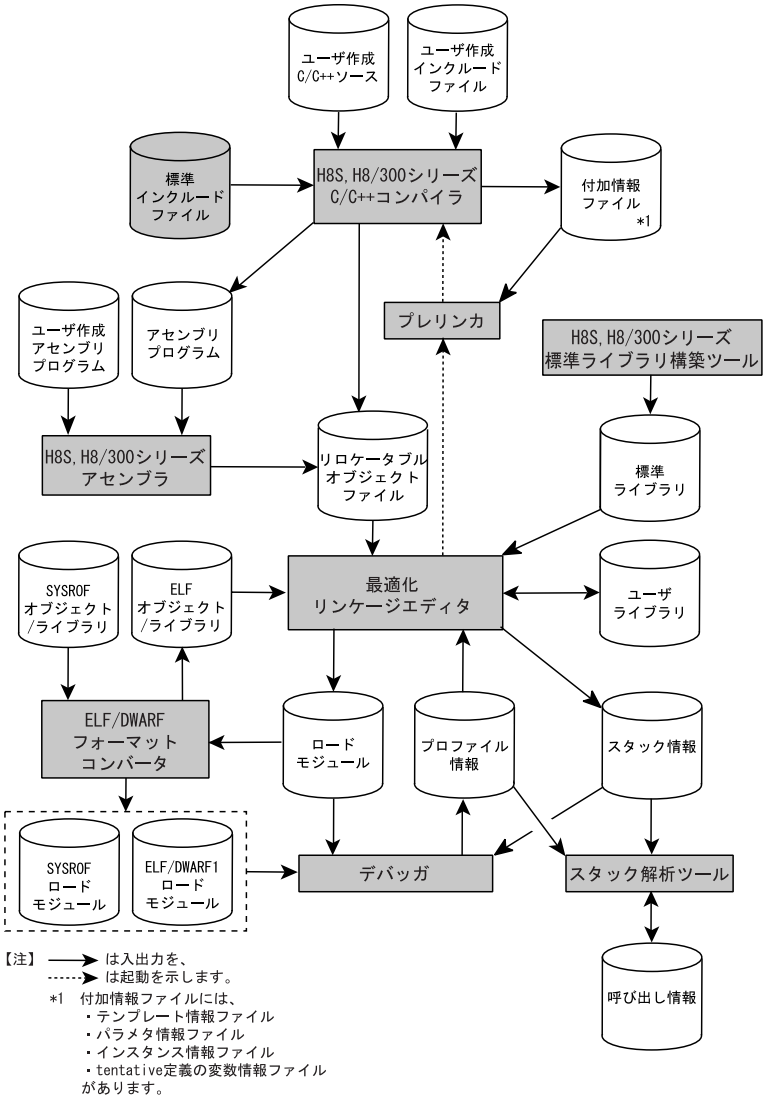


図 1.1 **プログラムの開発手順**

以下、C/C++コンパイラ、アセンブラ、最適化リンケージエディタ、プレリンカ、標準ライブラリ構築ツール、スタック解析ツール、フォーマットコンバータの概要を述べます。

1.2 コンパイラの概要

「H8S,H8/300 **シリーズ** C/C++**コンパイラ」**（以下**コンパイラ**と略す）は、C 言語および C++言語で記述したソースプログラムを入力し、H8S, H8/300 シリーズ用リロケータブルオブジェクトプログラムまたはアセンブリソースプログラムを出力します。

本コンパイラには次の特長があります。

(1) **機器組み込み用として** ROM **化可能なオブジェクトプログラムを生成します。**

(2) **オブジェクトプログラムの実行速度向上やサイズ縮小のための最適化機能をサポートしています。**

(3) CPU **の短絶対アドレッシングモードや間接アドレッシングモードなどの機能を活用するための拡張機能やオプションをサポートしています。**

(4) **プログラム記述言語として、**C **言語、**C++**言語をサポートしています。**

(5) C/C++**言語でサポートしていない割り込み関数やシステム命令記述など、組み込み用プログラム作成に必要な機能を、拡張機能としてサポートしています。**

(6) **デバッガによる** C/C++**ソースレベルデバッグを行うためのデバッグ情報出力を指定できます。**

(7) **アセンブリソースプログラムまたはリロケータブルオブジェクトプログラムを選択して出力することができます。**

(8) **最適化リンケージエディタによるリンク時最適化を行うためのモジュール間最適化情報出力を指定できます。**

1.3 アセンブラの概要

「H8S,H8/300 **シリーズアセンブラ」**（以下**アセンブラ**と略す）は、アセンブリ言語で記述したソースプログラムを入力し、H8S, H8/300 シリーズ用リロケータブルオブジェクトプログラムを出力します。

本アセンブラには次の特長があります。

(1) **次に示すプリプロセッサ機能により、効率よくソースプログラムを記述できます。**

- ファイルインクルード機能
- 条件付アセンブリ機能
- マクロ機能
- 構造化アセンブリ機能

(2) **実行命令、アセンブラ制御命令の名称(ニーモニック)は、**IEEE-694 **仕様で規定された命名規則に準拠し、統一された体系となっています。**

1.4 最適化リンケージエディタの概要

「最適化リンケージエディタ」は、コンパイラおよびアセンブラが出力した複数のオブジェクトプログラムを入力し、ロードモジュールまたはライブラリファイルを出力します。

本最適化リンケージエディタには次の特長があります。

(1) **コンパイラでは最適化できないメモリ配置や関数の呼び出し関係に依存した最適化をオブジェクトプログラムをまたがって実行します。**

(2) **以下の５種類のロードモジュールを選択出力できます。**

- リロケータブル ELF 形式
- アブソリュート ELF 形式
- S タイプ形式
- HEX 形式
- バイナリ形式

(3) **ライブラリファイルを作成・編集できます。**

(4) **シンボル参照回数リストを出力できます。**

(5) **ライブラリ、ロードモジュールファイルのデバッグ情報を削除できます。**

(6) **スタック解析ツールで使用するスタック情報ファイルの出力を指定できます。**

1.5 プレリンカの概要

「プレリンカ」は、最適化リンケージエディタから呼ばれ、C++プログラムのテンプレート、実行時型検査機能を使用している場合に、コンパイラを起動して必要なオブジェクトファイルを生成します。

通常はプレリンカを意識する必要はありませんが、C++プログラムのテンプレート、実行時型検査機能を使用していない場合、最適化リンケージエディタの noprelink オプションを指定することにより、リンク速度を向上できます。

1.6 標準ライブラリ構築ツールの概要

「H8S,H8/300 シリーズ標準ライブラリ構築ツール」（以下**標準ライブラリ構築ツール**と略す）は、コンパイラが提供する標準ライブラリファイルをユーザ指定オプションで構築するソフトウェアシステムです。

本コンパイラが提供する標準ライブラリ関数には、C ライブラリ関数群、組み込み向け C++クラスライブラリ関数群、実行時ルーチン群（プログラムを実行する上で必要な算術演算）があります。ソースプログラム上でライブラリ関数の使用を指定しなくても、実行時ルーチンが必要な場合がありますので注意してください。

1.7 スタック解析ツールの概要

「**スタック解析ツール**」は、最適化リンケージエディタが出力したスタック情報ファイルを入力し、C/C++プログラムのスタック使用量を算出します。

1.8 フォーマットコンバータの概要

「ELF/DWARF **フォーマットコンバータ**」（以下**フォーマットコンバータ**と略す）は、旧バージョンのコンパイラ、アセンブラ出力オブジェクトファイル、ライブラリファイルを入力し、ELF 形式に変換します。または、アブソリュート ELF 形式のロードモジュールを入力し、旧バージョンのリンケージエディタ出力形式に変換します。

2. C/C++コンパイラ操作方法

2.1 オプション指定規則

コンパイラを起動するコマンドラインの形式は以下の通りです。

ch38 [△<オプション> …][△<ファイル名>[△<オプション> …] …]
<オプション>：－<オプション> [=<サブオプション>][,…]

2.2 オプション解説

コマンドライン形式の英大文字は短縮形指定時の文字を、下線は省略時解釈を示します。
また、統合開発環境に対応するダイアログメニューを、タブ名<カテゴリ名>[項目]...で示します。
オプションの順序は、統合開発環境のタブとその中のカテゴリに対応しています。

2.2.1 ソースオプション

表 2.1 **ソースカテゴリオプション一覧**

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	**インクルードファイルディレクトリ**	Include = <**パス名**>[,…]	**コンパイラ** <**ソース**> **[オプション項目** :**]** **[インクルードファイルディレクトリ]**	**インクルードファイルの取り込み先パス名を指定**
2	**デフォルトインクルードファイル**	PREInclude = <**ファイル名**>[,…]	**コンパイラ** <**ソース**> **[オプション項目** :**]** **[デフォルトインクルードファイル]**	**指定したファイルをコンパイル単位の先頭にインクルード**
3	**マクロ名の定義**	DEFine = <sub>[,…] <sub>: <**マクロ名**>[=<**文字列**>]	**コンパイラ** <**ソース**> **[オプション項目** :**]** **[マクロ定義]**	<**文字列**>**を**<**マクロ名**>**として定義**
4	**インフォメーションメッセージ**	Message NOMessage [= <**エラー番号**> [-<**エラー番号**>][,…]]	**コンパイラ** <**ソース**> **[オプション項目** :**]** **[インフォメーションメッセージ]** **[インフォメーションレベルメッセージの表示]**	**出力あり** **出力なし** **（エラー番号、範囲を指定可能）**

インクルードファイルディレクトリ

Include

コンパイラ <ソース>[オプション項目 :][インクルードファイルディレクトリ]

書　式　Include = <パス名>[,…]

説　明　インクルードファイルの存在するパス名を指定します。
パス名が複数ある場合にはカンマ（,）で区切って指定することができます。
システムインクルードファイルの検索は、include オプション指定ディレクトリ、環境変数 CH38 指定ディレクトリの順序で行います。
ユーザインクルードファイルの検索は、カレントディレクトリ、include オプション指定ディレクトリ、環境変数 CH38 指定ディレクトリの順序で行います。

例　ch38 –include=\usr\inc,\usr\CH38 test.c
ディレクトリ\usr\inc と\usr\CH38 をインクルードファイルパスとして検索します。

デフォルトインクルードファイル

PREInclude

コンパイラ <ソース>[オプション項目 :][デフォルトインクルードファイル]

書　式　PREInclude = <ファイル名>[,…]

説　明　指定したファイルの内容をコンパイル単位の先頭に取り込みます。ファイル名が複数ある場合にはカンマ（,）で区切って指定することができます。

例　ch38 –preinclude=a.h test.c
<test.c>の内容

```
int a;
main() {  … }
```

コンパイル時解釈

```
#include "a.h"
int a;
main() {  … }
```

マクロ名の定義

DEFine

コンパイラ <ソース>[オプション項目 :][マクロ定義]

書　式　DEFine = <sub>[,…]
　　<sub>：<マクロ名> [= <文字列>]

説　明　C/C++ソース内で記述する#define と同等の効果を得ます。
<マクロ名>=<文字列>と記述することで<文字列>をマクロ名として定義できます。
サブオプションに<マクロ名>を単独で指定したときは、そのマクロ名が定義されたものとみなします。

インフォメーションメッセージ

Message
NOMessage

コンパイラ <ソース>[オプション項目 :][インフォメーションメッセージ][インフォメーションレベルメッセージの表示]

書　式　Message
NOMessage [= <エラー番号>[-<エラー番号>][,…]]

説　明　インフォメーションレベルのメッセージを出力するかどうかを指定します。
message オプションは、インフォメーションレベルのメッセージを出力します。
nomessage オプションは、インフォメーションレベルの全メッセージの出力を抑止します。
但し、サブオプションでエラー番号を指定すると、指定したメッセージの出力だけを抑止します。
<エラー番号>-<エラー番号>のようにハイフン(-)で抑止するエラー番号の範囲を指定することもできます。
本オプションの省略時解釈は nomessage です。

例　ch38 –nomessage=5,300-306 test.c
C0005 および C0300～C0306 のインフォメーションレベルメッセージの出力を抑止します。

備　考　change_message オプションでウォーニングレベルからインフォメーションレベルに
変更したメッセージは本オプションの対象になります。
前バージョンでは message オプションや nomessage オプションを複数指定すると最後に指定したオプションだけが有効になりましたが、本バージョンでは各 nomessage オプションが指定するエラー番号の和集合のエラー番号が抑止されます。

2.2.2 オブジェクトオプション

表 2.2 オブジェクトカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	プリプロセッサ展開	PREProcessor [= <ファイル名>]	コンパイラ <オブジェクト> [出力ファイル形式 :] [プリプロセッサ展開プログラム]	プリプロセッサ展開後のソースプログラムを出力
2	オブジェクト形式	Code = { Machinecode | Asmcode }	コンパイラ <オブジェクト> [出力ファイル形式 :] [機械語プログラム] [アセンブリプログラム]	機械語プログラムを出力 アセンブリプログラムを出力
3	デバッグ情報	DEBug NODEBug	コンパイラ <オブジェクト> [デバッグ情報出力]	出力あり 出力なし

4	セクション名	SEction = <sub>[,…]
		<sub>:{
		Program=<セクション名> |
		Const = <セクション名> |
		Data = <セクション名> |
		Bss = <セクション名> |
		}
			コンパイラ
			<オブジェクト>
			[セクション :]
			[プログラム領域 (P)]
			[定数領域 (C)]
			[初期化データ領域 (D)]
			[未初期化データ領域 (B)]
				プログラム領域のセクション名
				定数領域のセクション名
				初期化データ領域のセクション名
				未初期化データ領域のセクション名

5	文字列出力領域	STring = { Const | Data }	コンパイラ <オブジェクト> [文字列データ格納 :]	定数領域(C)へ出力 初期化データ領域(D)へ出力
6	乗除算仕様の拡張解釈	CPUExpand NOCPUExpand	コンパイラ <オブジェクト> [乗除算演算方法]	乗除算を CPU 命令仕様に合わせてコード展開 乗除算を ANSI C 言語仕様準拠でコード展開
7	ビットフィールド並び順指定	BIt_order { = < Left | Right > }	コンパイラ <オブジェクト> [ビットフィールド割付 :]	メンバを上位 bit から格納 メンバを下位 bit から格納
8	オブジェクトファイル出力指定	OBject [= <ファイル名>] NOOBject	コンパイラ <オブジェクト> [オブジェクト出力ディレクトリ :]	出力あり 出力なし
9	テンプレートインスタンス生成機能	Template = { None Static | Used | ALl | AUto }	コンパイラ <オブジェクト> [テンプレート生成 :]	インスタンスを生成しません 参照されたものだけ内部リンケージとして生成します 参照されたものだけ外部リンケージとして生成します 宣言、参照されたものを生成します リンク時に生成します
10	境界調整数、バウンダリ調整指定	ALign [=4] NOALign	コンパイラ <オブジェクト> [境界調整別割付 :]	境界調整による割付順変更の実施有無

プリプロセッサ展開

PREProcessor

コンパイラ <オブジェクト>[出力ファイル形式 :][プリプロセッサ展開プログラム]

書　式　PREProcessor [= <ファイル名>]

説　明　プリプロセッサ展開後のソースプログラムを出力します。
ファイル名を指定しない場合は、ソースファイル名と同じファイル名で拡張子が「p」（入力ソースファイルがＣプログラムの時）、または「pp」（入力ソースプログラムが C++プログラムの時）のファイルが作成されます。
preprocessor オプション指定時は、オブジェクトプログラムを出力しません。

備　考　preprocessor オプションを指定したとき、以下のオプションが無効になります。
code、object、outcode、debug、pack、string、
show=object,statistics,allocation、section、optimize、speed、goptimize、
byteenum、volatile、regexpansion、cmncode、case、indirect、abs8、abs16、
cpuexpand、eepmov、regparam、stack、align/noalign、structreg、longreg、
macsave、bit_order、ptr16、opt_range、del_vacant_loop、max_unroll、
infinite_loop、global_alloc、struct_alloc、const_var_propagate、library、
volatile_loop、sbr

オブジェクト形式

Code

コンパイラ <オブジェクト>[出力ファイル形式 :][機械語プログラム][アセンブリプログラム]

書　式　Code ＝ { Machinecode ｜ Asmcode }

説　明　オブジェクトプログラムの出力形式を指定します。
code＝machinecode オプションは、リロケータブルオブジェクト（機械語）プログラムを出力します。
code＝asmcode オプションは、アセンブリプログラムを出力します。
本オプションの省略時解釈は、code＝machinecode です。

備　考　code＝asmcode オプションを指定したとき、show=object,goptimize オプションは無効になります。

デバッグ情報

DEBug
NODEBug

コンパイラ <オブジェクト>[デバッグ情報出力]

書　式　DEBug
NODEBug

説　明　debug オプションは、ソースレベルデバッグに必要なデバッグ情報をオブジェクトファイルに出力します。
本オプションは、最適化オプションを指定した場合も有効となります。
nodebug オプションは、デバッグ情報をオブジェクトファイル中に出力しません。
本オプションの省略時解釈は、nodebug です。

セクション名

SEction

コンパイラ <オブジェクト>[セクション :][プログラム領域 (P)][定数領域 (C)]
[初期化データ領域 (D)][未初期化データ領域 (B)]

書　式
```
SEction  = <sub>[,…]
    <sub>: { Program     = <セクション名> |
             Const       = <セクション名> |
             Data        = <セクション名> |
             Bss         = <セクション名> }
```

説　明　オブジェクトプログラム中のセクション名を指定します。
section=program=<セクション名>は、プログラム領域のセクション名を指定します。
section=const=<セクション名>は、定数領域のセクション名を指定します。
section=data=<セクション名>は、初期化データ領域のセクション名を指定します。
section=bss=<セクション名>は、未初期化データ領域のセクション名を指定します。
<セクション名>は、英字、数字、アンダーライン（_）または、$の列で、先頭が数字以外のものです。セクション名は、8192 文字目まで有効です。
本オプションの省略時解釈は、section＝program＝P, const＝C, data＝D, bss＝B, です。

備　考　プログラムとセクション名の対応についての詳細は、**「9.1　プログラムの構造」**を参照してください。
領域が異なるセクションに同じセクション名を指定できません。Section オプションで P, C, B, D セクションを S セクションへ変更するとウォーニングエラーを発生します。S はスタック領域用に予約されたセクション名です。

文字列出力領域

STring

コンパイラ <オブジェクト>[文字列データ格納 :]

書　式　STring = { Const | Data }

説　明　文字列の出力先を指定します。
string＝const オプション指定時は、定数領域に出力します。
string＝data オプション指定時は、初期化データ領域に出力します。
初期化データ領域へ出力した文字列はプログラム実行時に変更することができますが、ROM 上と RAM 上に二重に領域を確保し、プログラム実行開始時に ROM から RAM へ転送する必要があります。初期化データ領域の初期設定、メモリ割り付けの方法については、**「9.2.1 メモリ領域の割り付け」**を参照してください。
本オプションの省略時解釈は、string＝const です。

CPUExpand
NOCPUExpand

コンパイラ <オブジェクト>[乗除算演算方法]

書　式　CPUExpand
NOCPUExpand

説　明　変数の乗除算のコード展開を ANSI 規格から拡張解釈して生成します。
nocpuexpand オプションは、乗除算のコード展開を ANSI 規格に準拠した形で生成します。
本オプションの省略時解釈は、nocpuexpand です。

備　考　cpuexpand オプションを指定した場合、言語仕様で規定された値の保証範囲と仕様が異なるため、演算結果が nocpuexpand オプション指定時と異なる場合があります。
本オプションの指定による乗除算のコード展開を**表** 2.3 に示します。

表 2.3　cpuexpand **オプションの演算仕様**

対象演算	us1*us2 **の演算サイズ**（H8S/2600 **の例**）	
	cpuexpand **指定時**	nocpuexpand **指定時**
unsigned short us1, us2; unsigned long ul; ul＝us1*us2;	us1*us2 **の中間結果を** unsigned long **で保持します。*** **例：**MOV.W @_us1,Rd MOV.W @_us2,Rs MULXU.W Rs,ERd MOV.L ERd,@_ul us1*us2 **の結果** 4 **バイトを** u1 **に代入します。**	us1*us2 **は** unsigned short **で演算します。** **例：**MOV.W @_us1,Rd MOV.W @_us2,Rs MULXU.W Rs,ERd EXTU.L ERd MOV.L ERd,@_ul us1*us2 **の結果の下位** 2 **バイトを** 0 **拡張して** ul **に代入します。**
unsigned short us1,us2,us3; unsigned short us; us= us1*us2/us3;	us1*us2 **の中間結果を** unsigned long **で保持します。*** **例：**MOV.W @_us1,Rd MOV.W @_us2,Rs MULXU.W Rs,ERd MOV.W @_us3,Rs DIVXU.W Rs,ERd MOV.W Rd,@_us us1*us2 **の結果** 4 **バイトを除算命令の被除数にします。**	us1*us2 **は** unsigned short **で演算します。** **例：**MOV.W @_us1,Rd MOV.W @_us2,Rs MULXU.W Rs,ERd EXTU.L ERd MOV.W @_us3, Rs DIVXU.W Rs,ERd MOV.W Rd,@_us us1*us2 **の結果の下位** 2 **バイトを** 0 **拡張した値を除算命令の被除数にします。**

【注】*２バイト同士の乗算の結果を４バイトに代入またはキャストする場合、または、２バイトで除算する場合に、４バイトで保持した中間結果を用います。

ビットフィールド並び順指定

BIt_order

コンパイラ <オブジェクト>[ビットフィールド割付 :]

書　式　　BIt_order ＝ { Left ｜ Right }

内　容　　ビットフィールドのメンバの並び順を指定します。
bit_order=left を指定した場合は上位ビットからメンバを割り付けます。
bit_order=right を指定した場合は下位ビットからメンバを割り付けます。
本オプションの省略時解釈は　bit_order=left です。

備　考　　ビットフィールドのメンバの割り付けについては、**「**10.1.2 **データの内部表現」**、および**「**10.2.1 #pragma **拡張子」**の#pragma bit_order を参照してください。
同一データのビットフィールドメンバの並び順がファイル間で整合性がとれるように注意してください。

オブジェクトファイル出力指定

OBject
NOOBject

コンパイラ <オブジェクト>[オブジェクト出力ディレクトリ :]

書　式　　OBject [= <オブジェクトファイル名>]
NOOBject

説　明　　オブジェクトファイルの出力有無を指定します。
noobject オプションは、オブジェクトファイルを出力しません。
object オプションでオブジェクトファイル名を指定しない場合には、ソースファイルと同じファイル名で拡張子が「obj」（出力ファイルがリロケータブルオブジェクトプログラムの時）、または「src」（出力ファイルがアセンブリソースプログラムの時）のオブジェクトファイルを出力します。
ファイル拡張子が「obj」か「src」かは、code オプションで決まります。
本オプションの省略時解釈は object です。

備　考　　noobject オプションを指定したとき、以下のオプションが無効になります。
outcode、debug、pack、string、show=object、statistics、allocation、section、optimize、speed、goptimize、byteenum、volatile、regexpansion、cmncode、case、indirect、abs8、abs16、cpuexpand、eepmov、regparam、stack、align/noalign、structreg、longreg、macsave、bit_order、ptr16、opt_range、del_vacant_loop、max_unroll、infinite_loop、global_alloc、struct_alloc、const_var_propagate、library、volatile_loop、sbr

テンプレートインスタンス生成機能

Template

コンパイラ <オブジェクト>[テンプレート生成 :]

書　式　Template ={ None |
Static |
Used |
ALl |
AUto }

説　明　テンプレートのインスタンス生成方法を指定します。
template=none を指定した場合、インスタンスの生成を行いません。
template=static を指定した場合、コンパイル単位内で参照されたテンプレートのみインスタンスを作成します。ただし、生成される関数は内部リンケージを持ちます。
template=used を指定した場合、コンパイル単位内で参照されたテンプレートのみインスタンスを作成します。ただし、生成される関数は外部リンケージを持ちます。
template=all を指定した場合、コンパイル単位内で宣言または参照されている全てのテンプレートのインスタンスを作成します。
templete=auto を指定した場合、リンク時に必要なインスタンスの生成を行います。

備　考　本オプションの省略時解釈は template=auto ですが、アセンブリソース出力時は、常に template=static になります。

ALign
NOALign

コンパイラ <オブジェクト>[境界調整別割付 :]

書　式　ALign [=4]
NOALign

説　明　noalign オプション指定時は、宣言された変数を宣言順に配置します。
align オプション指定時は、境界調整数による空き領域が小さくなるよう変数の再配置を行います。再配置を行った場合、一般的に空き領域となる部分が減少し、オブジェクトサイズが減少します。
align=4 オプション指定時は、データを境界調整数が 4 のセクションと 2 のセクションと 1 のセクションに分割します。サイズが 4 の倍数のデータを境界調整数 4 のセクションへ生成します。セクション名は元のセクション名に$4 を付加したものになります。CPU 種別が H8SX の場合に 4 バイト境界に割りついた 4 バイト変数へのアクセス速度が向上します。
また、サイズが奇数のデータを境界調整数 1 の別セクションへ生成します。セクション名は元のセクション名に$1 を付加したものになります。これにより空き領域を減らせます。
残りのデータつまりサイズが偶数であるが 4 の倍数ではないデータは元のセクションに残ります。
また、pragma 指定やオプション指定によりセクション名を変更した場合でも、変更されたセクション名に対して$4 や$1 が付加されます。

本オプションの省略時解釈は、align です。

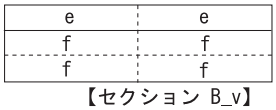
備　考　CPU 種別が H8SX の場合のみ、align=4 オプション指定が有効です。
Align=4 指定時の 1 バイト/4 バイトデータセクションを特定のアドレスに配置するには optlnk の start オプションでそれぞれのセクションを明示的に指定する必要があります。
バウンダリのデータ構成を変更しない場合は、noalign オプションを指定してください。

例

```
char a;
short b;
char c;
long d;
#pragma section _v
short e;
long f;
#pragma section

main()
{
  ...
```

● noalign指定

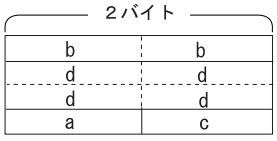
【セクション B】
サイズ：10 bytes
境界調整数：2



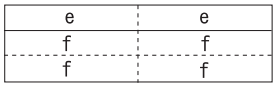
【セクション B_v】
サイズ：6 bytes
境界調整数：2

・データを宣言順に配置する
・境界調整数が2のデータは必ず偶数アドレスに割り当てられるので、奇数サイズデータが前にあると使用されない空き領域が生じる場合がある。

● align指定（デフォルト設定）



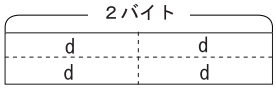
【セクション B】
サイズ：8 bytes
境界調整数：2



【セクション B_v】
サイズ：6 bytes
境界調整数：2

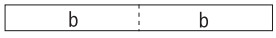
・空き領域が最小になるよう、境界調整数が2のデータを配列してから境界調整数が1のデータを配列する。

●align=4指定

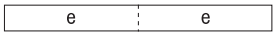


【セクション B$4】
サイズ：4 bytes
境界調整数：4

【セクション B_v$4】
サイズ：4 bytes
境界調整数：4



【セクション B】
サイズ：2 bytes
境界調整数：2



【セクション B_v】
サイズ：2 bytes
境界調整数：2

【セクション B$1】
サイズ：2 bytes
境界調整数：1

・データを３種類に分類する
X. サイズが4の倍数のデータ
Y. サイズが奇数のデータ
Z. その他（サイズが偶数だが4の倍数でないデータ）
・データのサイズを見てセクションを分割する。例えばBセクションはB$4、B$1、Bに分割する。
X. サイズが4の倍数のデータから成るセクションの境界調整数を4とし セクション名末尾に"$4"を付加する。（例：B$4）
Y. サイズが奇数のデータから成るセクションの境界整数を1とし、セクション名の末尾に"$1"を付加する。（例：B$1）
Z. その他のデータから成るセクションは境界調整数が2の元のセクションに残る。（例：B）

2.2.3 リストオプション

表 2.4 **リストカテゴリオプション一覧**

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	**リスト ファイル**	List [= <**ファイル名**>] NOList	**コンパイラ** <**リスト**> **[コンパイルリスト出力]**	**出力あり** **出力なし**
2	**リスト内容と形式**	SHow = <sub>[,…] <sub>:{ SOurce | NOSOurce | Object | NOObject | STatistics | NOSTatistics | Allocation | NOAllocation | Expansion|NOExpansion| Width = <**数値**> | Length = <**数値**> | Tab = { 4 | 8 } }	**コンパイラ** <**リスト**> **[リスト内容 :]**	**ソースリストの有無** **オブジェクトリストの有無** **統計情報の有無** **シンボル割り付けリストの有無** **マクロ展開後リストの有無** **１行の最大文字数**: 0, 80〜132 **ページ内の最大行数**: 0, 20〜255 **タブ使用時のカラム数：**4 | 8

List
NOList

コンパイラ <リスト>[コンパイルリスト出力]

書　式　List [= <リストファイル名>]
NOList

説　明　リストファイルの出力有無を指定します。
list オプション指定時は、<リストファイル名>を指定することができます。
nolist オプションを指定すると、リストファイルは出力しません。
<リストファイル名>は、**「8.1　ファイル名の付け方」**に従って指定できます。
list オプションで<リストファイル名>を指定しない場合には、ソースファイルと同じファイル名で、拡張子が UNIX 版のとき「lis」、PC 版のとき「lst」（入力ソースファイルがCプログラムの時）、または「lpp」（入力ソースプログラムが C++プログラムの時）のリストファイルが作成されます。
本オプションの省略時解釈は list です。

リスト内容と形式

SHow

コンパイラ <リスト>[リスト内容 :]

書　式

```
SHow = <sub>[,...]
     <sub>:{SOurce       |NOSOurce      |
            Object       |NOObject      |
            STatistics   |NOSTatistics  |
            Allocation   |NOAllocation  |
            Expansion    |NOExpansion   |
            Width =〈数値〉               |
            Length=〈数値〉               |
            Tab    = {  4 |  8  }       }
```

説　明　コンパイラが出力するリストの内容とその形式、および出力の解除を指定します。
本項で記した各リストの具体例については**「8.2　コンパイルリストの参照方法」**を参照してください。
本オプションの省略時解釈は、
show＝source、noobject、statistics、noallocation、noexpansion、width＝0、length＝0、tab＝8 です。

備　考　サブオプションの一覧を**表** 2.5 に示します。

表 2.5　show **オプションのサブオプション一覧**

サブオプション名	**意味**
source	**ソースプログラムのリストを出力します。**
nosource	**ソースプログラムのリストを出力しません。**
object	**オブジェクトプログラムのリストを出力します。**
noobject	**オブジェクトプログラムのリストを出力しません。**
statistics	**統計情報のリストを出力します。**
nostatistics	**統計情報のリストを出力しません。**
allocation	**シンボル割り付け情報のリストを出力します。**
noallocation	**シンボル割り付け情報のリストを出力しません。**
expansion	**インクルードファイル、マクロ展開した後のソースプログラムリストを出力します。**nosource **サブオプションが同時に指定された場合には、**expansion **サブオプションは無効となり、ソースプログラムリストは出力されません。**
noexpansion	**インクルードファイル、マクロを展開する前のソースプログラムリストを出力します。** nosource **サブオプションが同時に指定された場合は、**noexpansion **サブオプションは無効となり、ソースプログラムリストは出力されません。**
width＝**〈数値〉**	**〈数値〉で指定する文字数をリストの** 1 **行の最大文字数とします。〈数値〉は** 10 **進数で指定し、**0**、または** 80 **から** 132 **の間の数値を指定することができます。〈数値〉が** 0 **の場合、リストの** 1 **行の最大文字数は規定されません。**
length＝**〈数値〉**	**〈数値〉で指定する行数を、リストの** 1 **ページの最大行数とします。〈数値〉は** 10 **進数で指定し、**0**、または** 20 **から** 255 **の間の数値を指定することができます。** **〈数値〉が** 0 **の場合、リストの** 1 **ページの最大行数は規定されません。**
tab＝{ 4 | 8 }	**リスト表示時のタブのサイズを指定します。**

2.2.4 最適化オプション

表 2.6 最適化カテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	最適化レベル	OPtimize = { 0 | 1 }	コンパイラ <最適化> [最適化]	最適化なし 最適化あり
2	モジュール間最適化	Goptimize	コンパイラ <最適化> [モジュール間最適化]	モジュール間最適化用付加情報出力
3	スピード優先最適化	SPeed [= <sub>[,…]] <sub>: { Register | SHift | Loop [= { 1 | 2 }] | SWitch | Inline [=<数値>] | STruct | Expression }	コンパイラ <最適化> [最適化方法 :] [サイズ優先 :] [スピード優先 :] [スピード優先最適化オプション :]	スピード優先のコード生成を指定 レジスタ退避・回復を push,pop 展開 シフト演算の高速化 ループ文での帰納変数削除 ループ文での帰納変数削除、ループ展開 switch 文の高速化 自動インライン展開 構造体代入式の高速化 四則演算、比較、代入式の高速化
4	switch 文展開方式	CAse = { Auto | Ifthen | Table }	コンパイラ <最適化> [Switch 文展開 :]	speed オプション指定有無で判定 if_then 方式で展開 テーブルジャンプ方式で展開
5	メモリ間接形式	INDirect = { Normal | Extended }	コンパイラ <最適化> [関数呼び出し :]	関数呼び出しをメモリ間接形式で展開 関数呼び出しを拡張メモリ間接形式で展開
6	ポインタサイズ指定	PTr16	コンパイラ <最適化> [2 バイトポインタ]	変数を指すポインタ全てのサイズを 2 バイトに指定する
7	短絶対アドレス	ABS8 ABS16	コンパイラ <最適化> [データアクセス :]	8 ビットデータを 8 ビット絶対アドレスでアクセス 全データを 16 ビット絶対アドレスでアクセス
8	外部変数の最適化	Volatile NOVolatile	コンパイラ <最適化> [詳細…] [外部変数] [外部変数の volatile 扱い]	外部変数の最適化抑止 外部変数を最適化
9	外部変数最適化範囲指定	OPT_Range = { All | NOLoop | NOBlock }	コンパイラ <最適化> [詳細…] [外部変数] [外部変数の最適化範囲 :]	関数内の全範囲で外部変数を最適化対象とする ループ制御変数やループ内の外部変数のループ外への移動を抑止 ループや分岐をまたいだ外部変数に対する最適化を全て抑止
10	空ループ削除	DEL_vacant_loop = { 0 | 1 }	コンパイラ <最適化> [詳細…] [その他] [空ループ削除]	空ループ削除を抑止 空ループ削除を行う
11	ループ最大展開数の指定	MAX_unroll = <数値> <数値>：1-32	コンパイラ <最適化> [詳細…] [その他] [ループ展開最大数 :]	ループ展開時最大展開数を指定 default：1（speed, speed=loop[=2]指定時 2）

	項目	コマンドライン形式	ダイアログメニュー	指定内容
12	無限ループ前の式削除	INFinite_loop = { 0 | 1 }	コンパイラ <最適化> [詳細…] [外部変数] [無限ループ前の外部変数への代入式削除]	無限ループ前の外部変数の代入式の削除を行わない 無限ループ前の外部変数の代入式の削除を行う
13	外部変数のレジスタ割り付け	GLOBAL_Alloc = { 0 | 1 }	コンパイラ <最適化> [詳細…] [外部変数] [外部変数のレジスタ割付]	外部変数のレジスタ割り付けを抑止 外部変数のレジスタ割り付けを行う
14	構造体/共用体メンバのレジスタ割り付け	STRUCT_Alloc = { 0 | 1 }	コンパイラ <最適化> [詳細…] [その他] [構造体/共用体メンバのレジスタ割付]	構造体/共用体メンバのレジスタ割り付けを抑止 構造体/共用体メンバのレジスタ割り付けを行う
15	const 定数伝播	CONST_Var_propagate = { 0 | 1 }	コンパイラ <最適化> [詳細…] [外部変数] [外部定数の定数伝播]	const 宣言された外部定数の定数伝播を抑止 const 宣言された外部定数の定数伝播を行う
16	特定ライブラリ関数のインライン展開指定	LIBrary = { Function | Intrinsic }	コンパイラ <最適化> [詳細…] [その他] [memcpy/strcpy のインライン展開]	memcpy と strcpy を関数呼び出しする memcpy と strcpy をインライン展開する

最適化レベル

OPtimize

コンパイラ <最適化>[最適化]

書　式　OPtimize ＝ { 0 ｜ 1 }

説　明　オブジェクトプログラムの最適化レベルを指定します。
optimize＝0 オプション指定時は、オブジェクトプログラムの最適化を行いません。
optimize＝1 オプション指定時は、最適化を行います。
本オプションの省略時解釈は、optimize＝1 です。

備　考　optimize＝0 オプションを指定したとき、speed＝inline,loop オプションは無効となります。

モジュール間最適化

Goptimize

コンパイラ <最適化>[モジュール間最適化]

書　式　Goptimize

説　明　モジュール間最適化用付加情報を出力します。
本オプションを指定したファイルは、リンク時にモジュール間最適化の対象になります。

スピード優先最適化

SPeed

コンパイラ <最適化>[最適化方法 :][サイズ優先 :][スピード優先 :][スピード優先最適化オプション :]

書　式

```
SPeed = <sub>[,…]
  <sub>:{Register                                    |
         SHift                                       |
         Loop [={ 1 | 2 }]                           |
         SWitch                                      |
         Inline [ = <数値>]                          |
         STruct                                      |
         Expression                                 }
```

説　明　コンパイラが生成するオブジェクトに対し、実行速度の高速化を図る最適化を指定します。
speed＝register オプションは、CPU／動作モードが 300ha、300hn,300 の時、関数の入口／出口で実行時ルーチンを使用せず、レジスタの退避／回復を PUSH､POP 命令で展開します。
speed＝shift オプションは､実行時ルーチンを使用せずシフト演算をコードで展開します。
speed＝loop=1 を指定した場合、帰納変数の削除だけを行います。
spped＝loop=2 を指定した場合は､帰納変数の削除及びループ展開の最適化を実施します。
speed＝switch オプションは、switch 文判定式のコード展開方式をスピード優先で選択します。
speed＝inline オプションは、サイズの小さい関数をインライン展開します。
speed=inline=<数値>で、インライン展開対象の最大サイズを変更できます。CPU 種別が H8SX の場合、<数値>はプログラムサイズが何%増加するまでインライン展開を行うかを表します。例えば speed=inline=50 を指定した場合、プログラムサイズが 50%増加するまで（1.5 倍になるまで）インライン展開します。
CPU 種別が H8SX でない場合､<数値>はインライン展開可能な関数のノード数(宣言を除く、変数・演算子の語句の総数)を表します。即ち、<数値>で示した基準より小さい関数をインライン展開します｡このときプログラムサイズの増加量はインライン展開される関数の大きさと出現頻度に依存し、H8SX の場合のように増加量の上限を指定するものではありません。
<数値>省略時の解釈は CPU 種別が H8SX の場合は 100、それ以外は 110 です。
インライン展開の条件については、**「**10.2.1(2) **関数に関する機能拡張」の**#pragma inline/_ _inline **の項目**を参照してください。
speed＝struct オプションは、構造体型や double 型の代入文のコード展開時、実行時ルーチンを使用しません。
speed=expression オプションは、四則演算、比較、代入式を実行時ルーチンを使わないコードで展開します。（一部の式で対象外になるものがあります）
speed のみを指定した場合は、これら全ての実行速度優先の最適化を行います。本オプション省略時は、実行速度よりもオブジェクトコードのサイズ縮小を重視したオブジェクトを生成します。

備　考　最適化なし（optimize＝0）を指定したとき、speed＝loop,inline オプションは無効となります。

CAse

コンパイラ <最適化>[Switch 文展開 :]

書　式　CAse = { Auto | Ifthen | Table }

説　明　switch 文のコード展開方式を指定します。
case=auto オプションは、オブジェクトサイズの縮小を優先したいずれかの展開方式をコンパイラが自動的に選択します。
また、speed オプションあるいは speed＝switch オプションを指定した場合は、実行速度を優先した展開方式をコンパイラが自動的に選択します。
case=ifthen オプションは、switch 文を if_then 方式で展開します。if_then 方式は、switch 文の評価式の値と case ラベルの値を比較し、一致すれば case ラベルの文へ飛ぶ処理を case ラベルの回数繰り返す展開方式です。この方式は、switch 文に含まれる case ラベルの数に比例してオブジェクトコードのサイズが増大します。
case=table オプションは、switch 文をテーブル方式で展開します。テーブル方式は、case ラベルの飛び先をジャンプテーブルに確保し、1 回のジャンプテーブルの参照で switch 文の評価式と一致する case ラベルの文へ飛び越す展開方式です。この方式は、switch 文に含まれる case ラベルの数に比例して定数領域に確保されるジャンプテーブルのサイズが増えますが、実行速度は常に一定です。
本オプションの省略時解釈は、case=auto です。

例

```
int a, b;
 :
switch(a){
 case 1:    b=3; break;
 case 2:    b=2; break;
 case 3:    b=1; break;
}
```

上記のソースプログラムのコード展開例を次に示します。（cpu＝2600n の場合）

case=ifthen 時

```
  MOV.W   @_a:16,R0
  MOV.B   R0H,R0H
  BNE     Ld
  CMP.B   #1,R0L
  BEQ     L1
  CMP.B   #2,R0L
  BEQ     L2
  CMP.B   #3,ROL
  BNE     L4
  BRA     L3
L1:
```

case=table 時

```
  MOV.W   @_a:16,R0
  SUB.W   #H'1,R0
  CMP.W   #H'2,R0
  BHI     Ld
  MOV.B   @(Ll:16,ER0),R0L
  EXTU.W  R0
  ADD.W   #LWORD Lp,R0
  JMP     @ER0
Lp:
          :
L1:（ジャンプテーブル）
```

表 2.7　引数の変化によるサイズ、サイクルの違い

a の値	if_then 方式		テーブル方式	
	オブジェクトサイズ	実行サイクル	オブジェクトサイズ	実行サイクル
1 --- 3	22 バイト	9 --- 17	29（26+3）バイト	17

メモリ間接形式

INDirect

コンパイラ <最適化>[関数呼び出し :]

書　式　INDirect = {Normal | Extended}

説　明　ソースプログラム内で関数を呼び出す際のメモリ間接形式を指定します。
indirect=normal 指定の場合、全ての関数をメモリ間接（@@aa:8）で呼び出します。
indirect=extended 指定の場合、全ての関数を拡張メモリ間接(@@vec:7)で呼び出します。
ソースプログラム中に定義されている関数について、メモリ間接呼び出しのためのアドレステーブルが以下のセクションに作成されます。
・indirect=normal 指定の場合、 "$INDIRECT"セクション
・indirect=extended 指定の場合、"$EXINDIRECT"セクション
アドレステーブルのセクション名切り替えの方法については、**「10.2.1(1)メモリ配置に関する拡張機能」**の**セクションの切り替え指定**を参照してください。

備　考　アドレステーブルを格納できるエリアは、以下のアドレスの範囲に制限されています。
・"$INDIRECT"セクション　: 0x0000～0x00FF 番地
・"$EXINDIRECT"セクション : 0x0100～0x01FF 番地　H8SX ノーマルモード
: 0x0200～0x03FF 番地　H8SX その他のモード
リンク時には、start オプションでこれらセクションの配置を各アドレス範囲内に明示的に指定してください。
indirect=extended 指定は CPU 種別が H8SX の場合にのみ有効です。
特定の関数についてのみメモリ間接形式を指定したい場合は、#pragma indirect、_ _indirect、_ _indirect_ex を用います。これらの指定は、本オプション指定よりも優先されて扱われます。詳細は**「10.2.1(2)　関数に関する拡張機能」**を参照してください。

同一関数の定義側と呼び出し側とで normal または extended どちらかに統一してください。

ポインタサイズ指定

PTr16

コンパイラ <最適化>[2 バイトポインタ]

書　式　PTr16

説　明　データを指すポインタのサイズを２バイトにします。

備　考　本オプションが指定されなかった場合、データを指すポインタのサイズは４バイトです。
本オプションを指定した場合は、参照先のデータセクションを 16 ビット絶対アドレス領域に明示的に配置する必要があります。セクション配置のアドレス指定は最適化リンケージエディタの start オプション指定により行います。詳細は**「4.2.5 セクションオプション」の start オプションを参照してください。また、16 ビット絶対アドレス領域に関しては「18.3 短絶対アドレスのアクセス範囲」を参照してください。**
本オプション指定は CPU/動作モードが H8SXA,H8SXX の場合だけ有効となります。
データへのポインタのサイズが４バイトから２バイトになることによりリソースの割り当てだけでなく、関数の引数の渡し方やリターン値の受け取り方に影響します。同一データや同一関数の扱いがファイル間で整合性がとれるように注意してください。

ABS8
ABS16

コンパイラ <最適化>[データアクセス :]

書　式　ABS8
ABS16

説　明　静的領域に割り付けるデータを、短絶対アドレッシングモードでアクセスします。
abs8 オプションは、char 型、unsigned char 型データ、および char 型、unsigned char 型の要素、メンバを含む 1 バイト境界整合の複合型データを 8 ビット絶対アドレス（@aa:8）でアクセスするコードを生成します。
abs16 オプションは、CPU／動作モードが H8SXA、H8SXX、2600a、2000a、300ha のとき、データを 16 ビット絶対アドレス（@aa:16）でアクセスするコードを生成します。CPU／動作モードが H8SXN、H8SXM、2600n、2000n、300hn、300 のとき、abs16 オプションの指定は無効です。
abs8 オプションにより 8 ビット絶対アドレスでアクセスされるデータは、セクション名“ $ABS8C ”、“ $ABS8D ”または“ $ABS8B ”に出力されます。また、abs16 オプションにより、16 ビット絶対アドレスでアクセスされるデータは、セクション名“ $ABS16C ”、“ $ABS16D ”または“ $ABS16B ”に出力されます。

短絶対アドレッシングモードでアクセスする変数は、#pragma abs8,#pragma abs16 の拡張子および、__abs8,__abs16 のキーワードでも指定できます。
オプションと拡張子／キーワードの両方が指定された場合は、拡張子／キーワードの指定を優先します。

備　考　リンク時には、本オプションにより出力されたセクションを短絶対アドレス領域に割り付ける必要があります。短絶対アドレス領域の範囲については、**「18.3　短絶対アドレスのアクセス範囲」**を参照してください。また、短絶対アドレス領域のセクション名の切り替え方法については、**「10.2.1(1)メモリ配置に関する拡張機能」**の**セクションの切り替え指定**を参照してください。

Volatile
NOVolatile

コンパイラ <最適化>[詳細…][外部変数][外部変数の volatile 扱い]

書　式　Volatile
NOVolatile

説　明　volatile オプションを指定した場合、全ての外部変数に対して最適化を行いません。
novolatile オプションを指定した場合、volatile 修飾子のない外部変数に対して最適化を行います。
本オプションの省略時解釈は、novolatile です。

例　ソースプログラム

```
volatile int a;
int b;
void main(void){
  a;
  b;
}
```

・volatile 指定時

```
    mov.w @_a,R0
    mov.w @_b,R0    ;b を volatile 変数としてアクセスします
    rts
```

・novolatile 指定時

```
    mov.w @_a,R0
    rts             ;b のアクセスは最適化の結果削除されます
```

外部変数最適化範囲指定

OPT_Range

コンパイラ <最適化>[詳細…][外部変数][外部変数の最適化範囲 :]

書　式　OPT_Range = { All | NOLoop | NOBlock }

説　明　opt_range=all を指定した場合、関数内の全範囲を対象に外部変数に対する最適化を行います。
opt_range=noloop を指定した場合、ループ内にある外部変数やループ判定式で使用されている外部変数を最適化の対象外にします。
opt_range=noblock を指定した場合、分岐をまたいだ外部変数の最適化(ループを含む)をすべて抑止します。
本オプション省略時解釈は、opt_range=all です。

例　(1) 分岐をまたいだ最適化例(opt_range=all/noloop 指定時に行う)

```
int A,B,C;
void f(int a) {
    A = 1;
    if (a) {
        B = 1;
    }
    C = A;
}
```

＜最適化後のソースイメージ＞

```
void f(int a) {
    A = 1;
    if (a) {
        B = 1;
    }
    C = 1;    /* A の参照を削除し、A=1 を伝播する */
}
```

(2) ループにおける最適化例(opt_range=all 指定時に行う)

```
int A,B,C[100];          /* 外部変数 */
void f() {
    int i;
    for (i=0;i<A;i++) {
        C[i] = B;
    }
}
```

＜最適化後のソースイメージ＞

```
void f() {
    int i;
    int temp_A, temp_B; /* 局所変数 */
    temp_A = A;         /* ループ判定式の A の参照をループ外に移動 */
    temp_B = B;         /* ループ内の B の参照をループ外に移動     */
    for (i=0;i<temp_A;i++) {    /* A のループ内での参照を削除 */
        C[i] = temp_B;          /* B のループ内での参照を削除 */
    }
}
```

備　考　　本オプションは CPU 種別が H8SX の場合のみ有効となります。
opt_range=noloop を指定した場合、常に max_unroll=1 がデフォルトになります。
opt_range=noblock を指定した場合、常に max_unroll=1、const_var_propagate=0、global_alloc=0 がデフォルトになります。

空ループ削除

DEL_vacant_loop

コンパイラ <最適化>[詳細…][その他][空ループ削除]

書　式　　DEL_vacant_loop = { 0 | 1 }

説　明　　del_vacant_loop=0 を指定した場合、ループ内処理がない場合でもループを削除しません。
del_vacant_loop=1 を指定した場合、ループ内処理がないループは削除します。
本オプション省略時解釈は、del_vacant_loop=0 です。

備　考　　本オプションは CPU 種別が H8SX の場合のみ有効となります。

ループ最大展開数の指定

MAX_unroll

コンパイラ <最適化>[詳細…][その他][ループ展開最大数 :]

書　式　MAX_unroll = <数値>

説　明　ループ展開時の最大展開数を指定します。<数値>には 1 から 32 までの整数を指定することができます。それ以外の値を指定した場合はエラーになります。

本オプション省略時解釈は、speed または speed=loop[=2]オプションを指定した場合は max_unroll=2、それ以外の場合は max_unroll=1 です。

備　考　本オプションは CPU 種別が H8SX の場合のみ有効となります。
opt_range=noloop/noblock を指定した場合、常に max_unroll=1 がデフォルトになります。

無限ループ前の式削除

INFinite_loop

コンパイラ <最適化>[詳細…][外部変数][無限ループ前の外部変数への代入式削除]

書　式　INFinite_loop ＝ {0 | 1}

説　明　infinite_loop=0 を指定した場合、無限ループ直前での外部変数への代入を削除しません。
infinite_loop=1 を指定した場合、無限ループ直前にあり無限ループ内で参照されない外部変数への代入式を削除します。
本オプション省略時解釈は、infinite_loop=0 です。

例

```
int A;
void f()
{
    A = 1;          /* 外部変数 A への代入式 */
    while(1) {}     /* A は参照されない */
}

<infinite_loop=1 指定時のイメージ>
void f()
{
                    /* 外部変数 A への代入式を削除 */
    while(1) {}
}
```

備　考　本オプションは CPU 種別が H8SX の場合のみ有効となります。

外部変数のレジスタ割り付け

GLOBAL_Alloc

コンパイラ <最適化>[詳細…][外部変数][外部変数のレジスタ割付]

書　式　GLOBAL_Alloc = { 0 | 1 }

説　明　global_alloc=0 を指定した場合、外部変数のレジスタ割り付けを抑止します。
global_alloc=1 を指定した場合、外部変数のレジスタ割り付けを行います。
本オプション省略時解釈は、global_alloc=1 です。

備　考　本オプションは CPU 種別が H8SX の場合のみ有効となります。
opt_range=noblock を指定した場合、global_alloc=0 がデフォルトになります。

構造体/共用体メンバのレジスタ割り付け

STRUCT_Alloc

コンパイラ <最適化>[詳細…][その他][構造体/共用体メンバのレジスタ割付]

書　式　STRUCT_Alloc = { 0 | 1 }

説　明　struct_alloc=0 を指定した場合、構造体/共用体メンバのレジスタ割り付けを抑止します。
struct_alloc=1 を指定した場合、構造体/共用体メンバのレジスタ割り付けを行います。
本オプション省略時解釈は、struct_alloc=1 です。

備　考　本オプションは CPU 種別が H8SX の場合のみ有効となります。
opt_range=noblock もしくは global_alloc=0 を指定しかつ struct_alloc=1 を
指定した場合、ローカル構造体/共用体メンバのみレジスタ割り付けを行います。

const 定数伝播

CONST_Var_propagate

コンパイラ <最適化>[詳細…][外部変数][外部定数の定数伝播]

書　式　CONST_Var_propagate = { 0 | 1 }

説　明　const_var_propagate=0 を指定した場合、const 宣言された外部変数の定数伝播を抑止します。
const_var_propagate=1 を指定した場合、const 宣言された外部変数についても定数伝播を行います。
本オプション省略時解釈は、const_var_propagate=1 です。

例

```
const int X = 1;
int A;
void f() {
    A = X;
}

＜const_var_propagate =1 指定時のソースイメージ＞
void f() {
    A = 1;    /* X=1 を伝播 */
}
```

備　考　本オプションは CPU 種別が H8SX の場合のみ有効となります。
opt_range=noblock を指定した場合、const_var_propagate=0 がデフォルトになります。
C++プログラムで const 宣言された変数については本オプションで制御することはできません
(常に定数伝播されます)。

特定ライブラリ関数のインライン展開指定

LIBrary

コンパイラ <最適化>[詳細…][その他][memcpy/strcpy のインライン展開]

書　式　LIBrary = { Function | Intrinsic }

説　明　ライブラリ関数 memcpy, strcpy に関して
library=function を指定した場合、当該関数を関数呼び出しします。
library=intrinsic を指定した場合、当該関数をインライン展開します。

備　考　library=intrinsic 指定は CPU 種別が H8SX の場合のみ有効となります。

2.2.5 その他オプション

表 2.8 その他カテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	コメントのネスト	COMment	コンパイラ <その他> [その他のオプション :] [コメント(/* */)のネストを許す]	コメント(/* */)のネストを許す
2	組み込み向けC++言語	ECpp	コンパイラ <その他> [その他のオプション :] [EC++言語に基づいたチェック]	EC++言語に基づいたシンタックスチェックおよび、使用するメモリ管理用ライブラリの判別
3	MACレジスタ保証	MAcsave	コンパイラ <その他> [その他のオプション :] [割り込み関数の前後で MACレジスタを常に保証]	割り込み関数の前後で MACレジスタを常に保証
4	ループ判定式最適化抑止	VOLATILE_Loop	コンパイラ <その他> [その他のオプション :] [ループ判定式の最適化抑止]	ループ判定式の最適化を抑止する
5	列挙型サイズ.	Byteenum	コンパイラ <その他> [その他のオプション :] [列挙型データを char 型で扱う]	宣言した列挙型のデータをchar 型で扱う
6	変数割り付けレジスタ数の拡張	Regexpansion NORegexpansion	コンパイラ <その他> [その他のオプション :] [変数割付レジスタ数を拡張]	(E)R3～(E)R6 を使用 (E)R4～(E)R6 を使用
7	共通式の最適化	CMncode	コンパイラ <その他> [その他のオプション :] [共通式削除の最適化を強化]	共通式削除の最適化強化
8	ブロック転送命令	EEpmov	コンパイラ <その他> [その他のオプション :] [構造体の代入式を eepmov 命令で展開]	構造体の代入式を eepmov 命令で展開
9	プリプロセッサ展開時出力制限	NOLINe	コンパイラ <その他> [その他のオプション :] [プリプロセッサ展開時に#line 出力抑止]	プリプロセッサ展開時に#line の出力を抑止
10	メッセージレベル	CHAnge_message =<sub>[,…] <sub>:<level> [=<n>[-m],…] <level>:{Information | Warning | Error }	コンパイラ <その他> [ユーザ指定オプション :]	メッセージレベルの変更

コメントのネスト

COMment

コンパイラ <その他>[その他のオプション :][コメント(/* */)のネストを許す]

書　式　COMment

説　明　ネストしたコメントの記述を可能にします。
本オプションを省略した場合、コメントのネストを記述するとエラーになります。

例　
```
/* This is an example of/* nested */ comment */
                                        ↑
                                       [1]
```
comment オプションを指定すると全てコメントと解釈しますが、省略した場合は［1］でコメントが終わっていると解釈します。

組み込み向け C++言語

ECpp

コンパイラ <その他>[その他のオプション :][EC++言語に基づいたチェック]

書　式　ECpp

説　明　Embedded C++ 言語仕様に基づいて、C++プログラムのシンタックスチェックを行います。
Embedded C++ 言語仕様では、catch、const_cast、dynamic_cast、explicit、mutable、namespace、reinterpret_cast、static_cast、template、throw、try、typeid、typename、using をサポートしていません。これらのキーワードを記述した場合、エラーメッセージを出力します。

また本オプションは、EC++/C++で使用するメモリ管理用ライブラリを判別します。
EC++ライブラリを使用する場合は、必ずこのオプションを指定してください。

備　考　Embedded C++ 言語仕様では、多重継承、仮想基底クラスをサポートしていません。
多重継承、仮想基底クラスを記述した場合は、エラーメッセージ
"C5882 (E) Embedded C++ does not support multiple or virtual inheritance"
を出力します。
本オプションと exception オプションを同時に指定することはできません。

MAC レジスタ保証

MAcsave

コンパイラ <その他>[その他のオプション :][割り込み関数の前後で MAC レジスタを常に保証]

書　式　　MAcsave

説　明　　MAC レジスタを、割り込み関数の前後で常に保証します。
macsave オプションが指定され、割り込み関数内で MAC レジスタを使用する場合、または関数呼び出しがある場合に、MAC レジスタの退避／回復コードを生成します。
macsave オプションが指定されていないとき、割り込み関数内で MAC レジスタを使用する場合のみに、MAC レジスタの退避／回復コードを生成します。

ループ判定式最適化抑止

VOLATILE_Loop

コンパイラ <その他>[その他のオプション :][ループ判定式の最適化抑止]

書　式　　VOLATILE_Loop

説　明　　ループ判定式に外部変数を含む場合、最適化対象外にします。
ただし、型変換を伴う場合、外部変数を２つ以上含む場合、または複合演算の場合は、最適化抑止対象にならない場合があります。

備　考　　本オプションは CPU 種別が H8SX の場合のみ有効となります。
本オプションを指定した場合、volatile オプションを指定しない場合でも当該外部変数がループ内において最適化抑止対象となります。
本オプションを指定しなかった場合、ループ判定式がループ内で不変の時に削除される場合があります。

列挙型サイズ

Byteenum

コンパイラ <その他>[その他のオプション :][列挙型データを char 型で扱う]

書　式　Byteenum

説　明　enum 宣言した列挙型のデータを char 型、unsigned char 型として扱います。
本オプションが指定された場合で、enum 宣言した列挙型のメンバの値の範囲に応じて、列挙型データの型を選択します。値の範囲が-128～127 の場合は char 型、0～255 の場合は unsigned char 型として扱われます。
本オプションを省略した場合、および、本オプションが指定されても列挙型のメンバの値が上記の範囲外の場合は、列挙型データを int 型として扱います。

例　ソースプログラム

```
enum EM {a,b,c} E;
void main(void){E=b;}
```

・byteenum 指定時

```
     mov.b #1,R0L   ;1 バイトデータ転送を行います
     mov.b R0L,@_E
     rts
  _E:
     .res.b 1       ;E を 1 バイト領域に割り当てます
```

・byteenum 指定なし時

```
    mov.w #1,R0     ;2byte データ転送を行います
    mov.w R0,@_E
    rts
  _E:
    .res.w 1        ;E を 2 バイト領域に割り当てます
```

備　考　同一列挙型データの型解釈がファイル間で整合するように注意してください。

変数割り付けレジスタ数の拡張

Regexpansion
NORegexpansion

コンパイラ <その他>[その他のオプション :][変数割付レジスタ数を拡張]

書　式　Regexpansion
NORegexpansion

説　明　regexpansion オプションは、レジスタ変数を割り付けるレジスタの数を拡張します。
noregexpansion オプションは、レジスタ変数を割り付けるレジスタの数を拡張しません。
レジスタの数を拡張した場合、一般にレジスタに割り付く変数の数が多くなり、変数のアクセススピードが速くなります。
レジスタ変数の割り付け規則については、**「**9.3.2(3)　**レジスタに関する規則」**を参照してください。
本オプションの省略時解釈は、regexpansion です。

備　考　regexpansion 指定は CPU 種別が H8SX の場合は無効になります。

共通式の最適化

CMncode

コンパイラ <その他>[その他のオプション :][共通式削除の最適化を強化]

書　式　CMncode

説　明　共通式をテンポラリ変数に変換する最適化で、対象となる式の数を拡張します。
cmncode オプション指定により共通式最適化の対象式を拡張すると、テンポラリ変数をレジスタに割り付け、一般的にはオブジェクト性能がよくなります。しかし、レジスタの数が不足するとテンポラリ変数がメモリに割り付いて、逆にオブジェクト性能が低下してしまうことがあります。本オプションは、プログラムによってオブジェクト性能向上の効果が変わりますので、性能チューニング時に試してみてください。

備　考　cmncode 指定は CPU 種別が H8SX の場合は無効になります。

ブロック転送命令

EEpmov

コンパイラ <その他>[その他のオプション :][構造体の代入式を eepmov 命令で展開]

書　式　EEpmov

説　明　構造体の代入文や局所変数で宣言された配列の初期値代入式を、CPU 種別が H8SX の場合はブロック転送命令 MOVMD、その他の CPU 種別ではブロック転送命令 EEPMOV でコード展開します。転送サイズが大きくてブロック転送命令で対応できない場合は、実行時ルーチンで展開します。
本オプションを省略した場合は、構造体の代入文などを MOV 命令または、実行時ルーチンで展開します。

備　考　CPU 種別が H8SX、H8/300、H8/300L でない場合、EEPMOV.W 命令実行中に割り込みを受け付けると、割り込み処理終了後、次の命令に制御が移るため動作結果が保証されません。この場合、割り込みを受け付ける可能性がある関数に対しては、本オプションは指定しないでください。

プリプロセッサ展開時出力制限

NOLINe

コンパイラ <その他>[その他のオプション :][プリプロセッサ展開時に#line 出力抑止]

書　式　NOLINe

説　明　本オプションを指定した場合、プリプロセッサ展開時に#line の出力を抑止します。

備　考　本オプションは preprocessor オプションが指定された場合にのみ有効です。

メッセージレベル

CHAnge_message

コンパイラ <その他>[ユーザ指定オプション :]

書　式
```
CHAnge_message = <sub>[,…]
    <sub>         : <エラーレベル>[=<エラー番号>[- <エラー番号>][,…]]
    <エラーレベル> : { Information | Warning | Error }
```

説　明　インフォメーション、ウォーニングのメッセージレベルを変更します。

例

```
change_message=information=1001,5038-5047
```
C1001 および C5038 から C5047 までの Warning レベルの指定エラー番号を Information レベルに変更します。

```
change_message=warning=5007-5009
```
C5007 から C5009 までの Information レベルの指定エラー番号を Warning レベルに変更します。

```
change_message=error=2-1024
```
C0002 から C1024 までの Information および Warning レベルの指定エラー番号を Error レベルに変更します。

```
change_message=information
```
全ての Warning レベルメッセージを Information レベルに変更します。

```
change_message=warning
```
全ての Information レベルメッセージを Warning レベルに変更します。

```
change_message=error
```
全てのインフォメーション、ウォーニングメッセージを Error レベルに変更します。

備　考　インフォメーションレベルに変更したメッセージについては、nomessage オプション指定により出力を抑止することができます。
存在しないエラー番号は無視します。
本オプションを複数指定した場合、全て有効となります。但し複数指定で番号が重なった場合、後の指定を変更します。

2.2.6 CPU オプション

表 2.9 CPU タブオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	CPU 種別／動作モード	CPu = { H8SXN [:<*2>] | H8SXM [:<*1>][:<*2>] | H8SXA [:<*1>][:<*2>] | H8SXX [:<*1>][:<*2>] | 2600N | 2600A [:<*1>] | 2000N | 2000A [:<*1>] | 300HN | 300HA[:<*1>] | 300 | 300L | 300Reg }	CPU [CPU 種別 :] [乗除算器指定 :]	H8SX ノーマルモード H8SX ミドルモード H8SX アドバンストモード H8SX マキシマムモード H8S/2600 ノーマルモード H8S/2600 アドバンストモード H8S/2000 ノーマルモード H8S/2000 アドバンストモード H8/300H ノーマルモード H8/300H アドバンストモード H8/300
2	引数格納レジスタ	REGParam = { 2 | 3 }	CPU [引数格納レジスタを２つから３つに変更]	(E)R0,(E)R1 を使用 (E)R0,(E)R1,(E)R2 を使用
3	構造体パラメタ、リターン値のレジスタ割付	STRUctreg NOSTRUctreg	CPU [構造体パラメタ、リターン値をレジスタに割り付ける]	4byte 以下の構造体パラメタおよびリターン値をレジスタに割付ける
4	4byte パラメタ、リターン値のレジスタ割付	LONgreg NOLONgreg	CPU [4-byte パラメタ、リターン値をレジスタに割り付ける]	4byte のパラメタおよび、リターン値をレジスタに割付ける (cpu=300)
5	double→float 変換	DOuble=Float	CPU [double 型の変数/引数を float 型として扱う]	double 型の変数／数値を float 型として扱う
6	スタック計算サイズ指定	STAck = { Small | Medium | Large }	CPU [スタック計算時のサイズ :]	スタック計算時のサイズを指定 1byte 2byte 4byte
7	実行時型情報	RTti = { ON | OFf }	CPU [C++の dynamic_cast、typeid を有効にする]	dynamic_cast、typeid を 有効にする 無効にする
8	例外処理機能	EXception NOEXception	CPU [C++の try、throw、catch を有効にする]	例外処理機能を有効にする 例外処理機能を無効にする
9	構造体、共用体、クラスメンバの境界調整数	PAck = { 1 | 2 }	CPU [メンバの境界調整数を 1 とする]	データの境界調整数を 1 とする データの境界調整に従う
10	8bit 絶対領域アドレス値指定	SBr = <アドレス>	CPU [SBR 値 :]	8bit 絶対領域の開始アドレスを指定

*1 : アドレス空間のビット幅

*2 : 乗除算器指定

CPu

CPU [CPU 種別 :][乗除算器指定 :]

書　式　CPu = {H8SXN [: <乗除算器指定>]|
H8SXM[: <アドレス空間のビット幅>][: <乗除算器指定>]|
H8SXA[: <アドレス空間のビット幅>][: <乗除算器指定>]|
H8SXX[: <アドレス空間のビット幅>][: <乗除算器指定>]|
2600N |
2600A[: <アドレス空間のビット幅>] |
2000N |
2000A[: <アドレス空間のビット幅>] |
300HN |
300HA[: <アドレス空間のビット幅>] |
300｜300L｜300reg}

<アドレス空間のビット幅> : {20 | 24 | 28 | 32}
<乗除算器指定> : {M | D | MD} M:乗算器 D:除算器

説　明　作成するオブジェクトプログラムの CPU 種別と動作モードを指定します。
乗除算器指定は、入力がない場合には「乗除算器なし」として扱われます。
サブオプションの一覧と指定可能なビット幅を**表** 2.1 0 に示します。

表 2.10　cpu **オプションのサブオプション一覧**

サブオプション名	意　味	ビット幅	乗除算器
H8SXN	H8SX **用ノーマルモード**	−	M, D, MD
H8SXM	H8SX **用ミドルモード**	20, 24	M, D, MD
H8SXA	H8SX **用アドバンストモード**	20, 24, 28, 32	M, D, MD
H8SXX	H8SX **用マキシマムモード**	28, 32	M, D, MD
2600n	H8S/2600 **用ノーマルモード**	−	−
2600a	H8S/2600 **用アドバンストモード**	20, 24, 28, 32	−
2000n	H8S/2000 **用ノーマルモード**	−	−
2000a	H8S/2000 **用アドバンストモード**	20, 24, 28, 32	−
300hn	H8/300H **用ノーマルモード**	−	−
300ha	H8/300H **用アドバンストモード**	20, 24	−
300	H8/300 **のオブジェクト**	−	−
300l	H8/300 **のオブジェクト アセンブラとの互換のために用意しています。**	−	−
300reg	H8/300 **のオブジェクト 旧バージョンとの互換のために用意しています。**	−	−

(ビット幅が指定されなかった場合、下線部のデフォルト値に設定されます)

例
```
-cpu = H8SXM:20      ;乗除算器無し,ビット幅 20bit の H8SX ミドルモード
-cpu = h8sxa:32:md   ;乗除算器有り,ビット幅 32bit の H8SX アドバンストモード
-cpu = H8SXA:D       ;除算器有り,ビット幅 24bit の H8SX アドバンストモード
```

備　考　cpu オプションを省略した場合は、H38CPU 環境変数の内容を参照します。
cpu オプションと H38CPU 環境変数を同時に指定した場合、cpu オプションを優先します。
cpu オプションと H38CPU 環境変数の両方を省略した場合はエラーとなります。

引数格納レジスタ

REGParam

CPU [引数格納レジスタを２つから３つに変更]

書　式　REGParam = { 2 | 3 }

説　明　引数格納用レジスタの本数を指定します。
regparam=2 が指定されたとき、引数格納用レジスタとして ER0、ER1（H8/300 では R0、R1）の２本を使用します。
regparam=3 が指定されたとき、引数格納用レジスタとして ER0、ER1、ER2（H8/300 では R0、R1、R2）の３本を使用します。
本オプションの省略時解釈は、regparam=2 です。

構造体パラメタのレジスタ割り付け

STRUctreg
NOSTRUctreg

CPU [構造体パラメタ、リターン値をレジスタに割り付ける]

書　式　STRUctreg
NOSTRUctreg

説　明　構造体のパラメタおよびリターン値を、レジスタに割り付けるかどうかを指定します。
nostructreg オプションを指定した場合､レジスタを使用せずメモリを使用して引数を渡します。
structreg オプションを指定した場合、レジスタを使用して引数を渡します。
パラメタとして渡せる構造体のサイズの上限は、CPU=300 時は 2 バイト、それ以外の CPU では 4 バイトとなります。
本オプションの省略時解釈は、nostructreg です。

備　考　H8/300 時で longreg を指定した場合、4 バイトまでのデータを割り付けることができます。

4byte パラメタのレジスタ割り付け

LONgreg
NOLONgreg

CPU [4-byte パラメタ、リターン値をレジスタに割り付ける]

書　式　LONgreg
NOLONgreg

説　明　4 バイトのパラメタやリターン値をレジスタに割り付けるかどうかを指定します。
本オプションを指定することによってレジスタに割り付く変数は、long 型、unsigned long 型および、float 型です。
nolongreg オプションを指定した場合、レジスタを使用せずメモリを使用して引数を渡します。
longreg オプションを指定した場合、レジスタを使用して引数を渡します。
本オプションの省略時解釈は、nolongreg です。

備　考　本オプションは、CPU に H8/300 を指定したときのみ、指定が可能です。
CPU が H8/300 以外の時は、4 バイトのデータは常に割り付きます。

double→float 変換

DOuble=Float

CPU [double 型の変数/引数を float 型として扱う]

書　式　DOuble=Float

説　明　double(倍精度浮動小数点)型の変数／数値を float(単精度浮動小数点)型としてオブジェクトを生成します。

スタック計算サイズ指定

STAck

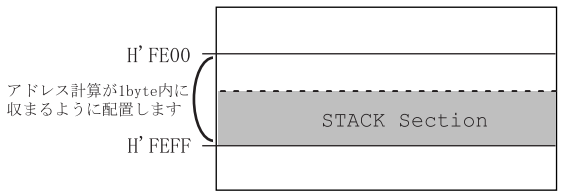
CPU [スタック計算時のサイズ :]

書　式　　STAck = { Small | Medium | Large }

説　明　　スタック計算サイズを指定します。
stack=small オプションを指定した場合、スタックアドレス計算を最下位 1 バイトだけで行います。上位バイトへの桁上がりをしません。
同様に stack=medium ではスタックアドレス計算を最下位 2 バイトだけで行います。上位バイトへの桁上がりをしません。
また、stack=large ではスタックアドレス計算を 4 バイトで行います。
本オプション省略時解釈は、stack=medium です。

備　考　　・本オプションはプログラム全体で同一オプションを指定してください。
・指定されたスタック計算サイズより大きいスタックサイズを使用した場合、または 1 バイト、2 バイトおよび 4 バイトの境界値を越えて変数が配置された場合、コンパイラではエラーメッセージやウォーニングメッセージは出力しませんが、リンケージエディタでウォーニングメッセージを出力します。
その場合、スタック計算サイズ指定を大きくしてください。

例:
```
-stack=small
```



実行時型情報

RTti

CPU [C++の dynamic_cast、typeid を有効にする]

書　式　　RTti = { ON | OFf }

説　明　　実行時型情報の有効/無効を指定します。
rtti=on を指定した場合、dynamic_cast、typeid を有効にします。
rtti=off を指定した場合、dynamic_cast、typeid を無効にします。
本オプション省略時解釈は、rtti=off です。

備　考　　本オプションを指定して作成したオブジェクトファイルをライブラリに登録したり、リロケータブルオブジェクトファイルに出力しないでください。シンボルの二重定義エラーや未定義エラーになることがあります。

例外処理機能

EXception
NOEXception

CPU [C++の try、throw、catch を有効にする]

書　式　EXception
NOEXception

説　明　noexception オプションは、C++例外処理機能を無効にします。
exception オプションは、C++例外処理機能(try, catch, throw)を有効にします。
例外処理機能を使用した場合、コード性能が低下する可能性があります。
本オプション省略時解釈は、noexception です。
exception オプションと ecpp オプションを同時に指定することはできません。

構造体、共用体、クラスメンバの境界調整数

PAck

CPU [メンバの境界調整数を 1 とする]

書　式　PAck = { 1 | 2 }

説　明　構造体、共用体、クラスメンバの境界調整数を指定します。
構造体メンバの境界調整数は、#pragma　pack 拡張子でも指定できます。オプションと #pragma 拡張子の両方が指定された場合には、拡張子の指定を優先します。
構造体、共用体、クラスの境界調整数は、メンバの最大の境界調整数と同じになります。
詳細は**「**10.1.2(2)　**複合型（C 言語）、クラス型（C++言語）」**を参照してください。
本オプションの省略時解釈は、pack=2 です。

備　考　・pack オプション指定時の構造体メンバの境界調整数を**表** 2.11 に示します。

表 2.11　pack **オプション指定時の構造体、共用体、クラスメンバの境界調整数**

メンバの型	pack=1	pack=2	**指定なし**
[unsigned] char	**1**	**1**	**1**
[unsigned] short、[unsigned] int、[unsigned] long、**浮動小数点型、ポインタ型**	**1**	**2**	**2**
境界調整数が 1 の構造体、共用体、クラス	**1**	**1**	**1**
境界調整数が 2 の構造体、共用体、クラス	**1**	**2**	**2**

・pack=1 オプションまたは#pragma pack 1 を指定した構造体、共用体、クラスのメンバは、ポインタを用いてアクセスできません（ポインタを用いたメンバ関数内でのアクセスを含みます）。

例：（cpu=2600a および pack=1 指定時）

```
struct S {
   char x;
   int y;
} s;
int *p=&s.y; // s.y のアドレスは奇数になることがあります
void test()
{
   s.y=1;    // 正しくアクセスできます
   *p =1;    // 正しくアクセスできません
}
```

8bit 絶対領域アドレス値指定

SBr

CPU [SBR 値 :]

書　式　SBr = <アドレス>

説　明　本オプションは、8 ビット絶対アドレス領域として設定するアドレスを指定します。
SBR=<アドレス>オプションを指定した場合、<アドレス>から 1 バイト分の領域をを 8 ビット絶対アドレス領域として使用します。アドレスは 16 進数で指定して下さい。下位 8 ビットは無視されます。

備　考　本オプションは、CPU 種別が H8SX の場合にのみ有効です。
<アドレス>にはデータ領域として使用できるアドレスを指定してください。
本オプション指定が省略された場合には、<アドレス>にデフォルトの 8 ビット絶対アドレスが指定されたものとして扱われます。8 ビット絶対アドレスについては**「18.3 短絶対アドレスのアクセス範囲」**を参照してください。

例　`ch38 –sbr=A0000 test.c`　8 ビット絶対アドレス領域が 0xA0000 番地から始まると仮定してコンパイルします。

2.2.7 残りのオプション

表 2.12 残りのオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	C/C++言語の選択	LANg = { C | CPp }	ー (拡張子で判断)	C プログラムとしてコンパイル C++プログラムとしてコンパイル
2	コピーライト出力抑止	LOGO NOLOGO	ー (常に nologo が有効)	コピーライトを出力します コピーライトの出力を抑止します
3	文字列内の文字コード	EUc SJis LATin1	ー	euc コードを選択 sjis コードを選択 latin1 コードを選択
4	オブジェクトコード内漢字変換	OUtcode = { Euc | Sjis }	ー	euc コード sjis コード
5	サブコマンドファイルの選択	SUbcommand = <ファイル名>	ー	<ファイル名>で指定したファイルからコマンドオプションを取りこむ

C/C++言語の選択

LANg

なし（常に拡張子で判断）

書 式　`LANg = { C | CPp }`

説 明　ソースプログラムの言語を指定します。
`lang=c` オプションを指定すると、C プログラムとしてコンパイルします。
`lang=cpp` オプションを指定すると、C++プログラムとしてコンパイルします。
本オプションを省略した場合は、ソースプログラムの拡張子によって判断します。拡張子が c のときには C プログラムとしてコンパイルします。また、拡張子が cpp、cc、cp のときには C++プログラムとしてコンパイルします。ソースプログラムの拡張子を指定しなかった場合は、C プログラムとしてコンパイルします。

例

```
ch38 test.c              C プログラムとしてコンパイルします。
ch38 test.cpp            C++プログラムとしてコンパイルします。
ch38 –lang=cpp test.c    C++プログラムとしてコンパイルします。
ch38 test                test.c を仮定し、C プログラムとしてコンパイルします。
```

備 考　`lang=c` オプションを指定したとき、`ecpp` オプションが無効になります。

コピーライト出力抑止

LOGO
NOLOGO

なし(常に nologo が有効)

書　式　LOGO
NOLOGO

説　明　コピーライトの出力を抑止します。
logo を指定した場合、コピーライト表示が出力されます。
nologo を指定した場合、コピーライトの表示の出力が抑止されます。
本オプション省略時解釈は、logo です。

文字列内の文字コード

EUc
SJis
LATin1

なし

書　式　EUc
SJis
LATin1

説　明　文字列、文字定数およびコメント内に日本語または ISO-Latin1 コードを記述できます。
ホストマシンと文字列内コードとの関係を**表** 2.13 に示します。

表 2.13 **ホストマシンと文字列内コード**

ホストマシン	オプション指定			
	euc	sjis	latin1	指定なし
PC	euc	sjis	latin1	sjis
SPARC	euc	sjis	latin1	euc
HP9000/700	euc	sjis	latin1	sjis

備　考　latin1 オプションを指定したとき、outcode オプションが無効になります。

オブジェクトコード内漢字変換

OUtcode

なし

書　式　　OUtcode = { Euc | Sjis }

説　明　　文字列、文字定数内に日本語を記述したときに、オブジェクトプログラムに出力する漢字コードを指定します。
outcode = euc オプションは、漢字コードを EUC コードで出力します。
outcode = sjis オプションは、漢字コードを SJIS コードで出力します。
ソースプログラム上の漢字コードは、euc または sjis オプションで指定できます。

サブコマンドファイルの選択

SUbcommand

なし

書　式　　SUbcommand ＝ <サブコマンドファイル名>

説　明　　subcommand オプションは、コンパイラ起動時のコンパイラオプションをサブコマンドファイルで指定します。サブコマンドファイル中の書式は、コマンドラインの書式と同一です。

例　　opt.sub　　：-show＝object -debug –byteenum
コマンドライン指定　：ch38 -cpu＝2600a -subcommand＝opt.sub test.c
コンパイラ解釈　　：ch38 -cpu＝2600a -show＝object -debug -byteenum test.c

3. アセンブラ操作方法

3.1 オプション指定規則

アセンブラを起動するコマンドラインの形式は以下のとおりです。

asm38 [△<**オプション**> …] [△<**ファイル名**> [, …*]] [△<**オプション**> …]
<**オプション**>：-<**オプション**> [=<**サブオプション**> [, …]]

【注】* 複数のソースファイル名を指定すると、それらのファイルを指定の順に連結したものがアセンブル処理の単位になります。この場合、.END **アセンブラ制御命令は最後のファイルにだけ記述してください。**

3.2 オプション解説

コマンドライン形式の英大文字は短縮形指定時の文字を、下線は省略時解釈を示します。
また、統合開発環境に対応するダイアログメニューを、タブ名<カテゴリ名>[項目]…で示します。
オプションの順序は、統合開発環境のタブとその中のカテゴリに対応しています。

3.2.1 ソースオプション

表 3.1 **ソースカテゴリオプション一覧**

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	**インクルードファイルディレクトリ**	Include = <**パス名**>[, …]	**アセンブラ<ソース>[オプション項目** :] [**インクルードファイルディレクトリ**]	**インクルードファイルの取り込み先を指定します。**
2	**置換シンボルの定義**	DEFine = <sub>[, …] <sub>： <**置換シンボル**> = "<**文字列**>"	**アセンブラ<ソース>[オプション項目** :] [**シンボル定義**]	**文字列の置き換えを定義します。**
3	**整数型プリプロセッサ変数の定義**	ASsignA = <sub>[, …] <sub>： <**変数名**> = <**整数定数**>	**アセンブラ<ソース>[オプション項目** :] [**プリプロセッサ変数定義**]	**整数型のプリプロセッサ変数を定義します。**
4	**文字型プリプロセッサ変数の定義**	ASsignC = <sub>[, …] <sub>： <**変数名**> = "<**文字列**>"	**アセンブラ<ソース>[オプション項目** :] [**プリプロセッサ変数定義**]	**文字型のプリプロセッサ変数を定義します。**

インクルードファイルディレクトリ

Include

アセンブラ<ソース>[オプション項目 :][インクルードファイルディレクトリ]

書　式　　Include=<パス名>［,...］

説　明　　include オプションは、インクルードするファイルのディレクトリ名を指定します。
ディレクトリ名はホストマシンの標準的な指定方法に従います。
ディレクトリ名の指定数はコマンドラインで１行入力可能な限り有効です。
検索の優先度はまずカレントディレクトリ、続いて include オプションで指定したディレクトリを指定した順序にしたがって検索します。

例　　asm38　aaa.mar -include=¥usr¥tmp,¥tmp

aaa.mar 内で.INCLUDE "file.h"を指定の場合、file.h をカレントディレクトリ、¥usr¥tmp、¥tmp の順にサーチします。

備　考　　アセンブラ制御文との関係

オプション	制御文	結　果
include	**（指定に関わらず）**	.INCLUDE **制御命令で指定したディレクトリ**
		include **オプションで指定したディレクトリ***
（指定なし）	.INCLUDE <**ファイル名**>	.INCLUDE **制御命令で指定したディレクトリ**

【注】* .INCLUDE **制御命令で指定したディレクトリ文字列の前に** include **オプションで指定したディレクトリ文字列を付加したディレクトリ名を使用します。**

置換シンボルの定義

DEFine

アセンブラ<ソース>[オプション項目 :][シンボル定義]

書　式　　DEFine = <sub>[,...]
　　<sub>：<置換シンボル>＝"<文字列>"

説　明　　define オプションは、プリプロセッサで置換シンボルを対応する文字列に置き換えます。define と assignc の各オプションの機能の違いは.DEFINE と.ASSIGNC の機能の違いに対応します。

備　考　　アセンブラ制御文との関係

オプション	制御文	結　果
define	.DEFINE*	define **オプションで指定した文字列**
	（指定なし）	define **オプションで指定した文字列**
（指定なし）	.DEFINE	.DEFINE **制御命令で指定した文字列**

【注】* define **オプションで置換シンボルに文字列を設定した場合、当該置換シンボルへの**.DEFINE **による定義がすべて無効になります。**

整数型のプリプロセッサ変数の定義

ASsignA

アセンブラ<ソース>[オプション項目 :][プリプロセッサ変数定義]

書　式　　ASsignA= <sub>[,...]
　　<sub>：<プリプロセッサ変数名>＝<整数定数>

説　明　　assigna オプションは、プリプロセッサ変数に整数定数を設定します。
プリプロセッサ変数名の書き方はシンボル名の書き方と同じです。
整数定数は基数（B'、Q'、D'、H'）と数値を組み合わせて指定します。基数を省略し、数値のみを限定した場合は 10 進数として扱います。
整数定数に指定できる値の範囲は-2,147,483,648～4,294,967,295 です。ただし、負の値を設定する場合は 10 進以外の基数で指定してください。

例　　asm38 aaa.mar -assigna=_$=H'FF
プリプロセッサ変数_$に値 H'FF を設定します。ソースプログラム内のプリプロセッサ変数_$のすべての参照箇所¥&_$を H'FF に設定します。

備　考　　ホスト OS が UNIX で、プリプロセッサ変数名に$がある場合、もしくは基数表示のアポストロフィがある場合は、直前にバックスラッシュ(円記号 "¥")を指定します。

アセンブラ制御文との関係

オプション	制御文	結　果
assigna	.ASSIGNA*	assigna **オプションで指定した値**
	（指定なし）	assigna **オプションで指定した値**
（指定なし）	.ASSIGNA	.ASSIGNA **制御命令で指定した値**

【注】* assigna **オプションでプリプロセッサ変数に値を設定した場合、当該プリプロセッサへの**.ASSIGNA **による定義が無効になります。**

文字型のプリプロセッサ変数の定義

ASsignC

アセンブラ<ソース>[オプション項目 :][プリプロセッサ変数定義]

書　式　　ASsignC= <sub>[,...]
　　<sub>：<プリプロセッサ変数名>＝"<文字列>"

説　明　　assignc オプションはプリプロセッサ変数に文字列を設定します。
プリプロセッサ変数名の書き方はシンボル名の書き方と同じです。
文字列は文字をダブルコーテーション（"）で囲んで指定します。
文字列には 255 文字まで指定できます。

例　　asm38　aaa.mar -assignc=_$="ON!OFF"

プリプロセッサ変数_$に文字列 ON!OFF を設定します。ソースプログラム内のプリプロセッサ変数_$のすべての参照箇所¥&_$を文字列 ON!OFF に設定します。

備　考　　ホスト OS が UNIX の場合は文字列の中に次の文字を指定する際､直前にバックスラッシュ(円記号 "¥")を指定します。また、前後に文字列を指定する場合は前後の文字列をダブルコーテーション（"）で囲みます。
・イクスクラメーション（!）
・ダブルコーテーション（"）
・ドル（$）
・逆コーテーション（`）

アセンブラ制御文との関係

オプション	制御文	結　果
assignc	.ASSIGNC*	assignc **オプションで指定した文字列**
	（指定なし）	assignc **オプションで指定した文字列**
（指定なし）	.ASSIGNC	.ASSIGNC **制御命令で指定した文字列**

【注】* assignc **オプションでプリプロセッサ変数に文字列を設定した場合、当該プリプロセッサ変数への**.ASSIGNC **による定義がすべて無効になります。**

3.2.2 オブジェクトオプション

表 3.2 オブジェクトカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	デバッグ情報の出力制御	Debug NODebug	アセンブラ <オブジェクト> [デバッグ情報出力 :]	デバッグ情報の出力を制御します。
2	プリプロセッサの展開結果出力	EXPand [= <出力ファイル名>]	アセンブラ <オブジェクト> [プリプロセッサ展開結果出力]	プリプロセッサの展開結果を出力します。
3	最適化の指定	OPtimize NOOPtimize	アセンブラ <オブジェクト> [最適化]	最適化を行います。 最適化を行いません。
4	ディスプレースメントサイズの設定	BR_relative = < sub > < sub > ： { 8 | 16 }	アセンブラ <オブジェクト> [ディスプレースメントサイズ設定 :]	分岐命令のディスプレースメントのデフォルトサイズを設定します。 8bit に設定します。 16bit に設定します。
5	モジュール間最適化	GOptimize	アセンブラ <オブジェクト> [モジュール間最適化]	モジュール間最適化用付加情報を出力します。
6	オブジェクトモジュールの出力制御	Object [= <出力ファイル名>] NOObject	アセンブラ <オブジェクト> [オブジェクト出力ディレクトリ :]	オブジェクトモジュールの出力を制御します。

デバッグ情報の出力制御

Debug
NODebug

アセンブラ<オブジェクト>[デバッグ情報出力 :]

書　式　　Debug
NODebug

説　明　　debug オプションは、デバッグ情報を出力します。
nodebug オプションは、デバッグ情報を出力しません。
debug、nodebug 各オプションによる指定は、オブジェクトモジュールを出力する場合に限り有効です。

備　考　　デバッグ情報はデバッガでプログラムをデバッグするのに必要です。ソースプログラムの行に関する情報やシンボルに関する情報（シンボルデバッグ情報）などを含みます。

アセンブラ制御命令との関係（アセンブラはオプションによる指定を優先します）

オプション	制御命令	結　果（オブジェクトモジュール出力時）
debug	（指定に関わらず）	デバッグ情報を出力する。
nodebug	（指定に関わらず）	デバッグ情報を出力しない。
（指定なし）	.OUTPUT DBG	デバッグ情報を出力する。
	.OUTPUT NODBG	デバッグ情報を出力しない。
	（指定なし）	デバッグ情報を出力しない。

プリプロセッサの展開結果を出力

EXPand

アセンブラ<オブジェクト>[プリプロセッサ展開結果出力]

書　式　　EXPand [=<出力ファイル名>]

説　明　　expand オプションは、マクロ展開、条件つきアセンブル、構造化アセンブル、ファイルのインクルードを行った後のアセンブラソースを出力します。
本オプションを指定するとオブジェクトの生成は行いません。
出力ファイルの指定を省略すると次のようになります。
・ファイル拡張子の指定を省略した場合
　ファイル型は exp になります。
・主ファイル名、ファイル拡張子ともに指定を省略した場合
　主ファイル名は入力ソースファイル（1 つめに指定したもの）と同じとなります。
　また、ファイル拡張子は exp になります。

備　考　　入力ソースファイルと出力ファイルに、同じファイル名を指定しないでください。

OPtimize
NOOPtimize

アセンブラ<オブジェクト>[最適化]

書　式　　OPtimize
NOOPtimize

説　明　　optimize オプションは、PC 相対形式、ディスプレースメントつきレジスタ間接のディスプレースメントサイズと絶対アドレス形式のアドレスサイズの最適化、最適化抑止を指定します。ただし、H8SX の MOVA 命令では下記表のようになります。

第１オペランド	最適化可否
@(disp,Reg) *1	○
@(disp,@ERn.sz) *2	○
@(disp,@±ERn.sz) *2	○
@(disp,@ERn±.sz) *2	○
@(disp,@(disp,Reg).sz) *2 *3	×
@(disp,@abs.sz) *2	×

【注】*1　Reg は RnL.B, RnH.B, Rn.W, En.W のいずれの場合も該当します。
*2　sz は B または W のいずれの場合も該当します。
*3　Reg は ERn, RnL.B, Rn.W, ERn.L のいずれの場合も該当します。

本オプションの対象となるのは、ディスプレースメントサイズ(：8,：16)、絶対アドレスの確保サイズ(:8, :16, :24, :32)の指定がない実行命令です。
PC 相対形式のディスプレースメントの値によってディスプレートメントサイズを次のように設定します。

H8S/2600 アドバンストモードで最適化を指定しない場合

ディスプレースメント値		ディスプレースメントサイズ
絶対値	(-32,768〜32,767)	16 **ビット***
相対値		16 **ビット**
外部参照値		16 **ビット**

【注】* 命令より後に定義した絶対シンボルを参照した場合に限ります。

H8S/2600 アドバンストモードで最適化を指定した場合

ディスプレースメント値		ディスプレースメントサイズ
絶対値	(-128〜127)	8 **ビット**
	(-32,768〜-129) (128〜32,767)	16 **ビット**
相対値		16 **ビット**
外部参照値		16 **ビット**

例　　`asm38 aaa.mar –optimize`
　オブジェクトモジュールの最適化を行います。

`asm38 aaa.mar`
　オブジェクトモジュールの最適化を行いません。

備　考　　アセンブラ制御命令との関係（アセンブラはオプションによる指定を優先します）

オプション 1	オプション 2	制御命令	結　果
optimize	(指定に関わらず)	(指定に関わらず)	最適化されたビット数
nooptimize	br_relative	(指定に関わらず)	br_relative オプションで指定されたビット数
	(指定なし)	.DISPSIZE	.DISPSIZE 制御命令のビット数
		(指定なし)	8 ビット

【注】* optimize オプションは、オブジェクトモジュールの出力に関するオプション(br_relative)、制御命令(.DISPSIZE)より優先します。

ディスプレースメントサイズの設定

BR_relative

アセンブラ<オブジェクト>[ディスプレースメントサイズ設定 :]

書　式　　BR_relative = { 8 | 16 }

説　明　　分岐命令のディスプレースメントが、前方命令である場合のディスプレースメントのデフォルトサイズを指定します。
・ 8 … デフォルトサイズを 8 ビットに指定します。
・16 … デフォルトサイズを 16 ビットに指定します。
本オプションの対象となるのは、ディスプレースメントサイズ(:8, :16)の指定があり、かつ、optimize オプションの指定がない場合のディスプレースメントサイズです。

備　考　　本オプションは、H8/300、H8/300L では、br_relative=8 で固定なので意味を持ちません。

アセンブラ制御命令との関係（アセンブラはオプションによる指定を優先します）

オプション 1	**オプション** 2	**制御命令、**CPU **種別**	**結　果**
optimize	(**指定に関わらず**)	(**指定に関わらず**)	**最適化されたビット数**
nooptimize	br_relative	(**指定に関わらず**)	br_relative **オプションで指定したビット数**
	(**指定なし**)	.DISPSIZE	.DISPSIZE **制御命令で指定したビット数**
		(**指定なし**) cpu= 300、300L、300HN、2000N、2600N、H8SXN	8 **ビット**
		(**指定なし**) cpu= 300HA、2000A、2600A、H8SXM、H8SXA、H8SXX	16 **ビット**

【注】* optimize **オプションは、オブジェクトモジュールの出力に関するオプション**(br_relative)**、制御命令**(.DISPSIZE)**より優先します。**

モジュール間最適化

GOptimize

アセンブラ<オブジェクト>[モジュール間最適化]

書　式　　GOptimize

説　明　　モジュール間最適化用付加情報を出力します。
本オプションを指定したファイルは、リンク時にモジュール間最適化の対象になります。

Object
NOObject

アセンブラ<オブジェクト>[オブジェクト出力ディレクトリ :]

書　式　　Object［=<出力オブジェクトファイル名>］
NOObject

説　明　　object オプションは、オブジェクトファイルを出力します。
noobject オプションは、オブジェクトファイルを出力しません。
出力オブジェクトファイルの指定を省略すると次のようになります。
・ファイル拡張子の指定を省略した場合
　ファイル拡張子は obj になります。
・主ファイル名、ファイル拡張子ともに指定を省略した場合
　主ファイル名は入力ソースファイル（１つめに指定したもの）と同じになります。
　また、ファイル拡張子は obj になります。

備　考　　アセンブラ制御命令との関係（アセンブラはオプションによる指定を優先します）

オプション	制御命令	結　果
object	**（指定に関わらず）**	**オブジェクトファイルを出力する。**
noobject	**（指定に関わらず）**	**オブジェクトファイルを出力しない。**
（指定なし）	.OUTPUT OBJ	**オブジェクトファイルを出力する。**
	.OUTPUT NOOBJ	**オブジェクトファイルを出力しない。**
	（指定なし）	**オブジェクトファイルを出力する。**

入力ソースファイルと出力オブジェクトファイルに、同じファイル名を指定しないでください。同じファイル名を指定した場合、入力ソースファイルが上書きされます。

3.2.3 リストオプション

表 3.3 リストカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	アセンブルリストの出力制御	LISt [= <出力ファイル名>] NOLISt [= <出力ファイル名>]	アセンブラ<リスト>[アセンブルリスト出力]	アセンブルリストの出力を制御します。
2	ソースプログラムリストの出力制御 *	SOurce NOSOurce	アセンブラ<リスト>[アセンブルリスト出力] [ソースプログラム :]	ソースプログラムリストの出力を制御します。
3	ソースプログラムリストの部分出力の制御 *	SHow [= <出力種別>[, …]] NOSHow [= <出力種別>[, …]] <出力種別>： {CONditionals | Definitions | CAlls | Expansions | Structured | CODe }	アセンブラ<リスト>[ソースプログラムリスト部分出力 :] [条件つき不成立] [定義] [コール] [展開] [構造化展開] [オブジェクトコード表示行]	ソースプログラムリストの部分出力を制御します。
4	クロスリファレンスリストの出力制御 *	CRoss_reference NOCRoss_reference	アセンブラ<リスト>[アセンブルリスト出力] [クロスリファレンス :]	クロスリファレンスリストの出力を制御します。
5	セクション情報リストの出力制御 *	SEction NOSEction	アセンブラ<リスト>[アセンブルリスト出力] [セクション :]	セクション情報リストの出力を制御します。

【注】* source / nosource , show / noshow , cross_reference / nocross_reference , section / nosection の各オプションは list オプションを指定した時のみ有効となります。

LISt
NOLISt

アセンブラ<リスト>[アセンブルリスト出力]

書　式　　LISt [=<出力リストファイル名>]
NOLISt [=<出力リストファイル名>]

説　明　　list オプションを指定した場合、アセンブルリストを出力します。
出力リストファイル名の指定を省略すると次のようになります。
・ファイル拡張子の指定を省略した場合
　ファイル拡張子は lis になります。
・主ファイル名、ファイル拡張子ともに指定を省略した場合
　主ファイル名は入力ソースファイル（１つめに指定したもの）と同じとなります。
　また、ファイル拡張子は lis になります。

nolist オプションを指定した場合、アセンブルリストを出力しません。
nolist でファイル名を指定した場合は、エラーが発生した行だけのアセンブルリストを
ファイルに出力します。

備　考　　アセンブラ制御命令との関係（アセンブラはオプションによる指定を優先します）

オプション	制御命令	結　果
list	**(指定に関わらず)**	**アセンブルリストを出力する。**
nolist	**(指定に関わらず)**	**アセンブルリストを出力しない。**
(指定なし)	.PRINT LIST	**アセンブルリストを出力する。**
	.PRINT NOLIST	**アセンブルリストを出力しない。**
	(指定なし)	**アセンブルリストを出力しない。**

入力ソースファイルと出力リストファイルに、同じファイル名を指定しないでください。
同じファイル名を指定した場合、入力ソースファイルが上書きされます。

ソースプログラムリストの出力制御

SOurce
NOSOurce

アセンブラ<リスト>[アセンブルリスト出力][ソースプログラム :]

書　式　　SOurce
NOSOurce

説　明　　source オプションは、アセンブルリストにソースプログラムリストを付加します。
nosource オプションは、アセンブルリストにソースプログラムリストを付加しません。
source、nosource による指定はアセンブルリストを出力する場合に限り有効です。

備　考　　アセンブラ制御命令との関係（アセンブラはオプションによる指定を優先します）

オプション	制御命令	結　果（アセンブルリスト出力時）
source	(指定に関わらず)	ソースプログラムリストを出力する。
nosource	(指定に関わらず)	ソースプログラムリストを出力しない。
(指定なし)	.PRINT SRC	ソースプログラムリストを出力する。
	.PRINT NOSRC	ソースプログラムリストを出力しない。
	(指定なし)	ソースプログラムリストを出力する。

ソースプログラムリストの部分出力制御

SHow
NOSHow

アセンブラ<リスト>[ソースプログラムリスト部分出力 :][条件つき不成立][定義][コール][展開][構造化展開][オブジェクトコード表示行]

書　式　　SHow [＝<出力種別>[,…]]
NOSHow [＝<出力種別>[,…]]
出力種別：｛CONditionals｜Definitions ｜CAlls ｜
Expansions ｜Structured ｜CODe ｝

説　明　　ソースプログラムリストのプリプロセッサ機能のソースステートメント部分出力、出力抑止、オブジェクトコード表示行の部分出力、出力抑止を指定します。
出力種別で指定した項目を出力、出力抑止します。出力種別を省略した場合は、全ての項目を出力、出力抑止します。
・show　‥‥　出力
・noshow ‥‥　出力抑止

出力種別の内容は次のとおりです。

出力種別	意　味	内　容
conditionals	条件つき不成立	.AIF, .AIFDEF の不成立部分
definitions	定義	マクロ定義部分
		.AREPEAT, .AWHILE 定義部分
		.INCLUDE 制御文
		.ASSIGNA, .ASSIGNC 制御文
calls	コール	マクロコール文
		.AIF, .AIFDEF, .AENDI 制御文
		構造化アセンブリ制御文
expansions	展開	マクロ展開部分
		.AREPEAT, .AWHILE 展開部分
structured	構造化展開	構造化アセンブリ展開部分
code	オブジェクトコード表示行	制御命令のオブジェクトコード表示が、ソーステートメントの行数を超える部分

備　考　show,noshow による指定はアセンブルリストを出力する場合に限り有効です。
PC 版の場合、出力種別を２つ以上指定する時はカッコ()で囲んで指定してください。

アセンブラ制御命令との関係（アセンブラはオプションによる指定を優先します）

オプション	制御命令	結　果
show=出力種別	(指定に関わらず)	出　力
noshow=出力種別	(指定に関わらず)	出力抑止
(指定なし)	.LIST 出力種別（出力）	出　力
	.LIST 出力種別（出力抑止）	出力抑止
	(指定なし)	出　力

CRoss_reference
NOCRoss_reference

アセンブラ<リスト>[アセンブルリスト出力][クロスリファレンス :]

書　式　　CRoss_reference
NOCRoss_reference

説　明　　cross_reference オプションは、アセンブルリストにクロスリファレンスリストを付加します。
nocross_reference オプションは、アセンブルリストにクロスリファレンスリストを付加しません。
cross_reference、nocross_reference による指定は、アセンブルリストを出力する場合に限り有効です。

備　考　　アセンブラ制御命令との関係（アセンブラはオプションによる指定を優先します）

オプション	制御命令	結　果（アセンブルリスト出力時）
cross_reference	(**指定に関わらず**)	**クロスリファレンスリストを出力する。**
nocross_reference	(**指定に関わらず**)	**クロスリファレンスリストを出力しない。**
(**指定なし**)	.PRINT　CREF	**クロスリファレンスリストを出力する。**
	.PRINT　NOCREF	**クロスリファレンスリストを出力しない。**
	(**指定なし**)	**クロスリファレンスリストを出力する。**

セクション情報リストの出力制御

SEction
NOSEction

アセンブラ<リスト>[アセンブルリスト出力][セクション :]

書　式　　SEction
NOSEction

説　明　　section オプションは、アセンブルリストにセクション情報リストを付加します。
nosection オプションは、アセンブルリストにセクション情報リストを付加しません。
section、nosection による指定はアセンブルリストを出力する場合に限り有効です。

備　考　　アセンブラ制御命令との関係（アセンブラはオプションによる指定を優先します）

オプション	制御命令	結　果（アセンブルリスト出力時）
section	(指定に関わらず)	セクション情報リストを出力する。
nosection	(指定に関わらず)	セクション情報リストを出力しない。
(指定なし)	.PRINT SCT	セクション情報リストを出力する。
	.PRINT NOSCT	セクション情報リストを出力しない。
	(指定なし)	セクション情報リストを出力する。

3.2.4　チューニングオプション

表 3.4　チューニングカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	8 または 16 ビット絶対アドレス形式シンボルの指定	ABS8 ABS16	アセンブラ <チューニング> [絶対アドレス形式 :]	8 または 16 ビット絶対アドレス形式でアクセスするシンボルを指定します。

8 または 16 ビット絶対アドレスの指定

ABS8
ABS16

アセンブラ<チューニング>[絶対アドレス形式 :]

書　式　　ABS8[= <シンボル>[,…]]
ABS16[= <シンボル>[,…]]

説　明　　abs8 オプションは、8 ビット絶対アドレス形式でアクセスするシンボルを指定します。
abs16 オプションは、16 ビット絶対アドレス形式でアクセスするシンボルを指定します。
シンボル省略時は、全ての外部参照／定義シンボルを対象とします。
同じシンボルに対して abs8／abs16 オプションを同時に指定した場合、後ろの指定を優先します。

・-abs8 –abs16 と指定した場合
　すべての外部シンボルは 16 ビット絶対アドレス形式になります。
・-abs8=<sym> –abs16=<sym>と指定した場合
　<sym>は 16 ビット絶対アドレス形式、その他は CPU によって決定されます。

ただし、シンボル指定とシンボル省略を指定した場合、扱いが異なります。

・-abs8=<sym> -abs16 と指定した場合
　<sym>のみ 8 ビット絶対アドレス形式、その他は 16 ビット絶対アドレス形式になります。

アクセスサイズの優先順位

優先順位		アクセスサイズの形式
高	1	**絶対アドレス形式の確保サイズ**
↑↓	2	.IMPORT／.EXPORT／.GLOBAL **制御命令のアクセスサイズ** .ABS8／.NOABS8 **制御命令**
低	3	abs8／abs16 **オプション**

例　　
```
asm38 aaa.mar –abs8=sym1 –abs16
```
絶対アドレス形式において、外部シンボルを指定する場合、sym1 は 8 ビット絶対アドレス形式、他の外部シンボルは 16 ビット絶対アドレス形式でアクセスします。
```
asm38 aaa.mar –abs8=sym1 –abs16=sym2,sym3,sym4
```
aaa.mar の内容
```
  .CPU     2600A
  .IMPORT  sym1,sym2,sym3,sym5
  .IMPORT  sym4:8
  MOV.B    @sym1  ,R1H        ; 8 ビット    (-abs8 指定)
  MOV.B    @sym2  ,R1H        ;16 ビット    (-abs16 指定)
  MOV.B    @sym3:8,R1H        ; 8 ビット    (確保サイズ指定)
  MOV.B    @sym4  ,R1H        ; 8 ビット    (.IMPORT のアクセスサイズ指定)
  MOV.B    @sym5  ,R1H        ;32 ビット    (指定なし)
  MOV.B    @(sym1+sym2),R1H   ; 8 ビット*   (-abs8,-abs16 混在指定)
```
【注】* 絶対アドレス形式に外部シンボルを複数記述した場合、アクセスサイズは最小のアクセスサイズを適用します。

3.2.5 その他オプション

表 3.5 その他カテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	**未参照外部参照シンボル情報の出力抑止**	Exclude NOExclude	**アセンブラ<その他>[その他のオプション :][未定義外部参照シンボル情報の出力抑止]**	**未参照外部参照シンボルのシンボル情報の出力、出力抑止を指定します。**

未参照シンボルの情報の出力抑止

Exclude
NOExclude

アセンブラ<その他>[その他のオプション :][未定義外部参照シンボル情報の出力抑止]

書 式

```
Exclude
NOExclude
```

説 明

exclude オプションは、未参照外部参照シンボルのシンボル情報を出力しません。
noexclude オプションは、未参照外部参照シンボルのシンボル情報を出力します。
未参照外部参照シンボルのシンボル情報を出力抑止することにより、オブジェクトモジュールのサイズを小さく出来ます。

例

```
asm38 aaa.mar –exclude
```
未参照外部参照シンボルのシンボル情報を出力しません。

```
asm38 aaa.mar –noexclude
```
未参照外部参照シンボルのシンボル情報を出力します。

3.2.6 CPU オプション

表 3.6 CPU タブオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	CPU 種別の指定	CPu = {H8SXN[:{M|D|MD}] | H8SXM[:<ビット幅>][:{M|D|MD}] | H8SXA[:<ビット幅>][:{M|D|MD}] | H8SXX[:<ビット幅>][:{M|D|MD}] | 2600N | 2600A[:<ビット幅>] | 2000N | 2000A[:<ビット幅>] | 300HN | 300HA[:<ビット幅>] | 300 | 300L }	CPU [CPU 種別 :] [乗除算器指定 :]	CPU 種別を指定します。
2	8 ビット短絶対領域の基点の指定	SBR	CPU [SBR 値 :]	8 ビット短絶対領域の基点を指定します。

CPU 種別の指定

CPu

CPU [CPU 種別 :][乗除算器指定 :]

```
書 式      CPu = { H8SXN [:{M|D|MD}]                              |
                   H8SXM [:<アドレス空間のビット幅>][:{M|D|MD}]  |
                   H8SXA [:<アドレス空間のビット幅>][:{M|D|MD}]  |
                   H8SXX [:<アドレス空間のビット幅>][:{M|D|MD}]  |
                   2600N                                          |
                   2600A [:<アドレス空間のビット幅>]              |
                   2000N                                          |
                   2000A [:<アドレス空間のビット幅>]              |
                   300HN                                          |
                   300HA [:<アドレス空間のビット幅>]              |
                   300 | 300L                                     }
```

説　明　　作成するオブジェクトプログラムの CPU 種別と動作モード、アドレス空間のビット幅、乗除算器の有無を指定します。
サブオプションは次のようになります。

サブオプション名	意　味
H8SXN[:{M|D|MD}]	H8SX 用ノーマルモードのオブジェクトを作成します。 乗除算器の指定ができます。
H8SXM [：<アドレス空間のビット幅>] [:{M|D|MD}]	H8SX 用ミドルモードのオブジェクトを作成します。 〈アドレス空間のビット幅〉は、20、24 のいずれかの数値で、それぞれ 1M バイト、16M バイトのアドレス空間を示します。 〈アドレス空間のビット幅〉の省略時解釈は 24 です。 乗除算器の指定ができます。
H8SXA [：<アドレス空間のビット幅>] [:{M|D|MD}]	H8SX 用アドバンストモードのオブジェクトを作成します。 〈アドレス空間のビット幅〉は、20、24、28、32 のいずれかの数値で、それぞれ 1M バイト、16M バイト、256M バイト、4G バイトのアドレス空間を示します。 〈アドレス空間のビット幅〉の省略時解釈は 24 です。 乗除算器の指定ができます。
H8SXX [：<アドレス空間のビット幅>] [:{M|D|MD}]	H8SX 用マキシマムモードのオブジェクトを作成します。 〈アドレス空間のビット幅〉は、28、32 のいずれかの数値で、それぞれ 256M バイト、4G バイトのアドレス空間を示します。 〈アドレス空間のビット幅〉の省略時解釈は 32 です。 乗除算器の指定ができます。
2600N	H8S/2600 用ノーマルモードのオブジェクトを作成します。
2600A [：<アドレス空間のビット幅>]	H8S/2600 用アドバンストモードのオブジェクトを作成します。 〈アドレス空間のビット幅〉は、20、24、28、32 のいずれかの数値で、それぞれ 1M バイト、16M バイト、256M バイト、4G バイトのアドレス空間を示します。 〈アドレス空間のビット幅〉の省略時解釈は 24 です。
2000N	H8S/2000 用ノーマルモードのオブジェクトを作成します。
2000A [：<アドレス空間のビット幅>]	H8S/2000 用アドバンストモードのオブジェクトを作成します。 〈アドレス空間のビット幅〉は、20、24、28、32 のいずれかの数値で、それぞれ 1M バイト、16M バイト、256M バイト、4G バイトのアドレス空間を示します。 〈アドレス空間のビット幅〉の省略時解釈は 24 です。
300HN	H8/300H 用ノーマルモードのオブジェクトを作成します。
300HA [：<アドレス空間のビット幅>]	H8/300H 用アドバンストモードのオブジェクトを作成します。 〈アドレス空間のビット幅〉は、20 または 24 の数値で、それぞれ 1M バイト、16M バイトのアドレス空間を示します。 〈アドレス空間のビット幅〉の省略時解釈は 24 です。
300	H8/300 のオブジェクトを作成します。
300L	H8/300L のオブジェクトを作成します。

乗算器／除算器の有無の指定は、次のようになります。

乗算器／除算器	指定方法
なし／なし	指定なし
あり／なし	M
なし／あり	D
あり／あり	MD

乗算器ありの場合の追加命令は、MAC, LDMAC, STMAC, CLRMAC, MULU/U, MULS/U です。
除算器ありの場合の追加命令はありません。

備　考　cpu オプションを省略した場合は、H38CPU 環境変数の内容を参照します。
また、cpu オプションと H38CPU 環境変数を同時に指定した場合は、cpu オプションを優先します。cpu オプションと H38CPU 環境変数の両方を省略した場合は、エラーメッセージ 933 を出力します。

アセンブラ制御命令との関係（アセンブラはオプションによる指定を優先します）

オプション	制御命令	環境変数	結　果
cpu=cpu 種別	(指定に関わらず)	(指定に関わらず)	cpu オプションで指定した cpu 種別
(指定なし)	.CPU　cpu 種別	(指定に関わらず)	cpu 制御命令で指定した cpu 種別
	(指定なし)	h38cpu=cpu 種別	環境変数の cpu 種別
		(指定なし)	エラーメッセージ 933 出力

8 ビット短絶対領域の基点の指定

SBR

CPU [SBR 値 :]

書　式　　SBR={<定数>|USER}

説　明　　SBR=<定数>は、<定数>を基点として 256 バイト分の領域を 8 ビット絶対アドレスのアクセス領域として使用します。定数は基数 H'を指定、最下位 8 ビットは 0 固定となります。
SBR=USER は、アドレス空間のビット幅により以下の基点となります。

	CPU/動作モード	8 ビット短絶対アドレス基点
H8SX マキシマムモード	H8SXX[:32]	H'FFFFFF00
	H8SXX:28	H'0FFFFF00
H8SX アドバンストモード	H8SXA:32	H'FFFFFF00
	H8SXA:28	H'0FFFFF00
	H8SXA[:24]	H'00FFFF00
	H8SXA:20	H'000FFF00
H8SX ミドルモード	H8SXM[:24]	H'00FFFF00
	H8SXM:20	H'000FFF00
H8SX ノーマルモード	H8SXN	H'0000FF00

SBR オプションを指定できるのは、CPU が H8SXN, H8SXM, H8SXA, H8SXX の場合です。

アセンブラ制御命令との関係

オプション	制御命令	8 ビット絶対アドレスのアクセス領域の基点
sbr=＜定数＞	.SBR ＜定数＞	SBR 制御命令で指定した定数
	.SBR	sbr オプションで指定した定数
	(指定なし)	sbr オプションで指定した定数
sbr=USER	.SBR ＜定数＞	SBR 制御命令で指定した定数
	.SBR	アドレス空間のビット幅により決められた値
	(指定なし)	アドレス空間のビット幅により決められた値
(指定なし)	.SBR ＜定数＞	SBR 制御命令で指定した定数
	.SBR	アドレス空間のビット幅により決められた値
	(指定なし)	アドレス空間のビット幅により決められた値

例

```
asm38 aaa.mar –sbr=H’ff0000
```

8 ビット短絶対領域を H’00ff0000～H’00ff00ff とします。

aaa.mar の内容

```
 .CPU     H8SXX:32
 MOV.L    #H’00ff0000,ER1
 LDC.L    ER1,SBR
 MOV.B    @sym1  ,R1H       ; 8 ビット    (–sbr 指定 8 ビット短絶対領域内)
 MOV.B    @sym2  ,R1H       ;16 ビット    (–sbr 指定 8 ビット短絶対領域外)
sym1:  .equ  H’00ff0040
sym2:  .equ  H’ffffff40
```

備　考　ホスト OS が UNIX の場合、定数は基数 H’のアポストロフィ直前にバックスラッシュ(円記号"¥")を指定します。

3.2.7 残りのオプション

表 3.7 残りのオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	異常終了とするエラーレベルの変更	ABort={ Warning | Error }	アセンブラ<その他>[ユーザ指定オプション :]	アセンブラが異常終了するエラーのレベルを変更します。
2	欧州コード文字	LATIN1	アセンブラ<その他>[ユーザ指定オプション :]	ソースファイル内に欧州コード文字を使えるようにします。
3	漢字コードをシフト JIS に指定	SJIS	アセンブラ<その他>[ユーザ指定オプション :]	ソースファイル内の漢字コードをシフト JIS コードとして扱います。
4	漢字コードを EUC に指定	EUC	アセンブラ<その他>[ユーザ指定オプション :]	ソースファイル内の漢字コードを EUC コードとして扱います。
5	出力漢字コードの指定	OUtcode	アセンブラ<その他>[ユーザ指定オプション :]	オブジェクトファイルに出力する漢字コードを指定します。
6	アセンブルリストの行数指定	LINes = <行数>	アセンブラ<その他>[ユーザ指定オプション :]	アセンブルリストの行数を設定します。
7	アセンブルリストの桁数指定	COlumns = <桁数>	アセンブラ<その他>[ユーザ指定オプション :]	アセンブルリストの桁数を設定します。
8	コピーライト出力抑止	LOGO NOLOGO	ー (常に nologo が有効)	コピーライトを出力します コピーライトの出力を抑止します。
9	サブコマンドファイルの指定	SUBcommand = <ファイル名>	ー	コマンドラインをファイルから入力します。

異常終了とするエラーのレベルの変更

ABort

アセンブラ<その他>[ユーザ指定オプション :]

書　式　　ABort = { Warning | Error }

説　明　　abort オプションは、エラーレベルを変更します。
OS へのリターン値が 1 以上の場合、オブジェクトモジュールの出力を抑止します。
abort による指定はオブジェクトモジュールを出力する場合に限り有効です。
OS へのリターン値は次のとおりです。

発生個数			オプション指定時の OS へのリターン値			
ウォーニング	エラー	致命的エラー	abort=Warning		abort=Error	
			PC	UNIX	PC	UNIX
0	0	0	0	0	0	0
1 以上	0	0	2	1	0	0
−	1 以上	0	2	1	2	1
−	−	1 以上	4	1	4	1

欧州コード文字

LATIN1

アセンブラ<その他>[ユーザ指定オプション :]

書　式　　LATIN1

説　明　　latin1 オプションは、文字列およびコメント内で欧州コード文字の記述を可能にします。
latin1 オプションは、sjis,euc,outcode と一緒に指定しないでください。

漢字コードをシフト JIS に指定

SJIS

アセンブラ<その他>[ユーザ指定オプション :]

書　式　　SJIS

説　明　　sjis オプションは、文字列、コメント内での日本語記述を可能とします。
sjis を指定すると文字列、コメント内の日本語はシフト JIS コードとして解釈します。
省略すると文字列、コメント内の日本語はホストマシンに依存する日本語コードとして解釈します。
sjis オプションは、latin1,euc と一緒に指定しないでください。

漢字コードを EUC に指定

EUC

アセンブラ<その他>[ユーザ指定オプション :]

書　式　　EUC

説　明　　euc オプションは、文字列、コメント内での日本語記述を可能とします。
euc を指定すると文字列、コメント内の日本語は EUC コードとして解釈します。
省略すると文字列、コメント内の日本語はホストマシンに依存する日本語コードとして解釈します。
euc オプションは、latin1,sjis と一緒に指定しないでください。

出力漢字コードを設定

OUtcode

アセンブラ<その他>[ユーザ指定オプション :]

書　式　　OUtcode= <漢字コード>
　　<漢字コード> = {SJIS | EUC}

説　明　　outcode オプションはソースファイル内の日本語記述を指定した漢字コードに変換し、ファイルに出力します。
outcode とソースファイル内の漢字コード（sjis,euc）の指定によるオブジェクトファイルへ出力する漢字コードは次のとおりです。

outcode の漢字コード	ソースファイル内の漢字コード		
	sjis	euc	指定なし
sjis	シフト JIS コード	シフト JIS コード	シフト JIS コード
euc	EUC コード	EUC コード	EUC コード
指定なし	シフト JIS コード	EUC コード	デフォルト漢字コード

デフォルトの漢字コードは次のとおりです。

SPARC ステーション	EUC コード
HP9000/700 シリーズ	シフト JIS コード
PC	シフト JIS コード

アセンブルリストの行数設定

LINes

アセンブラ<その他>[ユーザ指定オプション :]

書　式　　LINes =<行数>

説　明　　lines オプションは、アセンブルリストの１ページあたりの行数を設定します。
行数として有効な値は 20～255 です。
lines による指定はアセンブルリストを出力する場合に限り有効です。

アセンブラ制御命令との関係（アセンブラはオプションによる指定を優先します）

オプション	制御命令	結　果
lines=<**行数**>	(**指定に関わらず**)	**１ページあたり、**lines **オプションで指定した行数になる。**
(**指定なし**)	.FORM LIN=<**行数**>	**１ページあたり、**.FORM **制御命令で指定した行数になる。**
	(**指定なし**)	**１ページあたり、**60 **行になる。**

アセンブルリストの桁数設定

COlumns

アセンブラ<その他>[ユーザ指定オプション :]

書　式　　COlumns=<桁数>

説　明　　columns オプションは、アセンブルリストの１行あたりの桁数を設定します。
桁数として有効な値は 79～255 です。
columns による指定はアセンブルリストを出力する場合に限り有効です。

アセンブラ制御命令との関係（アセンブラはオプションによる指定を優先します）

オプション	制御命令	結　果
columns=<**桁数**>	(**指定に関わらず**)	**１行あたり、**columns **オプションで指定した桁数になる**
(**指定なし**)	.FORM COL=<**桁数**>	**１行あたり、**.FORM **制御命令で指定した桁数になる。**
	(**指定なし**)	**１行あたり、**132 **桁になる。**

コピーライト出力抑止

LOGO
NOLOGO

なし(常に nologo が有効)

書　式　LOGO
NOLOGO

説　明　コピーライトの出力を抑止します。
logo を指定した場合、コピーライト表示が出力されます。
nologo を指定した場合、コピーライトの表示の出力が抑止されます。
本オプション省略時解釈は、logo です。

サブコマンドファイル指定

SUBcommand

なし

書　式　SUBcommand=<サブコマンドファイル名>

説　明　subcommand オプションは、コマンドラインをファイルから入力します。
通常のコマンドライン指定と同じ順番で、入力ファイル名とオプションを指定してください。
1 行に 1 つの入力ファイル名またはオプションを指定してください。
subcommand オプションは、サブコマンドファイル内に指定しないでください。

例　asm38 aaa.src -subcommand=aaa.sub

サブコマンドファイルの内容をコマンドラインに展開し、アセンブルします。
aaa.sub の内容：

```
bbb.src
-list
-noobj
```

展開結果：

```
asm38 aaa.src,bbb.src -list -noobj
```

備　考　サブコマンドファイル全体のサイズは最大 65,535 バイトです。

4. 最適化リンケージエディタ操作方法

4.1 オプション指定規則

4.1.1 コマンドラインの形式

コマンドラインの形式は以下のとおりです。

```
optlnk [ {△<ファイル名> | △<オプション列>}...]
  <オプション列>：-<オプション>[= <サブオプション>[,...]]
```

4.1.2 サブコマンドファイルの形式

サブコマンドファイルの形式は以下のとおりです。

```
<オプション> {= | △}[<サブオプション>[,...]] [△&] [;<コメント>]
  &：継続行指定
```

サブコマンドファイル形式の詳細は、**「4.2.8 サブコマンドファイルオプション」**を参照してください。

4.2 オプション解説

オプション、サブオプションの英大文字は短縮形指定時の文字を、下線は省略時解釈を示します。
また、統合開発環境の対応するダイアログメニューを、タブ名<カテゴリ名>[項目]...で示します。
オプションの順序は、統合開発環境のタブとその中のカテゴリに対応しています。

4.2.1 入力オプション

表 4.1 入力カテゴリオプション一覧

	項目	オプション	ダイアログメニュー	指定内容
1	**入力ファイル**	Input = <sub> [{,|△}...] <sub>: <**ファイル名**> [(<**モジュール名**>[,...])]	**リンカ<入力>** **[オプション項目 :]** **[リロケータブルファイル/オブジェクトファイル]**	**入力ファイルを指定** (**コマンドラインでは** input **なしで指定します**)
2	**ライブラリファイル**	LIBrary = <**ファイル名**>[,…]	**リンカ<入力>** **[オプション項目 :]** **[ライブラリファイル]**	**入力ライブラリファイルを指定**
3	**バイナリファイル**	Binary = <sub>[,...] <sub> : <**ファイル名**> (<**セクション名**> [,<**シンボル名**>])	**リンカ<入力>** **[オプション項目 :]** **[バイナリファイル]**	**入力バイナリファイルを指定**
4	**シンボル定義**	DEFine = <sub>[,...] <sub>: <**シンボル名**> = {<**シンボル名**> | <**数値**> }	**リンカ<入力>** **[オプション項目 :]** **[シンボル定義]**	**未定義シンボルの強制定義** **シンボル名と同値として定義** **数値で定義**
5	**実行開始アドレス**	ENTry = { <**シンボル名**> | <**アドレス**> }	**リンカ<入力>** **[エントリポイント :]**	**エントリシンボルを指定** **エントリアドレスを指定**
6	**プレリンカ**	NOPRElink	**リンカ<入力>** **[プレリンカ制御 :]**	**プレリンカの起動を抑止**

入力ファイル

Input

リンカ<入力>[オプション項目 :][リロケータブルファイル/オブジェクトファイル]

書　式　Input = <サブオプション>[{,|△}…]
<サブオプション> : <ファイル名>[(<モジュール名>[,…])]

説　明　入力ファイルを指定します。複数ある場合にはカンマ(,)またはスペースで区切って指定します。
ワイルドカード(*,?)も指定できます。ワイルドカードで指定した文字列はアルファベット順に展開します。数字と英文字は数字が先、英大文字と英小文字は英大文字が先になります。
入力ファイルとして指定できるのは、コンパイラ、アセンブラ出力オブジェクトファイル、最適化リンケージエディタ出力のリロケータブルファイルおよびアブソリュートファイルです。またライブラリ名(<モジュール名>)の形式で、ライブラリ内モジュールを入力ファイルとして指定することもできます。モジュール名は拡張子なしで指定します。
入力ファイル名に拡張子の指定がない場合、モジュール名がない時は「obj」、モジュール名がある時は「lib」を仮定します。

例　
```
input=a.obj lib1(e)    ; a.obj と lib1.lib 内のモジュール e を入力します
input=c*.obj           ; c で始まる拡張子 obj のファイルを全て入力します
```

備　考　form=object および extract 指定時、本オプションは無効です。
コマンドライン上で入力ファイルを指定する場合は、input 無しで指定します。

ライブラリファイル

LIBrary

リンカ<入力>[オプション項目 :][ライブラリファイル]

書　式　LIBrary = <ファイル名>[,…]

説　明　ライブラリファイルを指定します。複数ある場合にはカンマ(,)で区切って指定します。
ワイルドカード(*,?)も指定できます。ワイルドカードで指定した文字列はアルファベット順に展開します。数字と英文字は数字が先、英大文字と英小文字は英大文字が先になります。
入力ファイル名に拡張子の指定がない場合は、「lib」を仮定します。
form=library オプションまたは extract オプション指定時は、ライブラリファイルを編集対象ライブラリとして入力します。
それ以外の場合は、入力ファイルとして指定されたファイル間でのリンケージ処理後に、未定義シンボルをライブラリファイルから検索します。
ライブラリファイル内シンボルの検索は、ライブラリオプション指定ユーザライブラリファイル(指定順)、ライブラリオプション指定システムライブラリファイル(指定順)、デフォルトライブラリ(環境変数 HLNK_LIBRARY1,2,3)の順序で行います。

例　
```
library=a.lib,b        ; a.lib と b.lib を入力します。
library=c*.lib         ; c で始まる拡張子 lib のファイルを全て入力します。
```

バイナリファイル

Binary

リンカ<入力>[オプション項目 :][バイナリファイル]

書　式　Binary = <サブオプション>[,…]
<サブオプション> : <ファイル名>(<セクション名>[,<シンボル名>])

説　明　入力バイナリファイルを指定します。複数ある場合にはカンマ(,)で区切って指定します。
ファイル名に拡張子の指定がない場合は、「bin」を仮定します。
入力したバイナリデータは、指定したセクションのデータとして配置します。セクションのアドレスは start オプションで指定します。セクションは省略できません。
またシンボルを指定することにより、定義シンボルとしてリンクすることもできます。C/C++プログラムで参照している変数名の場合、プログラム中での参照名先頭に_を付加します。

例
```
input=a.obj
start=P,D*/200
binary=b.bin(D1bin),c.bin(D2bin,_datab)
```

b.bin を D1bin セクションとして、0x200 番地から配置します。
c.bin を D2bin セクションとして、D1bin の後に配置します。
c.bin データを定義シンボル_datab としてリンクします。

備　考　form={object | relocate | library}または strip 指定時、本オプションは無効です。
また入力オブジェクトファイル指定がない場合、本オプションは指定できません。

シンボル定義

DEFine

リンカ<入力>[オプション項目 :][シンボル定義]

書　式　DEFine = <サブオプション>[,…]
<サブオプション> : <シンボル名> = {<シンボル名> | <数値>}

説　明　未定義シンボルを外部定義シンボルまたは数値で強制定義します。
数値は 16 進数で指定します。先頭が A～F の場合は先にシンボルを検索し、該当するシンボルがなければ数値として解釈します。先頭に 0 を付加した場合は常に数値と解釈します。シンボル名が C/C++変数名の場合、プログラム中での定義名先頭に_を付加します。C++関数名の場合は(main 関数は除く)引数列を含めたプログラム中の定義名をダブルクォーテーションで囲んで指定します。ただし引数が void の場合は、"関数名()"で指定します。

例
```
define=_sym1=data       ;_sym1 を外部定義シンボル data と同値として定義します。
define=_sym2=4000       ;_sym2 を 0x4000 として定義します。
```

備　考　form={object | relocate | library}指定時、本オプションは無効です。

実行開始アドレス

ENTry

リンカ<入力>[エントリポイント :]

書　式　ENTry = {<シンボル名> | <アドレス>}

説　明　実行開始アドレスを外部定義シンボルまたはアドレスで指定します。
アドレスは 16 進数で指定します。先頭が A～F の場合は先に定義シンボルを検索し、該当するシンボルがなければアドレスと判断します。先頭に 0 を付加した場合は常にアドレスと解釈します。
シンボル名は、C 関数名の場合プログラム中での定義名先頭に_を付加します。C++関数名の場合は(main 関数は除く)引数列を含めたプログラム中の定義名をダブルクォーテーションで囲んで指定します。ただし引数が void の場合は、"関数名()"で指定します。
コンパイル、アセンブル時に entry シンボルを指定している場合、本オプション指定を優先します。

例
```
entry=_main             ;C/C++の main 関数を実行開始アドレスとして設定します。
entry="init()"          ;C++の init 関数を実行開始アドレスとして設定します。
entry=100               ;0x100 を実行開始アドレスとして設定します。
```

備　考　form={object | relocate | library}または strip 指定時、本オプションは無効です。
未参照シンボル削除最適化(optimize=symbol_delete)指定時には、実行開始アドレスは必ず必要です。指定がない場合は、未参照シンボル削除最適化指定は無効です。

プレリンカ

NOPRElink

リンカ<入力>[プレリンカ制御 :]

書　式　NOPRElink

説　明　プレリンカの起動を抑止します。
プレリンカは、C++テンプレートインスタンスの自動生成機能および実行時型検査機能をサポートします。C++テンプレート機能および実行時型検査機能を使用していない場合は、noprelink オプションを指定してください。リンク速度を速くすることができます。

備　考　extract または strip 指定時、本オプションは無効です。

4.2.2 出力オプション

表 4.2 出力カテゴリオプション一覧

	項目	オプション	ダイアログメニュー	指定内容
1	出力形式	FOrm = { Absolute | Relocate | Object | Library [= {S | U}] | Hexadecimal | Stype | Binary }	リンカ<出力> [出力形式 :]	アブソリュート形式 リロケータブル形式 オブジェクト形式 ライブラリ形式 HEX 形式 S タイプ形式 バイナリ形式
2	デバッグ情報	DEBug SDebug NODEBug	リンカ<出力> [デバッグ情報 :]	出力あり(出力ファイル内) デバッグ情報ファイル出力 出力なし
3	レコードサイズ統一	REcord = { H16 | H20 | H32 | S1 | S2 | S3 }	リンカ<出力> [レコードサイズ統一 :]	HEX レコード 拡張 HEX レコード 32bitHEX レコード S1 レコード S2 レコード S3 レコード
4	ROM 化支援	ROm = <sub>[,...] <sub> : <ROM セクション名> =<RAM セクション名>	リンカ<出力> [オプション項目 :] [ROM から RAM へマップするセクション]	RAM に領域を確保し、シンボルを RAM 上のアドレスでリロケーション解決
5	出力ファイル	OUtput = <sub>[,…] <sub> : <ファイル名> [=<出力範囲>] <出力範囲>: { <先頭アドレス> - <終了アドレス> | <セクション名>[:...] }	リンカ<出力> [オプション項目 :] [出力ファイル/インフォメーション抑止] [出力ファイルの分割]	出力ファイルを指定 (範囲指定、分割出力可能)
6	外部シンボル割り付け情報ファイル	MAp [= <ファイル名>]	リンカ<出力> [マップファイル出力]	外部シンボル割り付け情報ファイル出力を指定(SuperH 向け)
7	空きエリア出力指定	SPace = [<数値>]	リンカ<出力> [オプション項目 :] [空きエリア出力指定] [空きエリア出力]	空きエリアへの出力値の指定
8	インフォメーションメッセージ	Message NOMessage [= <sub>[,...]] <sub> : <エラー番号> [- <エラー番号>]	リンカ<出力> [オプション項目 :] [出力ファイル/インフォメーション抑止] [インフォメーションレベルメッセージ抑止]	出力あり 出力なし (エラー番号、範囲指定可能)

出力形式

FOrm

リンカ<出力>[出力形式 :]

書　式　　FOrm = {Absolute | Relocate | Object | Library[={S|U}]
| Hexadecimal | Stype | Binary}

説　明　　出力形式を指定します。
本オプションの省略時解釈は、form=absolute です。サブオプションの一覧を**表** 4.3 に示します。

表 4.3　form **オプションのサブオプション一覧**

	サブオプション名	内　容
1	absolute	**アブソリュートファイルを出力します。**
2	relocate	**リロケータブルファイルを出力します。**
3	object	**オブジェクトファイルを出力します。**extract **オプションでライブラリから１個のモジュールをオブジェクトファイルとして取り出すときに使用します。**
4	library	**ライブラリファイルを出力します。** library=s **指定時、出力ライブラリファイルをシステムライブラリとします。** library=u **指定時、出力ライブラリファイルをユーザライブラリとします。** **省略時解釈は、**library=u **です。**
5	hexadecimal	HEX **ファイルを出力します。**HEX **フォーマットは「**18.1.2 HEX **ファイル形式」を参照してください。**
6	stype	S **タイプファイルを出力します。**S **タイプフォーマットは「**18.1.1 S **タイプファイル形式」を参照してください。**
7	binary	**バイナリファイルを出力します。**

備　考　　出力形式と入力ファイル、他オプションとの関係を**表** 4.4 に示します。

表 4.4 出力形式と入力ファイル、他オプションとの関係

	出力形式	指定オプション	入力可能なファイル形式	指定可能なオプション[*1]
1	Absolute	strip **あり**	**アブソリュートファイル**	input, output, hide, show=symbol,reference
		上記以外	**オブジェクトファイル** **リロケータブルファイル** **バイナリファイル** **ライブラリファイル**	input, library, binary, debug/nodebug, sdebug, cpu, start, rom, entry, output, map, hide, optimize/nooptimize, samesize, symbol_forbid, samecode_forbid, variable_forbid, function_forbid, absolute_forbid, profile, cachesize, sbr, compress, rename, delete, define, fsymbol, stack, noprelink, show=symbol, reference, memory
2	Relocate	extract **あり**	**ライブラリファイル**	library, output, show=symbol,reference
		上記以外	**オブジェクトファイル** **リロケータブルファイル** **バイナリファイル** **ライブラリファイル**	input, library, debug/nodebug, output, hide, rename, delete, noprelink, show=symbol,reference
3	Object	extract **あり**	**ライブラリファイル**	library, output, show=symbol,reference
4	Hexadecimal Stype Binary		**オブジェクトファイル** **リロケータブルファイル** **バイナリファイル** **ライブラリファイル**	input, library, binary, cpu, start, rom, entry, output, map, space, optimize/nooptimize, samesize, symbol_forbid, samecode_forbid, variable_forbid, function_forbid, absolute_forbid, profile, cachesize, sbr, rename, delete, define, fsymbol, stack, noprelink, record, s9[*2], show=symbol,reference, memory
			アブソリュートファイル	input, output, record, s9[*2], show=symbol,reference
5	Library	strip **あり**	**ライブラリファイル**	library, output, hide, show=symbol,section
		extract **あり**	**ライブラリファイル**	library, output, show=symbol,section
		上記以外	**オブジェクトファイル** **リロケータブルファイル**	input, library, output, hide, rename, delete, replace, noprelink, show=symbol,section

【注】 *1: message/nomessage, change_message, logo/nologo, form, list, subcommand **は常に指定できます。**

*2: s9 **は出力形式が** form=stype **のときだけ指定できます。**

デバッグ情報

DEBug
SDebug
NODEBug

リンカ<出力>[デバッグ情報 :]

書　式　DEBug
SDebug
NODEBug

説　明　debug 情報の出力有無を指定します。
debug オプションは、出力ファイル中にデバッグ情報を出力します。
sdebug オプションは、<出力ファイル名>.dbg ファイルにデバッグ情報を出力します。
nodebug オプションは、デバッグ情報を出力しません。
form=relocate 指定時に sdebug オプションを指定したときは、debug オプションと解釈します。
output オプションで複数ファイル出力を指定時に debug オプションを指定したときは、sdebug オプションと解釈して、<先頭出力ファイル名>.dbg に出力します。
本オプション省略時解釈は、debug です。

備　考　form={object | library | hexadecimal | stype | binary}、strip、または、extract 指定時、本オプションは無効です。

レコードサイズ統一

REcord

リンカ<出力>[レコードサイズ統一 :]

書　式　REcord = {H16 | H20 | H32 | S1 | S2 | S3}

説　明　アドレス範囲に関係なく、一定のデータレコードで出力します。
指定したデータレコードより大きいアドレスが存在した場合、アドレスに合わせてデータレコードを選択します。
本オプション省略時は、それぞれのアドレスに合わせて混在したデータレコードを出力します。

備　考　form=hexadecimal または stype 指定がないとき、本オプションは無効です。

ROM 化支援

ROm

リンカ<出力>[オプション項目 :][ROM から RAM へマップするセクション]

書　式　ROm = <サブオプション>[,…]
<サブオプション> : <ROM セクション名>=<RAM セクション名>

説　明　初期化データ領域の ROM 用、RAM 用領域を確保し、ROM セクション内定義シンボルを RAM セクション内アドレスになるようリロケーションします。
ROM セクションには初期値のあるリロケータブルセクションを指定します。
RAM セクションには存在しないセクションまたはサイズ 0 のリロケータブルセクションを指定します。

例　rom=D=R
start=D/100,R/8000
D セクションと同サイズの R セクションを確保し、D セクション内定義シンボルを R セクション上のアドレスでリロケーションします。

備　考　form={object | relocate | library}または strip 指定時、本オプションは無効です。

出力ファイル

OUtput

リンカ<出力>[オプション項目 :][出力ファイル/インフォメーション抑止][出力ファイルの分割]

書　式　OUtput ＝ <サブオプション>[,…]
<サブオプション> :　<ファイル名>[=<出力範囲>]
<出力範囲>: {<先頭アドレス>-<終了アドレス>|<セクション名>[:…]}

説　明　出力ファイル名を指定します。form={absolute | hexadecimal | stype | binary} のときは、複数ファイルを指定できます。アドレスは 16 進数で指定します。先頭が A～F の場合は先にセクションを検索し、該当するセクションがなければアドレスと判断します。先頭に 0 を付加した場合は常にアドレスと解釈します。
本オプションの省略時解釈は、<先頭入力ファイル名>.<デフォルト拡張子>です。
デフォルト拡張子は、次のようになります。
form=absolute：「abs」、form=relocate　：「rel」、form=object：「obj」
form=library ：「lib」、form=hexadecima：「hex」、form=stype ：「mot」
form=binaruy ：「bin」

例　output=file1.abs=0-ffff,file2.abs=10000-1ffff
0～0xffff 間を file1.abs に、0x10000～0x1ffff 間を file2.abs に出力します。

output=file1.abs=sec1:sec2,file2.abs=sec3
sec1,sec2 セクションを file1.abs に、sec3 セクションを file2.abs に出力します。

外部シンボル割り付け情報ファイル出力

MAp

リンカ<出力>[マップファイル出力]

書　式　MAp [= <ファイル名>]

説　明　コンパイラが外部変数アクセス最適化で使用する外部変数割り付け情報ファイルを出力します。
<ファイル名>を指定しなかった場合は、output オプションで指定したファイル名、もしくは先頭入力ファイル名で、拡張子が bls のファイルを出力します。
外部変数割り付け情報ファイル作成時の変数宣言順と、再コンパイル後のオブジェクトを読み込んだ時の変数宣言順が変わっている場合はエラーを出力します。

備　考　form={absolute | hexadecimal | stype | binary}を指定した場合のみ、
本オプションは有効です。

空きエリア出力指定

SPace

リンカ<出力>[オプション項目 :][空きエリア出力指定][空きエリア出力]

書　式　SPace ＝ [<数値>]

説　明　出力範囲のメモリの空き領域を埋める 16 進数値を指定します。
空きエリアを埋める方法は、output オプション指定時の出力範囲指定方法によって下記のように異なります。
・出力範囲:セクション指定
　指定されたセクション間に空きが存在した場合に指定データを出力
・出力範囲:アドレス範囲指定
　指定された範囲内に空きが存在した場合に指定データを出力
出力データサイズは、1,2,4 バイト単位で有効となります。出力データサイズサイズは space オプションに指定する 16 進数の個数で決まります。3 バイトデータを指定した場合、上位桁を 0 拡張し 4 バイトのデータとして扱われます。また、奇数桁データを指定した場合も、上位桁に 0 拡張して偶数桁入力として扱われます。
空きエリアのサイズが出力データサイズの倍数でない場合、出力できるだけ出力し、メッセージによる警告を行います。

備　考　本オプションにて数値の指定がされなかった場合は、空きエリアへの出力は行いません。
本オプションは form={ binary |stype | hexadecimal}オプションを指定した場合にのみ有効となります。
output オプションによる出力範囲指定がされなかった場合は、本オプション指定は無効となります。

Message
NOMessage

リンカ<出力>[オプション項目 :][出力ファイル/インフォメーション抑止][インフォメーションレベルメッセージ抑止]

書　式　Message
NOMessage [= <サブオプション>[,…]]
<サブオプション> : <エラー番号>[-<エラー番号>]

説　明　インフォメーションレベルメッセージの出力有無を指定します。
message オプション指定時は、インフォメーションレベルメッセージを出力します。
nomessage オプション指定時は、インフォメーションレベルメッセージの出力を抑止します。
またエラー番号を指定すると、指定したエラー番号のメッセージ出力を抑止できます。ハイフン(-)を使用して抑止するエラー番号の範囲を指定することもできます。エラー番号としてウォーニング、エラーレベルメッセージ番号を指定した場合、change_message でインフォメーションレベルに変更したと仮定し、メッセージ出力を抑止します。
本オプションの省略時解釈は nomessage です。

例　nomessage=4,200-203,1300
L0004 および L0200～L0203 および L1300 のメッセージ出力を抑止します。

4.2.3 リストオプション

表 4.5 リストカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	リスト ファイル	LISt [= <ファイル名>]	リンカ <リスト> [リンケージリスト 出力]	リストファイル出力を指定
2	リスト 内容	SHow [= <sub>[,...]] <sub> : { SYmbol | Reference | SEction }	リンカ <リスト> [リスト内容 :]	 シンボル情報 参照回数 セクション情報

LISt

リンカ <リスト>[リンケージリスト出力]

書 式 `LISt [= <ファイル名>]`

説 明 リストファイル出力およびリストファイル名を指定します。
リストファイル名を指定しない場合には、出力(または先頭出力)ファイルと同じファイル名で、拡張子が `form=library` または `extract` 指定時「`lbp`」、それ以外のとき「`map`」のリストファイルが作成されます。

リスト内容

SHow

リンカ <リスト>[リスト内容 :]

書　式　　SHow[= <sub>[,…]]
　　<sub> : { SYmbol | Reference | SEction }

説　明　　リストの出力内容を指定します。
サブオプションの一覧を**表** 4.6 に示します。
各リストの具体例については**「**8.4 **リンケージリストの参照方法」、**
「8.5 **ライブラリリストの参照方法」**を参照してください。

表 4.6　show **オプションのサブオプション一覧**

	出力形式	サブオプション名	意味
1	form=library **または** extract **指定時**	symbol	**モジュール内シンボル名一覧を出力します。**
		reference	**指定できません。**
		section	**モジュール内セクション一覧を出力します。**
2	form=library **以外または** extract **指定なし時**	symbol	**シンボルアドレス、サイズ、種別、最適化内容を出力します。**
		reference	**シンボルの参照回数を出力します。**
		section	**指定できません。**

備　考　　form={object|relocate}を指定した場合、show=reference は無効です。

4.2.4 最適化オプション

表 4.7 最適化カテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	最適化	OPtimize [= <sub>[,...]] <sub> : { STring_unify | SYmbol_delete | Variable_access | Register | SAMe_code | SHort_format | Function_call | Branch | SPeed | SAFe } NOOPtimize	リンカ<最適化> [最適化方法 :] [最適化設定] [設定 :]	最適化あり 定数/文字列の統合 未参照シンボルの削除 短絶対アドレッシングモード活用 レジスタ退避・回復の最適化 共通コードの統合 アドレッシングモードの短縮 間接アドレッシングモード活用 分岐命令の最適化 スピード重視の最適化 安全な最適化 最適化なし
2	共通コードサイズ	SAMESize = <サイズ> (省略時：sames=1e)	リンカ<最適化> [統合サイズ :]	共通コード統合の対象となる最小サイズの指定
3	プロファイル情報	PROfile = <ファイル名>	リンカ<最適化> [プロファイル情報 :]	プロファイル情報ファイルの指定 (動的最適化を行います)
4	キャッシュサイズ	CAchesize = <sub>[,...] <sub>:{ Size = <サイズ> | Align = <ラインサイズ>} (省略時：ca=s=8,a=20)	リンカ<最適化> [キャッシュサイズ :]	キャッシュサイズの指定 キャッシュラインサイズの指定
5	最適化部分抑止	SYmbol_forbid = <シンボル名>[,...] SAMECode_forbid = <関数名>[,...] Variable_forbid = <シンボル名>[,...] FUnction_forbid = <関数名>[,...] Absolute_forbid = <アドレス> [+ <サイズ>] [,...]	リンカ<最適化> [最適化方法 :] [最適化部分抑止]	未参照シンボル削除抑止シンボル 共通コード統合抑止シンボル 短絶対アドレッシングモード活用抑止シンボル 間接アドレッシングモード活用抑止シンボル 最適化抑止アドレス範囲

OPtimize
NOOPtimize

リンカ<最適化>[最適化方法 :][最適化設定][設定 :]

書　式　OPtimize[= <サブオプション>[,…]]
NOOPtimize
<サブオプション> : {STring_unify | SYmbol_delete | Variable_access
| Register | SAMe_code | SHort_format
| Function_call | Branch | SPeed | SAFe}

説　明　モジュール間最適化実行有無を指定します。
optimize オプション指定時は、コンパイル、アセンブル時に goptimize オプションを指定したファイルに対して最適化を行います。
nooptimize オプション指定時は、モジュールの最適化を行いません。
本オプションの省略時解釈は、optimize です。サブオプションの一覧を**表** 4.8 に示します。

表 4.8　optimize **オプションのサブオプション一覧**

サブオプション	意　味	最適化対象プログラム[*1]			
		SHC	SHA	H8C	H8A
パラメタなし	**全ての最適化を実行します。**	○	×	○	○
string_unify	const **属性を持つ定数に対し、同一値定数を統合します。** const **属性を持つ定数には次のものが含まれます。** **・**C/C++**プログラム中で** const **宣言した変数** **・文字列データの初期値・リテラル定数**	○	×	○	×
symbol_delete	**１度も参照のない変数／関数を削除します。必ず** entry **オプションを指定してください。**	○	×	○	×
variable_access	8/16 **ビット絶対アドレッシングモードでアクセス可能な領域にアクセス回数の多い変数を割り当てます。必ず** cpu **オプションを指定してください。**	×	×	○	○
register	**関数の呼び出し関係を解析し、レジスタの再割付および冗長なレジスタ退避・回復コードを削除します。必ず** entry **オプションを指定してください。**	○	×	○	×
same_code	**複数の同一命令列をサブルーチン化します。**	○	×	○	×
short_format	**ディスプレースメント／イミディエートのコードサイズを短縮可能な場合、コードサイズがより小さくなる命令に置き換えます。**	×	×	○	○
function_call	0～0xFF **の範囲に空きがあれば、アクセス回数の多い関数のアドレスを割り当てます。また、**CPU **種別が** H8SX **の場合には、下記領域も使用されます。** H8SXN : 0x100～0x1FF H8SXM,H8SXA,H8SXX : 0x200～0x3FF **必ず** cpu **オプションを指定してください。**	×	×	○	○
branch	**プログラムの配置情報に基づいて、分岐命令サイズを最適化します。他の最適化項目を実行すると、指定の有無に関わらず必ず実行します。**	○	×	○	○

サブオプション	意　味	最適化対象プログラム*1			
		SHC	SHA	H8C	H8A
speed	**オブジェクトスピード低下を招く可能性のある最適化以外を実行します。** optimize=string_unify,symbol_delete,varaible_access, register,short_format,branch **と同じです。**	○	×	○	○
safe	**変数や関数の属性によって制限される可能性のある最適化以外を実行します。** optimize=string_unify,register, short_format,branch **と同じです。**	○	×	○	○

【注】*1: SHC: SH **用** C/C++**プログラム、**SHA: SH **用アセンブリプログラム、**
H8C: H8 **用** C/C++**プログラム、**H8A: H8 **用アセンブリプログラム**

備　考　form=｛object | relocate | library｝または strip 指定時、本オプションは無効です。
optimize=short_format 指定は、CPU 種別が H8SX の場合にのみ有効です。

共通コードサイズ

SAMESize

リンカ<最適化>[統合サイズ :]

書　式　SAMESize = <サイズ>

説　明　共通コード統合最適化(optimize=same_code)で、最適化対象となる最小コードサイズを指定します。8～7FFF までの 16 進数で指定してください。
本オプションの省略時解釈は、samesize=1E です。

備　考　optimize=same_code の指定がないとき、本オプションは無効です。

プロファイル情報

PROfile

リンカ<最適化>[プロファイル情報 :]

書　式　　PROfile = <ファイル名>

説　明　　プロファイル情報ファイルを指定します。
プロファイル情報ファイルとして指定できるのは、HDI Ver. 5.0 以降、または、HEW Ver. 2.0 以降が出力するプロファイル情報ファイルだけです。
プロファイル情報ファイルを指定すると、モジュール間最適化で動的情報に基づいた最適化を実行することができます。
プロファイル情報入力により影響がある最適化を**表** 4.9 に示します。

表 4.9　**プロファイル情報と最適化の関係**

サブオプション	意　味	最適化対象プログラム*1			
		SHC	SHA	H8C	H8A
variable_access	**動的アクセス回数の多い変数を優先的に割り当てます。**	×	×	○	○
function_call	**動的アクセス回数の多い関数の最適化優先順位を下げます。**	×	×	○	○
branch	**動的に呼び出し回数の多い関数を呼び出し元の関数の近くに配置します。** SH **用プログラムの場合は、**cachesize **オプションで指定するキャッシュサイズを意識した配置最適化を行います。**	○	△ *2	○	○

【注】 *1　SHC: SH **用** C/C++**プログラム、**SHA: SH **用アセンブリプログラム、**
H8C: H8 **用** C/C++**プログラム、**H8A: H8 **用アセンブリプログラム**
*2　**関数単位の移動は行いませんが、入力ファイル単位の移動は実行します。**

備　考　　optimize 指定がないとき、本オプションは無効です。

キャッシュサイズ

CAchesize

リンカ<最適化>[キャッシュサイズ :]

書　式　　CAchesize = <sub>[,...]
<sub>:{Size = <サイズ> | Align = <ラインサイズ> }

説　明　　キャッシュサイズおよびキャッシュラインサイズを指定します。
profile オプション指定時、分岐命令最適化(optimize=branch)で使用します。
サイズはキロバイト単位、ラインサイズはバイト単位の 16 進数で指定してください。
本オプションの省略時解釈は、cachesize=size=8,align=20 です。

備　考　　profile 指定がないとき、本オプションは無効です。

最適化部分抑止

SYmbol_forbid
SAMECode_forbid
Variable_forbid
FUnction_forbid
Absolute_forbid

リンカ<最適化>[最適化方法 :][最適化部分抑止]

書　式　　SYmbol_forbid = <シンボル名>[,…]
SAMECode_forbid = <関数名>[,…]
Variable_forbid = <シンボル名>[,…]
FUnction_forbid = <関数名>[,…]
Absolute_forbid = <アドレス>[+ <サイズ>][,…]

説　明　　特定のシンボル、アドレス範囲の最適化を抑止します。アドレス、サイズは 16 進数で指定してください。C/C++変数名、C 関数名はプログラム中での定義名先頭に_を付加します。C++関数の場合は、引数列を含めたプログラム中の定義名をダブルクォーテーションで囲んで指定します。但し引数が void の場合は、"関数名()"で指定します。サブオプションの一覧を**表** 4. 10 に示します。

表 4.10　**最適化部分抑止オプション一覧**

オプション	パラメタ	意味
symbol_forbid	**関数名|変数名**	**未参照シンボル削除最適化を抑止します。**
samecode_forbid	**関数名**	**共通コード統合最適化を抑止します。**
variable_forbid	**変数名**	**短絶対アドレッシングモード活用最適化を抑止します。**
function_forbid	**関数名**	**間接アドレッシングモード活用最適化を抑止します。**
absolute_forbid	**アドレス**[+**サイズ**]	**アドレス＋サイズの範囲の最適化を抑止します。**

例　　symbol_forbid="f(int)"　; C++関数 f(int) は参照回数 0 でも削除しません。

備　考　　optimize 指定がないとき、本オプションは無効です。

4.2.5 セクションオプション

表 4.11 セクションカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	セクション アドレス	STAR t= <sub>[,...] <sub> : <セクション名> [{: | ,} <セクション名>[,...]] [/<アドレス>]	リンカ <セクション> [設定項目 :] [セクション]	セクションの開始アドレス指定
2	シンボル アドレス ファイル	FSymbol = <セクション名>[,...]	リンカ <セクション> [設定項目 :] [シンボルアドレス ファイル]	外部定義シンボルアドレスの定義ファイル出力

セクションアドレス

STARt

リンカ <セクション>[設定項目 :][セクション]

書 式 STARt = <サブオプション>[,…]
<サブオプション> : <セクション名>[{:|,} <セクション名>[,…]][/<アドレス>]

説 明 セクションの開始アドレスを指定します。アドレスは 16 進数で指定してください。
コロン(:)で区切ることにより、同一アドレスへの複数セクションの割り付けを指定することもできます。
セクション名はワイルドカード(*)も指定できます。ワイルドカードで指定したセクションは入力順に展開します。
同一アドレスに割り付け指定したセクション間は、指定順に割り付けます。
同一セクション内オブジェクトは、入力ファイルの指定順、入力ライブラリの指定順に割り付けます。
アドレスの指定がない場合は、0 番地に割り付けます。
start オプションで指定していないセクションは、最終割り付けアドレスに続いて割り付けます。

例
```
start=P,C,D*/100,R1:R2/8000  ; D で始まるセクションに D1,D2 があると仮定
ROM=D1=R1,D2=R2
```
P,C,D1,D2 の順に 0x100 番地から割り付けます。R1,R2 はどちらも 0x8000 番地に割り付けます。

```
input=a.obj b.obj  ;a.obj は d.lib 内シンボル,b.obj は c.lib 内シンボルを参照
library=c.lib,d.lib
start=P/100
```
P セクション内割り付け順序は、a(P),b(P),c(P),d(P)になります。

備 考 form={object | relocate | library}または strip 指定時、本オプションは無効です。

シンボルアドレスファイル

FSymbol

リンカ <セクション>[設定項目 :][シンボルアドレスファイル]

書　式　FSymbol = <セクション名>[,…]

説　明　指定したセクション内外部定義シンボルをアセンブラ制御命令形式でファイルに出力します。
ファイル名は、<出力ファイル>.fsy です。

例　
```
fsymbol=sct2,sct3
output=test.abs
```
セクション sct2,sct3 の外部定義シンボルを test.fsy に出力します。

[test.fsy の出力例]
```
;OPTIMIZING LINKAGE EDITOR GENERATED FILE 1999.11.26
;fsymbol = sct2, sct3

;SECTION NAME = sct2
 .export _f
_f: .equ h'00000000
 .export _g
_g: .equ h'00000016
;SECTION NAME = sct3
 .export _main
_main: .equ h'00000020
 .end
```

備　考　form={object | relocate | library}または strip 指定時、本オプションは無効です。

4.2.6 ベリファイオプション

表 4.12 ベリファイカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	アドレス整合性のチェック	CPu = { <cpu 情報ファイル名> | <メモリ種別> = <アドレス範囲>[,...] } <メモリ種別> = { ROm | RAm | XROm | XRAm | YROm | YRAm } <アドレス範囲>: <先頭アドレス> - <終了アドレス>	リンカ <ベリファイ> [アドレス整合チェック :]	cpu 情報ファイルを指定 セクションアドレスの割り付け可能範囲を指定

アドレス整合性のチェック

CPu

リンカ <ベリファイ>[アドレス整合チェック :]

書 式 CPu = { <cpu 情報ファイル名>
| <メモリ種別> = <アドレス範囲>[,…] }
＜メモリ種別＞ = { ROm | RAm | XROm | XRAm | YROm | YRAm }
＜アドレス範囲> : <先頭アドレス> - <終了アドレス>

説 明 セクションの割り付けアドレスの整合性をチェックします。
xrom/xram は DSP の X メモリ領域、yrom/yram は DSP の Y メモリ領域を指定します。
セクション割り付けが可能なアドレス範囲を 16 進数で指定してください。ROM／RAM の属性は、モジュール間最適化で使用します。
旧製品添付の cia(cpu information analyzer)で作成した cpu 情報ファイルを指定することもできます。

例 cpu=ROM=0-FFFF,RAM=10000-1FFFF
セクションアドレスが、0-FFFF または 10000-1FFFF の間に入っているかチェックします。
モジュール間最適化では、異なる属性間でのオブジェクトの移動は行いません。

備 考 form={object | relocate | library}または strip 指定時、本オプションは無効です。
CPU 種別が SHDSP,SH2DSP,SH3DSP,SH4ALDSP 以外の場合は、メモリ種別が xrom,xram,yrom,yram の指定は無効となります。

4.2.7 その他オプション

表 4.13 その他カテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	終端コード	S9	リンカ <その他> [その他のオプション :] [S9 レコードを終端に出力]	S9 レコードを常に出力
2	スタック情報ファイル	STACk	リンカ <その他> [その他のオプション :] [スタック情報ファイル(sni)出力]	スタック使用量情報ファイル出力
3	デバッグ情報圧縮	COmpress NOCOmpress	リンカ <その他> [その他のオプション :] [デバッグ情報圧縮]	デバッグ情報を圧縮する デバッグ情報を圧縮しない
4	メモリ使用量削減指定	MEMory = [High | Low]	リンカ <その他> [その他のオプション :] [入力ファイルロード時のメモリ使用量削減]	入力ファイルをロードする際のメモリ使用量指定
5	シンボル名変更	REName = <sub>[,...] <sub> : { [<**ファイル名**>] (<**名前**>=<**名前**>[,...]) | [<**モジュール名**>] (<**名前**>=<**名前**>[,...]) }	リンカ <その他> [ユーザ指定オプション :]	シンボル名、セクション名の変更
6	シンボル名削除	DELete = <sub>[,...] <sub> : { <**モジュール名**> | [<**ファイル名**>] (<**名前**>[,...]) }	リンカ <その他> [ユーザ指定オプション :]	シンボル名、モジュール名の削除
7	モジュールの置き換え	REPlace = <sub>[,...] <sub> : <**ファイル**> [(<**モジュール**>[,...])]	リンカ <その他> [ユーザ指定オプション :]	ライブラリファイル内同名モジュールの置き換え
8	モジュールの抽出	EXTract = <**モジュール**>[,...]	リンカ <その他> [ユーザ指定オプション :]	ライブラリファイル内指定モジュールの抽出
9	デバッグ情報削除	STRip	リンカ <その他> [ユーザ指定オプション :]	アブソリュートファイル、ライブラリファイルのデバッグ情報削除
10	メッセージレベル	CHange_message = <sub>[,...] <sub>: {Information | Warning | Error} [=<**エラー番号**> [-<**エラー番号**>] [,...]]	リンカ <その他> [ユーザ指定オプション :]	メッセージレベルの変更
11	ローカルシンボル名秘匿指定	Hide	リンカ <その他> [ユーザ指定オプション :]	ローカルシンボル名情報を削除

終端コード

S9

リンカ <その他>[その他のオプション :][S9 レコードを終端に出力]

書　式　S9

説　明　エントリアドレスが 0x10000 を超える場合でも、S9 レコードを終端に出力します。

備　考　form=stype 指定がないとき、本オプションは無効です。

スタック情報ファイル

STACk

リンカ <その他>[その他のオプション :][スタック情報ファイル(sni)出力]

書　式　STACk

説　明　スタック使用量情報ファイルを出力します。
ファイル名は、<出力ファイル名>.sni になります。

備　考　form={object | relocate | library}および strip 指定時、本オプションは無効です。

デバッグ情報圧縮

COmpress
NOCOmpress

リンカ <その他>[その他のオプション :][デバッグ情報圧縮]

書　式　COmpress

NOCOmpress

説　明　デバッグ情報の圧縮有無を指定します。
compress オプションを指定した場合、デバッグ情報を圧縮します。
nocompress オプションを指定した場合、デバッグ情報を圧縮しません。
デバッグ情報を圧縮すると、デバッガのロード速度が速くなります。また、nocompress オプションを指定すると、リンク速度を速くすることができます。
本オプションの省略時解釈は、nocompress です。

備　考　form={object | relocate | library | hexadecimal | stype | binary}
または strip オプションを指定した場合、本オプションは無効です。

メモリ使用量削減指定

MEMory

リンカ <その他>[その他のオプション :][入力ファイルロード時のメモリ使用量削減]

書　式　MEMory ＝ [High | Low]

説　明　リンク時に使用するメモリ量を指定します。
memory=high オプションを指定した場合、従来通りの処理を行います。
memory=low オプションを指定した場合、リンク時に必要な情報のロードを細かく行うことにより、使用するメモリ量の削減を行います。ファイルアクセスの頻度が増えるため、メモリ使用量が実装メモリを超えない状況では memory=high オプション指定より処理が遅くなります。

大規模なプロジェクトをリンクした際、最適化リンケージエディタのメモリ使用量が稼動マシンの実装メモリ量を越えてしまい、動作が遅くなっているような場合には memory=low オプション指定をお試しください。

備　考　下記オプションを指定した場合、本オプション指定は無効となります。
optimize, compress, delete, rename, map, stack,
list と show=reference(同時指定した場合),
また、入力ファイルや出力ファイル形式よっても無効となる組み合わせがあります。詳細は、**「**4.2.2 **出力オプション」**の**表** 4.4 を参照してください。

シンボル名変更

REName

リンカ <その他>[ユーザ指定オプション :]

書　式　REName = <サブオプション>[,…]
<サブオプション> : { [<ファイル>](<名前> = <名前>[,…])
| [<モジュール>](<名前> = <名前>[,…])}

説　明　外部シンボル名、セクション名を変更します。
特定のファイルまたは特定のライブラリ内モジュールに含まれるシンボル名、セクション名を変更することもできます。
C/C++変数名の場合、プログラム中での定義名先頭に_を付加します。
関数名を変更した場合の動作は保証できません。
指定した名前がセクション、シンボルの両方に存在した場合、シンボル名を優先します。
同一ファイル名、モジュール名が複数存在する場合は、先に入力した方を優先します。

例
```
rename=(_sym1=data)      ;_sym1 を data に変更します。
rename=lib1(P=P1)        ;ライブラリモジュール lib1 内の P セクションを
                         ;P1 セクションに変更します。
```

備　考　extract または strip 指定時、本オプションは無効です。

シンボル名削除

DELete

リンカ <その他>[ユーザ指定オプション :]

書　式　DELete = <サブオプション>[,…]
<サブオプション> : { [<ファイル>](<名前>[,…])
| <モジュール> }

説　明　外部シンボル名またはライブラリモジュールを削除します。
特定のファイルに含まれるシンボル名、モジュールを削除することもできます。
C/C++変数名、C 関数名はプログラム中での定義名先頭に_を付加します。C++関数の場合は、引数列を含めたプログラム中の定義名をダブルクォーテーションで囲んで指定します。但し引数が void の場合は、"関数名()"で指定します。同一ファイル名が複数存在する場合は、先に入力した方を優先します。
本オプションでは、シンボル名削除のときオブジェクトは削除しません。

例
```
delete=(_sym1)           ;全ファイル中のシンボル名_sym1 を削除します。
delete=file1.obj(_sym2) ;file1.obj 内のシンボル名_sym2 を削除します。
```

備　考　extract または strip 指定時、本オプションは無効です。

モジュールの置き換え

REPlace

リンカ <その他>[ユーザ指定オプション :]

書　式　REPlace = <サブオプション>[,…]

<サブオプション> : <ファイル名>[(<モジュール名>[,…])]

説　明　ライブラリモジュールを置換します。
指定したファイルまたはライブラリモジュールと library オプションで指定したライブラリ内同名モジュールを置き換えます。

例
```
replace=file1.obj       ;モジュール file1 と file1.obj を置換します。
replace=lib1.lib(mdl1)  ;モジュール mdl1 とライブラリファイル lib1.lib 内
                        ;モジュール mdl1 を置換します。
```

備　考　form={object | relocate | absolute | hexadecimal | stype | binary}
および extract、strip 指定時、本オプションは無効です。

モジュールの抽出

EXTract

リンカ <その他>[ユーザ指定オプション :]

書　式　EXTract = <モジュール名>[,…]

説　明　ライブラリモジュールを抽出します。
指定したライブラリモジュールを library オプションで指定したライブラリファイルから抽出します。

例
```
extract=file1           ;モジュール file1 を抽出します。
```

備　考　form={absolute | hexadecimal | stype | binary}および strip 指定時、
本オプションは無効です。

デバッグ情報削除

STRip

リンカ <その他>[ユーザ指定オプション :]

書　式　STRip

説　明　アブソリュートファイル、ライブラリファイルのデバッグ情報を削除します。
strip オプション指定時は、入力ファイルと出力ファイルは 1 対 1 対応になります。

例　input=file1.abs file2.abs file3.abs
strip
file1.abs, file2.abs のデバッグ情報を削除し、それぞれ file1.abs, file2.abs, file3.abs に出力します。デバッグ情報削除前のファイルは、file1.abk, file2.abk, file3.abk にバックアップします。

備　考　form={object | relocate | hexadecimal | stype | binary}指定時、
本オプションは無効です。

メッセージレベル

CHange_message

リンカ <その他>[ユーザ指定オプション :]

書　式　CHange_message = <サブオプション>[,…]
<サブオプション> : <エラーレベル>[=<エラー番号>[-<エラー番号>][,…]]
<エラーレベル>　: {Information | Warning | Error}

説　明　インフォメーション、ウォーニング、エラーレベルのメッセージレベルを変更します。
メッセージ出力時の実行継続/中断を変更できます。

例　change_message=warning=2310
L2310 をウォーニングレベルに変更し、L2310 出力時も処理を継続します。

change_message=error
全てのインフォメーション、ウォーニングメッセージをエラーレベルに変更します。
メッセージを一つでも出力すると、処理を中断します。

ローカルシンボル名秘匿指定

Hide

リンカ <その他>[ユーザ指定オプション :]

書　式　Hide

説　明　本オプションを指定した場合、出力ファイル内のローカルシンボル名情報を消去します。
ローカルシンボルに関する名前の情報が消去されますので、バイナリエディタなどでファイルを開いてもローカルシンボル名は確認できなくなります。生成されるファイルの動作への影響は一切ありません。
ローカルシンボル名を機密扱いにしたい場合などに本オプションを指定してください。

秘匿対象となるシンボルの種類を以下に挙げます。
・C ソース：static 修飾子を指定した変数名、関数名など
・C ソース：goto 文のラベル名
・アセンブリソース：外部定義(参照)シンボル宣言していないシンボル名

例　C ソースで本オプションの機能が有効となる記述の例を以下に示します。

```
int g1;
int g2=1;
const int g3=3;
static int s1;           //<--- static 変数名は秘匿対象
static int s2=1;         //<--- static 変数名は秘匿対象
static const int s3=2;   //<--- static 変数名は秘匿対象

static int sub1()        //<--- static 関数名は秘匿対象
{
    static int s1;       //<--- static 変数名は秘匿対象
    int l1;

    s1 = l1; l1 = s1;
    return(l1);
}

int main()
{
    sub1();
    if (g1==1)
        goto L1;
    g2=2;
L1:                      //<--- goto 文のラベル名は秘匿対象
    Return(0);
}
```

備　考　本オプションは出力ファイル形式が absolute,relocate,library の場合のみ有効です。
コンパイル/アセンブル時に goptimize オプションを指定したファイルを入力する場合、出力ファイル形式が relocate,library の場合は本オプションを指定できません。
map 最適化を行う状況で本オプションを指定する場合は、一度目のリンク時には指定せず、二度目のリンク時にのみ本オプションを指定してください。
デバッグ情報内のシンボル名は、本オプションを指定しても削除されません。

4.2.8 サブコマンドファイルオプション

表 4.14 サブコマンドファイルカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	サブコマンドファイル	SUbcommand = <ファイル名>	リンカ<サブコマンドファイル> [サブコマンドファイルを指定]	サブコマンドファイルによるオプション指定

サブコマンドファイル

SUbcommand

リンカ<サブコマンドファイル> [サブコマンドファイルを指定]

書　式　SUbcommand = <ファイル名>

説　明　オプションをサブコマンドファイルで指定します。
サブコマンドファイルの書式は以下の通りです。

<オプション> {= | △}［<サブオプション>［,...］][△&] [;<コメント>]

オプションとサブオプションの区切りは、=の代わりに空白も指定できます。
input オプションの場合は、サブオプション区切りに空白を指定できます。
サブコマンドファイル内では 1 オプション／行で指定します。
サブオプションを 1 行に記述できない場合は、&を用いて継続指定できます。
サブコマンドファイル中に subcommand オプションは指定できません。

例　コマンドライン指定：optlnk file1.obj -sub=test.sub file4.obj
サブコマンド指定　：input　file2.obj file3.obj ;ここはコメントです。
library　lib1.lib, &　;継続行を指定します。
lib2.lib
サブコマンドファイルで指定したオプション内容を、コマンドライン上のサブコマンド指定位置に展開し、実行します。
ファイルの入力順序は、file1.obj, file2.obj, file3.obj, file4.obj になります。

4.2.9 CPU オプション

表 4.15 CPU タブオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	SBR アドレス指定	SBr = { <SBR アドレス> | User}	CPU [SBR 値 :]	8bit 絶対領域の開始アドレスを指定

8bit 絶対領域アドレス値指定

SBr

CPU [SBR 値 :]

書 式 SBr = { <アドレス> | User }

説 明 SBR のアドレス値を指定します。
本オプションでアドレス値を指定することにより、abs8 領域を用いた最適化が可能になります。本オプションで user を指定した場合は、abs8 領域への最適化は抑止されます。

備 考 本オプションは CPU 種別が H8SX の場合にのみ有効です。
ソース内、あるいはツールのオプション指定などで、複数の SBR アドレスが指定された場合には、本オプションは指定の如何に関わらず user が指定されたものとして扱われます。

4.2.10 残りのオプション

表 4.16 残りのオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	コピーライト	LOgo	-	出力あり
		NOLOgo		出力なし
2	継続指定	END	-	既入力オプション列を実行し、処理終了後は以降のオプション列を入力し、処理を継続
3	終了指定	EXIt	-	オプション入力の終了を指定

コピーライト

LOgo
NOLOgo

なし(常に nologo が有効)

書　式　LOgo
NOLOgo

説　明　コピーライトの出力有無を指定します。
logo オプション指定時はコピーライト表示を出力します。
nologo オプション指定時はコピーライト表示出力を抑止します。
本オプションの省略時解釈は、logo です。

継続処理

END

なし

書　式　　END

説　明　　END より前に指定したオプション列を実行します。リンケージ処理終了後、END 以降に指定したオプション列の入力、リンケージ処理を継続します。
本オプションは、コマンドライン上では指定できません。

例

```
input=a.obj,b.obj              ; 処理(1)
start=P,C,D/100,B/8000         ; 処理(2)
output=a.abs                   ; 処理(3)
end
input=a.abs                    ; 処理(4)
form=stype                     ; 処理(5)
output=a.mot                   ; 処理(6)
```

(1)～(3)の処理を実行し、a.abs を出力します。
その後、(4)～(6)の処理を実行し、a.mot を出力します。

終了処理

EXIt

なし

書　式　　EXIt

説　明　　オプション指定の終了を指定します。
本オプションは、コマンドライン上では指定できません。

例　　コマンドライン指定： optlnk -sub=test.sub -nodebug

```
test.sub:          input=a.obj,b.obj          ; 処理(1)
                     start=P,C,D/100,B/8000   ; 処理(2)
                     output=a.abs             ; 処理(3)
                     exit
```

(1)～(3)の処理を実行し、a.abs を出力します。
Exit 実行の後のコマンドライン指定の nodebug オプションは無効になります。

5. 標準ライブラリ構築ツール操作方法

5.1 オプション指定規則

5.1.1 コマンドラインの形式

コマンドラインの形式は以下の通りです。

```
lbg38 [△<オプション列> ...]
  <オプション列>：- <オプション>[=<サブオプション>[,...]]
```

5.2 オプション解説

標準ライブラリ構築ツールのオプション、サブオプションは、C/C++コンパイラオプションに準拠します。以下に C/C++コンパイラオプションとの相違を示します。C/C++コンパイラオプションの詳細は、**「2 C/C++コンパイラ操作方法」**を参照して下さい。

コマンドライン形式の英大文字は短縮形指定時の文字を、下線は省略時解釈を示します。また、統合開発環境の対応するダイアログメニューを、タブ名<カテゴリ名>[項目]...で示します。

5.2.1 追加オプション

表 5.1 追加オプション一覧

	項目	オプション	ダイアログメニュー	指定内容
1	**対象ヘッダファイル**	Head = <sub>[,...] <sub>:{ ALL | RUNTIME | CTYPE | MATH | MATHF | STDARG | STDIO | STDLIB | STRING | IOS | NEW | COMPLEX | CPPSTRING }	**標準ライブラリ** **<標準ライブラリ>** **[対象ヘッダファイル:]**	**構築対象種別を指定** **全てのライブラリ関数** **実行時ルーチン** ctype.h**＋実行時ルーチン** math.h**＋実行時ルーチン** mathf.h**＋実行時ルーチン** stdarg.h**＋実行時ルーチン** stdio.h**＋実行時ルーチン** stdlib.h**＋実行時ルーチン** string.h**＋実行時ルーチン** ios**＋実行時ルーチン** new**＋実行時ルーチン** complex**＋実行時ルーチン** string**＋実行時ルーチン**
2	**出力ファイル**	OUTPut = <**ファイル名**>	**標準ライブラリ** **<オブジェクト>** **[出力ファイル:]**	**出力ライブラリファイル名を指定**
3	**リエントラントライブラリ**	REent	**標準ライブラリ** **<オブジェクト>** **[リエントラントライブラリ]**	**リエントラントライブラリを生成**

出力ファイル

Head

標準ライブラリ<標準ライブラリ>[対象ヘッダファイル :]

書　式　　Head ＝ <sub>[,...]

```
     <sub>:{ALL          |
             RUNTIME     |
             CTYPE       |
             MATH        |
             MATHF       |
             STDARG      |
             STDIO       |
             STDLIB      |
             STRING      |
             IOS         |
             NEW         |
             COMPLEX     |
             CPPSTRING   }
```

説　明　　構築対象種類別をヘッダファイル名で指定します。
各ヘッダファイルとライブラリ関数との関係は、**「**10.3 C/C++**ライブラリ」**を参照してください。実行時ルーチン(runtime)は常に構築対象ファイルになります。
本オプションの省略時解釈は、head=all です。

例　　lbg38 -output=h8s.lib -head=mathf -cpu=2600a
mathf.h で定義されたライブラリ関数と実行時ルーチンを -cpu=2600a でコンパイルし、ライブラリファイル h8s.lib を出力します。

出力ファイル

OUTPut

標準ライブラリ<オブジェクト>[出力ファイル :]

書　式　　OUTPut ＝ <ファイル名>

説　明　　出力ファイル名を指定します。
本オプションの省略時解釈は、output=stdlib.lib です。

例　　lbg38 -output=h8s.lib -optimize -speed -goptimize -cpu=2600a
全標準ライブラリ用ソースファイルを、-optimize -speed -goptimize -cpu=2600a でコンパイルし、ライブラリファイル h8s.lib を出力します。

REent

標準ライブラリ<オブジェクト>[リエントラントライブラリ]

書　式　　REent

説　明　　リエントラントライブラリを生成します。ただし、rand、srand 関数はリエントラントではありません。また、strtok 関数の同一文字列に対する連続的な呼び出しは保証しません。

例　　（ユーザプログラム）

```
#define  _REENTRANT
#include <stdlib.h>
```

備　考　　リエントラントライブラリをリンクする場合は、プログラム内で標準インクルードファイルをインクルードする前に、_REENTRANT というマクロ名を#define 文で定義(#define _REENTRANT)するか、コンパイル時に define オプションで_REENTRANT を定義してください。

5.2.2 指定不可オプション

C/C++コンパイラオプションのうち、標準ライブラリ構築ツールで指定できないオプションを**表** 5.2 に示します。指定した場合は無視します。

表 5.2 **指定不可オプション一覧**

	項目	オプション	コンパイラ解釈	内容
1	**インクルードファイル ディレクトリ**	Include	**なし**	―
2	**マクロ名の定義**	DEFine	**なし**	―
3	**プリプロセッサ展開時** #line **出力抑止**	NOLINe	**なし**	―
4	**インフォメーション メッセージ**	Message NOMessage	NOMessage	**出力なし**
5	**プリプロセッサ展開**	PREProcessor	**なし**	―
6	**オブジェクト形式**	Code	Code = Machinecode	**機械語プログラムを出力**
7	**デバッグ情報**	DEBug NODEBug	NODEBug	**出力なし**
8	**オブジェクトファイル 出力指定**	OBject NOOBject	OBject	**出力あり**
9	**テンプレートインスタンス 生成機能**	Template	**なし**	**テンプレート機能は使用していません**
10	**リストファイル**	List NOList	NOList	**出力なし**
11	**リスト内容と形式**	SHow	**なし**	―
12	**コメントのネスト**	COMment	**なし**	**コメントネストは使用していません**
13	MAC **レジスタ保証**	MAcsave	**なし**	**割り込み関数は含まれません**
14	**メッセージレベル**	CHAnge_message	**なし**	―
15	C/C++**言語の選択**	LANg	**なし**	**ファイル拡張子に従います**
16	**コピーライト出力抑止**	LOGo NOLOGo	NOLOGo	**コピーライトの出力を抑止**
17	**文字列内の文字コード**	EUc SJis LATin1	**なし**	**文字コードは使用していません**
18	**オブジェクトコード内 漢字変換**	OUtcode	**なし**	**文字コードは使用していません**

5.2.3 オプション指定時の注意事項

オプション指定時は、次の規則に従ってください。

(1) cpu, regparam, structreg/nostructreg, longreg/nolongreg, stack, double=float, byteenum, pack, rtti, exception, bit_order, indirect=normal/extended, ptr16, sbr オプションは、コンパイル時と同じオプションを指定してください。

(2) #pragma global_register 使用時、preinclude オプションで#pragma global_register 宣言を含むヘッダファイルをインクルード指定してください。統合開発環境で指定する場合は、標準ライブラリ <その他>[ユーザ指定オプション :]で指定してください。

6. スタック解析ツール操作方法

スタック解析ツールは、最適化リンケージエディタが出力したスタック情報ファイル(*.sni)またはシミュレータ・デバッガが出力したプロファイル情報ファイル(*.pro)を読み込んでスタック使用量を表示します。

また、スタック情報ファイルに出力できないアセンブリプログラムのスタック使用量は、編集機能を用いて情報を追加・修正することが可能であり、システム全体のスタック使用量を求めることもできます。

編集したスタック使用量に関する情報は、呼び出し情報ファイル(*.cal)として保存・読込み可能です。

6.1 スタック解析ツールの起動

スタック解析ツールを起動するには、Windows®のスタートメニューより“ ファイル名を指定して実行 ”を選択し、Call.exe を指定し実行して下さい。

また、統合開発環境をご使用の場合は、Windows®のスタートメニューで“ プログラム ”を選び、HEW に登録されている“ Call Walker ”を選択して下さい。 統合開発環境を起動後は、Tools メニューより起動することもできます。

6.2 スタック解析ツールの機能概要

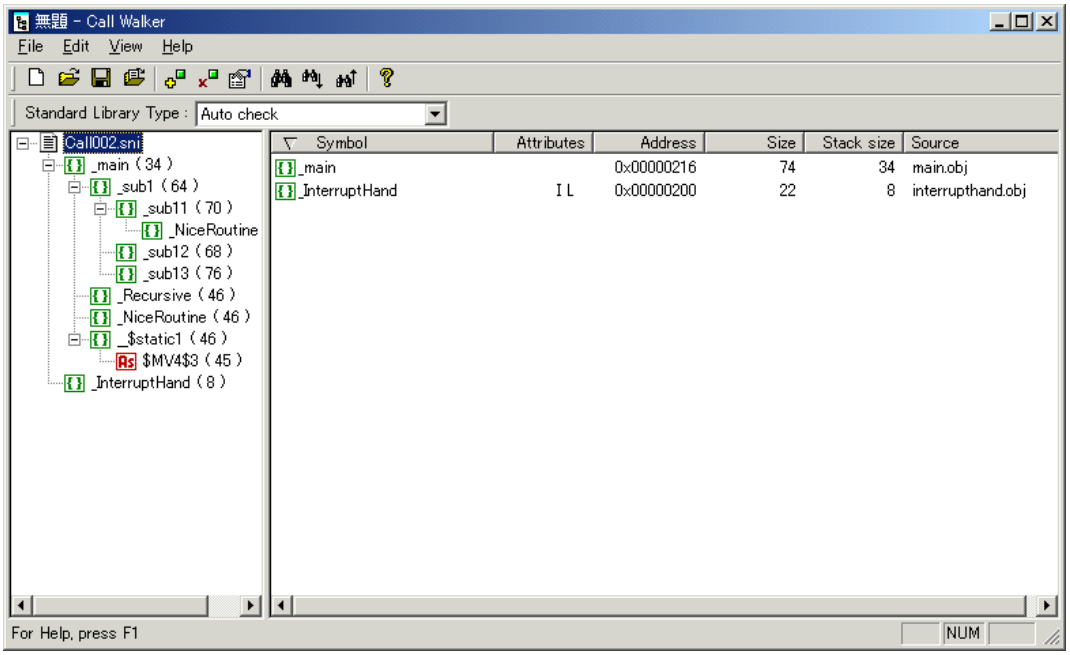


図 6.1 **スタック解析ツールの使用例**

スタック解析ツールの使用例を**図** 6.1 に、メニューごとの機能一覧を**表** 6.1 に示します。

左側の呼び出し情報ビューには、関数の呼び出し情報と使用スタック情報が表示されます。右側のシンボル情報ビューには、関数およびシンボルの更に詳しい情報が表示されます。

なお、{}は C または C++関数を、Asはアセンブリ関数を示します。

【注】 割込み関数のスタック使用量には、コンディションコードレジスタ（CCR）退避分および CPU 種別が H8SX、H8S/2600、H8S/2000 **の場合にエクステンドレジスタ（**EXR**）退避分が常に含まれます。**

表 6.1 スタック解析ツール機能一覧表

メニュー		機能
File	New	編集情報をクリアし、新規作成します。
	Open	既存の呼び出し情報ファイル(*.cal)をオープンします。
	Save	現在編集中の呼び出し情報ファイルを上書きで保存します。
	Save As	現在編集中の呼び出し情報ファイルをファイル名を指定して保存します。
	Import Stack File	最適化リンケージエディタが出力するスタック使用量情報ファイル(*.sni)、またはシミュレータ・デバッガが出力するプロファイル情報ファイル(*.pro) を読み込みます。
	Recent File	最近使用した呼び出し情報ファイルをオープンします。
	Exit	スタック解析ツールを終了します。
Edit	Add	新しく関数およびシンボル情報を追加します。
	Modify	既にある関数およびシンボル情報を編集します。
	Delete	関数およびシンボル情報を削除します。
	Find	指定した検索条件の関数およびシンボル情報を検索します。
	Find Next	Findコマンドにて検索した検索情報で次を検索します。
	Find Previous	Findコマンドにて検索した検索情報で前を検索します。
View	Toolbar	ツールバーの表示、非表示を切り替えます。
	Status Bar	ステータスバーの表示、非表示を切り替えます。
	Radix	シンボル情報内の数値部分(Address、Size、Stack size)の基数表示を切り替えます。
Help	Help Topics	スタック解析ツールのヘルプを表示します。
	About Call Walker	スタック解析ツールのバージョンや著作権等を表示します。

さらに詳しい操作方法については、スタック解析ツールのヘルプを参照ください。

7. 環境変数

7.1 環境変数一覧

環境変数の一覧を**表** 7.1 に示します。

表 7.1 **環境変数**

	環境変数	説明
1	path	**実行ファイルの格納ディレクトリを指定します。** **指定フォーマット：** PC **版** C> path= <**実行ファイルパス名**>[;<**既存パス名**>;...] UNIX **版 C シェル** %set path =(<**実行ファイルパス名**> $path) **ボーンシェル** %PATH=:<**実行ファイルパス名**>[:<**既存パス名**>:...] %export PATH
2	H38CPU	**コンパイラ・アセンブラの** cpu **オプションによる** CPU **種別の指定を、環境変数によって指定します。** <CPU／**動作モード**> **アドレス空間のビット幅**<*1> **乗除算器指定**<*2> H8SXN ー m | d | md H8SXM 20 | 24 (24) m | d | md H8SXA 20 | 24 | 28 | 32 (24) m | d | md H8SXX 28 | 32 (32) m | d | md 2600n ー ー 2600a 20 | 24 | 28 | 32 (24) ー 2000n ー ー 2000a 20 | 24 | 28 | 32 (24) ー 300hn ー ー 300ha 20 | 24 (24) ー 300 ー ー 300l ー ー **（ ( )内は指定省略時に設定されるデフォルト値です ）** H38CPU **環境変数による** CPU **の指定と、**cpu **オプションによる** CPU **の指定が相反する場合は、ウォーニングメッセージを出力し、**cpu **オプションの指定を優先します。** **指定フォーマット：** PC **版** C> set H38CPU= <CPU／**動作モード**>[:<*1>][:<*2>] UNIX **版 C シェル** % setenv H38CPU <CPU／**動作モード**>[:<*1>][:<*2>] **ボーンシェル** % H38CPU=<CPU／**動作モード**>[:<*1>][:<*2>] % export H38CPU
3	CH38 *[3]	**コンパイラのインクルードファイル格納ディレクトリを指定します。** **システムインクルードファイルの検索順序は、**include **オプション指定ディレクトリ、**CH38 **指定ディレクトリとなります。** **ユーザインクルードの検索順序は、カレントディレクトリ、**include **オプション指定ディレクトリ、**CH38 **指定ディレクトリとなります。** **環境変数** CH38 **の指定がない場合、**UNIX **版では**/usr/CH38 **を仮定します。**PC **版には省略時解釈がありません。** **指定フォーマット：** PC **版** C> set CH38= <**インクルードパス名**> [;<**インクルードパス名**> ;...] UNIX **版 C シェル** %setenv CH38 <**インクルードパス名**>[:<**インクルードパス名**> :...] **ボーンシェル** % CH38 = <**インクルードパス名**> [: <**インクルードパス名**>:...] % export CH38

	環境変数	説明
4	CH38TMP	コンパイラがテンポラリファイルを作成するディレクトリを指定します。この環境変数の指定がない場合は、カレントディレクトリにテンポラリファイルを作成します。 指定フォーマット： PC 版　C> set CH38TMP= <テンポラリファイルパス名> UNIX 版 C シェル　% setenv CH38TMP <テンポラリファイルパス名> ボーンシェル　% CH38TMP = <テンポラリファイルパス名> % export CH38TMP
5	HLNK_LIBRARY1 HLNK_LIBRARY2 HLNK_LIBRARY3	最適化リンケージエディタが使用するデフォルトライブラリ名を指定します。 library オプションで指定したライブラリを優先してリンクします。その後未解決のシンボルがある場合、1,2,3 の順にデフォルトライブラリを検索します。 指定フォーマット： PC 版　C> set HLNK_LIBRARY1= <ライブラリ名 1> C> set HLNK_LIBRARY2= <ライブラリ名 2> C> set HLNK_LIBRARY3= <ライブラリ名 3> UNIX 版 C シェル　% setenv HLNK_LIBRARY1 <ライブラリ名 1> % setenv HLNK_LIBRARY2 <ライブラリ名 2> % setenv HLNK_LIBRARY3 <ライブラリ名 3> ボーンシェル　% HLNK_LIBRARY1=<ライブラリ名 1> % export HLNK_LIBRARY1 % HLNK_LIBRARY2=<ライブラリ名 2> % export HLNK_LIBRARY2 % HLNK_LIBRARY3=<ライブラリ名 3> % export HLNK_LIBRARY3
6	HLNK_TMP	最適化リンケージエディタがテンポラリファイルを作成するディレクトリを指定します。この環境変数の指定がない場合は、カレントディレクトリにテンポラリファイルを作成します。 指定フォーマット： PC 版　C> set HLNK_TMP= <テンポラリファイルパス名> UNIX 版 C シェル　% setenv HLNK_TMP <テンポラリファイルパス名> ボーンシェル　% HLNK_TMP = <テンポラリファイルパス名> % export HLNK_TMP
7	HLNK_DIR *3	最適化リンケージエディタの入力ファイル格納ディレクトリを指定します。 input オプション、library オプションで指定したファイルの検索順序は、カレントディレクトリ、HLNK_DIR 指定ディレクトリになります。 但し、ワイルドカードで指定したファイルは、カレントディレクトリ内だけ検索します。 指定フォーマット： PC 版　C> set HLNK_DIR= <入力ファイルパス名>[;<入力ファイルパス名> ;...] UNIX 版 C シェル % setenv HLNK_DIR <入力ファイルパス名>[:<入力ファイルパス名> :...] ボーンシェル % HLNK_DIR = <入力ファイルパス名> [:<入力ファイルパス名> :...] % export HLNK_DIR

【注】*3　複数ディレクトリを指定する場合は、PC 版は、"; " (セミコロン)、
UNIX 版は、":" (コロン)で区切ってください。

7.2 コンパイラの暗黙の宣言

コンパイラについては、オプション指定やバージョンに合わせて、以下のような暗黙の#define 宣言が行われます。

表 7.2 暗黙の宣言

	オプション	暗黙の宣言
1	cpu = 300L	#define _ _300L_ _
	cpu = 300	#define _ _300_ _
	cpu = 300HN	#define _ _300HN_ _
	cpu = 300HA	#define _ _300HA_ _
	cpu = 2000N	#define _ _2000N_ _
	cpu = 2000A	#define _ _2000A_ _
	cpu = 2600N	#define _ _2600N_ _
	cpu = 2600A	#define _ _2600A_ _
	cpu = H8SXN	#define _ _H8SXN_ _
	cpu = H8SXM	#define _ _H8SXM_ _
	cpu = H8SXA	#define _ _H8SXA_ _
	cpu = H8SXX	#define _ _H8SXX_ _
	cpu = <H8SX>:M **または** MD	#define _ _HAS_MULTIPLIER_ _
	cpu = <H8SX>:D **または** MD	#define _ _HAS_DIVIDER_ _
2	double=float	#define _ _FLT_ _
3	byteenum	#define _ _BENM_ _
4	cpuexpand	#define _ _CPUEX_ _
5	library=intrinsic	#define _ _INTRINSIC_LIB_ _
6	ー	#define _ _ADDRESS_SPACE_ _ *1 *4
7	ー	#define _ _DATA_ADDRESS_SIZE_ _ *2 *4
8	ー	#define _ _H8_ _ *4
9	ー	#define _ _RENESAS_VERSION_ _ *3 *4
10	ー	#define _ _HITACHI_VERSION_ _ *3 *4
11	ー	#define _ _RENESAS_ _ *4
12	ー	#define _ _HITACHI_ _ *4

【注】*1 **アドレス空間サイズ**(16|20|24|28|32 bit)**が定義されます。**

*2 _ _DATA_ADDRESS_SIZE_ _**の値は次に示すように** 2 **または** 4 **に定義されます。**

2 : (300|**ノーマルモード**|**ミドルモード**)、**または、**

(**アドバンストモード**|**マキシマムモード**)**かつ** ptr16 **オプション指定有り**

4 : (**アドバンストモード**|**マキシマムモード**)**かつ** ptr16 **オプション指定無し**

*3 _ _RENESAS_VERSION_ _**と**_ _HITACHI_VERSION_ _**の値は、次のように参照します。**

・C **ソースプログラム内：**_ _RENESAS_VERSION_ _==0xaabb

aa: version **部分**

bb: revision **部分**

・コンパイラ内での定義例

```
#define _ _RENESAS_VERSION_ _ 0x0301      // Version 3.1C の場合
#define _ _RENESAS_VERSION_ _ 0x0400      // Version 4.0 の場合
```

*4 **常に定義されます**

8. ファイル仕様

8.1 ファイル名の付け方

ファイル名指定時に拡張子を省略した場合、標準のファイル拡張子を付加したファイル名を使用します。本開発環境で使用する標準のファイル拡張子を**表** 8.1 に示します。

表 8.1 **本開発環境で使用する標準のファイル拡張子**

No.	**拡張子**	**意味**
1	c	C **ソースプログラムファイル**
2	cpp, cc, cp	C++**ソースプログラムファイル**
3	h	**インクルードファイル**
4	lis, lst *[1]	C **プログラム用リストファイル**
5	lis, lpp *[1]	C++**プログラム用リストファイル**
6	p	C **プログラム用プリプロセッサ展開ファイル**
7	pp	C++**プログラム用プリプロセッサ展開ファイル**
8	src,mar	**アセンブリソースプログラムファイル**
9	exp	**アセンブリプログラム用プリプロセッサ展開ファイル**
10	lis	**アセンブリプログラム用リストファイル**
11	obj	**リロケータブルオブジェクトプログラムファイル**
12	rel	**リロケータブルロードモジュールファイル**
13	abs	**アブソリュートロードモジュールファイル**
14	map	**リンケージリストファイル**
15	lib	**ライブラリファイル**
16	lbp	**ライブラリリストファイル**
17	mot	S **タイプフォーマット**
18	hex	HEX **フォーマット**
19	bin	**バイナリファイル**
20	fsy	**最適化リンケージエディタ出力シンボルアドレスファイル**
21	sni	**スタック情報ファイル**
22	pro	**プロファイル情報ファイル**
23	dbg	DWARF2 **フォーマットデバッグ情報ファイル**
24	rti	**拡張子** td **のファイルで指定された定義を含むオブジェクト**
25	cal	**呼び出し情報ファイル**

【注】 *1 UNIX **版では** lis**、**PC **版では** lst **または** lpp **です。**

rti_で始まるファイル名は、システム予約名ですので使用しないでください。
プロジェクトで生成されるtpldirのフォルダの下に出力されるファイル拡張子を**表**8.2に示します。

表 8.2 tpldir **フォルダ出力ファイル**

No.	**拡張子**	**意味**
1	td	tentative **定義の変数情報ファイル**
2	ti	**テンプレート情報ファイル**
3	pi	**パラメタ情報ファイル**
4	ii	**インスタンス情報ファイル**

ファイル名の付け方の一般的な規則は、各ホストマシンに準じています。ご使用になるホストマシンのマニュアルを参照してください。

8.2 コンパイルリストの参照方法

本節では、コンパイルリストの内容と形式について説明します。

8.2.1 コンパイルリストの構成

コンパイルリストの構成と内容を**表** 8.3 に示します。

表 8.3 **コンパイルリストの構成と内容**

No.	リストの作成	内容	オプション指定方法	オプション省略時
1	ソースリスト情報	ソースプログラムのリスト*1	show＝source show＝nosource	出力する
		インクルードファイル、マクロ展開後のソースプログラムのリスト*2	show＝expansion show＝noexpansion	出力しない
2	エラー情報	コンパイル時のエラー情報	—	出力する
3	シンボル割り付け情報	関数のスタックフレームでの変数割り付け情報	show＝allocation show＝noallocation	出力しない
4	オブジェクト情報	オブジェクトプログラムの機械語、アセンブリコード	show＝object show＝noobject	出力しない
5	統計情報	各セクションのバイト数、シンボル数情報、オブジェクト種類	show＝statistics show＝nostatistics	出力する

【注】 *1 **ソースプログラムのリストは、**noexpansion **と** object **サブオプションを同時に指定した場合、オブジェクト情報内に出力されます。**

*2 **インクルードファイル、マクロ展開後のソースプログラムのリストは** show＝source **指定時に有効になります。**

8.2.2 ソースリスト情報

ソースリスト情報の出力形式には、プリプロセッサを通す前のプログラムを出力する形式（show=noexpansion を指定する場合）と、プリプロセッサを通した後のソースプログラムを出力する形式（show=expansion を指定する場合）があります。それぞれの出力形式を**図** 8.1 (a)、(b) に示します。また、**図** 8.1(b)に相違点を網掛けで示します。

(a) show=noexpansionのソースリスト情報

```
************ SOURCE LISTING ************

      Line Pi 0----+----1----+----2----+----3----+----4----+----5----+----6----
FILE NAME: m0260.c
         1 [1] #include "header.h"
         2
         3     int sum2(void)
         4     {  int j;
         5
         6     #ifdef SMALL
         7        j=SML_INT;
         8     #else
         9        j=LRG_INT;
        10     #endif
        11
        12        return j;  /*
continue 12345678901234567890123456789012345678901234567890123456789012345678901234567890
23456789012345678901234567890 */
        13     }
        [2]
```

(b) show=expansionのソースリスト情報

```
************ SOURCE LISTING ************

      Line Pi 0----+----1----+----2----+----3----+----4----+----5----+----6----
FILE NAME: m0260.c
         1 [1] #include "header.h"
FILE NAME: header.h
         1     #define SML_INT         1
         2     #define LRG_INT       100
FILE NAME: m0260.c
         2
         3     int sum2(void)
         4     {  int j;
         5
         6     #ifdef SMALL
         7 X      j=SML_INT;
         8[3]  #else
         9  E     j=100;
        10 [4] #endif
        11
        12        return j;  /* continue12345678901234567890123456789012345678901
23456789012345678901234567890 */
        13     }
        [2]
```

【注】
- [1] ソースプログラムファイル名、またはインクルードファイル名
- [2] ソースプログラムまたはインクルードファイル内の行番号
- [3] show＝expansion指定時、＃ifdef文、＃elif文等の条件コンパイル文でコンパイル対象とならないソース行
- [4] show＝expansion指定時、#define文によるマクロ置換のあったソース行

図 8.1 ソースリスト情報の出力形式

8.2.3 エラー情報

エラー情報の出力例を**図** 8.2 に示します。

```
************ SOURCE LISTING ************

      Line Pi 0----+----1----+----2----+----3----+----4----+----5----+----6----
FILE NAME: m0260.c
         1     #include "header.h"
         2
         3     extern int sum3(int);
         4
         5     sum3(int x)
         6     {
         7        int i;
         8        int j;
         9
        10        j=0;
        11        for (i=0; i<=x; i++){
        12           j+=k;                    <-------- エラー発生行
        13        }
        14
        15        return j;
        16     }


*********** ERROR INFORMATION **********


m0260.c(12) : C2225 (E) Undeclared name "k"
   [1]  [2]    [3]  [4]       [5]

NUMBER OF ERRORS:          1 }[6]
NUMBER OF WARNINGS:        0
NUMBER OF INFORMATIONS:    0  [7]
```

【注】

[1] エラーの発生したソースプログラム名，先頭から10文字まで表示
[2] エラーの発生したソースプログラム中の行番号
[3] エラーメッセージを識別するための番号
[4] (I) インフォメーションレベル
(W) ウォーニングレベル
(E) エラーレベル
(F) フェータルレベル
[5] エラーの内容
[6] エラーレベルメッセージ，ウォーニングレベルメッセージの総数
[7] インフォメーションレベルメッセージの総数
(messageオプションを指定した時のみ)

図 8.2 **エラーを含んだソースエラーリストとエラー情報**

8.2.4 シンボル割り付け情報

関数の引数や局所変数の割り付け情報を表します。H8S/2600 用アドバンストモードでコンパイルしたときのシンボル割り付け情報の例を**図** 8.3 に示します。

```
************ SOURCE LISTING ************

      Line Pi 0----+----1----+----2----+----3----+----4----+----5----+----6-
FILE NAME: m0280.c
         1     extern int h(char, char *, double );
         2
         3     int
         4     h(char a, register char *b, double c)
         5     {
         6        char    *d;
         7
         8        d= &a;
         9        h(*d,b,c);
        10        {
        11           register int i;
        12
        13           i= *d;
        14           return i;
        15        }
        16     }

******* STACK FRAME INFORMATION ********

FILE NAME: m0280.c
Function (File m0280.c   , Line     4): h
                  [1]
  Parameter Allocation
    a                                0xfffffff7 saved from R0L
    b                           REG  ER5        saved from ER1  [2]
    c                                0x00000008

  Level  1 (File m0280.c  , Line   5) Automatic/Register Variable Allocation
    d                                0xfffffff2                              [3]

  Level  2 (File m0280.c  , Line  10) Automatic/Register Variable Allocation
    i                           REG  R4

Parameter Area Size     : 0x00000008 Byte(s)
Linkage Area Size       : 0x00000008 Byte(s)
Local Variable Size     : 0x00000006 Byte(s)  [4]
Temporary Size          : 0x00000000 Byte(s)
Register Save Area Size : 0x00000008 Byte(s)
Total Frame Size        : 0x0000001e Byte(s)
【注】
 [1] 関数が定義されたファイル名，行番号，関数名
 [2] 引数の割り付け場所  X saved from Y   Yで渡された引数を関数入口でXにコピーした場合
                         REG ERx          割り付け場所がレジスタの場合，REGを表示
                         0xffffffxx       割り付け場所がスタックの場合，フレームポインタ（ER6）からのオフセット
                         0x000000xx       を表示
 [3] 複文内で宣言された局所変数の割り付け場所，スタックの場合はER6からのオフセット，レジスタの場合はREGを表示
 [4] 関数内で使用するスタックフレームの割り付け情報
        Parameter Area Size      : スタックで渡される引数領域とリターン値設定アドレス領域のサイズ
        Linkage Area Size        : リンケージ領域(リターンPC領域＋フレームポインタ退避領域)
                                   の合計サイズ。但し、フレームポインタ退避領域が無い場合もある。
                                   割り込み関数ではCCR退避分およびH8SX, H8S/2600, H8S/2000ではEXR退避分を加算。
        Local Variable Size      : 関数内で使用する局所変数領域とレジスタで渡された引数がスタックに割り付けら
                                   れた場合に使用する引数退避領域の合計サイズ
        Temporary Size           : 関数内でＣコンパイラが使用するテンポラリ領域のサイズ
        Register Save Area Size  : 関数内で使用するレジスタの値を退避しておく領域のサイズ
        Total Frame Size         : 関数内で割り付けるスタックフレームの合計サイズ
```

図 8.3　**シンボル割り付け情報**（cpu=2600a）

【注】 最適化オプション optimize＝1 **が指定されている場合または** H8SX **の場合、引数割り付け情報および局所変数割り付け情報は出力しません。このとき以下のメッセージを表示します。**

Optimize Option Specified : No Allocation Information Available

図 8.3 のシンボル割り付け情報に対応する、スタック上の割り付け例を**図** 8.4 に示します。

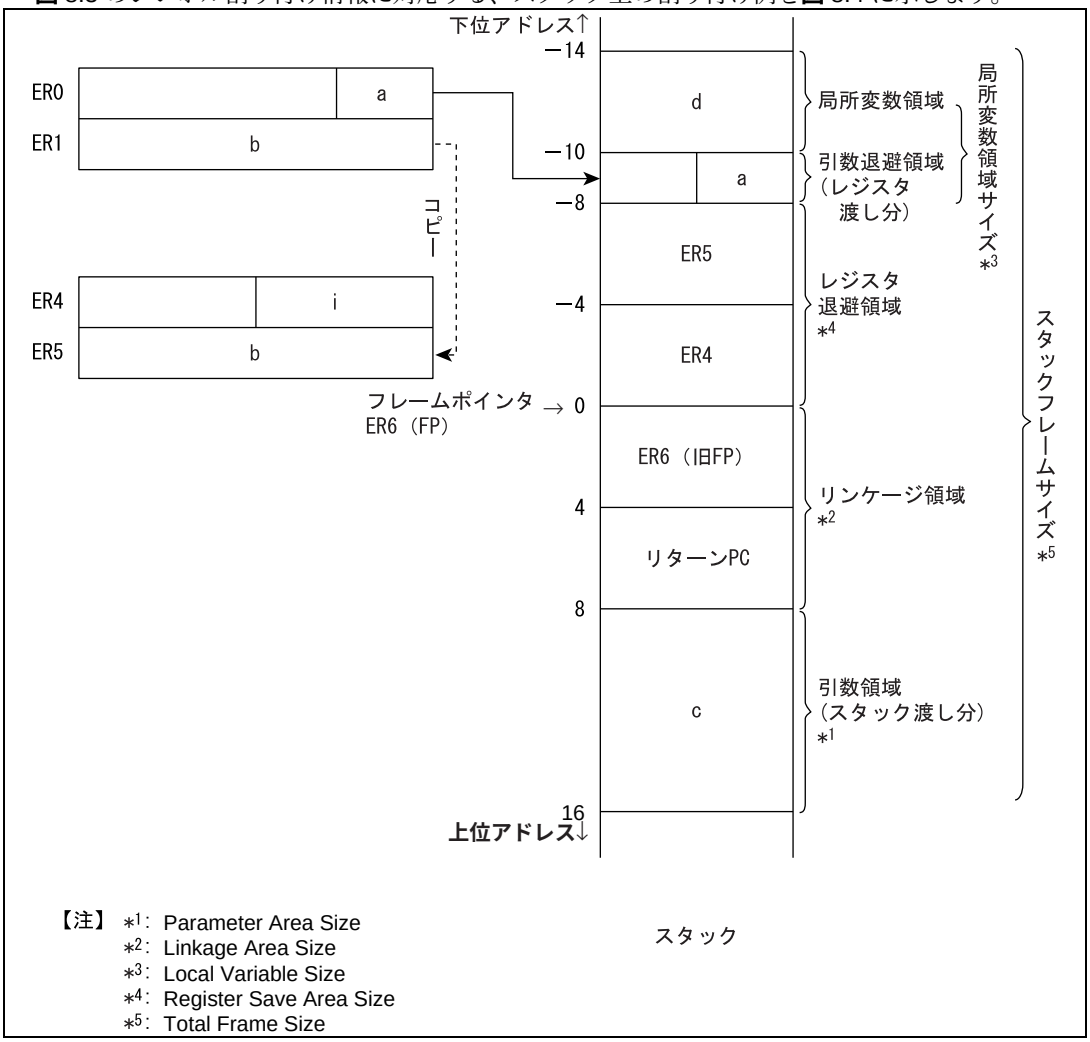


図 8.4　**スタック上の割り付け例**（cpu=2600a）

8.2.5 オブジェクト情報

オブジェクト情報にソースプログラムのリストが出力される場合と、出力されない場合のリスト例を**図** 8.5、**図** 8.6 に示します。

```
************ OBJECT LISTING ************

FILE NAME: m0251.c

SCT OFFSET CODE                LABEL      INSTRUCTION OPERAND     COMMENT
[1]   [2]   [3]                                      [4]
P                                                                 ; section
          1:    extern int sum(int);
          2:          [5]
          3:    int
          4:    sum(int x)
  00000000                     _sum:                              ; function: sum
          5:    {
          6:       int i;
          7:       int j;
          8:
          9:       j=0;
         10:
         11:       for(i=0; i<=x; i++){
  00000000 1988                    SUB.W       E0,E0
  00000002 4000                    BRA         L8:8
  00000004                     L7:
  00000004 0B58                    INC.W       #1,E0
  00000006                     L8:
  00000006 1D08                    CMP.W       R0,E0
  00000008 4F00                    BLE         L7:8
         12:          j+=1;
         13:       }
         14:
         15:       return;
         16:    }
  0000000A 5470                    RTS

【注】
  [1]   各セクションのセクション名（P, C, D, B）
  [2]   各セクションの先頭からのオフセット
  [3]   各セクションのオフセットアドレスの内容
  [4]   機械語に対応するアセンブリコード
  [5]   ソースプログラム内の行番号とソースリスト
```

図 8.5　**ソースプログラムリストが出力される場合のオブジェクト情報**
（show＝source, object　cpu＝2600a**）**

【注】show=expansion オプションを指定した場合は、常に**図** 8.6 のオブジェクト情報となります。

```
************ OBJECT LISTING ************

FILE NAME: m0251.c

SCT OFFSET CODE       C LABEL     INSTRUCTION OPERAND    COMMENT
[1]  [2]   [3]                        [4]
P                                                        ; section
                      ;*** File m0251.c   , Line 4       ; block
  00000000            _sum:                              ; function: sum
                      ;*** File m0251.c   , Line 5       ; block
                      ;*** File m0251.c   , Line 9       ; expression statement
  00000000 1911               SUB.W       R1,R1
                      ;*** File m0251.c   , Line 10      ; expression statement
  00000002 1988               SUB.W       E0,E0
                      ;*** File m0251.c   , Line 10      ; for
  00000004 4004               BRA         L8:8
  00000006            L7:
                      ;*** File m0251.c   , Line 10      ; block
                      ;*** File m0251.c   , Line 11      ; expression statement
  00000006 0981               ADD.W       E0,R1
                      ;*** File m0251.c   , Line 10      ; expression statement
  00000008 0B58               INC.W       #1,E0
  0000000A            L8:
  0000000A 1D08               CMP.W       R0,E0
  0000000C 4FF8               BLE         L7:8
                      ;*** File m0251.c   , Line 13      ; return
  0000000E 0D10               MOV.W       R1,R0
                      ;*** File m0251.c   , Line 14      ; block
  00000010 5470               RTS


【注】
 [1]  各セクションのセクション名（P,C,D,B）
 [2]  各セクションの先頭からのオフセット
 [3]  各セクションのオフセットアドレス内容
 [4]  機械語に対応するアセンブリコード
```

図 8.6　**ソースプログラムリストが出力されない場合のオブジェクト情報**
（show＝nosource, object　cpu＝2600a）

8.2.6 統計情報

統計情報の出力例を**図** 8.7 に示します。

```
******* SECTION SIZE INFORMATION *******

PROGRAM  SECTION(P):                                0x00000012 Byte(s)
CONSTANT SECTION(C):                                0x00000000 Byte(s)
DATA     SECTION(D):                                0x00000000 Byte(s)
BSS      SECTION(B):                                0x00000000 Byte(s)

TOTAL PROGRAM  SECTION: 0x00000012 Byte(s)                              [1]
TOTAL CONSTANT SECTION: 0x00000000 Byte(s)
TOTAL DATA     SECTION: 0x00000000 Byte(s)
TOTAL BSS      SECTION: 0x00000000 Byte(s)

    TOTAL PROGRAM SIZE: 0x00000012 Byte(s)



** ASSEMBLER/LINKAGE EDITOR LIMITS INFORMATION **

NUMBER OF EXTERNAL REFERENCE  SYMBOLS:        0
NUMBER OF EXTERNAL DEFINITION SYMBOLS:        1 } [2]
NUMBER OF INTERNAL/EXTERNAL SYMBOLS:          3


***** COMPILE CONDITION INFORMATION ****

COMMAND LINE: -sh=allocation -opt=0 test.c [3]
cpu         : 2600a                        [4]
```

【注】
[1] 各セクションのサイズとその合計
[2] オブジェクトプログラムの外部参照シンボルの数、外部定義シンボルの数、内部ラベルと外部ラベルの合計数
[3] コマンドライン指定内容
[4] CPU／動作モード

図 8.7 **統計情報**

【注】オプション noobject 指定時およびエラーレベル、フェータルレベルのエラーが発生した場合には、統計情報を出力しません。また、オプション code＝asmcode 指定時には、統計情報のセクションサイズ情報（SECTION SIZE INFORMATION）を出力しませんので注意してください。

8.3 アセンブルリストの参照方法

8.3.1 アセンブルリストの構成

アセンブルリストの構成と内容を**表** 8.4 **に**示します。

表 8.4 **リンケージリストの構成と内容**

No	リストの作成	内容	オプション	オプション省略時
1	**ソースリスト情報**	**ソースプログラムに関する情報を示します。**	source	**出力する**
2	**クロスリファレンスリスト情報**	**ソースプログラムのシンボルに関する情報を示します。**	cross_reference	**出力する**
3	**セクション情報リスト**	**ソースプログラムのセクションに関する情報を示します。**	section	**出力する**

【注】　全てのリストオプションは list オプション指定時に有効です。

8.3.2 ソースリスト情報

ソースリスト情報を出力します。ソースリスト情報の出力例を**図** 8.8 に示します。

```
 1                                1               .CPU 2600A:32
 2                                2       ;
 3 00000000                       3               .SECTION AAA,CODE,ALIGN=2
 4 00000000                       4       START
 5 00000000 7A0700000000          5               MOV.L   #STACK:32,SP
 6 00000006 F800                  6               MOV.B   #0:8,R0L
 7 00000008 6AA800000000          7               MOV.B   R0L,@ANS:32
 8 0000000E 7A0200001000          8               MOV.L   #DATA:32,ER2
 9                                9               .FOR.B (R1L=#1,#8,+#1)
10 00000014 F901                     S                          MOV #1,R1L
11 00000016 5800000A                 S                          BRA _$F00002
12 0000001A                          S                  _$F00000: .EQU $
13 0000001A 6828                 10                             MOV.B @ER2,R0L
14 0000001C 0B02                 11                             ADDS.L #1,ER2
15 0000001E 5E000000             12                             JSR @CHANGE:24
16                               13                      .ENDF
17 00000022                          S                          _$F00001: .EQU $
18 00000022 8901                     S                                    ADD #1,R1L
19 00000024                          S                          _$F00002: .EQU $
20 00000024 A908                     S                                    CMP #8,R1L
21 00000026 4FF2                     S                                    BLE _$F00000
22 00000028                          S                          _$F00003: .EQU $
23 00000028 0180                 14              SLEEP
24 0000002A 40D4                 15              BRA     START
25                               16      ;
26 0000002C                      17      CHANGE
27 0000002C 6A2900000000         18              MOV.B   @ANS:32,R1L
28                               19              .IF.B (R1L<LT>R0L)
29 00000032 1C98                     S                          CMP R1L,R0L
30 00000034 58F00006                 S                          BLE _$I00000
31 00000038 6AA800000000         20                             MOV.B R0L,@ANS:32
32                               21              .ENDI
33 0000003E                          S                          _$I00000: .EQU $
34 0000003E                          S                          _$I00001: .EQU $
35 0000003E 5470                 22              RTS
36                               23      ;
37 00001000                      24              .SECTION BBB,DATA,LOCATE=H'00001000
38 00001000                      25      DATA
39 00001000 03020405             26              .DATA.B H'03,H'02,H'04,H'05
40 00001004 01080607             27              .DATA.B H'01,H'08,H'06,H'07
41                               28      ;
42 00000000                      29              .SECTION CCC,DATA,ALIGN=2
43 00000000                      30      ANS
44 00000000 00000001             31              .RES.B 1
45                               32      ;
46 00000000                      33              .SECTION DDD,STACK,ALIGN=2
47 00000000 00000500             34              .RES.B H'500
48 00000500                      35      STACK
49                               36      ;
50 00000000                      37              .END    START
 (1)   (2)          (3)          (4)(5)              (6)
*****TOTAL ERRORS     0
*****TOTAL WARNINGS   0
```

図 8.8 **ソースプログラムリスト**

ソースリスト内（1）～（6）の内容は、次のとおりです。

（1）リスト行番号

（2）ロケーションカウンタ値

絶対アドレスセクションの場合は絶対アドレスを、相対アドレスセクションの場合は相対アドレスを表示します。

（3）オブジェクトコード

（4）ソース行番号

ソースファイル内でのソースステートメントの行番号です。アセンブラが展開したソースステートメントに対しては、行番号は表示しません。

（5）展開区分

プリプロセッサ機能のソースステートメント区分です。

展開区分には、次のものがあります。

I ・・・・・・ ファイルインクルード
C ・・・・・・ 条件付きアセンブルの成立、繰り返し展開、条件付き繰り返し展開
M ・・・・・・ マクロ展開
S ・・・・・・ 構造化アセンブリ展開

展開区分 I には、インクルードのネストレベルを併せて表示します。

（6）ソースステートメント

8.3.3 クロスリファレンスリスト

クロスリファレンス情報を出力します。クロスリファレンス情報の出力例を**図** 8.9 に示します。

```
*** CROSS REFERENCE LIST
NAME                            SECTION     ATTR   VALUE          SEQUENCE
AAA                             AAA         SCT    00000000       3*
ANS                             CCC                00000000       7      27      31
                                                                   43*
BBB                             BBB         SCT    00001000       37*
CCC                             CCC         SCT    00000000       42*
CHANGE                          AAA                0000002C       15     26*
DATA                            BBB                00001000        8     38*
DDD                             DDD         SCT    00000000       46*
STACK                           DDD                00000500        5     48*
START                           AAA                00000000        4*    24      50
_$F00000                        AAA         EQU    0000001A       12*    21
_$F00001                        AAA         EQU    00000022       17*
_$F00002                        AAA         EQU    00000024       11     19*
_$F00003                        AAA         EQU    00000028       22*
_$I00000                        AAA         EQU    0000003E       30     33*
_$I00001                        AAA         EQU    0000003E       34*
```

(1) (2) (3) (4) (5)

図 8.9 **クロスリファレンス**

クロスリファレンスリスト内（1）～（5）の内容は、次のとおりです。

（1）シンボル名

（2）セクション名

シンボルが含まれるセクションの名称です。最大 8 文字まで表示します。

（3）シンボル属性

シンボルの属性です。シンボルの属性には、次のものがあります。

表示なし ・・・・・・・ ラベル定義
EQU ・・・・・・・・・・ .EQU 定義
ASGN ・・・・・・・・・ .ASSIGN 定義
IMPT ・・・・・・・・・ 外部参照
EXPT ・・・・・・・・・ 外部定義
SCT ・・・・・・・・・・ セクション名
REG ・・・・・・・・・・ .REG 定義
MDEF ・・・・・・・・・ 二重定義
UDEF ・・・・・・・・・ 未定義

（4）シンボル値

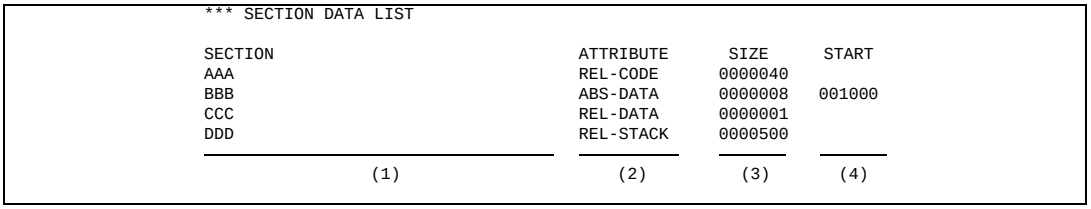
シンボルの値です。8 桁の 16 進数で表示します。

（5）シンボル定義、参照のリスト行番号

シンボルを定義、参照している行のリスト行番号です。定義行にはアスタリスク（*）を表示します。

8.3.4 セクション情報リスト

セクション情報を出力します。セクション情報の出力例を**図** 8.10 に示します。



```
*** SECTION DATA LIST

SECTION                          ATTRIBUTE      SIZE     START
AAA                              REL-CODE       0000040
BBB                              ABS-DATA       0000008  001000
CCC                              REL-DATA       0000001
DDD                              REL-STACK      0000500
                                 (1)            (2)      (3)      (4)
```

図 8.10 **セクション情報**

セクション情報リスト内（1）～（4）の内容は、次のとおりです。

（1）セクション名

（2）セクション属性

セクションの属性です。形式種別とセクション属性を表示します。

（a）形式種別

ABS ・・・・・・・ 絶対アドレス形式

REL ・・・・・・・ 相対アドレス形式

（b）セクション属性

CODE ・・・・・・・・・ コードセクション

DATA ・・・・・・・・・ データセクション

STACK ・・・・・・・ スタックセクション

DUMMY ・・・・・・ ダミーセクション

（3）セクションサイズ

セクションのサイズです。16 進数で表示します。

（4）セクション先頭アドレス

絶対アドレスセクションの先頭アドレスです。相対アドレスセクションには表示しません。

8.4 リンケージリストの参照方法

最適化リンケージエディタが出力するリンケージリストの内容と形式について説明します。

8.4.1 リンケージリストの構成

リンケージリストの構成と内容を**表** 8.5 に示します。

表 8.5 **リンケージリストの構成と内容**

No	リストの作成	内容	サブオプション	show **オプション省略時**[*1]
1	**オプション情報**	**コマンドライン、サブコマンドで指定したオプション列を表示**	—	**出力する**
2	**エラー情報**	**エラーメッセージを表示**	—	**出力する**
3	**リンケージマップ情報**	**セクション名、先頭／最終アドレス、サイズ、種別を表示**	—	**出力する**
4	**シンボル情報**	**静的定義シンボル名、アドレス、サイズ、種別をアドレス順に表示**	show=symbol	**出力しない**
		show=reference **オプション指定時には、各シンボルの参照回数、最適化実行有無も表示**	show=reference	**出力しない**
5	**シンボル削除最適化情報**	**最適化で削除したシンボルを表示**	show=symbol	**出力しない**
6	**変数アクセス最適化対象シンボル情報**	8bit/16bit **絶対アドレッシングモードでの参照回数を表示**	show=reference	**出力しない**
7	**関数アクセス最適化対象シンボル情報**	**シンボルの参照回数を表示**	show=reference	**出力しない**

【注】*1　showオプションはlistオプションを指定時のみ有効となります。

8.4.2 オプション情報

コマンドライン、サブコマンドファイルで指定したオプション列を出力します。オプション情報の出力例を**図** 8.11 に示します。（`optlnk -sub=test.sub -list -show` 指定時）

(test.subの内容)

```
INPUT test.obj
```

```
*** Options ***

-sub=test.sub
INPUT test.obj (2)   } (1)
-list
-show
```

図 8.11 **オプション情報の出力例（リンケージリスト）**

(1) コマンドライン、サブコマンドで指定したオプション列を、指定順に出力します。
(2) サブコマンドファイル test.sub 内のサブコマンドです。

8.4.3 エラー情報

エラーメッセージを出力します。エラー情報の出力例を**図** 8.12 に示します。

```
*** Error information ***

** L2310 (E) Undefined external symbol "strcmp" referred to in "test.obj"  } (1)
```

図 8.12 **エラー情報の出力例（リンケージリスト）**

(1) エラーメッセージを出力します。

8.4.4 リンケージマップ情報

各セクションの先頭／最終アドレス、サイズ、種別をアドレス順に出力します。リンケージマップ情報の出力例を**図** 8.13 に示します。

```
*** Mapping List ***

SECTION                          START      END         SIZE   ALIGN
  (1)                             (2)       (3)          (4)    (5)
P
                                 00000000  000004d6       4d6   2
C
                                 000004d6  00000533        5d   2
D
                                 00000534  0000053c         8   2
B
                                 0000053c  00004112      3bd6   2
```

図 8.13 **リンケージマップ情報の出力例（リンケージリスト）**

(1) セクション名を表示します。
(2) 先頭アドレスを表示します。
(3) 最終アドレスを表示します。
(4) セクションサイズを表示します。
(5) セクションの境界調整数を表示します。

8.4.5 シンボル情報

show=symbol オプション指定時、外部定義シンボルまたは静的内部定義シンボルのアドレス、サイズ、種別をアドレス順に出力します。また、show=reference オプション指定時は、各シンボルの参照回数、最適化実行の有無も出力します。シンボル情報の出力例を**図** 8.14 に示します。

```
*** Symbol List ***

SECTION=(1)
FILE=(2)                              START       END     SIZE
                                        (3)       (4)      (5)
  SYMBOL                              ADDR      SIZE     INFO      COUNTS  OPT
    (6)                                (7)       (8)      (9)       (10)  (11)

SECTION=P
FILE=test.obj
                                    00000000  00000428       428
  _main
                                    00000000         2   func ,g         0
  _malloc
                                    00000000        32   func ,l         0
FILE=mvn3
                                    00000428  00000490        68
  $MVN#3
                                    00000428         0   none ,g         0
```

図 8.14 **シンボル情報の出力例（リンケージリスト）**

(1) セクション名を表示します。
(2) ファイル名を表示します。
(3) (2)のファイルに含まれる該当セクションの先頭アドレスを表示します。
(4) (2)のファイルに含まれる該当セクションの最終アドレスを表示します。
(5) (2)のファイルに含まれる該当セクションのセクションサイズを表示します。
(6) シンボル名を表示します。
(7) シンボルアドレスを表示します。
(8) シンボルサイズを表示します。
(9) シンボル種別を次のように表示します。

データ種別：	func	・・・・・・	関数名
	data	・・・・・・	変数名
	entry	・・・・・	エントリ関数名
	none	・・・・・・	未設定（ラベル、アセンブラシンボル）
宣言種別 ：	g	・・・・・・・・・・	外部定義
	l	・・・・・・・・・・	内部定義

(10)シンボル参照回数を表示します。show=reference オプション指定時のみ表示します。参照回数を表示しないときは、*を表示します。
(11)最適化有無を次のように表示します。

ch	・・・・・・・・	最適化によって変更されたシンボル
cr	・・・・・・・・	最適化によって生成されたシンボル
mv	・・・・・・・・	最適化によって移動されたシンボル

8.4.6 シンボル削除最適化情報

シンボル削除最適化（optimize=symbol_delete）によって削除されたシンボルのサイズ、種別を出力します。シンボル削除最適化情報の出力例を**図** 8.15 に示します。

```
*** Delete Symbols ***

SYMBOL                                    SIZE    INFO
  (1)                                      (2)     (3)
_Version
                                            4  data ,g
```

図 8.15 **シンボル削除情報の出力例（リンケージリスト）**

(1) 削除シンボル名を表示します。
(2) 削除シンボルサイズを表示します。
(3) 削除シンボルの種別を以下のように表示します。

データ種別：	func	……	関数名
	data	……	変数名
宣言種別 ：	g	……	外部定義
	l	……	内部定義

8.4.7 変数アクセス最適化対象シンボル情報

show=reference 指定時、変数アクセス最適化（optimize=variable_access）の対象となるシンボルのサイズ、参照回数、最適化実行の有無を出力します。

8 ビット絶対アドレッシングモードまたは 16 ビット絶対アドレッシングモードでアクセス可能なシンボルを"Variable Accessible with Abs8"に、16 ビット絶対アドレッシングモードでアクセス可能なシンボルを"Variable Accessible with Abs16"に出力します。変数アクセス最適化対象シンボル情報の出力例を**図** 8.16 に示します。

```
*** Variable Accessible with Abs8 ***

SYMBOL                                    SIZE    COUNTS  OPTIMIZE
  (1)                                      (2)     (3)     (4)
_Char1Glob
                                            1         2  done

*** Variable Accessible with Abs16 ***

SYMBOL                                    SIZE    COUNTS  OPTIMIZE
  (1)                                      (2)     (3)     (4)
_IntGlob
                                            2         2
```

図 8.16 **変数アクセス最適化対象シンボル情報の出力例（リンケージリスト）**

(1) シンボル名を表示します。
(2) シンボルサイズを表示します。
(3) シンボルの参照回数を表示します。
(4) 最適化実行の有無を表示します。最適化済みであれば"done"を出力します。

8.4.8 関数アクセス最適化対象シンボル情報

show=reference オプション指定時、関数アクセス最適化（optimize=function_call）の対象となるシンボルの参照回数、最適化実行の有無を出力します。関数アクセス最適化対象シンボル情報の出力例を**図** 8.17 に示します。

```
*** Function Call ***

SYMBOL                                  COUNTS  OPTIMIZE
  (1)                                     (2)     (3)
_malloc
                                             5  done
_Proc0
                                             4
```

図 8.17 **関数アクセス最適化対象シンボル情報の出力例（リンケージリスト）**

(1) シンボル名を表示します。
(2) シンボルの参照回数を表示します。
(3) 最適化実行の有無を表示します。最適化済みであれば"done"を出力します。

8.5 ライブラリリストの参照方法

本節では、最適化リンケージエディタが出力するライブラリリストの内容と形式について説明します。

8.5.1 ライブラリリストの構成

ライブラリリストの構成と内容を**表** 8.6 に示します。

表 8.6 **ライブラリリストの構成と内容**

No	リストの作成	内容	サブオプション	show **オプション省略時**[*1]
1	**オプション情報**	**コマンドライン、サブコマンドで指定したオプション列を表示**	—	**出力する**
2	**エラー情報**	**エラーメッセージを表示**	—	**出力する**
3	**ライブラリ情報**	**ライブラリ情報を表示**	—	**出力する**
4	**ライブラリ内モジュール、セクション、シンボル情報**	**ライブラリ内モジュールを表示**	—	**出力する**
		show=symbol **オプション指定時には、モジュール内シンボル名一覧も表示**	show=symbol	**出力しない**
		show=section **オプション指定時には、各モジュール内セクション名、シンボル名一覧も表示**	show=section	**出力しない**

【注】*1 showオプションは、listオプション指定時にのみ有効です。

8.5.2 オプション情報

コマンドライン、サブコマンドファイルで指定したオプション列を出力します。オプション情報の出力例を**図** 8.18 に示します。 （`optlnk -sub=test.sub -list -show` 指定時）

```
 test.sub の内容
form   library
in     adhry.obj
output test.lib
```

```
*** Options ***

-sub=test.sub
form   library      ┐
in     adhry.obj    ┘(2)  ┐
output test.lib           ┘(1)
-list                     ┐
-show                     ┘
```

図 8.18 **オプション情報の出力例（ライブラリリスト）**

(1) コマンドライン、サブコマンドで指定したオプション列を、指定順に出力します。
(2) サブコマンドファイル `test.sub` 内のサブコマンドです。

8.5.3 エラー情報

エラーメッセージを出力します。エラー情報の出力例を**図** 8.19 に示します。

```
*** Error information ***

** L1200 (W) Backed up file "main.lib" into "main.lbk"   }(1)

```

図 8.19 **エラー情報の出力例（ライブラリリスト）**

(1) エラーメッセージを出力します。

8.5.4 ライブラリ情報

ライブラリの種別を出力します。ライブラリ情報の出力例を**図** 8.20 に示します。

```
*** Library Information ***

LIBRARY NAME=test.lib   (1)
CPU=H8S                 (2)
ENDIAN=Big              (3)
ATTRIBUTE=system        (4)
NUMBER OF MODULE=1      (5)
```

図 8.20 **ライブラリ情報の出力例（ライブラリリスト）**

(1) ライブラリ名を表示します。
(2) cpu 名を表示します。
(3) エンディアン種別を表示します。
(4) ライブラリファイルの属性がシステムライブラリかユーザライブラリかを表示します。
(5) ライブラリ内モジュール数を表示します。

8.5.5 ライブラリ内モジュール、セクション、シンボル情報

ライブラリ内のモジュール一覧を出力します。

また、show=symbol オプション指定時にはモジュール内シンボル名一覧、show=section オプション指定時にはモジュール内セクション名、シンボル名一覧も出力します。

ライブラリ内モジュール、セクション、シンボル情報の出力例を**図** 8.21 に示します。

```
*** Library List ***

MODULE    LAST UPDATE
  (1)         (2)
  SECTION
    (3)
    SYMBOL
      (4)
adhry
          29-Feb-2000 12:34:56
  P
    _main
    _Proc0
    _Proc1
  C
  D
    _Version
  B
    _IntGlob
    _CharGlob
```

図 8.21 **ライブラリ内モジュール、セクション、シンボル情報の出力例（ライブラリリスト）**

(1) モジュール名を表示します。
(2) モジュールを登録した日付を表示します。モジュールが更新された場合は、最新の更新日付を表示します。
(3) モジュール内セクション名を表示します。
(4) セクション内をシンボル表示します。

9. プログラミング

9.1 プログラムの構造

9.1.1 セクション

C/C++コンパイラ、アセンブラが出力するオブジェクトプログラムの実行命令、データの各領域は、セクションを構成します。セクションは、メモリ上の配置を行う最小単位です。セクションの性質には、以下の項目があります。

セクション属性
- code　実行命令を格納します。
- data　データを格納します。
- stack　スタック領域です。

形式種別
- 相対アドレス形式・・・・・・・・最適化リンケージエディタで再配置可能なセクションです。
- 絶対アドレス形式・・・・・・・・アドレス決定済みのセクションです。最適化リンケージエディタで再配置できません。

初期値
- プログラム実行開始時の初期値の有無です。同一セクション内で初期値があるデータと初期値がないデータは混在できません。一つでも初期値があると、初期値のない領域は0で初期化します。

書き込み操作
- プログラム実行時における書き込み操作の可／不可を示します。

境界調整数
- セクションを割り付けるアドレスの補正値です。最適化リンケージエディタでは、境界調整数の倍数アドレスになるよう、アドレスを補正します。

9.1.2 C/C++プログラムのセクション

C/C++プログラム、標準ライブラリの使用メモリ領域の種類とセクションとの対応を**表** 9.1 に示します。

表 9.1　**メモリ領域の種類とその性質の概要**

	名称	セクション 名称	セクション 属性	形式 種別	初期値 書き込み操作	境界 調整数	内容
1	プログラム領域	P*[1]	code	相対形式	有 不可	2byte	機械語を格納
2	定数領域	C*[1]	data	相対形式	有 不可	2byte	const 型のデータを格納
3	初期化データ領域	D*[1]	data	相対形式	有 可	2byte	初期値のあるデータを格納
4	未初期化データ領域	B*[1]	data	相対形式	無 可	2byte	初期値のないデータを格納

	名称	セクション		形式	初期値	境界	内容
		名称	属性	種別	書き込み操作	調整数	
5	定数領域 (8bit アドレス空間)	$ABS8C*[1]	data	相対形式	有 不可	1byte	abs8 オプション、または_ _abs8, #pragma abs8 で指定された const 型の 8 ビットデータを格納
6	初期化データ領域 8bit アドレス空間)	$ABS8D*[1]	data	相対形式	有 可	1byte	abs8 オプション、または_ _abs8, #pragma abs8 で指定された初期値のある 8 ビットデータを格納
7	未初期化データ領域 8bit アドレス空間)	$ABS8B*[1]	data	相対形式	無 可	1byte	abs8 オプション、または_ _abs8, #pragma abs8 で指定された初期値のない 8 ビットデータを格納
8	定数領域 (16bit アドレス空間)	$ABS16C*[1]	data	相対形式	有 不可	2byte	abs16 オプション指定時、または _ _abs16, #pragma abs16 で指定された const 型のデータを格納
9	初期化データ領域(16bit アドレス空間)	$ABS16D*[1]	data	相対形式	有 可	2byte	abs16 オプション指定時、または _ _abs16, #pragma abs16 で指定された初期値のあるデータを格納
10	未初期化データ領域(16bit アドレス空間)	$ABS16B*[1]	data	相対形式	無 可	2byte	abs16 オプション指定時、または _ _abs16, #pragma abs16 で指定された初期値のないデータを格納
11	関数アドレス領域 (メモリ間接空間)	$INDIRECT*[1]	data	相対形式	有 不可	2byte	indirect=normal オプション指定時、または_ _indirect, #pragma indirect で指定された関数のアドレスを格納
12	関数アドレス領域 (拡張メモリ間接空間)	$EXINDIRECT*[1]	data	相対形式	有 不可	2byte	indirect=extended オプション指定時、または_ _indirect_ex で指定された関数のアドレスを格納
13	関数アドレス領域(メモリ間接空間)	$VECTxx xx: ベクタ番号	data	絶対形式	有 不可	2byte	_ _indirect, #pragma indirect, _ _indirect_ex, _ _interrupt, #pragma interrupt, _ _entry, #pragma entry の vect=xx で指定された関数のアドレスを格納
14	1 バイトデータ領域	yy$1*[2] yy:C*[1],D*[1],B*[1], $ABS16C*[1], $ABS16D*[1], $ABS16B*[1]	data	相対形式	—	1byte	align=4 オプション指定時の 1 バイトデータを扱う領域 各セクションごとに生成される
15	4 バイトデータ領域	yy$4*[2] yy:C*[1],D*[1],B*[1], $ABS16C*[1], $ABS16D*[1], $ABS16B*[1]	data	相対形式	—	4byte	align=4 オプション指定時の 4 バイトデータを扱う領域 各セクションごとに生成される
16	初期化データセクションのアドレス領域	C$DSEC*[3]	data	相対形式	有 不可	2byte	初期値化データ領域セクションの ROM アドレス、ROM 上の最終アドレス、RAM アドレス
17	未初期化データセクションのアドレス領域	C$BSEC*[3]	data	相対形式	有 不可	2byte	未初期値化データ領域セクションのアドレス、最終アドレスを格納
18	C++初期処理/後処理データ領域	C$INIT*[3]	data	相対形式	有 不可	2byte	グローバルクラスオブジェクトに対して呼び出されるコンストラクタおよびデストラクタのアドレスを格納

	名称	セクション		形式種別	初期値 書き込み操作	境界調整数	内容
		名称	属性				
19	C++仮想関数表領域	C$VTBL*[3]	data	相対形式	有 不可	2byte	クラス宣言中に仮想関数があるときに仮想関数をコールするためのデータを格納
20	スタック領域	S	stack	相対形式	無 可	2byte	プログラム実行に必要な領域 「9.2.1 (2) 動的領域の割り付け」参照
21	ヒープ領域	—	—	相対形式	無 可	—	ライブラリ関数 malloc、realloc、calloc、new で使用する領域 「9.2.1 (2) 動的領域の割り付け」参照

【注】 *1 コンパイラオプション section または拡張子#pragma section、#pragma abs8 section、#pragma abs16 section、#pragma indirect section でセクション名を切り替えることができます。

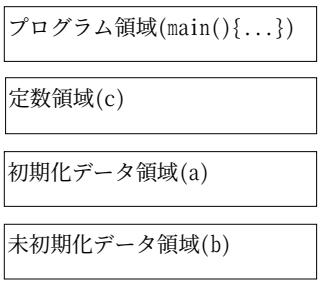
*2 yy にはデータが分割される前のデータセクション名が入ります。例：C → C$1、C$4

*3 コンパイラオプション section=C=zz を指定すると接頭辞 C は zz に切り替わります。

例 1：C プログラムとコンパイラ生成セクションとの対応をプログラム例を用いて示します。

```
int a=1;
char b;
const int c=0;
void main(){
 ...
}
```

Cプログラム

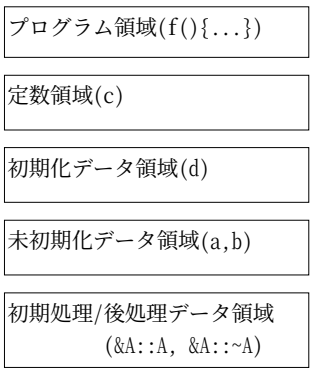


コンパイラが生成する領域と格納されるデータ

例 2：C++**プログラムとコンパイラ生成セクションとの対応をプログラム例を用いて示します。**

```
class A{
  int m;
public:
  A(int p);
  ~A();
};
A a(1);
int b;
extern const char c='a';
int d=1;
void f(){...}
```

C++プログラム



コンパイラが生成する領域と格納されるデータ

9.1.3 アセンブリプログラムのセクション

アセンブリプログラムでは、".section"制御命令を用いて、セクションの開始や属性、形式種別を宣言します。".section"制御命令の宣言形式は次のとおりです。詳細は**「11.3 アセンブラ制御命令」**を参照してください。

```
.section <セクション名>[,<セクション属性>[,<形式種別>]]
<形式種別>: 相対アドレス形式セクションの場合、align=<境界調整数>
            絶対アドレス形式セクションの場合、locate=<アドレス値>
```

例：アセンブリプログラムのセクション宣言例を示します。

```
                .CPU 2600A
                .OUTPUT DBG
SIZE :          .EQU 8
;
                .SECTION A,CODE,ALIGN=2             …… (1)
START:
                MOV.L #CONST:32,ER0
                MOV.L #DATA:32,ER1
                MOV.L #SIZE:32,ER2
LOOP:
                CMP.L #0:32,ER2
                BEQ EXIT
                MOV.B @ER0,R3L
                MOV.B R3L,@ER1
                ADD.L #1:32,ER0
                ADD.L #1:32,ER1
                SUB.L #1:32,ER2
                BRA LOOP
EXIT:
                SLEEP
                BRA START
;
                .SECTION B,DATA,LOCATE=H'00001000   …… (2)
CONST
                .DATA.B H'01,H'02,H'03,H'04
                .DATA.B H'05,H'06,H'07,H'08
;
                .SECTION C,STACK,ALIGN=2            …… (3)
DATA
                .RES.B SIZE
;
                .END START
```

(1) セクション名 A、境界調整数 2、相対アドレス形式の code セクションを宣言しています。

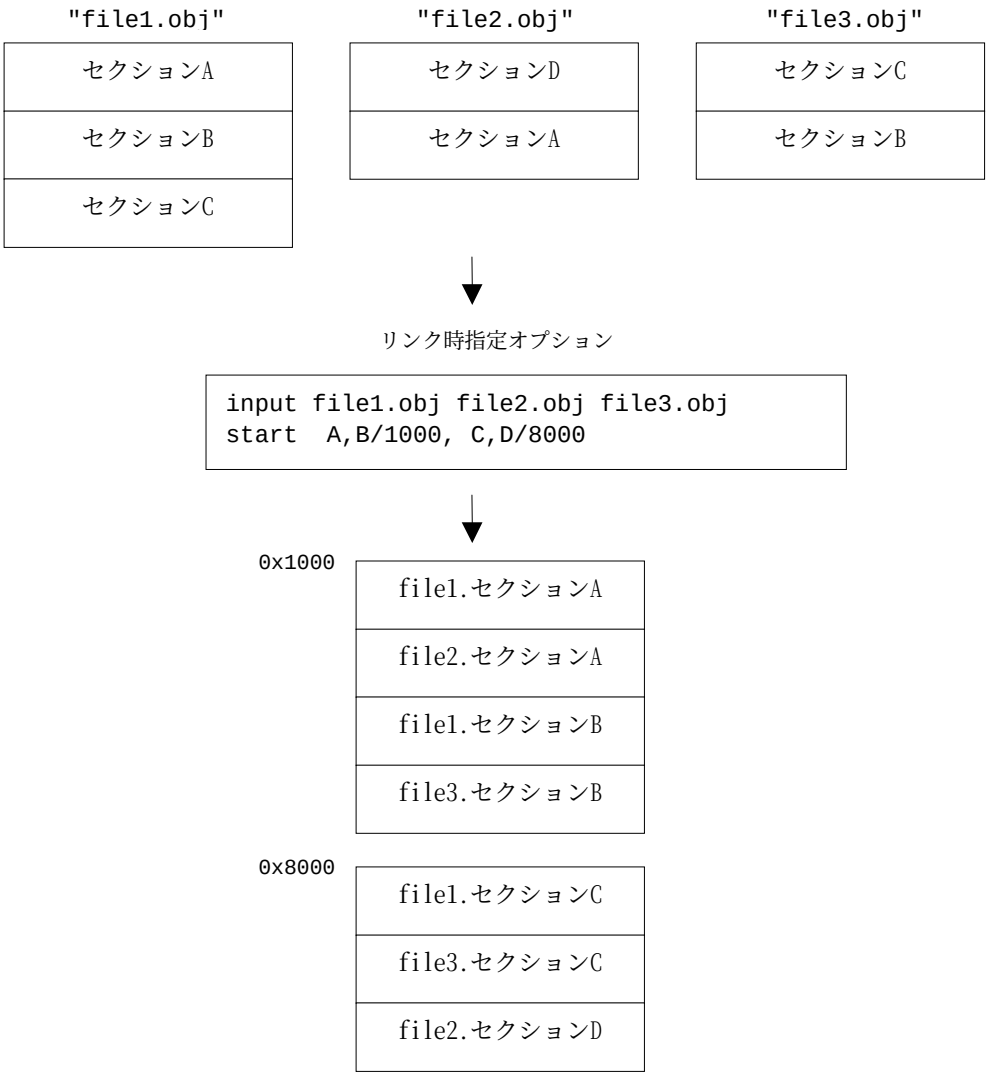
(2) セクション名 B、割り付けアドレス H'1000、絶対アドレス形式の data セクションを宣言しています。

(3) セクション名 C、境界調整数 2、相対アドレス形式の stack セクションを宣言しています。

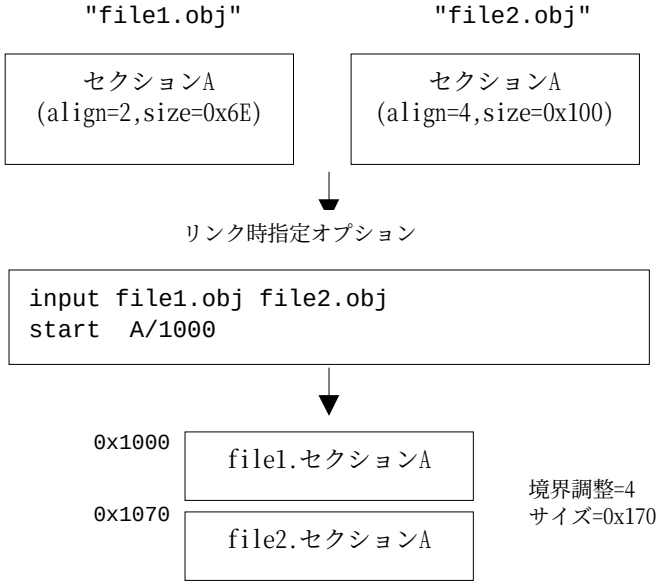
9.1.4 セクションの結合

最適化リンケージエディタでは、入力オブジェクトプログラム内の同一セクションを結合し、start オプションによって指定されたアドレスに割り付けます。

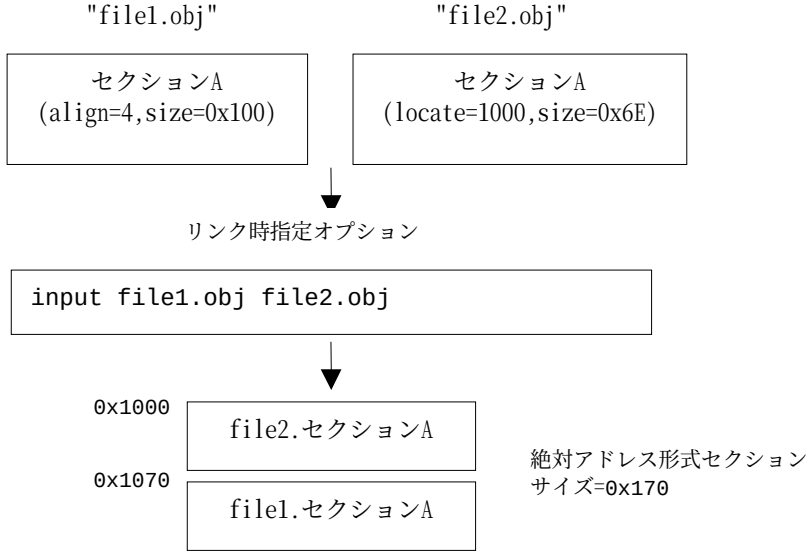
(1) 異なるファイルの同名セクションは、ファイルの入力順に連続して割り付けます。



(2) 境界調整数の異なる同名セクションは、境界調整後に結合します。セクションの境界調整数は大きい方に合わせます。

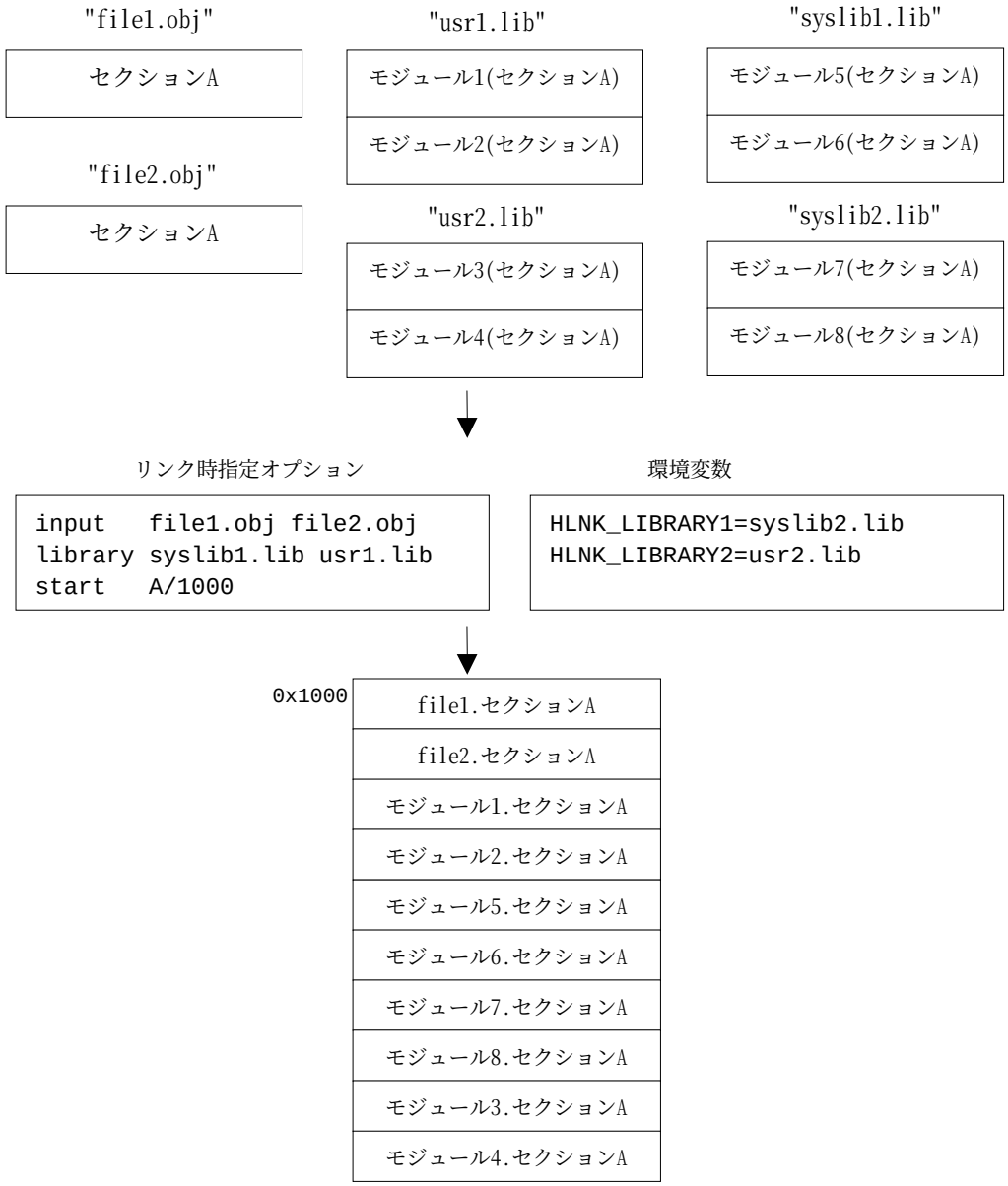


(3) 同名セクションに絶対アドレス形式と相対アドレス形式が含まれている場合、絶対アドレス形式オブジェクトの後に相対アドレス形式オブジェクトを結合します。リロケータブルファイル（form=relocate）出力指定時でも、当該セクションは絶対アドレス形式セクションになります。



(4) 同名セクション内オブジェクトの結合順序に関する規則は以下のとおりです。

input オプションまたはコマンドライン上の入力ファイル指定順
library オプションのユーザライブラリ指定順およびライブラリ内モジュール入力順
library オプションのシステムライブラリ指定順およびライブラリ内モジュール入力順
環境変数(HLNK_LIBRARY1～3)のライブラリ指定順およびライブラリ内モジュール入力順



9.2 初期設定プログラムの作成

本章では、プログラムを H8SX、H8S/2600、H8S/2000、H8/300H または H8/300 を応用したシステムに組み込む方法を説明します。

プログラムをシステムに組み込むには、以下の準備が必要です。

・ メモリの割付け

各セクション、スタック領域、ヒープ領域を、システム上の ROM、RAM のメモリ領域に割り当てる必要があります。

・ プログラム実行環境の設定

プログラムの実行環境を設定する処理には、レジスタの初期設定、メモリ領域の初期化、プログラムの起動があります。

また、入出力等の C/C++ライブラリ関数をご使用になる場合は、実行環境の設定時にライブラリの初期化をする必要があります。特に入出力（stdio.h、ios、streambuf、istream、ostream）とメモリ割り付け（stdlib.h、new）の機能をご使用になる場合は、システムごとに、低水準の入出力ルーチンやメモリ割り付けルーチンを作成する必要があります。
プログラムの終了処理を行う C ライブラリ関数（exit、atexit、abort 関数）をご使用になる場合も、別途ユーザシステムに合わせてこれらの関数を作成する必要があります

9. 2. 1 ではプログラムのメモリ領域のアドレスを決定する考え方を説明し、実際にアドレスを決定するための最適化リンケージエディタのオプション指定方法について実例を挙げて説明します。

9. 2. 2 では実行環境設定の項目を説明し、設定プログラムの実例について説明します。
また、ライブラリ関数の初期設定処理、低水準ルーチンの作成方法および終了処理関数の作成例についても説明します。

9.2.1 メモリ領域の割り付け

オブジェクトプログラムをシステムに組み込むためには、プログラムが使用するメモリ領域のサイズを決定し、それぞれの領域を適切なメモリアドレスに割り付ける必要があります。

プログラムが使用するメモリ領域には、プログラム中の関数に対応する実行命令や外部データ定義で宣言したデータ領域のように静的に割り付ける領域と、スタック領域のように動的に割り付ける領域があります。以下、各領域の割り付け方を説明します。

(1) 静的領域の割り付け

(a) 静的領域の内容

スタック領域、ヒープ領域以外のセクションは静的領域に割り付けます。

C/C++プログラムの各セクション（プログラム領域、定数領域、初期化データ領域、未初期化データ領域、関数アドレス領域、初期化データセクションアドレス領域、未初期化データセクションアドレス領域、C++初期処理/後処理データ領域、C++仮想関数表領域）は静的領域に割り付けます。

(b) サイズの算出法

静的領域のサイズは、コンパイラ、アセンブラが生成するオブジェクトプログラムサイズとC/C++プログラムが使用するライブラリ関数のサイズの合計になります。

オブジェクトプログラムをリンクしたあと、リンケージリストのリンケージマップ情報にライブラリを含めた各セクションのサイズを出力しますので、静的領域のサイズを知ることができます。**図** 9.1 にリンケージリスト内リンケージマップ情報の例を示します。

```
*** Mapping List ***

SECTION                          START      END        SIZE   ALIGN
  (1)                              (2)       (3)        (4)    (5)
P
                                 00000000  000004d6       4d6   2
C
                                 000004d6  00000533        5d   2
D
                                 00000534  0000053c         8   2
B
                                 0000053c  00004112      3bd6   2
```

図 9.1 **リンケージリスト内リンケージマップ情報例**

コンパイル、アセンブル単位のセクションサイズは、コンパイルリスト内統計情報およびアセンブルリスト内セクション情報に出力されます。**図** 9.2 にコンパイルリスト内統計情報の例、**図** 9.3 にアセンブルリスト内セクション情報の例を示します。

```
******* SECTION SIZE INFORMATION *******
PROGRAM   SECTION (P):                          0x00000080 Byte(s)
CONSTANT  SECTION (C):                          0x00000004 Byte(s)
DATA      SECTION (D):                          0x00000004 Byte(s)
BSS       SECTION (B):                          0x00000004 Byte(s)

TOTAL  PROGRAM   SECTION:  0x00000080 Byte(s)
TOTAL  CONSTANT  SECTION:  0x00000004 Byte(s)
TOTAL  DATA      SECTION:  0x00000004 Byte(s)
TOTAL  BSS       SECTION:  0x00000004 Byte(s)

TOTAL  PROGRAM  SIZE:  0x0000008C Byte(s)
```

図 9.2　**コンパイルリスト内統計情報例**

```
                  *** SECTION DATA LIST

                  SECTION                         ATTRIBUTE    SIZE            START

                  P                               REL-CODE    000000604
                  D                               REL-DATA    000000008
                  C                               REL-DATA    00000005D
                  B                               REL-DATA    000003BD6
```

図 9.3 **アセンブルリスト内セクション情報例**

標準ライブラリを使用しない場合は、ファイル単位のセクションサイズの合計が静的領域のサイズになります。

標準ライブラリを使用している場合、各セクションのメモリ領域サイズにはライブラリ関数の使用するメモリ領域サイズが加算されます。コンパイラが提供する標準ライブラリの中には、C 言語仕様で規定した C ライブラリ関数や組み込み向け C++クラスライブラリ以外に、プログラムを実行する上で必要な算術演算を行うルーチン（実行時ルーチン）を含みます。そのため、ソースプログラム上でライブラリ関数の使用を指定しなくても、標準ライブラリが必要な場合がありますので注意してください。

プログラムで使用する実行時ルーチンは、コンパイラが出力するコンパイルリストのシンボル割り付け情報から知ることができます。以下に具体例を示します。

Cプログラム

```
long a,b;
main()
{
        a *= b;
}
```

Cコンパイラ出力のシンボル割り付け情報

```
******* STACK FRAME INFORMATION *******

FILE NAME:  main.c

Function (File main.c   , Line     2): main

Parameter Area Size     : 0x00000000 Byte(s)
Linkage Area Size       : 0x00000008 Byte(s)
Local Variable Size     : 0x00000000 Byte(s)
Temporary Size          : 0x00000000 Byte(s)
Register Save Area Size : 0x00000000 Byte(s)
Total Frame Size        : 0x00000008 Byte(s)

Used Runtime Library Name
$MULL$3                                  ;実行時ルーチン
```

(c) ROM、RAM の割り付け

プログラムを ROM 化する場合、セクションの初期値の有無、書き込み操作の可／不可で、ROM に割り付けるか RAM に割り付けるかが決まります。

C/C++プログラムの各セクションを ROM 化する場合は、以下のように ROM と RAM に分けて割り付けます。

- プログラム領域　（セクションP）　→ ROM
- 定数領域　（セクションC、$ABS8C、$ABS16C）　→ ROM
- 未初期化データ領域（セクションB、$ABS8B、$ABS16B）　→ RAM
- 初期化データ領域　（セクションD、$ABS8D、$ABS16D）　→ ROM、RAM（(d)**参照**）
- 関数アドレス領域　（セクション$INDIRECT、$EXINDIRECT）　→ ROM
- 初期化データセクションアドレス領域　（セクションC$DSEC）　→ ROM
- 未初期化データセクションアドレス領域（セクションC$BSEC）　→ ROM
- 初期処理データ領域*1　（セクションC$INIT）　→ ROM
- 仮想関数表領域*2　（セクションC$VTBL）　→ ROM

【注】　*1　C++プログラムでグローバルクラスオブジェクトがあるときにコンパイラが生成します。

*2　C++プログラムで仮想関数宣言があるときにコンパイラが生成します。

(d) 初期化データ領域の割り付け

初期化データ領域のように、初期値を持ち、プログラム実行時に値の変更が可能なセクションは、リンク時には ROM 上に置き、プログラムの実行開始時に RAM 上にコピーします。したがって、最適化リンケージエディタの rom オプションを用いて、ROM 上と RAM 上に、二重に領域をとる必要があります。指定例については、**「(e) メモリの割り付け例とリンク時のアドレス指定方法」**を参照してください。また ROM 上から RAM 上へ値をコピーするセクションの初期設定については、**「**9.2.2 (2) **初期設定」**で説明します。

(e) メモリの割り付け例とリンク時のアドレス指定方法

アブソリュートロードモジュール作成時に、最適化リンケージエディタのオプションまたはサブコマンドで各セクション毎に割り付ける領域のアドレスを指定します。以下、静的領域のメモリ割り付け例とリンク時の指定方法について説明します。

図 9.4 に、H8S/2600 アドバンストモードにおける静的な領域の割り付け例を示します。

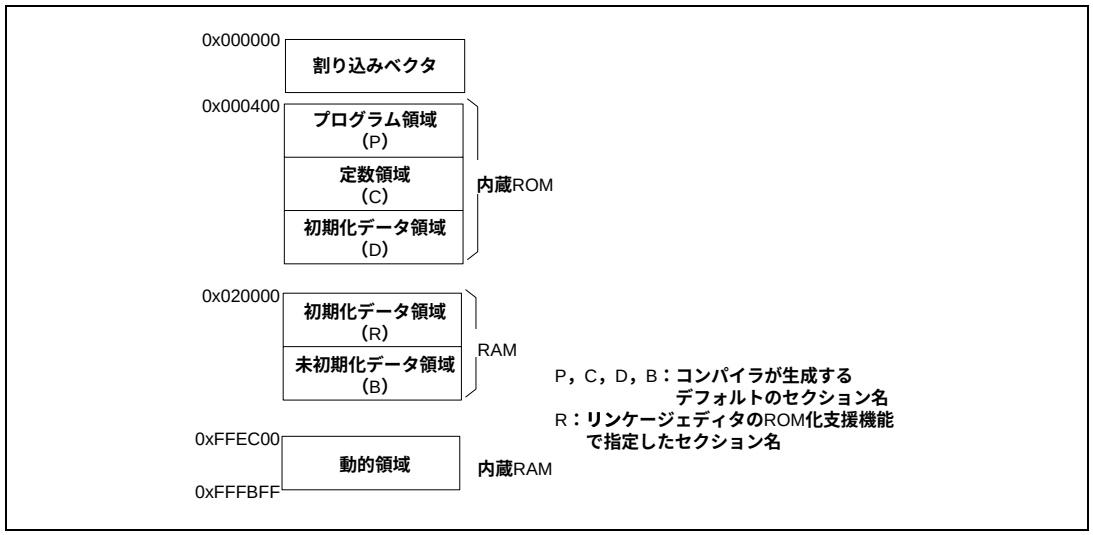


図 9.4　**静的な領域の割り付け例**

図 9.4 に示すメモリ割り付けを行う場合、リンク時に以下のサブコマンドを指定します。

```
ROM△D/R                          ……… [1]
START△P,C,D/400,R,B/20000        ……… [2]
```

説明：[1] セクション名 D と同じ大きさのセクション R を出力ロードモジュールに確保します。また、セクションDに割り付けられたシンボルを参照している場合、セクションR上のアドレスとなるようリロケーションします。セクションDはROM上、セクションRはRAM上の初期化データセクション名になります。

[2] セクションP、C、Dを内蔵ROMのアドレス0x400から連続した領域に割り付けます。また、セクションR、BをRAMのアドレス0x20000から連続したアドレスに割り付けます。

(2) 動的領域の割り付け

(a) 動的領域の内容

C/C++プログラムで使用する動的領域には、以下の二つがあります。

スタック領域

ヒープ領域（メモリ割り付けライブラリ関数で使用）

(b) スタック領域サイズの算出法

C/C++プログラム、標準ライブラリの使用するスタック領域は、最適化リンケージエディタの stack オプションを指定してスタック情報ファイルを出力すると、スタック解析ツールを用いて最大使用量を算出することができます。スタック解析ツールの使用方法については、**「6 スタック解**

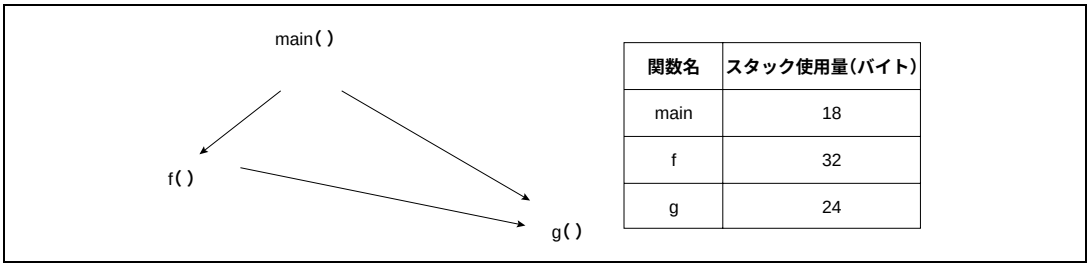
析ツール操作方法」を参照してください。
アセンブリプログラムの使用するスタック領域は、スタック解析ツールでは算出できません。以下の C/C++プログラムのスタック使用量計算の考え方を参考にアセンブリプログラムのスタック使用量を算出し、スタック解析ツールで算出したスタック使用量に加算してください。

- **C/C++プログラムのスタック使用量計算の考え方**

C/C++プログラムの使用するスタック領域は、関数呼び出しのたびにスタック上に割り付け、関数のリターン時に解放します。スタック領域のサイズを算出するためには、まず各関数ごとのスタック使用量を算出し、関数の呼び出し関係から実際のスタック使用量を算出します。
各関数の使用するスタック領域は、コンパイルリストのシンボル割り付け情報（Total Frame Size）から知ることができます。

```
******** STACK FRAME INFORMATION ********
FILE NAME: test.c

Function (File test.c   , Line   2) : main

  Optimize Option Specified : No Allocation Information Available

Parameter Area Size       : 0x00000008 Byte(s)
Linkage Area Size         : 0x00000004 Byte(s)
Local Variable Size       : 0x00000002 Byte(s)
Temporary Size            : 0x00000000 Byte(s)
Register Save Area Size   : 0x00000004 Byte(s)
Total Frame Size          : 0x00000012 Byte(s)
```

関数の使用するスタック領域サイズは、Total Frame Size の 0x12 つまり 18 バイトとなります。
関数の呼び出し関係と各関数のスタック使用量の例を**図** 9.5 に示します。
この場合、関数 f を介して関数 g が呼ばれた時のスタック領域のサイズは、**表** 9.2 によって計算します。



関数名	スタック使用量（バイト）
main	18
f	32
g	24

図 9.5　**関数呼び出しの関係とスタック使用量の例**

表 9.2　スタックサイズの計算例

呼出し経路	スタックサイズ計	
main（18）→f（32）→g（24）	74	←スタック使用量（最大値）
main（18）→g（24）	42	

このように、呼び出しレベルの一番深い関数についてスタック領域のサイズを計算し、その最大値（この場合 74 バイト）のスタック領域を割り付けます。

- **スタック使用量の計算に関する注意事項**

Ver.4 までおよび Ver.6.0 で CPU が H8SX でない場合と、Ver.6.0 で CPU が H8SX である場合とではスタック使用量の考え方が異なります。本説明では、以下、Ver.4 までおよび Ver.6.0 で CPU が H8SX でない場合を A 方式と呼び、Ver.6.0 で CPU が H8SX である場合を B 方式と呼びます。A 方式でコンパイルした関数と B 方式でコンパイルした関数を相互に呼び出す場合にスタック使用量の計算に注意が必要です。

A 方式と B 方式とではパラメタをスタックで渡すときの SP（スタックポインタ）の動きが異なります。A 方式では下の例の[1] のようにパラメタをスタックで渡すときにプッシュ命令またはプリデクリメントアドレッシングモード(@-SP)を利用して SP をデクリメントしてパラメタをスタックに格納します。この場合、関数呼び出しから戻った時に、下の例の[2] のようにスタックで渡したパラメタのサイズだけ SP をインクリメントして、スタックで渡したパラメタの領域を解放します。この方式ではスタックで渡すパラメタ領域のサイズが呼び出す関数毎に異なり、そのサイズは下の例の[3] のように呼び出し先関数のスタックフレームサイズの Parameter Area Size に算入されます。

一方、B 方式では関数内で使用するスタックサイズの最大値をコンパイラがあらかじめ計算して下の例の[4] のように関数入口でその最大値を確保します。関数出口まで SP は一定の値を保ち下の例の[6] のように関数出口で SP を呼び出し前の状態に戻します。この場合、関数のパラメタをスタックで渡すときに下の例の[5] のように SP はそのままで SP から 0 または正のオフセット値でパラメタをスタックに格納します。この方式ではスタックで渡すパラメタ領域の最大値が下の例の[7] のように呼び出し元関数のスタックフレームサイズの Temporary Size に算入されます。

次に示す CASE 1 と CASE 4 のように呼び出し元と呼び出し先の関数の方式が同じ場合は関数 g から関数 f を呼び出したときのスタック使用量はそれぞれの Total Frame Size を加算して 12 バイトと正しい値になります。CASE 2 のように A 方式から B 方式を呼ぶ場合の Total Frame Size を加算すると８バイトとなりますが、スタックで渡すパラメタ領域のサイズ算入されず、スタック使用量を正しい値より少なく見積もってしまいます。CASE 3 のように B 方式から A 方式を呼ぶ場合は Total Frame Size を加算すると 16 バイトとなり、スタックで渡すパラメタ領域のサイズが２重に算入され、スタック使用量を正しい値より多く見積もってしまいます。

このような過小見積りや過大見積りを避けるには A 方式と B 方式を混在しないか、A 方式から B 方式、B 方式から A 方式を呼んでいる箇所を探してスタック使用量を補正します。

スタック使用量
CASE 1: A 方式の関数 g から A 方式の関数ｆを呼び出す場合：８＋４＝１２
CASE 2: A 方式の関数 g から B 方式の関数ｆを呼び出す場合：４＋４＝８
CASE 3: B 方式の関数 g から A 方式の関数ｆを呼び出す場合：８＋８＝１６
CASE 4: B 方式の関数 g から B 方式の関数ｆを呼び出す場合：８＋４＝１２

例： ソースプログラム

```
int f(struct S);
void g(void);
struct S{long p;} st;
int x;
int f(struct S s){
    return 0;
}
void g(void)
{
    x=f(st);
}
```

A 方式

```
_f:
  SUB.W   R0,R0
  RTS
_g:
  MOV.L   @_st:32,ER0
  PUSH.L  ER0       ;[1]
  BSR     _f:8
  ADDS.L  #4,SP     ;[2]
  MOV.W   R0,@_x:32
  RTS
```

B 方式

```
_f:
  SUB.W       R0,R0
  RTS
_g:
  ADD.W       #-4:16,R7   ;[4]
  MOV.L       @_st:32,ER0
  MOV.L       ER0,@SP     ;[5]
  BSR         _f:8
  MOV.W       R0,@_x:32
  ADDS.L      #4,SP       ;[6]
  RTS
```

関数 f

```
Parameter Area Size     : 0x00000004 Byte(s) [3]   0x00000000 Byte(s)
Linkage Area Size       : 0x00000004 Byte(s)       0x00000004 Byte(s)
Local Variable Size     : 0x00000000 Byte(s)       0x00000000 Byte(s)
Temporary Size          : 0x00000000 Byte(s)       0x00000000 Byte(s)
Register Save Area Size : 0x00000000 Byte(s)       0x00000000 Byte(s)
Total Frame Size        : 0x00000008 Byte(s)       0x00000004 Byte(s)
```

関数 g

```
Parameter Area Size     : 0x00000000 Byte(s)       0x00000000 Byte(s)
Linkage Area Size       : 0x00000004 Byte(s)       0x00000004 Byte(s)
Local Variable Size     : 0x00000000 Byte(s)       0x00000000 Byte(s)
Temporary Size          : 0x00000000 Byte(s)       0x00000004 Byte(s) [7]
Register Save Area Size : 0x00000000 Byte(s)       0x00000000 Byte(s)
Total Frame Size        : 0x00000004 Byte(s)       0x00000008 Byte(s)
```

(c) ヒープ領域サイズの算出法

ヒープ領域で使用するメモリ領域のサイズは、C/C++プログラム内でメモリ管理ライブラリ関数（calloc、malloc、realloc、new 関数）によって割り付ける領域の合計です。ただし、メモリ管理ライブラリ関数は、1 回の呼び出しのたびに管理用の領域として 4 バイト（cpu＝H8SXN、H8SXM、H8SXA(ptr16 オプション有り) 、 H8SXX(ptr16 オプション有り)、2600n、2000n、300hn、300 指定時）または 8 バイト（cpu＝H8SXA(ptr16 オプション無し) 、H8SXX(ptr16 オプション無し)、2600a、2000a、300ha 指定時）を使用しますので、実際に確保する領域サイズにこの管理領域のサイズを加えて計算してください。

また、コンパイラはヒープ領域をユーザ指定のメモリサイズ（_sbrk_size）の単位で管理しています。_sbrk_size の指定方法は**「9.2.2 (5) C/C++ライブラリ関数の初期設定_INITLIB」**を参照してください。ヒープ領域として確保する領域サイズ（HEAPSIZE）は次のように計算してください。

HEAPSIZE＝_sbrk_size × n（n≧1）

（メモリ管理ライブラリによって割り付ける領域サイズ）＋管理領域サイズ≦HEAPSIZE

入出力ライブラリ関数は、内部処理の中でメモリ管理ライブラリ関数を使用しています。入出力の中で割り付ける領域のサイズは、

cpu＝H8SXN、H8SXM、H8SXA(ptr16 オプション有り) 、H8SXX(ptr16 オプション有り)、
2600n、2000n、300hn、300 指定時：
514 バイト×（同時にオープンするファイルの数の最大値）
cpu＝H8SXA(ptr16 オプション無し)、H8SXX(ptr16 オプション無し)、
2600a、2000a、300ha 指定時：
516 バイト×（同時にオープンするファイルの数の最大値）
になります。

注意　メモリ管理ライブラリ関数の free 関数、delete 演算子(C++)で解放した領域は、再びメモリ管理ライブラリ関数で領域を確保するときに再利用しますが、割り付けを繰り返すことによって空き領域のサイズの合計は十分でも、空き領域が小さな領域に分割しているために、新たに要求した大きなサイズの領域を確保できないという状況が生じることがあります。このような状況を避けるために、サイズの大きな領域は、なるべくプログラムの実行開始直後に確保してください。また、解放して再利用するデータ領域のサイズをなるべく一定にしてください。

(d) 動的領域の割り付け方

動的領域は RAM 上に割り付けます。

スタック領域については、プログラム起動時のリセットルーチンでスタックセクションの最上位アドレスを SP（スタックポインタ）に設定することにより割り付ける場所が決まります。C/C++コンパイラの_ _entry（または#pragma entry）、#pragma stacksize を用いることにより、スタック領域(S セクション）の生成、リセットプログラムでの SP の初期設定コードの出力をコンパイラが自動的に行います。

ヒープ領域については、低水準インタフェースルーチン（sbrk）の初期設定で割り付ける場所が決まります。**「**9.2.2 (2) **初期設定（**PowerON_Reset**）」**、**「**9.2.2 (7) **低水準インタフェースルーチン」**を参照してください。

9.2.2 実行環境の設定

本節では、プログラムの実行に必要な環境を設定するための処理について説明します。ただし、プログラムを実行する環境はユーザシステムごとに異なりますので、ユーザシステムの仕様に合わせて実行環境の設定プログラムを作成する必要があります。

図 9.6 にプログラムの構成例を示します。

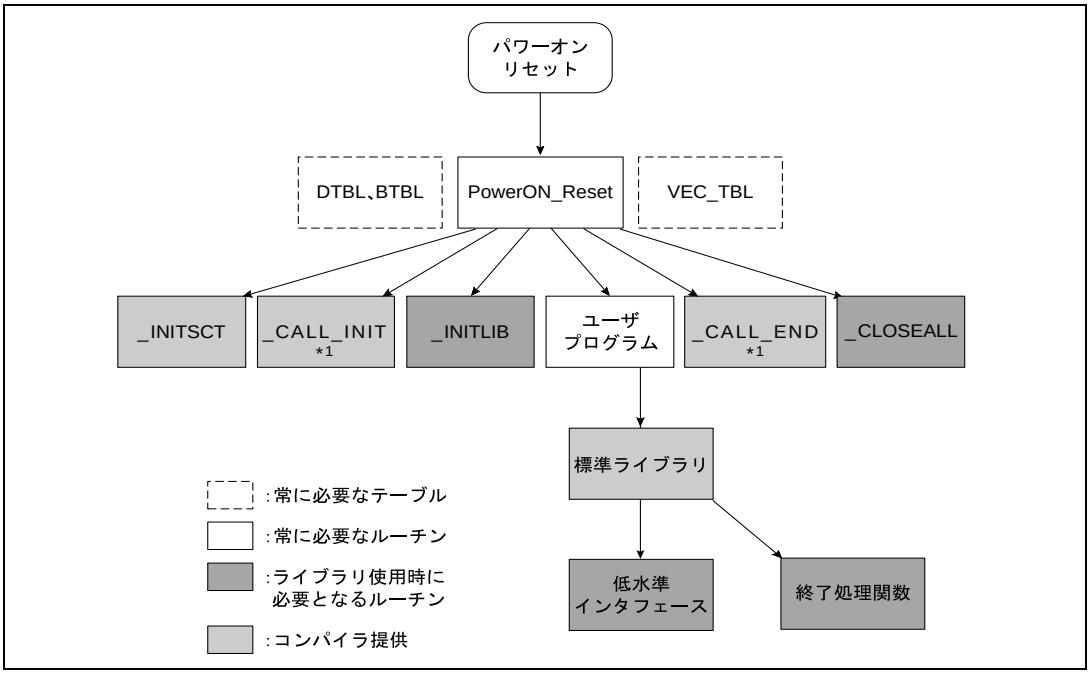


図 9.6 **プログラムの構成例**

【注】*1 C++プログラム中にグローバルクラスオブジェクトの宣言があるとき必要になります。

各構成ルーチンの内容は以下のとおりです。

・ベクタテーブル（VEC_TBL）
パワーオンリセットでレジスタの初期設定プログラム（PowerON_Reset）が起動されるように、ベクタテーブルを設定します。

・ 初期設定（PowerON_Reset）
レジスタの初期設定を行ったあと、初期設定ルーチンを順次呼び出します。

・ セクション初期化用テーブル（DTBL、BTBL）
セクションの初期化ルーチンで使用するセクションの先頭アドレスおよび最終アドレスを、セクションアドレス演算子を用いて設定します。

・セクションの初期化（_INITSCT）*1
初期値が設定されていない静的変数領域（未初期化データ領域）をゼロで初期化します。また、初期化データ領域の初期値をROM上からRAM上にコピーします。

・グローバルクラスオブジェクト初期処理 (_CALL_INIT) *1*2
グローバルに宣言されたクラスオブジェクトに対してコンストラクタを呼び出します。

・グローバルクラスオブジェクト後処理 (_CALL_END) *1*2
main関数の実行後、グローバルクラスオブジェクトに対してデストラクタを呼び出します。

・ C/C++ライブラリ関数の初期設定（_INITLIB）
C/C++ライブラリ関数をご使用になる場合、初期設定の必要なものについて、初期設定を行います。
・ ファイルのクローズ（_CLOSEALL）
オープンしているファイルを全てクローズします。
・低水準インタフェースルーチン
標準入出力（stdio.h、ios、streambuf、istream、ostream）、メモリ管理ライブラリ（stdlib.h、new）を使用する場合に必要なライブラリ関数とユーザシステムとの間のインタフェースルーチンです。
・ 終了処理ルーチン（exit, atexit, abort）*3
プログラムの終了処理を行います。

【注】*1 標準ライブラリとして提供しています。_INITSCT,_CALL_INIT,_CALL_END を使用する場合は<_h_c_lib.h>をインクルードして下さい。
*2 C++プログラム中にグローバルクラスオブジェクトの宣言があるときに必要な処理です。
*3 プログラムの終了処理を行う C ライブラリ関数 exit、atexit、abort 関数を使用する場合は、ユーザシステムに合わせてこれらの関数を作成する必要があります。C++プログラムを使用する場合、または C ライブラリ関数 assert マクロを使用する場合、abort 関数は必ず作成する必要があります。

以下、この構成に従って各処理の実現方法について解説します。

(1) ベクタテーブルの設定（VEC_TBL）

パワーオンリセットで、初期設定関数「PowerON_Reset」が呼び出されるようにするために、ベクタテーブルの 0 番地に関数「PowerON_Reset」のアドレスを設定する必要があります。

また、ユーザシステムで割り込み処理やメモリ間接関数呼び出しを使用する場合は、割り込みベクタやアドレステーブルを設定する必要があります。

ベクタテーブルは、C/C++コンパイラ拡張機能の_ _entry（または#pragma entry）、_ _interrupt（または#pragma interrupt）や_ _indirect（または#pragma indirect）で vect パラメタを指定することにより、コンパイラが自動生成します。以下にコーディング例を示します。

例:

```
_ _entry(vect=0) void PowerON_Reset(void)  // PowerON_Reset 関数アドレスを 0 番地に設定
{
                :
}
_ _interrupt(vect=3) void INT_NMI(void)    // INT_NMI 関数アドレスをベクタ番号 3 に設定
{
                :
}
_ _indirect(vect=4) char f(void)          // f 関数アドレスをベクタ番号 4 に設定
{
                :
}
```

(2) 初期設定（PowerON_Reset）

初期設定関数では、スタックポインタ（SP）やコンディションコードレジスタ（CCR）などのレジスタの初期設定を行い、セクションの初期化ルーチン（_INITSCT）を呼び出したあと、main 関数を呼び出します。C++プログラムでグローバルクラスオブジェクトが存在するときは、main 関数呼び出し前後に初期／終了処理関数を順次呼び出す_CALL_INIT、_CALL_END 関数を呼び出します。

SP の設定は、__entry（または#pragma entry）を用いることによりコンパイラが自動生成します。またコンディションコードレジスタの設定は、組み込み関数（set_imask_ccr 等）を用いて記述します。

_INITSCT および_CALL_INIT、_CALL_END 関数は標準ライブラリ関数として提供しています。これらの関数を使用する場合は、<_h_c_lib.h>をインクルードしてください。

C/C++ライブラリ関数を使用する場合には、ここでライブラリの初期設定を行う「_INITLIB」とファイルのクローズ処理を行う「_CLOSEALL」を呼び出します。

以下にコーディング例を示します。

例:

```
#include <machine.h>          // <machine.h>をインクルードします
#include <_h_c_lib.h>         // <_h_c_lib.h>をインクルードします
#pragma stacksize 0x200       // S セクション（スタック）サイズを指定します
extern void PowerON_Reset(void);
extern void main(void);

#ifdef __cplusplus
extern "C" {
#endif
extern void _INITLIB(void);
extern void _CLOSEALL(void);
#ifdef __cplusplus
}
#endif

__entry(vect=0) void PowerON_Reset(void)
{                             // SP に S セクションの最上位アドレスを設定します
      _ _asm{                 // H8SX で必要な場合は SBR/VBR の初期設定をしてください
          MOV.L #0xFFFFFF00,ER0
          LDC.L ER0,SBR      // H8SX で必要な場合は SBR を初期化します。
          MOV.L #0x00000000,ER0
          LDC.L ER0,VBR      // H8SX で必要な場合は VBR を初期化します。
        }                     // H8SX の場合
        set_imask_ccr(1);      // 割り込みをマスクします
          _INITSCT();         // セクションの初期化ルーチンを呼び出します

#ifdef __cplusplus
          _CALL_INIT();    // C++のグローバルクラスオブジェクトが存在するときに呼び出します
#endif

          _INITLIB();          // ライブラリの初期設定関数を呼び出します
          set_imask_ccr(0);    // 割り込みマスクを解除します
          main();
          _CLOSEALL();         // ファイルのクローズ処理関数を呼び出します

#ifdef __cplusplus
          _CALL_END();     // C++のグローバルクラスオブジェクトが存在するときに呼び出します
#endif

          sleep();
}
```

(3) セクション初期化用テーブル（DTBL、BTBL）

セクションの初期化ルーチン（_INITSCT)では、未初期化データセクションをゼロで初期化し、初期化データセクションの ROM 上にある初期化データを RAM 上にコピーします。ここでは、_INITSCT 関数が使用するセクションの先頭アドレスおよび最終アドレスを、セクションアドレス演算子を用いて、セクションの初期化用テーブルに設定します。

セクション初期化用テーブルのセクション名は、未初期化データ領域を C$BSEC、初期化データ領域を C$DSEC で宣言します。

以下にコーディング例を示します。

例:

```
#pragma section $DSEC         // セクション名をC$DSECにします
static const struct DSEC{
    void * rom_s;             // 初期化データセクションのROM上の先頭アドレスメンバ
    void * rom_e;             // 初期化データセクションのROM上の最終アドレスメンバ
    void * ram_s;             // 初期化データセクションのRAM上の先頭アドレスメンバ
}DTBL[]={
    {_ _sectop (“D”), _ _secend (“D”), _ _sectop (“R”)},
    {_ _sectop (“$ABS8D”), _ _secend (“$ABS8D”), _ _sectop (“$ABS8R”)},
    {_ _sectop (“$ABS16D”), _ _secend (“$ABS16D”), _ _sectop (“$ABS16R”)}
};

#pragma section $BSEC         // セクション名をC$BSECにします
static const struct BSEC{
    void * b_s;               // 未初期化データセクションの先頭アドレスメンバ
    void * b_e;               // 未初期化データセクションの最終アドレスメンバ
}BTBL[]={
    {_ _sectop (“B”), _ _secend (“B”)},
    {_ _sectop (“$ABS8B”), _ _secend (“$ABS8B”)},
    {_ _sectop (“$ABS16B”), _ _secend (“$ABS16B”)}
};
```

【注】 上記プログラムは、必ず C 言語としてコンパイル（ファイル拡張子を c **とするか、または、**lang=c **オプションを指定）してください。**C++**言語としてコンパイル（ファイル拡張子を** cpp, cc **または** cp **とするか、または、**lang=cpp **オプションを指定）すると、セクション初期化用テーブルは、コンパイラによって未参照** static **データとして削除されるため、動作が不正になります。**

セクションの初期化ルーチン（_INITSCT）は標準ライブラリとして提供していますが、以下の例に示すプログラムと同様の動作をします。

例:

```
static const struct DSEC{     // 前例で定義したDセクション初期化テーブル構造体
    void * rom_s;             // 初期化データセクションのROM上の先頭アドレスメンバ
    void * rom_e;             // 初期化データセクションのROM上の最終アドレスメンバ
    void * ram_s;             // 初期化データセクションのRAM上の先頭アドレスメンバ
};

static const struct BSEC{     // 前例で定義したBセクション初期化テーブル構造体
    void * b_s;               // 未初期化データセクションの先頭アドレスメンバ
    void * b_e;               // 未初期化データセクションの最終アドレスメンバ
};

static void clearblock(void *b_top, void *b_end);
static void copyblock (void *d_top, void *d_end, void *r_top);

#ifdef _ _cplusplus
extern “c”                    // Cリンケージします
#endif
void _INITSCT(void)           // セクション初期化ルーチン
{
    const struct BSEC *btbl;  // Bセクション初期化テーブル構造体
    const struct DSEC *dtbl;  // Dセクション初期化テーブル構造体

                              // 未初期化データセクションを初期化します
    for( btbl = _ _sectop (“C$BSEC”);
            btbl < (struct BSEC *)_ _secend (“C$BSEC”); btbl++)
        clearblock( btbl->b_s, btbl->b_e );

                              // 初期化データをROMからRAMコピーします
    for( dtbl = _ _sectop (“C$DSEC”);
            dtbl < (struct DSEC *)_ _secend (“C$DSEC”); dtbl++)
        copyblock( dtbl->rom_s, dtbl->rom_e, dtbl->ram_s );
}

static void clearblock(void *b_top, void *b_end)
{                             // 未初期化データセクションをゼロで初期化します
    char *p;
    for( p=b_top; p<(char *)b_end; p++)
        *p = 0;
}
static void copyblock(void *d_top, void *d_end, void *r_top)
{                             // 初期化データをROMからRAMコピーします
    char *p, *q;
    for( p=r_top, q=d_top; q<(char *)d_end; p++, q++)
        *p = *q;
}
```

(4) C++グローバルクラスオブジェクトの初期設定（_CALL_INIT）

_CALL_INIT 関数は C++でグローバルに宣言されたクラスオブジェクトのコンストラクタを呼びます。本_CALL_INIT 関数はライブラリヘッダファイル<_h_c_lib.h>で提供されていますが、動作を説明するために例を示します。

例：

```
extern “C”  void _CALL_INIT(void);

typedef void (**FPP)(void);   // 関数ポインタ型

extern “C”  void _CALL_INIT(void)
{                             // グローバルクラスオブジェクト初期化ルーチン
    FPP top = (FPP)_ _sectop("C$INIT");
    FPP end = (FPP)_ _secend("C$INIT");

    while (top < end)
        (*top++)();           // コンストラクタ呼び出し
}
```

(5) C/C++ライブラリ関数の初期設定（_INITLIB）

ここでは、C/C++ライブラリ関数の初期設定方法を説明します。

実際に使用する機能に合わせた必要最低限の初期設定を行うために、以下の指針を参考にしてください。

- <stdio.h>、<ios>、<streambuf>、<istream>、<ostream>の各関数と assert マクロを使用する場合、標準入出力の初期設定（_INIT_IOLIB）が必要です。
- 作成した低水準インタフェースルーチンの中で初期設定が必要な場合、低水準インタフェースルーチンの仕様に合わせた初期設定（_INIT_LOWLEVEL）が必要です。
- rand 関数、strtok 関数を使用する場合、標準入出力以外の初期設定（_INIT_OTHERLIB）が必要です。

ライブラリの初期設定を行うプログラム例を以下に示します。また、**図** 9.7 に FILE 型データを示します。

```
#include <stdio.h>
#include <stdlib.h>
#define IOSTREAM 3
const size_t _sbrk_size = 520;   // ヒープ領域確保サイズの最小単位を指定します。但し、本行
// 省略時はadvanced(ptr16オプション無し),maximum(ptr16オプション無し)の時1032、
// normal,middle,advanced(ptr16オプション有り),maximum(ptr16オプション有り),300の時1028
const int _nfiles = IOSTREAM;    // 入出力ファイル数を指定します(省略時:20)
struct _iobuf _iob[IOSTREAM];
unsigned char sml_buf[IOSTREAM];
extern char *_s1ptr;

#ifdef __cplusplus
extern "C" {
#endif
void _INITLIB (void)
{
   _INIT_LOWLEVEL();             // 低水準インタフェースルーチンの初期設定をします
   _INIT_IOLIB();                // 入出力ライブラリの初期設定をします
   _INIT_OTHERLIB();             // rand関数、strtok関数の初期設定をします
}
void _INIT_LOWLEVEL (void)
{                                // 低水準ライブラリで必要な初期設定をしてください
}
void _INIT_IOLIB(void)
{
FILE *fp;
   for( fp = _iob; fp < _iob + _nfiles; fp++ ) // FILE型データの初期設定です
   {
      fp->_bufptr = NULL;
      fp->_bufcnt = 0;
      fp->_buflen = 0;
      fp->_bufbase = NULL;
      fp->_ioflag1 = 0;
      fp->_ioflag2 = 0;
      fp->_iofd = 0;
   }
      if(freopen("stdin*1","r",stdin)== NULL) // 標準入力ファイルをオープンします
         stdin->_ioflag1 = 0xff;             // オープン失敗時のファイルアクセスを禁止
      stdin->_ioflag1 |= _IOUNBUF;           // データのバッファリングなしに設定します*2
      if(freopen("stdout*1","w",stdout)== NULL)// 標準出力ファイルをオープンします
         stdout->_ioflag1 = 0xff;            // オープン失敗時のファイルアクセスを禁止
      stdout->_ioflag1 |= _IOUNBUF;          // データのバッファリングなしに設定します*2
      if(freopen("stderr*1","w",stderr)== NULL)// 標準エラーファイルをオープンします
         stderr->_ioflag1 = 0xff;            // オープン失敗時のファイルアクセスを禁止
      stderr->_ioflag1 |= _IOUNBUF;          // データのバッファリングなしに設定します*2
}
void _INIT_OTHERLIB(void)
{
   srand(1);                                 // rand関数を使用する場合の初期設定です
   _s1ptr=NULL;                              // strtok関数を使用する場合の初期設定です
}
#ifdef __cplusplus
}
#endif
```

【注】*1　標準入出力ファイルのファイル名を指定します。この名前は、低水準インタフェースルーチン「open」で使用します。
*2　コンソール等対話的な装置の場合、バッファリングを行わないためのフラグを立てます。

```
                    /* ファイル型データのC言語での宣言 */


        struct _iobuf{
                    unsigned char *_bufptr;  /* バッファへのポインタ    */
                    long          _bufcnt;   /* バッファカウンタ        */
                    unsigned char *_bufbase; /* バッファベースポインタ  */
                    long           _buflen;  /* バッファ長              */
                    char           _ioflag1; /*   i/oフラグ             */
                    char           _ioflag2; /*   i/oフラグ             */
                    char           _iofd;    /*   i/oフラグ             */
        }iob[_nfiles];
```

図 9.7　FILE **型データ**

(6)　**ファイルのクローズ（**_CLOSEALL**）**

通常ファイルへの出力は、メモリ領域上のバッファにためておき、バッファが一杯になったときに実際の外部記憶装置への書き出しを行います。したがってファイルのクローズを行わないと、ファイルへの出力内容が外部記憶装置へ書き出されないことがあります。

機器組み込み用のプログラムの場合、通常プログラムが終了することはありません。しかし、プログラムの誤りなどにより main 関数が終了する場合、オープンしているファイルは、すべてクローズしなければなりません。

本処理は、main 関数終了時にオープンしているファイルのクローズを行います。

ファイルのクローズを行うプログラム例を以下に示します。

```
#include <stdio.h>

#ifdef __cplusplus
extern "C"
#endif
void _CLOSEALL(void)
{
int i;

    for( i=0; i < _nfiles; i++ )

    // ファイルがオープンしているかどうかチェックします
        if( _iob[i]._ioflag1 & (_IOREAD | _IOWRITE | _IORW ) )
            fclose( &_iob[i] );  // ファイルをクローズします
}
```

(7) 低水準インタフェースルーチン

標準入出力、メモリ管理ライブラリを C/C++プログラムで使用する場合は、低水準インタフェースルーチンを作成しなければなりません。**表** 9.3 に C ライブラリ関数で使用している低水準インタフェースルーチンの一覧を示します。

表 9.3 **低水準インタフェースルーチンの一覧**

	名称	機能
1	open	**ファイルのオープン**
2	close	**ファイルのクローズ**
3	read	**ファイルからの読み込み**
4	write	**ファイルへの書き出し**
5	lseek	**ファイルの読み込み／書き出しの位置の設定**
6	sbrk	**メモリ領域の確保**
7	error_addr※	errno **アドレスの取得**
8	wait_sem※	**セマフォの確保**
9	signal_sem※	**セマフォの解放**

【注】※リエントラントライブラリを使用する場合に必要です。

低水準インタフェースルーチンで必要な初期化は、プログラム起動時に行う必要があります。これは、**「**9.2.2(5) C/C++**ライブラリ関数の初期設定（**_INITLIB**）」**の中の「_INIT_LOWLEVEL」という関数の中で行ってください。

以下、低水準入出力の基本的な考え方を説明したあと、各インタフェースルーチンの仕様を説明します。

注意　関数名 open、close、read、write、lseek、sbrk は低水準インタフェースルーチンの予約語です。ユーザのプログラム中では使用しないでください。

(a) 入出力の考え方

標準入出力ライブラリでは、ファイルを FILE 型のデータによって管理しますが、低水準インタフェースルーチンでは、実際のファイルと 1 対 1 に対応する正の整数を与え、これによって管理します。この整数をファイル番号といいます。

open ルーチンでは、与えられたファイル名に対してファイル番号を与えます。open ルーチンでは、この番号によってファイルの入出力ができるように、以下の情報を設定する必要があります。

・ファイルのデバイスの種類（コンソール、プリンタ、ディスクファイル等）。
（コンソールやプリンタ等の特殊なデバイスに対しては、特別なファイル名をシステムで決めておいて open ルーチンで判定する必要があります。）
・ファイルのバッファリングをする場合はバッファの位置、サイズ等の情報。
・ディスクファイルならば、ファイルの先頭から次に読み込み／書き出しを行う位置までのバイトオフセット。

open ルーチンで設定した情報に基づいて、以後、入出力（read、write ルーチン）、読み込み／書き出し位置の設定（lseek ルーチン）を行います。

close ルーチンでは、出力ファイルのバッファリングを行っている場合はバッファの内容を実際のファイルに書き出し、open ルーチンで設定したデータの領域が再使用できるようにしてください。

(b) 低水準インタフェースルーチンの仕様

本項では低水準インタフェースルーチンを作成するための仕様を説明します。以下、各ルーチンごとに、ルーチンを呼び出す際のインタフェースとその動作および実現上の注意事項を示します。

各ルーチンのインタフェースは以下の形式で示します。なお、低水準インタフェースルーチンは必ず原型宣言してください。また C++プログラム内で宣言する場合は「extern "C"」を付加してください。

凡例：

簡易説明

(ルーチン名)

説　明　（ルーチンの機能概要を示します。）

リターン値	正常：	（正常に終了した場合のリターン値の意味を示します。）
	異常：	（エラーが生じた場合のリターン値を示します。）
引　数	（名前）	（意味）
	（インタフェースに示す引数名です。）	（引数として渡される値の意味を示します。）

ファイルのオープン

*int open(char *name, int mode, int flg)*

説　明　第 1 引数のファイル名に対応するファイルを操作するための準備をします。
open ルーチンでは、後で読み込み／書き出しを行うために、ファイルの種類（コンソール、プリンタ、ディスクファイル等）を決定しなければなりません。ファイルの種類は、以後 open ルーチンで返したファイル番号を用いて読み込み／書き出しを行うたびにアクセスする必要があります。
第 2 引数の mode は、ファイルをオープンする時の処理の指定です。このデータの各ビットの意味について以下に示します。

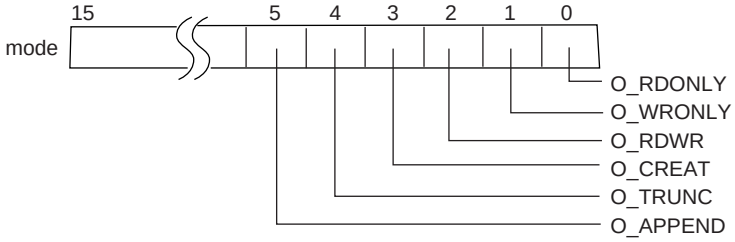


表 9.4　open **ルーチン** mode **ビット説明**

mode **ビット**	**説明**
O_RDONLY（0 **ビット**）	**ビットが** 1 **のとき、ファイルを読み込み専用にオープン**
O_WRONLY（1 **ビット**）	**ビットが** 1 **のとき、ファイルを書き出し専用にオープン**
O_RDWR（2 **ビット**）	**ビットが** 1 **のとき、ファイルを読み込み、書き出し両用にオープン**
O_CREAT（3 **ビット**）	**ビットが** 1 **のとき、ファイル名で示すファイルが存在しない場合にファイルを新規に作成**
O_TRUNC（4 **ビット**）	**ビットが** 1 **のとき、ファイル名で示すファイルが存在する場合にファイルの内容を捨て、ファイルのサイズを** 0 **に更新**
O_APPEND（5 **ビット**）	**次に読み込み／書き出しを行うファイル内の位置を設定** **ビットが** 0 **のとき、：ファイルの先頭に設定** **ビットが** 1 **のとき、：ファイルの最後に設定**

mode で示したファイルの処理の指定と実際のファイルの性質が矛盾する場合はエラーにしてください。正常にファイルがオープンできた場合は、以後の read、write、lseek、close ルーチンで使用されるファイル番号（正の整数）を返してください。ファイル番号と実際のファイルの対応は低水準インタフェースルーチンで管理する必要があります。オープンに失敗した場合は−1 を返してください。

リターン値　正常：　正常オープンしたファイルのファイル番号
　　異常：　-1

引　数　name　ファイルのファイル名を指す文字
　　mode　ファイルをオープンするときの処理の指定
　　flg　ファイルをオープンするときの処理の指定(常に 0777)

ファイルのクローズ

int close(int fileno)

説　明　open ルーチンで得られたファイル番号が引数として渡されます。
open ルーチンで設定したファイル管理情報の領域を、再び使用できるように解放してください。また、低水準インタフェースルーチン内で出力ファイルのバッファリングを行っている場合は、バッファの内容を実際のファイルに書き出してください。
ファイルを正常にクローズできた場合は 0、失敗した場合は－1 を返してください。

リターン値　正常：　0
異常：　-1

引　数　fileno　クローズするファイル番号

データの読み込み

*int read(int fileno, char *buf, unsigned int count)*

説　明　第 1 引数（fileno）で示すファイルから、第 2 引数（buf）の指す領域へデータを読み込みます。読み込むデータのバイト数は第 3 引数（count）で示します。
ファイルが終了した場合、count で示されたバイト数以下のバイト数しか読み込むことができません。
ファイルの読み込み／書き出しの位置は、読み込んだバイト数だけ先に進みます。
正常に読み込みができた場合は、実際に読み込んだバイト数を返してください。読み込みに失敗した場合は－1 を返してください。

リターン値　正常：　実際に読み込んだバイト数
異常：　-1

引　数　fileno　読み込みの対象となるファイル番号
buf　読み込んだデータを格納する領域
count　読み込むバイト数

データの書き出し

*int write(int fileno, char *buf, unsigned int count)*

説　明　第２引数（buf）の指す領域から、第１引数（fileno）の示すファイルにデータを書き出します。書き込むデータのバイト数は第３引数（count）で示します。
ファイルを書き出そうとしているデバイス（ディスク等）に空きがない時は、count で示されたバイト数以下のバイト数しか書き出すことができません。実際に書き出すことのできたバイト数が何度か連続して０バイトの場合、ディスクが満杯であると判断してエラー（－1）を返すように実現することをお勧めします。
ファイルの読み込み／書き出しの位置は、書き出したバイト数だけ先に進みます。
正常に書き出しができた場合は、実際に書き出したバイト数を返してください。書き出しに失敗した場合は－1 を返してください。

リターン値　正常：　実際に書き出されたバイト数
異常：　-1

引　数

fileno	書き出しの対象となるファイル番号
buf	書き出すデータ領域
count	書き出すバイト数

ファイル内位置の設定

int lseek(int fileno, long offset, int base)

説　明　ファイルの読み込み／書き出しを行うファイル内の位置を、バイト単位で設定します。
新しいファイル内の位置は、第３引数（base）によって、以下の方法で計算し設定してください。

[1] base が０のとき	ファイルの先頭から offset バイトの位置に設定します。
[2] base が１のとき	現在の位置に offset バイトを加えた位置に設定します。
[3] base が２のとき	ファイルのサイズに offset バイトを加えた位置に設定します。

ファイルがコンソールやプリンタ等の対話的なデバイスの場合や、新しいオフセットの値が負になったり、[1]、[2]のときファイルのサイズをこえる場合はエラーにします。
正しくファイル位置を設定できた場合は、新しい読み込み／書き出し位置のファイルの先頭からのオフセットを、そうでない場合は－1 を返してください。

リターン値　正常：　新しいファイルの読み込み／書き出し位置のファイルの先頭からのオフセット（バイト単位）
異常：　-1

引　数

fileno	対象となるファイル番号
offset	読み込み／書き出しの位置を示すオフセット（バイト単位）
base	オフセットの起点

メモリ領域の割り付け

*char *sbrk(size_t size)*

説　明　メモリ領域を割り付けるサイズが引数として渡されます。
連続して sbrk ルーチンを呼び出す場合は、下位アドレスから順に連続した領域が割り付けられるようにしてください。割り付けるメモリ領域が不足した場合はエラーにしてください。
正常に割り付けができた場合は、割り付けた領域の先頭のアドレスを、割り付けに失敗した場合は、「（char *）－1」を返してください。

リターン値　正常：　割り付けた領域の先頭アドレス
異常：　(char *) -1

引　数　size　割り付けるデータのサイズ

errno アドレス取得

*int *errno_addr(void)*

説　明　現在のタスクが持つエラー番号のアドレスを返却します。
標準ライブラリ構築ツールで reent オプションを指定して作成した標準ライブラリを使用する場合に、本関数は必要になります。

リターン値　現在のタスクが持つエラー番号のアドレス

セマフォ確保

int wait_sem (int semnum)

説　明　semnum で示されたセマフォを確保します。
確保できた場合は 1、確保できなかった場合は 0 を返してください。
標準ライブラリ構築ツールで reent オプションを指定して作成した標準ライブラリを使用する場合に、本関数は必要になります。

リターン値　正常：　1
異常：　0

引　数　semnum　セマフォ ID

セマフォ解放

int signal_sem (int semnum)

説　明　semnum で示されたセマフォを解放します。
解放できた場合は 1、解放できなかった場合は 0 を返してください。
標準ライブラリ構築ツールで reent オプションを指定して作成した標準ライブラリを使用する場合に、本関数は必要になります。

リターン値　正常：　1
異常：　0

引　数　semnum　セマフォ ID

(c) 低水準インタフェースルーチンの作成例

```
/*********************************************************************/
/*                              lowsrc.c:                            */
/*―――――――――――――――――――――――――――――――――――――――――――――――――*/
/* H8S, H8/300 シリーズ  シミュレータ・デバッガ  インタフェースルーチン          */
/*       ―標準入出力(stdin,stdout,stderr)だけをサポートしています―          */
/*********************************************************************/
#include <string.h>

/* ファイル番号 */

#define STDIN  0                        /* 標準入力       (コンソール)    */
#define STDOUT 1                        /* 標準出力       (コンソール)    */
#define STDERR 2                        /* 標準エラー出力 (コンソール)    */

#define FLMIN  0                        /* 最小のファイル番号             */
#define FLMAX  3                        /* ファイル数の最大値             */

/* ファイルのフラグ */

#define O_RDONLY 0x0001                 /* 読み込み専用                   */
#define O_WRONLY 0x0002                 /* 書き出し専用                   */
#define O_RDWR   0x0004                 /* 読み書き両用                   */

/* 特殊文字コード */

#define CR 0x0d                         /* 復帰                           */
#define LF 0x0a                         /* 改行                           */

/* sbrk で管理する領域サイズ */

#if _ _DATA_ADDRESS_SIZE_ _ = = 4
#define HEAPSIZE 2064
#else
#define HEAPSIZE 2056
#endif

/*********************************************************************/
/* 参照関数の宣言：                                                  */
/* シミュレータ・デバッガでコンソールへの文字入出力を行うアセンブリプログラムの参照  */
/*********************************************************************/

extern  void charput(char);   /* 一文字出力処理                       */
extern  char charget(void);   /* 一文字入力処理                       */
```

```
/*****************************************************************************/
/* 静的変数の定義：                                                          */
/* 低水準インタフェースルーチンで使用する静的変数の定義                      */
/*****************************************************************************/

char flmod[FLMAX];                          /* オープンしたファイルのモード設定場所 */

static  union {
            short  dummy ;                  /* 2 バイト境界にするためのダミー       */
            char heap[HEAPSIZE];            /* sbrk で管理する領域の宣言            */
 }heap_area ;

static  char  *brk=(char *)&heap_area;      /* sbrk で割り付けた領域の最終アドレス */

/*****************************************************************************/
/*  open:ファイルのオープン                                                  */
/*         リターン値：ファイル番号（成功）                                  */
/*                     -1          （失敗）                                  */
/*****************************************************************************/
extern  open(char *name,                    /* ファイル名                     */
     int  mode,                             /* ファイルのモード                 */
     int  flg)                              /* 未使用                         */
{
       /* ファイル名に従ってモードをチェックし、ファイル番号を返す             */
       if(strcmp(name,"stdin")==0){         /* 標準入力ファイル      */
             if((mode&O_RDONLY)==0)
                      return -1;
             flmod[STDIN]=mode;
             return STDIN;
       }

       else if(strcmp(name,"stdout")==0){   /* 標準出力ファイル      */
             if((mode&O_WRONLY)==0)
                      return -1;
             flmod[STDOUT]=mode;
             return STDOUT;
       }

       else if(strcmp(name,"stderr")==0){   /* 標準エラー出力ファイル */
             if((mode&O_WRONLY)==0)
                      return -1;
             flmod[STDERR]=mode;
             return STDERR;
       }

       else
             return -1;                     /* エラー                         */
}
```

```
/*****************************************************************************/
/*  close:ファイルのクローズ                                                 */
/*        リターン値：0          （成功）                                     */
/*                   -1          （失敗）                                     */
/*****************************************************************************/
extern  close(int fileno)                       /* ファイル番号              */
{
        if(fileno<FLMIN || FLMAX<=fileno)       /* ファイル番号の範囲チェック  */
              return -1;

        flmod[fileno]=0;                        /* ファイルのモードリセット    */
        return 0;
}

/*****************************************************************************/
/*  read:データの読み込み                                                    */
/*         リターン値：実際に読み込んだ文字数          （成功）              */
/*                            -1                （失敗）                    */
/*****************************************************************************/
extern  read(int  fileno,                       /* ファイル番号              */
     char *buf,                                 /* 転送先バッファアドレス      */
     int  count)                                /* 読み込み文字数            */
{
        int i;

        /* ファイル番号に従ってモードをチェックし、一文字づつ入力してバッファに格納    */

        if(flmod[fileno]&O_RDONLY || flmod[fileno]&O_RDWR){
              for(i=count; i>0; i--){
                    *buf=charget();
                    if(*buf==CR)                /* 改行文字の置き換え        */
                          *buf=LF;
                    buf++;
              }
              return count;
        }
        else
              return -1;
}
```

```
/*************************************************************************/
/*  write:データの書き出し                                               */
/*          リターン値：実際に書き出した文字数        （成功）           */
/*                          -1                        （失敗）           */
/*************************************************************************/
extern  write(int  fileno,                      /* ファイル番号          */
      char *buf,                                /* 転送元バッファアドレス */
      int  count)                               /* 書き出し文字数        */
{
       int  i;
       char c;

       /* ファイル番号に従ってモードをチェックし、一文字づつ出力          */

       if(flmod[fileno]&O_WRONLY || flmod[fileno]&O_RDWR){
             for(i=count; i>0; i--){
                   c=*buf++;
                   charput(c);
             }
             return count;
       }
       else
             return -1;
}
/*************************************************************************/
/*  lseek:ファイルの読み込み／書き出し位置の設定                         */
/*          リターン値：読み込み／書き出し位置のファイル先頭からのオフセット（成功）*/
/*                      -1                                  （失敗）     */
/*      (コンソール入出力では、lseek はサポートしていません)             */
/*************************************************************************/
extern  long lseek(int  fileno,                 /* ファイル番号          */
           long offset,                         /* 読み込み／書き出し位置 */
           int  base)                           /* オフセットの起点      */
{
       return -1L;
}
/*************************************************************************/
/*  sbrk:データの書き出し                                                */
/*       リターン値：割り付けた領域の先頭アドレス（成功）                */
/*                   -1                         （失敗）                 */
/*************************************************************************/
extern  char  *sbrk(int size)                   /* 割り付ける領域のサイズ */
{
      char  *p ;
      if (brk+size>heap_area.heap+HEAPSIZE)     /* 空き領域のチェック    */
            return (char *)-1 ;
      p=brk ;                                   /* 領域の割り付け        */
      brk += size ;                             /* 最終アドレスの更新    */
      return p ;
}
```

```
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;                                       lowlvl.nor                                   |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;  H8S, H8/300 シリーズ　シミュレータ・デバッガ　インタフェースルーチン                  |
;                          －一文字入出力を行います－                                 |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
; H8SX,H8S/2600,H8S/2000,H8/300H ノーマルモード                                       |
; (cpu=H8SXN, 2600n, 2000n, 300hn)                                                    |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
            .CPU                2600N               ; または H8SXN,2000N,300HN
            .EXPORT             _charput
            .EXPORT             _charget
SIM_IO:     .EQU                H'00FE              ; TRAP_ADDRESS の指定

            .SECTION            P,CODE,ALIGN=2
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;  _charput: 一文字出力                                                               |
;            C プログラムインタフェース：  charput(char)                              |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

_charput:
            MOV.B               R0L,@IO_BUF         ; パラメタをバッファに設定
            MOV.W               #H'0102,R0          ; パラメタ、機能コードの設定
            MOV.W               #LWORD IO_BUF,R1
            MOV.W               R1,@PARM            ; 入出力バッファアドレスの設定
            MOV.W               #LWORD PARM,R1      ; パラメタブロックアドレスの設定
            JSR                 @SIM_IO
            RTS

;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;  _charget: 一文字入力                                                               |
;            C プログラムインタフェース：  char charget(void)                         |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

_charget:
            MOV.W               #H'0101,R0          ; パラメタ、機能コードの設定
            MOV.W               #LWORD IO_BUF,R1
            MOV.W               R1,@PARM            ; 入出力バッファアドレスの設定
            MOV.W               #LWORD PARM,R1      ; パラメタブロックアドレスの設定
            JSR                 @SIM_IO
            MOV.B               @IO_BUF,R0L
            RTS

;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
; 入出力用バッファの定義                                                              |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
            .SECTION            B,DATA,ALIGN=2

PARM:       .RES.W              1                   ; パラメタブロック領域
IO_BUF:     .RES.B              1                   ; 入出力バッファ領域

            .END
```

```
;―――――――――――――――――――――――――――――――――――――――――――――――――――――――――
;                                lowlvl.adv                          |
;―――――――――――――――――――――――――――――――――――――――――――――――――――――――――
;  H8S, H8/300 シリーズ  シミュレータ・デバッガ  インタフェースルーチン          |
;                     ――一文字入出力を行います―                           |
;―――――――――――――――――――――――――――――――――――――――――――――――――――――――――
; H8SX、H8S/2600、H8S/2000、H8/300H アドバンストモード(20、24 ビットアドレス)       |
; (cpu=H8SXA:20|24, 2600a:20|24, 2000a:20|24, 300ha)                 |
;―――――――――――――――――――――――――――――――――――――――――――――――――――――――――
            .CPU              2600A             ; または H8SXA,2000A,300HA
            .EXPORT           _charput
            .EXPORT           _charget
SIM_IO:     .EQU              H'01FE            ; TRAP_ADDRESS の指定

            .SECTION          P,CODE,ALIGN=2
;―――――――――――――――――――――――――――――――――――――――――――――――――――――――――
;  _charput: 一文字出力                                               |
;            C プログラムインタフェース：  charput(char)                  |
;―――――――――――――――――――――――――――――――――――――――――――――――――――――――――

_charput:
            MOV.B             R0L,@IO_BUF       ; パラメタをバッファに設定
            MOV.W             #H'0112,R0        ; パラメタ、機能コードの設定
            MOV.L             #IO_BUF,ER1
            MOV.L             ER1,@PARM         ; 入出力バッファアドレスの設定
            MOV.L             #PARM,ER1         ; パラメタブロックアドレスの設定
            JSR               @SIM_IO
            RTS

;―――――――――――――――――――――――――――――――――――――――――――――――――――――――――
;  _charget: 一文字入力                                               |
;            C プログラムインタフェース：  char charget(void)             |
;―――――――――――――――――――――――――――――――――――――――――――――――――――――――――

_charget:
            MOV.W             #H'0111,R0        ; パラメタ、機能コードの設定
            MOV.L             #IO_BUF,ER1
            MOV.L             ER1,@PARM         ; 入出力バッファアドレスの設定
            MOV.L             #PARM,ER1         ; パラメタブロックアドレスの設定
            JSR               @SIM_IO
            MOV.B             @IO_BUF,R0L
            RTS

;―――――――――――――――――――――――――――――――――――――――――――――――――――――――――
; 入出力用バッファの定義                                                |
;―――――――――――――――――――――――――――――――――――――――――――――――――――――――――
            .SECTION          B,DATA,ALIGN=2
PARM:       .RES.L            1                 ; パラメタブロック領域
IO_BUF:     .RES.B            1                 ; 入出力バッファ領域
            .END
```

```
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;                                   lowlvl.mid                                |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;  H8S, H8/300 シリーズ  シミュレータ・デバッガ  インタフェースルーチン         |
;                    －一文字入出力を行います－                               |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
; H8SX ミドルモード、H8SX アドバンスト/マキシマムモード(16 ビットデータアドレス)  |
; (cpu=H8SXM, cpu=H8SXA ptr16, cpu=H8SXX ptr16)                              |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
            .CPU                H8SXM
            .EXPORT             _charput
            .EXPORT             _charget
SIM_IO:     .EQU                H'01FE              ; TRAP_ADDRESS の指定

            .SECTION            P,CODE,ALIGN=2
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;  _charput: 一文字出力                                                        |
;            C プログラムインタフェース：  charput(char)                         |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

_charput:
            MOV.B               R0L,@IO_BUF         ; パラメタをバッファに設定
            MOV.W               #H'0102,R0          ; パラメタ、機能コードの設定
            MOV.W               #LWORD IO_BUF,R1
            MOV.W               R1,@PARM            ; 入出力バッファアドレスの設定
            MOV.W               #LWORD PARM,R1      ; パラメタブロックアドレスの設定
            JSR                 @SIM_IO
            RTS

;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;  _charget: 一文字入力                                                        |
;            C プログラムインタフェース：  char charget(void)                    |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

_charget:
            MOV.W               #H'0101,R0          ; パラメタ、機能コードの設定
            MOV.W               #LWORD IO_BUF,R1
            MOV.W               R1,@PARM            ; 入出力バッファアドレスの設定
            MOV.W               #LWORD PARM,R1      ; パラメタブロックアドレスの設定
            JSR                 @SIM_IO
            MOV.B               @IO_BUF,R0L
            RTS

;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
; 入出力用バッファの定義                                                        |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
            .SECTION            B,DATA,ALIGN=2

PARM:       .RES.W              1                   ; パラメタブロック領域
IO_BUF:     .RES.B              1                   ; 入出力バッファ領域
```

```
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;                                       lowlvl.max                                      |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;  H8S, H8/300 シリーズ  シミュレータ・デバッガ  インタフェースルーチン                   |
;                          ――文字入出力を行います―                                       |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
; H8SX マキシムモード、H8SX、H8S/2600、H8S/2000 アドバンストモード(28、32 ビットアドレス)|
; (cpu=H8SXX, H8SXA:28|32, 2600a:28|32, 2000a:28|32)                                    |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
            .CPU                H8SXX
            .EXPORT             _charput
            .EXPORT             _charget

SIM_IO:     .EQU                H'01FE              ; TRAP_ADDRESS の指定

            .SECTION            P,CODE,ALIGN=2
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;  _charput: 一文字出力                                                                 |
;            C プログラムインタフェース：  charput(char)                                 |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

_charput:
            MOV.B               R0L,@IO_BUF         ; パラメタをバッファに設定
            MOV.W               #H'0122,R0          ; パラメタ、機能コードの設定
            MOV.L               #IO_BUF,ER1
            MOV.L               ER1,@PARM           ; 入出力バッファアドレスの設定
            MOV.L               #PARM,ER1           ; パラメタブロックアドレスの設定
            JSR                 @SIM_IO
            RTS

;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;  _charget: 一文字入力                                                                 |
;            C プログラムインタフェース：  char charget(void)                            |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

_charget:
            MOV.W               #H'0121,R0          ; パラメタ、機能コードの設定
            MOV.L               #IO_BUF,ER1
            MOV.L               ER1,@PARM           ; 入出力バッファアドレスの設定
            MOV.L               #PARM,ER1           ; パラメタブロックアドレスの設定
            JSR                 @SIM_IO
            MOV.B               @IO_BUF,R0L
            RTS

;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
; 入出力用バッファの定義                                                                |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
            .SECTION            B,DATA,ALIGN=2
PARM:       .RES.L              1                   ; パラメタブロック領域
IO_BUF:     .RES.B              1                   ; 入出力バッファ領域
            .END
```

```
;―――――――――――――――――――――――――――――――――――――――――――
;                                   lowlvl.reg                          |
;―――――――――――――――――――――――――――――――――――――――――――
;  H8S, H8/300 シリーズ  シミュレータ・デバッガ  インタフェースルーチン          |
;                          ―一文字入出力を行います―                        |
;―――――――――――――――――――――――――――――――――――――――――――
;                      H8/300 (cpu=300)                                   |
;―――――――――――――――――――――――――――――――――――――――――――
            .CPU            300
            .EXPORT         _charput
            .EXPORT         _charget

SIM_IO:     .EQU            H'00FE          ; TRAP_ADDRESS の指定

            .SECTION        P,CODE,ALIGN=2
;―――――――――――――――――――――――――――――――――――――――――――
;  _charput: 一文字出力                                                    |
;            C プログラムインタフェース：  charput(char)                    |
;―――――――――――――――――――――――――――――――――――――――――――
_charput:
            MOV.B           R0L,@IO_BUF     ; パラメタをバッファに設定
            MOV.W           #H'0102,R0      ; パラメタ、機能コードの設定
            MOV.W           #IO_BUF,R1
            MOV.W           R1,@PARM        ; 入出力バッファアドレスの設定
            MOV.W           #PARM,R1        ; パラメタブロックアドレスの設定
            JSR             @SIM_IO
            RTS

;―――――――――――――――――――――――――――――――――――――――――――
;  _charget: 一文字入力                                                    |
;            C プログラムインタフェース：  char charget(void)               |
;―――――――――――――――――――――――――――――――――――――――――――
_charget:
            MOV.W           #H'0101,R0      ; パラメタ、機能コードの設定
            MOV.W           #IO_BUF,R1
            MOV.W           R1,@PARM        ; 入出力バッファアドレスの設定
            MOV.W           #PARM,R1        ; パラメタブロックアドレスの設定
            JSR             @SIM_IO
            MOV.B           @IO_BUF,R0L
            RTS

;―――――――――――――――――――――――――――――――――――――――――――
; 入出力用バッファの定義                                                   |
;―――――――――――――――――――――――――――――――――――――――――――
            .SECTION        B,DATA,ALIGN=2

PARM:       .RES.W          1               ; パラメタブロック領域
IO_BUF:     .RES.B          1               ; 入出力バッファ領域
            .END
```

(d) リエントラントライブラリ用低水準インタフェースルーチン例

リエントラントライブラリ用低水準インタフェース例を示します。標準ライブラリ構築ツールで reent オプションを指定して作成したライブラリを使用する場合に必要になります。

wait_sem 関数、signal_sem 関数で NG が返った場合、errno に以下を設定し、ライブラリ関数からリターンします。

wait_sem	EMALRESM	malloc 用セマフォ資源確保に失敗しました
	ETOKRESM	strtok 用セマフォ資源確保に失敗しました
	EIOBRESM	_iob 用セマフォ資源確保に失敗しました
signal_sem	EMALFRSM	malloc 用セマフォ資源解放に失敗しました
	ETOKFRSM	strtok 用セマフォ資源解放に失敗しました
	EIOBFRSM	_iob 用セマフォ資源解放に失敗しました

割り込みに関しては、セマフォ確保後により優先度の高い割り込みが発生し、セマフォ確保を行うとデッドロックが発生するため、リソースを共有するような処理が割り込みでネストしないようにしてください。

```
#define MALLOC_SEM            1     /* malloc 用セマフォ No.        */
#define STRTOK_SEM            2     /* strtok 用セマフォ No.        */
#define FILE_TBL_SEM          3     /* _iob 用セマフォ No.          */
#define SEMSIZE               4
#define TRUE                  1
#define FALSE                 0
#define OK                    1
#define NG                    0

extern int *errno_addr(void);
extern int wait_sem(int);
extern int signal_sem(int);

int sem_errno;
int force_fail_signal_sem = FALSE;
static int semaphore[SEMSIZE];


/*****************************************************************************/
/*                           errno_addr:errno アドレスの取得                 */
/*                              リターン値：errno アドレス                   */
/*****************************************************************************/
int *errno_addr(void)
{
    /* 現在のタスクの errno アドレスを返してください */
    return (&sem_errno);
}


/*****************************************************************************/
/*                           wait_sem:指定されたセマフォの確保               */
/*                              リターン値：OK(=1) (成功)                    */
/*                                          NG(=0) (失敗)                    */
/*****************************************************************************/
int wait_sem(int semnum)      /* セマフォ ID      */
{
    if((0 <= semnum) && (semnum < SEMSIZE)) {
         if(semaphore[semnum] == FALSE) {
             semaphore[semnum] = TRUE;
             return(OK);
         }
    }
    return(NG);
}
```

```
/*****************************************************************************/
/*                          signal_sem:指定されたセマフォの解放                  */
/*                               リターン値：OK(=1) (成功)                       */
/*                                          NG(=0) (失敗)                      */
/*****************************************************************************/
int signal_sem(int semnum)    /* セマフォ ID      */
{
    if(!force_fail_signal_sem) {
         if((0 <= semnum) && (semnum < SEMSIZE)) {
             if( semaphore[semnum] == TRUE ) {
                semaphore[semnum] = FALSE;
                return(OK);
             }
         }
    }
    return(NG);
}
```

(8) 終了処理ルーチン

(a) 終了処理の登録と実行（atexit）終了処理の登録を行うライブラリ atexit 関数の作成法を示します。

atexit 関数では、引数として渡された関数のアドレスを、終了処理のテーブルに登録します。登録された関数の個数が限界値（ここでは、登録できる個数を 32 個とします）をこえた場合、あるいは同じ関数が二度以上登録された場合はリターン値として NULL を返します。そうでなければ NULL 以外の値（この場合は、登録した関数のアドレス）を返します。

以下にプログラム例を示します。
例:

```
#include <stdlib.h>
typedef void *atexit_t ;

int _atexit_count=0 ;

atexit_t (*_atexit_buf[32])(void) ;

#ifdef __cplusplus
extern "C"
#endif
atexit_t atexit(atexit_t (*f)(void))
{
    int i;

   for(i=0; i<_atexit_count ; i++)              // 既に登録されていないかチェックします
      if(_atexit_buf[i]==f)
         return NULL ;

   if (_atexit_count==32)                       // 登録数の限界値をチェックします
      return NULL ;
   else{
      _atexit_buf[_atexit_count++]=f;           // 関数のアドレスを登録します
      return f;
   }
}
```

(b) プログラムの終了（exit）ルーチンの作成例

プログラムの終了処理を行うライブラリ exit 関数の作成法を示します。プログラムの終了処理は、ユーザシステムによって異なりますので、以下のプログラム例を参考に、ユーザシステムの仕様に従った終了処理を作成してください。

exit 関数は、引数として渡されたプログラムの終了コードに従ってプログラムの終了処理を行い、プログラム起動時の環境に戻ります。ここでは、終了コードを外部変数に設定して、main 関数を呼び出す直前に setjmp 関数で退避した環境に戻ることによって実現します。プログラム実行前の環境に戻るためには、次の関数「callmain」を作成し、初期設定関数「PowerON_Reset」から関数「main」を呼び出す代わりに、関数「callmain」を呼び出してください。

以下にプログラム例を示します。

```
#include <setjmp.h>
#include <stddef.h>

typedef void * atexit_t ;
extern int _atexit_count ;
extern atexit_t (*_atexit_buf[32])(void) ;
#ifdef __cplusplus
extern "C"
#endif
void _CLOSEALL(void);
int main(void);
extern jmp_buf _init_env ;
int _exit_code ;

#ifdef __cplusplus
extern "C"
#endif
void exit(int code)
{
    int i;
    _exit_code=code ;                    // _exit_code にリターンコードを設定します
    for(i=_atexit_count-1; i>=0; i--)    // atexit 関数で登録した関数を順次実行します
        (*_atexit_buf[i])();
    _CLOSEALL();                         // オープンした関数を全てクローズします
    longjmp(_init_env, 1) ;              // setjmp で退避した環境にリターンします
}
#ifdef __cplusplus
extern "C"
#endif
void callmain(void)
{
     //  setjmp を用いて現在の環境を退避し、main 関数を呼び出します
     if(!setjmp(_init_env))
         _exit_code=main();              // exit 関数からのリターン時には処理を終了します
}
```

(c) 異常終了（abort）ルーチンの作成例

異常終了の場合は、ご使用になっているユーザシステムの仕様に従ってプログラムを異常終了させる処理を行ってください。

C++プログラムを使用する場合、以下のときにも abort 関数を呼び出します。

例外処理が正しく動作しなかった場合
純粋仮想関数自体をコールした場合
dynamic_cast に失敗した場合
typeid に失敗した場合
クラス配列の delete 時に情報が取れなかった場合
クラスオブジェクトのデストラクタコール情報登録時に矛盾が発生した場合

以下、標準出力装置にメッセージを出力したあと、ファイルをクローズしてから無限ループしてリセットを待つプログラム例を示します。

例:

```
#include <stdio.h>

#ifdef __cplusplus
extern "C"
#endif
void _CLOSEALL(void);
#ifdef __cplusplus
extern "C"
#endif
void abort(void)
{
        printf(“program aborted !!\n”);  // メッセージを出力します
        _CLOSEALL();                     // ファイルをクローズします
        while(1)    ;                    // 無限ループします
}
```

9.3 C/C++プログラムとアセンブリプログラムとの結合

本コンパイラでは、#pragma、キーワードなどの拡張機能や組み込み関数をサポートすることにより、機器組み込み用プログラムに必要な全ての機能を C 言語および C++言語で記述できます。

しかしながら、ハードウェアのタイミング要求やメモリサイズの制限などのように性能要求が厳しい場合、アセンブリ言語で記述し、C/C++プログラムと結合する必要があります。

ここでは、C/C++プログラムとアセンブリプログラムの結合時に留意すべき以下の内容について述べます。

外部名の相互参照方法
関数呼び出しのインタフェース

9.3.1 外部名の相互参照方法

C/C++プログラムの中で外部名として宣言されたものは、アセンブリプログラムとの間で相互に参照あるいは更新することができます。コンパイラは、次のものを外部名として扱います。

グローバル変数であって、かつ static 記憶クラスでないもの(C/C++プログラム)
extern 記憶クラスで宣言されている変数名(C/C++プログラム)
static 記憶クラスを指定されていない関数名(C プログラム)
static 記憶クラスを指定されていない非メンバ非インライン関数名(C++プログラム)
非インラインメンバ関数名(C++プログラム)
静的データメンバ名(C++プログラム)

(1) アセンブリプログラムの外部名を C/C++プログラムで参照する方法

アセンブリプログラムでは、「.EXPORT」制御命令を用いてシンボル名(先頭に下線"_"を付与)を外部定義宣言します。

C/C++プログラムでは、シンボル名(先頭に下線"_"がない)を「extern」宣言します。

アセンブリプログラム(定義する側)

```
     .EXPORT   _a,_b
     .SECTION
D,DATA,ALIGN=2
_a: .DATA.W  1
_b: .DATA.W  1
     .END
```

C/C++プログラム(参照する側)

```
extern int a,b;
f()
{
    a+=b;
}
```

(2) C/C++プログラムの外部(変数および C 関数)名をアセンブリプログラムから参照する方法

C/C++プログラムでは、変数名（先頭に下線"_"がない）を外部定義します。

アセンブリプログラムでは、「.IMPORT」制御命令を用いて外部名（先頭に下線"_"を付与）を外部参照宣言します。

C/C++プログラム(定義する側)

```
char a,b;
```

アセンブリプログラム(参照する側)

```
  .IMPORT   _a,_b
  .SECTION P,CODE,ALIGN=2
  MOV.B     @_a,R5L
  MOV.B     R5L,@_b
  RTS
  .END
```

(3) C++プログラムの外部（関数）名をアセンブリプログラムから参照する方法

アセンブリプログラムで参照する関数を「extern "C"」を用いて宣言することにより、(2)と同じ規則で参照できます。ただし、「extern "C"」を用いて宣言した関数は多重定義できません。

C++プログラム(定義する側)

```
extern "C"
int f(int a)
{
    …
}
```

アセンブリプログラム(参照する側)

```
.IMPORT   _f
.SECTION P,CODE,ALIGN=2
    :
JSR       @_f
    :
.END
```

9.3.2 関数呼び出しのインタフェース

C/C++プログラムとアセンブリプログラム間で相互に関数呼び出しを行うときに、アセンブリプログラム側で守るべき次の 4 つの規則について説明します。

スタックポインタに関する規則
スタックフレームの割り付け、解放に関する規則
レジスタに関する規則
引数、リターン値の設定、参照に関する規則

(1) スタックポインタに関する規則

スタックポインタの指すアドレスよりも下位（0 番地の方向）のスタック領域に、有効なデータを格納してはいけません。割り込み処理で破壊される可能性があります。

(2) スタックフレームの割り付け、解放に関する規則

関数呼び出しが行われた時点（JSR または BSR 命令の実行直後）では、スタックポインタはリターンアドレスの領域を指しています。この領域より上位アドレスのデータの割り付け、設定は呼び出す側の関数の役目です。

関数のリターン時は、リターンアドレスの領域の解放を呼び出される側の関数で行います。これは、通常 RTS 命令を用いて行います。これより上位アドレスの領域（リターン値アドレスおよび引数領域）は、呼び出した側の関数で解放します。

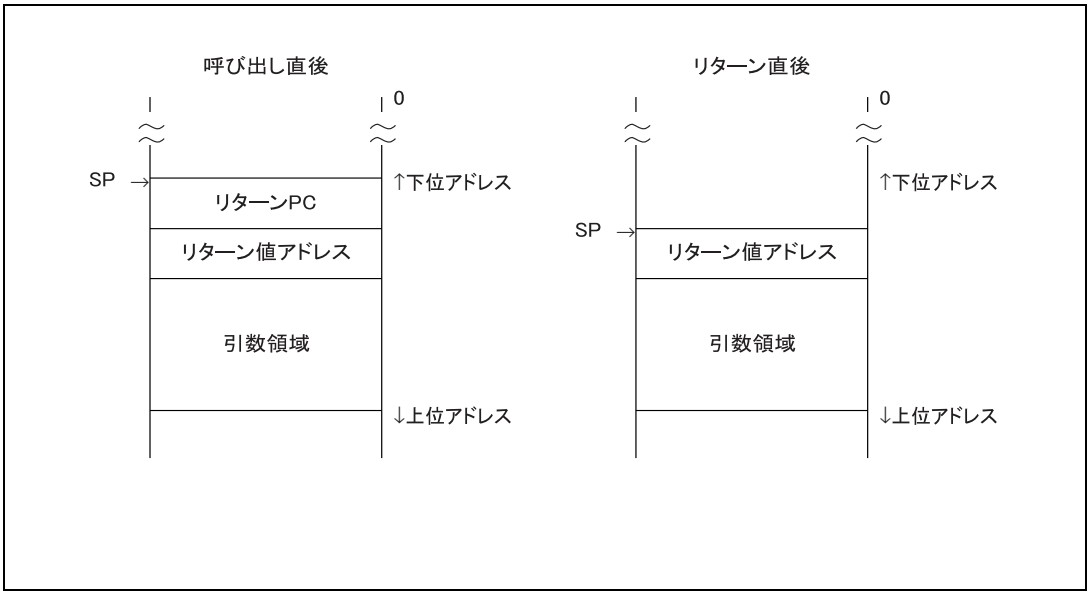


図 9.8 スタックフレームの割り付け、解放に関する規則

(3) **レジスタに関する規則**

関数呼び出し前後において、値を保証するレジスタと保証しないレジスタがあります。
各 CPU 種類におけるレジスタの保証規則を**表** 9.5 に示します。

表 9.5 関数呼び出し前後のレジスタ保証規則

	項目	引数格納レジスタ本数	CPU 種類と対象レジスタ		プログラミングにおける留意点
			H8SX H8S/2600 H8S/2000 H8/300H	H8/300	
1	**保証しないレジスタ** caller-save	2	ER0,ER1	R0,R1	**関数呼び出し時に対象レジスタに有効な値があれば、呼び出し側で値を退避する。呼び出される側の関数では、退避せずに使用可能。**
		3	ER0～ER2	R0～R2	
2	**保証するレジスタ** callee-save	2	ER2～ER6	R2～R6	**対象レジスタのうち関数内で使用するレジスタの値を退避し、リターン時に復帰させる。**
		3	ER3～ER6	R3～R6	

注意 *1：**引数格納レジスタ本数は、**regparam **オプション、**_ _regparam2**、**_ _regparam3 **で指定できます。**

以下、レジスタ保証規則について H8S/2600 アドバンストモードの場合の具体例を示します。

(a) アセンブリプログラムのサブルーチンを C/C++プログラムから呼び出す場合

アセンブリプログラム(呼び出される側)

```
       .EXPORT   _sub
       .SECTION P,CODE,ALIGN=2
_sub: STM.L    (ER4-ER6),@-SP
       SUB.L    #10,SP
             :
       ADD.L    #10,SP
       LDM.L    @SP+,(ER4-ER6)
       RTS
       .END
```

呼び出される側(callee)が
関数内で使用するレジスタの退避

関数本体処理
(ER0,ER1 は退避せずに使用可能)

退避したレジスタの回復

C/C++プログラム(呼び出す側)

```
#ifdef _ _cplusplus
extern “C”
#endif
void sub(void);
void f(void)
{
    sub();
}
```

(b) C プログラムのサブルーチンをアセンブリプログラムから呼び出す場合

C プログラム(呼び出される側)

```
void sub(void)
{
    …
}
```

アセンブリプログラム(呼び出す側)

```
    .IMPORT   _sub
    .SECTION P,CODE,ALIGN=2
             :
    MOV.L    ER1,@(4,SP)
    MOV.L    ER0,ER6
    JSR      @_sub
              :
    RTS
    .END
```

呼び出す側(caller)で
レジスタ ER0,ER1 に有効な値があれば
空きレジスタまたはスタックに退避

関数名は"_"を付加して参照

(c) C++プログラムのサブルーチンをアセンブリプログラムから呼び出す場合

C++プログラム(呼び出される側)

```
extern "C"
void sub(void)
{
    …
}
```

アセンブリプログラム(呼び出す側)

```
    .IMPORT   _sub
    .SECTION P,CODE,ALIGN=2
             :
    MOV.L    ER1,@(4,SP)
    MOV.L    ER0,ER6
    JSR      @_sub
              :
    RTS
    .END
```

} レジスタ ER0,ER1 に有効な値があれば
呼び出す側が空きレジスタまたはスタックに退避

【注】 *1 **「**extern “C”**」を用いて宣言した関数は多重定義できません。**

(4) 引数とリターン値の設定、参照に関する規則

以下、引数とリターン値の設定、参照法について説明します。引数とリターン値の規則は、関数の宣言において、個々の引数とリターン値の型が明示的に宣言されているかどうかによって異なります。引数とリターン値の型を明示的に宣言するには、関数の原型宣言を用います。

以下の解説では、まず引数とリターン値に対する一般的な規則について述べたあと、引数の割り付け方とリターン値の設定場所について述べます。

(a) 引数とリターン値に対する一般的な規則

- **引数の渡し方**

  引数の値を、必ず引数の割り付け領域にコピーしたあとで関数を呼び出します。呼び出した側の関数では、リターン後に引数の割り付け領域を参照することはありませんので、呼び出された側の関数で引数の値を変更しても呼び出した側の処理は直接には影響を受けません。

- **型変換の規則**

  引数を渡す場合、またはリターン値を返す場合、自動的に型変換を行う場合があります。以下、この型変換の規則について説明します。

  – リターン値の型変換
  リターン値は、その関数の返す型に変換します。
  – 型の宣言された引数の型変換
  原型宣言によって型が宣言されている引数は、宣言された型に変換します。
  – 型の宣言されていない引数の型変換
  原型宣言によって型が宣言されていない引数の型変換は、以下の規則に従って変換します。
    - char 型、unsigned char 型の引数は、int 型に変換します。
    - float 型の引数は、double 型に変換します。
    - 上記以外の型は、変換しません。

例：

```
(1) long f();

    long f()
     {    float x;
          return x;
     }
```

リターン値はlong型に型変換します。

```
(2) void p(int,...);

    f()
     {    char c;
          p(1.0, c);
     }
```

cは、対応する引数の型宣言がないので、int型に変換します。

1.0は、対応する引数の型がint型なので、int型に変換します。

注意　原型宣言によって引数の型を宣言していない場合、正しく引数が渡されるように呼び出される側と呼び出す側で同じ型を指定しないと、動作を保証しません。

```
f(x)
float x;
{
      .
      .
      .
 }

main()
{
   float x;
   f(x);
 }
```

動作を保証しない指定例

```
f(float x)
{
      .
      .
      .
 }

main()
{
   float x;
   f(x);
 }
```

正しい指定例

動作を保証しない指定例では、関数「f」の引数の原型宣言がないため、関数「main」の側で呼び出すときに引数 x を double 型に変換します。

一方、関数「f」の側では引数を float 型として宣言していますので正しく引数を受け渡しすることはできません。原型宣言によって引数の型を宣言するか、関数「f」の側の引数宣言を double 型にする必要があります。

正しい指定例は、原型宣言によって引数の型を宣言した例です。

(b)　引数の割り付け領域

引数は、スタック上の引数領域に割り付ける場合と、レジスタに割り付ける場合があります。

オブジェクト種類ごとの引数の割り付け領域を**表** 9.6 に、引数割り付け領域の一般規則を**図** 9.9 にそれぞれ示します。

C++プログラムの非静的関数メンバの this ポインタは、R0 または ER0 に割り付けられます。

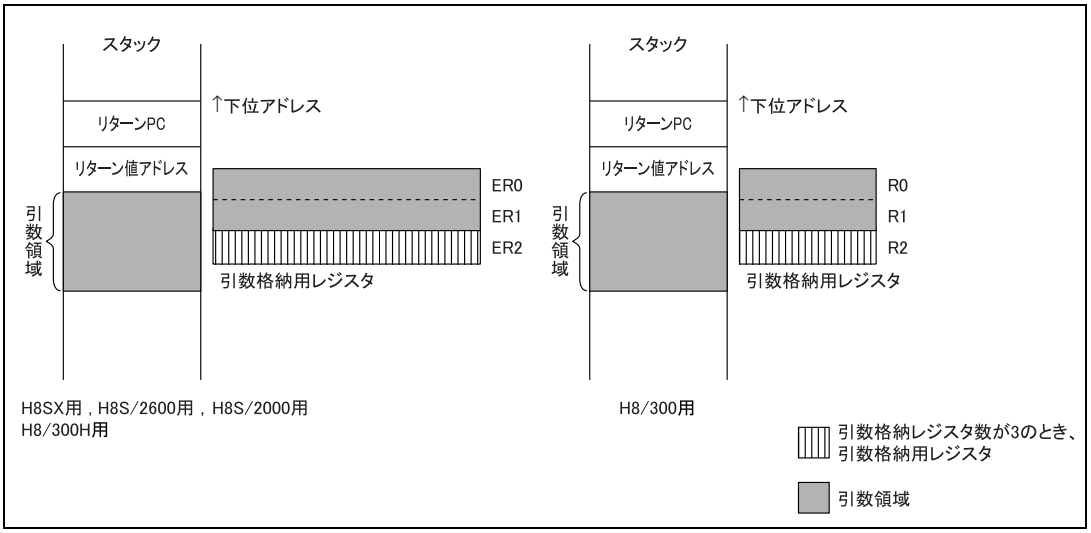


図 9.9 引数の割り付け領域

表 9.6 引数割り付け領域の一般規則

CPU 種別	引数格納レジスタ数	割り付け規則		スタックに割り付ける引数
		レジスタに割り付ける引数		
		引数格納用レジスタ	対象の型	
H8SX H8S/2600 H8S/2000 H8/300H	2	ER0、ER1	char、unsigned char、short、unsigned short、int、unsigned int、long、unsigned long、float、**構造体**(4byte **以下**）*4、**ポインタ、リファレンス、データメンバへのポインタ**	[1] **引数の型がレジスタ渡しの対象の型以外のもの** [2] **原型宣言により可変個の引数をもつ関数として宣言しているもの***2 [3] **引数の数が多いため、レジスタに割り付かなかったもの**
	3	ER0、ER1、ER2		
H8/300	2	R0、R1	char、unsigned char、short、unsigned short、int、unsigned int、long*3、unsigned long*3、float*3、**構造体**(2byte **以下**）*4、**構造体**(4byte **以下**）*3*4、**ポインタ、リファレンス、データメンバへのポインタ**	
	3	R0、R1、R2		

【注】 *1 **引数格納レジスタ数は、**regparam **オプション、**__regparam2**、**__regparam3 **で指定できます。**

*2 **原型宣言により可変個の引数をもつ関数として宣言している場合、**...**部分およびその直前の引数はスタックに割り付けます。**

例： int f2(int,int,...);

f2(x,y,z); → y,z **はスタックに割り付けます。**

*3 longreg **オプション指定時に対象になります。**

*4 structreg **オプション指定時に対象になります。**

(c) 引数の割り付け

・引数格納用レジスタへの割り付け

引数格納用レジスタには、ソースプログラムの宣言順に番号の小さい、LSB 側のレジスタから割り付けます。引数格納用レジスタの割り付け例を**図** 9.10 に示します。

```
f(char a,int b)
{
      :
}
```

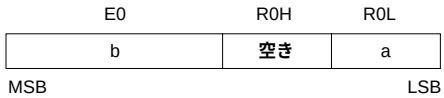


図 9.10 **引数格納用レジスタの割り付け例（**H8S/2600 **用）**

・スタック上の引数領域への割り付け
スタック上の引数領域には、ソースプログラム上で指定した順に下位アドレスから割り付けます。

注意　構造体型、共用体型、クラス型の引数を設定する場合は、その型の本来の境界調整にかかわらず 2 **バイト境界に割り付け、しかもその領域として偶数バイトの領域を使用します。**
これは、H8SX**、**H8S**、**H8/300H**、**H8/300 **シリーズのスタックポインタが** 2 **バイト単位で変化するためです。**

「9.3.3 **引数割り付けの具体例」**に、各 CPU／動作モードに対する引数割り付けの具体例がありますので、合わせて参照してください。

(d)　**リターン値の設定場所**

関数のリターン値の型により、リターン値をレジスタに設定する場合とメモリに設定する場合があります。リターン値の型と設定場所の関係は**表**9.7を参照してください。
関数のリターン値をメモリに設定する場合、リターン値はリターン値アドレスの指す領域に設定します。呼び出す側では、引数領域のほかにリターン値設定領域を確保し、そのアドレスをリターン値アドレスの領域に設定してから関数を呼び出します（**図**9.11**参照**）。

関数のリターン値がvoid型の場合、リターン値を設定しません。

表 9.7 リターン値の型と設定場所

リターン値の型	リターン値の設定場所 H8SX 用 H8S/2600 用 H8S/2000 用 H8/300H 用	H8/300 用
char、unsigned char	レジスタ（R0L）	レジスタ（R0L）
short、unsigned short、int、unsigned int	レジスタ（R0）	レジスタ（R0）
関数へのポインタ	レジスタ ノーマルモード：（R0） それ以外のモード：（ER0）	レジスタ（R0）
データへのポインタ、リファレンス、データメンバへのポインタ	レジスタ ノーマル/ミドルモード：（R0） アドバンスト/マキシマムモード かつ、ptr16 オプションまたは _ _ptr16 キーワード有り：（R0）*3 アドバンスト/マキシマムモード かつ、ptr16 オプションと _ _ptr16 キーワード無し：（ER0）	レジスタ（R0）
long, unsigned long, float	レジスタ（ER0）	リターン値設定領域（メモリ） レジスタ（R0, R1）*1
2byte 以下の構造体	リターン値設定領域（メモリ） レジスタ（R0）*2	リターン値設定領域（メモリ） レジスタ（R0）*2
3 byte または 4byte の構造体	リターン値設定領域（メモリ） レジスタ（ER0）*2	リターン値設定領域（メモリ） レジスタ（R0,R1）*1*2
double、long double、構造体、共用体、クラス、関数メンバへのポインタ	リターン値設定領域（メモリ）	リターン値設定領域（メモリ）

【注】 *1 longreg オプション指定時

*2 structreg オプション指定時

*3 ptr16 オプションと_ _ptr16 キーワードは H8SX 指定時のみ有効

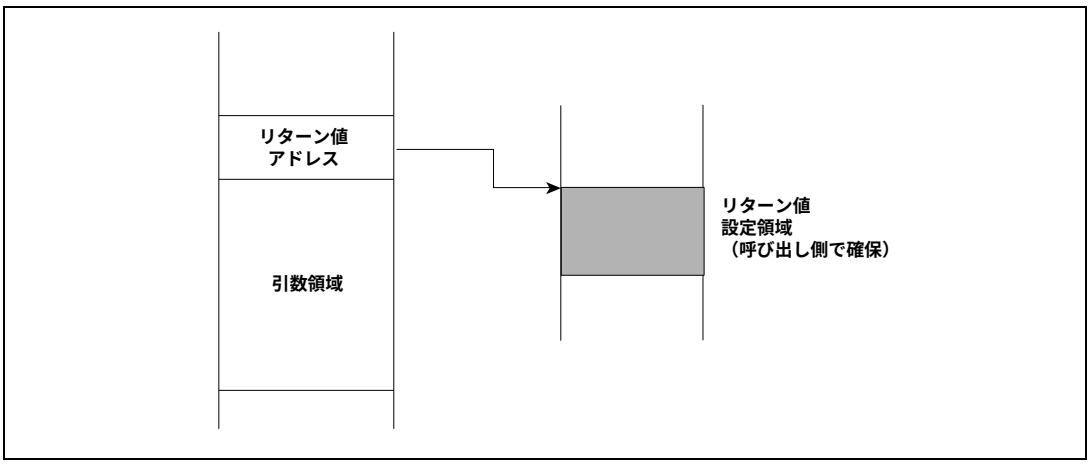


図 9.11 リターン値をメモリに設定する場合の設定領域

9.3.3 引数割り付けの具体例

(1) H8SX 用、H8S/2600 用、H8S/2000 用、H8/300H 用
（cpu＝H8SXN、cpu＝H8SXM、cpu＝H8SXA、cpu＝H8SXX、
cpu＝2600a、cpu＝2600n、cpu＝2000a、cpu＝2000n、cpu＝300ha、cpu＝300hn）

例 1：レジスタ渡しの対象の型である引数は、宣言順にレジスタ ER0、ER1[*1] に割り付けます。

[1]
```
int f(char,char,char);
        :
f(1,2,3);
        :
```
R0L 1
R0H 2
R1L 3

[2]
```
int f(int,int,int);
        :
f(1,2,3);
        :
```
R0 1
E0 2
R1 3

[3]
```
int f(long,long);
        :
f(1,2);
        :
```
ER0 1
ER1 2

[4]
```
int f(char,int,int,char);
        :
f(1,2,3,4);
        :
```
R0L 1
E0 2
R1 3
R0H 4

*1:**引数格納レジスタ数が３のときは、**ER0,ER1,ER2

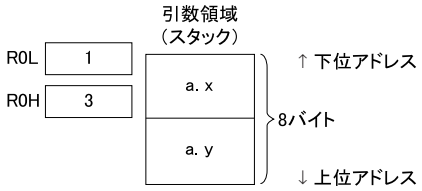
例 2：レジスタに割り付けることができなかった引数は、スタックに割り付けます。
また、引数の型が char 型で、スタック上の引数領域に割り付けた場合、下位アドレスに無効バイトができます。

（引数格納レジスタ数２個の場合）

```
int f(int,long,char);
        :
f(1,2,3);
        :
```
R0 1
ER1 2
引数領域（スタック）
無効バイト
3
2バイト
↑下位アドレス
↓上位アドレス

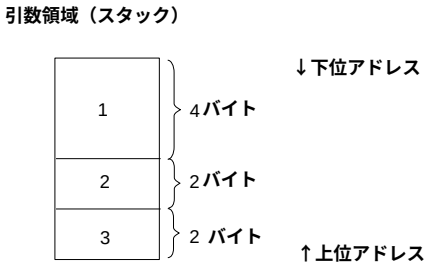
例 3：レジスタに割り付けられない型の引数は、スタックに割り付けます。

```
struct s{long x,y;)a;
int f(char,struct s,char);
      :
    f(1,a,3);
      :
```

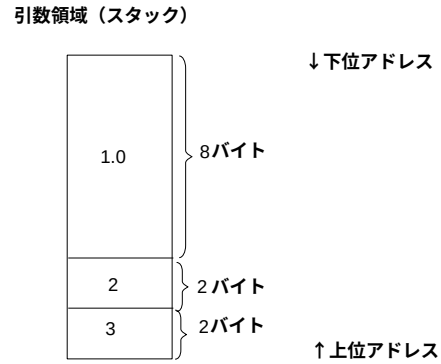


例 4：原型宣言により可変個の引数を持つ関数として宣言している場合、…部分の引数およびその直前の引数は、宣言順にスタックに割り付けます。

[1]
```
int f(long,...);
      :
    f(1,2,3);
      :
```

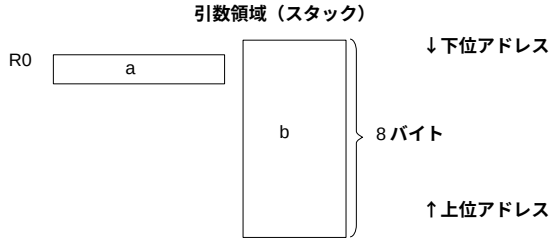


[2]
```
int f(double,int,...);
      :
    f(1.0,2,3);
      :
```



例 5：C プログラムで原型宣言がない場合、char 型は int 型に、float 型は double 型に拡張して渡します。

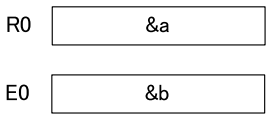
```
int f( );
char a;
float b;
      :
    f(a,b);
      :
```



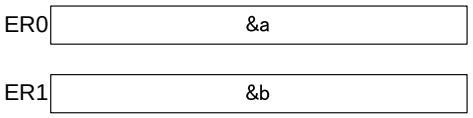
例 6：データへのポインタ型と C++のリファレンス型は、ノーマルモード／ミドルモードでは 2 バイト、アドバンストモード／マキシマムモードかつ ptr16 オプション有りまたは_ _ptr16 キーワード有りでは 2 バイト、アドバンストモード／マキシマムモードかつ ptr16 オプション無しかつ_ _ptr16 キーワード無しでは 4 バイトの領域に割り付けられます。但し、ptr16 オプションと_ _ptr16 キーワードは H8SX のみで有効です。

```
int a,b;
int f(int *,int &);
          :
     f(&a,b);
          :
```

ノーマル/ミドルモードのとき
アドバンスト/マキシマムモードでptr16オプション有りのとき

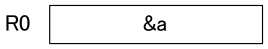


アドバンスト/マキシマムモードでptr16オプション無しのとき




```
int g(int _ _ptr16 *);
          :
      g(&a);
          :
```

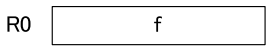
H8SXアドバンスト/マキシマムモードで
_ _ptr16キーワード有りのとき



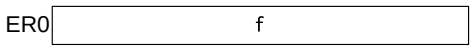
例 7： データへのポインタ型のリターン値は、ノーマルモード／ミドルモードでは 2 バイト、アドバンストモード／マキシマムモードかつ ptr16 オプション有りまたは_ _ptr16 キーワード有りでは 2 バイト、アドバンストモード／マキシマムモードかつ ptr16 オプション無しかつ_ _ptr16 キーワード無しでは 4 バイトになります。但し、ptr16 オプションと_ _ptr16 キーワードは H8SX のみで有効です。

```
int *f(void);
int *p;
          :
      p = f( );
          :
```

ノーマル/ミドルモードのとき
アドバンスト/マキシマムモードでptr16オプション有りのとき

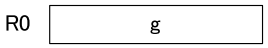


アドバンスト/マキシマムモードでptr16オプション無しのとき



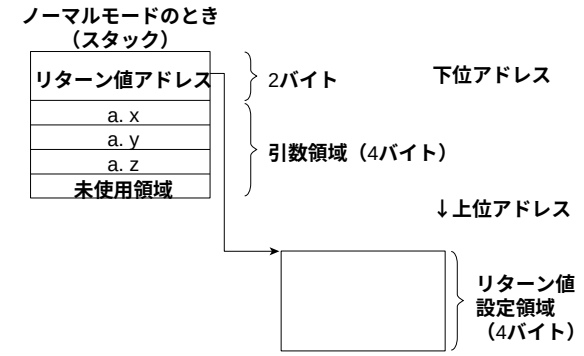

```
int _ _ptr16 *g(void);
int _ _ptr16 *q;
          :
     q = g( );
          :
```

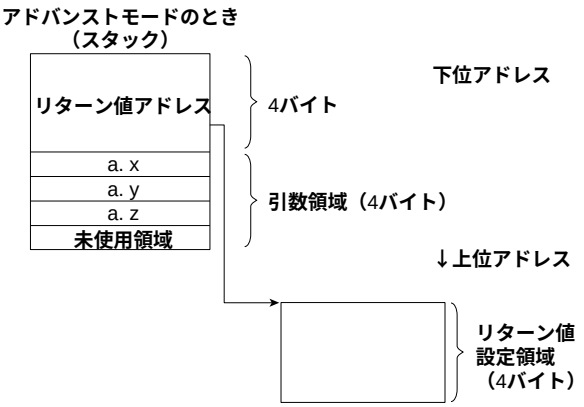
アドバンスト/マキシマムモードで
_ _ptr16キーワード有りのとき



例 8：関数の返す型が 4 バイトをこえる場合または構造体(structreg を指定していないとき、または 4 バイトをこえる構造体）の場合、引数領域の直前にリターン値アドレスを設定します。また、構造体のサイズが奇数バイトのとき、1 バイトの未使用領域が生じます。

```
struct s{char x,y,z;}a,b;
float f(struct s);
          :
         f(a);
          :
          :
```

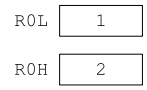




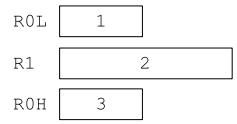
(2) H8/300 用（cpu＝300）

例 1：レジスタ渡しの対象の型である引数は、宣言順にレジスタ R0、R1[*1] **に割り付けます。**

```
[1] int f(char,char);
      :
    f(1,2);
      :
```




```
[2] int f(char,int,char);
      :
    f(1,2,3);
      :
```

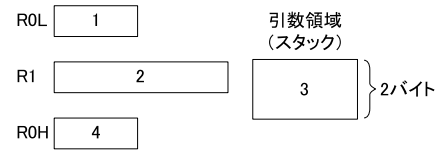


*1:**引数格納レジスタ数が３のときは、**R0,R1,R2

例 2：レジスタに割り付けることができなかった引数は、スタックに割り付けます。

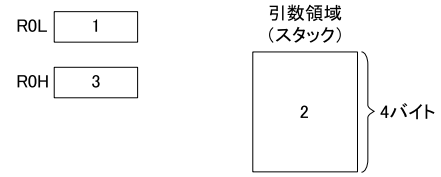
（引数格納レジスタ数２個の場合）

```
int f(char,int,int,char);
     :
   f(1,2,3,4);
     :
```



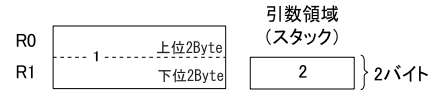
例 3：レジスタに割り付けられない型の引数は、スタックに割り付けます。

```
int f(char,long,char);
       :
    f(1,2,3);
       :
```



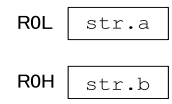
例 4：longreg **オプションを指定した場合、**4 **バイトデータをレジスタ** R0､R1 **に割り付けます。**

```
int f(long,Short);
     :
   f(1,2);
     :
```



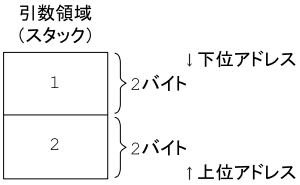
例 5：structreg **オプションを指定した場合、**2 **バイト以下の構造体をレジスタに割り付けます。**

```
struct A{
   char a,b;
}str;

int f(struct A);
     :
   f(str)
     :
```

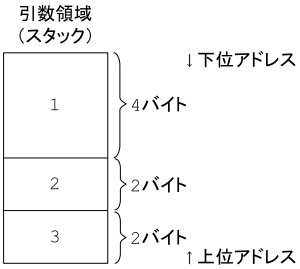


例 6：原型宣言により可変個の引数を持つ関数として宣言している場合、対応する型のない引数およびその直前の引数は、宣言順にスタックに割り付けます。

```
[1]  int f(int,...);
       :
     f(1,2);
       :
```

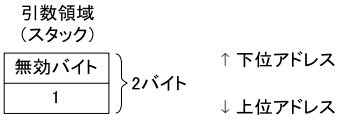



```
[2]  int f(long,int,...);
       :
     f(1,2,3);
       :
```



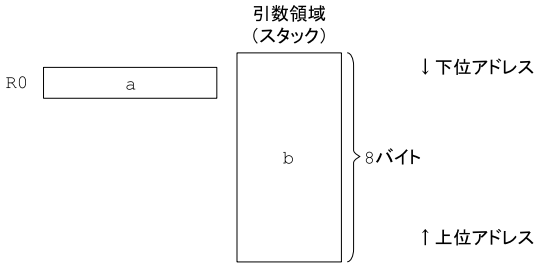
例 7：引数の型が char 型で、スタック上の引数領域に割り付けた場合、下位アドレスに無効バイトができます。

```
int f(char,...);
       :
     f(1);
       :
```



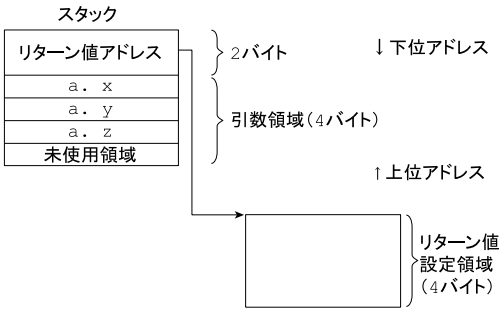
例 8：C プログラムで原型宣言がない場合、char 型は int 型に、float 型は double 型に拡張して渡します。

```
int f( );
char a;
float b;
       :
     f(a,b);
       :
```



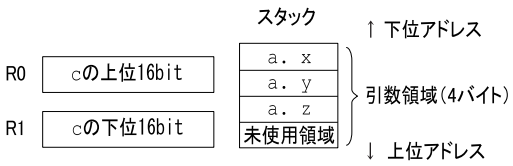
例 9：関数の返す型が 2 バイトをこえる場合、引数領域の直前にリターン値アドレスを設定します。また、構造体のサイズが奇数バイトのとき、1 バイトの未使用領域が生じます。

```
struct s{char x,y,z;}a,b;
float f(struct s);
        :
f(a);
        :
        :
```



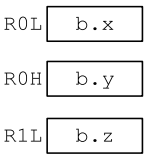
例 10：longreg を指定し、関数の返す型が 2 バイトをこえる場合、リターン値を R0、R1 に割り付けます。

```
struct s{char x,y,z;}a,b;
long c;
long f(struct s);
     :
c=f(a);
     :
```



例 11：structreg、longreg を指定し、関数の返す型が 4 バイト以下の構造体である場合、リターン値を R0、R1 に割り付けます。

```
struct s{char x,y,z;}a,b;
struct s f(void);
     :
b=f();
     :
```



9.3.4 レジスタとスタック領域の使用法

(1) H8SX **用アドバンストモード、マキシマムモード**（cpu＝H8SXA 、cpu＝H8SXX）

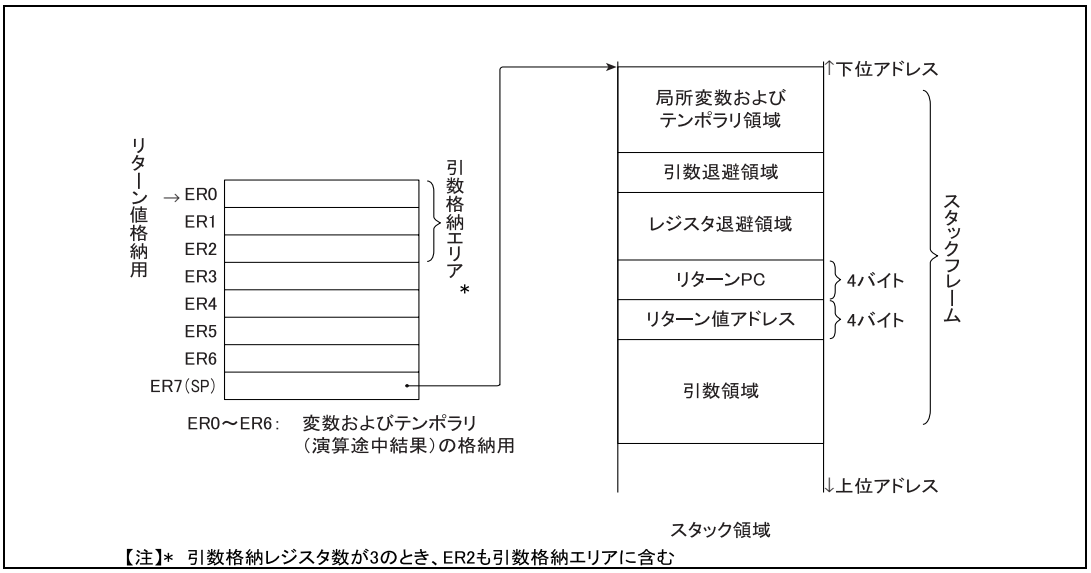


図 9.13 **レジスタとスタック領域の使用法**（cpu＝H8SXA*[1]、cpu＝H8SXX*[1]）

[注] *1: ptr16 オプション無し。

(2) H8SX **用ミドルモード、アドバンストモード**(ptr16)、**マキシマムモード**(ptr16)
（cpu＝H8SXM 、cpu＝H8SXA **かつ** ptr16、cpu=H8SXX **かつ** ptr16）

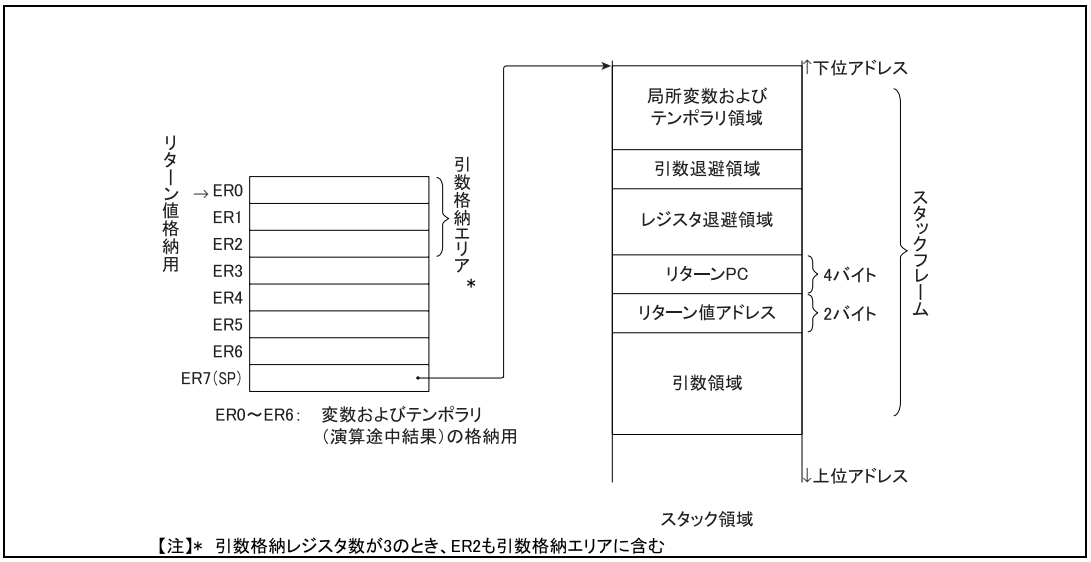


図 9.14 **レジスタとスタック領域の使用法**（cpu＝H8SXM、cpu＝H8SXA*[2]、cpu＝H8SXX*[2]）

[注] *2: ptr16 オプション有り。

(3) H8SX 用ノーマルモード（cpu＝H8SXN）

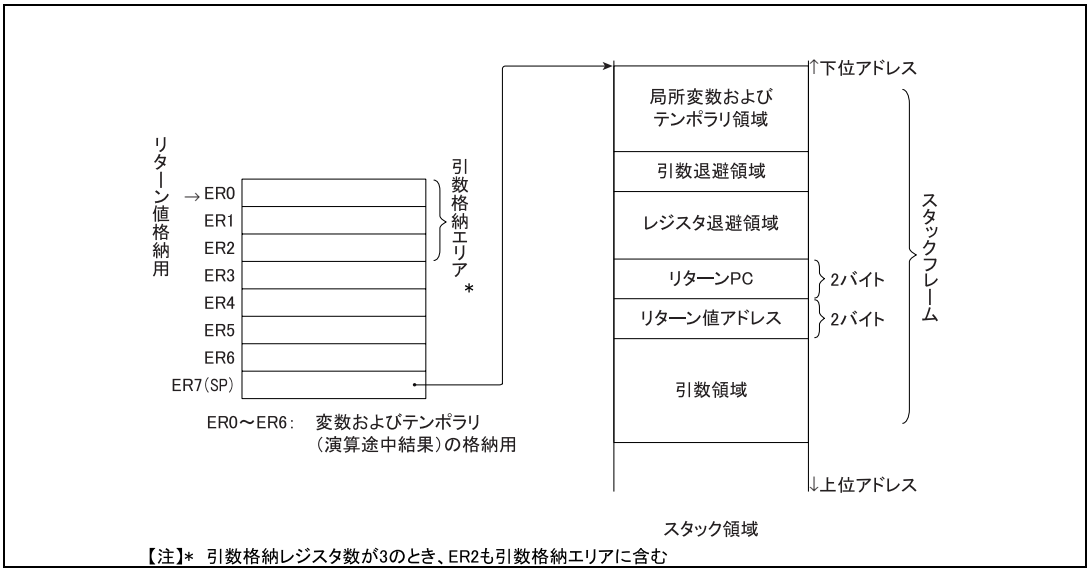


図 9.15 **レジスタとスタック領域の使用法**（cpu＝H8SXN）

(4) H8S/2600 **用、**H8S/2000 **用、**H8/300H **用アドバンストモード**
(cpu＝2600a、cpu＝2000a、cpu＝300ha**)**

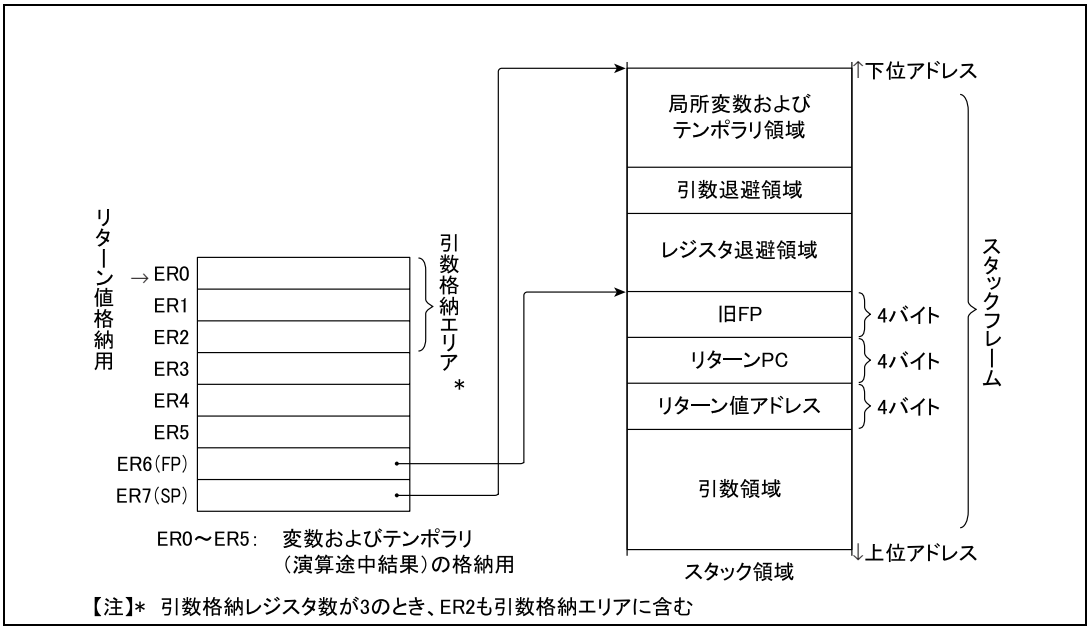


図 9.16 **非最適化時のレジスタとスタック領域の使用法**
(cpu＝2600a、cpu＝2000a、cpu＝300ha**)**

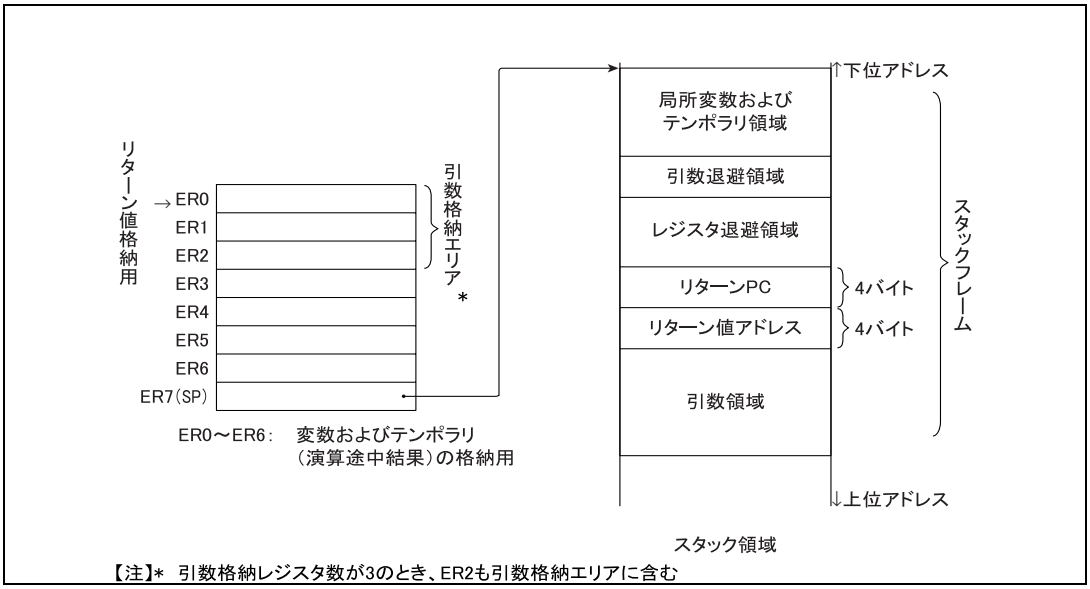


図 9.17 **最適化時のレジスタとスタック領域の使用法**
(cpu＝2600a、cpu＝2000a、cpu＝300ha**)**

(5) H8S/2600 用、H8S/2000 用、H8/300H 用ノーマルモード
（cpu＝2600n、cpu＝2000n、cpu＝300hn）

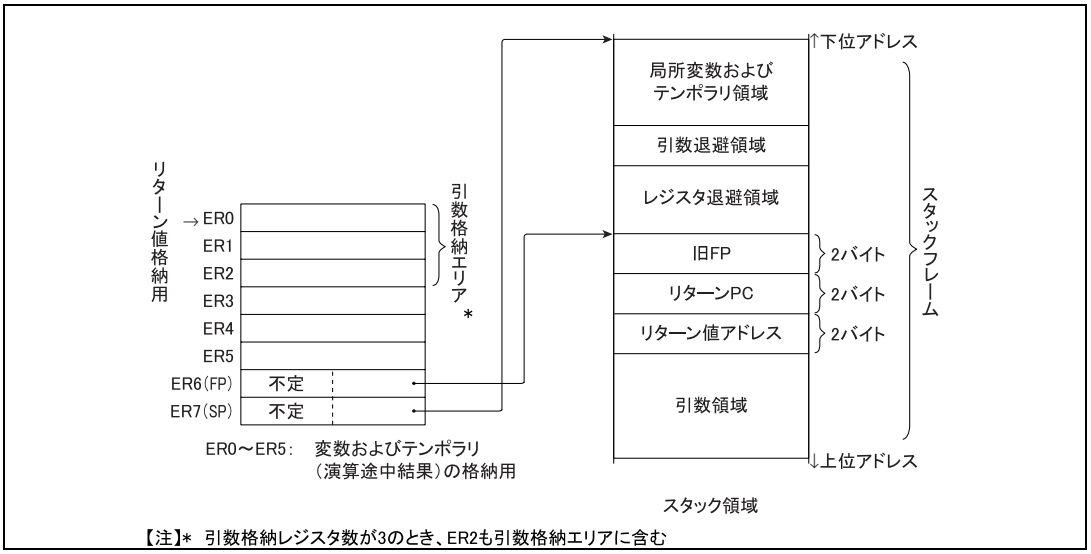


図 9.18 **非最適化時のレジスタとスタック領域の使用法**
（cpu＝2600n、cpu＝2000n、cpu＝300hn）

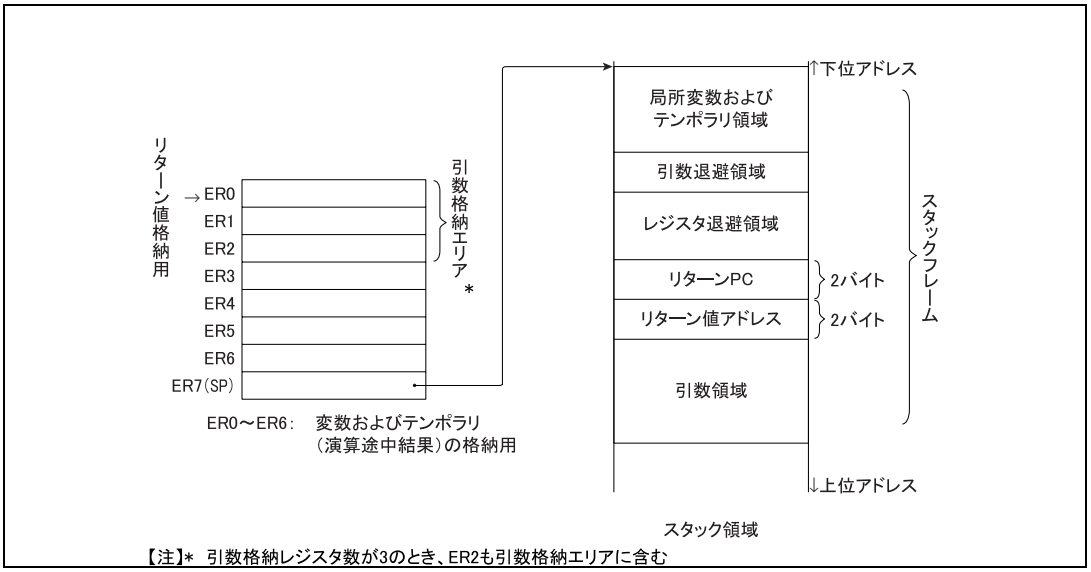


図 9.19 **最適化時のレジスタとスタック領域の使用法**
（cpu＝2600n、cpu＝2000n、 cpu＝300hn）

(6) H8/300 **用（**cpu＝300**）**

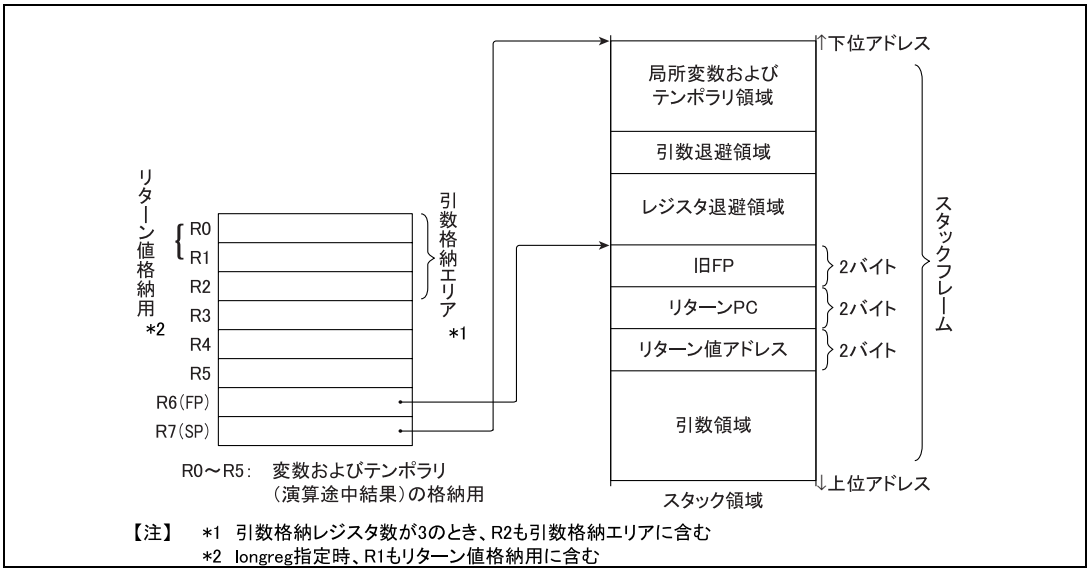


図 9.20 **非最適化時のレジスタとスタック領域の使用法（**H8/300 **用）**

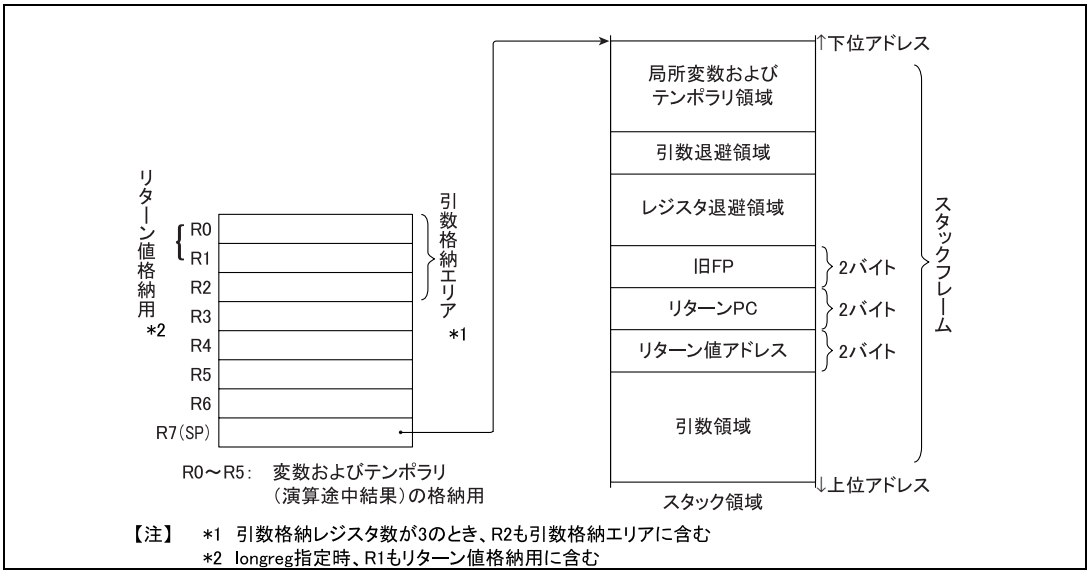


図 9.21 **最適化時のレジスタとスタック領域の使用法（**H8/300 **用）**

9.4 プログラム作成上の注意事項

本節では、コンパイラにおけるコーディング上の注意事項と、コンパイルからデバッグまでのプログラム開発上の注意事項を述べます。

9.4.1 コーディング上の注意事項

(1) float 型引数の関数

float 型の引数を宣言している関数は、必ず、原型宣言を行うか、引数の宣言の float 型を double 型に変更してください。原型宣言を行わず float 型引数を持つ関数を呼び出した場合、動作は保証しません。

例：

```
void f(float);――――［1］
void g( )
{
    float a;
     :
     f(a);
}

void f(float x)
{
     :
}
```

関数「f」は、float 型の引数を持つ関数です。この場合は、必ず［1］に示すように原型宣言を行ってください。

(2) C/C++言語仕様で評価順序を規定していない式

C/C++言語仕様で評価順序を規定していない式で、評価順序で結果が変わるようなコーディングをした場合、その動作は保証しません。

例：

`a[i]=a[++i];` 代入式の右辺を先に評価するか後に評価するかで、左辺の`i`の値が変わります。

`sub(++i, i);` 関数の第`1`引数を先に評価するか後に評価するかで第`2`引数の`i`の値が変わります。

(3) 最適化により削除される可能性のあるコーディング

連続した同一変数の参照や、結果を使用しない式を記述した場合、コンパイラの最適化により冗長コードとして削除される場合があります。常にアクセスを保証する場合は、宣言時に volatile を指定してください。

例：

```
[1] b=a;          /* 1行目の式は冗長コードとして削除されることがあります。*/
     b=a;
[2] while(1)a;    /* 変数aの参照およびループ文は冗長コードとして削除される*/
                  /* ことがあります。                                      */
```

(4) オーバフロー演算、ゼロ除算

オーバフロー演算やゼロ除算があっても、エラーメッセージを出力しません。ただし、一つの定数または定数どうしの演算で、オーバフロー演算やゼロ除算があれば、コンパイル時にエラーメッセージを出力します。但し、H8SX の場合、コンパイラがゼロ除算を検出しない場合があります。

例：

```
void main(void)
{
    int ia;
    int ib;
    float fa;
    float fb;
    ib=32767;
    fb=3.4e+38f;

/*  定数または定数どうしの演算時はオーバフロー、0除算に対する                */
/*  コンパイルエラーメッセージを出力します                                    */
    ia=99999999999;       /*（W）定数のオーバフローを検出します          */
    fa=3.5e+40f;          /*（W）浮動小数点演算のオーバフローを検出します*/
    ia=1/0;               /*（E）0除算を検出します(H8SX除く)             */
    fa=1.0/0.0;           /*（W）浮動小数点の0除算を検出します(H8SX除く)*/

/*  実行時のオーバフローに対するエラーメッセージは出力しません              */
    ib=ib+32767;          /* 演算結果のオーバフローを無視します          */
    fb=fb+3.4e+38f ;      /* 浮動小数点演算結果のオーバフローを無視します*/
}
```

注意　cpuexpand **オプションを指定した場合、オーバフロー、アンダフローのエラーは出力しません。**

(5) 数学関数ライブラリの精度について

acos (x)、asin (x)関数では x　1 で誤差が大きくなりますので注意が必要です。

誤差範囲は以下のとおりです。

acos(1.0 – ε)における絶対誤差倍精度2^{-39} ($\varepsilon = 2^{-33}$)

単精度2^{-21} ($\varepsilon = 2^{-19}$)

asin(1.0 – ε)における絶対誤差倍精度2^{-39} ($\varepsilon = 2^{-28}$)

単精度2^{-21} ($\varepsilon = 2^{-16}$)

(6) const 型変数への書き込み

const 型の変数を宣言していても、型変換で const 型でない型に変換して代入した場合や、分割コンパイルしたプログラムの間で、型を統一して扱っていない場合は、const 型変数への書き込みをコンパイラでチェックできませんので、注意が必要です。

例：

```
［1］const char *p ;       /* ライブラリ関数strcatの第1引数はchar型     */
          :                /* へのポインタ型なので、                     */
     strcat(p,"abc") ;     /* 引数の指す領域が書き換わることがあります。 */
```

［2］ファイル１

```
const int i;
```

ファイル２

```
extern int i;          /* 変数iは、ファイル2ではconst型で宣言して     */
     :                 /* いませんのでファイル2の中で                 */
i=10;                  /* 書き込んでもエラーになりません。            */
```

(7) ビット操作命令に関する注意事項

本コンパイラは、BSET、BCLR、BNOT、BST、BIST の各ビット操作命令を生成します。これらの命令は、バイト単位でデータを読み込み、ビット操作後に再びバイト単位でデータを書き込みます。一方 CPU は、ライト専用レジスタを読み込むと、レジスタの内容に関係なく不定値のデータを取り込みます。このため、ライト専用レジスタのビット操作命令では、操作するビット以外のビットの内容が変化してしまう場合があります。以下にライト専用レジスタに対する操作例を示します。

例：

インクルードファイル(300x.h)の内容

```
struct S_p4ddr{
   unsigned char p7:1;
           :
   unsigned char p0:1;
};
union
   unsigned char Schar;
   struct S_p4ddr Sstr;
};
#define P4DDR (*(union SS *)0xffffc5)
#define P0 0x1
```

Ｃソースプログラムの内容

```
#include "300x.h"
unsigned char DDR;
//書き込み専用レジスタのバック
//アップ用データを用意します
void sub(void)
{
   DDR &=~P0;
   P4DDR.Schar=DDR;
}
```

9.4.2 C プログラムを C++コンパイラでコンパイルするときの注意事項

(1) 関数のプロトタイプ宣言

関数を使用する前にプロトタイプ宣言が必要です。そのときには、仮引数の型も必ず宣言してください。

```
extern void func1();
void g()
{
    func1(1); // C++でエラー
}
```

```
extern void func1(int);
void g()
{
    func1(1); // OK
}
```

(2) const オブジェクトのリンケージ

const オブジェクトのリンケージは、C プログラムでは外部結合であるのに対し、C++プログラムでは内部結合になります。また、const オブジェクトは初期値を必要とします。

```
const cvalue1;     // C++でエラー

const cvalue2 = 1; // C++で内部的
```

```
const cvalue1=0;
   // 初期値を与えます

extern const cvalue2 = 1;
   // Cプログラムと同様に外部結合に
   // なります
```

(3) void*からの代入

C++プログラムでは、明示的なキャストを用いないと他のオブジェクト型へのポインタ（関数へのポインタ、メンバへのポインタ除く）へ代入できません。

```
void func(void *ptrv, int *ptri)
{
    ptri = ptrv; // C++でエラー
}
```

```
void func(void *ptrv, int *ptri)
{
    ptri = (int *)ptrv; //OK
}
```

9.4.3 プログラム開発上の注意事項

プログラムの作成からデバッグまでのプログラム開発上の注意事項を示します。

(1) CPU／動作モードの選択に関する注意事項

(a) コンパイル、アセンブル時に指定する CPU／動作モードは統一してください。

コンパイル、アセンブル時に cpu オプションを用いて指定する CPU／動作モードは、必ず統一してください。異なった CPU／動作モードで作成したオブジェクトプログラムを一緒にリンクした場合、オブジェクトプログラム実行時の動作は保証しません。

(b) アセンブル時はコンパイル時の CPU／動作モードと同じ CPU 種類を指定してください。

C コンパイラが生成したアセンブリプログラムをアセンブルするとき、コンパイル時に指定した CPU／動作モードと同じ CPU 種類を cpu オプションで指定してください。

(c) 標準ライブラリ作成時にはコンパイル時の CPU／動作モードと同じ CPU 種類を指定してください。

標準ライブラリ構築ツールを用いて標準ライブラリを作成する時、コンパイル時に指定した CPU／動作モードと同じ CPU 種類を cpu オプションで指定してください。

(2) オプションに関する注意事項

下記のオプションは、コンパイル時、ライブラリ構築時に必ず統一して下さい。異なるオプションを用いて作成したオブジェクトプログラムを一緒にリンクした場合、オブジェクトプログラム実行時の動作は保証しません。

exception/noexception
rtti = on/off
regparam
longreg/nolongreg
structreg/nostructreg
stack
double=float
byteenum
pack
bit_order = left/right
indirect = normal/extended
(indirect オプション自体の指定のあり/なしは混在可能、normal と extended は混在不可能)
ptr16
sbr

10. C/C++言語仕様

10.1 言語仕様

10.1.1 コンパイラの仕様

言語仕様で規定していない処理系定義項目について、コンパイラの仕様を示します。

(1) 環境

表 10.1 環境の仕様

	項 目	コンパイラの仕様
1	main 関数への実引数の意味	規定しません。
2	対話的入出力装置の構成	規定しません。

(2) 識別子

表 10.2 識別子の仕様

	項 目	コンパイラの仕様
1	外部結合とならない識別子（内部名）の有効文字数	8189 文字まで有効です。
2	外部結合となる識別子（外部名）の有効文字数	8191 文字まで有効です。
3	外部結合となる識別子（外部名）の大文字と小文字の区別	大文字と小文字を区別します。

(3) 文字

表 10.3 文字の仕様

	項 目	コンパイラの仕様
1	ソース文字集合および実行環境文字集合の要素	どちらも ASCII 文字集合です。ただし、文字列、文字定数にはシフト JIS、EUC 漢字コードまたは Latin1 コードを記述できます。
2	多バイト文字のコード化で使用されるシフト状態	シフト状態はサポートしていません。
3	プログラム実行時の文字集合の文字のビット数	ビット数は 8 ビットです。
4	文字定数内、文字列内のソース文字集合の文字と実行環境文字集合の文字との対応付け	同じ ASCII 文字に対応します。
5	言語で規定していない文字や拡張表記を含む整数文字定数の値	言語で規定する以外の文字、拡張表記はサポートしていません。
6	2 文字以上の文字を含む文字定数または 2 文字以上の多バイト文字を含む広角文字定数の値	文字定数は上位 2 文字を有効とします。広角文字定数はサポートしていません。また、1 文字より多く指定した場合はウォーニングエラーを出力します。
7	多バイト文字を広角文字に変換するために使用される locale の仕様	locale はサポートしていません。
8	char 型の値	signed char 型と同じ値の範囲を持ちます。

(4) **整数**

表 10.4 **整数の仕様**

	項　目	コンパイラの仕様
1	整数型の表現方法とその値	表 10.5 に示します。
2	整数の値がより短いサイズの符号付き整数型、または符号なし整数型を同一のサイズの符号付き整数型に変換したときの値（結果の値が変換先の型で表現できない場合）	整数の値の下位 2 バイトあるいは下位 1 バイトが変換後の値になります。
3	符号付き整数に対するビットごとの演算の結果	符号付きの値になります。
4	整数除算における剰余の符号	被除数の符号と同符号になります。
5	負の値を持つ符号付き汎整数型の右シフトの結果	符号ビットを保持します。

表 10.5 **整数型とその値の範囲**

	型	値の範囲	データサイズ
1	char	－128～127	1 バイト
2	signed char	－128～127	1 バイト
3	unsigned char	0～255	1 バイト
4	short	－32768～32767	2 バイト
5	unsigned short	0～65535	2 バイト
6	int	－32768～32767	2 バイト
7	unsigned int	0～65535	2 バイト
8	long	－2147483648～2147483647	4 バイト
9	unsigned long	0～4294967295	4 バイト

(5) **浮動小数点**

表 10.6 **浮動小数点の仕様**

	項　目	コンパイラの仕様
1	浮動小数点型の表現方法とその値	浮動小数点型には、float 型、double 型と long double 型があります。浮動小数点型の内部表現や変換仕様、演算仕様等の性質は「10.1.3　浮動小数点数の仕様」で説明します。表 10.7 に、浮動小数点型の表現可能な値の限界値を示します。
2	整数を本来の値に正確に表現することができない浮動小数点数に変換したときの切り捨て方向	
3	浮動小数点数をより狭い浮動小数点数に変換したときの切り捨てまたは丸め方法	

表 10.7 **浮動小数点数の限界値**

	項　目	限界値	
		10 進数表現 *1	16 進数表現
1	float 型の最大値	3.4028235677973364e＋38f (3.4028234663852886e＋38f)	7f7fffff
2	float 型の正の最小値	7.0064923216240862e－46f (1.4012984643248171e－45f)	00000001
3	double, long double } 型の最大値	1.7976931348623158e＋308 (1.7976931348623157e＋308)	7fefffffffffffff
4	double, long double } 型の正の最小値	4.9406564584124655e－324 (4.9406564584124654e－324)	0000000000000001

【注】 *1　10 進数表現の限界値は 0 または無限大にならない限界値です。また、（）内は理論値を示します。

(6) 配列とポインタ

表 10.8 配列とポインタの仕様

	項 目	コンパイラの仕様
1	配列の大きさの最大値を保持するために必要な整数の型（size_t）	unsigned int 型(H8/300)
		unsigned int 型（ノーマルモード、H8SX ミドルモード、H8SX アドバンストモードかつ ptr16 オプション有り、H8SX マキシマムモードかつ ptr16 オプション有り）
		unsigned long 型（H8S/2600,H8S/2000,H8/300H アドバンストモード、H8SX アドバンストモードかつ ptr16 オプション無し、H8SX マキシマムモードかつ ptr16 オプション無し）
2	ポインタ型から整数型への変換（ポインタ型のサイズ≧整数型のサイズ）	ポインタ型の下位バイトの値になります。
3	ポインタ型から整数型への変換（ポインタ型のサイズ＜整数型のサイズ）	0 拡張します。
4	整数型からポインタ型への変換（整数型のサイズ≧ポインタ型のサイズ）	整数型の下位バイトの値となります。
5	整数型からポインタ型への変換（整数型のサイズ＜ポインタ型のサイズ）	0 拡張します。
6	同じ配列内のメンバのポインタ間の差を保持するために必要な整数の型（ptrdiff_t）	int 型（H8/300）
		int 型（ノーマルモード、H8SX ミドルモード、H8SX アドバンストモードかつ ptr16 オプション有り、H8SX マキシマムモードかつ ptr16 オプション有り）
		long 型（H8S/2600,H8S/2000,H8/300H アドバンストモード、H8SX アドバンストモードかつ ptr16 オプション無し、H8SX マキシマムモードかつ ptr16 オプション無し）

(7) レジスタ

表 10.9 レジスタの仕様

	項 目	コンパイラの仕様	
1	レジスタ変数*5 を割り付けることができるレジスタ	H8/300	最適化あり（R3）*1、R4、R5 、R6 最適化なし（R3）*1、R4、R5
		上記以外	最適化あり（ER3）*1、ER4、ER5 、ER6 最適化なし（ER3）*1、ER4、ER5 、ER6*4
2	レジスタに割り付けることができるレジスタ変数*5 の型	char、unsigned char、short、 unsigned short、int、unsigned int、long*2、unsigned long*2、float*2、ポインタ リファレンス、データメンバへのポインタ、4byte 以下の構造体データ*3	

【注】 *1 （）内のレジスタは noregexpansion オプションを指定した時は、CPU が H8SX シリーズの場合を除き、レジスタ変数を割り付けません。

*2 CPU が H8/300 シリーズの場合、レジスタに割り付けません。

*3 CPU が H8/300 シリーズの場合、2byte 以下の構造体データを割り付けることが可能です。

*4 CPU が H8SX シリーズの場合のみ最適化無しのときも ER6 にレジスタ変数を割り付けます。

*5 変数へのレジスタ割付は register 記憶クラス宣言を行っているか否かの影響を受けません。

(8) **クラス、構造体、共用体、列挙型、ビットフィールド**

表 10.10 **クラス、構造体、共用体、列挙型、ビットフィールドの仕様**

	項 目	コンパイラの仕様
1	異なる型のメンバでアクセスされる共用体型のメンバ参照	参照はできますが、値は保証しません。
2	クラスメンバの境界整合	char 型のメンバだけからなるクラスは、1 バイト整合となります。それ以外は、2 バイト整合となります。割り付け方の詳細な仕様は「10.1.2(2) 複合型、クラス型」を参照してください。
3	単なる int 型のビットフィールドの符号	signed int 型とします。
4	int 型のサイズ内のビットフィールドの割り付け順序	上位ビットから割り付けます。*1 *2
5	int 型のサイズ内にビットフィールドが割り付けられているとき、次に割り付けるビットフィールドのサイズが int 型内の残っているサイズをこえたときの割り付け方	次の int 型の領域に割り付けます。*1
6	ビットフィールドで許される型指定子	char、unsigned char、 short、unsigned short、int、unsigned int、long、unsigned long
7	列挙型の値を表現する整数型	int 型

【注】 *1 ビットフィールドの割り付け方の詳細については、「10.1.2(3) ビットフィールド」を参照してください。

*2 bit_order=right オプションを指定すると下位ビットから割り付けられます。

(9) **修飾子**

表 10.11 **修飾子の仕様**

	項 目	コンパイラの仕様
1	volatile 型データへのアクセスの種類	規定しません。

(10) **宣言**

表 10.12 **宣言の仕様**

	項 目	コンパイラの仕様
1	基本型（算術型、構造体型、共用体型）を修飾する宣言子の数	16 個まで指定できます。

基本型を修飾する型の数の数え方を、以下に例を用いて示します。

例：

(i) `int i;` `i`は`int`型（基本型）であり、基本型を修飾する型の数は`0`個です。

(ii) `char *f( );` `f`は`char`型（基本型）へのポインタ型を返す関数型であり、基本型を修飾する型の数は 2 個です。

(11) **文**

表 10.13 **文の仕様**

	項 目	コンパイラの仕様
1	一つの switch 文中で指定できる case ラベルの数	511 個まで指定できます。

(12) **プリプロセッサ**

表 10.14 **プリプロセッサの仕様**

	項 目	コンパイラの仕様
1	条件コンパイルの定数式内の単一文字の文字定数と実行環境文字集合の対応	プリプロセッサ文の文字定数と実行環境文字集合は一致します。
2	インクルードファイルの読み込み方法	「〈」、「〉」で囲まれたファイルは include オプションで指定されたパスから読み込みます（省略時は環境変数 CH38 で設定されたパス）。
3	二重引用符で囲まれたインクルードファイルのサポートの有無	サポートします。インクルードファイルを現在のディレクトリから読み込みます。現在のディレクトリになければ、本表 2 項の読み込み方法に従います。
4	ソースファイルの文字の並びの対応（マクロ展開後の文字列の空白文字）	空白文字列は、空白文字 1 文字として展開します。
5	#pragma 文の動作	「10.2.1 #pragma 拡張子、キーワード」を参照してください。
6	__DATE__、__TIME__の値	コンパイル開始時のホストマシンのタイマに基づく値が設定されます。

10.1.2 データの内部表現

本節では、データ型と、データの内部表現の対応について述べます。データの内部表現は、以下の項目から成り立っています。

- データのサイズ
  データの占有する領域のサイズです。
- データの境界調整数
  データを割り付けるアドレスに関する制約です。任意のアドレスに割り付ける1バイト境界調整と、偶数バイトに割り付ける2バイト境界調整があります。
- データの範囲
  スカラ型(C言語)，基本型(C++言語)の値のとり得る範囲を示します。
- データの割り付け例
  複合型(C言語)、クラス型(C++言語)の要素となるデータの割り付け方を示します。

(1) スカラ型(C 言語)、基本型 (C++言語)

C 言語におけるスカラ型および、C++言語における基本型の内部表現を**表** 10.15 に示します。

表 10.15 **スカラ型・基本型の内部表現**

	データ型	サイズ (byte)	境界調整数(byte)	符号の有無	データの範囲	
					最小値	最大値
1	char	1	1	有	-2^7 (-128)	2^7-1 (127)
2	signed char	1	1	有	-2^7 (-128)	2^7-1 (127)
3	unsigned char	1	1	無	0	2^8-1 (255)
4	short	2	2	有	-2^{15} (-32768)	$2^{15}-1$ (32767)
5	unsigned short	2	2	無	0	$2^{16}-1$ (65535)
6	int	2	2	有	-2^{15} (-32768)	$2^{15}-1$ (32767)
7	unsigned int	2	2	無	0	$2^{16}-1$ (65535)
8	long	4	2	有	-2^{31} (-2147483648)	$2^{31}-1$ (2147483647)
9	unsigned long	4	2	無	0	$2^{32}-1$ (4294967295)
10	enum (**値の範囲が**-128～127 **かつ** byteenum **オプション指定時**)	1	1	有	-2^7 (-128)	2^7-1 (127)
11	enum (**値の範囲が** 0～255 **かつ** byteenum **オプション指定時**)	1	1	無	0	2^8-1 (255)
12	enum (**上記以外**)	2	2	有	-2^{15} (-32768)	$2^{15}-1$ (32767)
13	bool *[1]	1	1	有	-2^7 (-128)	2^7-1 (127)
14	float	4	2	有	-∞	+∞
15	double, long double *[2]	8	2	有	-∞	+∞
16	**ポインタ** (H8SX **ノーマルモード**、H8SX **ミドルモード**、H8S/2600 **ノーマルモード**、H8S/2000 **ノーマルモード**、H8/300H **ノーマルモード**、H8/300)	2	2	無	0	$2^{16}-1$ (65535)
17	**ポインタ** *[3] (H8/300H **アドバンストモード**)	4	2	無	0	$2^{24}-1$ (16777215)

	データ型	サイズ (byte)	境界調整数(byte)	符号の有無	データの範囲	
					最小値	最大値
18	ポインタ *4 (H8SX アドバンストモード、 H8SX マキシマムモード、 H8S/2600 アドバンストモード、 H8S/2000 アドバンストモード)	4	2	無	0	2^{32}-1 (4294967295)
19	リファレンス *1 (H8SX ノーマルモード、 H8SX ミドルモード、 H8S/2600 ノーマルモード、 H8S/2000 ノーマルモード、 H8/300H ノーマルモード、 H8/300)	2	2	無	0	2^{16}-1（65535）
20	リファレンス *1 *3 (H8/300H アドバンストモード)	4	2	無	0	2^{24}-1 (16777215)
21	リファレンス *1 *4 (H8SX アドバンストモード、 H8SX マキシマムモード、 H8S/2600 アドバンストモード、 H8S/2000 アドバンストモード)	4	2	無	0	2^{32}-1 (4294967295)
22	データメンバへのポインタ *1 (H8SX ノーマルモード、 H8SX ミドルモード、 H8S/2600 ノーマルモード、 H8S/2000 ノーマルモード、 H8/300H ノーマルモード、 H8/300)	2	2	無	0	2^{16}-1（65535）
23	データメンバへのポインタ *1 *3 (H8/300H アドバンストモード)	4	2	無	0	2^{24}-1 (16777215)
24	データメンバへのポインタ *1 *4 (H8SX アドバンストモード、 H8SX マキシマムモード、 H8S/2600 アドバンストモード、 H8S/2000 アドバンストモード)	4	2	無	0	2^{32}-1 (4294967295)
25	関数メンバへのポインタ *1 *6 (H8SX ノーマルモード、 H8S/2600 ノーマルモード、 H8S/2000 ノーマルモード、 H8/300H ノーマルモード、 H8/300)	6	2	—	—	—
26	関数メンバへのポインタ *1 *6 (H8SX ミドルモード)	8	2	—	—	—
27	関数メンバへのポインタ *1 *5 *6 (H8SX アドバンストモード、 H8SX マキシマムモード、 H8S/2600 アドバンストモード、 H8S/2000 アドバンストモード、 H8/300H アドバンストモード)	10	2	—	—	—

	データ型	サイズ (byte)	境界調整数 (byte)	符号の有無	データの範囲 最小値	データの範囲 最大値
28	**仮想関数メンバへのポインタ** *1 *6 (H8SX **ノーマルモード**、H8S/2600 **ノーマルモード**、H8S/2000 **ノーマルモード**、H8/300H **ノーマルモード**、H8/300)	6	2	—	—	—
29	**仮想関数メンバへのポインタ** *1 *6 (H8SX **ミドルモード**)	8	2	—	—	—
30	**仮想関数メンバへのポインタ** *1 *5 *6 (H8SX **アドバンストモード**、H8SX **マキシマムモード**、H8S/2600 **アドバンストモード**、H8S/2000 **アドバンストモード**、H8/300H **アドバンストモード**)	10	2	—	—	—

【注】 *1 C++**コンパイル時のみ有効です。**

*2 double=float **を指定した場合、**double **型のサイズは４バイトになります。**

*3 **下位** 3 **バイトがアドレスデータで、上位** 1 **バイトは不定値です。**

*4 H8SX **のアドバンストモードとマキシマムモードでは、**ptr16 **オプションを指定した場合、または** _ _ptr16 **キーワードを指定した場合にサイズが** 2byte **となります。**

*5 H8SX **のアドバンストモードとマキシマムモードでは、**ptr16 **オプションを指定した場合にサイズが８バイトになります。**

*6 **関数メンバ・仮想関数メンバへのポインタは、以下のクラスで表現しています。**

```
class _PMF{
  public:
    size_t delta;              // オブジェクトのオフセット値
    short index;               // 対象メンバ関数が仮想関数のときの仮想関数表中での
                               // インデックス
    union{
        int (*_deffun)();      // 対象メンバ関数が非仮想関数のときの関数のアドレス
        size_t vt_offset;      // 対象メンバ関数が仮想関数のときの仮想関数表のオブジェクト
                               // 中のオフセット
    };
};
```

(2) 複合型（C 言語）、クラス型（C++言語）

本項では、C 言語における配列型、構造体型、共用体型および、C++言語におけるクラス型の内部表現について説明します。

表 10.16 に複合型、クラス型の内部表現を示します。

表 10.16 **複合型、クラス型の内部表現**

	データ型	境界調整数（byte）	サイズ（byte）	データの割り付け例	
1	**配列型**	**配列要素の境界調整数**	**配列要素の数×要素サイズ**	char a[10];	**境界調整数** 1byte **サイズ** 10byte

2	**構造体型**	**構造体メンバの境界調整数のうち最大値**	**メンバのサイズの和「(a) 構造体データの割り付け方」参照**	struct {
				char a,b;
				};
					境界調整数 1byte
					サイズ 2byte

3	**共用体型**	**共用体メンバの境界調整数のうち最大値**	**メンバのサイズの最大値「(b) 共用体データの割り付け方」参照**	union {
				char a,b;
				};
					境界調整数 1byte
					サイズ 1byte

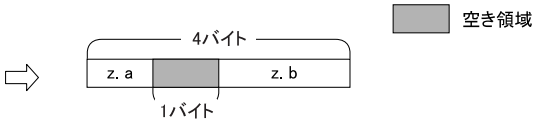
4	**クラス型**	1)**仮想関数がある場合**:
		常に 2
		
		2)**上記以外**:
		データメンバの境界調整数のうち最大値
			データメンバ、仮想関数表へのポインタ、仮想基底クラスへのポインタの和「(c) クラスデータの割り付け方」参照	(H8S/2600 **アドバンストモード時**)
				class B: public A{
				virtual void f();
				};
				class A{
				char a;
				};
					境界調整数 2byte
					サイズ 6byte
					
					境界調整数 1byte
					サイズ 1byte

(a) 構造体データの割り付け方

・ 構造体型の各メンバを割り付ける場合、そのメンバのデータ型の境界調整数に合わせるために直前のメンバとの間に1バイトの空き領域が生じる場合があります。

例：

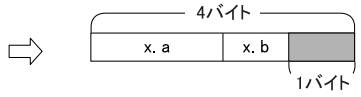
```
struct  {
     char a;
     int b;
  }z;
```



・ 構造体が2バイトの境界調整数を持ち、最後のメンバが奇数バイト目で終わっている場合、その次のバイトも含めて構造体型の領域として扱います。

例：

```
struct  {
     int a;
     char b;
  }x;
```

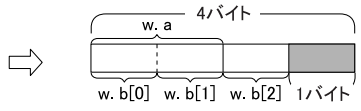


(b) 共用体データの割り付け方

・ 共用体が2バイトの境界調整数を持ち、メンバのサイズの最大値が奇数バイトの場合、その次のバイトも含めて共用体型の領域として扱います。

例：

```
union  {
     int a;
     char b[3];
  }w;
```

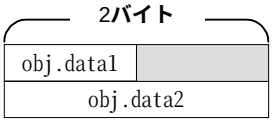


(c) **クラスデータの割り付け方**

・ 基底クラス、仮想関数がないクラスの場合、構造体データの割り付け規則に従ってデータメンバを割り付けます。

例：

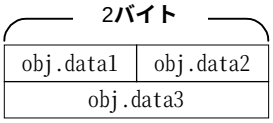
```
class A{
    char data1;
    short data2;
  public:
    A();
    int getData1(){return data1;}
}obj;
```



・ 境界調整数が１の基底クラスから派生したクラスの先頭メンバが1byteデータの場合、空き領域を作らないようにデータメンバを割り付けます。

例：

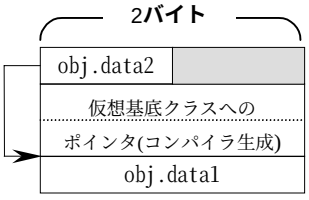
```
class A{
    char data1;
};
class B:public A{
    char data2;
   Short data3;
}obj;
```



・ クラスに仮想基底クラスがある場合、仮想基底クラスへのポインタを割り付けます。

例：

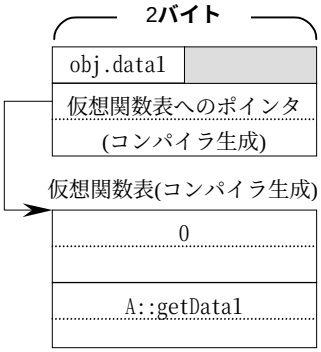
```
 class A{
     short data1;
 };
 class B: virtual protected A{
     char data2;
 }obj;
```



・ クラスに仮想関数がある場合、コンパイラは仮想関数表を生成し、仮想関数表へのポインタを割り付けます。

例：

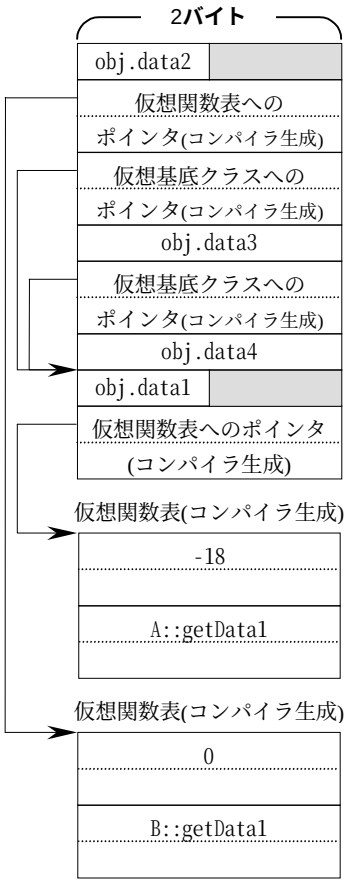
```
class A{
    char data1;
  public:
    virtual int getData1();
}obj;
```



・ 仮想基底クラス、基底クラス、仮想関数があるクラスの例を示します。

例：

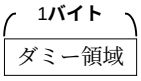
```
classA{
    char data1 ;
    virtual short get Data1();
};
class B:virtual public A{
    char data2;
    char getData2();
    short getData1();
};
class C:virtual protected A{
    int data3;
};
class D:virtual public A,public B,public C{
  public:
    int data4;
    short getData1();
}obj;
```



・ 空クラスの場合、1バイトのダミー領域を割り付けます。

例：

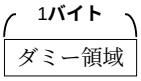
```
class A{
    void fun();
}obj;
```



・ 空クラスを基底クラスに持つ空クラスの場合でも、ダミー領域は1バイトになります。

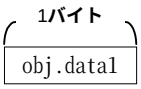
例：

```
class A{
    void fun();
};
class B: A{
    void sub();
};
```



・ 空クラスのダミー領域は、クラスサイズが０の場合に割り付けます。基底クラスや派生クラスにデータメンバがある場合や、仮想関数があるクラスの場合には、ダミー領域は割り付けません。
例：

```
class A{
    void fun();
};
class B: A{
    char data1;
}obj;
```



(3) ビットフィールド

ビットフィールドは、構造体、クラスの中にビット幅を指定して割り付けるメンバです。
本項では、ビットフィールド特有の割り付け規則について説明します。

(a) ビットフィールドのメンバ

表 10.17 にビットフィールドメンバの仕様を示します。

表 10.17 **ビットフィールドメンバの仕様**

	項目	仕様
1	**ビットフィールドで許される型指定子**	char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long
2	**宣言された型に拡張するときの符号の扱い** *1	**符号なし（**unsigned **を指定した型）→ ゼロ拡張** *2 **符号あり（**unsigned **を指定しない型）→符号拡張**

【注】 *1 **ビットフィールドのメンバを使用する場合は、ビットフィールドに格納したデータを、宣言した型に拡張して使用します。**

*2 **ゼロ拡張： 拡張するときに上位のビットにゼロを補います。**
符号拡張： 拡張するときにビットフィールドデータの最上位ビットを符号として解釈し、上位のビットに符号ビットを補います。

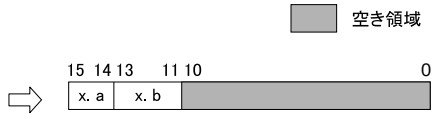
【注】 符号付き（signed**）で宣言されたサイズが** 1 **ビットのビットフィールドのデータは、データそのものを符号として解釈します。したがって、表現できる値は** 0 **と**−1 **だけになります。** 0 **と** 1 **を表現する場合には、必ず符号なし（**unsigned**）で宣言してください。**

(b) ビットフィールドの割り付け方

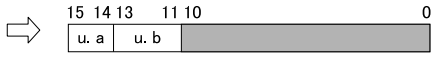
ビットフィールドは、以下の５つの規則に従って割り付けます。

・ ビットフィールドのメンバは領域内で左（上位ビット側）から順に詰め込みます。
例：

```
struct b1{
          int a:2;
          int b:3;
     }x;
```

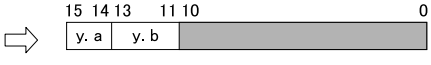



```
struct b1{
      enum E1{o,p,q} a:2;
      enum E1        b:3;
     }u;
```



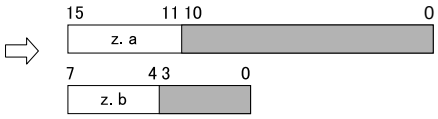
・ 同じサイズの型指定子が連続している場合は、可能な限り同じ領域に詰め込みます。
例：

```
struct b1{
         int            a:2;
         unsigned short b:3;
     }y;
```



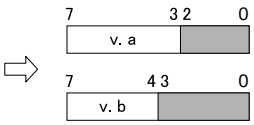
・ 異なるサイズの型指定子で宣言されたメンバは、次の領域に割り付けます。
例：

```
struct b1{
         int  a:5;
         char b:4;
    }z;
```



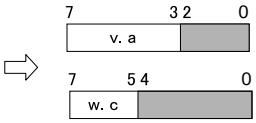
・ 同じサイズの型指定子が連続していても、詰め込み先の領域の残りビットが、次のビットフィールドのサイズより小さい場合は、残りの領域は未使用領域となり、次の領域に割り付けます。
例：

```
struct b2{
      char a:5;
      char b:4;
  }v;
```



・ ビット幅0のビットフィールドのメンバを指定すると、次のメンバからは、強制的に次の領域に割り付けます。
例：

```
struct b2{
      char a:5;
      char  :0;
      char c:3;
   }w;
```



【備考】ビットフィールドメンバを下位ビット側から詰め込むことが可能です。詳細は、「2.2 **オプション解説」の** bit_order **オプション、もしくは「**10.2.1 #pragma **拡張子、キーワード」の**#pragma bit_order **を参照してください。**

10.1.3 浮動小数点数の仕様

(1) 浮動小数点数の内部表現

コンパイラで扱う浮動小数点数の内部表現は、IEEE の標準形式に従っています。ここでは、IEEE 形式の浮動小数点数の内部表現の概要について述べます。

(a) 内部表現の形式

float 型は IEEE の単精度形式（32 ビット）、double 型と long double 型は IEEE の倍精度形式（64 ビット）で表現します。

(b) 内部表現の構成

float 型および double 型と long double 型の内部表現の構成を**図** 10.1 に示します。

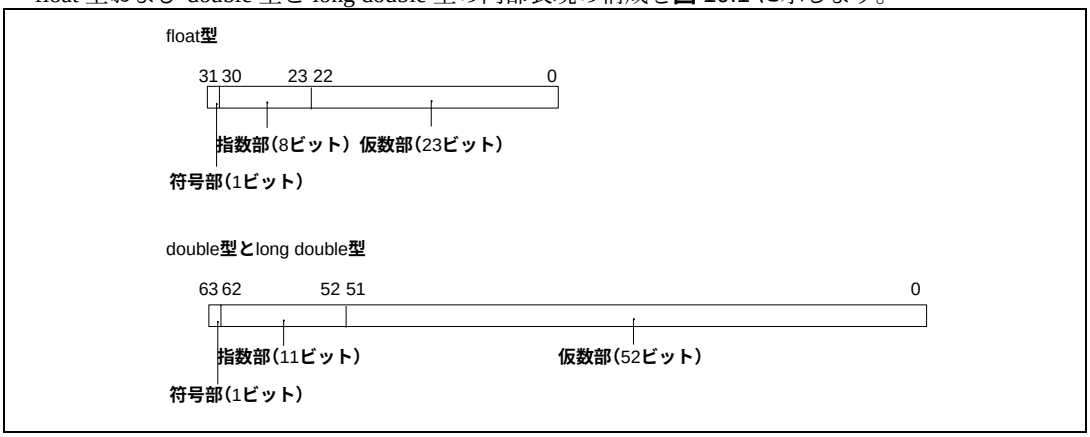


図 10.1 **浮動小数点数の内部表現の構成**

内部表現の各構成要素の意味を以下に示します。

（i） 符号部
浮動小数点数の符号を示します。0のとき正、1のとき負を示します。

（ii） 指数部
浮動小数点数の指数を2のべき乗で示します。

（iii） 仮数部
浮動小数点数の有効数字に対応するデータです。

(c) 表現する値の種類

浮動小数点数は、通常の実数値のほかに、無限大等の値も表現することができます。浮動小数点数が表現する値の種類を以下に示します。

（i） 正規化数
指数部が0または全ビット1ではない場合です。通常の実数値を表現します。

（ii） 非正規化数
指数部が0で、仮数部が0でない場合です。絶対値の小さな実数値を表現します。

（iii） ゼロ
指数部および仮数部が0の場合です。値0.0を表現します。

（iv） 無限大
指数部が全ビット1で仮数部が0の場合です。無限大を表現します。

（v） 非数
指数部が全ビット1で仮数部が0でない場合です。「0.0/0.0」、「∞／∞」、「∞－∞」等、結果が数値に対応しない演算の結果として得られます。

浮動小数点数の表現する値を決定する条件を**表** 10.18 に示します。

表 10.18　**浮動小数点数の表現する値の種類**

仮数部	指数部		
	0	0 でも全ビット 1 でもない	全ビット 1
0	0	正規化数	無限大
0 以外	非正規化数		非数

【注】　非正規化数は、正規化数で表現できない範囲の絶対値の小さな浮動小数点数を表現しますが、正規化数に比較して有効桁数が少なくなっています。したがって、演算の結果あるいは途中結果が非正規化数となる場合、結果の有効桁数は保証しません。

(2)　float **型**

float 型の内部表現は、1 ビットの符号部、8 ビットの指数部、23 ビットの仮数部からなります。

（i）　正規化数

符号部は、0（正）または1（負）で、値の符号を示します。

指数部は、1～254（2^8-2）の値をとります。実際の指数は、この値から127を引いた値で、その範囲は－126～127です。

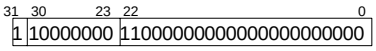
仮数部は、0～$2^{23}-1$の値をとります。実際の仮数は、2^{23}のビットを1と仮定し、その直後に小数点があるものとして解釈します。

正規化数の表現する値は、

$$(-1)^{<符号部>} \times 2^{<指数部>-127} \times (1 + <仮数部> \times 2^{-23})$$

となります。

例：



符号：　－

指数：　$10000000_{(2)}-127=1$

仮数：　$1.11_{(2)}=1.75$

値　：　$-1.75\times2^1=-3.5$

(2)は2進数を意味します。

（ii）　非正規化数

符号部は0（正）または1（負）で、値の符号を示します。

指数部は0で、実際の指数は－126になります。

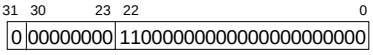
仮数部は、1～$2^{23}-1$で、実際の仮数は、2^{23}のビットを0と仮定し、その直後に小数点があるものとして解釈します。

非正規化数の表現する値は、

$$(-1)^{<符号部>} \times 2^{-126} \times (<仮数部> \times 2^{-23})$$

となります。

例：



符号：　＋

指数：　－126

仮数：　$0.11_{(2)}=0.75$

値　：　0.75×2^{-126}

(2)は2進数を意味します。

（iii） ゼロ
符号部は0（正）または1（負）で、それぞれ＋0.0、－0.0を示します。
指数部、仮数部はともに0です。
＋0.0、－0.0は、ともに値としては0.0を示します。ゼロの符号による、各演算での機能の違いについては**「10.1.3(4) 浮動小数点演算の仕様」**を参照してください。

（iv） 無限大
符号部は0（正）または1（負）で、それぞれ＋∞、－∞を示します。
指数部は255（2^8-1）です。
仮数部は0です。

（v） 非数
指数部は255（2^8-1）です。
仮数部は0以外の値です。

【注】 非数の符号、および仮数部の（0 以外の）値については規定していません。

(3) double 型と long double 型

double 型と long double 型の内部表現は、1 ビットの符号部、11 ビットの指数部、52 ビットの仮数部からなります。

（i） 正規化数
符号部は0（正）または1（負）で、値の符号を示します。
指数部は、1～2046（$2^{11}-2$）の値をとります。実際の指数は、この値から1023を引いた値で、その範囲は－1022～1023です。
仮数部は、0～$2^{52}-1$の値となります。実際の仮数は、2^{52}のビットを1と仮定し、その直後に小数点があるものとして解釈します。
正規化数の表現する値は、

$$(-1)^{<符号部>} \times 2^{<指数部>-1023} \times (1 + <仮数部> \times 2^{-52})$$

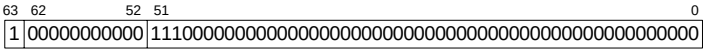
となります。
例：

63	62 … 52	51 … 0
0	01111111111	1110000000000000000000000000000000000000000000000000

符号： ＋
指数： $1111111111_{(2)}-1023=0$
仮数： $1.111_{(2)}=1.875$
値 ： $1.875\times 2^0=1.875$

$_{(2)}$は2進数を意味します。

（ii） 非正規化数
符号部は0（正）または1（負）で、値の符号を示します。
指数部は0で、実際の指数は－1022になります。
仮数部は、1～2^{52}－1で、実際の仮数は、2^{52}のビットを0と仮定し、その直後に小数点があるものとして解釈します。
非正規化数が表現する値は、
（－1）$^{<符号部>}\times 2^{-1022}\times$（＜仮数部＞$\times 2^{-52}$）
となります。
例：



符号： －
指数： －1022
仮数： $0.111_{(2)}=0.875$
値 ： 0.875×2^{-1022}

$_{(2)}$は2進数を意味します。

（iii） ゼロ
符号部は0（正）または1（負）で、それぞれ＋0.0、－0.0を示します。
指数部、仮数部はともに0です。
＋0.0、－0.0は、ともに値としては0.0を示します。ゼロの符号による、各演算での機能の違いについては**「10.1.3(4) 浮動小数点演算の仕様」**を参照してください。

（iv） 無限大
符号部は0（正）または1（負）で、それぞれ＋∞、－∞を示します。
指数部は2047（2^{11}－1）です。
仮数部は0です。

（v） 非数
指数部は2047（2^{11}－1）です。
仮数部は0以外の値です。

【注】 非数の符号、および仮数部の（0 以外の）値については規定していません。

(4) 浮動小数点演算の仕様

本項では、C/C++言語の機能として表現されている浮動小数点の四則演算、およびコンパイル時やCライブラリ関数の処理で生じる浮動小数点の10進表現と内部表現の間の変換の仕様について解説します。

(a) 四則演算の仕様

（i） 結果の値の丸め方

浮動小数点の四則演算の結果の正確な値が、内部表現の仮数の有効数字をこえた場合は、以下の規則に従って丸めを行います。

［1］ 結果の値は、その値を近似する二つの浮動小数点数の内部表現のうち、近い方に向かって丸めます。

［2］ 結果の値が、その値を近似する二つの浮動小数点数のちょうど中央になる場合は、仮数の最後の桁が0となる方向に丸めます。

（ii） オーバフロー、アンダフロー、無効演算時の処理

実行時のオーバフロー、アンダフロー、無効演算に対しては、以下の処理を行います。

［1］ オーバフローの場合は、結果の符号に従って正または負の無限大になります。

［2］ アンダフローの場合は、結果の符号に従って正または負のゼロになります。

［3］ 無効演算は、符号が逆の無限大を加算した場合、符号が同じ無限大を減算した場合、ゼロと無限大を乗算した場合、ゼロをゼロで、あるいは無限大を無限大で除算した場合に生じます。

これらの場合、結果は非数になります。

［4］ 上記のいずれの場合も、エラーの発生を示す変数errnoに対応するエラーの番号を設定します。この番号については、**「12.3 Cライブラリ関数のエラーメッセージ」**を参照してください。

エラーチェックが必要な場合は、このerrnoの値によってエラーの発生を判定してください。

【注】 定数式に関しては、コンパイル時に演算を行います。この時にオーバフロー、アンダフロー、無効演算を検出した場合は、ウォーニングレベルのエラーになります。

（iii） 特殊な値（ゼロ、無限大、非数）の演算に関する注意事項

［1］ 正のゼロと負のゼロの和は正のゼロとなります。

［2］ 同符号のゼロの差は正のゼロになります。

［3］ 被演算子の一方あるいは両方に非数を含む演算の結果は、常に非数になります。

［4］ 比較演算においては、正のゼロと負のゼロは等しいものとして扱います。

［5］ 被演算子の一方あるいは両方が非数であるような比較演算、等値演算の結果は、「！＝」については常に真、その他は常に偽となります。

(b) 10 進表現と内部表現の間の変換

本項ではソースプログラム上の浮動小数点定数と内部表現の間の変換、あるいは C ライブラリ関数による ASCII 文字列による浮動小数点数の 10 進表現と、内部表現の間の変換の仕様について解説します。

（i） 10進表現から内部表現に変換する場合、まず10進表現を10進表現の正規形に変換します。
10進表現の正規形は、「$\pm M\times10^{\pm N}$」の形式で、M、Nの範囲は以下のとおりです。

［1］ float型の正規形
$0 \leqq M \leqq 10^{9}-1$
$0 \leqq N \leqq 99$

［2］ double型とlong double型の正規形
$0 \leqq M \leqq 10^{17}-1$
$0 \leqq N \leqq 999$

正規形に変換できない10進表現については、オーバフロー、またはアンダフローになります。また、10進表現が正規形よりも多くの有効数字を含んでいる場合は、下位の桁は切り捨てます。これらの場合、コンパイル時にはウォーニングレベルのエラーになり、実行時には対応するエラーの番号を変数errnoに設定します。
また、正規形に変換するためには、もとの10進表現のASCII文字列としての長さが511文字以下でなければなりません。そうでない場合、コンパイル時にはエラーになり、実行時には対応するエラーの番号を変数errnoに設定します。
内部表現から10進表現に変換する場合には、一度10進表現の正規形に変換してから、指定した書式に従ってASCII文字列に変換します。

（ii） 10進表現の正規形と内部表現の間の変換
10進表現の正規形と内部表現の間の変換は、指数が大きいときや小さいときには、正確な変換ができません。以下に、正確な変換ができる範囲と、その範囲外の場合の誤差の限界値について解説します。

［1］ 正確な変換ができる範囲
以下に示す指数の範囲の浮動小数点数については、**「(a)(i) 結果の値の丸め方」**に示す丸めが正確に行なわれます。この範囲ではオーバフロー、アンダフローは生じません。
float型の場合： $0 \leqq M \leqq 10^{9}-1$、$0 \leqq N \leqq 13$
double型とlong double型の場合： $0 \leqq M \leqq 10^{17}-1$、$0 \leqq N \leqq 27$

［2］ 誤差の限界値
［1］で示す範囲に入っていない値を変換する場合の誤差と、正確な丸めを行なったときの誤差の差は、有効数字の最小位桁の0.47倍をこえません。
また、［1］で示した範囲をこえている場合、変換の際にオーバフローやアンダフローが生じる場合があります。この場合、コンパイル時にはウォーニングレベルのエラーになり、実行時には対応するエラーの番号を変数 errno に設定します。

10.1.4 演算子の評価順序

式の中に複数の演算子がある場合、それらの演算子の評価順序は、優先順位と呼ばれる正数と「右」または「左」で表わされる結合性と呼ばれる性質によって決まります。
各演算子の優先順位と結合性を**表**10.19に示します。

表 10.19 **演算子の優先順位と結合性**

優先順位	演算子	結合性	適用される式
1	( ) [ ] -> . ++ -- (後置)	左	後置式
2	! ~ ++ -- (前置) + - * & sizeof	右	単項式
3	(型名)	右	キャスト式
4	* / %	左	乗法式
5	+ -	左	加法式
6	<< >>	左	シフト式
7	< <= > >=	左	関係式
8	== !=	左	等価式
9	&	左	ビット毎の AND 式
10	^	左	ビット毎の XOR 式
11	|	左	ビット毎の OR 式
12	&&	左	論理 AND 演算
13	||	左	論理 OR 式
14	?:	左	条件式
15	= += == *= /= %= <<= >>= &= |= ^=	右	代入式
16	,	左	コンマ式

10.2 拡張機能

コンパイラの拡張機能として、以下の機能をサポートしています。

- #pragma 拡張子、キーワード
- セクションアドレス演算子
- 組み込み関数

10.2.1 #pragma 拡張子、キーワード

#pragma 拡張子、キーワードの一覧を示します。

表 10.20 メモリ配置に関する拡張機能

	#pragma 拡張子	キーワード	機能
1	#pragma stacksize	−	スタックセクションの作成
2	#pragma section、 #pragma abs8 section、 #pragma abs16 section、 #pragma indirect section	−	セクションの切り替え指定
3	#pragma abs8、 #pragma abs16	__abs8 __abs16	短絶対アドレス形式でアクセスする変数の指定
4	−	__near8 __near16	配列・構造体のアドレス計算サイズ指定
5	−	__ptr16	ポインタサイズ指定
6	#pragma bit_order	−	ビットフィールド並び順指定

表 10.21 関数に関する拡張機能

	#pragma 拡張子	キーワード	機能
1	#pragma interrupt	__interrupt	割り込み関数の作成
2	#pragma entry	__entry	エントリ関数の作成
3	#pragma indirect	__indirect	メモリ間接で関数呼び出しを行う関数の指定
4	−	__indirect_ex	拡張メモリ間接で関数呼び出しを行う関数の指定
5	#pragma inline	__inline	関数のインライン展開を指定
6	#pragma inline_asm	−	アセンブリ記述関数のインライン展開
7	#pragma regsave、 #pragma noregsave	__regsave __noregsave	レジスタの退避／回復コード出力の制御
8	−	__regparam2 __regparam3	引数用レジスタ数を指定
9	#pragma option	−	最適化オプションを関数単位で指定

表 10.22 その他の拡張機能

	#pragma 拡張子	キーワード	機能
1	#pragma asm、 #pragma endasm	−	アセンブラ埋め込み機能
2	−	__asm	埋め込みアセンブル機能
3	#pragma global_register	__global_register	グローバル変数のレジスタを割り付け
4	#pragma pack 1、 #pragma pack 2、 #pragma unpack	−	構造体・共用体・クラスの境界調整数を指定
5	−	__evenaccess	偶数バイトアクセス指定

【注】キーワード指定／pragma指定は、同一関数,変数に対して最初に指定されたものが有効です。
一度属性を指定した後に、さらに別属性を指定することはできません。

・エラーとなる例

```
// プロトタイプ宣言と定義で別キーワードは指定できません
_ _regsave void func(void);
_ _interrupt void func(void) {}

// pragma でも同様に別属性の指定はできません
#pragma regsave func
_ _interrupt void func(void) {}
```

同一関数,変数に対して複数の属性を指定したい場合には、宣言・定義の段階でキーワードでの指定をまとめて行ってください。

・正しくコンパイルできる例

```
// 宣言(もしくは定義)でキーワードを一度に指定することは可能です
_ _regsave  _ _interrupt void func(void);
void func(void) {}
```

(1) メモリ配置に関する拡張機能

スタックセクションの作成

#pragma stacksize

書　式　　#pragma stacksize <定数>

説　明　　セクション名 S、サイズ<定数>のスタックセクションを作成します。

例　　#pragma stacksize 100

```
<コード展開例>
.SECTION  S, STACK
.RES.W    50
```

備　考　　・サイズとして指定する<定数>は必ず偶数を指定してください。
・#pragma stacksize はファイル内で一回しか指定できません。

セクションの切り替え指定

#pragma section
#pragma abs8 section
#pragma abs16 section
#pragma indirect section

書　式　　#pragma section [{<名前> | <数値>}]
#pragma abs8 section [{<名前> | <数値>}]
#pragma abs16 section [{<名前> | <数値>}]
#pragma indirect section [{<名前> | <数値>}]

説　明　　コンパイラの出力するセクション名を切り替えます。
デフォルト、切り替え後のセクション名は**表** 10.23 の通りです。

表 10.23 **セクション切り替え機能とセクション名**

	対象領域		指定方法	デフォルト名	切り替え後
1	**プログラム領域**		#pragma section <xx>	P *[1]	P<xx>
2	**定数領域**			C *[1]	C<xx>
3	**初期化データ領域**			D *[1]	D<xx>
4	**未初期化データ領域**			B *[1]	B<xx>
5	**8 ビット絶対アドレス**	**定数領域**	#pragma abs8 section <xx>	$ABS8C	$ABS8C<xx>
6		**初期化データ領域**		$ABS8D	$ABS8D<xx>
7		**未初期化データ領域**		$ABS8B	$ABS8B<xx>
8	**16 ビット絶対アドレス**	**定数領域**	#pragma abs16 section <xx>	$ABS16C	$ABS16C<xx>
9		**初期化データ領域**		$ABS16D	$ABS16D<xx>
10		**未初期化データ領域**		$ABS16B	$ABS16B<xx>
11	**メモリ 間接**	**関数アドレス領域**	#pragma indirect section <xx>	$INDIRECT $EXINDIRECT	$INDIRECT<xx> $EXINDIRECT<xx>

【注】*1　section **オプションでデフォルトセクション名を変更できます。**
<名前>や<数値>を省略すると、以降はデフォルトのセクション名に戻ります。

例
```
#pragma section abc
int a;                      /*   a は,セクション Babc に割り付きます      */
const int c=1;              /*   c は,セクション Cabc に割り付きます      */
void f(void)                /*   f は,セクション Pabc に割り付きます      */
{
   a=c;
}
#pragma section
int b;                      /*   b は,セクション B に割り付きます         */
void g(void)                /*   g は,セクション P に割り付きます         */
{
  b=c;
}
```

備　考　・#pragma section、#pragma abs8 section、#pragma abs16 section、#pragma indirect section は関数定義の外で宣言しなければなりません。
・1 ファイルで宣言できるセクション名はそれぞれ最大 64 個です。

短絶対アドレスでアクセスする変数の指定

#pragma abs8
#pragma abs16
__abs8
__abs16

書　式

```
#pragma abs8 (<変数名> [,…] )
#pragma abs16 (<変数名> [,…] )
__abs8 <型指定子> <変数名>
<型指定子> __abs8 <変数名>
__abs16 <型指定子> <変数名>
<型指定子> __abs16 <変数名>
```

説　明

8/16 ビット絶対アドレス領域に割り付ける変数を宣言します。

・#pragma abs8 及び__abs8 で宣言された変数は、セクション名“ $ABS8C ”、“ $ABS8D ”、“ $ABS8B ”に出力され、8 ビット絶対アドレス（@aa：8）でアクセスするコードを生成します。

・#pragma abs16 及び__abs16 で宣言された変数は、セクション名“ $ABS16C ”、“ $ABS16D ”、“ $ABS16B ”に出力され、16 ビット絶対アドレス（@aa：16）でアクセスするコードを生成します。

・セクション名の切り替え方法については、前項目の#pragma abs8 section および #pragma abs16 section を参照してください。

例

```
#pragma abs8(c1)
#pragma abs16(i1)
char c1;                    /* c1 はセクション名$ABS8B に割り付きます    */
int i1;                     /* i1 はセクション名$ABS16B に割り付きます   */
char __abs8 c2;             /* c2 はセクション名$ABS8B に割り付きます    */
char __abs16 i2;            /* i2 はセクション名$ABS16B に割り付きます   */
long l;                     /* l はセクション名 B に割り付きます         */
void f(void){
  c1=c2=10;         /* c1,c2 を 8 ビット絶対アドレスでアクセスします       */
  i1=i2=100;        /* i1,i2 を 16 ビット絶対アドレスでアクセスします      */
  l=1000;           /* l を 32 ビット絶対アドレスでアクセスします          */
}
```

備　考

・#pragma abs8、#pragma abs16 宣言後の変数定義・変数宣言が対象になります。

・#pragma abs8、__abs8、#pragma abs16、__abs16 で宣言できる変数は、静的領域へ割り付ける変数のみです。

・#pragma abs8、#pragma abs16 文 1 行に宣言できる変数の数は 63 個までです。

・#pragma abs8、__abs8、#pragma abs16、__abs16 で宣言した変数は、#pragma abs8 section <XX> や#pragma abs16 section <XX>を使用しない場合、セクション名“ $ABS8C”、“ $ABS8D”、“ $ABS8B”、“ $ABS16C”、“ $ABS16D” または“ $ABS16B”へ出力されます。リンク時には、当該セクションを必ず 8 ビット／16 ビット絶対アドレス領域に割り付けてください。

・#pragma abs8 で宣言した変数が 1 バイトアクセス対象でない場合は、エラーになります。境界調整数 1 の変数、配列、構造体が対象になります。

配列・構造体のアドレス計算サイズ指定

__near8
__near16

書　式　<型指定子> __near8 <変数名>
__near8 <型指定子> <変数名>
<型指定子> __near16 <変数名>
__near16 <型指定子> <変数名>

説　明　8 または 16 ビットでアドレス計算可能な配列・構造体を指定します。
__near8 を指定した場合、配列・構造体を下位 1 バイトでアドレス計算を行います。
また__near16 を指定した場合、下位 2 バイトでアドレス計算を行います。

例　__near8 未指定時

```
struct a{
    short a1;
    short a2,a3;
};
struct a aa[10];
void f(){
   int i;
   for(i=0;i<11;i++)
      aa[i].a1 = 0;
}
```

<コード展開例>

```
   MOV.L    #_aa,ER1
   SUB.L    ER0,ER0
Ld:
   MOV.W    R0,@ER1
   INC.W    #H'1,E0
   ADDS.L   #H'4,ER1
   INC.L    #H'2,ER1
   CMP.W    #H'B,E0
   BLT      Ld:8
   RTS
```

__near8 指定時

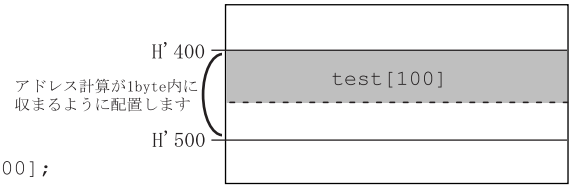
```
struct a{
    short a1;
    short a2,a3;
};
struct a __near8 aa[10];
void f(){
   int i;
   for(i=0;i<11;i++)
      aa[i].a1 = 0;
}
```

<コード展開例>

```
   MOV.L    #_aa,ER1
   SUB.L    ER0,ER0
Ld:
   MOV.W    R0,@ER1
   INC.W    #H'1,E0
   ADD.B    #H'6,R1L
   CMP.W    #H'B,E0
   BLT      Ld:8
   RTS
```

備　考
- __near8、__near16 を指定した配列・構造体は、それぞれ 8 ビット、16 ビットアドレス計算がオーバフローしない位置に配置してください。
- __near8、__near16 を指定した配列・構造体が正しい位置に配置されない場合、リンク時にエラー出力されます。
- 8 ビット、16 ビットのアドレスの境界値を越えて変数が配置された場合、実行時の動作は保証しません。__near8、__near16 指定をはずしてください。

```
struct b{
   char buffer1;
   char buffer2;
};
struct b __near8 test[100];
```



ポインタサイズ 2byte 指定

_ _ptr16

書　式　　<型指定子> _ _ptr16 <*>

説　明　　ポインタのサイズを 2byte に指定します。
符号有りの 2byte によりポインタ値を指定しますので、アクセスする先が 16 ビット絶対アドレス領域に割りついている必要があります。

例　　_ _ptr16 不指定時

```
_ _abs16 int a;
int *b;

func()
{
  b = &a;
}
```

<コード展開例>

```
_func:
  mov.l   #_a,er0
  mov.l   er0,@_b:32
```

_ _ptr16 指定時

```
_ _abs16 int a;
int _ _ptr16 *b;

func()
{
  b=(int _ _ptr16 *)&a;
}
```

<コード展開例>

```
_func:
  mov.l     #_a,er0
  mov.w     r0,@_b:32
```

備　考　　本キーワードを指定する場合は、単項演算子*の前に指定してください。
本キーワードは、H8SX アドバンストモードおよび H8SX マキシマムモードでのみ有効です。

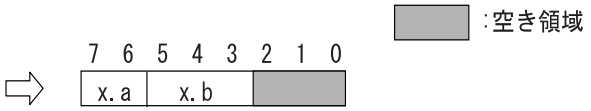
#pragma bit_order

書　式　　`#pragma bit_order  [{left | right}]`

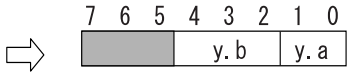
説　明　　ビットフィールドの並び順の切り替えを指定します。
`left` を指定した場合は最上位ビット側から、`right` を指定した場合は最下位ビット側から、それぞれメンバが割り付けられます。
デフォルトの設定は `bit_order` オプションの解釈に従います。
`left` または `right` を省略して`#pragma bit_order` すると、それの行より先は `bit_order` オプションの解釈に従います。

例

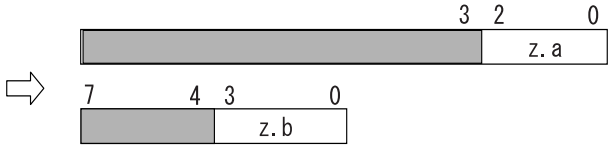
```
#pragma bit_order left
typedef struct{
  unsigned char a:2;
  unsigned char b:3;
}x;
```



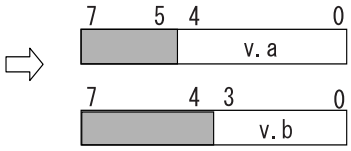

```
#pragma bit_order right
typedef struct{
  unsigned char a:2;
  unsigned char b:3;
}y;
```




```
// 異なるサイズのメンバの場合
#pragma bit_order right
typedef struct{
  unsigned int  a:3;
  unsigned char b:4;
}z;
```




```
// 型のサイズを超える場合
#pragma bit_order right
typedef struct{
  unsigned char a:5;
  unsigned char b:4;
}v;
```



備　考　　並び順の切り替えをしない限り、指定したビットフィールドの並び順は有効です。
ビットフィールドの並び順はコンパイラオプションからも指定可能です。詳細については**「**2.2.2 **オブジェクトオプション」**を参照してください。
ビットフィールドの詳細については**「**10.1.2(3) **ビットフィールド」**を参照してください。

(2) **関数に関する拡張機能**

割り込み関数の作成

#pragma interrupt
_ _interrupt

書　式　`#pragma interrupt (<関数名>[(割り込み仕様)][, ...])`
`_ _interrupt[(割り込み仕様)] <型指定子> <関数名>`
`<型指定子> _ _interrupt[(割り込み仕様)] <関数名>`

説　明　`#pragma interrupt` を用いて割り込み関数となる関数を宣言します。
割り込み仕様の一覧を**表** 10.24 に示します。

表 10.24　**割り込み仕様の一覧**

	項目	形式	オプション	指定内容
1	**スタック切り替え指定**	sp=	{ <**変数**> | &<**変数**> | <**定数**> | <**変数**>+<**定数**> | &<**変数**>+<**定数**> }	**新しいスタックのアドレスを変数または定数で指定** <**変数**>：**変数（ポインタ型）** &<**変数**>：**変数（オブジェクト型）のアドレス** <**定数**> ：**定数値**
2	**トラップ命令リターン指定**	tn=	<**定数**>	**終了を** TRAPA **命令で指定** **定数値（トラップベクタ番号）**
3	**割り込み関数終了指定**	sy=	{ <**関数名**> | <**定数**> | $<**関数名**> }	**終了を割り込み関数へのジャンプ命令で指定** <**関数名**> ：**割り込み関数名** <**定数**> ：**絶対アドレス** $<**関数名**>：**下線（ _ ）を付加しない割り込み関数名**
4	**ベクタテーブル指定**	vect=	<**ベクタ番号**>	**割り込み関数のアドレスを配置するベクタ番号**

・`#pragma interrupt` を用いて宣言した関数は、関数の処理の前後で使用している時は H8/300 で R0、R1、(R2 `regparam=3` の時)、または、その他の CPU で ER0、ER1、(ER2 `regparam=3` の時)を含むレジスタを保証し、RTE 命令でリターンします。
・トラップ命令リターン指定（`tn=`）をした場合は TRAPA 命令でリターンします。
例：

```
extern char STK[100];
#pragma interrupt ( f(sp=STK+100, tn=2) )
                          *1        *2
_ _interrupt(sp=STK+100, tn=2) void g(void);
              *1        *2
```

*1　STK+100 を割り込み関数「f」および「g」で使用するスタックポインタとして設定します。
*2　割り込み関数終了時に TRAPA #2 でトラップ例外処理を開始します。トラップ例外処理開始時の SP は**図** 10.2 のようになっています。トラップルーチンの側で RTE 命令を使用して PC、CCR（コンディションコードレジスタ）、EXR（エクステンドレジスタ：H8SX,H8S/2600,H8S/2000 で製品がサポートしてる時のみ）割込み関数から復帰してください。

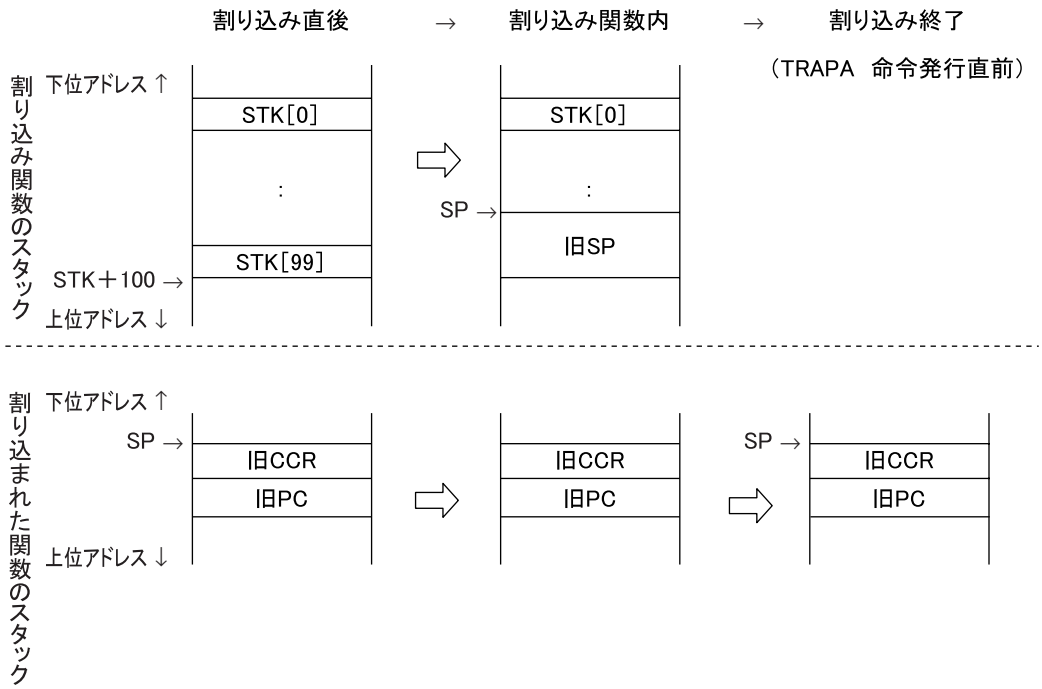


図 10.2　**割り込み関数によるスタック使用例**

・割り込み関数終了指定（sy＝）をした場合は JMP 命令で指定されたアドレスへジャンプ します。割り込み関数終了指定の関数名には“ 関数名 ”のみの指定以外に、“ $関数名 ”の指定が可能です。“ $関数名 ”指定の時は、外部名となる関数名の先頭に下線（ _ ）が付加されません。
例：

```
#pragma interrupt (f1(sy=$f2))    /* 下線(_)が付加されません        */
void f2(void)                     /* JMP @f2:24 でリターンします    */
{
    :
}
```

・ベクタテーブル指定（vect＝）をした場合は指定したベクタテーブル番号へその関数アドレスを割り付けます。
例：(cpu=300 の時)

```
#pragma interrupt (f2(vect=4))    /* 関数 f2 のアドレスを 8 番地      */
void f1(void)                     /* (ベクタ番号 4)へ割り付けます    */
{
    :
}
```

・割り込み仕様を指定しない場合は単純な割り込み関数として処理します。

備　考　・宣言後の関数定義または関数宣言が対象になります。

```
#pragma interrupt (A::f)/* #pragma interrupt 宣言後の          */
                         /* 関数定義、宣言が対象になります       */
class A{
public:
  static void f(void);  /* 静的メンバ関数を割り込み関数として扱います */
};
void A::f(void)
{
  …
}
```

・割り込み関数として定義できる関数は、グローバル関数と静的メンバ関数だけです。また、関数の返すデータ型は void のみです。return 文のリターン値を指定することはできません。指定があった場合はエラーを出力します。

```
#pragma interrupt(f1(sp=100),f2)
void f1(void)           /*   正しい宣言です                    */
{
  …
}
int f2(void)            /* 関数の返す型が void 以外だとエラーになります*/
{
  …
}
```

・割り込み関数として宣言した関数をプログラムの中で呼び出すことはできません。呼び出しがあった場合はエラーを出力します。ただし、割り込み関数として定義した関数を、割り込み関数の宣言のないプログラム内で呼び出した場合は、エラーは出力しません。この場合実行時の動作は保証しません。

```
#pragma interrupt(f1)
void f1(void)
{
  …
}
int f2(void)            /* 関数 f1 は割り込み関数なので、エラーになります */
{
  f1();
}
```

・割り込み関数として宣言した関数に対して、明示的な関数呼び出しによる参照を除いて関数の参照をすることができます。

```
#pragma interrupt f
void f(void)
{
  …
}
void (*VTBL)(void)={f};
                /*   関数呼び出し以外の参照は正常にコンパイルできます   */
```

・#pragma interrupt 文 1 行に宣言できる関数の数は 63 個までです。スタック切り替え指定とトラップ命令リターン指定、およびスタック切り替え指定と割り込み関数終了指定は重複して指定できます。割り込み関数にスタック切り替え指定を指定した場合、コンパイルリストのシンボル割り付け情報の Linkage Area Size には、旧 SP と SP 計算のための ER0（H8/300 時は R0）の退避領域のサイズも含まれます。

#pragma entry
_ _entry

書　式　#pragma entry <関数名>[(<entry 仕様>)]
_ _entry[(<entry 仕様>)] <型指定子> <関数名>
<型指定子>_ _entry[(<entry 仕様>)] <関数名>
<entry 仕様>：{sp=<定数> | vect=<ベクタ番号>}

説　明　<関数名>で指定した関数をエントリ関数として扱います。

・sp を指定した場合、エントリ関数の入り口でスタックポインタの初期設定コードを出力します。スタックポインタの初期値として sp で指定した<定数>を使用します。

例：cpu=300 の時

```
#pragma entry INIT(sp=0x8000)
void INIT()
    :
}
```

<コード展開例>

```
.SECTION P,CODE
_INIT:
 MOV.W      #H'8000,SP
        :
```

・sp 指定がない場合、#pragma stacksize を用いて作成したスタックセクションのセクション終了アドレスをスタックポインタ初期値として使用します。

例：cpu=300 の時

```
#pragma stacksize 100
#pragma entry INIT
void INIT()
{
    :
}
```

<コード展開例>

```
.SECTION S,STACK
.RES.W   50
.SECTION P,CODE
_INIT:
 MOV.W    #STARTOF S+SIZEOF S,SP
                :
```

・sp 指定、#pragma stacksize 宣言のどちらもない場合、サイズ 0 の S セクションを生成し、S セクションの最終アドレスをスタックポインタ初期値として使用します。#pragma stacksize 宣言をプログラムで宣言するか、リンク時にセクション S が正しいアドレスに割りつくよう、start オプションで指定してください。

例：cpu=300 の時

```
#pragma entry INIT
void INIT()
{
   :
}
```

<コード展開例>

```
.SECTION S,STACK
       ;サイズ 0 の S セクションを生成
.SECTION P,CODE
_INIT:
 MOV.W    #STARTOF S + SIZEOF S,SP
                :
```

・vect を指定した場合、指定したベクタ番号に対応するアドレスにその関数のアドレスを割り付けます。

例：cpu=300 の時

```
#pragma entry INIT(vect=0)  /* 関数 INIT のアドレスを 0 に割り付けます */
void INIT()
{
   :
}
```

<コード展開例>

```
.section  $VECT0,data,locate=0
.data.w   _INIT
```

・エントリ関数の入口／出口のレジスタ退避／回復コード出力を抑止します。

備　考　・#pragma entry <関数名>指定は、<関数名>の宣言前に行ってください。
・キーワードは、宣言・定義のどちらにでも指定することができますが、関数宣言に対するキーワード指定時は SP、vect の指定はできません。
・エントリ関数はロードモジュール全体で１つしか指定できません。

メモリ間接で関数呼び出しを行う関数の指定

#pragma indirect
__indirect

書　式　#pragma indirect (<関数名>[(vect=<ベクタ番号>)][,…])
<型指定子> __indirect[(vect=<ベクタ番号>)] <関数名>
__indirect[(vect=<ベクタ番号>)] <型指定子> <関数名>

説　明　#pragma indirect、__indirect を用いて、メモリ間接（@@aa：8）呼び出しとなる関数を宣言します。

・#pragma indirect、__indirect を用いて宣言された関数は「JSR　@@$関数名：8」の形で呼び出します。
また、vect が指定された場合、その関数のアドレスを指定したベクタに対応するアドレスに割り付けます。ベクタ番号はノーマルモード・H8/300 の場合 0 から 127 です。その他の場合 0 から 63 です。
vect が指定されていない場合、メモリ間接呼び出し宣言された関数定義に対して、“ $関数名 ”のラベルと関数のアドレスが、セクション名“ $INDIRECT ”にメモリ間接呼び出しのためのアドレステーブルとして確保されます。

・セクション名の切り替え方法については、**「10.2.1(1) メモリ配置に関する拡張機能」**の#pragma indirect section を参照してください。

例　(cpu=300 の時)

```
__indirect(vect=5) char f(void);    /* 関数 f のアドレスを 10 番地へ    */
char f(void)                        /* 割り付けます。                   */
{
  …
}
#pragma indirect (g)
unsigned char g(void)       /* $indirect セクションに$g を生成し、      */
{                           /* 関数 g のアドレスを格納します。          */
  …
}
void sub( )
{
  f();                      /* @@$f:8 メモリ間接で関数を呼び出します。  */
  g();                      /* @@$g:8 メモリ間接で関数を呼び出します。  */
}
```

備　考

・#pragma indirect は、宣言後の最初に出現した同名の関数定義・関数宣言を対象にします。
・#pragma indirect 文 1 行に宣言できる関数の数は 63 個までです。
・#pragma indirect、__indirect で宣言できる関数の数は、ノーマルモード・H8/300 の場合 128 個までです。その他の場合は 64 個です。
vect 指定なし時に生成されたアドレステーブルのセクション($INDIRECT)はリンク時に 0x0000～0x00FF 番地に割り付けてください。
・#include <indirect.h>を宣言することにより、実行時ルーチンの呼び出しコードをメモリ間接呼び出しにすることができます。メモリ間接呼び出しを行う実行時ルーチンを選択したい場合には、indirect.h の中で必要な#pragma indirect 文以外をコメントにしてください。

拡張メモリ間接で関数呼び出しを行う関数の指定

__indirect_ex

書　式　　<型指定子> __indirect_ex[(vect=<ベクタ番号>)] <関数名>
__indirect_ex[(vect=<ベクタ番号>)] <型指定子> <関数名>

説　明　　__indirect を用いて、拡張メモリ間接（@@vec）呼び出しとなる関数を宣言します。
__indirect_ex を用いて宣言された関数は「JSR @@$$関数名：7」の形で呼び出されます。
また、vect が指定された場合、その関数のアドレスを指定したベクタ番号に対応するアドレスに割り付けます。ベクタ番号は 128 から 255 です。
vect が指定されていない場合、拡張メモリ間接呼び出し宣言された関数定義に対して、“ $$関数名 ”のラベルと関数のアドレスが、セクション名“ $EXINDIRECT ”に拡張メモリ間接呼び出しのためのアドレステーブルとして確保されます。

例　　(CPU=h8sxa の時)

```
_ _indirect_ex(vect=128) char fl(void); /*関数 fl のアドレスを 0x200 番地
                                          へ割り付けます。/*
char fl(void)
{
}
void subl(void)
{
    fl( )； /* @@$$fl:7 拡張メモリ間接で関数を呼び出します */
}
```

備　考　　・本キーワード指定は、CPU 種別が H8SX の場合にのみ有効です。
・_ _indirect_ex で宣言できる関数の数は 128 個までです・
vect 指定なし時に生成されたアドレステーブルのセクション($EXINDIRECT)はリンク時に、H8SX ノーマルモードの場合、0x0100～0x01FF 番地、H8SX ミドルモード、H8SX アドバンストモード、H8SX マキシマムモードの場合は 0x000200～0x0003FF 番地に割り付けて下さい。

関数のインライン展開

#pragma inline
__inline

書　式　#pragma inline（<関数名>[,…]）
__inline <型指定子> <関数名>
<型指定子> __inline <関数名>

説　明　#pragma inline を用いて、インライン展開となる関数を宣言します。
#pragma inline を用いて宣言した関数を呼び出すと JSR、BSR 命令で関数を呼び出すコードは出力されず、呼び出した場所へ関数のコードが直接展開されます。

例
```
#pragma inline (f)          /* 関数 f をインライン展開として宣言します。  */
int a,b,c;
int f(int x,int y)
{
  return x+y;
}
void sub(void)
{
  a=f(b,c);                 /* 直接 a=b+c のコードに展開されます。        */
}
```

備　考
- #pragma inline 宣言後、最初に出現した同名の関数定義・関数宣言を対象にします。
- #pragma inline 文 1 行に宣言できる関数の数は 63 個までです。
- #pragma inline、__inline で宣言された関数が次のいずれかの条件を満たす時、インライン展開されません。
  - #pragma inline、__inline 指定より前に関数の定義がある。
  - 可変引数を持つ。
  - 引数のアドレスを参照している。
  - 実引数と仮引数の型が不一致である。
  - インライン展開の制限サイズを超えている。
- #pragma inline、__inline を指定した場合､外部定義を生成します。各ソースプログラムファイル中にインライン関数の実体の記述がある場合は、必ず関数の宣言に static を指定してください。static を指定した場合は、外部定義を生成しません。

アセンブラ記述関数のインライン展開

#pragma inline_asm

書　式　#pragma inline_asm (<関数名>[,…])
<関数名>:C++メンバ関数、オーバロード関数は指定不可

説　明　#pragma inline_asm で宣言したアセンブリ記述関数をインライン展開します。
アセンブラ埋め込みインライン関数のパラメタは、通常の関数呼び出しと同様にレジスタ、あるいはスタックに設定されますので、inline_asm 関数から参照することができます。アセンブラ埋め込みインライン関数のリターン値は(E)R0 に設定してください。

例

```
#pragma inline_asm(shlu)
extern unsigned int x;
static unsigned int shlu(unsigned int a)
{                                       ;関数 shlu は削除されます。
  SHLL.W    R0
  BCC       ?L1
  SUB.W     R0,R0
 ?L1:                                   ;ローカルラベルは?で始まります。
}
void main(void)
{
  x = shlu(x);                          /*main 関数内にインライン展開します。*/
}
```

備　考
- 本機能を使用する際は、オブジェクト形式指定オプション code=asmcode を用いてコンパイルしてください。
- #pragma inline_asm 宣言後に出現する関数定義を対象にします。
- #pragma inline_asm は、関数本体の定義の前に指定してください。
- #pragma inline_asm で指定した関数に対しても外部定義を生成します。各ソースプログラムファイル中にインライン関数の実体の記述がある場合は､必ず関数の宣言に static を指定してください。static を指定した場合は、外部定義を生成しません。
- アセンブラ埋め込みインライン関数内でラベルを使用する場合、必ずローカルラベルを使用してください。ローカルラベルの詳細は、**「11 アセンブラ言語仕様」**を参照してください。
- アセンブラ埋め込みインライン関数内で ER2 から ER6 のレジスタを使用する場合は、 アセンブラ埋め込みインライン関数の先頭と最後でこれらレジスタの退避／回復が必要です。
- アセンブラ埋め込みインライン関数の最後に RTS を記述しないでください。
- 本機能を使用した場合、コンパイラ出力のアセンブリプログラムに対してクロスアセンブラで“402 ILLEGAL VALUE IN OPERAND”のエラーが出ることがあります。これはアセンブラ埋め込み箇所を含んだ分岐幅を 16 ビットディスプレースメントで出力しているため、実際の分岐幅がその範囲を超えると出力されます。分岐幅が届くように JMP 命令を使用して、コンパイラ出力のアセンブリプログラムを修正してください。

例：

```
（修正前）                              （修正後）
   :                                       :
  BEQ L1              →                   BNE Ld
    :                                     JMP L1
                                        Ld:
```

#pragma regsave
#pragma noregsave
_ _regsave
_ _noregsave

書　式　#pragma regsave (<関数名>[,…])
#pragma noregsave (<関数名>[,…])
_ _regsave <型指定子> <関数名>
<型指定子> _ _regsave <関数名>
_ _noregsave <型指定子> <関数名>
<型指定子> _ _noregsave <関数名>

説　明　#pragma regsave、_ _regsave、#pragma noregsave、_ _noregsave を用いて、レジスタ退避／回復コードを制御します。

・#pragma regsave、_ _regsave で宣言された関数は、関数内でレジスタの使用／未使用にかかわらず、関数呼び出し前後で値を保証するレジスタを全て関数の入口で退避し、出口で回復するコードを生成します。また、関数呼び出しをまたいで関数呼び出し前後で値を保証するレジスタを割り付けません。
・#pragma noregsave、_ _noregsave で宣言された関数は、関数内でレジスタの使用／未使用に関わらず、レジスタの退避／回復コードを生成しません。
・#pragma noregsave、_ _noregsave で宣言された関数を呼び出す場合、関数呼び出しをまたいで値を保証するレジスタを割り付けません。

例　(CPU=2600a でコンパイル)

```
#pragma regsave (f,g)      /* レジスタ退避／回復コード生成を宣言します。 */
#pragma interrupt g        /* 関数 g は割り込み関数です。               */
void f(void){}             /* ER2～ER6 を退避／回復します。             */
void g(void){}             /* ER0～ER6 を退避／回復します。             */
```

備　考
・#pragma regsave、#pragma noregsave 宣言後に、最初に出現した関数定義、関数宣言を対象にします。
・#pragma regsave／noregsave 文 1 行に宣言できる関数の数は 63 個までです。
・_ _noregsave または#pragma noregsave 指定関数のアドレスを関数へのポインタに代入した時、関数へのポインタによる関数呼び出しは通常関数呼び出しになります。即ち、ポインタによる関数呼び出しをまたいで値を保証するレジスタに値を割り付けることがあります。そのレジスタの値が_ _noregsave または#pragma noregsave 指定関数により破壊される可能性があります。
例：

```
#pragma noregsave f
void (*p)(void);
int sub(void)
{
   int a=8;  // R4 に a を割りつけたと仮定
   p=f;
   f();      // noregsave 関数呼び出し（R4 退避後 f を呼び出してから R4 回復）
   (*p)();   // 通常関数呼び出し（R4 を退避／回復しない）
   return a;
}
```

__regparam2
__regparam3

書　式　<型指定子> __regparam2 <関数名>
<型指定子> __regparam3 <関数名>

説　明　引数用レジスタ数を指定します。
__regparam2 で指定された関数は ER0,ER1(H8/300 時は R0,R1)、__regparam3 で指定された関数は ER0,ER1,ER2(H8/300 時は R0,R1,R2)を使用します。

例
```
void __regparam2 func1(long a, int b, int c, long d);
void __regparam3 func2(long a, int b, int c, long d);
int long a; int b; int c; long d;
void main(void)
{                          /* cpu=2600a の場合                    */
        :                  /*  変数の割り付けパターン             */
   func1(a, b, c, d);      /* long a ：ER0                        */
       :                   /* int b  ：E1                         */
        :                  /* int c  ：R1                         */
       :                   /* long d ：stack                      */
       :
   func2(a, b, c, d);      /* long a ：ER0                        */
       :                   /* int b  ：E1                         */
       :                   /* int c  ：R1                         */
}                          /* long d ：ER2                        */
```

備　考　本キーワードは<型指定子>の前に指定することはできませんので、必ず関数名の前に指定してください。

オプションの関数単位指定

#pragma option

書　式　#pragma option [<オプション列>]

説　明　#pragma option を用いて、オプション列で指定したオプションを有効にします。
指定されたオプションはファイルの終わり、又は<オプション例>が無い#pragma option が設定された部分まで適用されます。
pragma option <キーワード>を指定すると、キーワードで指定された最適化を行います。
使用することが可能な最適化は、**表** 10.25 の通りです。各最適化オプションについては**「**2 C/C++**コンパイラ操作方法」**を参照してください。

表 10.25　**使用可能最適化オプション**

	オプション指定方法	オプション解除方法
1	case = { auto | ifthen | table }	**なし**
2	cmncode	nocmncode
3	cpuexpand	nocpuexpand
4	macsave	nomacsave
5	regexpansion	noregexpansion
6	optimize	nooptimize
7	speed = { speed **サブオプション** }	**なし**

また、speed サブオプションは以下のようになります。

表 10.26　**指定可能** speed **サブオプション**

	オプション指定方法	オプション解除方法
1	register	noregister
2	shift	noshift
3	loop	noloop
4	switch	noswitch
5	Inline	noinline
6	struct	nostruct
7	expression	noexpression

・<オプション例>が無い#pragma option を指定した場合は、今まで指定された#pragma option <オプション列>が無視され、コマンドラインで指定されたオプションになります。

例
```
#pragma option speed
void func(void){                        //speed オプションが有効になります。
    :
}
#pragma option cpuexpand
void test(void){                        //speed,cpuexpand が有効になります。
    :
}
#pragma option
void sub1(void){                        // コマンドライン指定に従います。
    :
}
```

備　考　H8SX では#pragma option=speed=inline=<数値>は指定できません。
また、H8SX で#pragma option=speed=inline=<数値>を指定した場合、#pragma option=speed=inline が指定されたものとみなします。

(3) その他の拡張機能

アセンブラ埋め込み機能

#pragma asm

書　式

```
#pragma asm
     <アセンブラ命令列>
#pragma endasm
```

説　明　#pragma asm～#pragma endasm で囲まれた範囲にアセンブラ命令列を記述することができます。
コンパイラは#pragma asm～#pragma endasm で囲まれた命令列をコンパイラが生成するオブジェクトコードの中に展開します。

例

```
void func(void)
{
#pragma asm
     CLRMAC                          ;MAC レジスタを０設定します。
#pragma endasm
       :
}
```

備　考
- コンパイル時に code＝asmcode オプションを用いてアセンブリプログラムの出力を指定してください。指定がない場合は#pragma asm、#pragma endasm を含むアセンブラ命令列を無視します。
- コンパイラはアセンブラ命令列の文法や、コンパイラ生成コードへの影響についてはチェックしません。また、コンパイル時に optimize＝1 オプションや speed オプションを指定した場合、アセンブラ命令列の展開内容や展開位置が実際の指定と一致しない場合があります。アセンブラ埋め込み機能を使用する場合は、出力コードおよびプログラムの動作を十分確認してください。
- #pragma asm～#pragma endasm をネストして指定することはできません。ネストがある場合、エラーを出力します。
- 選択文、繰り返し文で#pragma asm～#pragma endasm を指定する場合、#pragma asm、#pragma endasm を含むアセンブラ命令列を複文 { } で囲む必要があります。複文で囲まれていない場合、結果は保証しません。

例：

```
while(a==0)
{                ..... 必ず指定してください。
#pragma asm
  <アセンブラ命令列>
#pragma endasm
}                ..... 必ず指定してください。
```

埋め込みアセンブル機能

__asm

書　式

```
__asm{
  [<ラベル名>:]   [<アセンブラ命令>]
   ...
  }
```

説　明

__asm{ と } とで囲まれた範囲にアセンブリ言語プログラムを記述することができます。
__asm{ と } とで囲まれた範囲を_ _asm ブロックと呼びます。_ _asm ブロックの言語仕様はアセンブラと若干の相違があります。

(1) 構文

・コンパイラは_ _asm ブロックを C/C++言語の文として扱います。
_ _asm ブロックは C/C++言語の文を記述できる位置に記述することができますが、関数の外や C 言語の複文の変数宣言の前に記述できません。
・１行に１命令まで記述できます。
・一つの命令を複数行にわたり書くことはできません。
アセンブラでは決められた位置に + 印を指定すると次行に命令記述を継続することができますが、_ _asm ブロックでは + 印が無効です。
・ラベルに必ずコロン(:)を付けてください。
アセンブラは１カラム目から書き始めたシンボルをラベルと見なしますが、_ _asm ブロックでは、１カラム目から命令を書くことができます。コンパイラがラベルを認識するためにラベル末尾にコロン(:)が必要です。
・？で始まるローカルラベルは記述できません。
・コメントは C/C++言語形式(/* */と//)で記述してください。
アセンブラ形式(;)のコメントは記述できません。
・コメントはアセンブリソース出力やオブジェクトリスト出力に出力されません。
・.DATA 制御命令を除き、アセンブラ制御命令を記述することはできません。また、ファイルインクルード機能、条件付きアセンブリ機能、マクロ機能、構造化アセンブリ機能をサポートしていません。

(2) シンボル

(2-1) 変数名

・静的変数名をアドレスとして解釈します。auto 変数名は SP（スタックポインタ）からのオフセットとして解釈します。変数名にコンパイラが付加する接頭辞 _ は付加しないで記述します。下の例では x は絶対アドレス、y は SP からのオフセット値になります。

```
int x;
void func()
{
    int y;
    _ _asm {
       mov.w   @x,r0           //mov.w   @_x,r0
       mov.w   @(y,sp),r1      //mov.w   @(0,sp),r1
    }
}
```

・_ _asm ブロックで参照される C/C++の変数はメモリに割り付けられます。
・C++の auto 変数とパラメタは参照できません。

(2-2) 関数名

・関数名の参照は C リンケージ関数のみ可能とします。
関数名にコンパイラが付加する接頭辞 _ は付加しないで記述します。

(2-3) ラベル

・C/C++ラベルと_ _asm 内ラベルの相互参照はできません。他の_ _asm ブロックのラベルを参照できません。

・ロケーションカウンタ($)を記述できます。

(2-4) enum メンバ名

・enum 型データのメンバ名を定数として使用可能です。

(2-5) 構造体メンバ名

・「変数名.メンバ名」で静的変数の場合はアドレス、auto 変数の場合は SP からのオフセットと解釈します。

・「OFFSET 変数名.メンバ名」または「OFFSET(変数名.メンバ名)」のように OFFSET 演算子を指定すると構造体先頭からのメンバのオフセット値と解釈します。

・「変数名->メンバ名」「OFFSET(変数名->メンバ名)」のように->を記述することはできません。

・ビットフィールドは参照できません。

・例：

```
struct S {
    int a;
    int b;
} x, *p;
void func()
{
    _ _asm {
        mov.w   @x.b,r0                 // mov.w   @_x+2,r0
        mov.l   @p,er1                  // mov.l   @_p,er1
        mov.w   r0,@(OFFSET(x.b),er1)   // mov.w   r0,@(2,er1)
    }
}
```

(2-6) セクション名

・セクション名は STARTOF,SIZEOF 演算子のオペランドとしてのみ使用可能です。

(3) 演算子

・演算子は以下に示すアセンブリ言語の演算子のみ使用可能です。但し、排他的論理和演算子は「~」の代わりに「^」を使用してください。

単項プラス(+) 単項マイナス(-) 加算(+) 減算(-) 乗算(*) 除算(/)
単項否定(~) 論理積(&) 論理和(|) 排他的論理和(^)
算術左シフト(<<) 算術右シフト(>>)
セクションの先頭アドレス(STARTOF) セクションのサイズ(SIZEOF)
上位バイト抽出(HIGH) 下位バイト抽出(LOW)
上位ワード抽出(HWORD) 下位ワード抽出(LWORD)

(4) 整数定数

・整数定数はアセンブリ言語形式を使用不可とし、C/C++言語形式のみ使用可能とします。例えば、16 進数は H'FF ではなく 0xFF のように記述します。

(5) 文字定数

・文字定数はアセンブリ言語形式を使用不可とし、C/C++言語形式の文字列リテラルを使用します。例えば、文字列定数を"a"ではなく'a'のように記述します。"a"と書くとヌル文字で終わる文字列とみなされます。

(6) レジスタ規約

・_ _asm ブロックのレジスタ規約は関数呼び出しと同様です。_ _asm ブロック内で ER0, ER1, (ER2)のような caller-save レジスタを使用しても_ _asm ブロックの入口/出口で退避/回復しません。_ _asm ブロック内で(ER2), ER3, ER4, ER5, ER6 のような callee-save レジスタを使用した場合、_ _asm ブロックの入口/出口で退避/回復するコードが自動的に生成されます。_ _asm ブロック内の入口から出口まで SP は変化しないことを前提としていますが、関数呼び出しで SP を一時的に変更した場合は、関数呼び出し後に元に戻してください。

・_ _asm ブロック内で MAC レジスタを使用しても、_ _asm ブロックの入口/出口で MAC レジスタを退避/回復するコードを一切生成しません。_ _asm ブロック内で MAC レ

ジスタを変更し、かつ、_ _asm ブロックの前後で MAC レジスタの値を保持する場合は、_ _asm ブロック内で MAC レジスタを退避／回復するコードを書いてください。割り込み関数に macsave オプションを指定した場合に_ _asm ブロック内で MAC レジスタを書き換えていてもコンパイラは MAC レジスタが使用されていると認識しません。

例

```
CPU=H8SXA
int g_x;
struct ST {
    int  a;
    char b;
    char c;
}   g_st;
enum color {BLUE, GREEN, YELLOW, RED};
void func(void)
{
    int x;                          // SP からのオフセット=0
    int y;                          // SP からのオフセット=2
    struct ST l_st;                 // SP からのオフセット=4
    _ _asm{                         // 以降が実際のコードイメージ
        // y : local, scalar, SP からのオフセット=2
        mov.w @(y,sp),r0            // mov.w @(2,sp),r0
        mov.l #y,er1                // mov.l #2,er1

        // l_st : local, struct, SP からのオフセット=4
        mov.w @(l_st.b, sp),r0      // mov.w @(6,sp),r0
        mov.l #l_st.c, er1          // mov.l #7,er1
        mov.l #OFFSET l_st.c,er0    // mov.l #3,er0
        mov.l #l_st,er2             // mov.l #4,er2
        bra   L1
CHAR:   .data.b  'a'
STRING: .data.w  "abc"
ENUM:   .data.w  YELLOW
BOTTOM: .data.l  STARTOF P + SIZEOF P
L1:
        // g_st : global, struct
        mov.b #0xFF,@g_st.b         // mov.w #H'FF,@_g_st+2
        mov.l #g_st.b,er1           // mov.l #_g_st+2,er1
        mov.l #OFFSET g_st.b,er2    // mov.l #2,er2
        mov.l #g_st,er3             // mov.l #_g_st,er3

        // g_x : global, scalar
        mov.w #func,@g_x            // mov.w #_func,@g_x
        mov.l #g_x,er0              // mov.l #_g_x,er0
    }
}
```

備　考　_ _asm ブロックで記述したアセンブリプログラムは code=machinecode オプションでオブジェクトファイルに直接出力できます。
_ _asm 内で SP を動かした場合、ソースレベルデバッグの動作は保証しません。

グローバル変数のレジスタ割り付け

#pragma global_register
__global_register

書　式　#pragma global_register (<変数名>=<レジスタ名>[,…])
__global_register (<レジスタ名>) <型指定子> <変数名>
<型指定子> __global_register(<レジスタ名>) <変数名>
<変数名>　：local 変数、C++非静的メンバデータは指定不可
<レジスタ名>：ER4、ER5 (H8/300 時は R4、R5)

説　明　<変数名>で指定したグローバル変数に、<レジスタ名>で指定したレジスタを割り付けます。

例

```
#pragma global_register(x=R4)       /* 外部変数 x を R4 に割り付けます。*/
int x;
__global_register(R5L) char y;      /* 外部変数 y を R5L に割り付けます。*/
void func1(void)
{
  x++;
}
void func2(void)
{
  y=0;
}
void func(int  a)
{
  x = a;
  func1();
  func2();
}
```

備　考
- pragma global_register 宣言後の変数定義・変数宣言が対象になります。
- グローバル変数で、単純型またはポインタ型の変数に使用できます。double 型の変数は指定できません。
- 初期値の設定はできません。また、アドレスの参照もできません。
- 指定された変数の、(ファイル内にレジスタ指定のない)リンク先からの参照は保証されません。
- 割り込み関数内での設定・参照は保証されません。
- 変数、レジスタの重複指定、#pragma abs8、#pragma abs16、__abs8、__abs16、__near8、__near16 との二重指定はできません。

構造体、共用体、クラスの境界調整数の指定

#pragma pack 1
#pragma pack 2
#pragma unpack

書　式　
```
#pragma pack 1
#pragma pack 2
#pragma unpack
```

説　明　ソースプログラム中の指定位置以降の構造体、共用体、クラスメンバの境界調整数を指定します。
本拡張子が指定されていない場合または#pragma unpack 指定位置以降で宣言された構造体、共用体、クラスメンバの境界調整数は pack オプションの指定に従います。#pragma pack 拡張子と境界調整数の関係を**表** 10.27 に示します。

表 10.27 #pragma pack **拡張子と構造体、共用体、クラスメンバの境界調整数**

拡張子／メンバの型	#pragma pack 1	#pragma pack 2	#pragma unpack **または指定なし**
[unsigned]char	**1**	**1**	**1**
[unsigned]short、[unsigned]int、[unsigned]long、**浮動小数点型、ポインタ型**	**1**	**2**	pack **オプションに従う**
境界調整数が１の構造体、共用体、クラス	**1**	**1**	**1**
境界調整数が２の構造体、共用体、クラス	**1**	**2**	pack **オプションに従う**

例

```
#pragma pack 2
struct S1 {
   char a;                 /*   offset:0                                          */
                           /*   gap:  1 byte                                      */
   int b;                  /*   offset:2                                          */
   char c;                 /*   offset:4                                          */
                           /*   gap:  1 byte                                      */
};
#pragma pack 1
struct S2 {
   char a;                 /*   offset:0                                          */
   int b;                  /*   offset:1                                          */
   char c;                 /*   offset:3                                          */
};
#pragma unpack             /*   pack オプション指定に従う。デフォルト pack=2 を仮定   */
struct S3 {
   char a;                 /*   offset:0                                          */
                           /*   gap:  1 byte                                      */
   int b;                  /*   offset:2                                          */
   char c;                 /*   offset:4                                          */
                           /*   gap:  1 byte                                      */
};
struct S1 s1 = {1,2,3};    /*   _s1: .data.b 1,0,0,2,3,0                          */
struct S2 s2 = {1,2,3};    /*   _s2: .data.b 1,0,2,3                              */
struct S3 s3 = {1,2,3};    /*   _s3: .data.b 1,0,0,2,3,0                          */
void test()                /*   _test:                                  */
{                          /*   mov.w #1,R0                             */
   s1.b=1;                 /*   mov.w R0,@_s1+2                         */
   s2.b=2;                 /*   mov.w #2,R0      ; 境界調整が 1 のメンバへの          */
     :                     /*   mov.b R0H,@_s2+1 ; 設定・参照は、バイト単位           */
}                          /*   mov.b R0L,@_s2+2 ; で行います           */
```

備　考

・構造体メンバの境界調整数は、pack オプションでも指定できます。オプションと拡張子の両方が指定された場合には、拡張子の指定を優先します。

・構造体、共用体、クラスの境界調整数は、メンバの中の最大の境界調整数と同じになります。詳細は**「10.1.2 データの内部表現 (2)複合型、クラス型」**を参照してください。

・#pragma pack 1 または pack=1 オプションを指定した構造体、共用体、クラスのメンバは、ポインタを用いてアクセスできません（ポインタを用いたメンバ関数内でのアクセスを含みます）。

例：（cpu=2600a 指定時）

```
#pragma pack 1
struct S {
   char x;
   int y;
} s;
int *p=&s.y; // s.y のアドレスは奇数になることがあります
void test()
{
   s.y=1;     // 正しくアクセスできます
   *p =1;     // 正しくアクセスできません
}
```

変数アクセス時のバイトサイズ指定

__evenaccess

書　式　__evenaccess <型指定子> <変数名>
<型指定子> __evenaccess <変数名>

説　明　整数型の変数に対して、必ず宣言した変数のサイズでのメモリアクセスを行います。
ただし、H8/300 では、4 バイトサイズのスカラ型変数については 2 バイトでのアクセスとなります。H8SX では、備考を参照してください。

例

```
#define A (*(volatile unsigned short __evenaccess *)0xff0178)
void test(void)
{
   A &= ~0x2000 ;
}
```

__evenaccess 未指定時
(BCLR.B での 1 バイトメモリアクセス)

```
_test:
  MOV.L     #H'FF0178,ER0
  BCLR.B    #H'5,@ER0
  RTS
```

__evenaccess 指定時
(MOV.W での 2 バイトメモリアクセス)

```
_test:
   MOV.W    @H'FF0178:24,R0
   BCLR.B #H'5,R0H
   MOV.W    R0,@H'FF0178:24
   RTS
```

備　考　2 バイトのカウンタレジスタなどの場合 1 バイトアクセスすると、アクセスしていない他方の 1 バイトが不定値になることがあります。
それらの場合には本キーワードを指定してアクセスを行うようにしてください。

また、CPU 種別が H8SX の場合には__evenaccess を全ての型に指定できます。構造体、共用体およびクラスのビットフィールドを含むメンバにも指定できます。構造体、共用体およびクラス全体に指定した場合は個々のメンバに指定した場合と同様になります。double 型データに対する 8 バイト単位でのアクセスはできません。
H8SX でリトルエンディアン空間がサポートされている場合、リトルエンディアン空間のデータは__evenaccess を活用して宣言した型のサイズでアクセスしてください。また、H8SX ではリトルエンディアン空間にビッグエンディアンの初期値を置くことを防ぐために__evenaccess 指定した静的変数に初期値を指定するとエラーになります。
例：

```
__evenaccess long x=0x12345678;   /* エラー */
 void f (void)
 {
    ...
 }
```

10.2.2 セクションアドレス演算子

_ _sectop
_ _secend

書 式 `_ _sectop("<セクション名>")`
`_ _secend ("<セクション名>")`

説 明 `_ _sectop` で指定した<セクション名>の先頭アドレスを参照します。
`_ _secend` で指定した<セクション名>の末尾+1 アドレスを参照します。

例

```
#include <machine.h>
#pragma section $DSEC
static const struct {
  void *rom_s;     /* 初期化データセクションの ROM 上の先頭アドレス       */
  void *rom_e;     /* 初期化データセクションの ROM 上の最終アドレス       */
  void *ram_s;     /* 初期化データセクションの RAM 上の先頭アドレス       */
}DTBL[]=  {_ _sectop (“D”), _ _secend (“D”), _ _sectop (“R”)};

#pragma section $BSEC
static const struct {
  void *b_s;       /* 未初期化データセクションの先頭アドレス             */
  void *b_e;       /* 未初期化データセクションの最終アドレス             */
}BTBL[]=  {_ _sectop (“B”), _ _secend (“B”)};

#pragma section
#pragma stacksize 0x100    /* スタックセクション S を宣言                */
#pragma entry INIT /* 関数 INIT をエントリ関数として宣言                  */
void main(void);   /* main 関数を宣言                                    */
void INIT(void)    /* _INIT:                        ;エントリ関数スタート
                   */
{                  /*     MOV   #STARTOF S+SIZEOF S,SP ;SP 初期設定*/
    _INITSCT();    /*     JSR   @_ _INITSCT   ;セクション領域の初期化 */
    main();        /*     JSR   @_main          ;main 関数呼び出し     */
    sleep();       /*     SLEEP                  ;低消費電力状態で待機*/
}
```

セクション初期化の方法の詳細は**「9.2.2 実行環境の作成」**を参照してください。

10.2.3 組み込み関数

C/C++言語で記述できない以下の機能を、組み込み関数として提供します。

・コンディションコードレジスタの設定・参照
・エクステンドレジスタの設定・参照
・積和演算
・ローテート演算
・特殊命令(TRAPA、SLEEP、MOVFPE、MOVTPE、TAS、EEPMOV、NOP、XCH)
・オーバフロー判定
・10 進演算

組み込み関数は、通常の関数と同様に関数呼び出し形式で記述します。
ただし、組み込み関数を使用する場合は、必ず#include <machine.h>を宣言してください。

組み込み関数の一覧を**表** 10.28 に示します。

表 10.28 **組み込み関数の一覧**

	項目	仕様	機能
1	コンディションコードレジスタ	void set_imask_ccr(unsigned char mask)	**割り込みマスクに** mask **の値を設定**
2		unsigned char get_imask_ccr(void)	**割り込みマスクの参照**
3		void set_ccr(unsigned char ccr)	**コンディションコードレジスタの設定（引数** ccr **の値→**CCR**）**
4		unsigned char get_ccr(void)	**コンディションコードレジスタの参照**
5		void and_ccr(unsigned char ccr)	**コンディションコードレジスタの論理積（**CCR & **引数** ccr → CCR**）**
6		void or_ccr(unsigned char ccr)	**コンディションコードレジスタの論理和（**CCR | **引数** ccr → CCR**）**
7		void xor_ccr(unsigned char ccr)	**コンディションコードレジスタの排他的論理和（**CCR ^ **引数** ccr → CCR**）**
8	エクステンドレジスタ	void set_imask_exr(unsigned char mask)	**割り込みマスクに** mask **の値を設定**
9		unsigned char get_imask_exr(void)	**割り込みマスクの参照**
10		void set_exr(unsigned char exr)	**エクステンドレジスタの設定（引数** exr → EXR**）**
11		unsigned char get_exr(void)	**エクステンドレジスタの参照**
12		void and_exr(unsigned char exr)	**エクステンドレジスタの論理積（**EXR & **引数** exr → EXR**）**
13		void or_exr(unsigned char exr)	**エクステンドレジスタの論理和（**EXR | **引数** exr → EXR**）**
14		void xor_exr(unsigned char exr)	**エクステンドレジスタの排他的論理和（**EXR ^ **引数** exr → EXR**）**
15	積和演算	long mac(long val,int *ptr1,int *ptr2, unsigend long count)	MAC **命令を用いて** val+ **Σ** i=0,count-1(ptr1[i]＊ptr2[i])**を演算**
		long macl(long val,int *ptr1,int *ptr2, unsigned long count,unsigned long mask)	**またはリングバッファ機能を用いて、** val+ **Σ** i=0,count-1(ptr1[i]＊*((ptr2+i)&mask))
16	64bit 乗算	long mulsu(long val1, long val2)	MULS/U **命令に展開**
		unsigned long muluu(unsigned long val1, unsigned long val2)	MULU/U **命令に展開**

	項目	仕様	機能
17	ローテート演算	char rotlc(int count,char data) int rotlw(int count,int data) long rotll(int count,long data)	data を count ビット分左ローテート
18		char rotrc(int count,char data) int rotrw(int count,int data) long rotrl(int count,long data)	data を count ビット分右ローテート
19	特殊命令	void trapa(unsigned int trap_no)	TRAPA #trap_no に展開
20		void sleep(void)	SLEEP 命令に展開
21		void movfpe(char *addr,char data) char _movfpe(char *addr)	MOVFPE 命令で*addr を data に設定、または、*addr をリターン
22		void movtpe(char data,char *addr)	MOVTPE 命令を用いて、data を*addr に設定
23		void tas(char *addr)	TAS 命令を用いて、*addr を 0 と比較、その結果をコンディションコードに設定し、*addr の最上位ビットを"1"にセット
24		void eepmov(void *dst,const void *src, unsigned char size) void eepmov(void *dst,const void *src, unsigned int size) void eepmovb(void *dst,const void *src, unsigned char size) void eepmovw(void *dst,const void *src, unsigned int size)	EEPMOV 命令を用いて size バイト分 *src を*dst に転送
25		void eepmovi(void *dst,const void *src, unsigned int size)	
26		void movmdb(void *src, const void *dst, unsigned int count) void movmdw(int *src, const int *dst, unsigned int count) void movmdl(long *src, const long *dst, unsigned int count)	movmd.b 命令を用いて count で指定した回数分だけ*src を*dst に転送 movmd.w 命令を用いて count で指定した回数分だけ*src を*dst に転送 movmd.l 命令を用いて count で指定した回数分だけ*src を*dst に転送
27		unsigned int movsd(char *src, const char *dst, unsigned int size)	movsd 命令を用いて最大で size バイト分だけ*src を*dst に転送。但しゼロデータを転送した時点で転送終了。
28		void nop(void)	NOP 命令に展開
29	コンディションコード反映演算	int ovfaddc(char dst,char src,char *rst) int ovfadduc(unsigned char dst, unsigned char src,unsigned char *rst) int ovfaddw(int dst,int src,int *rst) int ovfadduw(unsigned int dst, unsigned int src,unsigned int *rst) int ovfaddl(long dst,long src,long *rst) int ovfadddul(unsigned long dst, unsigned long src,unsigned long *rst)	dst + src を*rst に設定し、演算結果のコンディションコードを反映

	項目	仕様	機能
30	コンディションコード反映演算	int ovfsubc(char dst,char src,char *rst) int ovfsubuc(unsigned char dst, unsigned char src,unsigned char *rst) int ovfsubw(int dst,int src,int *rst) int ovfsubuw(unsigned int dst, unsigned int src,unsigned int *rst) int ovfsubl(long dst,long src,long *rst) int ovfsubul(unsigned long dst, unsigned long src,unsigned long *rst)	dst - src を*rst に設定し、 演算結果のコンディションコードを反映
31		int ovfshalc(char dst,char *rst) int ovfshalw(int dst,int *rst) int ovfshall(long dst,long *rst)	dst << 1 を*rst に設定し、 演算結果のコンディションコードを反映 (算術的シフト)
32		int ovfshlluc(unsigned char dst,unsigned char *rst) int ovfshlluw(unsigned int dst,unsigned int *rst) int ovfshllul(unsigned long dst,unsigned long *rst)	dst << 1 を*rst に設定し、 演算結果のコンディションコードを反映 (論理的シフト)
33		int ovfnegc(char dst,char *rst) int ovfnegw(int dst,int *rst) int ovfnegl(long dst,long *rst)	dst の 2 の補数を*rst に設定し、 演算結果のコンディションコードを反映
34	10 進演算	void dadd(unsigned char size,char *ptr1, char *ptr2,char *rst)	ptr1、ptr2 を size 桁の 10 進数配列として、 10 進加算、結果を*rst に設定
35		void dsub(unsigned char size,char *ptr1, char *ptr2,char *rst)	ptr1、ptr2 を size 桁の 10 進数配列として、 10 進減算、結果を*rst に設定

割り込みマスクビットの設定

void set_imask_ccr (unsigned char mask)

説　明　コンディションコードレジスタ（CCR）の割り込みマスクビット（I）に mask 値（0 または 1）を設定します。

ヘッダ　<machine.h>

引　数　mask　mask 値（0 または 1）

例
```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
void main(void)
{
     set_imask_ccr(0);     /* 割り込みマスクビットをクリアします。      */
}
```

割り込みマスクビットの参照

unsigned char get_imask_ccr(void)

説　明　コンディションコードレジスタ（CCR）の割り込みマスクビット（I）の値（0 または 1）を参照します。

ヘッダ　<machine.h>

リターン値　コンディションコードの割り込みマスクビットの参照値

例
```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
void main(void)
{
     if(get_imask_ccr())   /* 割り込みマスクビットを参照します。        */
       :
}
```

コンディションコードレジスタの設定

void set_ccr(unsigned char ccr)

説　明　コンディションコードレジスタ（CCR）に ccr の値（8 ビット）を設定します。

ヘッダ　<machine.h>

引　数　ccr　設定値(8 ビット)

例

```
#include <machine.h>        /* 必ず<machine.h>をインクルードします。    */
main()
{
    set_ccr(0);             /* CCR をクリアします。                     */
}
```

コンディションコードレジスタの参照

unsigned char get_ccr(void)

説　明　コンディションコードレジスタ（CCR）の値を参照します。

ヘッダ　<machine.h>

リターン値　コンディションコードの参照値

例

```
#include <machine.h>        /* 必ず<machine.h>をインクルードします。    */
void main(void)
{
     unsigned char a;
      a=get_ccr();          /* CCR を参照します。                        */
         :
}
```

コンディションコードレジスタとの論理積

void and_ccr(unsigned char ccr)

説　明　コンディションコードレジスタ（CCR）の値と ccr の論理積を算出し、結果を CCR に設定します。

ヘッダ　<machine.h>

引　数　ccr　論理積の被演算子

例
```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
void main(void)
{
     and_ccr(0x10);        /* CCR & 0x10 を CCR に設定します。         */
}
```

コンディションコードレジスタとの論理和

void or_ccr(unsigned char ccr)

説　明　コンディションコードレジスタ（CCR）の値と ccr の論理和を算出し、結果を CCR に設定します。

ヘッダ　<machine.h>

引　数　ccr　論理和の被演算子

例
```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
void main(void)
{
     or_ccr(0x10);         /* CCR | 0x10 を CCR に設定します。         */
}
```

コンディションコードレジスタとの排他的論理和

void xor_ccr(unsigned char ccr)

説　明　コンディションコードレジスタ（CCR）の値と ccr の排他的論理和を算出し、結果を CCR に設定します。

ヘッダ　<machine.h>

引　数

ccr	排他的論理和の被演算子	

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
void main(void)
{
     xor_ccr(0x10);        /* CCR ^ 0x10 を CCR に設定します。         */
}
```

エクステンドレジスタの割り込みビット設定

void set_imask_exr(unsigned char mask)

説　明　エクステンドレジスタ（EXR）の割り込みマスクビット（I2～I0）に mask 値（0～7）を設定します。
本関数は、CPU／動作モードが H8SXN、H8SXM、H8SXA、H8SXX、2600a、2000a、2600n、2000n のときのみ有効です。

ヘッダ　<machine.h>

引　数

mask	mask 値	

例

```
#include <machine.h>       /*必ず<machine.h>をインクルードします。     */
void main(void)
{
     set_imask_exr(0);     /*エクステンドレジスタの割り込みマスクビット  */
         :                 /*にマスクレベル 0 を設定します。           */
}
```

エクステンドレジスタの割り込みビット参照

unsigned char get_imask_exr(void)

説　明　エクステンドレジスタ（EXR）の割り込みマスクビット（I2～I0）の値（0～7）を参照します。
本関数は、CPU／動作モードが H8SXN、H8SXM、H8SXA、H8SXX、2600a、2000a、2600n、2000n のときのみ有効です。

ヘッダ　<machine.h>

リターン値　エクステンドレジスタの割り込みマスクビットの参照値

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
void main(void)
{
     if(get_imask_exr())   /*エクステンドレジスタの割り込みマスクビット  */
         :                 /*を参照します。                            */
}
```

エクステンドレジスタの設定

void set_exr(unsigned char exr)

説　明　エクステンドレジスタ（EXR）に exr の値（8 ビット）を設定します。
本関数は、CPU／動作モードが H8SXN、H8SXM、H8SXA、H8SXX、2600a、2000a、2600n、2000n のときのみ有効です。

ヘッダ　<machine.h>

引　数　exr　設定値

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
void main(void)
{
     set_exr(0);           /* エクステンドレジスタをクリアします。      */
}
```

エクステンドレジスタの参照

unsigned char get_exr(void)

説　明　エクステンドレジスタ（EXR）の値を参照します。
本関数は、CPU／動作モードが H8SXN、H8SXM、H8SXA、H8SXX、2600a、2000a、2600n、2000n のときのみ有効です。

ヘッダ　<machine.h>

リターン値　エクステンドレジスタの参照値

例
```
#include <machine.h>        /* 必ず<machine.h>をインクルードします。    */
void main(void)
{
unsigned char a;
     a=get_exr();           /* エクステンドレジスタを参照します。        */
         :
}
```

エクステンドレジスタとの論理積

void and_exr(unsigned char exr)

説　明　エクステンドレジスタ（EXR）の値と exr の論理積を算出し、結果を EXR に設定します。
本関数は、CPU／動作モードが H8SXN、H8SXM、H8SXA、H8SXX、2600a、2000a、2600n、2000n のときのみ有効です。

ヘッダ　<machine.h>

引　数　exr　論理積の被演算子

例
```
#include <machine.h>        /* 必ず<machine.h>をインクルードします。    */
void main(void)
{
     and_exr(0x10);         /* EXR & 0x10 を EXR に設定します。         */
}
```

エクステンドレジスタとの論理和

void or_exr(unsigned char exr)

説　明　エクステンドレジスタ（EXR）の値と exr の論理和を算出し、結果を EXR に設定します。
本関数は、CPU／動作モードが H8SXN、H8SXM、H8SXA、H8SXX、2600a、2000a、2600n、2000n のときのみ有効です。

ヘッダ　<machine.h>

引　数　exr　論理和の被演算子

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
void main(void)
{
     or_exr(0x10);         /* EXR | 0x10 を EXR に設定します。     */
}
```

エクステンドレジスタとの排他的論理和

void xor_exr(unsigned char exr)

説　明　エクステンドレジスタ（EXR）の値と exr の排他的論理和を EXR に設定します。
本関数は、CPU／動作モードが H8SXN、H8SXM、H8SXA、H8SXX、2600a、2000a、2600n、2000n のときのみ有効です。

ヘッダ　<machine.h>

引　数　exr　排他的論理和の被演算子

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
void main(void)
{
     xor_exr(0x10);        /* EXR ^ 0x10 を EXR に設定します。         */
}
```

*long mac(long val,int *ptr1,int *ptr2,unsigned long count)*
*long macl(long val,int *ptr1,int *ptr2,unsigned long count,unsigned long mask)*

説　明　積和演算の MAC 命令に展開します。
mac 関数は、val を MAC レジスタの初期値とします。次に、ptr1 と ptr2 で示される 2 バイトのデータを符号付きで乗算し、結果の 4 バイトデータを MAC レジスタに加算後、ptr1 と ptr2 の内容をともに＋2 します。これを count 回数繰り返します。
macl 関数は、ptr2 のデータをリングバッファとして使用するため、mask との論理積演算を行います。
mac、macl 関数は、CPU／動作モードが H8SXN:{M|MD}、H8SXM:{M|MD}、H8SXA:{M|MD}、H8SXX:{M|MD}、2600a、2600n のときのみ有効です。

ヘッダ　<machine.h>

リターン値　積和演算結果

引　数

val	MAC レジスタの初期値
ptr1、ptr2	乗算用データへのポインタ
count	ループ回数
mask	リングバッファ用 mask 値

例

```
#include <machine.h>          /* 必ず<machine.h>をインクルードします。  */
int ptr1[10]={0,1,2,3,4,5,6,7,8,9};
int ptr2[10]={9,8,7,6,5,4,3,2,1,0};
long l1,l2;
：
void main(void)
{
l1=mac(100,ptr1,ptr2,4);      /* l1=100+0*9+1*8+2*7+3*6 を行います。  */
l2=macl(100,ptr1,ptr2,4,~4);  /* l2=100+0*9+1*8+2*9+3*8 を行います。  */
}                             /* ptr2[0], ptr2 [1]のデータをリングバッ */
                              /* ファとして繰り返し使用します。        */
                              /* ptr2 & mask をアドレスとして使用する   */
                              /* ため、ptr2 は8の倍数アドレスに割り付け  */
                              /* る必要があります。                    */
```

備　考　macl の ptr2 の指すテーブルは、mask 値の補数の 2 倍の値に境界調整されていなければなりません。
上記例の場合、ptr2 が 8 の倍数のアドレスに割り付けられていることを、リンケージマップで確認してください。

64bit 乗算結果の上位 32bit 取得

long mulsu(long val1, long val2)
unsigned long muluu(unsigned long val1, unsigned long val2)

説　明　32bit×32bit＝64bit 乗算を行う muls/u,mulu/u 命令への展開を行います。
本組み込み関数の 32bit の各引数(val1,val2)同士を乗算し、結果の上位 32bit を演算結果として返します。

ヘッダ　<machine.h>

引　数

val1	被乗数
val2	乗数

例

```
#include <machine.h>
long          s_val1, s_val2, s_ans;
unsigned long u_val1, u_val2, u_ans;
void f(void)
{
    s_ans = mulsu(s_val1, s_val2);  /* 符号付き 32bit 乗算 */
    u_ans = muluu(u_val1, u_val2);  /* 符号無し 32bit 乗算 */
}
```

備　考　本組み込み関数は、CPU 種別が H8SXN:{M|MD}、H8SXM:{M|MD}、 H8SXA:{M|MD}、H8SXX:{M|MD}の場合にのみ有効です。

ビット左ローテート命令

char rotlc(int count,char data)
int rotlw(int count, int data)
long rotll(int count, long data)

説　明　rotlc、rotlw、rotll 関数は、それぞれ 1 バイト、2 バイト、4 バイトの
data を、左方向に count ビット分ローテート（回転）し、その結果を返します。

ヘッダ　<machine.h>

リターン値　data を count ビット分左ローテートした結果の値

引　数

count	ローテートビット数
data	ビットローテート対象データ

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
int i,data;
void f(void)
{
    i=rot1w(5,data);       /* data を 5bit 左ローテートします。        */
}
```

ビット右ローテート命令

char rotrc(int count, char data)
int rotrw(int count,int data)
long rotrl(int count, long data)

説　明　rotrc、rotrw、rotrl 関数は、それぞれ 1 バイト、2 バイト、4 バイトの data を、右方向に count ビット分ローテート（回転）し、その結果を返します。

ヘッダ　<machine.h>

リターン値　data を count ビット分右ローテートした結果の値

引　数

count	ローテートビット数
data	ビットローテート対象データ

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
int i,data;
void f(void)
{
    i=rotrw(5,data);       /* data を 5bit 右ローテートします。        */
}
```

トラップ命令

void trapa(unsigned int trap_no)

説　明　無条件トラップの TRAPA #trap_no 命令に展開します。trap_no は 0～3 の定数です。
また、本関数は CPU／動作モードが 300 以外のときに有効です。

ヘッダ　<machine.h>

引　数

trap_no	ジャンプ先ベクタアドレスに対する trap 番号

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
void f(void)
{
      :
    trapa(0);              /* trapa #0 でリターンします。              */
}
```

SLEEP 命令

void sleep(void)

説　明　低消費電力状態命令 SLEEP に展開します。

ヘッダ　<machine.h>

例
```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
void f(void)
{
      :
    sleep();               /* sleep 命令に展開します。                 */
}
```

E クロック同期転送命令

*void movfpe(char *addr,char data)*
*char _movfpe(char *addr)*

説　明　E クロック同期データ転送命令 MOVFPE を用いて、*addr を E クロックに同期したタイミングで取り出した値を、movfpe 関数は data へ設定し、_movfpe 関数はリターン値として返却します。*addr は 16 ビット絶対アドレスでアクセス可能なデータを指定してください。

ヘッダ　<machine.h>

リターン値

movfpe 関数の場合	なし
_movfpe 関数の場合	転送先データ

引　数

addr	転送元データへのポインタ
data	転送先データ（movfpe 関数の場合）

例
```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
#pragma abs16 a            /* 第一引数は、16 ビット絶対アドレスで      */
char a,data;               /* アクセスできるように#pragma abs16 で     */
                           /* 宣言します。                             */
void f(void)
{
    movfpe(&a,data);       /* MOVFPE 命令により a を data に転送します。 */
    data = _movfpe(&a);    /* 左記の二つの記述の意味は同一です。 */
}
```

備　考　char _movfpe(char*addr)は、H8SX でのみ有効です。

E クロック同期転送命令

*void movtpe(char data,char *addr)*

説　明　E クロック同期データ転送命令 MOVTPE を用いて、data を E クロックに同期したタイミングで addr へ設定します。*addr は 16 ビット絶対アドレスでアクセス可能なデータを指定してください。

ヘッダ　<machine.h>

引　数

data	転送元データ
addr	転送先へのポインタ

例

```
#include <machine.h>        /* 必ず<machine.h>をインクルードします。   */
#pragma abs16 a             /* 第二引数は、16 ビット絶対アドレスで     */
char a,data;                /* アクセスできるように#pragma abs16 で   */
                            /* 宣言します。                           */
void f(void)
{
    movtpe(data,&a);        /* E クロックに同期したタイミングで data   */
}                           /* を a に転送します。                     */
```

テスト・アンド・セット命令

*void tas(char *addr)*

説　明　テスト・アンド・セット命令 TAS を用いて、addr の内容を 0 と比較し、その結果をコンディションコードレジスタ（CCR）に設定した後、addr の内容の最上位ビットを 1 にします。
本関数は、CPU／動作モードが H8SXN,H8SXM,H8SXA,H8SXX,2600a、2000a、2600n、2000n でのみ有効です。

ヘッダ　<machine.h>

引　数

addr	テスト・アンド・セットを行うデータへのポインタ

例

```
#include <machine.h>        /* 必ず<machine.h>をインクルードします。   */
char a;
void f(void)
{
    tas(&a);                /* a-0 の結果を CCR に設定し、a|=0x80 を   */
}                           /* 行います。                             */
```

ブロック転送命令

*void eepmov(void *dst, const void *src, unsigned char size)*
*void eepmov(void *dst, const void *src ,unsigned int size)*
*void eepmovb(void *dst, const void *src, unsigned char size)*
*void eepmovw(void *dst, const void *src, unsigned int size)*

説　明　ブロック転送命令 EEPMOV を用いて、src で示されるアドレスから size で示されるバイト数分 dst で示すアドレスへブロック転送します。

eepmov 組み込み関数の場合、size には必ず定数値を指定してください。
size の値は CPU／動作モードが 300 のとき最大 255、CPU／動作モードが 300 以外のとき最大 65535 まで指定できます。ただし、256～65535 のときは、EEPMOV.W に展開されますので割り込み要求が発生する場合には使用しないでください。また、size がゼロの場合、転送しません。

eepmovb,eepmobw 組み込み関数の場合、size には変数も指定可能です。
eepmovb 組み込み関数は、常に EEPMOV.B に展開されます。
eepmovw 組み込み関数は、常に EEPMOV.W に展開されます。

ヘッダ　<machine.h>

引　数

dst	転送先へのポインタ
src	転送元へのポインタ
size	転送サイズ

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
char a[10],b[10];
void f(void)
{
    eepmov(b,a,10);        /* EEPMOV 命令を用いて配列 a のデータを      */
}                          /* 配列 b に転送します。                    */
```

備　考　eepmovb 組み込み関数および eepmovw 組み込み関数は、CPU が H8SX の場合にのみ有効です。

割り込み要求対応ブロック転送命令

*void eepmovi(void *dst, const void *src, unsigned int size)*

説　明　ブロック転送命令 EEPMOV を用いて、src で示されるアドレスから size で示されるバイト数分 dst で示すアドレスへブロック転送します。EEPMOV 命令実行中に割り込みが発生して割り込みから復帰後に継続可能な形で命令展開されます。

size には定数値、および変数が指定可能です。
size に定数値を指定する場合は最大 65535 まで指定できます。
size がゼロの場合、転送しません。
size に 255 以下の定数が指定された場合は EEPMOV.B 命令を 1 回出力します。
size に 256 以上 510 以下の定数が指定された場合は EEPMOV.B 命令を 2 回出力します。
size に変数、または 512 以上の定数が指定された場合は以下のようなコードを出力します。
EEPMOV.W 命令実行中に割り込みが発生して中断しても割り込みから復帰後に継続可能です。

```
       L1: EEPMOV.W
           MOV.W    R4,R4
           BNE      L1
```

ヘッダ　<machine.h>

引　数

dst	転送先へのポインタ
src	転送元へのポインタ
size	転送サイズ

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
char a[10],b[10];
void f(void)
{
    eepmovi(b,a,10);       /* EEPMOV 命令を用いて配列 a のデータを      */
}                          /* 配列 b に転送します。                    */
```

備　考　本組み込み関数は、CPU が H8SX の場合にのみ有効です。

H8SX 用ブロック転送命令

*void movmdb(void *dst, const void *src, unsigned int count)*
*void movmdw(int *dst, const int *src, unsigned int count)*
*void movmdl(long *dst, const long *src, unsigned int count)*

説　明　MOVMD.B, MOVMD.W, MOVMD.L 命令がそれぞれ１バイト、２バイト、４バイトのメモリブロックを count で指定した回数分だけ src で示すアドレスから、dst で示すアドレスへ転送します。
count にはゼロから 65535 まで指定することが可能です。但し、count にゼロを指定した場合、転送回数は 65536 になります。

ヘッダ　<machine.h>

引　数

src	転送元へのポインタ
dst	転送先へのポインタ
count	転送回数

例

```
#include <machine.h>     /* 必ず<machine.h>をインクルードします。   */
char s1[100], d1[100];
int  s2[50],  d2[50];
long s4[25],  d4[25];
void f(void)
{
    movmdb(d1, s1, 100); /* MOVMD.B 命令により配列 s1 を配列 d1 へ */
                         /* 100 バイト転送します。 */
    movmdw(d2, s2, 50);  /* MOVMD.W 命令により配列 s2 を配列 d2 へ */
                         /* 100 バイト転送します。 */
    movmdl(d4, s4, 25);  /* MOVMD.L 命令により配列 s4 を配列 d4 へ */
                         /* 100 バイト転送します。 */
}
```

備　考　本組み込み関数は、CPU が H8SX の場合にのみ有効です。

H8SX 用文字列転送命令

*unsigned int movsd(char *dst, const char *src, unsigned int size)*

説　明　ブロック転送命令 movsd によって、src で示すアドレスから dst で示すアドレスへブロック転送を行います。但し、データとしてゼロ(H'00)を転送した時点、または、size で示されるバイト数分だけ転送した時点で終了します。リターン値は size から実際に転送したバイト数を引いた値です。
size にゼロから 65535 まで指定することが可能です。但し、size にゼロを指定すると最大転送バイト数が 65536 と解釈します。

ヘッダ　<machine.h>

リターン値　size で指定したバイト数から実際に転送したバイト数を減算した値

引　数

src	転送元へのポインタ
dst	転送先へのポインタ
size	最大転送バイト数

例

```
#include <machine.h>      /* 必ず<machine.h>をインクルードします。   */
const char *s;
char d[100];
unsigned int remain;

void f(void)
{
    remain = movsd(d, s, 100); /* MOVSD 命令により文字列 s を配列 d へ転送   */
}                              /* します。最大 100 バイト転送します。       */
```

備　考　本組み込み関数は、CPU が H8SX の場合にのみ有効です。

NOP 命令

void nop(void)

説　明　nop 命令に展開します。

ヘッダ　<machine.h>

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
int a;
void f(void)
{
    while(a) nop();        /* a!=0 の間、nop 命令を実行します。         */
}
```

加算オーバフロー判定

*int ovfaddc(char dst,char src,char *rst)*
*int ovfaddw(int dst,int src,int *rst)*
*int ovfaddl(long dst, long src,long *rst)*
*int ovfadduc(unsigned char dst,unsigned char src,unsigned char *rst)*
*int ovfadduw(unsigned int dst,unsigned int src,unsigned int *rst)*
*int ovfaddul(unsigned long dst,unsigned long src,unsigned long *rst)*

説　明　ovfaddc、ovfaddw、ovfaddl 関数は、それぞれ符号付き 1 バイト、2 バイトおよび 4 バイト同士の dst と src の加算を行います。また、ovfadduc、ovfadduw、ovfaddul 関数はそれぞれ符号なしの加算を行います。
そして dst が 0 でない場合のみ結果を rst の示すエリアへ格納します。
加算結果がオーバフローでなければ 0 を、オーバフローのときは 0 以外の値を返します。
これらの関数は、if 文、do 文、while 文、for 文の条件を判定する式でのみ指定することが可能です。
また、ovfaddl、ovfaddul 関数は、CPU が H8/300 以外のときに有効です。

ヘッダ　<machine.h>

リターン値

オーバフローした場合	0 以外の値
オーバフローしていない場合	0

引　数

dst,src	加算の被演算子
rst	結果の格納場所 (rst が 0 の場合は結果を格納しない)

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
int dst,src;
void f(void)
{
    if(ovfaddw(dst,src,0)) /* dst + src の結果を BVC で判定します。     */
        dst=0;
}
```

減算オーバフロー判定

*int ovfsubc(char dst,char src,char *rst)*
*int ovfsubw(int dst,int src,int *rst)*
*int ovfsubl(long dst, long src,long *rst)*
*int ovfsubuc(unsigned char dst,unsigned char src,unsigned char *rst)*
*int ovfsubuw(unsigned int dst,unsigned int src,unsigned int *rst)*
*int ovfsubul(unsigned long dst,unsigned long src,unsigned long *rst)*

説　明　ovfsubc、ovfsubw、ovfsubl 関数は、それぞれ符号付き 1 バイト、2 バイトおよび 4 バイト同士の dst と src の減算(dst-src)を行います。また、ovfsubuc、ovfsubuw、ovfsubul 関数はそれぞれ符号なしの減算を行います。
そして rst が 0 でない場合のみ結果を rst の示すエリアへ格納します。
減算結果がオーバフローでなければ 0 を、オーバフローのときは 0 以外の値を返します。
これらの関数は、if 文、do 文、while 文、for 文の条件を判定する式でのみ指定することが可能です。
また、ovfsubl、ovfsubul 関数は、CPU が H8/300 以外のときに有効です。

ヘッダ　<machine.h>

リターン値

オーバフローした場合	0 以外の値
オーバフローしていない場合	0

引　数

dst,src	減算の被演算子
rst	結果の格納場所(rst が 0 の場合は結果を格納しない)

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
int dst,src;
void f(void)
{
    if(ovfsubw(dst,src,0)) /* dst - src の結果を BVC で判定します。     */
      dst=0;
}
```

算術的シフトオーバフロー判定

*int ovfshalc(char dst,char *rst)*
*int ovfshalw(int dst,int *rst)*
*int ovfshall(long dst,long *rst)*

説　明　ovfshalc、ovfshalw および ovfshall 関数は、1 バイト、2 バイト、4 バイトの dst を左方向へ算術的 1 ビットシフトし、rst が 0 でない場合のみ結果を rst の示すエリアへ格納します。
その後、算術シフト結果がオーバフローでなければ 0 を、オーバフローのときは 0 以外の値を返します。
これらの関数は、if 文、do 文、while 文、for 文の条件を判定する式でのみ指定することができます。
また、ovfshalw および ovfshall 関数は、CPU が H8/300 以外のときに有効です。

ヘッダ　<machine.h>

リターン値

オーバフローした場合	0 以外の値
オーバフローしていない場合	0

引　数

dst	ビットシフト演算の被演算子
rst	結果の格納場所(rst が 0 の時は結果を格納しない)

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
int dst;
void f(void)
{
    if(ovfshalw(dst,0))    /* dst<<1 の結果を BVC で判定します。        */
        dst=0;
 }
```

*int ovfshlluc(unsigned char dst,unsigned char *rst)*
*int ovfshlluw(unsigned int dst,unsigned int *rst)*
*int ovfshllul(unsigned long dst,unsigned long *rst)*

説　明　ovfshlluc、ovfshlluw および ovfshllul 関数は、1 バイト、2 バイト、4 バイトの dst を左方向へ論理的 1 ビットシフトし、rst が 0 でない場合のみ結果を rst の示すエリアへ格納します。
その後、論理シフト結果がオーバフローでなければ 0 を、オーバフローのときは 0 以外の値を返します。
これらの関数は、if 文、do 文、while 文、for 文の条件を判定する式でのみ指定することができます。
また、ovfshlluw および ovfshllul 関数は、CPU が H8/300 以外のときに有効です。

ヘッダ　<machine.h>

リターン値

オーバフローした場合	0 以外の値
オーバフローしていない場合	0

引　数

dst	ビットシフト演算の被演算子
rst	結果の格納場所(rst が 0 の時は結果を格納しない)

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
int dst;
void f(void)
{
    if(ovfshlluw(dst,0))   /* dst<<1 の結果を BCC で判定します。        */
        dst=0;
}
```

補数のオーバフロー判定

*int ovfnegc(char dst, char *rst)*
*int ovfnegw(int dst, int *rst)*
*int ovfnegl(long dst, long *rst)*

説　明　ovfnegc、ovfnegw および ovfnegl 関数は、1 バイト、2 バイト、4 バイトの dst の 2 の補数を算出し、rst が 0 でない場合のみ結果を rst の示すエリアへ格納します。
その後、2 の補数の結果がオーバフローでなければ 0 を、オーバフローのときは 0 以外の値を返します。
これらの関数は、if 文、do 文、while 文、for 文の条件を判定する式でのみ指定することができます。
また、ovfnegw および ovfnegl 関数は、CPU が H8/300 以外のときに有効です。

ヘッダ　<machine.h>

リターン値

オーバフローした場合	0 以外の値
オーバフローしていない場合	0

引　数

dst	補数演算の被演算子
rst	結果の格納場所(rst が 0 の時は結果を格納しない)

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
int dst,rst;
void f(void)
{
    if(ovfnegw(dst,&rst))  /* dst の結果を rst に設定し、-dst の結果    */
        dst=0;             /* のボローで分岐します。                   */
}
```

10 進加算

*void dadd(unsigned char size,char *ptr1,char *ptr2,char *rst)*

説　明　ptr1 から始まる size バイトのデータからと、ptr2 から始まる size バイトのデータの 10 進加算を行い、結果を rst から始まる size バイトのエリアへ格納します。
size は 1～255 の定数です。

ヘッダ　<machine.h>

引　数

size	データサイズ
ptr1,ptr2	10 進加算を行う被演算子
rst	結果の格納領域

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
char ptr1[5]={0x01,0x23,0x45,0x67,0x89};       /* 10 進 123456789 */
char ptr2[5]={0x01,0x23,0x45,0x67,0x89};       /* 10 進 123456789 */
char rst[5];                                   /* 10 進 246913578 */
void main(void)
{
   dadd((char)5,ptr1,ptr2,rst);
                /* ptr1,ptr2 を 10 桁の 10 進数として加算します。       */
}               /* rst= 0x02,0x46,0x91,0x35,0x78                    */
```

10 進減算

*void dsub(unsigned char size,char *ptr1,char *ptr2,char *rst)*

説　明　ptr1 から始まる size バイトのデータと、ptr2 から始まる size バイトのデータの 10 進減算を行い、結果を rst から始まる size バイトのエリアへ格納します。
size は 1～255 の定数です。

ヘッダ　<machine.h>

引　数

size	データサイズ
ptr1,ptr2	10 進減算を行う被演算子
rst	結果の格納場所

例

```
#include <machine.h>       /* 必ず<machine.h>をインクルードします。    */
char ptr1[5]={0x10,0x00,0x00,0x00,0x00};      /* 10 進 1000000000 */
char ptr2[5]={0x01,0x23,0x45,0x67,0x89};      /* 10 進 0123456789 */
char rst[5];                                  /* 10 進 0876543211 */
void main(void)
{
    dsub((char)5,ptr1,ptr2,rst);
                /* ptr1,ptr2 を 10 桁の 10 進数として減算します。       */
}               /* rst=0x08,0x76,0x54,0x32,0x11                  */
```

10.3 C/C++ライブラリ

10.3.1 標準 C ライブラリ

(1) ライブラリの概要

C/C++言語の中で標準的に利用できる関数である C ライブラリ関数の仕様について説明します。ここでは、ライブラリの構成を概説し、本節の読み方および用語について説明します。以降ではライブラリの構成に従って各ライブラリ関数の仕様を説明します。

(a) ライブラリの種類

ライブラリとは、入出力、文字列操作等の標準的な処理を C/C++言語の関数の形式で実現したものです。また、これらのライブラリは、各処理単位ごとに対応した標準インクルードファイルを取り込むことによって使用可能となります。

標準インクルードファイルには、対応するライブラリの宣言とそれらを使用するために必要なマクロ名が定義されています。

表 10.29 にライブラリの種類と対応する標準インクルードファイルを示します。

表 10.29 **ライブラリの種類と対応する標準インクルードファイル**

	ライブラリの種類	内容	標準インクルードファイル
1	プログラム診断用ライブラリ	プログラムの診断情報の出力を行うライブラリです。	＜assert.h＞
2	文字操作用ライブラリ	文字の操作およびチェックを行うライブラリです。	＜ctype.h＞
3	数値計算用ライブラリ	三角関数等の数値計算を行うライブラリです。	＜math.h＞ ＜mathf.h＞
4	プログラムの制御移動用ライブラリ	関数間の制御の移動をサポートするライブラリです。	＜setjmp.h＞
5	可変個の実引数アクセス用ライブラリ	可変個の実引数を持つ関数に対し、その実引数へのアクセスをサポートするライブラリです。	＜stdarg.h＞
6	入出力用ライブラリ	入出力操作を行うライブラリです。 <no_float.h>を用いることで浮動小数点をサポートしない簡易入出力関数を提供します。	＜stdio.h＞ ＜no_float.h＞
7	標準処理用ライブラリ	記憶域管理等の C プログラムでの標準的処理を行うライブラリです。	＜stdlib.h＞
8	文字列操作用ライブラリ	文字列の比較、複写等を行うライブラリです。	＜string.h＞

また、以上の標準インクルードファイルの他にプログラムの作成作業の効率向上を図るため**表** 10.30 に示すマクロ名の定義だけからなる標準インクルードファイルがあります。

表 10.30 **マクロ名定義からなる標準インクルードファイル**

	標準インクルードファイル	内容
1	＜stddef.h＞	各標準インクルードファイルで共通に使用するマクロ名を定義します。
2	＜float.h＞	浮動小数点数の内部表現に関する各種制限値を定義します。
3	＜limits.h＞	コンパイラの内部処理に関する各種制限値を定義します。
4	＜errno.h＞	ライブラリ関数においてエラーが発生した時に errno に設定する値を定義します。

(b)　**ライブラリの説明形式**

ライブラリの各関数を標準インクルードファイルごとに分類し、その標準インクルードファイルごとに説明します。その各分類は、まず、標準インクルードファイルの中で定義されているマクロ名や関数宣言に対する説明を行い **（図 10.3 参照）**、その後、各関数ごとの説明を行う **（図 10.4 参照）** という形式で構成されています。

図 10.3 に標準インクルードファイルの説明形式、**図** 10.4 に関数の説明形式を示します。

> 項番　＜標準インクルードファイル名＞
>
> ・本インクルードファイルがもつ全体的な機能の概要を説明します。
>
> ・本インクルードファイル内で定義・宣言される名前を名前種別（【型】、【定数】、【変数】、【関数】）に分類して説明します。マクロである場合、名前種別のタイトル（【】内）または名前の説明箇所に（マクロ）と表記しています。
>
> ・処理系定義仕様がある場合や、本インクルードファイル内で宣言されている関数に共通する注意事項がある場合、説明を補足します。

図 10.3　標準インクルードファイルの説明形式

> *機能の概要を示します。*
>
> *ラ [illegible] ラ関数 [illegible] 型 [illegible] 引数 [illegible] 示 [illegible] び [illegible] ま [illegible] 。*
>
> | | |
> |---|---|
> | 説　明 | ライブラリ関数の機能を説明します。 |
> | ヘッダ | 宣言元の標準インクルードファイル名です。 |
> | リターン値 | 正常：　ライブラリ関数が正常終了したときの値です。
異常：　ライブラリ関数が異常終了したときの値です。 |
> | 引　数 | 引数の意味を説明します。 |
> | 例 | 呼び出し手順を説明します。 |
> | エラー条件 | ライブラリ関数の処理でリターン値からでは、判断できないエラーが発生する条件を示します。このようなエラーが発生したとき、エラーの種類に対応する、コンパイラごとに定義された値が errno* に設定されます。 |
> | 備　考 | 補足説明、または使用上の注意事項です。 |
> | 処理系定義仕様 | 本コンパイラの処理方法です。 |

図 10.4　関数の説明形式

【注】* 　errno **は、ライブラリ関数実行中に生じたエラーの種類を格納する変数です。詳細については「**10.3.1(2) ＜stddef.h＞**」を参照してください。**

(c) ライブラリ関数の説明で使用する用語

（i）ストリーム入出力

データの入出力において、1 文字ごとの入出力関数の呼び出しの度に入出力装置を駆動したり、OS の機能を呼び出していたのでは、効率が悪くなります。そこで、通常はバッファと呼ばれる記憶域を用意しておき、バッファ内のデータを一括して入出力を行います。

一方、プログラムの側から見ると、1 文字ごとに入出力関数を呼び出せた方が便利です。

ライブラリ関数では、バッファの管理を自動的に行うことにより、プログラム内でバッファの状態を意識することなしに、1 文字単位ごとの入出力を効率よく行うことができます。

このように、データの入出力を効率よく実現するために詳細な手段を意識せず、入出力をひとつのデータの流れ（ストリーム）と考えてプログラムを作ることのできる機能をストリーム入出力といいます。

（ii）FILE 構造体およびファイルポインタ

ストリーム入出力に必要なバッファや、その他の情報は、ひとつの構造体の中に記憶されており、標準インクルードファイル＜stdio.h＞の中で FILE という名前で定義されています。

ストリーム入出力においては、ファイルはすべて FILE 構造体のデータ構造を持つものとして扱います。このようなファイルをストリームファイルと呼びます。このファイル構造体へのポインタをファイルポインタと呼び、入出力ファイルを指定するために用います。

ファイルポインタは、

```
FILE *fp;
```

と定義します。

fopen 関数等でファイルをオープンすると、ファイルポインタが得られますが、オープン処理が失敗すると NULL が返ってきます。NULL ポインタを、他のストリーム入出力関数に指定すると、その関数は異常終了しますので、注意が必要です。ファイルをオープンした時は、必ずファイルポインタの値をチェックするようにしてください。

（iii）関数とマクロ

ライブラリ関数の実現方法としては、関数とマクロの二通りがあります。

関数は、通常のユーザ作成の関数と同じインタフェースを持ち、リンク時に取り込みます。

マクロは、その関数に関連した標準インクルードファイルの中で＃define 文を用いて定義されています。

マクロについては、以下の点に注意する必要があります。

- マクロは、プリプロセッサによって自動的に展開されてしまうので、ユーザが同じ名前の関数を宣言してもマクロを無効にすることはできません。
- マクロのパラメタとして副作用のある式（代入式、インクリメント、デクリメント）を指定した時、その効果は保証されません。

例：

パラメタの絶対値を求める MACRO を以下のようにマクロ定義します。

```
#define  MACRO(a) ((a) >= 0 ? (a) : -(a))
```

と定義されている時、

```
X=MACRO(a++)
```

がプログラム内にあると、

```
X = ((a++) >= 0 ? (a++) : -(a++))
```

と展開され、a は 2 回インクリメントされることになり、また結果の値も最初の a の値の絶対値とは異なります。

（iv）EOF

getc 関数、getchar 関数、fgetc 関数等のファイルからデータを入力する関数において、ファイル終了（End Of File）時に返される値です。EOF は、標準インクルードファイル＜stdio.h＞の中で定義されています。

（v）NULL

ポインタが何も指していない時の値です。NULL という名前は、標準インクルードファイル＜stddef.h＞の中で定義されています。

（vi）ヌル文字

C/C++言語における文字列の終わりは、文字"\0"によって示されることになっています。ライブラリ関数における文字列のパラメタも、すべてこの約束に従っていなければなりません。この文字列の終わりを示す文字"\0"を、以下ヌル文字と呼びます。

（vii）リターンコード

ライブラリ関数の中には、リターン値によって、指定された処理が成功したか、失敗したか等の結果を判断するものがあります。このような場合のリターン値を特にリターンコードと呼びます。

（viii）テキストファイルとバイナリファイル

多くのシステムでは、データを格納するために、特殊なファイル形式を持っています。これをサポートするために、ライブラリ関数には、テキストファイルとバイナリファイルの 2 種類のファイル形式があります。

- テキストファイル
  テキストファイルは、通常のテキストを格納するためのファイルで、行の集まりとして構成されています。テキストファイルの入力の時、行の区切りとして改行文字（"\n"）が入力されます。また、出力の時、改行文字（"\n"）を出力することにより、現在の行の出力を終了します。テキストファイルは、処理系ごとの標準的なテキストを格納するファイルの入出力を行うためのファイルです。テキストファイルでは、ライブラリ関数で入出力する文字は必ずしもファイル内の物理的なデータの並びと対応していません。
- バイナリファイル
  バイナリファイルは、バイトデータの列として構成されているファイルです。ライブラリ関数で入出力するデータは、ファイル内の物理的なデータの並びと対応しています。

（ix）標準入出力ファイル

入出力のライブラリ関数で、ファイルのオープン等の準備を行わずに標準的に使用できるファイルを標準入出力ファイルといいます。標準入出力ファイルには、標準入力ファイル（stdin）、標準出力ファイル（stdout）、標準エラー出力ファイル（stderr）があります。

- 標準入力ファイル（stdin）
  プログラムへの入力となる標準的なファイルです。
- 標準出力ファイル（stdout）
  プログラムからの出力となる標準的なファイルです。
- 標準エラー出力ファイル（stderr）
  プログラムからのエラーメッセージ等の出力を行うための標準的なファイルです。

（x）浮動小数点数

浮動小数点数は、実数を近似して表現したものです。C/C++言語のソースプログラム上では浮動小数点数を 10 進数で表現していますが、計算機の内部では通常 2 進数で表現されます。
2 進数の場合の浮動小数点数の表現は次のようになります。

$2^n \times m$（n：整数、m：2 進小数）

ここで n を浮動小数点数の指数部、m を仮数部といいます。浮動小数点数を一定のデータサイズで表現するために、n と m のビット数は通常固定されています。
以下、浮動小数点数に関する用語を説明します。

- 基数
  浮動小数点数が何進法で表現されているかを示す整数値です。通常、基数は2です。
- 丸め
  浮動小数点数よりも精度の高い演算の途中結果を浮動小数点数に格納する場合に丸めが行われます。丸めには、切り上げ、切り捨て、四捨五入（2進小数の場合は、0捨1入となります。）があります。
- 正規化
  浮動小数点数を、$2^n \times m$の形式で表現する場合、同一の数値を表わす異なる表現が可能です。

例：

$2^5 \times 1.0_{(2)}$　（$_{(2)}$ は 2 進数を示します。）
$2^6 \times 0.1_{(2)}$
どちらも同じ値です。

通常は、有効桁数を確保するために、前者のように先頭の桁が0でないような表現を用います。これを正規化された浮動小数点数といい、浮動小数点数をこのような表現に変換する操作を正規化といいます。

- ガードビット
  浮動小数点数の演算の途中結果を保持する場合、通常は丸めを行うために実際の浮動小数点数よりも1ビット多いデータを用意します。しかし、これだけでは桁落ち等が生じた時に正確な結果を求めることができません。このために、もう1ビット設けて演算の途中結果を保持する手法があります。このビットをガードビットといいます。

（xi）ファイルアクセスモード

ファイルをオープンする時にどのような処理をファイルに行うかを示す文字列です。文字列の種類には**表** 10.31 に示す 12 種類があります。

表 10.31　**ファイルアクセスモードの種類**

	アクセスモード	意味
1	”r”	**テキストファイルを読み込み用にオープンします。**
2	”w”	**テキストファイルを書き出し用にオープンします。**
3	”a”	**テキストファイルを追加用にオープンします。**
4	”rb”	**バイナリファイルを読み込み用にオープンします。**
5	”wb”	**バイナリファイルを書き出し用にオープンします。**
6	”ab”	**バイナリファイルを追加用にオープンします。**
7	”r＋”	**テキストファイルを読み込み用でかつ更新用にオープンします。**
8	”w＋”	**テキストファイルを書き出し用でかつ更新用にオープンします。**
9	”a＋”	**テキストファイルを追加用でかつ更新用にオープンします。**
10	”r＋b”	**バイナリファイルを読み込み用でかつ更新用にオープンします。**
11	”w＋b”	**バイナリファイルを書き出し用でかつ更新用にオープンします。**
12	”a＋b”	**バイナリファイルを追加用でかつ更新用にオープンします。**

（xii）処理系定義

コンパイラが異なることによって定義が異なるという意味です。

（xiii）エラー指示子、ファイル終了指示子

ストリームファイルごとに、ファイルの入出力の際にエラーが生じたかどうかを示すエラー指示子と、入力ファイルが終了したかどうかを示すファイル終了指示子というデータを保持しています。

これらのデータは、それぞれ ferror 関数、feof 関数によって参照することができます。

ストリームファイルを扱う関数のうち、そのリターン値だけからではエラーの発生やファイルの終了の情報が得られないものがあります。エラー指示子とファイル終了指示子は、このような関数の実行後にファイルの状態を調べるために使用することができます。

（xiv）位置指示子

ディスク上のファイル等、ファイル内の任意の位置からの読み書きができるストリームファイルは、現在読み書きしているファイル内の位置を示すデータを保持しています。これを位置指示子と言います。端末装置等、ファイル内の読み書きの位置を変更できないストリームファイルでは、位置指示子は使用しません。

(d) ライブラリ使用時の注意事項

- ライブラリの中で定義されているマクロの内容は、コンパイラごとに異なります。
  ライブラリを使用する場合、これらのマクロの内容を再定義した場合、動作は保証されません。
- ライブラリは、すべての場合についてエラーを検出しているわけではありません。以降の説明に示す以外の形式でライブラリ関数を呼び出した場合、動作は保証されません。

(2) ＜stddef.h＞

標準インクルードファイルの中で共通に使用されるマクロ名を定義します。

以下は、すべて処理系定義です。

種別	定義名	説明
型	ptrdiff_t	二つのポインタを減算した結果の型です。
(マクロ)	size_t	sizeof 演算子による演算結果の型です。
定数 (マクロ)	NULL	ポインタが何も指していない時の値です。 これは、0 と等値演算子(==)による比較結果が真になるような値です。
変数 (マクロ)	errno	ライブラリ関数の処理中にエラーが発生した場合、そのライブラリごとに定義されたエラーコードがこの errno に設定されます。 ライブラリ関数を呼び出す前に errno に 0 を設定しておき、ライブラリ関数の処理終了後に errno に設定されているコードを調べることによってライブラリ関数の処理中に発生したエラーをチェックすることができます。

処理系定義仕様

	項目		
1	マクロ NULL の値	void 型へのポインタ型の値 0 です	
2	マクロ ptrdiff_t の内容	int 型	H8SX 用ノーマルモード H8SX 用ミドルモード H8SX 用アドバンストモードかつ ptr16 オプション有り H8SX 用マキシマムモードかつ ptr16 オプション有り H8S/2600 用ノーマルモード H8S/2000 用ノーマルモード H8/300H 用 ノーマルモード H8/300 用
		long 型	H8SX 用アドバンストモードかつ ptr16 オプション無し H8SX 用マキシマムモードかつ ptr16 オプション無し H8S/2600 用アドバンストモード H8S/2000 用アドバンストモード H8S/300H 用アドバンストモード

(3) ＜assert.h＞

プログラム中に診断機能を付け加えます。

種別	定義名	説明
関数（マクロ）	assert	プログラム中に診断機能を付け加えます。

＜assert.h＞で定義される診断機能を無効にするためには、＜assert.h＞を取り込む前に NDEBUG というマクロ名を#define 文で定義してください（#define NDEBUG）。

【注】 assert **というマクロ名に対して**#undef **文を使用すると、それ以降の** assert **の呼び出しの効果は保証されません。**

void assert(int expression)

説　明　プログラム中に診断機能を付け加えます。

ヘッダ　`<assert.h>`

引　数　`expression`　評価する式

例

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

備　考　`assert` マクロは `expression` が真の時は値を返さずに処理を終了します。`expression` が偽の時は、診断情報をコンパイラによって定義された書式で標準エラーファイルに出力し、その後 `abort` 関数を呼び出します。
診断情報の中には、パラメタのプログラムテキスト、ソースファイル名、ソース行番号が含まれています。

処理系定義仕様　`assert(expression)`において、`expression` が偽の時メッセージを出力します。
`ASSERTION FAILED:`Δ式Δ`FILE`Δ<ファイル名>,`line`Δ<行番号>

(4) ＜ctype.h＞

文字に対して、その種類の判定や変換を行います。

種別	定義名	説明
関数	isalnum	英字または 10 進数字かどうかを判定します。
	isalpha	英字かどうかを判定します。
	iscntrl	制御文字かどうかを判定します。
	isdigit	10 進数字かどうかを判定します。
	isgraph	空白を除く印字文字かどうかを判定します。
	islower	英小文字かどうかを判定します。
	isprint	空白を含む印字文字かどうかを判定します。
	ispunct	特殊文字かどうかを判定します。
	isspace	空白類文字かどうかを判定します。
	isupper	英大文字かどうかを判定します。
	isxdigit	16 進数字かどうかを判定します。
	tolower	英大文字を英小文字に変換します。
	toupper	英小文字を英大文字に変換します。

上記の関数において、入力パラメタの値が unsigned char 型で表現できる範囲に含まれず、なおかつ EOF でない場合、その関数の動作は保証されません。また、文字の種類の一覧を**表** 10.32 に示します。

表 10.32 **文字の種類**

	文字の種類	内容
1	英大文字	以下の 26 文字のいずれかの文字です。 ‘A’、‘B’、‘C’、‘D’、‘E’、‘F’、‘G’、‘H’、‘I’、‘J’、‘K’、‘L’、‘M’、 ‘N’、‘O’、‘P’、‘Q’、‘R’、‘S’、‘T’、‘U’、‘V’、‘W’、‘X’、‘Y’、‘Z’
2	英小文字	以下の 26 文字のいずれかの文字です。 ‘a’、‘b’、‘c’、‘d’、‘e’、‘f’、‘g’、‘h’、‘i’、‘j’、‘k’、‘l’、‘m’、 ‘n’、‘o’、‘p’、‘q’、‘r’、‘s’、‘t’、‘u’、‘v’、‘w’、‘x’、‘y’、‘z’
3	英字	英大文字と英小文字のいずれかの文字です。
4	10 進数字	以下の 10 文字のいずれかの文字です。 ‘0’、‘1’、‘2’、‘3’、‘4’、‘5’、‘6’、‘7’、‘8’、‘9’
5	印字文字	空白（‘ ’）を含む、ディスプレイ上に表示される文字のことです。 ASCII コードの 0x20～0x7E に対応します。
6	制御文字	印字文字以外の文字のことです。
7	空白類文字	以下の 6 文字のいずれかの文字です。 空白（‘ ’）、書式送り（‘\f’）、改行（‘\n’）、復帰（‘\r’）、水平タブ（‘\t’）、垂直タブ（‘\v’）
8	16 進数字	以下の 22 文字のいずれかの文字です。 ‘0’、‘1’、‘2’、‘3’、‘4’、‘5’、‘6’、‘7’、‘8’、‘9’、 ‘A’、‘B’、‘C’、‘D’、‘E’、‘F’、‘a’、‘b’、‘c’、‘d’、‘e’、‘f’
9	特殊文字	空白（‘ ’）、英字、および 10 進数字を除く任意の印字文字のことです。

処理系定義仕様

	項目	コンパイラの仕様
1	isalnum 関数、isalpha 関数、iscntrl 関数、islower 関数、isprint 関数、isupper 関数で判定される文字集合	unsigned char 型で表現できる文字集合です。判定の結果、真になる文字を表 10.33 に示します。

表 10.33 真となる文字の集合

	関数名	真となる文字
1	isalnum	'0'〜'9', 'A'〜'Z', 'a'〜'z'
2	isalpha	'A'〜'Z', 'a'〜'z'
3	iscntrl	'\x00'〜'\x1f', '\x7f'
4	islower	'a'〜'z'
5	isprint	'\x20'〜'\x7E'
6	isupper	'A'〜'Z'

英字、10 進数字判定

int isalnum(int c)

説　明　文字が英字または 10 進数字であるかどうか判定します。

ヘッダ　`<ctype.h>`

リターン値　文字 c が英字または 10 進数字の時　：0 以外
文字 c が英字または 10 進数字以外の時　：0

引　数　c　判定する文字

例
```
#include  <ctype.h>
int  c, ret;
    ret=isalnum(c);
```

英字判定

int isalpha(int c)

説　明　　文字が英字であるかどうか判定します。

ヘッダ　　<ctype.h>

リターン値　文字 c が英字の時　　：0 以外
　　文字 c が英字以外の時 ：0

引　数　　c　　判定する文字

例　　
```
#include  <ctype.h>
int  c, ret;
    ret=isalpha(c);
```

制御文字判定

int iscntrl(int c)

説　明　　文字が制御文字であるかどうか判定します。

ヘッダ　　<ctype.h>

リターン値　文字 c が制御文字の時　　：0 以外
　　文字 c が制御文字以外の時　：0

引　数　　c　　判定する文字

例　　
```
#include  <ctype.h>
int  c, ret;
    ret=iscntrl(c);
```

10 進数字判定

int isdigit(int c)

説　明　文字が 10 進数字であるかどうか判定します。

ヘッダ　<ctype.h>

リターン値　文字 c が 10 進数字の時　：0 以外
文字 c が 10 進数字以外の時　：0

引　数　c　判定する文字

例
```
#include  <ctype.h>
int  c, ret;
    ret=isdigit(c);
```

空白を除く印字文字判定

int isgraph(int c)

説　明　文字が空白（‘ ’）を除く任意の印字文字かどうかを判定します。

ヘッダ　<ctype.h>

リターン値　文字 c が空白を除く印字文字の時　：0 以外
文字 c が空白を除く印字文字以外の時　：0

引　数　c　判定する文字

例
```
#include  <ctype.h>
int  c, ret;
    ret=isgraph(c);
```

英小文字判定

int islower(int c)

説　明　文字が英小文字であるかどうか判定します。

ヘッダ　<ctype.h>

リターン値　文字 c が英小文字の時　：0 以外
文字 c が英小文字以外の時　：0

引　数　c　判定する文字

例
```
#include  <ctype.h>
int  c, ret;
    ret=islower(c);
```

印字文字判定

int isprint(int c)

説　明　文字が空白文字（‘ ’）を含む印字文字であるかどうか判定します。

ヘッダ　<ctype.h>

リターン値　文字 c が空白文字を含む印字文字の時　：0 以外
文字 c が空白文字を含む印字文字以外の時　：0

引　数　c　判定する文字

例
```
#include  <ctype.h>
int  c, ret;
    ret=isprint(c);
```

特殊文字判定

int ispunct(int c)

説　明　文字が特殊文字であるかどうか判定します。

ヘッダ　<ctype.h>

リターン値　文字 c が特殊文字の時　：0 以外
文字 c が特殊文字以外の時　：0

引　数　c　判定する文字

例
```
#include  <ctype.h>
int  c, ret;
    ret=ispunct(c);
```

空白類文字判定

int isspace(int c)

説　明　文字が空白類文字であるかどうか判定します。

ヘッダ　<ctype.h>

リターン値　文字 c が空白類文字の時　：0 以外
文字 c が空白類文字以外の時　：0

引　数　c　判定する文字

例
```
#include  <ctype.h>
int  c, ret;
    ret=isspace(c);
```

英大文字判定

int isupper(int c)

説　明　　文字が英大文字であるかどうか判定します。

ヘッダ　　<ctype.h>

リターン値　文字 c が英大文字の時　　：0 以外
　　文字 c が英大文字以外の時　：0

引　数　　c　　判定する文字

例　　
```
#include  <ctype.h>
int  c, ret;
    ret=isupper(c);
```

16 進数字判定

int isxdigit(int c)

説　明　　文字が 16 進数字かどうか判定します。

ヘッダ　　<ctype.h>

リターン値　文字 c が 16 進数字の時　　：0 以外
　　文字 c が 16 進数字以外の時　：0

引　数　　c　　判定する文字

例　　
```
#include  <ctype.h>
int  c, ret;
    ret=isxdigit(c);
```

英小文字に変換

int tolower(int c)

説　明　英大文字を対応する英小文字に変換します。

ヘッダ　<ctype.h>

リターン値　文字 c が英大文字の時　：文字 c に対応する英小文字
文字 c が英大文字以外の時　：文字 c

引　数　c　変換する文字

例
```
#include  <ctype.h>
int  c, ret;
    ret=tolower(c);
```

英大文字に変換

int toupper(int c)

説　明　英小文字を対応する英大文字に変換します。

ヘッダ　<ctype.h>

リターン値　文字 c が英小文字の時　：文字 c に対応する英大文字
文字 c が英小文字以外の時　：文字 c

引　数　c　変換する文字

例
```
#include  <ctype.h>
int  c, ret;
    ret=toupper(c);
```

(5) ＜float.h＞

浮動小数点数の内部表現に関する各種制限値を定義します。

以下はすべて処理系定義です。

種別	定義名	定義値	説明
定数	FLT_RADIX	2	指数部表現における基数です。
(マクロ)	FLT_ROUNDS	1	加算演算結果を丸めるかどうかを示します。本マクロの定義の意味は以下のとおりです。 ・演算結果を丸める場合　　：正の値 ・演算結果を切り捨てる場合：0 ・特に規定しない場合　　：－1 丸め、切り捨ての方法は、処理系定義です。
	FLT_GUARD	1	乗算演算結果においてガードビットを用いるかどうかを示します。本マクロの定義の意味は以下のとおりです。 ・ガードビットを用いる場合　：1 ・ガードビットを用いない場合：0
	FLT_NORMALIZE	1	浮動小数点数の値を正規化するかどうかを示します。本マクロの定義の意味は以下のとおりです。 ・正規化する場合　：1 ・正規化しない場合：0
	FLT_MAX	3.4028235677973364e+38F	float 型が浮動小数点数値として表現できる最大値です。
	DBL_MAX	1.7976931348623158e+308	double 型が浮動小数点数値として表現できる最大値です。
	LDBL_MAX	1.7976931348623158e+308	long double 型が浮動小数点数値として表現できる最大値です。
	FLT_MAX_EXP	127	float 型が浮動小数点数値として表現できる基数のべき乗の最大値です。
	DBL_MAX_EXP	1023	double 型が浮動小数点数値として表現できる基数のべき乗の最大値です。
	LDBL_MAX_EXP	1023	long double 型が浮動小数点数値として表現できる基数のべき乗の最大値です。
	FLT_MAX_10_EXP	38	float 型が浮動小数点数値として表現できる 10 のべき乗の最大値です。
	DBL_MAX_10_EXP	308	double 型が浮動小数点数値として表現できる 10 のべき乗の最大値です。
	LDBL_MAX_10_EXP	308	long double 型が浮動小数点数値として表現できる 10 のべき乗の最大値です。
	FLT_MIN	1.175494351e-38F	float 型が浮動小数点数値として表現できる正の値での最小値です。
	DBL_MIN	2.2250738585072014e-308	double 型が浮動小数点数値として表現できる正の値での最小値です。
	LDBL_MIN	2.2250738585072014e-308	long double 型が浮動小数点数値として表現できる正の値での最小値です。
	FLT_MIN_EXP	-149	loat 型が正の値として表現できる浮動小数点数値の基数のべき乗の最小値です。
	DBL_MIN_EXP	-1074	double 型が正の値として表現できる浮動小数点数値の基数のべき乗の最小値です。

種別	定義名	定義値	説明
定数（マクロ）	LDBL_MIN_EXP	-1074	long double 型が正の値として表現できる浮動小数点数値の基数のべき乗の最小値です。
	FLT_MIN_10_EXP	-44	float 型が正の値として表現できる浮動小数点数値の 10 のべき乗の最小値です。
	DBL_MIN_10_EXP	-323	double 型が正の値として表現できる浮動小数点数値の 10 のべき乗の最小値です。
	LDBL_MIN_10_EXP	-323	long double 型が正の値として表現できる浮動小数点数値の 10 のべき乗の最小値です。
	FLT_DIG	6	float 型の浮動小数点数値の 10 進精度の最大桁数です。
	DBL_DIG	15	double 型の浮動小数点数値の 10 進精度の最大桁数です。
	LDBL_DIG	15	long double 型の浮動小数点数値の 10 進精度の最大桁数です。
	FLT_MANT_DIG	24	float 型の浮動小数点数値を基数に合わせて表現した時の仮数部の最大桁数です。
	DBL_MANT_DIG	53	double 型の浮動小数点数値を基数に合わせて表現した時の仮数部の最大桁数です。
	LDBL_MANT_DIG	53	long double 型の浮動小数点数値を基数に合わせて表現した時の仮数部の最大桁数です。
	FLT_EXP_DIG	8	float 型の浮動小数点数値を基数に合わせて表現した時の指数部の最大桁数です。
	DBL_EXP_DIG	11	double 型の浮動小数点数値を基数に合わせて表現した時の指数部の最大桁数です。
	LDBL_EXP_DIG	11	long double 型の浮動小数点数値を基数に合わせて表現した時の指数部の最大桁数です。
	FLT_POS_EPS	5.9604648328104311e-8F	float 型において、1.0＋x≠1.0 である最小の浮動小数点数値 x を示します。
	DBL_POS_EPS	1.1102230246251567e-16	double 型において、1.0＋x≠1.0 である最小の浮動小数点数値 x を示します。
	LDBL_POS_EPS	1.1102230246251567e-16	long double 型において、1.0＋x≠1.0 である最小の浮動小数点数値 x を示します。
	FLT_NEG_EPS	2.9802324164052156e-8F	float 型において、1.0－x≠1.0 である最小の浮動小数点数値 x を示します。
	DBL_NEG_EPS	5.5511151231257834e-17	double 型において、1.0－x≠1.0 である最小の浮動小数点数値 x を示します。
	LDBL_NEG_EPS	5.5511151231257834e-17	long double 型において、1.0－x≠1.0 である最小の浮動小数点数値 x を示します。
	FLT_POS_EPS_EXP	-23	float 型において、1.0＋(基数)n≠1.0 となる最小の整数 n を示します。
	DBL_POS_EPS_EXP	-52	double 型において、1.0＋(基数)n≠1.0 となる最小の整数 n を示します。
	LDBL_POS_EPS_EXP	-52	long double 型において、1.0＋(基数)n≠1.0 となる最小の整数 n を示します。

種別	定義名	定義値	説明
定数 (マクロ)	FLT_NEG_EPS_EXP	-24	float 型において、1.0－(基数)n≠1.0 となる最小の整数 n を示します。
	DBL_NEG_EPS_EXP	-53	double 型において、1.0－(基数)n≠1.0 となる最小の整数 n を示します。
	LDBL_NEG_EPS_EXP	-53	long double 型において、1.0－(基数)n≠1.0 となる最小の整数 n を示します。

(6) ＜limits.h＞

整数型データの内部表現に関する各種制限値を定義します。

以下はすべて処理系定義です。

種別	定義名	定義値	説明
定数 (マクロ)	CHAR_BIT	8	char 型が何ビットから構成されるかを示します。
	CHAR_MAX	127	char 型の変数が値として持つことができる最大値です。
	CHAR_MIN	-128	char 型の変数が値として持つことができる最小値です。
	SCHAR_MAX	127	signed char 型の変数が値として持つことができる最大値です。
	SCHAR_MIN	-128	signed char 型の変数が値として持つことができる最小値です。
	UCHAR_MAX	255u	unsigned char 型の変数が値として持つことができる最大値です。
	SHRT_MAX	32767	short 型の変数が値として持つことができる最大値です。
	SHRT_MIN	-32768	short 型の変数が値として持つことができる最小値です。
	USHRT_MAX	65535u	uunsigned short 型の変数が値として持つことができる最大値です。
	INT_MAX	32767	int 型の変数が値として持つことができる最大値です。
	INT_MIN	-32768	int 型の変数が値として持つことができる最小値です。
	UINT_MAX	65535u	unsigned int 型の変数が値として持つことができる最大値です。
	LONG_MAX	2147483647	long 型の変数が値として持つことができる最大値です。
	LONG_MIN	-2147483647L-1L	long 型の変数が値として持つことができる最小値です。
	ULONG_MAX	4294967295u	unsigned long 型の変数が値として持つことができる最大値です。

(7) ＜errno.h＞

ライブラリ関数においてエラーが発生したときに errno に設定する値を定義します。

以下は、すべて処理系定義です。

種別	定義名	説明
変数 (マクロ)	errno	int 型変数です。ライブラリ関数においてエラーが発生したときにエラー番号が設定されます。
定数 (マクロ)	ERANGE	「12.3 C ライブラリ関数のエラーメッセージ」を参照してください。
	EDOM	
	EDIV	
	ESTRN	
	PTRERR	
	ECBASE	
	ETLN	
	EEXP	
	EEXPN	
	EFLOATO	
	EFLOATU	
	EDBLO	
	EDBLU	
	ELDBLO	
	ELDBLU	
	NOTOPN	
	EBADF	
	ECSPEC	

(8) ＜math.h＞

各種の数値計算を行います。

以下の定数（マクロ）はすべて処理系定義です。

種別	定義名	説明
定数 (マクロ)	EDOM	関数に入力するパラメタの値が関数内で定義している値の範囲を超える時、errno に設定する値です。
	ERANGE	関数の計算結果が double 型の値として表わせない時、あるいはオーバーフロー／アンダフローとなった時、errno に設定する値です。
	HUGE_VAL	関数の計算結果がオーバーフローした時に、関数のリターン値として返す値です。
関数	acos	浮動小数点数の逆余弦を計算します。
	asin	浮動小数点数の逆正弦を計算します。
	atan	浮動小数点数の逆正接を計算します。
	atan2	浮動小数点数どうしを除算した結果の値の逆正接を計算します。
	cos	浮動小数点数のラジアン値の余弦を計算します。
	sin	浮動小数点数のラジアン値の正弦を計算します。
	tan	浮動小数点数のラジアン値の正接を計算します。
	cosh	浮動小数点数の双曲線余弦を計算します。
	sinh	浮動小数点数の双曲線正弦を計算します。
	tanh	浮動小数点数の双曲線正接を計算します。
	exp	浮動小数点数の指数関数を計算します。
	frexp	浮動小数点数を[0.5, 1.0]の値として 2 のべき乗の積に分解します。
	ldexp	浮動小数点数と 2 のべき乗の乗算を計算します。
	log	浮動小数点数の自然対数を計算します。
	log10	浮動小数点数の 10 を底とする対数を計算します。
	modf	浮動小数点数を整数部分と小数部分に分解します。
	pow	浮動小数点数のべき乗を計算します。
	sqrt	浮動小数点数の正の平方根を計算します。
	ceil	浮動小数点数の小数点以下を切り上げた整数値を求めます。
	fabs	浮動小数点数の絶対値を計算します。
	floor	浮動小数点数の小数点以下を切り捨てた整数値を求めます。
	fmod	浮動小数点数どうしを除算した結果の余りを計算します。

エラーが発生した時の動作を以下に説明します。

(1) 定義域エラー
関数に入力するパラメタの値が関数内で定義している値の範囲を超えている時、定義域エラーというエラーが発生します。この時errnoにはEDOMの値が設定されます。また、関数のリターン値は、処理系定義です。

(2) 範囲エラー
関数における計算結果がdouble型の値として表わせない時には範囲エラーというエラーが発生します。この時、errnoにはERANGEの値が設定されます。また、計算結果がオーバーフローの時は、正しく計算が行われた時と同様の符号のHUGE_VALの値をリターン値として返します。逆に計算結果がアンダフローの時は、0をリターン値として返します。

【注】1. **<math.h>の関数の呼び出しによって定義域エラーが発生する可能性がある場合は、結果の値をそのまま用いるのは危険です。必ず**errno**をチェックしてから用いてください。**

例：

```
        .
        .
        .
1   x=asin(a);
2   if (errno==EDOM)
3           printf ("error\n");
4   else
5           printf ("result is : %lf\n", x);
        .
        .
        .
```

1 行目で、asin **関数を使って逆正弦値を求めます。このとき、引数** a **の値が、**asin **関数の定義域［**−1.0, 1.0**］の範囲を超えていると、**errno **に値** EDOM **が設定されます。**2 **行目で定義域エラーが生じたかどうかの判定をします。定義域エラーが生じれば、**3 **行目で、**error **を出力します。定義域エラーが生じなければ** 5 **行目で、逆正弦値を出力します。**

2. **範囲エラーが発生するかどうかは、コンパイラによって定まる、浮動小数点数の内部表現形式によって異なります。たとえば無限大を値として表現できる内部表現形式を採用している場合、範囲エラーの生じないように＜**math.h**＞のライブラリ関数を実現することができます。**

処理系定義仕様

	項目	コンパイラの仕様
1	数学関数の入力引数値が範囲を超えたときの数学関数が返す値	非数を返します。非数の形式は「10.1.3 浮動小数点の仕様」を参照してください
2	数学関数でアンダーフローエラーが発生したときマクロ「ERANGE」の値が「errno」に設定されるかどうか	設定しません。
3	fmod 関数で第 2 実引数の値が 0 の場合、範囲エラーとなるかどうか	範囲エラーとなります。

逆余弦

double acos(double d)

説　明　浮動小数点数の逆余弦を計算します。

ヘッダ　<math.h>

リターン値　正常：　d の逆余弦値
異常：　定義域エラーの時は、非数を返します。

引　数　d　逆余弦を求める浮動小数点数

例
```
#include  <math.h>
double  d, ret;
    ret=acos(d);
```

エラー条件　d の値が[－1.0, 1.0]の範囲を超えている時、定義域エラーになります。

備　考　acos 関数のリターン値の範囲は[0, π]です。

逆正弦

double asin(double d)

説　明　浮動小数点数の逆正弦を計算します。

ヘッダ　<math.h>

リターン値　正常：　d の逆正弦値
異常：　定義域エラーの時は、非数を返します。

引　数　d　逆正弦を求める浮動小数点数

例
```
#include  <math.h>
double  d, ret;
    ret=asin(d);
```

エラー条件　d の値が[－1.0, 1.0]の範囲を超えている時、定義域エラーになります。

備　考　asin 関数のリターン値の範囲は[－π／2, π／2]です。

double atan(double d)

説　明　　浮動小数点数の逆正接を計算します。

ヘッダ　　<math.h>

リターン値　d の逆正接値

引　数　　d　　逆正接を求める浮動小数点数

例　　
```
#include  <math.h>
double  d, ret;
    ret=atan(d);
```

備　考　　atan 関数のリターン値の範囲は（－π／2, π／2）です。

除算後の逆正接

double atan2(double y, double x)

説　明　浮動小数点数どうしを除算した結果の値の逆正接を計算します。

ヘッダ　<math.h>

リターン値　正常：　y を x で除算したときの逆正接値
異常：　定義域エラーの時は、非数を返します。

引　数

x	除数
y	被除数

例

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

エラー条件　x, y の値がともに 0.0 の時、定義域エラーになります。

備　考　atan2 関数のリターン値の範囲は（－π, π］です。atan2 関数の示す意味を**図** 10.5 に示します。図に示すように、atan2 関数の結果は、点（x, y）と原点を通る直線と x 軸をなす角を求めます。y＝0.0 で x が負の時、結果はπ、x＝0.0 の時、y の値の正負に従って結果は±π／2 となります。

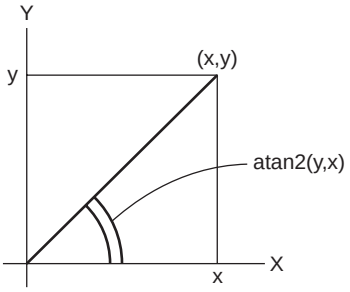


図 10.5　atan2 **関数の意味**

余弦

double cos(double d)

説　明　　浮動小数点数のラジアン値の余弦を計算します。

ヘッダ　　<math.h>

リターン値　d の余弦値

引　数　　d　　余弦を求めるラジアン値

例

```
#include  <math.h>
double  d, ret;
    ret=cos(d);
```

正弦

double sin(double d)

説　明　　浮動小数点数のラジアン値の正弦を計算します。

ヘッダ　　<math.h>

リターン値　d の正弦値

引　数　　d　　正弦を求めるラジアン値

例

```
#include  <math.h>
double  d, ret;
    ret=sin(d);
```

正接

double tan(double d)

説　明　浮動小数点数のラジアン値の正接を計算します。

ヘッダ　<math.h>

リターン値　d の正接値

引　数　d　正接を求めるラジアン値

例
```
#include  <math.h>
double  d, ret;
    ret=tan(d);
```

双曲線余弦

double cosh(double d)

説　明　浮動小数点数の双曲線余弦を計算します。

ヘッダ　<math.h>

リターン値　d の双曲線余弦値

引　数　d　双曲線余弦を求める浮動小数点数

例
```
#include  <math.h>
double  d, ret;
    ret=cosh(d);
```

双曲線正弦

double sinh(double d)

説　明　　浮動小数点数の双曲線正弦を計算します。

ヘッダ　　<math.h>

リターン値　d の双曲線正弦値

引　数　　d　　双曲線正弦を求める浮動小数点数

例

```
#include  <math.h>
double  d, ret;
    ret=sinh(d);
```

双曲線正接

double tanh(double d)

説　明　　浮動小数点数の双曲線正接を計算します。

ヘッダ　　<math.h>

リターン値　d の双曲線正接値

引　数　　d　　双曲線正接を求める浮動小数点数

例

```
#include  <math.h>
double  d, ret;
    ret=tanh(d);
```

指数関数

double exp(double d)

説　明　浮動小数点数の指数関数を計算します。

ヘッダ　<math.h>

リターン値　d の指数関数値

引　数　d　指数関数を求める浮動小数点数

例
```
#include  <math.h>
double  d, ret;
    ret=exp(d);
```

浮動小数点数を仮数、指数に分解

double frexp(double value, double ret)

説　明　浮動小数点数を［0.5, 1.0）の値と 2 のべき乗の積に分解します。

ヘッダ　<math.h>

リターン値　value が 0.0 の時：0.0
value が 0.0 でない時：ret*2^e の指している領域の値=value で定義される ret の値

引　数　value　［0.5, 1.0）の値と 2 のべき乗の積に分解する浮動小数点数
e　2 のべき乗値を格納する記憶域へのポインタ

例
```
#include  <math.h>
double  ret, value;
int  *e;
    ret=frexp(value, e);
```

備　考　frexp 関数は value を［0.5, 1.0）の値と 2 のべき乗の積に分解します。e の指す領域には、分解した結果の 2 のべき乗値を設定します。
リターン値 ret の値の範囲は［0.5, 1.0）または 0.0 になります。
value が 0.0 ならば、e の指す int 型の記憶域の内容と ret の値は 0.0 になります。

仮数、指数を浮動小数点数に変換

double ldexp(double ret, int f)

説　明　浮動小数点数と 2 のべき乗の積を計算します。

ヘッダ　<math.h>

リターン値　$e*2^f$ の演算結果の値

引　数

e	2 のべき乗値を求める浮動小数点数
f	2 のべき乗値

例

```
#include  <math.h>
double  ret, e;
int  f;
    ret=ldexp(e, f);
```

自然対数

double log(double d)

説　明　浮動小数点数の自然対数を計算します。

ヘッダ　<math.h>

リターン値　正常：　d の自然対数の値
異常：　定義域エラーの時は、非数を返します。

引　数

d	自然対数を求める浮動小数点数

例

```
#include  <math.h>
double  d, ret;
    ret=log(d);
```

エラー条件　d の値が負の時、定義域エラーになります。
d の値が 0.0 の時、範囲エラーになります。

常用対数

double log10(double d)

説　明　浮動小数点数の 10 を底とする対数を計算します。

ヘッダ　<math.h>

リターン値　正常：　d は 10 を底とする対数値
異常：　定義域エラーの時は、非数を返します。

引　数　d　10 を底とする対数を求める浮動小数点数

例
```
#include  <math.h>
double  d, ret;
    ret=log10(d);
```

エラー条件　d の値が負の値の時、定義域エラーになります。
d の値が 0.0 の時、範囲エラーになります。

浮動小数点数を整数部、小数部に分解

*double modf(double a, double *b)*

説　明　浮動小数点数を整数部分と小数部分に分解します。

ヘッダ　<math.h>

リターン値　a の小数部分

引　数　a　整数部分と小数部分に分解する浮動小数点数
b　整数部分を格納する記憶域を指すポインタ

例
```
#include  <math.h>
double  a, *b, ret;
    ret=modf(a, b);
```

べき乗

double pow(double x, double y)

説　明　浮動小数点数のべき乗を計算します。

ヘッダ　<math.h>

リターン値　正常：　x の y 乗の値
異常：　定義域エラーの時は、非数を返します。

引　数

x	べき乗される値
y	べき乗する値

例

```
#include  <math.h>
double  x, y, ret;
    ret=pow(x, y);
```

エラー条件　x の値が 0.0 で、かつ y の値が 0.0 以下の時、あるいは x の値が負で y の値が整数値でない時、定義域エラーになります。

平方根

double sqrt(double d)

説　明　浮動小数点数の正の平方根を計算します。

ヘッダ　<math.h>

リターン値　正常：　d の正の平方根の値
異常：　定義域エラーの時は、非数を返します。

引　数

d	正の平方根を求める浮動小数点数

例

```
#include  <math.h>
double  d, ret;
    ret=sqrt(d);
```

エラー条件　d の値が負の値の時、定義域エラーになります。

切り上げ

double ceil(double d)

説　明　浮動小数点数の小数点以下を切り上げた整数値を求めます。

ヘッダ　<math.h>

リターン値　d の小数点以下を切り上げた整数値

引　数　d　小数点以下を切り上げる浮動小数点数

例
```
#include  <math.h>
double  d, ret;
    ret=ceil(d);
```

備　考　ceil 関数は d の値より大きいかまたは等しい最小の整数値を double 型の値として返す関数です。したがって d の値が負の値の時は小数点以下を切り捨てた時の値を返します。

絶対値

double fabs(double d)

説　明　浮動小数点数の絶対値を計算します。

ヘッダ　<math.h>

リターン値　d の絶対値

引　数　d　絶対値を求める浮動小数点数

例
```
#include  <math.h>
double  d, ret;
    ret=fabs(d);
```

切り捨て

double floor(double d)

説　明　浮動小数点数の小数点以下を切り捨てた整数値を求めます。

ヘッダ　<math.h>

リターン値　d の小数点以下を切り捨てた整数値

引　数　d　小数点以下を切り捨てる浮動小数点数

例
```
#include  <math.h>
double  d, ret;
    ret=floor(d);
```

備　考　floor 関数は d の値を超えない範囲の整数の最大値を、double 型の値として返す関数です。
したがって d の値が負の値の時は小数点以下を切り上げた時の値を返します。

余り

double fmod(double x, double y)

説　明　浮動小数点数どうしを除算した結果の余りを計算します。

ヘッダ　<math.h>

リターン値　y の値が 0.0 の時：x
y の値が 0.0 でない時：x を y で除算した結果の余り

引　数　x　被除数
y　除数

例
```
#include  <math.h>
double  x, y, ret;
    ret=fmod(x, y);
```

備　考　fmod 関数では、パラメタ x, y、リターン値 ret の間には、次に示す関係が成立します。
x=y*i+ret（ただし i は整数値）
また、リターン値 ret の符号は x の符号と同じ符号になります。
x/y の商を表現できない場合、結果の値は、保証されません。

(9) ＜mathf.h＞

各種の数値計算を行います。
<mathf.h>では ANSI 規格規定外の単精度形式の数学関数の宣言とマクロの定義をしています。
各関数は float 型の引数を受け取り、float 型の値を返します。

以下の定数（マクロ）はすべて処理系定義です。

種別	定義名	説明
定数（マクロ）	EDOM	関数に入力するパラメタの値が関数内で定義している値の範囲を超える時、errno に設定する値です。
	ERANGE	関数の計算結果が float 型の値として表わせない時、あるいはオーバーフロー／アンダフローとなった時、errno に設定する値です。
	HUGE_VAL	関数の計算結果がオーバーフローした時に、関数のリターン値として返す値です。
関数	acosf	浮動小数点数の逆余弦を計算します。
	asinf	浮動小数点数の逆正弦を計算します。
	atanf	浮動小数点数の逆正接を計算します。
	atan2f	浮動小数点数どうしを除算した結果の値の逆正接を計算します。
	cosf	浮動小数点数のラジアン値の余弦を計算します。
	sinf	浮動小数点数のラジアン値の正弦を計算します。
	tanf	浮動小数点数のラジアン値の正接を計算します。
	coshf	浮動小数点数の双曲線余弦を計算します。
	sinhf	浮動小数点数の双曲線正弦を計算します。
	tanhf	浮動小数点数の双曲線正接を計算します。
	expf	浮動小数点数の指数関数を計算します。
	frexpf	浮動小数点数を[0.5, 1.0]の値として 2 のべき乗の積に分解します。
	ldexpf	浮動小数点数と 2 のべき乗の乗算を計算します。
	logf	浮動小数点数の自然対数を計算します。
	log10f	浮動小数点数の 10 を底とする対数を計算します。
	modff	浮動小数点数を整数部分と小数部分に分解します。
	powf	浮動小数点数のべき乗を計算します。
	sqrtf	浮動小数点数の正の平方根を計算します。
	ceilf	浮動小数点数の小数点以下を切り上げた整数値を求めます。
	fabsf	浮動小数点数の絶対値を計算します。
	floorf	浮動小数点数の小数点以下を切り捨てた整数値を求めます。
	fmodf	浮動小数点数どうしを除算した結果の余りを計算します。

エラーが発生した時の動作を以下に説明します。

（1） 定義域エラー
関数に入力するパラメタの値が関数内で定義している値の範囲を超えている時、定義域エラーというエラーが発生します。この時errnoにはEDOMの値が設定されます。また、関数のリターン値は、処理系定義です。

（2） 範囲エラー
関数における計算結果がfloat型の値として表わせない時には範囲エラーというエラーが発生します。この時、errnoにはERANGEの値が設定されます。また、計算結果がオーバーフローの時は、正しく計算が行われた時と同様の符号のHUGE_VALの値をリターン値として返します。逆に計算結果がアンダフローの時は、0をリターン値として返します。

【注】 1. **＜mathf.h＞の関数の呼び出しによって定義域エラーが発生する可能性がある場合は、結果の値をそのまま用いるのは危険です。必ず** errno **をチェックしてから用いてください。**
例：

```
        :
        :
        :
1   x=asinf(a);
2   if (errno==EDOM)
3         printf ("error\n");
4   else
5         printf ("result is : %f\n", x);
        :
        :
        :
```

1 行目で、asinf **関数を使って逆正弦値を求めます。このとき、引数** a **の値が、**asinf **関数の定義域［－**1.0, 1.0**］の範囲を超えていると、**errno **に値** EDOM **が設定されます。**2 **行目で定義域エラーが生じたかどうかの判定をします。定義域エラーが生じれば、**3 **行目で、**error **を出力します。定義域エラーが生じなければ** 5 **行目で、逆正弦値を出力します。**

2. **範囲エラーが発生するかどうかは、コンパイラによって定まる、浮動小数点数の内部表現形式によって異なります。たとえば無限大を値として表現できる内部表現形式を採用している場合、範囲エラーの生じないように＜**mathf.h**＞のライブラリ関数を実現することができます。**

処理系定義仕様

	項目	コンパイラの仕様
1	数学関数の入力引数値が範囲を超えたときの数学関数が返す値	非数を返します。非数の形式は「10.1.3 浮動小数点の仕様」を参照してください
2	数学関数でアンダーフローエラーが発生したときマクロ「ERANGE」の値が「errno」に設定されるかどうか	設定しません。
3	Fmod 関数で第 2 実引数の値が 0 の場合、範囲エラーとなるかどうか	範囲エラーとなります。

逆余弦

float acosf(float f)

説　明　浮動小数点数の逆余弦を計算します。

ヘッダ　<mathf.h>

リターン値　正常：　f の逆余弦値
異常：　定義域エラーの時は、非数を返します。

引　数　f　逆余弦を求める浮動小数点数

例
```
#include  <mathf.h>
float  f, ret;
    ret=acosf(f);
```

エラー条件　f の値が[－1.0, 1.0]の範囲を超えている時、定義域エラーになります。

備　考　acosf 関数のリターン値の範囲は[0, π]です。

逆正弦

float asinf(float f)

説　明　浮動小数点数の逆正弦を計算します。

ヘッダ　<mathf.h>

リターン値　正常：　f の逆正弦値
異常：　定義域エラーの時は、非数を返します。

引　数　f　逆正弦を求める浮動小数点数

例
```
#include  <mathf.h>
float  f, ret;
    ret=asinf(f);
```

エラー条件　f の値が[－1.0, 1.0]の範囲を超えている時、定義域エラーになります。

備　考　asinf 関数のリターン値の範囲は[－π／2, π／2]です。

float atanf(float f)

説　明　浮動小数点数の逆正接を計算します。

ヘッダ　<mathf.h>

リターン値　f の逆正接値

引　数　f　逆正接を求める浮動小数点数

例
```
#include  <mathf.h>
float  f, ret;
    ret=atanf(f);
```

備　考　atanf 関数のリターン値の範囲は（－π／2, π／2）です。

除算後の逆正接

float atan2f(float y, float x)

説　明　浮動小数点数どうしを除算した結果の値の逆正接を計算します。

ヘッダ　<mathf.h>

リターン値　正常：　y を x で除算したときの逆正接値
異常：　定義域エラーの時は、非数を返します。

引　数

x	除数
y	被除数

例

```
#include  <mathf.h>
float  x, y, ret;
    ret=atan2f(y, x);
```

エラー条件　x, y の値がともに 0.0 の時、定義域エラーになります。

備　考　atan2f 関数のリターン値の範囲は（－π, π］です。atan2f 関数の示す意味を**図** 10.6 に示します。図に示すように、atan2f 関数の結果は、点（x, y）と原点を通る直線と x 軸をなす角を求めます。

y＝0.0 で x が負の時、結果はπ、x＝0.0 の時、y の値の正負に従って結果は±π／2 となります。

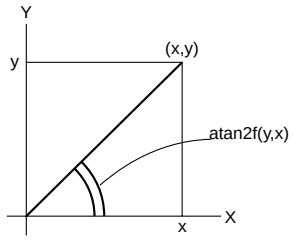


図 10.6　atan2f **関数の意味**

余弦

float cosf(float f)

説　明　浮動小数点数のラジアン値の余弦を計算します。

ヘッダ　<mathf.h>

リターン値　f の余弦値

引　数　f　余弦を求めるラジアン値

例
```
#include  <mathf.h>
float  f, ret;
    ret=cosf(f);
```

正弦

float sinf(float f)

説　明　浮動小数点数のラジアン値の正弦を計算します。

ヘッダ　<mathf.h>

リターン値　f の正弦値

引　数　f　正弦を求めるラジアン値

例
```
#include  <mathf.h>
float  f, ret;
    ret=sinf(f);
```

正接

float tanf(float f)

説　明　浮動小数点数のラジアン値の正接を計算します。

ヘッダ　<mathf.h>

リターン値　f の正接値

引　数　f　正接を求めるラジアン値

例
```
#include  <mathf.h>
float  f, ret;
    ret=tanf(f);
```

双曲線余弦

float coshf(float f)

説　明　浮動小数点数の双曲線余弦を計算します。

ヘッダ　<mathf.h>

リターン値　f の双曲線余弦値

引　数　f　双曲線余弦を求める浮動小数点数

例
```
#include  <mathf.h>
float  f, ret;
    ret=coshf(f);
```

双曲線正弦

float sinhf(float f)

説　明　浮動小数点数の双曲線正弦を計算します。

ヘッダ　<mathf.h>

リターン値　f の双曲線正弦値

引　数　f　双曲線正弦を求める浮動小数点数

例
```
#include  <mathf.h>
float  f, ret;
    ret=sinhf(f);
```

双曲線正接

float tanhf(float f)

説　明　浮動小数点数の双曲線正接を計算します。

ヘッダ　<mathf.h>

リターン値　f の双曲線正接値

引　数　f　双曲線正接を求める浮動小数点数

例
```
#include  <mathf.h>
float  f, ret;
    ret=tanhf(f);
```

指数関数

float expf(float f)

説　明　浮動小数点数の指数関数を計算します。

ヘッダ　<mathf.h>

リターン値　f の指数関数値

引　数　f　指数関数を求める浮動小数点数

例
```
#include  <mathf.h>
float  f, ret;
    ret=expf(f);
```

浮動小数点数を仮数、指数に分解

float frexpf(float value, float ret)

説　明　浮動小数点数を[0.5, 1.0])の値と 2 のべき乗の積に分解します。

ヘッダ　<mathf.h>

リターン値　正常：　value が 0.0 の時：0.0
value が 0.0 でない時：ret*2$^{e\ \text{の指している領域の値}}$=value で定義される ret の値

引　数　value　［0.5, 1.0）の値と 2 のべき乗の積に分解する浮動小数点数
e　2 のべき乗値を格納する記憶域へのポインタ

例
```
#include  <mathf.h>
float  ret, value;
int  *e;
    ret=frexpf(value, e);
```

備　考　frexpf 関数は value を［0.5, 1.0）の値と 2 のべき乗の積に分解します。e の指す領域には、分解した結果の 2 のべき乗値を設定します。
リターン値 ret の値の範囲は［0.5, 1.0）または 0.0 になります。
value が 0.0 ならば、e の指す int 型の記憶域の内容と ret の値は 0.0 になります。

仮数、指数を浮動小数点数に変換

float ldexpf(float ret, int f)

説　明　浮動小数点数と 2 のべき乗の積を計算します。

ヘッダ　<mathf.h>

リターン値　$e*2^f$ の演算結果の値

引　数

e	2 のべき乗値を求める浮動小数点数
f	2 のべき乗値

例

```
#include  <mathf.h>
float  ret, e;
int  f;
    ret=ldexpf(e, f);
```

自然対数

float logf(float f)

説　明　浮動小数点数の自然対数を計算します。

ヘッダ　<mathf.h>

リターン値　正常：　f の自然対数の値
異常：　定義域エラーの時は、非数を返します。

引　数

f	自然対数を求める浮動小数点数

例

```
#include  <mathf.h>
float  f, ret;
    ret=logf(f);
```

エラー条件　f の値が負の時、定義域エラーになります。
f の値が 0.0 の時、範囲エラーになります。

常用対数

float log10f(float f)

説　明　浮動小数点数の 10 を底とする対数を計算します。

ヘッダ　<mathf.h>

リターン値　正常：　f は 10 を底とする対数値
異常：　定義域エラーの時は、非数を返します。

引　数　f　10 を底とする対数を求める浮動小数点数

例
```
#include  <mathf.h>
float  f, ret;
    ret=log10f(f);
```

エラー条件　f の値が負の値の時、定義域エラーになります。
f の値が 0.0 の時、範囲エラーになります。

浮動小数点数を整数部、小数部に分解

*float modff(float a, float *b)*

説　明　浮動小数点数を整数部分と小数部分に分解します。

ヘッダ　<mathf.h>

リターン値　a の小数部分

引　数　a　整数部分と小数部分に分解する浮動小数点数
b　整数部分を格納する記憶域を指すポインタ

例
```
#include  <mathf.h>
float  a, *b, ret;
    ret=modff(a, b);
```

べき乗

float powf(float x, float y)

説　明　浮動小数点数のべき乗を計算します。

ヘッダ　`<mathf.h>`

リターン値　正常：　x の y 乗の値
異常：　定義域エラーの時は、非数を返します。

引　数

x	べき乗される値
y	べき乗する値

例

```
#include  <mathf.h>
float  x, y, ret;
    ret=powf(x, y);
```

エラー条件　x の値が 0.0 で、かつ y の値が 0.0 以下の時、あるいは x の値が負で y の値が整数値でない時、定義域エラーになります。

平方根

float sqrtf(float f)

説　明　浮動小数点数の正の平方根を計算します。

ヘッダ　`<mathf.h>`

リターン値　正常：　f の正の平方根の値
異常：　定義域エラーの時は、非数を返します。

引　数

f	正の平方根を求める浮動小数点数

例

```
#include  <mathf.h>
float  f, ret;
    ret=sqrtf(f);
```

エラー条件　f の値が負の値の時、定義域エラーになります。

切り上げ

float ceilf(float f)

説　明　浮動小数点数の小数点以下を切り上げた整数値を求めます。

ヘッダ　<mathf.h>

リターン値　f の小数点以下を切り上げた整数値

引　数　f　小数点以下を切り上げる浮動小数点数

例
```
#include  <mathf.h>
float  f, ret;
    ret=ceilf(f);
```

備　考　ceilf 関数は f の値より大きい、または等しい最小の整数値を float 型の値として返す関数です。
したがって f の値が負の値の時は小数点以下を切り捨てた時の値を返します。

絶対値

float fabsf(float f)

説　明　浮動小数点数の絶対値を計算します。

ヘッダ　<mathf.h>

リターン値　f の絶対値

引　数　f　絶対値を求める浮動小数点数

例
```
#include  <mathf.h>
float  f, ret;
    ret=fabsf(f);
```

切り捨て

float floorf(float f)

説　明　浮動小数点数の小数点以下を切り捨てた整数値を求めます。

ヘッダ　<mathf.h>

リターン値　f の小数点以下を切り捨てた整数値

引　数　f　小数点以下を切り捨てる浮動小数点数

例
```
#include  <mathf.h>
float  f, ret;
    ret=floorf(f);
```

備　考　floorf 関数は f の値を超えない範囲の整数の最大値を、float 型の値として返す関数です。
したがって f の値が負の値の時は小数点以下を切り上げた時の値を返します。

余り

float fmodf(float x, float y)

説　明　浮動小数点数どうしを除算した結果の余りを計算します。

ヘッダ　<mathf.h>

リターン値　y の値が 0.0 の時：x
y の値が 0.0 でない時：x を y で除算した結果の余り

引　数　x　被除数
y　除数

例
```
#include  <mathf.h>
float  x, y, ret;
    ret=fmodf(x, y);
```

備　考　fmodf 関数では、パラメタ x, y、リターン値 ret の間には、次に示す関係が成立します。
x=y*i+ret（ただし i は整数値）
また、リターン値 ret の符号は x の符号と同じ符号になります。
x/y の商を表現できない場合、結果の値は、保証されません。

(10) ＜setjmp.h＞

関数間の制御の移動をサポートします。

以下のマクロは、処理系定義です。

種別	定義名	説明
型（マクロ）	jmp_buf	関数間の制御の移動を可能とする情報を保存しておくための記憶域に対応する型名です。
関数	setjmp	現在実行中の関数の jmp_buf で定義した実行環境を指定した記憶域に退避します。
	longjmp	setjmp 関数で退避していた関数の実行環境を回復し、setjmp 関数を呼び出したプログラムの位置に制御を移動します。

setjmp 関数は現在の関数の実行環境を退避します。その後 longjmp 関数を呼び出すことにより、setjmp 関数を呼び出したプログラム上の位置にもどることができます。

以下に setjmp、longjmp 関数を使用して関数間の制御の移動をサポートした例を示します。

```
 1  #include <stdio.h>
 2  #include <setjmp.h>
 3  jmp_buf env;
 4  void main( )
 5  {
 6
 7
 8         if (setjmp(env)!=0){
 9                printf("return from longjmp\n");
10                exit(0);
11         }
12         sub( );
13  }
14
15  void sub( )
16  {
17         printf("subroutine is running \n");
18         longjmp(env, 1);
19  }
```

【説明】

8 行目で setjmp 関数を呼んでいます。この時、setjmp 関数の呼び出された環境を、jmp_buf 型の変数 env に退避します。この時のリターン値は 0 なので、次に関数 sub が呼び出されます。

関数 sub の中で呼び出される longjmp 関数により、変数 env に退避した環境を回復します。その結果、プログラムは、あたかも 8 行目の setjmp 関数からリターンしたかのようにふるまいます。ただし、この時のリターン値は longjmp 関数の第 2 パラメタで指定した値（1）になります。
その結果、9 行目以降が実行されます。

大域 goto の飛び先設定

int setjmp(jmp_buf env)

説　明　現在実行中の関数の実行環境を、指定した記憶域に退避します。

ヘッダ　<setjmp.h>

リターン値　setjmp 関数を呼び出した時：0
longjmp 関数からのリターン時：0 以外

引　数　env　実行環境を退避する記憶域へのポインタ

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

備　考　setjmp 関数により退避された実行環境は、longjmp 関数において使用されます。
setjmp 関数として呼び出された時のリターン値は 0 ですが、longjmp 関数からリターンしてきた時のリターン値は、longjmp 関数で指定した第 2 パラメタの値となります。
setjmp 関数を複雑な式から呼び出す場合、式の評価の途中結果等の現在の実行環境の一部が失われる可能性があります。setjmp 関数は setjmp 関数の結果と定数式の比較という形態だけで使用し、複雑な式の中では呼び出さないようにしてください。

大域 goto

void longjmp(jmp_buf env, int ret)

説　明　setjmp 関数で退避していた関数の実行環境を回復し、setjmp 関数を呼び出したプログラムの位置に制御を移動します。

ヘッダ　<setjmp.h>

引　数　env　実行環境を退避した記憶域へのポインタ
ret　setjmp 関数へのリターンコード

例
```
#include  <setjmp.h>
int  ret;
jmp_buf  env;
    longjmp(env, ret);
```

備　考　longjmp 関数は同じプログラム中で最後に呼び出された setjmp 関数によって退避された関数の実行環境を env で指定された記憶域から回復し、その setjmp 関数を呼び出したプログラムの位置に制御を移します。この時 longjmp 関数の ret が setjmp 関数のリターン値として返ります。ただし、ret が 0 の時は setjmp 関数へのリターン値としては 1 が返ります。
setjmp 関数が呼び出されていない時、あるいは setjmp 関数を呼び出した関数がすでに return 文を実行している時は、longjmp 関数の動作は保証されません。

(11) ＜stdarg.h＞

可変個の引数を持つ関数に対し、その引数の参照を可能にします。

以下はすべて処理系定義です。

種別	定義名	説明
型(マクロ)	va_list	可変個の引数を参照するために、va_start, va_arg, va_end マクロで共通に使用される変数の型です。
間数(マクロ)	va_start	可変個の引数の参照を行うため、初期設定処理を行います。
	va_arg	可変個の引数を持つ関数に対して、現在参照中引数の次の引数への参照を可能とします。
	va_end	可変個の引数を持つ関数の引数への参照を終了させます。

本標準インクルードファイルで定義しているマクロを使用したプログラムの例を以下に示します。

```
 1  #include <stdio.h>
 2  #include <stdarg.h>
 3
 4  extern void prlist(int count, ...);
 5
 6  void main( )
 7  {
 8         prlist(1, 1);
 9         prlist(3, 4, 5, 6);
10         prlist(5, 1, 2, 3, 4, 5);
11  }
12
13  void prlist(int count, ...)
14  {
15         va_list ap;
16         int i;
17
18         va_start(ap, count);
19         for(i=0; i<count; i++)
20                printf("%d", va_arg(ap, int));
21         putchar('\n');
22         va_end(ap);
23  }
```

【説明】

この例では、第 1 引数に出力するデータの数を指定し、以下の引数をその数だけ出力する関数 prlist を実現しています。

18 行目で、可変個の引数への参照を va_start で初期化します。その後引数を一つ出力するたびに、va_arg マクロによって次の引数を参照します（20 行目）。va_arg マクロでは、引数の型名（この場合は int 型）を第 2 引数に指定します。

引数の参照が終了したら、va_end マクロを呼び出します（22 行目）。

可変個引数取り出し開始

void va_start(va_list ap, parmN)

説　明　可変個のパラメタへの参照を行うため、初期設定処理を行います。

ヘッダ　<stdarg.h>

引　数

ap	可変個のパラメタにアクセスするための変数
parmN	最右端の引数の識別子

例

```
#include  <stdarg.h>
void va_list(int count, ...)
{
    va_list  ap;
       va_start(ap, count);
}
```

備　考　va_start マクロは、va_arg, va_end マクロによって使用される ap の初期化を行います。
また、parmN には、外部関数定義におけるパラメタの並びの最右端のパラメタの識別子、すなわち「, ...」の直前の識別子を指定します。
可変個の名前のない引数を参照するためには、va_start マクロ呼び出しを一番初めに実行する必要があります。

可変個引数取り出し

type va_arg(va_list ap, type)

説　明　可変個のパラメタを持つ関数に対して、現在参照中のパラメタの次のパラメタへの参照を可能とします。

ヘッダ　<stdarg.h>

リターン値　パラメタの値

引　数

ap	可変個のパラメタにアクセスするための変数
type	アクセスするパラメタの型

例

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

備　考　va_start マクロで初期化した va_list 型の変数を第 1 パラメタに指定します。
ap の値は va_arg を使用する度に更新され、結果として可変個のパラメタが順次本マクロのリターン値として返されます。
呼び出し手順の type のところには、参照する引数の型を指定してください。
ap は va_start によって初期化された ap と同じでなければなりません。
type の型が char 型、unsigned char 型、short 型、unsigned short 型、float 型を関数の引数に指定した時に型変換によってサイズが変わる型を指定した場合、正しくパラメタを参照することができなくなります。このような型を指定すると動作は保証されません。

可変個引数取り出し終了

void va_end(va_list ap)

説　明　　可変個の引数を持つ関数の引数への参照を終了させます。

ヘッダ　　<stdarg.h>

引　数　　ap　　可変個の引数を参照するための変数

例

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

備　考　　ap は va_start によって初期化された ap と同じでなければなりません。
また、関数からの return 前に va_end マクロが呼び出されない時は、その関数の動作は保証されません。

(12) ＜stdio.h＞

ストリーム入出力用ファイルの入出力に関する処理を行います。

以下の定数（マクロ）はすべて処理系定義です。

種別	定義名	説明
定数（マクロ）	FILE	ストリーム入出力処理で必要とするバッファへのポインタやエラー指示子、終了指示子などの各種制御情報を保存しておく構造体の型です。
	_IOFBF	バッファ領域の使用方法として、入出力処理はすべてバッファ領域を使用することを示しています。
	_IOLBF	バッファ領域の使用方法として、入出力処理は行単位でバッファ領域を使用することを示しています。
	_IONBF	バッファ領域の使用方法として、入出力処理はバッファ領域を使用しないことを示しています。
	BUFSIZ	入出力処理において必要とするバッファの大きさです。
	EOF	ファイルの終わり（end_of_file）すなわちファイルからそれ以上の入力が無いことを示しています。
	L_tmpnam *1	tmpnam 関数によって生成される一時ファイル名の文字列を格納するのに十分な大きさの配列のサイズです。
	SEEK_CUR	ファイルの現在の読み書き位置を現在の位置からのオフセットに移すことを示しています。
	SEEK_END	ファイルの現在の読み書き位置をファイルの終了位置からのオフセットに移すことを示しています。
	SEEK_SET	ファイルの現在の読み書き位置をファイルの先頭位置からのオフセットに移すことを示しています。
	SYS_OPEN *1	処理系が同時にオープンすることができることを保証するファイルの数です。
	TMP_MAX *1	tmpnam 関数によって生成される一意なファイル名の個数の最小値です。
	stderr	標準エラーファイルに対するファイルポインタです。
	stdin	標準入力ファイルに対するファイルポインタです。
	stdout	標準出力ファイルに対するファイルポインタです。
関数	fclose	ストリーム入出力用ファイルをクローズします。
	fflush	ストリーム入出力用ファイルのバッファの内容をファイルへ出力します。
	fopen	ストリーム入出力用ファイルを指定したファイル名によってオープンします。
	freopen	現在オープンされているストリーム入力出用ファイルをクローズし、新しいファイルを指定したファイル名で再オープンします。
	setbuf	ストリーム入出力用のバッファ領域をユーザプログラム側で定義して設定します。
	setvbuf	ストリーム入出力用のバッファ領域をユーザプログラム側で定義して設定します。
	fprintf	書式に従ってストリーム入出力用ファイルへデータを出力します。
	fscanf	ストリーム入出力用ファイルからデータを入力し、書式に従って変換します。

【注】 *1 本処理系では、定義されません。

種別	定義名	説明
関数	printf	データを書式に従って変換し、標準出力ファイル（stdout）へ出力します。
	scanf	標準入力ファイル（stdin）からデータを入力し、書式に従って変換します。
	sprintf	データを書式に従って変換し、指定した領域へ出力します。
	sscanf	指定した記憶域からデータを入力し、書式に従って変換します。
	vfprintf	可変個のパラメタリストを書式に従って指定したストリーム入出力用ファイルに出力します。
	vprintf	可変個のパラメタリストを書式に従って標準出力ファイルに出力します。
	vsprintf	可変個のパラメタリストを書式に従って指定した記憶域に出力します。
	fgetc	ストリーム入出力用ファイルから 1 文字入力します。
	fgets	ストリーム入出力用ファイルから文字列を入力します。
	fputc	ストリーム入出力用ファイルへ 1 文字出力します。
	fputs	ストリーム入出力用ファイルへ文字列を出力します。
	getc	（マクロ） ストリーム入出力用ファイルから 1 文字入力します。
	getchar	（マクロ） 標準入力ファイルから 1 文字入力します。
	gets	標準入力ファイルから文字列を入力します。
	putc	（マクロ） ストリーム入出力用ファイルへ 1 文字出力します。
	putchar	（マクロ） 標準出力ファイルへ 1 文字出力します。
	puts	標準出力ファイルへ文字列を出力します。
	ungetc	ストリーム入出力用ファイルへ 1 文字をもどします。
	fread	ストリーム入出力用ファイルから指定した記憶域にデータを入力します。
	fwrite	記憶域からストリーム入出力用ファイルにデータを出力します。
	fseek	ストリーム入出力用ファイルの現在の読み書き位置を移動させます。
	ftell	ストリーム入出力用ファイルの現在の読み書き位置を求めます。
	rewind	ストリーム入出力用ファイルの現在の読み書き位置をファイルの先頭に移動します。
	clearerr	ストリーム入出力用ファイルのエラー状態をクリアします。
	feof	ストリーム入出力用ファイルが終わりであるかどうかを判定します。
	ferror	ストリーム入出力用ファイルがエラー状態であるかどうかを判定します。
	perror	標準エラーファイル(stderr)に、エラー番号に対応したエラーメッセージを出力します。

処理系定義仕様

	項目	
1	入力テキストの最終の行が終了を示す改行文字を必要とするかどうか	規定しません。低水準インターフェースルーチンの仕様によります。
2	改行文字の直前にかき出された空白文字は、読み込み時に読み込まれるかどうか	
3	バイナリファイルに書かれたデータに付加されるヌル文字の数	
4	追加モード時のファイル位置指定子の初期値	
5	テキストファイルへの出力によってそれ以降のファイルのデータが失われるかどうか	
6	ファイルバッファリングの仕様	
7	長さ 0 のファイルが存在するかどうか	
8	正当なファイル名の構成規則	
9	同時に同じファイルをオープンできるかどうか	
10	fprintf 関数における%p 書式変換の出力形式	16 進数出力となります
11	fscanf 関数における%p 書式変換の入力形式 fscanf 関数での変換文字「－」の意味	16 進数出力となります。 先頭、最後あるいは「^」の直後でない場合、直前の文字と直後の範囲を示します。
12	fgetpos,ftell 関数で設定される errno の値	fgetpos 関数はサポートしていません。 ftell 関数については規定しません。 低水準インタフェースルーチンの仕様によります。
13	perror 関数が生成するメッセージ出力形式	メッセージの出力形式を(a)に示します。
14	calloc,malloc,realloc 関数でサイズが 0 のときの動作	0 バイトの領域を割り付けます。

(a) perror 関数の出力形式は、

<文字列> ： <error に設定したエラー番号に対応するエラーメッセージ>

となります。

(b) printf 関数、fprintf 関数で、浮動小数点の無限大および非数を表示するときの形式を表 10.34 に示します。

表 10.34　無限大および非数の表示形式

	値	表示形式
1	正の無限大	＋＋＋＋＋＋
2	負の無限大	－－－－－－
3	非数	＊＊＊＊＊＊

ストリーム入出力用ファイルに対する一連の入出力処理を行ったプログラムの例を以下に示します。

```
 1  #include <stdio.h>
 2
 3  void main( )
 4  {
 5         int c;
 6         FILE *ifp, *ofp;
 7
 8         if ((ifp=fopen("INPUT.DAT","r"))==NULL){
 9                fprintf(stderr,"cannot open input file\n");
10                exit(1);
11         }
12         if ((ofp=fopen("OUTPUT.DAT","w"))==NULL){
13                fprintf(stderr,"cannot open output file\n");
14                exit(1);
15         }
16         while ((c=getc(ifp))!=EOF)
17                putc(c, ofp);
18         fclose(ifp);
19         fclose(ofp);
20  }
```

【説明】

ファイル INPUT.DAT の内容をファイル OUTPUT.DAT へコピーするプログラムです。

8 行目の fopen 関数で入力ファイル INPUT.DAT を、12 行目の fopen 関数で出力ファイル OUTPUT.DAT をオープンします。オープンに失敗した場合、fopen 関数のリターン値として NULL が返されますので、エラーメッセージを出力してプログラムを終了させます。

fopen 関数が正常に終了した時、オープンしたファイルの情報を格納するデータ（FILE 型）へのポインタが返されますので、これらを変数 ifp、ofp に設定します。

オープンが成功した後は、これらの FILE 型のデータを用いて入出力を行います。

ファイルの処理が終了したら、fclose 関数でファイルをクローズします。

ファイルクローズ

*int fclose(FILE *fp)*

説　明　ストリーム入出力用ファイルをクローズします。

ヘッダ　<stdio.h>

リターン値　正常：　0
異常：　0 以外

引　数　fp　ファイルポインタ

例
```
#include  <stdio.h>
FILE  *fp;
int  ret;
    ret=fclose(fp);
```

備　考　fclose 関数はファイルポインタ fp の示すストリーム入出力用ファイルをクローズします。
fclose 関数は、ストリーム入出力用ファイルの出力ファイルがオープンされており、まだ出力されていないデータがバッファに残っている時は、それをファイルに出力してからクローズします。
また、入出力用のバッファがシステムによって自動的に割り付けられていた場合は、それを解放します。

バッファフラッシュ

*int fflush(FILE *fp)*

説　明　ストリーム入出力用ファイルのバッファの内容をファイルへ出力します。

ヘッダ　<stdio.h>

リターン値　正常：　0
異常：　0 以外

引　数　fp　ファイルポインタ

例
```
#include  <stdio.h>
FILE  *fp;
int  ret;
    ret=fflush(fp);
```

備　考　fflush 関数はストリーム入出力用ファイルの出力ファイルがオープンされている時、ファイルポインタ fp で指定されたストリーム入出力用ファイルのバッファの未出力内容をファイルに出力します。また、入力ファイルがオープンされている時、ungetc 関数の指定を無効にします。

ファイルオープン

*FILE *fopen(const char *fname, const char *mode)*

説　明　ストリーム入出力用ファイルを、指定したファイル名によってオープンします。

ヘッダ　<stdio.h>

リターン値　正常：　オープンしたファイルのファイル情報を指すファイルポインタ
異常：　NULL

引　数

fname	ファイル名を示す文字列へのポインタ
mode	ファイルアクセスモードを示す文字列へのポインタ

例

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

備　考　fopen関数はfnameが指す文字列をファイル名とするストリーム入出力用ファイルをオープンします。書き出しモードあるいは追加モードで存在しないファイルをオープンしようとした時は、可能な限り新しいファイルを作成します。また既存のファイルに対して書き出しモードでオープンした時は、ファイルの先頭から書き込みが行われ、以前に書き込まれていたファイルの内容は消去されます。
追加モードでオープンしたファイルは、そのファイルの終わりの位置から書き出しの処理が行われます。更新モードでオープンしたファイルは、このファイルに対して入力と出力の両方の処理を行うことができます。
ただし、出力処理は後に fflush，fseek，rewind 関数が実行されることなしに入力処理を続けることはできません。
また同様に入力処理の後に fflush，fseek，rewind 関数が実行されることなしに出力処理を続けることはできません。
また、ファイルアクセスモードを示す文字列の後にオープンの方法を指示する文字が付くこともあります。

ファイル再オープン

*FILE *freopen(const char *fname, const char *mode, FILE *fp)*

説　明　現在オープンされているストリーム入出力用ファイルをクローズし、新しいファイルを指定したファイル名で再オープンします。

ヘッダ　<stdio.h>

リターン値　正常：　fp
異常：　NULL

引　数

fname	新しいファイル名を示す文字列へのポインタ
mode	ファイルアクセスモードを示す文字列へのポインタ
fp	現在オープンされているストリーム入出力用ファイルのファイルポインタ

例

```
#include  <stdio.h>
const char  *fname, *mode;
FILE  *ret, *fp;
    ret=freopen(fname, mode, fp);
```

備　考　freopen 関数は、まず、ファイルポインタ fp の示すストリーム入出力用ファイルをクローズします（このクローズ処理が正しく行われない時でも以下の処理は続けます。）。
次に、その fp の指す FILE 構造体を再使用して、ファイル名 fname で示すファイルを、ストリーム入出力用にオープンします。
freopen 関数は一時にオープンするファイル数の限られているときなどに有効です。
freopen 関数は通常、fp と同じ値を返しますが、エラーが発生した時は、NULL を返します。

バッファ領域設定

*void setbuf(FILE *fp, char buf[BUFSIZ])*

説　明　ストリーム入出力用のバッファ領域をユーザプログラム側で定義して設定します。

ヘッダ　`<stdio.h>`

引　数

`fp`	ファイルポインタ
`buf`	バッファ領域へのポインタ

例

```
#include  <stdio.h>
FILE  *fp;
char  buf[BUFSIZ];
    setbuf(fp, buf);
```

備　考　`setbuf` 関数は、ファイルポインタ `fp` の示すストリーム入出力用ファイルに対して `buf` の指す記憶域を入出力用のバッファ領域として使用するように定義します。この結果、大きさが `BUFSIZ` のバッファ領域を使用した入出力処理が行われます。

バッファリング制御

*int setvbuf(FILE *fp, char *buf, int type, size_t size)*

説　明　ストリーム入出力用のバッファ領域をユーザプログラムの側で定義して設定します。

ヘッダ　<stdio.h>

リターン値　正常：　0
異常：　0 以外

引　数

fp	ファイルポインタ
buf	バッファ領域へのポインタ
type	バッファの管理方式
size	バッファ領域の大きさ

例

```
#include  <stdio.h>
FILE  *fp;
char  *buf;
int  type, ret;
size_t  size;
    ret=setvbuf(fp, buf, type, size);
```

備　考　setvbuf 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルに対して buf の指す記憶域を入出力用のバッファ領域として使用するように定義します。
このバッファ領域の使用方法としては、以下の三通りの方法があります。
(a) type に_IOFBF を指定した時
入出力処理はすべてバッファ領域を使用して行います。
(b) type に_IOLBF を指定した時
入出力処理は行単位でバッファ領域を使用して行います。すなわち、入出力データは、改行文字が書かれた時、バッファ領域が一杯になった時、入力が要求された時にバッファ領域から取り出されることになります。
(c) type に_IONBF を指定した時
入出力処理はバッファ領域を使用せず行います。
setvbuf 関数は通常 0 を返しますが、type あるいは size に不正な値が与えられた時、あるいはバッファ領域の使用方法等の要求が受け入れられなかった時には 0 以外の値を返します。
バッファ領域は、オープンされているストリーム入出力用ファイルがクローズされる前に解放してはいけません。また、setvbuf 関数は、ストリーム入出力用ファイルがオープンされてから入出力用処理が行われるまでの間で使用してください。

書式付きファイル出力

*int fprintf(FILE *fp, const char *control [, arg] …)*

説　明　書式に従って、ストリーム入出力用ファイルへデータを出力します。

ヘッダ　<stdio.h>

リターン値　正常：　変換し出力した文字数
異常：　負の値

引　数

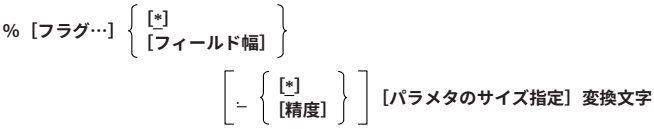
fp	ファイルポインタ
control	書式を示す文字列へのポインタ
arg, …	書式に従って出力されるデータの並び

例

```
#include  <stdio.h>
FILE  *fp;
const char  *control="%s";
int  ret;
char buffer[]="Hellow World\n";
    ret=fprintf(fp, control, buffer);
```

備　考　fprintf 関数は、control が指す書式を示す文字列に従って、引数 arg を変換、編集し、ファイルポインタ fp の示すストリーム入出力用ファイルへ出力します。
fprintf 関数は、通常は変換し出力したデータの個数を返しますが、エラー発生時には負の値を返します。
書式の仕様は以下のとおりです。

【書式の概要】
書式を表わす文字列は、2 種類の文字列から構成されます。
・通常の文字
次の変換仕様を示す文字列以外の文字はそのまま出力されます。
・変換仕様
変換仕様は、%で始まる文字列で、後に続く引数の変換方法を指定します。変換仕様の形式は次の規則に従います。



この変換仕様に対して、実際に出力するパラメタが無い時は、その動作は保証されません。また、変換仕様よりも実際に出力するパラメタの個数が多い時は、余分なパラメタはすべて無視されます。

【変換仕様の説明】
(a) フラグ
符号を付けるなどの出力するデータに対する修飾を指定します。指定できるフラグの種類と意味を**表** 10.35 に示します。

表 10.35 **フラグの種類と意味**

	項目	
1	－	**変換したデータの文字数が指定したフィールド幅より少ない時、そのデータをフィールド内で左詰めにして出力します。**
2	＋	**符号付きのデータに変換する時、そのデータの符号に従って、変換したデータの先頭にプラスあるいはマイナス符号を付けます。**
3	空白	**符号付きのデータの変換において、変換したデータの先頭に符号が付かない時、そのデータの先頭に空白を付けます。** **「＋」と共に使用した時、本フラグは無視されます。**
4	#	**表** 10.37 **で説明する変換の種類に従って、変換後のデータに修飾を行います。** 1. c，d，i，s，u **変換の時** **本フラグは無視されます。** 2. o **変換の時** **変換したデータの先頭に** 0 **を付けます。** 3. x**（あるいは** X**）変換の時** **変換したデータの先頭に** 0x**（あるいは** 0X**）を付けます。** 4. e，E，f，g，G **変換の時** **変換したデータに小数点以下がない時でも、小数点を出力します。また、**g，G **変換の時は、変換後のデータの後に付く** 0 **は取り除きません。**

(b) フィールド幅
変換したデータを出力する文字数を任意の 10 進数で指定します。
変換したデータの文字数がフィールド幅より少ない時、フィールド幅までそのデータの先頭に空白が付けられます。（ただし、フラグとして‘－’を指定した時は、データの後に空白が付けられます。）
もし、変換したデータの文字数がフィールド幅より大きい時は、フィールド幅は、変換結果を出力できる幅に拡張されます。
また、フィールド幅指定の先頭が 0 で始まっている時は、出力するデータの先頭には空白ではなく文字「0」が付けられます。

(c) 精度
表 10.37 で説明する変換の種類に従って変換したデータの精度を指定します。
精度は、ピリオド（.）の後に 10 進整数を続ける形式で指定します。10 進整数を省略した時は、0 を指定したものと仮定します。
精度を指定した結果、フィールド幅の指定との間に矛盾が生じれば、フィールド幅の指定を無効とします。
各変換の種類と精度指定の意味を以下に示します。
・d, i, o, u, x, X 変換の時
　変換したデータの最小の桁数を示します。
・e, E, f 変換の時
　変換したデータの小数点以下の桁数を示します。
・g, G 変換の時
　変換したデータの最大有効桁数を示します。
・s 変換の時
　印字される最大文字数を示します。

(d) パラメタのサイズ指定
d, i, o, u, x, X, e, E, f, g, G 変換の時**（表** 10.37 **参照）**
変換するデータのサイズ（short 型、long 型、long double 型）を指定します。これ以外の変換の時は、本指定を無視します。**表** 10.36 にサイズ指定の種類とその意味を示します。

表 10.36 **パラメタのサイズ指定の種類とその意味**

	種類	意味
1	h	d，i，o，u，x，X **変換において、変換するデータが** short **型あるいは** unsigned short **型であることを指定します。**
2	l	d，i，o，u，x，X **変換において、変換するデータが** long **型、**unsigned long **型あるいは、**double **型であることを指定します。**
3	L	e，E，f，g，G **変換において、変換するデータが** long double **型であることを指定します。**

(e) 変換文字

変換するデータをどのような形式に変換するかを指定します。

もし、変換するデータが構造体や配列型の時や、それらの型を指すポインタの時は、s 変換で文字の配列を変換する時、p 変換でポインタを変換する時を除いてその動作は保証されません。**表** 10.37 に変換文字と変換方式を示します。この表に述べられていない英文字を変換文字として指定した時は、その動作は保証されません。また、それ以外の文字を指定した時の動作はコンパイラによって異なります。

表 10.37 変換文字と変換の方式

	変換文字	変換の種類	変換の方式	変換の対象とするデータの型	精度に対する注意事項
1	d	d 変換	int 型データを符号付き 10 進数の文字列に変換します。d 変換と i 変換は同じ仕様です。	int 型	精度指定は、最低で何文字出力されるかを示しています。もし、変換後の文字数が精度の値より少ない時は、文字列の先頭に 0 が付きます。また、精度を省略した時は、1 が仮定されます。さらに、0 の値を持つデータを精度に 0 を指定して変換し出力しようとした時は、何も出力されません。
2	i	i 変換		int 型	
3	o	o 変換	int 型データを符号無しの 8 進数の文字列に変換します。	int 型	
4	u	u 変換	int 型データを符号無しの 10 進数の文字列に変換します。	int 型	
5	x	x 変換	int 型データを符号無しの 16 進数に変換します。16 進文字には a，b，c，d，e，f を用います。	int 型	
6	X	X 変換	int 型データを符号無しの 16 進数に変換します。16 進文字には A，B，C，D，E，F を用います。	int 型	
7	f	f 変換	double 型データを「［−］ddd.ddd」の形式の 10 進数の文字列に変換します。	double 型	精度の指定は、小数点以降の桁数を表わします。小数点以降の文字が存在する時には、必ず小数点の前に 1 桁の数字が出力されます。精度を省略した時は、6 が仮定されます。また、精度に 0 を指定した時は、小数点と小数点以降の文字は出力しません。出力するデータは丸められます。
8	e	e 変換	double 型データを「［−］d.ddde±dd」の形式の 10 進数の文字列に変換します。指数は、少なくとも 2 桁出力されます。	double 型	精度の指定は、小数点以降の桁数を表わします。変換した文字は小数点の前に 1 桁の数字が出力され、小数点以降に精度に等しい桁数の数字が出力される形式となります。精度を省略した時は 6 が仮定されます。また、精度に 0 を指定した時は、小数点以降の文字は出力しません。出力するデータは丸められます。
9	E	E 変換	double 型データを「［−］d.dddE±dd」の形式の 10 進数の文字列に変換します。指数は、少なくとも 2 桁出力されます。	double 型	
10	g	g 変換（あるいは G 変換）	変換する値と有効桁数を指定する精度の値から f 変換の形式で出力するか e 変換（あるいは E 変換）の形式で出力するかを決め double 型データを出力します。もし、変換されたデータの指数が−4 より小さいか、有効桁数を指定する精度より大きい時には e 変換（あるいは E 変換）の形式に変換します。	double 型	精度の指定は、変換されたデータの最大有効桁数を示します。
11	G			double 型	
12	c	c 変換	int 型のデータを unsigned char 型データとし、そのデータに対応する文字に変換します。	int 型	精度の指定は無効です。

	変換文字	変換の種類	変換の方式	変換の対象とするデータの型	精度に対する注意事項
13	s	s 変換	char 型へのポインタ型データが指す文字列を文字列の終了を示すヌル文字まで、あるいは、精度で指定された文字数分出力します。（ただしヌル文字は出力されません。また、空白、水平タブ、改行文字は変換文字列に含まれません。）	char 型へのポインタ型	精度の指定は出力する文字数を示します。もし、精度が省略された時は、データが指す文字列のヌル文字までの文字が出力されます。（ただし、ヌル文字は出力されません。また、空白、水平タブ、改行文字は変換文字列に含まれません。）
14	p	p 変換	データをポインタとして、コンパイラごとに定義された印字可能な文字列に変換します。	void 型へのポインタ	精度の指定は無効です。
15	n	データの変換は生じません。	データは int 型へのポインタ型とみなされ、このデータが指す記憶域にいままで、出力したデータの文字数を設定します。	int 型へのポインタ型	
16	%	この変換ではデータの変換は生じません。	%を出力します。	無し	

(f) フィールド幅あるいは精度に対する*指定
フィールド幅あるいは精度指定の値として*を指定することができます。この時は、この変換仕様に対応するパラメタの値がフィールド幅あるいは精度指定の値として使用されます。このパラメタが負のフィールド幅を持つ時は、正のフィールド幅にフラグの－が指定されたと解釈します。また、負の精度を持つ時は、精度が省略されたものと解釈します。

書式付きファイル入力

*int fscanf(FILE *fp, const char *control [, ptr] ･･･)*

説　明　ストリーム入出力用ファイルからデータを入力し、書式に従って変換します。

ヘッダ　<stdio.h>

リターン値　正常：　入力変換に成功したデータの個数
異常：　入力データの変換を行う前に入力データが終了した時：EOF

引　数

fp	ファイルポインタ
control	書式を示す文字列へのポインタ
ptr	入力したデータを格納する記憶域へのポインタ

例

```
#include  <stdio.h>
FILE  *fp;
const char  *control="%d";
int  ret,buffer[10];
    ret=fscanf(fp, control, buffer);
```

備　考　fscanf 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルからデータを入力し、control が指す書式を文字列に従って変換、編集して、その結果を ptr の指す記憶域へ格納します。データを入力するための書式の仕様を以下に示します。

【書式の概要】
書式を表わす文字列は、以下の 3 種類の文字列から構成されます。
・空白文字
空白（' '）水平タブ（'\t'）あるいは改行文字（'\n'）を指定すると、入力データを次の空白類文字でない文字まで読み飛ばす処理を行います。
・通常の文字
上の空白文字でも%でもない文字を指定すると、入力データを 1 文字入力します。ここで入力した文字は書式を表わす文字列の中に指定した文字と一致していなければなりません。
・変換仕様
変換仕様は、%で始まる文字列で、書式を表わす文字列の後に続く引数の指す領域に入力データを変換して格納する方法を指定します。変換仕様の形式は次の規則に従います。

%［*］［フィールド幅］［変換後のデータのサイズ］変換文字

書式中の変換仕様に対して入力したデータを格納する記憶域へのポインタがない時は、その動作は保証されません。また、書式が終了したにもかかわらず、入力データを格納する記憶域へのポインタが残っている時は、そのポインタは無視されます。

【変換仕様の説明】
・*指定
入力したデータをパラメタが指す記憶域に格納することを抑止します。
・フィールド幅
入力するデータの最大文字数を 10 進数字で指定します。
・変換後のデータのサイズ
d, i, o, u, x, X, e, E, f 変換の時**（表 10.39 参照）**、変換後のデータのサイズ（short 型、long 型、long double 型）を指定します。これ以外の変換の時は、本指定を無視します。**表** 10.38 にサイズ指定の種類とその意味を示します。

表 10.38 変換後のデータのサイズ指定の種類とその意味

	種類	意味
1	h	d，i，o，u，x，X 変換において、変換後のデータは short 型であることを指定します。
2	l	d，i，o，u，x，X 変換において、変換後のデータは long 型であることを指定します。 また、e，E，f 変換において、変換後のデータは double 型であることを指定します。
3	L	e，E，f 変換において、変換後のデータは、long double 型であることを指定します。

・変換文字

入力するデータは、各変換文字が指定する変換の種類に従って変換します。ただし、空白類文字を読み込んだ場合、変換の対象として許されていない文字を読み込んだ場合、あるいは指定されたフィールド幅を超えた場合は処理を終了します。

表 10.39 変換文字と変換の内容

	変換文字	変換の種類	変換の方式	対応するパラメタのデータ型
1	d	d 変換	10 進数字の文字列を整数型データに変換します。	整数型
2	i	i 変換	先頭に符号が付いている 10 進数字の文字列、あるいは最後に u（U）または l（L）が付いている 10 進数字の文字列を整数型データに変換します。また、先頭が 0x（あるいは 0X）で始まっている文字列は、16 進数字として解釈し、文字列を int 型データに変換します。さらに、先頭が 0 で始まっている文字列は、8 進数字として解釈し文字列を int 型データに変換します。	整数型
3	o	o 変換	8 進数字の文字列を整数型データに変換します。	整数型
4	u	u 変換	符号無しの 10 進数字の文字列を整数型データに変換します。	整数型
5	x	x 変換	16 進数字の文字列を整数型データに変換します。	整数型
6	X	X 変換	x 変換と X 変換に意味の違いはありません。	
7	s	s 変換	空白、水平タブ、改行文字を読み込むまでをひとつの文字列として変換します。文字列の最後にはヌル文字を付加します。（変換したデータを設定する文字列は、ヌル文字を含めて格納できるサイズが必要です。）	文字型
8	c	c 変換	1 文字を入力します。この時、入力する文字が空白類文字であっても読み飛ばすことはしません。もし、、空白類文字以外の文字だけを読み込む時は、%1s と指定してください。また、フィールド幅が指定されている時は、その指定分の文字が読み込まれます。したがって、この時、変換したデータを格納する記憶域は、指定分の大きさが必要です。	char 型
9	e	e 変換	浮動小数点数を示す文字列を浮動小数点型データに変換します。e 変換と E 変換、g 変換と G 変換にそれぞれ意味の違いはありません。入力形式は strtod 関数で表現できる浮動小数点数です。	浮動小数点型
10	E	E 変換		
11	f	f 変換		
12	g	g 変換		
13	G	G 変換		
14	p	p 変換	fprintf 関数において、P 変換で変換される形式の文字列をポインタ型データに変換します。	void 型へのポインタ型
15	n	データの変換は生じません。	データの入力は行わず、いままでに入力したデータの文字数が設定されます。	整数型
16	[	［変換	［の後に文字の集合、その後に］を指定します。この文字集合は、文字列を構成する文字の集合を定義しています。もし、文字集合の最初の文字が ^ でない時は、入力データはこの文字集合にない文字が最初に読み込まれるまでをひとつの文字列として入力します。もし、最初の文字が ^ の時は、 ^ を除いた文字集合の文字が最初に読み込まれるまでをひとつの文字列として入力します。入力した文字列の最後には自動的にヌル文字を付加します（変換したデータを設定する文字列は、ヌル文字を含めて格納できるサイズが必要です）。	文字型
17	%	データの変換は生じません。	%を読み込みます。	無し

変換文字が表 10.39 に示す文字以外の英文字の時は、その動作は保証されません。また、その他の文字の時は、その動作は処理系定義です。

書式付き出力

*int printf(const char *control [, arg] …)*

説　明　データを書式に従って変換し、標準出力ファイル（stdout）へ出力します。

ヘッダ　<stdio.h>

リターン値　正常：　変換し出力した文字数
異常：　負の値

引　数

control	書式を示す文字列へのポインタ
arg	書式に従って出力されるデータ

例

```
#include  <stdio.h>
char  *s;
const char  *control="%s";
int  ret;
char buffer[]="Hellow World\n";
    ret=sprintf(fp, control, buffer);
```

備　考　printf 関数は control が指す書式を示す文字列に従って、パラメタ arg を変換、編集し、標準出力ファイル（stdout）へ出力します。
書式の仕様の詳細は fprintf()を参照してください。

書式付き入力

*int scanf(const char *control [, ptr] …)*

説　明　標準入力ファイル（stdin）からデータを入力し、書式に従って変換します。

ヘッダ　<stdio.h>

リターン値　正常：　入力変換に成功したデータの個数
異常：　EOF

引　数

control	書式を示す文字列へのポインタ
ptr	入力変換したデータを格納する記憶域へのポインタ

例

```
#include  <stdio.h>
const char  *control="%d";
int  ret,buffer[10];
    ret=scanf(control, buffer);
```

備　考　scanf 関数は標準入力ファイル（stdin）からデータを入力し、control が指す書式を示す文字列に従って、そのデータを変換、編集して、その結果を ptr の指す記憶域へ格納します。
scanf 関数は、入力変換に成功したデータの個数をリターン値として返します。最初の変換の前に標準入力ファイルが終了した時には EOF を返します。
書式の仕様の詳細は fscanf()を参照してください。
%e 変換について double 型の場合は l、long double 型の場合は L で指定することになっています。デフォルトの型は float 型です。

書式付き文字列出力

*int sprintf(char *s, const char *control [, arg] …)*

説　明　データを書式に従って変換し、指定した領域へ出力します。

ヘッダ　<stdio.h>

リターン値　変換した文字数

引　数

s	データを出力する記憶域へのポインタ
control	書式を示す文字列へのポインタ
arg	書式に従って出力されるデータ

例

```
#include  <stdio.h>
char  *s;
const char  *control="%s";
int  ret;
char buffer[]="Hellow World\n";
    ret=sprintf(fp, control, buffer);
```

備　考　sprintf 関数は、control が指す書式を示す文字列に従って、パラメタ arg を変換、編集し、s の指す記憶域へ出力します。
変換して出力した文字の列の最後には、ヌル文字が付加されます。このヌル文字はリターン値である出力した文字数の中には含まれません。
書式の仕様の詳細は fprintf()を参照してください。

書式付き文字列入力

*int sscanf(const char *s, const char *control [, ptr] …)*

説　明　指定した記憶域からデータを入力し、書式に従って変換します。

ヘッダ　<stdio.h>

リターン値　正常：　入力変換に成功したデータの個数
異常：　EOF

引　数

s	入力するデータがある記憶域
control	書式を示す文字列へのポインタ
ptr	入力変換したデータを格納する記憶域へのポインタ

例

```
#include  <stdio.h>
const char  *s, *control="%d";
int  ret,buffer[10];
    ret=sscanf(s, control, buffer);
```

備　考　sscanf 関数は、s の指す記憶域からデータを入力し、control が指す書式を示す文字列に従って、そのデータを変換、編集して、その結果を ptr の指す記憶域へ格納します。
sscanf 関数は、入力変換に成功したデータの個数を返します。また、最初の変換の前に入力するデータが終了した時には EOF を返します。
書式の仕様の詳細は fscanf()を参照してください。

可変個パラメタのファイル出力

*int vfprintf(FILE *fp, const char *control, va_list arg)*

説　明　可変個のパラメタリストを書式に従って、指定したストリーム入出力用ファイルに出力します。

ヘッダ　<stdio.h>

リターン値　正常：　変換し出力した文字数
異常：　負の値

引　数

fp	ファイルポインタ
control	書式を示す文字列へのポインタ
arg	引数リスト

例

```
#include  <stdarg.h>
#include  <stdio.h>
FILE  *fp;
const char  *control="%d";
int  ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++)
     ret=vfprintf(fp, control, ap);
   va_end(ap);
}
```

備　考　vfprintf 関数は、control が指す書式を示す文字列に従って、可変個の引数リストを順に変換、編集し、fp の示すストリーム入出力用ファイルへ出力します。
vfprintf 関数は、変換し出力したデータの個数を返しますが出力エラーが発生した時は負の値を返します。
また、vfprintf 関数では va_end マクロは呼び出しません。
書式の仕様の詳細は fprintf()を参照してください。
引数リストを示す arg は、va_start および va_arg マクロによって初期化されていなければなりません。

可変個パラメタの出力

*int vprintf(const char *control, va_list arg)*

説　明　可変個のパラメタリストを書式に従って標準出力ファイル（stdout）に出力します。

ヘッダ　<stdio.h>

リターン値　正常：　変換し出力した文字数
異常：　負の値

引　数

control	書式を示す文字列へのポインタ
arg	引数リスト

例

```
#include  <stdarg.h>
#include  <stdio.h>
FILE  *fp;
const char  *control="%d";
int  ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++)
      ret=vprintf(control, ap);
   va_end(ap);
}
```

備　考　vprintf 関数は、control が指す書式を示す文字列に従って、可変個のパラメタリストを順に変換、編集し、標準出力ファイルへ出力します。
vprintf 関数は、変換し出力したデータの個数を返しますが出力エラーが発生した時は負の値を返します。
また、vprintf 関数では va_end マクロは呼び出しません。
書式の仕様の詳細は fprintf()を参照してください。
引数リストを示す arg は、va_start および va_arg マクロによって初期化されていなければなりません。

可変個パラメタの文字列出力

*int vsprintf(char *s, const char *control, va_list arg)*

説　明　可変個のパラメタリストを書式に従って、指定した記憶域に出力します。

ヘッダ　<stdio.h>

リターン値　正常：　変換した文字数
異常：　負の数

引　数

s	データを出力する記憶域へのポインタ
control	書式を示す文字列へのポインタ
arg	引数リスト

例

```
#include  <stdarg.h>
#include  <stdio.h>
FILE  *fp;
const char  *control="%d";
int  ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++)
      ret=vsprintf(s, control, ap);
   va_end(ap);
}
```

備　考　vsprintf 関数は、control が指す書式を示す文字列に従って、可変個の引数リストを順に変換、編集し、s により指される記憶域へ出力します。
変換して出力した文字列の最後にはヌル文字が付加されます。このヌル文字はリターン値である出力した文字数の中には含まれません。
書式の仕様の詳細は fprintf()を参照してください。
引数リストを示す arg は、va_start および va_arg マクロによって初期化されていなければなりません。

ファイルから 1 文字入力

*int fgetc(FILE *fp)*

説　明　ストリーム入出力用ファイルから 1 文字入力します。

ヘッダ　<stdio.h>

リターン値　正常：　ファイルの終了の時：EOF
　　ファイルの終了でない時：入力した文字
　　異常：　EOF

引　数　fp　　ファイルポインタ

例
```
#include  <stdio.h>
FILE  *fp;
int  ret;
    ret=fgetc(fp);
```

エラー条件　読み込みエラーが発生した時、そのファイルに対してのエラー指示子が設定されます。

備　考　fgetc 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルから 1 文字入力します。
fgetc 関数は、通常入力した 1 文字を返しますが、ファイルの終了やエラー発生の時は、EOF を返します。また、ファイルの終了の時には、そのファイルに対するファイル終了指示子が設定されます。

ファイルから文字列入力

*char *fgets(char *s, int n, FILE *fp)*

説　明　ストリーム入出力用ファイルから文字列を入力します。

ヘッダ　<stdio.h>

リターン値　正常：　ファイルの終了の時：NULL
　　ファイルの終了でない時：s
　　異常：　NULL

引　数

s	文字列を入力する記憶域へのポインタ
n	文字列を入力する記憶域のバイト数
fp	ファイルポインタ

例

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

備　考　fgets 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルから、ポインタ s の指す記憶域に文字列を入力します。
fgets 関数は、n-1 文字まであるいは改行文字を入力するまで、またはファイルの終わりになるまで文字を入力し、入力文字列の最後にヌル文字を付け加えます。
fgets 関数は通常、文字列を入力する記憶域へのポインタ s を返しますが、ファイルが終了した時やエラー発生の時は NULL を返します。
ファイルが終了した時は、s が指す記憶域の内容は変化しませんが、エラー発生の時は、s が指す記憶域の内容は保証されません。

ファイルに 1 文字出力

*int fputc(int c, FILE *fp)*

説　明　ストリーム入出力用ファイルへ 1 文字出力します。

ヘッダ　<stdio.h>

リターン値　正常：　出力した文字
異常：　EOF

引　数

c	出力する文字
fp	ファイルポインタ

例

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

エラー条件　書き出しエラーが発生した時は、そのファイルに対してエラー指示子が設定されます。

備　考　fputc 関数は、文字 c をファイルポインタ fp の示すストリーム入出力ファイルへ出力します。
fputc 関数は、通常出力した文字 c を返しますが、エラー発生の時は、EOF を返します。

ファイルに文字列出力

*int fputs(const char *s, FILE *fp)*

説　明　ストリーム入出力用ファイルへ文字列を出力します。

ヘッダ　<stdio.h>

リターン値　正常：　0
異常：　0 以外

引　数

s	出力する文字列へのポインタ
fp	ファイルポインタ

例

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

備　考　fputs 関数は、s の指すヌル文字の直前までの文字列をファイルポインタ fp の示すストリーム入出力用ファイルへ出力します。この時、文字列の終了を示すヌル文字は出力されません。
fputs 関数は、通常 0 を返しますが、エラー発生の時は、0 以外の値を返します。

ファイルから 1 文字入力

*int getc(FILE *fp)*

説　明　ストリーム入出力用ファイルから 1 文字入力します。

ヘッダ　<stdio.h>

リターン値　正常：　ファイルの終了の時：EOF
　　ファイルの終了でない時：入力した文字
　　異常：　EOF

引　数　fp　　ファイルポインタ

例
```
#include  <stdio.h>
FILE  *fp;
int  ret;
    ret=getc(fp);
```

エラー条件　読み込みエラーが発生した時、そのファイルに対してエラー指示子が設定されます。

備　考　getc 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルから 1 文字入力します。
getc 関数は、通常入力した 1 文字を返しますがファイルの終了やエラー発生の時は、EOF を返します。またファイルの終了の時には、そのファイルに対するファイル終了指示子が設定されます。

1 文字入力

int getchar()

説　明　標準入力ファイル（stdin）から、1 文字入力します。

ヘッダ　<stdio.h>

リターン値　正常：　ファイルの終了の時：EOF
　　ファイルの終了でない時：入力した文字
　　異常：　EOF

例
```
#include  <stdio.h>
int  ret;
    ret=getchar();
```

エラー条件　読み込みエラーが発生した時、そのファイルに対してエラー指示子が設定されます。

備　考　getchar 関数は標準入力ファイル（stdin）から 1 文字入力します。
getchar 関数は、通常入力した 1 文字を返しますが、ファイルの終了やエラー発生の時は EOF を返します。また、ファイルの終了の時には、そのファイルに対するファイル終了指示子が設定されます。

文字列入力

*char *gets(char *s)*

説　明　標準入力ファイル（stdin）から文字列を入力します。

ヘッダ　<stdio.h>

リターン値　正常：　ファイルの終了の時：NULL
　　ファイルの終了でない時：s
　　異常：　NULL

引　数　s　　文字列を入力する記憶域へのポインタ

例
```
#include  <stdio.h>
char  *ret, *s;
    ret=gets(s);
```

備　考　gets 関数は、標準入力ファイル（stdin）から、s で始まる記憶域へ文字列を入力します。
gets 関数は、ファイルの終了か、改行文字を入力するまで文字を入力し、改行文字の代わりにヌル文字を付け加えます。
gets 関数は、通常文字列を入力する記憶域へのポインタ s を返しますが、標準入力ファイルの終了やエラー発生の時は、NULL を返します。
標準入力ファイルが終了した時は、s が指す記憶域の内容は変化しませんが、エラー発生の時は s が指す記憶域の内容は保証されません。

ファイルに 1 文字出力

*int putc(int c, FILE *fp)*

説　明　ストリーム入出力用ファイルへ 1 文字出力します。

ヘッダ　<stdio.h>

リターン値　正常：　出力した文字
　　異常：　EOF

引　数　c　　出力する文字
　　fp　　ファイルポインタ

例
```
#include  <stdio.h>
FILE  *fp;
int  c, ret;
    ret=putc(c, fp);
```

エラー条件　書き出しエラーが発生した時は、そのファイルに対してエラー指示子が設定されます。

備　考　putc 関数は、文字 c をファイルポインタ fp の示すストリーム入出力ファイルへ出力します。
putc 関数は、通常出力した文字 c を返しますがエラー発生の時は、EOF を返します。

1 文字出力

int putchar(int c)

説　明　標準出力ファイル（stdout）へ 1 文字出力します。

ヘッダ　<stdio.h>

リターン値　正常：　出力した文字
異常：　EOF

引　数　c　出力する文字

例
```
#include  <stdio.h>
int  c, ret;
    ret=putchar(c);
```

エラー条件　書き出しエラーが発生した時は、そのファイルに対してエラー指示子が設定されます。

備　考　putchar 関数は、文字 c を標準出力ファイル（stdout）へ出力します。putchar マクロは、通常出力した文字 c を返しますが、エラー発生の時は EOF を返します。

文字列出力

*int puts(const char *s)*

説　明　標準出力ファイル（stdout）へ文字列を出力します。

ヘッダ　<stdio.h>

リターン値　正常：　0
異常：　0 以外

引　数　s　出力する文字列へのポインタ

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

備　考　puts 関数は、s の指す文字列を標準出力ファイル（stdout）へ出力します。この時、文字列の終了を示す文字は出力されず、代わりに改行文字を出力します。
puts 関数は、通常 0 を返しますが、エラー発生の時、0 以外の値を返します。

ファイルに 1 文字返却

*int ungetc(int c, FILE *fp)*

説　明　ストリーム入出力用ファイルへ 1 文字を戻します。

ヘッダ　<stdio.h>

リターン値　正常：　戻した文字
異常：　EOF

引　数

c	戻す文字
fp	ファイルポインタ

例

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

備　考　ungetc 関数は、文字 c を、ファイルポインタ fp に示すストリーム入出力用ファイルへ戻します。
また、ここで戻された文字は、fflush，fseek，rewind 関数を呼び出さなければ次の入力データとなります。
ungetc 関数は、通常戻した文字 c を返しますが、エラー発生の時は、EOF を返します。
ungetc 関数が fflush，fseek，rewind 関数を実行することなく 2 回以上呼び出された時の動作は保証されません。また、ungetc 関数が実行されるとファイルに対する現在の位置指示子がひとつ戻されますが、この位置指示子がすでにファイルの先頭に位置している時は、位置指示子は保証されなくなります。

ファイル読み込み

*size_t fread(void *ptr, size_t size, size_t n, FILE *fp)*

説　明　ストリーム入出力用ファイルから、指定した記憶域にデータを入力します。

ヘッダ　<stdio.h>

リターン値　size もしくは n が 0 の時：0
size, n がともに 0 でない時：入力に成功したメンバ数

引　数

ptr	データを入力する記憶域へのポインタ
size	1 メンバのバイト数
n	入力するメンバの数
fp	ファイルポインタ

例

```
#include  <stdio.h>
void  *ptr;
size_t  size;
size_t  n, ret;
FILE  *fp;
    ret=fread(ptr, size, n, fp);
```

備　考　fread 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルから ptr が指す記憶域に size で指定したバイト数を 1 メンバとしたデータを n メンバ入力します。この時ファイルに対する位置指示子は入力したバイト数分進められます。
fread 関数は、実際に入力に成功したメンバ数を返しますので、通常 n と同じ値になります。しかし、ファイルが終了した時やエラー発生の時は、それまで入力に成功したメンバ数を返しますので、n より小さな値となります。ファイルの終了かエラー発生かの区別は、ferror, feof 関数を用いて行ってください。
size もしくは n が 0 の時、リターン値として 0 を返し ptr の指す記憶域の内容は変化しません。また、エラーが発生した時、または、メンバの途中までしか入力できなかった時は、そのファイルの位置指示子は保証されません。

ファイル書き込み

*size_t fwrite(const void *ptr, size_t size, size_t n, FILE *fp)*

説　明　メモリ領域からストリーム入出力用ファイルにデータを出力します。

ヘッダ　`<stdio.h>`

リターン値　出力に成功したメンバ数

引　数

`ptr`	出力するデータを格納している記憶域へのポインタ
`size`	`1` メンバのバイト数
`n`	出力するメンバの数
`fp`	ファイルポインタ

例

```
#include  <stdio.h>
const void  *ptr;
size_t  size;
size_t  n, ret;
FILE  *fp;
    ret=fwrite(ptr, size, n, fp);
```

備　考　`fwrite` 関数は、`ptr` の指す記憶域から、ファイルポインタ `fp` の示すストリーム入出力用ファイルに、`size` で指定したバイト数を `1` メンバとしたデータを `n` メンバ出力します。
この時、ファイルに対する位置指示子は出力したバイト数進められます。
`fwrite` 関数は、実際に出力に成功したメンバ数を返しますので、通常 `n` と同じ値になります。しかし、エラー発生の時はそれまで出力に成功したメンバ数を返しますので、それより小さな値となります。
エラー発生の時、そのファイルの位置指示子は保証されません。

ファイル読み書き位置移動

*int fseek(FILE *fp, long offset, int type)*

説　明　ストリーム入出力用ファイルの現在の読み書き位置を移動させます。

ヘッダ　<stdio.h>

リターン値　正常：　0
異常：　0 以外

引　数

fp	ファイルポインタ
offset	オフセットの種類で指定された位置からのオフセット
type	オフセットの種類

例

```
#include  <stdio.h>
FILE  *fp;
long  offset;
int  type, ret;
    ret=fseek(fp, offset, type);
```

備　考　fseek 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルの現在の読み書き位置をオフセットの種類 type で指定した場所から offset バイト先の位置に移動します。オフセットの種類を**表** 10.40 に示します。
fseek 関数は、通常は 0 を返しますが、不適当な要求に対しては 0 以外の値を返します。

表 10.40　**オフセットの種類**

	オフセットの種類	意味
1	SEEK_SET	**ファイルの先頭から** offset **バイト先の位置に移動します。この時、**offset **で指定する値は** 0 **か正でなければなりません。**
2	SEEK_CUR	**ファイルの現在位置から** offset **バイト先の位置に移動します。この時、**offset **で指定する値が正ならばファイルの後方に、負ならばファイルの先頭に向かって移動します。**
3	SEEK_END	**ファイルの終わりから** offset **バイト先の位置に移動します。この時** offset **で指定する値は** 0 **か負でなければなりません。**

テキストファイルの時は、オフセットの種類は SEEK_SET でかつ、offset は 0 かそのファイルに対する ftell 関数によって返された値でなければなりません。また、fseek 関数を呼び出すことによって ungetc 関数の効果はなくなりますので注意が必要です。

ファイル読み書き位置取得

*long ftell(FILE *fp)*

説　明　ストリーム入出力用ファイルの現在の読み書き位置を求めます。

ヘッダ　<stdio.h>

リターン値　現在の位置指示子の位置（テキストファイル）
ファイルの先頭から現在位置までのバイト数（バイナリファイル）

引　数　fp　ファイルポインタ

例
```
#include  <stdio.h>
FILE  *fp;
long  ret;
    ret=ftell(fp);
```

備　考　ftell 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルの現在の読み書き位置を求めます。
ftell 関数は、バイナリファイルの時、ファイルの先頭から現在位置までのバイト数を返しますが、テキストファイルの時は、ここで返した値が fseek 関数で使用できるように処理系定義の値を位置指示子の位置として返します。
ftell 関数を 2 回テキストファイルに適用した時、そのリターン値の差が実際のファイル上の隔たりを表わすことにはなりません。

ファイル先頭に移動

*void rewind(FILE *fp)*

説　明　ストリーム入出力用ファイルの現在の読み書き位置を、ファイルの先頭に移動します。

ヘッダ　<stdio.h>

引　数　fp　ファイルポインタ

例
```
#include  <stdio.h>
FILE  *fp;
    rewind(fp);
```

備　考　rewind 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルの現在の読み書き位置をファイルの先頭に移動します。
また、rewind 関数は、そのファイルに対する終了指示子とエラー指示子をクリアします。
rewind 関数を呼び出すことによって、ungetc 関数の効果はなくなりますので、注意が必要です。

エラー状態クリア

*void clearerr(FILE *fp)*

説　明　ストリーム入出力用ファイルのエラー状態をクリアします。

ヘッダ　<stdio.h>

引　数　fp　　ファイルポインタ

例
```
#include  <stdio.h>
FILE  *fp;
    clearerr(fp);
```

備　考　clearerr 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルに対するエラー指示子と終了指示子をクリアします。

ファイル終了判定

*int feof(FILE *fp)*

説　明　ストリーム入出力用ファイルが終わりであるかどうかを判定します。

ヘッダ　<stdio.h>

リターン値　ファイルが終わりの時：0 以外
ファイルが終わりでない時：0

引　数　fp　　ファイルポインタ

例
```
#include  <stdio.h>
FILE  *fp;
int  ret;
    ret=feof(fp);
```

備　考　feof 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルが終了したかどうかを判定します。
feof 関数は、指定したストリーム入出力用ファイルに対するファイル終了指示子を調べ、設定されていればファイルが終わりであるとして、0 以外の値を返します。設定されていなければ、ファイルはまだ終わりではないとして 0 を返します。

ファイルエラー状態判定

*int ferror(FILE *fp)*

説　明　ストリーム入出力用ファイルがエラー状態であるかどうかを判定します。

ヘッダ　<stdio.h>

リターン値　ファイルがエラー状態の時：0 以外
ファイルがエラー状態でない時：0

引　数　fp　ファイルポインタ

例
```
#include  <stdio.h>
FILE  *fp;
int  ret;
    ret=ferror(fp);
```

備　考　ferror 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルがエラー状態であるかどうかを判定します。
ferror 関数は、指定したストリーム入出力用ファイルに対するエラー指示子を調べ、設定されていれば、エラー状態にあるとして 0 以外の値を返します。設定されていなければ、エラー状態ではないとして 0 を返します。

エラーメッセージ出力

*void perror(const char *s)*

説　明　標準エラーファイル（stderr）に、エラー番号に対応したエラーメッセージを出力します。

ヘッダ　<stdio.h>

引　数　s　エラーメッセージへのポインタ

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

備　考　perror 関数は標準エラーファイル（stderr）へ s で示されるエラーメッセージと errno とを対応させ出力します。
出力するメッセージは、もし、s が NULL でなく、s の指す文字列がヌル文字でないならば、s の指す文字列にコロンと空白とその後に処理系定義のエラーメッセージを続け最後に改行文字を付けた形式で出力されます。

(13) ＜no_float.h＞

浮動小数点変換（%f,%e,%E,%g,%G）をサポートしない、簡易入出力関数を提供します。浮動小数点変換を必要としないファイル入出力を行う場合、ROM サイズを削減することができます。

【関数】

種別	定義名	説明
関数	fprintf	書式に従ってストリーム入出力用ファイルへデータを出力します。
	fscanf	ストリーム入出力用ファイルからデータを入力し、書式に従って変換します。
	printf	データを書式に従って変換し、標準出力ファイル（stdout）へ出力します。
	scanf	標準入力ファイル（stdin）からデータを入力し、書式に従って変換します。
	sprintf	データを書式に従って変換し、指定した領域へ出力します。
	sscanf	指定した記憶域からデータを入力し、書式に従って変換します。
	vfprintf	可変個のパラメタリストを書式に従って指定したストリーム入出力用ファイルに出力します。
	vprintf	可変個のパラメタリストを書式に従って標準出力ファイルに出力します。
	vsprintf	可変個のパラメタリストを書式に従って指定した記憶域に出力します。

本関数を使用する場合、<stdio.h>をインクルードする前に<no_float.h>をインクルードしてください。

例：

```
#include <no_float.h>
#include <stdio.h>
void main(void)
{
    printf(“Hello\n”);
}
```

【注】 #include <no_float.h>**を指定したときに、本関数で浮動小数点数を指定した場合、実行時の動作は保証しません。**

(14) ＜stdlib.h＞

C プログラムでの標準的処理を行う関数を定義しています。

以下のマクロは、処理系定義です。

種別	定義名	説明
型 (マクロ)	onexit_t	onexit 関数で登録する関数の返す型および onexit 関数のリターン値の型です。
	div_t	div 関数のリターン値の構造体の型です。
	ldiv_t	ldiv 関数のリターン値の構造体の型です。
定数 (マクロ)	RAND_MAX	rand 関数において生成する擬似乱数整数の最大値です。
関数	atof	数を表現する文字列を double 型の浮動小数点数値に変換します。
	atoi	10 進数を表現する文字列を int 型の整数値に変換します。
	atol	10 進数を表現する文字列を long 型の整数値に変換します。
	strtod	数を表現する文字列を double 型の浮動小数点数値に変換します。
	strtol	数を表現する文字列を long 型の整数値に変換します。
	rand	0 から RAND_MAX の間の擬似乱数整数を生成します。
	srand	rand 関数で生成する擬似乱数列の初期値を設定します。
	calloc	記憶域を割り当てて、すべての割当てられた記憶域を 0 によって初期化します。
	free	指定された記憶域を解放します。
	malloc	記憶域を割り当てます。
	realloc	記憶域の大きさを指定された大きさに変更します。
	bsearch	2 分割検索を行います。
	qsort	ソートを行います。
	abs	int 型整数の絶対値を計算します。
	div	int 型整数の除算の商と余りを計算します。
	labs	long 型整数の絶対値を計算します。
	ldiv	long 型整数の除算の商と余りを計算します。

文字列を double 型に変換

*double atof(const char *nptr)*

説　明　数を表現する文字列を、double 型の浮動小数点数値に変換します。

ヘッダ　<stdlib.h>

リターン値　変換された double 型の浮動小数点数値

引　数　nptr　変換する数を表現する文字列のポインタ

例
```
#include  <stdlib.h>
const char  *nptr;
double  ret;
    ret=atof(nptr);
```

備　考　変換は、浮動小数点数の形式に合わない最初の文字までに対して行います。
atof 関数は、オーバフロー等のエラーが生じても errno を設定しません。また、エラーが生じた場合、結果の値は保証されません。変換のエラーが生じる可能性がある場合は、strtod 関数を使用してください。

文字列を int 型に変換

*int atoi(const char *nptr)*

説　明　10 進数を表現する文字列を、int 型の整数値に変換します。

ヘッダ　<stdlib.h>

リターン値　変換された int 型の整数値

引　数　nptr　変換する数を表現する文字列のポインタ

例
```
#include  <stdlib.h>
const char  *nptr;
int  ret;
    ret=atoi(nptr);
```

備　考　変換は、10 進数の形式に合わない最初の文字までに対して行います。
atoi 関数は、オーバフロー等のエラーが生じても errno を設定しません。また、エラーが生じた場合、結果の値を保証しません。変換のエラーが生じる可能性がある場合は、strtol 関数を使用してください。

文字列を long 型に変換

*long atol(const char *nptr)*

説　明　10 進数を表現する文字列を、long 型の整数値に変換します。

ヘッダ　<stdlib.h>

リターン値　変換された long 型の整数値

引　数　nptr　変換する数を表現する文字列のポインタ

例
```
#include  <stdlib.h>
const char  *nptr;
long  ret;
    ret=atol(nptr);
```

備　考　変換は、10 進数の形式に合わない最初の文字までに対して行います。
atol 関数は、オーバフロー等のエラーが生じても errno を設定しません。また、エラーが生じた場合、結果の値を保証しません。変換時にエラーが生じる可能性がある場合は、strtol 関数を使用してください。

文字列を double 型に変換

*double strtod(const char *nptr, char **endptr)*

説　明　数を表現する文字列を double 型の浮動小数点数値に変換します。

ヘッダ　<stdlib.h>

リターン値　正常：　nptr が指している文字列が浮動小数点数を構成しない文字で始まっている時：0
nptr が指している文字列が浮動小数点数を構成する文字で始まっている時
：変換された double 型の浮動小数点数値
異常：　変換後の値がオーバフローの時：変換する文字列の符号と同符号をもつ HUGE_VAL
変換後の値がアンダフローの時：0

引　数

nptr	変換する数を表現する文字列へのポインタ
endptr	浮動小数点数値を構成していない最初の文字へのポインタを格納する記憶域へのポインタ

例

```
#include  <stdlib.h>
const char  *nptr;
char  **endptr;
double  ret;
    ret=strtod(nptr, endptr);
```

エラー条件　変換後の値がオーバフロー／アンダフローをおこす時は、errno に ERANGE が設定されます。

備　考　strtod 関数は、最初の数字もしくは小数点から浮動小数点数値を構成しない文字の直前までを「10.1.3(4)　**浮動小数点演算の仕様**」の規則に従って double 型の浮動小数点数値に変換します。ただし、指数部も小数点も現われなかった時は、小数点は文字列の最後の数字の後に続くと仮定されます。endptr の指す領域には、浮動小数点数を構成しない最初の文字へのポインタを設定します。数字を読み込む前に浮動小数点数を構成しない文字がある場合は nptr の値を設定します。endptr が NULL の場合、この設定は行われません。

文字列を long 型に変換

*long strtol(const char *nptr, char **endptr, int base)*

説　明　数を表現する文字列を long 型の整数値に変換します。

ヘッダ　<stdlib.h>

リターン値　正常：nptr が指している文字列が整数を構成しない文字で始まっている時：0
nptr が指している文字列が整数を構成する文字で始まっている時
：変換された long 型の整数値
異常：変換後の値がオーバフローの時：変換する文字列の符号に従って LONG_MAX
あるいは LONG_MIN

引　数

nptr	変換する数を表現する文字列へのポインタ
endptr	整数を構成しない最初の文字へのポインタを格納する記憶域へのポインタ
base	変換の基数（0 又は 2～36）

例

```
#include  <stdlib.h>
long  ret;
const char  *nptr;
char  **endptr;
int  base;
    ret=strtol(nptr, endptr, base);
```

エラー条件　変換後の値がオーバフローをおこす時は、errno に ERANGE が設定されます。

備　考　strtol 関数は、最初の数字から整数を構成しない最初の文字の前までを long 型の整数値に変換します。
endptr の指す記憶域に、整数を構成しない最初の文字へのポインタを設定します。最初の数字を読み込む前に整数を構成しない文字がある場合は nptr の値を設定します。endptr が NULL 場合、この設定は行われません。
base の値が 0 の時は、**「10.1.1(4)　整数」**の規則に従って変換されます。base の値が 2 から 36 の間の時は、変換する時の基数を示しています。ここで変換する文字列中の a（もしくは A）から z（もしくは Z）までの文字は、10 から 35 の値に対応付けられます。base の値より大きいか等しい文字が、変換する文字列の中にある時は、そこで変換処理を終了します。また、符号の後にある 0 は、変換の時は無視され、また、base が 16 の時の 0x（もしくは 0X）も無視されます。

擬似乱数生成

int rand()

説　明　0 から RAND_MAX の間の擬似乱数整数を生成します。

ヘッダ　<stdlib.h>

リターン値　擬似乱数整数値

例
```
#include  <stdlib.h>
int  ret;
    ret=rand();
```

擬似乱数列の初期設定

void srand(unsigned int seed)

説　明　rand 関数で生成する擬似乱数列の初期値を設定します。

ヘッダ　<stdlib.h>

引　数　seed　　擬似乱数列生成の初期値

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

備　考　srand 関数は、rand 関数が擬似乱数列を生成するための初期値を設定します。したがって、rand 関数で擬似乱数値を生成している時に、再度 srand 関数で、同じ値の初期値を設定すると、擬似乱数列はくり返し生成されることになります。
rand 関数が srand 関数より先に呼ばれた時は、擬似乱数列の生成の初期値として 1 が設定されます。

初期化付き記憶域割り当て

*void *calloc(size_t nelem, size_t elsize)*

説　明　記憶域を割り当てて、すべての割り当てられた記憶域を 0 によって初期化します。

ヘッダ　<stdlib.h>

リターン値　正常：　割り当てられた記憶域の先頭のアドレス
異常：　記憶域の割り当てができなかった時、またはパラメタのいずれかが 0 の時：NULL

引　数

nelem	要素の数
elsize	ひとつの要素の占めるバイト数

例

```
#include  <stdlib.h>
size_t  nelem, elsize;
void  *ret;
    ret=calloc(nelem, elsize);
```

備　考　elsize バイト単位の記憶域を nelem 個記憶域に割り当てます。また、その割り当てられた記憶域のすべてのビットは 0 で初期化されます。

記憶域解放

*void free(void *ptr)*

説　明　指定された記憶域を解放します。

ヘッダ　<stdlib.h>

引　数

ptr	解放する記憶域のアドレス

例

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

備　考　ptr が指す記憶域を解放し、再度割り当てて使用することを可能とします。ptr が NULL であれば何もしません。
解放しようとした記憶域が、calloc、 malloc、 realloc 関数で割り当てられた記憶域でない時、または、すでに free、realloc 関数によって解放されていた時の動作は保証されません。また、解放された後の記憶域を参照した時の動作も保証されません。

記憶域割り当て

*void *malloc(size_t size)*

説　明　記憶域を割り当てます。

ヘッダ　<stdlib.h>

リターン値　正常：　割り当てられた記憶域の先頭アドレス
異常：　記憶域の割り当てができなかった時、または size が 0 の時：NULL

引　数　size　割り当てる記憶域のバイト数

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

備　考　size で示されるバイトの分だけ記憶域を割り当てます。

記憶域割り当てサイズ変更

*void *realloc(void *ptr, size_t size)*

説　明　記憶域の大きさを指定された大きさに変更します。

ヘッダ　<stdlib.h>

リターン値　正常：　変更した記憶域の先頭アドレス
異常：　記憶域の割り当てができなかった時、または size が 0 の時：NULL

引　数　ptr　変更する記憶域の先頭アドレス
size　変更後の記憶域のバイト数

例
```
#include  <stdlib.h>
size_t  size;
void  *ptr, *ret;
    ret=realloc(ptr, size);
```

備　考　ptr の指す記憶域の大きさを size で示されるバイト分の大きさの記憶域に変更します。
もし、新しく割り当てられた記憶域の大きさが、変更前の記憶域の大きさより小さい時は、新しく割り当てられた記憶域の大きさまでの内容は変化しません。
ptr が calloc、malloc、realloc 関数で割り当てられた記憶域へのポインタでない時、またはすでに free、realloc 関数によって解放されている記憶域へのポインタの時、動作は保証されません。

二分割検索

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

説　明　二分割検索を行います。

ヘッダ　<stdlib.h>

リターン値　一致するメンバが検索できた時：一致したメンバへのポインタ
一致するメンバが検索できなかった時：NULL

引　数

key	検索するデータへのポインタ
base	検索対象となるテーブルへのポインタ
nmemb	検索対象のメンバの数
size	検索対象のメンバのバイト数
compar	比較を行う関数へのポインタ

例

```
#include  <stdlib.h>
const void  *key, *base;
size_t  nmemb, size;
int  (*compar)(const void *, const void *);
void  *ret;
    ret=bsearch(key, base, nmemb, size, compar);
```

備　考　key の指すデータと一致するメンバを、base の指すテーブルの中で二分割検索法によって検索します。比較を行う関数は、比較する二つのデータへのポインタ p1（第 1 引数）、p2（第 2 引数）を受け取り、以下の仕様に従って結果を返してください。

*p1 < *p2 の時　負の値を返します。
*p1 == *p2 の時　0　を返します。
*p1 > *p2 の時　正の値を返します。

検索対象となる各メンバは、昇順に並んでいる必要があります。

ソート

*void qsort(const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))*

説　明　ソートを行います。

ヘッダ　<stdlib.h>

引　数

base	ソート対象となるテーブルへのポインタ
nmemb	ソート対象のメンバの数
size	ソート対象のメンバのバイト数
compar	比較を行う関数へのポインタ

例

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

備　考　base の指すテーブルのデータをソートします。データの並べる順序は、比較を行う関数へのポインタによって指定します。この関数は、比較する二つのデータへのポインタ p1（第 1 引数）、p2（第 2 引数）を受け取り、以下の仕様に従って結果を返してください。

*p1＜ *p2 の時　負の値を返します。
*p1＝＝*p2 の時　0　を返します。
*p1＞ *p2 の時　正の値を返します。

絶対値

int abs(int i)

説　明　絶対値を求めます。

ヘッダ　<stdlib.h>

リターン値　i の絶対値

引　数

i	絶対値を求める整数

例

```
#include  <stdlib.h>
int  i, ret;
    ret=abs(i);
```

備　考　i の絶対値を求めた結果、int 型整数値として表現できない時の動作は保証されません。

商と余り

div_t div(int numer, int denom)

説　明　int 型整数の除算の商と余りを計算します。

ヘッダ　<stdlib.h>

リターン値　numer を denom で除算した結果の商と余り。

引　数

numer	被除数
denom	除数

例

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

絶対値

long labs(long j)

説　明　long 型整数の絶対値を計算します。

ヘッダ　<stdlib.h>

リターン値　j の絶対値

引　数

j	絶対値を求める整数

例

```
#include  <stdlib.h>
long  j;
long  ret;
    ret=labs(j);
```

備　考　j の絶対値を求めた結果、long 型の整数値として表現できない時の動作は保証されません。

商と余り

ldiv_t ldiv(long numer, long denom)

説　明　long 型整数の除算の商と余りを計算します。

ヘッダ　<stdlib.h>

リターン値　numer を denom で除算した結果の商と余り。

引　数

numer	被除数
denom	除数

例

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

(15) ＜string.h＞

文字配列の操作に必要な種々の関数を定義します。

種別	定義名	説明
関数	memcpy	複写元の記憶域の内容を指定した大きさ分、複写先の記憶域に複写します。
	strcpy	複写元の文字列の内容を、複写先の記憶域にヌル文字も含めて複写します。
	strncpy	複写元の文字列を指定された文字数分、複写先の記憶域に複写します。
	strcat	文字列の後に、文字列を連結します。
	strncat	文字列に文字列を指定した文字数分、連結します。
	memcmp	指定された二つの記憶域の比較を行います。
	strcmp	指定された二つの文字列を比較します。
	strncmp	指定された二つの文字列を指定された文字数分まで比較します。
	memchr	指定された記憶域において、指定された文字が最初に現われる位置を検索します。
	strchr	指定された文字列において、指定された文字が最初に現われる位置を検索します。
	strcspn	指定された文字列を先頭から調べ、別に指定した文字列中の文字以外の文字が先頭から何文字続くか求めます。
	strpbrk	指定された文字列において、別に指定された文字列中の文字が最初に現われる位置を検索します。
	strrchr	指定された文字列において指定された文字が最後に現われる位置を検索します。
	strspn	指定された文字列を先頭から調べ別に指定した文字列中の文字が先頭から何文字続くかを求めます。
	strstr	指定された文字列において、別に指定した文字列が最初に現われる位置を検索します。
	strtok	指定した文字列をいくつかの字句に切り分けます。
	memset	指定された記憶域の先頭から指定された文字を指定された文字数分設定します。
	strerror	エラーメッセージを設定します。
	strlen	文字列の長さを計算します。
	memmove	複写元の記憶域の内容を、指定した大きさ分、複写先の記憶域に複写します。複写元と複写先の記憶域が重なっていても、正しく複写されます。

本標準インクルードファイル内で定義されている関数を使用する時は、以下の二つの事項に注意する必要があります。

（1） 文字列の複写を行う時、複写先の領域が複写元の領域よりも、小さい場合、動作は保証されませんので注意が必要です。

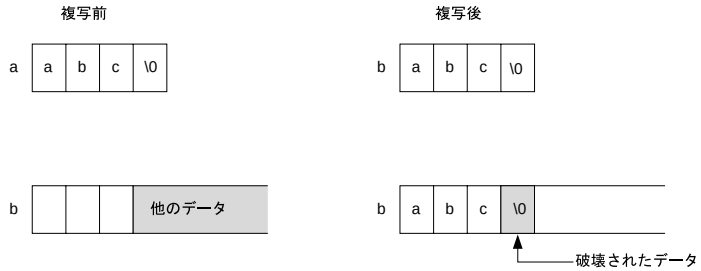
処理系定義仕様

	項目	C コンパイラの仕様
1	strerror 関数が返すエラーメッセージの内容	「12.3 C ライブラリ関数のエラーメッセージ」を参照してください。

例：

```
char  a[ ]="abc";
char  b[3];
      :
      :
   strcpy(b, a);
```

この場合、配列 a のサイズは（ヌル文字を含めて）4 バイトです。したがって、strcpy 関数によって複写を行うと、配列 b の領域以外のデータを書き換えることになります。

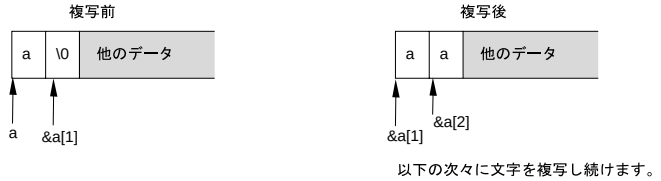


（2） 文字列の複写を行う時、複写元の領域と複写先の領域が重なっていると正しい動作が保証されませんので注意が必要です。

例：

```
int a [ ]="a";
      .
      .
      .
strcpy(&a[1], a);
      .
      .
      .
```

この場合、複写元の文字列がヌル文字に達する以前に、ヌル文字の上に文字'a'を書き込むことになります。したがって、複写元の文字列のデータに続くデータを書き換えることになります。



記憶域複写

*void *memcpy(void *s1, const void *s2, size_t n)*

説　明　複写元の記憶域の内容を、指定した大きさ分、複写先の記憶域に複写します。

ヘッダ　<string.h>

リターン値　s1 の値

引　数

s1	複写先の記憶域へのポインタ
s2	複写元の記憶域へのポインタ
n	複写する文字数

例

```
#include  <string.h>
void  *ret, *s1;
const void  *s2;
size_t  n;
    ret=memcpy(s1, s2, n);
```

文字列複写

*char *strcpy(char *s1, const char *s2)*

説　明　複写元の文字列の内容を、複写先の記憶域にヌル文字も含めて複写します。

ヘッダ　<string.h>

リターン値　s1 の値

引　数

s1	複写先の記憶域へのポインタ
s2	複写元の文字列へのポインタ

例

```
#include  <string.h>
char  *s1, *ret;
const char  *s2;
    ret=strcpy(s1, s2);
```

文字列複写

*char *strncpy(char *s1, const char *s2, size_t n)*

説　明　複写元の文字列を指定された文字数分、複写先の記憶域に複写します。

ヘッダ　<string.h>

リターン値　s1 の値

引　数

s1	複写先の記憶域へのポインタ
s2	複写元の文字列へのポインタ
n	複写する文字数

例

```
#include  <string.h>
char  *s1, *ret;
const char  *s2;
size_t  n;
    ret=strncpy(s1, s2, n);
```

備　考　s2 で指された文字列から最後の n 文字を s1 で指される記憶域に複写します。s2 で指定された文字列の長さが n 文字より短い時は、n 文字になるまでヌル文字が付加されます。
s2 で指された文字列の長さが n 文字より長い時は、s1 に複写された文字列はヌル文字で終了しないことになります。

文字列連結

*char *strcat(char *s1, const char *s2)*

説　明　文字列の後に、文字列を連結します。

ヘッダ　<string.h>

リターン値　s1 の値

引　数

s1	連結される文字列へのポインタ
s2	連結する文字列へのポインタ

例

```
#include  <string.h>
char  *s1, *ret;
const char  *s2;
    ret=strcat(s1, s2);
```

備　考　s1 で指された文字列の最後に、s2 で指された文字列を連結します。この時、s2 の指す文字列の最後を示すヌル文字も複写します。また、s1 で指された文字列の最後のヌル文字は削除されます。

文字列連結

*char *strncat(char *s1, const char *s2, size_t n)*

説　明　文字列に文字列を指定した文字数分連結します。

ヘッダ　<string.h>

リターン値　s1 の値

引　数

s1	連結される文字列へのポインタ
s2	連結する文字列へのポインタ
n	連結する文字数

例

```
#include  <string.h>
char  *s1, *ret;
const char  *s2;
size_t  n;
    ret=strncat(s1, s2, n);
```

備　考　s2 で指された文字列の先頭から最高 n 文字を s1 で指された文字列の最後に付加します。s1 で指された文字列の最高のヌル文字は s2 の先頭文字で置き換えられます。
また、連結された後の文字列の最後には、必ずヌル文字が付加されます。

記憶域比較

*int memcmp(const void *s1, const void *s2, size_t n)*

説　明　指定された二つの記憶域の内容を比較します。

ヘッダ　<string.h>

リターン値　s1 で指された記憶域 > s2 で指された記憶域の時：正の値
s1 で指された記憶域 == s2 で指された記憶域の時：0
s1 で指された記憶域 < s2 で指された記憶域の時：負の値

引　数

s1	比較される記憶域へのポインタ
s2	比較する記憶域へのポインタ
n	比較する記憶域の文字数

例

```
#include  <string.h>
const void  *s1, *s2;
size_t  n;
int  ret;
    ret=memcmp(s1, s2, n);
```

備　考　s1 で指された記憶域と s2 で指された記憶域の最初の n 文字分の内容を比較します。
この比較は処理系定義です。

文字列比較

*int strcmp(const char *s1, const char *s2)*

説　明　　指定された二つの文字列の内容を比較します。

ヘッダ　　<string.h>

リターン値　s1 で指された文字列 > s2 で指された文字列の時：正の値
s1 で指された文字列 == s2 で指された文字列の時：0
s1 で指された文字列 < s2 で指された文字列の時：負の値

引　数　　s1　　比較される文字列へのポインタ
s2　　比較する文字列へのポインタ

例
```
#include  <string.h>
const char  *s1, *s2;
int  ret;
    ret=strcmp(s1, s2);
```

備　考　　s1 で指された文字列と、s2 で指された文字列の内容を比較し、その結果をリターン値として設定します。
この比較は処理系定義です。

文字列比較

*int strncmp(const char *s1, const char *s2, size_t n)*

説　明　　指定された二つの文字列を指定された文字分まで比較します。

ヘッダ　　<string.h>

リターン値　s1 で指された文字列 > s2 で指された文字列の時：正の値
s1 で指された文字列 == s2 で指された文字列の時：0
s1 で指された文字列 < s2 で指された文字列の時：負の値

引　数　　s1　　比較される文字列へのポインタ
s2　　比較する文字列へのポインタ
n　　比較する文字数の最大値

例
```
#include  <string.h>
const char  *s1, *s2;
size_t  n;
int  ret;
    ret=strncmp(s1, s2, n);
```

備　考　　s1 で指された文字列と、s2 で指された文字列を最初の n 文字以内の範囲で、その内容を比較します。
この比較は処理系定義です。

記憶域内文字検索

*void *memchr(const void *s, int c, size_t n)*

説　明　指定された記憶域において、指定された文字が最初に現われる位置を検索します。

ヘッダ　<string.h>

リターン値　検索の結果見つかった時：見つけられた文字へのポインタ
検索の結果見つからなかった時：NULL

引　数

s	検索を行う記憶域へのポインタ
c	検索する文字
n	検索を行う文字数

例

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

備　考　s で指定された記憶域の先頭から n 文字の中で最初に現われた c の文字と同一文字の位置へのポインタをリターン値として返します。

最初の文字位置

*char *strchr(const char *s, int c)*

説　明　指定された文字列において、指定された文字が最初に現われる位置を検索します。

ヘッダ　<string.h>

リターン値　検索の結果見つかった時：見つけられた文字へのポインタ
検索の結果見つからなかった時：NULL

引　数

s	検索を行う文字列へのポインタ
c	検索する文字

例

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

備　考　s で指定された文字列中で最初に現われた c の文字と同一文字へのポインタをリターン値として返します。
s によって指される文字列の終了を現わすヌル文字も検索の対象として含まれます。

指定文字群が最初に現れるまでの文字数

*size_t strcspn (const char *s1, const char *s2)*

説　明　指定された文字列を先頭から調べ、別に指定した文字列中の文字以外の文字が先頭から何文字続くか求めます。

ヘッダ　<string.h>

リターン値　s2 が指す文字列を構成する文字以外の文字が構成される文字列 s1 の先頭からの長さ

引　数

s1	調べられる文字列へのポインタ
s2	s1 を調べるための文字列へのポインタ

例

```
#include  <string.h>
const char  *s1, *s2;
size_t  ret;
    ret=strcspn(s1, s2);
```

備　考　s2 が指す文字列を構成する文字以外の文字が、文字列として何文字続くかを s1 で指された文字列の先頭から調べ、その文字列の長さをリターン値として返します。
s2 によって指される文字列の終了を表わすヌル文字は、s2 で指された文字列の一部とはみなされません。

指定文字群が最初に現れる位置

*char *strpbrk(const char *s1, const char *s2)*

説　明　指定された文字列内において、別に指定された文字列中の文字が最初に現われる位置を検索します。

ヘッダ　<string.h>

リターン値　検索の結果見つかった時：見つかった文字へのポインタ
検索の結果見つからなかった時：NULL

引　数

s1	検索を行う文字列へのポインタ
s2	s1 内で検索する文字を示す文字列へのポインタ

例

```
#include  <string.h>
const char  *s1, *s2;
char  *ret;
    ret=strpbrk(s1, s2);
```

備　考　s1 で指された文字列において、s2 で指された文字列中の文字の一つが最初に現われる所を検索し、そのポインタをリターン値として返します。

最後の文字位置

*char *strrchr(const char *s, int c)*

説　明　指定された文字列において、指定された文字が最後に現われる位置を検索します。

ヘッダ　<string.h>

リターン値　検索の結果見つかった時：見つかった文字へのポインタ
検索の結果見つからなかった時：NULL

引　数

s	検索を行う文字列へのポインタ
c	検索する文字

例

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

備　考　s で指された文字列の中で c で指定する文字と同一の文字が最後に現われた位置へのポインタをリターン値として返します。
s によって指される文字列の終了を表わすヌル文字も検索の対象として含まれます。

指定文字群が連続する部分の長さ

*size_t strspn(const char *s1, const char *s2)*

説　明　指定された文字列を先頭から調べ、別に指定した文字列中の文字が先頭から何文字続くかを求めます。

ヘッダ　<string.h>

リターン値　s1 の先頭から、s2 で指定した文字が続いている文字数

引　数

s1	調べられる文字列へのポインタ
s2	s1 を調べるための文字列へのポインタ

例

```
#include  <string.h>
const char  *s1, *s2;
size_t  ret;
    ret=strspn(s1, s2);
```

備　考　s2 が指す文字列を構成する文字が文字列として何文字続くかを s1 で指された文字列の先頭から調べ、その文字列の長さをリターン値として返します。

最初の文字列位置

*char *strstr(const char *s1, const char *s2)*

説　明　指定された文字列において、別に指定した文字列が最初に現われる位置を検索します。

ヘッダ　<string.h>

リターン値　検索の結果見つかったとき：見つけられた文字へのポインタ
検索の結果見つからなかったとき：NULL

引　数

s1	検索を行う文字列へのポインタ
s2	検索する文字列へのポインタ

例

```
#include  <string.h>
const char  *s1, *s2;
char  *ret;
    ret=strstr(s1, s2);
```

備　考　s1 で指された文字列において、s2 で指された文字列が最初に現われる所を検索し、そのポインタをリターン値として返します。

字句切り分け

*char *strtok(char *s1, const char *s2)*

説　明　指定した文字列をいくつかの字句に切り分けます。

ヘッダ　<string.h>

リターン値　字句に切り分けられた時：切り分けた字句の先頭へのポインタ
字句に切り分けられなかった時：NULL

引　数

s1	いくつかの字句に切り分ける文字列へのポインタ
s2	文字列を切り分けるための文字からなる文字列へのポインタ

例

```
#include  <string.h>
char  *s1, *ret;
const char  *s2;
    ret=strtok(s1, s2);
```

備　考　strtok 関数は文字列を切り分けるために連続的に呼び出されます。
(a) 最初の呼び出し時
s1 で指された文字列を先頭から s2 で指された文字列中の文字によって字句に切り分けます。その結果字句に切り分けられれば、その字句の先頭へのポインタを、分けられなければ NULL をリターン値として返します。
(b) 2 回目以降の呼び出し時
以前に切り分けられた字句の次の文字から、s2 で指された文字列中の文字によって字句に切り分けます。その結果字句に切り分けられれば、その字句の先頭へのポインタを、分けられなければ NULL をリターン値として返します。
2 回目以降の呼び出しの時は、第 1 パラメタには NULL を指定します。また、s2 で指された文字列は呼び出しのたびに異なっていてもかまいません。切り出された字句の最後にはヌル文字が付きます。
strtok 関数の使用例を以下に示します。
例：

```
1   #include  <string.h>
2   static  char  s1[ ]="a@b, @c/@d";
3   char  *ret;
4
5   ret = strtok(s1, "@");
6   ret = strtok(NULL, ",@");
7   ret = strtok(NULL, "/@");
8   ret = strtok(NULL, "@");
```

【説明】
この例は、文字列「“a@b,@c/@d”」を strtok 関数を用いて a, b, c, d という字句に切り分けるプログラムを示しています。
2 行目で文字列 s1 に初期値として、文字列“a@b、 @c／@d”を設定しています。
5 行目では、「@」を区切り文字として字句を切り分けるため、strtok 関数を呼び出します。この結果、文字‘a’へのポインタがリターン値として得られ、文字‘a’の次の最初の区切り文字である「@」にヌル文字を埋め込みます。この結果、文字列“a”が切り出されます。
以下、同一の文字列から次々に字句を切り出すために第 1 引数に NULL を指定して strtok 関数を呼び出します。
この結果、文字列“b”、“c”、“d”が次々に切り出されます。

文字の繰り返し

*vold *memset(void *s, int c, size_t n)*

説　明　指定された記憶域の先頭から、指定された文字を指定された文字数分設定します。

ヘッダ　<string.h>

リターン値　s の値

引　数

s		文字が設定される記憶域へのポインタ
c		設定する文字
n		設定する文字数

例

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

備　考　s で指された記憶域に n 文字分、文字 c を設定します。

エラーメッセージ文字列

*char *strerror(int s)*

説　明　エラー番号を指定して、それに対するエラーメッセージを返します。

ヘッダ　<string.h>

リターン値　エラー番号に対応するエラーメッセージ（文字列）へのポインタ

引　数

s		エラー番号

例

```
#include  <string.h>
char  *ret;
int  s;
    ret=strerror(s);
```

備　考　エラー番号 s に対応するエラーメッセージへのポインタをリターン値として返します。
エラーメッセージの内容に関しては処理系定義です。
リターン値として返されたエラーメッセージを修正した時、動作は保証されません。

文字列の長さ

*size_t strlen(const char *s)*

説　明　文字列の長さを計算します。

ヘッダ　<string.h>

リターン値　文字列の文字数

引　数　s　長さを求める文字列へのポインタ

例
```
#include  <string.h>
const char  *s;
size_t  ret;
    ret=strlen(s);
```

備　考　s が指す文字列の終了を表わすヌル文字は、文字列の長さとしては計算に入れません。

記憶域移動

*void *memmove(void *s1, const void *s2, size_t n)*

説　明　複写元の記憶域の内容を指定した大きさ分、複写先の記憶域に複写します。
また、複写元と複写先の記憶域が、重なっている部分があっても、複写元の重なっている部分を上書きする前に複写するので正しく複写されます。

ヘッダ　<string.h>

リターン値　s1 の値

引　数　s1　複写先の記憶域へのポインタ
s2　複写元の記憶域へのポインタ
n　複写する文字数

例
```
#include  <string.h>
void  *ret, *s1;
const void  *s2;
size_t  n;
    ret=memmove(s1, s2, n);
```

10.3.2 C++クラスライブラリ

(1) ライブラリの概要

C++プログラムから標準的に利用できる組み込み向け C++クラスライブラリの仕様について説明します。ここでは、クラスライブラリの種類と対応する標準インクルードファイルについて説明します。以降では、ライブラリの構成に従って各クラスライブラリの仕様について説明します。

- **ライブラリの種類**

表 10.41 にクラスライブラリの種類と対応する標準インクルードファイルを示します。

表 10.41 **クラスライブラリの種類と標準インクルードファイルの対応**

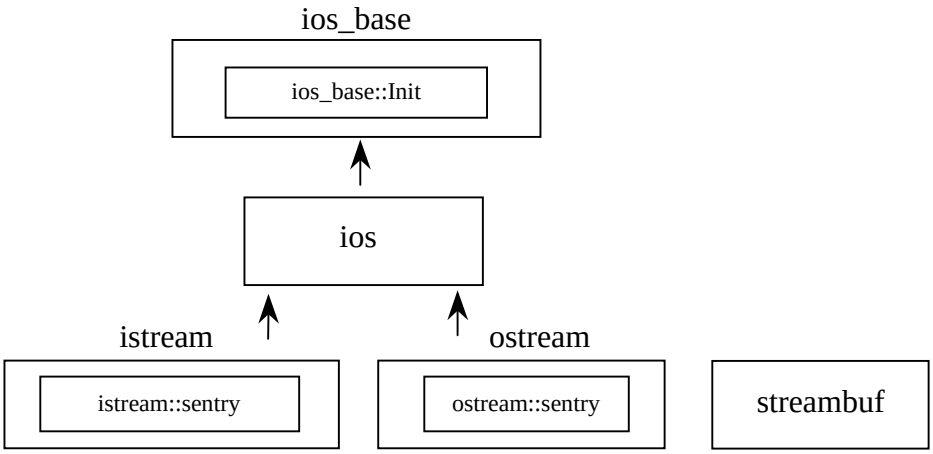
	ライブラリの種類	内容	標準インクルードファイル
1	**ストリーム入出力用クラスライブラリ**	**入出力操作を行うライブラリです。**	<ios>, <streambuf>, <istream>,<ostream>, <iostream>,<iomanip>
2	**メモリ操作用ライブラリ**	**メモリの確保・解放を行うライブラリです。**	<new>
3	**複素数計算用クラスライブラリ**	**複素数データ演算を行うライブラリです。**	<complex>
4	**文字列操作用クラスライブラリ**	**文字列操作を行うライブラリです。**	<string>

(2) ストリーム入出力用クラスライブラリ

ストリーム入出力用クラスライブラリに対応するヘッダファイルは以下の通りです。

- <ios>
  入出力用書式設定、入出力状態管理を行うデータメンバおよび関数メンバを定義します。
  ios クラスの他に、Init クラス、ios_base クラスを定義します。
- <streambuf>
  ストリームバッファに対する関数を定義します。
- <istream>
  入力ストリームからの入力関数を定義します。
- <ostream>
  出力ストリームへの出力関数を定義します。
- <iostream>
  入出力関数を定義します。
- <iomanip>
  引数を持つマニピュレータを定義します。

これらのクラスの派生関係は次のようになります。矢印は、派生クラスから基底クラスを参照していることを示します。なお、streambuf クラスには派生関係はありません。



ストリーム入出力用クラスライブラリで共通に使用される型名を示します。

種別	定義名	説明
型	streamoff	long 型で定義された型です。
	streamsize	size_t 型で定義された型です。
	int_type	int 型で定義された型です。
	pos_type	long 型で定義された型です。
	off_type	long 型で定義された型です。

(a) ios_base::Init クラス

種別	定義名	説明
変数	init_cnt	ストリーム入出力オブジェクト数をカウントする静的データメンバです。 低水準インタフェースで 0 に初期化する必要があります。
関数	init()	コンストラクタです
	~init()	デストラクタです。

ios_base::Init::Init()
クラス Init のコンストラクタです。
init_cnt をインクリメントします。

ios_base::Init::~Init()
クラス Init のデストラクタです。
init_cnt をデクリメントします。

(b) ios_base クラス

種別	定義名	説明
型	fmtflags	フォーマット制御情報を表す型です。
	iostate	ストリームバッファの入出力状態を表す型です。
	openmode	ファイルのオープンモードを表す型です。
	seekdir	ストリームバッファのシーク状態を表す型です。
変数	fmtfl	書式フラグです。
	wide	フィールド幅です。
	prec	出力時の精度(小数点以下の桁数)です。
	fillch	詰め文字です。
関数	void_ec2p_init_base()	初期化します。
	void_ec2p_copy_base(ios_base&ios_base_dt)	ios_base_dt をコピーします。
	ios_base()	コンストラクタです。
	~ios_base()	デストラクタです。
	fmtflags flags() const	書式フラグ(fmtfl)を参照します。
	fmtflags flags(fmtflags fmtflg)	fmtflg&書式フラグ(fmtfl)を書式フラグ(fmtfl)に設定します。
	fmtflags setf(fmtflags fmtflg)	fmtflg を書式フラグ(fmtfl)に設定します。
	fmtflags setf(fmtflags fmtflg, fmtflags mask)	mask&fmtflg を書式フラグ(fmtfl)に設定します。
	void unsetf(fmtflags mask)	~mask&書式フラグ(fmtfl)を書式フラグ(fmtfl)に設定します。
	char fill() const	詰め文字(fillch)を参照します。
	char fill(char ch)	ch を詰め文字(fillch)に設定します。
	int precision() const	精度(prec)を参照します。
	streamsize precision(streamsize preci)	preci を精度(prec)に設定します。
	streamsize width() const	フィールド幅(wide)を参照します。
	streamsize width(streamsize wd)	wd をフィールド幅(wide)に設定します。

ios_base::fmtflags
入出力に関するフォーマット制御情報を定義します。
fmtflags の各ビットマスクの定義は以下のようになります。

```
const ios_base::fmtflags ios_base::boolalpha      = 0x0000;
const ios_base::fmtflags ios_base::skipws         = 0x0001;
const ios_base::fmtflags ios_base::unitbuf        = 0x0002;
const ios_base::fmtflags ios_base::uppercase      = 0x0004;
const ios_base::fmtflags ios_base::showbase       = 0x0008;
const ios_base::fmtflags ios_base::showpoint      = 0x0010;
const ios_base::fmtflags ios_base::showpos        = 0x0020;
const ios_base::fmtflags ios_base::left           = 0x0040;
const ios_base::fmtflags ios_base::right          = 0x0080;
const ios_base::fmtflags ios_base::internal       = 0x0100;
const ios_base::fmtflags ios_base::adjustfield    = 0x01c0;
const ios_base::fmtflags ios_base::dec            = 0x0200;
const ios_base::fmtflags ios_base::oct            = 0x0400;
const ios_base::fmtflags ios_base::hex            = 0x0800;
const ios_base::fmtflags ios_base::basefield      = 0x0e00;
const ios_base::fmtflags ios_base::scientific     = 0x1000;
const ios_base::fmtflags ios_base::fixed          = 0x2000;
const ios_base::fmtflags ios_base::floatfield     = 0x3000;
const ios_base::fmtflags ios_base::_fmtmask       = 0x3fff;
```

ios_base::iostate
ストリームバッファの入出力状態を定義します。
iostate の各ビットマスクの定義は以下のようになります。

```
const ios_base::iostate ios_base::goodbit         = 0x0;
const ios_base::iostate ios_base::eofbit          = 0x1;
const ios_base::iostate ios_base::failbit         = 0x2;
const ios_base::iostate ios_base::badbit          = 0x4;
const ios_base::iostate ios_base::_statemask      = 0x7;
```

ios_base::openmode
ファイルのオープンモードを定義します。
openmode の各ビットマスクの定義は以下のようになります。

```
const ios_base::openmode ios_base::in     = 0x01;  入力用のファイルを open します。
const ios_base::openmode ios_base::out    = 0x02;  出力用のファイルを open します。
const ios_base::openmode ios_base::ate    = 0x04;  オープン後一度だけ eof に seek します。
const ios_base::openmode ios_base::app    = 0x08;  書き込む度に eof に seek します。
const ios_base::openmode ios_base::trunc  = 0x10;  ファイルを上書きモードで open します。
const ios_base::openmode ios_base::binary = 0x20;  ファイルをバイナリモードで open します。
```

ios_base::seekdir
ストリームバッファのシーク状態を定義します。
引き続き入力または出力を行うためのストリーム内の位置を決定します。
seekdir の各ビットマスクの定義は以下のようになります。
const ios_base::seekdir ios_base::beg = 0x0;
const ios_base::seekdir ios_base::cur = 0x1;
const ios_base::seekdir ios_base::end = 0x2;

void ios_base::_ec2p_init_base()
以下の値で初期設定します。
fmtfl = skipws | dec;
wide = 0;
prec = 6;
fillch = ‘ ‘;

void ios_base::_ec2p_copy_base(ios_base& ios_base_dt)
ios_base_dt をコピーします。

ios_base::ios_base()
クラス ios_base のコンストラクタです。
Init::Init()を呼び出します。

ios_base::~ios_base()
クラス ios_base のデストラクタです。

ios_base::fmtflags ios_base::flags() const
書式フラグ(fmtfl)を参照します。
リターン値は、書式フラグ(fmtfl)です。

ios_base::fmtflags ios_base::flags(fmtflags fmtflg)
fmtflg&書式フラグ(fmtfl)を書式フラグ(fmtfl)に設定します。
リターン値は、設定前の書式フラグ(fmtfl)です。

ios_base::fmtflags ios_base::setf(fmtflags fmtflg)
fmtflg を書式フラグ(fmtfl)に設定します。
リターン値は、設定前の書式フラグ(fmtfl) です。

ios_base::fmtflags ios_base::setf(fmtflags fmtflg, fmtflags mask)
mask&fmtflg の値を書式フラグ(fmtfl)に設定します。
リターン値は、設定前の書式フラグ(fmtfl) です。

void ios_base::unsetf(fmtflags mask)
~mask&書式フラグ(fmtfl)を書式フラグ(fmtfl)に設定します。

char ios_base::fill() const
詰め文字(fillch)を参照します。
リターン値は、詰め文字(fillch) です。

char ios_base::fill(char ch)
　ch を詰め文字として設定します。
　リターン値は、設定前の詰め文字(fillch)です。

int ios_base::precision() const
　精度(prec)を参照します。
　リターン値は、精度(prec) です。

streamsize ios_base::precision(streamsize preci)
　preci を精度(prec)に設定します。
　リターン値は、設定前の精度(prec) です。

streamsize ios_base::width() const
　フィールド幅(wide)を参照します。
　リターン値は、フィールド幅(wide) です。

streamsize ios_base::width(streamsize wd)
　wd をフィールド幅(wide)に設定します。
　リターン値は、設定前のフィールド幅(wide) です。

(c) ios クラス

種別	定義名	説明
変数	sb	streambuf オブジェクトへのポインタです。
	tiestr	ostream オブジェクトへのポインタです。
	state	streambuf への状態フラグです。
関数	ios()	コンストラクタです。
	ios(streambuf* sbptr)	
	void init(streambuf* sbptr)	初期設定を行います。
	virtual ~ios()	デストラクタです。
	operator void*() const	エラー有無(!state&(badbit | failbit))を判定します。
	bool operator!() const	エラー有無(state&(badbit | failbit))を判定します。
	iostate rdstate() const	状態フラグ(state)を参照します。
	void clear(iostate st = goodbit)	指定された状態(st)を除いて状態フラグ(state)をクリアします。
	void setstate(iostate st)	st を状態フラグ(state)に設定します。
	bool good() const	エラー有無(state==goodbit)を判定します。
	bool eof() const	入力ストリームの最後かどうか(state&eofbit)を判定します。
	bool bad() const	エラー有無(state&badbit)を判定します。
	bool fail() const	入力テキストが要求パターンと不一致であるかどうか(state&(badbit | failbit))判定します。
	ostream* tie() const	ostream オブジェクトへのポインタ(tiestr)を参照します。
	ostream* tie(ostream* tstrptr)	tstrptr を ostream オブジェクトへのポインタ(tiestr)に設定します。
	streambuf* rdbuf() const	streambuf オブジェクトへのポインタ(sb)を参照します。
	streambuf* rdbuf(streambuf* sbptr)	sbptr を streambuf オブジェクトへのポインタ(sb)に設定します。
	ios& copyfmt(const ios& rhs)	rhs の状態フラグ(state)をコピーします。

ios::ios()
クラス ios のコンストラクタです。
init(0)を呼び出し、初期値をそのメンバオブジェクトに設定します。

ios::ios(streambuf* sbptr)
クラス ios のコンストラクタです。
init(sbptr)を呼び出し、初期値をそのメンバオブジェクトに設定します。

void ios::init(streambuf* sbptr)
sbptr を sb に設定します。
state、tiestr を 0 に設定します。

virtual ios::~ios()
クラス ios のデストラクタです。

ios::operator void*() const
エラー有無(!state&(badbit | failbit))を判定します。
リターン値は次のとおりです。
エラー有の場合：false
エラー無の場合：true

bool ios::operator!() const
エラー有無(state&(badbit | failbit))を判定します。
リターン値は次のとおりです。
エラー有の場合：true
エラー無の場合：false

iostate ios::rdstate() const
状態フラグ(state) を参照します。
リターン値は、状態フラグ(state)です。

void ios::clear(iostate st = goodbit)
指定された状態(st)を除いて状態フラグ(state)をクリアします。
streambuf オブジェクトへのポインタ(sb)が 0 のときは、状態フラグ(state)に badbit を設定します。

void ios::setstate(iostate st)
st を状態フラグ(state)に設定します。

bool ios::good() const
エラー有無(state==goodbit)を判定します。
リターン値は次のとおりです。
エラー有の場合：false
エラー無の場合：true

bool ios::eof() const
入力ストリームの最後かどうか(state&eofbit)を判定します。
リターン値は次のとおりです。
入力ストリームの最後の場合　：true
入力ストリームの最後以外の場合：false

bool ios::bad() const
エラー有無(state&badbit)を判定します。
リターン値は次のとおりです。
エラー有の場合：true
エラー無の場合：false

bool ios::fail() const
入力テキストが要求パターンと不一致であるかどうか(state&(badbit | failbit))を判定します。
リターン値は次のとおりです。
不一致の場合：true
一致の場合　：false

ostream* ios::tie() const
ostream オブジェクトへのポインタ(tiestr)を参照します。
リターン値は、ostream オブジェクトへのポインタ(tiestr)です。

ostream* ios::tie(ostream* tstrptr)
tstrptr を ostream オブジェクトへのポインタ(tiestr)に設定します。
リターン値は、設定前の ostream オブジェクトへのポインタ(tiestr) です。

streambuf* ios::rdbuf() const
streambuf オブジェクトへのポインタ(sb) を参照します。
リターン値は、streambuf オブジェクトへのポインタ(sb) です。

streambuf* ios::rdbuf(streambuf* sbptr)
sbptr を streambuf オブジェクトへのポインタ(sb)に設定します。
リターン値は、設定前の streambuf オブジェクトへのポインタ(sb) です。

ios& ios::copyfmt(const ios& rhs)
rhs の状態フラグ(state)をコピーします。
リターン値は*this です。

(d) ios クラスマニピュレータ

種別	定義名	説明
関数	ios_base& boolalpha(ios_base& str)	bool 型の書式に設定します。
	ios_base& noboolalpha(ios_base& str)	bool 型の書式をクリアします。
	ios_base& showbase(ios_base& str)	基数表示接頭辞モードに設定します。
	ios_base& showpoint(ios_base& str)	小数点生成モードに設定します。
	ios_base& noshowpoint(ios_base& str)	小数点生成モードをクリアします。
	ios_base& showpos(ios_base& str)	+記号生成モードに設定します。
	ios_base& noshowpos(ios_base& str)	+記号生成モードをクリアします。
	ios_base& skipws(ios_base& str)	空白読み飛ばしモードに設定します。
	ios_base& noskipws(ios_base& str)	空白読み飛ばしモードをクリアします。
	ios_base& uppercase(ios_base& str)	大文字変換モードに設定します。
	ios_base& nouppercase(ios_base& str)	大文字変換モードをクリアします。
	ios_base& internal(ios_base& str)	内部補充モードに設定します。
	ios_base& left(ios_base& str)	p左側補充モードに設定します。
	ios_base& right(ios_base& str)	右側補充モードに設定します。
	ios_base& dec(ios_base& str)	10 進モードに設定します。
	ios_base& hex(ios_base& str)	16 進モードに設定します。
	ios_base& oct(ios_base& str)	8 進モードに設定します。
	ios_base& fixed(ios_base& str)	固定小数点モードに設定します。
	ios_base& scientific(ios_base& str)	科学表記法モードに設定します。

ios_base& boolalpha(ios_base& str)
bool 型の書式に設定します。
リターン値は str です。

ios_base& noboolalpha(ios_base& str)
bool 型の書式をクリアします。
リターン値は str です。

ios_base& showbase(ios_base& str)
データのはじめに基数を表示させるモードに設定します。
16 進数のときは、0x を行の先頭に付加します。10 進数のときは、そのまま出力します。
8 進数のときは、0 を行の先頭に付加します。
リターン値は str です。

ios_base& noshowbase(ios_base& str)
データのはじめに基数を表示させるモードをクリアします。
リターン値は str です。

ios_base& showpoint(ios_base& str)
小数点を出力するモードに設定します。
精度の指定がない場合、小数点以下 6 桁で表示します。
リターン値は str です。

ios_base& noshowpoint(ios_base& str)
小数点を出力するモードをクリアします。
リターン値は str です。

ios_base& showpos(ios_base& str)
+記号生成出力モード(正の数に対して+の符号を付加)に設定します。
リターン値は str です。

ios_base& noshowpos(ios_base& str)
+記号生成出力モードをクリアします。
リターン値は str です。

ios_base& skipws(ios_base& str)
空白読み飛ばし入力モード(連続する空白をスキップ)に設定します。
リターン値は str です。

ios_base& noskipws(ios_base& str)
空白読み飛ばし入力モードをクリアします。
リターン値は str です。

ios_base& uppercase(ios_base& str)
大文字変換出力モードに設定します。
16 進の基数表現が大文字の 0X になり、数値自体も大文字になります。
浮動小数点の指数表現も大文字の E になります。
リターン値は str です。

ios_base& nouppercase(ios_base& str)
大文字変換出力モードをクリアします。
リターン値は str です。

ios_base& internal(ios_base& str)
フィールド幅(wide)の範囲で出力時に
符号、基数
詰め文字(fill)
数値
の順で出力します。
リターン値は str です。

ios_base& left(ios_base& str)
フィールド幅(wide)の範囲で出力時に左詰めします。
リターン値は str です。

ios_base& right(ios_base& str)
フィールド幅(wide)の範囲で出力時に右詰めします。
リターン値は str です。

ios_base& dec(ios_base& str)
　変換基数を 10 進モードに設定します。
　リターン値は str です。

ios_base& hex(ios_base& str)
　変換基数を 16 進モードに設定します。
　リターン値は str です。

ios_base& oct(ios_base& str)
　変換基数を 8 進モードに設定します。
　リターン値は str です。

ios_base& fixed(ios_base& str)
　固定小数点出力モードに設定します。
　リターン値は str です。

ios_base& scientific(ios_base& str)
　科学表記法出力モード(指数表記)に設定します。
　リターン値は str です。

(e) streambuf クラス

種別	定義名	説明
定数	eof	ファイル終了を示します。
変数	_B_cnt_ptr	バッファの有効データ長へのポインタです。
	B_beg_ptr	バッファのベースポインタへのポインタです。
	_B_len_ptr	バッファの長さへのポインタです。
	B_next_ptr	バッファの次の読み出し位置へのポインタです。
	B_end_ptr	バッファの終端位置へのポインタです。
	B_beg_pptr	制御バッファの先頭位置へのポインタです。
	B_next_pptr	バッファの次の読み出し位置へのポインタです。
	C_flg_ptr	ファイルの入出力制御フラグへのポインタです。
関数	char* _ec2p_getflag() const	ファイル入出力制御フラグのポインタを参照します。
	char*& _ec2p_gnptr()	バッファの次の読み出し位置へのポインタを参照します。
	char*& _ec2p_pnptr()	バッファの次の書き込み位置へのポインタを参照します。
	void _ec2p_bcntplus()	バッファの有効データ長をインクリメントします。
	void _ec2p_bcntminus()	バッファの有効データ長をデクリメントします。
	void _ec2p_setbPtr(char** begptr, char** curptr, long* cntptr, long* lenptr, char* flgptr)	streambuf のポインタを設定します。
	streambuf()	コンストラクタです。
	virtual ~streambuf()	デストラクタです。
	streambuf* pubsetbuf(char* s, streamsize n)	ストリーム入出力用のバッファを確保します。この関数では setbuf(s,n)*1 を呼び出します。
	pos_type pubseekoff(off_type off, ios _base::seekdir way, ios_base::openmode which = ios_base::in | ios_base::out)	way で指定された方法で入出力ストリームの読み書き位置を移動させます。この関数では seekoff(off,way,which)*1 を呼び出します。
	pos_type pubseekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out)	ストリームの先頭から現在の位置までのオフセットを求めます。この関数では seekpos(sp,which)*1 を呼び出します。
	int pubsync()	出力ストリームをフラッシュします。この関数では sync()*1 を呼び出します。
	streamsize in_avail()	入力ストリームの最後尾から現在位置までのオフセットを求めます。
	int_type snextc()	次の一文字を読み込みます。
	int_type sbumpc()	一文字読み込みポインタを次に設定します。
	int_type sgetc()	一文字読み込みます。

種別	定義名	説明
関数	int sgetn(char* s, streamsize n)	s の指す記憶領域に n 個の文字を設定します。
	int_type sputbackc(char c)	読み込み位置をプットバックします。
	int sungetc()	読み込み位置をプットバックします。
	int sputc(char c)	文字 c を挿入します。
	int_type sputn(const char* s, streamsize n)	s の指す n 個の文字を挿入します。
	char* eback() const	入力ストリームの先頭ポインタを求めます。
	char* gptr() const	入力ストリームの次ポインタを求めます。
	char* egptr() const	入力ストリームの最後尾ポインタを求めます。
	void gbump(int n)	入力ストリームの次ポインタを n 進めます。
	void setg(char* gbeg, char* gnext, char* gend)	入力ストリームの各ポインタを代入します。
	char* pbase() const	出力ストリームの先頭ポインタを求めます。
	char* pptr() const	出力ストリームの次ポインタを求めます。
	char* epptr() const	出力ストリームの最後尾ポインタを求めます。
	void pbump(int n)	出力ストリームの次ポインタを n 進めます。
	void setp(char* pbeg, char* pend)	出力ストリームの各ポインタを設定します。
	virtual streambuf* setbuf(char* s, streamsize n) *1	派生する各クラスごとに、個別に定義する演算を実行します。
	virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out)) *1	ストリーム位置を変更します。
	virtual pos_type seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out)) *1	ストリーム位置を変更します。
	virtual int sync() *1	出力ストリームをフラッシュします。
	virtual int showmanyc() *1	入力ストリームの有効な文字数を求めます。
	virtual streamsize xsgetn(char* s, streamsize n)	s の指す記憶領域に n 個の文字を設定します。
	virtual int_type underflow() *1	ストリーム位置を動かさずに一文字読み込みます。
	virtual int_type uflow() *1	次ポインタの一文字を読み込みます。
	virtual int_type pbackfail(int_type c = eof) *1	c によって示される文字をプットバックします。
	virtual streamsize xsputn(const char* s, streamsize n)	s の指す n 個の文字を挿入します。
	virtual int_type overflow(int_type c = eof) *1	c を出力ストリームに挿入します。

【注】 *1 このクラスでは処理を定義していません。

streambuf::streambuf()
コンストラクタです。
以下の値で初期化します。
_B_cnt_ptr = B_beg_ptr = B_next_ptr = B_end_ptr = C_flg_ptr =_B_len_ptr = 0
B_beg_pptr = &B_beg_ptr
B_next_pptr = &B_next_ptr

virtual streambuf::~streambuf()
デストラクタです。

streambuf* streambuf::pubsetbuf(char* s, streamsize n)
ストリーム入出力用のバッファを確保します。
この関数では setbuf(s,n)を呼び出します。
リターン値は、setbuf(s,n) です。

pos_type streambuf::pubseekoff(off_type off, ios _base::seekdir way,
ios_base::openmode which = (ios_base::openmode)(ios_base::in | ios_base::out))
way で指定された方法で入出力ストリームの読み書き位置を移動させます。
この関数では seekoff(off,way,which)を呼び出します。
リターン値は、新たに設定されたストリームの位置です。

pos_type streambuf::pubseekpos(pos_type sp, ios_base::openmode which =
(ios_base::openmode)(ios_base::in | ios_base::out))
ストリームの先頭から現在の位置までのオフセットを求めます。
現在のストリームポインタから sp だけ移動します。
この関数では seekpos(sp,which)を呼び出します。
リターン値は、先頭からのオフセットです。

int streambuf::pubsync()
出力ストリームをフラッシュします。
この関数では sync()を呼び出します。
リターン値は 0 です。

streamsize streambuf::in_avail()
入力ストリームの最後尾から現在位置までのオフセットを求めます。
リターン値は次のとおりです。
読み込み位置が有効の場合　：最後尾から現在位置までのオフセット
読み込み位置が有効でない場合：0(showmanyc()を呼び出します)

int_type streambuf::snextc()
一文字読み込みます。読み込んだ文字が eof でなければ、次の一文字を読み込みます。
リターン値は次のとおりです。
eof でない場合：読み込んだ文字
eof の場合　：eof

int_type streambuf::sbumpc()
一文字読み込みポインタを次に設定します。

リターン値は次のとおりです。
読み込み位置が無効でない場合：読み込んだ文字
読み込み位置が無効の場合　：eof

int_type streambuf::sgetc()
一文字読み込みます。
リターン値は次のとおりです。
読み込み位置が無効でない場合：読み込んだ文字
読み込み位置が無効の場合　：eof

int streambuf::sgetn(char* s, streamsize n)
s の指す記憶領域に n 個の文字を設定します。
文字列中に eof を検出した場合、設定を終了します。
リターン値は、設定した文字数です。

int_type streambuf::sputbackc(char c)
読み込み位置が正常で読み込み位置のプットバックデータが c と同一の場合、読み込み位置をプットバックします。
リターン値は次のとおりです。
プットバックできた場合　：c の値
プットバックできなかった場合：eof

int streambuf::sungetc()
読み込み位置が正常である場合、読み込み位置をプットバックします。
リターン値は次のとおりです。
プットバックできた場合　：プットバックした値
プットバックできなかった場合：eof

int streambuf::sputc(char c)
文字 c を挿入します。
リターン値は次のとおりです。
書き込み位置が正しい場合：c の値
書き込み位置が不正な場合：eof

int_type streambuf::sputn(const char* s, streamsize n)
s の指す n 個の文字を挿入します。
バッファが n より小さい場合は、バッファサイズ分だけ挿入します。
リターン値は、挿入された文字数です。

char* streambuf::eback() const
入力ストリームの先頭ポインタを求めます。
リターン値は、先頭ポインタです。

char* streambuf::gptr() const
入力ストリームの次ポインタを求めます。
リターン値は、次ポインタです。

char* streambuf::egptr() const
入力ストリームの最後尾ポインタを求めます。
リターン値は、最後尾ポインタです。

void streambuf::gbump(int n)
入力ストリームの次ポインタを n 進めます。

void streambuf::setg(char* gbeg, char* gnext, char* gend)
入力ストリームの各ポインタに、以下の設定を行います。
*B_beg_pptr = gbeg;
*B_next_pptr = gnext;
B_end_ptr = gend;
*_B_cnt_ptr = gend-gnext;
*_B_len_ptr = gend-gbeg;

char* streambuf::pbase() const
出力ストリームの先頭ポインタを求めます。
リターン値は、先頭ポインタです。

char* streambuf::pptr() const
出力ストリームの次ポインタを求めます。
リターン値は、次ポインタです。

char* streambuf::epptr() const
出力ストリームの最後尾ポインタを求めます。
リターン値は、最後尾ポインタです。

void streambuf::pbump(int n)
出力ストリームの次ポインタを n 進めます。

void streambuf::setp(char* pbeg, char* pend)
出力ストリームの各ポインタに、以下の設定を行います。
*B_beg_pptr = pbeg;
*B_next_pptr = pbeg;
B_end_ptr = pend;
*_B_cnt_ptr=pend-pbeg;
*_B_len_ptr=pend-pbeg;

virtual streambuf* streambuf::setbuf(char* s, streamsize n)
streambuf から派生する各クラスごとに、個別に定義する演算を実行します。
リターン値は*this です。このクラスでは処理を定義していません。

virtual pos_type streambuf::seekoff(off_type off, ios_base::seekdir way, ios_base::openmode =
(ios_base::openmode)(ios_base::in | ios_base::out))
ストリーム位置を変更します。
リターン値は(-1) です。このクラスでは処理を定義していません。

virtual pos_type streambuf::seekpos(pos_type off, ios_base::openmode =
(ios_base::openmode)(ios_base::in | ios_base::out))
ストリーム位置を変更します。
リターン値は(-1) です。このクラスでは処理を定義していません。

virtual int streambuf::sync()
出力ストリームをフラッシュします。
リターン値は 0 です。このクラスでは処理を定義していません。

virtual int streambuf::showmanyc()
入力ストリームの有効な文字数を求めます。
リターン値は 0 です。このクラスでは処理を定義していません。

virtual streamsize streambuf::xsgetn(char* s, streamsize n)
s の指す記憶領域に n 個の文字を設定します。
バッファが n より小さい場合は、バッファサイズ分だけ設定します。
リターン値は、入力された文字数です。

virtual int_type streambuf::underflow ()
ストリーム位置を動かさずに一文字読み込みます。
リターン値は eof です。このクラスでは処理を定義していません。

virtual int_type streambuf::uflow()
次ポインタの一文字を読み込みます。
リターン値は eof です。このクラスでは処理を定義していません。

virtual int_type streambuf::pbackfail(int_type c = eof)
c によって示される文字をプットバックします。
リターン値は eof です。このクラスでは処理を定義していません。

virtual streamsize streambuf::xsputn(const char* s, streamsize n)
s の指す n 個の文字を挿入します。
バッファが n より小さい場合は、バッファサイズ分だけ挿入します。
リターン値は、挿入された文字数です。

virtual int_type streambuf::overflow(int_type c = eof)
c を出力ストリームに挿入します。
リターン値は eof です。このクラスでは処理を定義していません。

(f) istream::sentry クラス

種別	定義名	説明
変数	ok_	入力可能状態か否かを意味します。
関数	sentry(istream& is, bool noskipws = false)	コンストラクタです。
	~sentry()	デストラクタです。
	operator bool()	ok_を参照します。

istream::sentry::sentry(istream& is, bool noskipws = _false)
内部クラス sentry のコンストラクタです。
good()が非 0 の場合、フォーマット付きまたはフォーマットなし入力を可能にします。
tie()が非 0 の場合、関連する出力ストリームをフラッシュします。

istream::sentry::~sentry()
内部クラス sentry のデストラクタです。

istream::sentry::operator bool()
ok_を参照します。
リターン値は ok_です。

(g) istream クラス

種別	定義名	説明
変数	chcount	最後にコールされた入力関数が抽出した文字数です。
関数	int _ec2p_getistr(char* str, unsigned int dig, int mode)	str を dig が示す基数で変換します。
	istream(streambuf* sb)	コンストラクタです。
	virtual ~istream()	デストラクタです。
	istream& operator>>(bool& n)	抽出した文字を n に格納します。
	istream& operator>>(short& n)	
	istream& operator>>(unsigned short& n)	
	istream& operator>>(int& n)	
	istream& operator>>(unsigned int& n)	
	istream& operator>>(long& n)	
	istream& operator>>(unsigned long& n)	
	istream& operator>>(float& n)	
	istream& operator>>(double& n)	
	istream& operator>>(long double& n)	
	istream& operator>>(void*& p)	void を指すポインタに変換して p に格納します。
	istream& operator>>(streambuf* sb)	文字を抽出し、sb の指す記憶領域へ格納します。
	streamsize gcount()	constchcount(抽出文字数)を求めます。
	int_type get()	文字を抽出します。
	istream& get(char& c)	文字を抽出し c に格納します。
	istream& get(signed char& c)	
	istream& get(unsigned char& c)	
	istream& get(char* s, streamsize n)	サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
	istream& get(signed char* s, streamsize n)	
	istream& get(unsigned char* s, streamsize n)	
	istream& get(char* s, streamsize n, char delim)	サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。文字列内に'delim'を検出したら、入力を終了します。
	istream& get(signed char* s, streamsize n, char delim)	
	istream& get(unsigned char* s, streamsize n, char delim)	
	istream& get(streambuf& sb)	文字列を抽出し、sb の指す記憶領域に格納します。
	istream& get(streambuf& sb, char delim)	文字列を抽出し、sb の指す記憶領域に格納します。途中で文字'delim'を検出したら、入力を終了します。
	istream& getline(char* s, streamsize n)	サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
	istream& getline(signed char* s, streamsize n)	
	istream& getline(unsigned char* s, streamsize n)	

種別	定義名	説明
関数	istream& getline(char* s, streamsize n, char delim)	サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。途中で文字'delim'を検出したら、入力を終了します。
	istream& getline(signed char* s, streamsize n, char delim)	
	istream& getline(unsigned char* s, streamsize n, char delim)	
	istream& ignore(streamsize n = 1, int_type delim = streambuf::eof)	n 個の文字を読み飛ばします。途中で文字'delim'を検出したら、読み飛ばし処理を中止します。
	int_type peek()	次の入手可能な入力文字を求めます。
	istream& read(char* s, streamsize n)	サイズ n の文字列を抽出し、s の指す記憶領域に格納します。
	istream& read(signed char* s, streamsize n)	
	istream& read(unsigned char* s, streamsize n)	
	streamsize readsome(char* s, streamsize n)	サイズ n の文字列を抽出し、s の指す記憶領域に格納します。
	streamsize readsome(signed char* s, streamsize n)	
	streamsize readsome(unsigned char* s, streamsize n)	
	istream& putback(char c)	文字を入力ストリームに戻します。
	istream& unget()	入力ストリームの位置を戻します。
	int sync()	入力ストリームがあるかどうかを調べます。この関数は streambuf::pubsync()を呼び出します。
	pos_type tellg()	入力ストリームの位置を調べます。この関数は streambuf::pubseekoff(0,cur,in)を呼び出します。
	istream& seekg(pos_type pos)	現在のストリームポインタから pos だけ移動します。この関数は streambuf::pubseekpos(pos)を呼び出します。
	istream& seekg(off_type off, ios_base::seekdir dir)	dir で指定された方法で入力ストリームの読み込み位置を移動します。この関数は streambuf::pubseekoff(off,dir)を呼び出します。

int istream::_ec2p_getistr(char* str, unsigned int dig, int mode)

str を dig が示す基数で変換します。

リターン値は、変換した基数です。

istream::istream(streambuf* sb)

クラス istream のコンストラクタです。

ios::init(sb)を呼び出します。

chcount=0 の設定を行います。

virtual istream::~istream ()
クラス istream のデストラクタです。

istream& istream::operator>>(bool& n)
istream& istream::operator>>(short& n)
istream& istream::operator>>(unsigned short& n)
istream& istream::operator>>(int& n)
istream& istream::operator>>(unsigned int& n)
istream& istream::operator>>(long& n)
istream& istream::operator>>(unsigned long& n)
istream& istream::operator>>(float& n)
istream& istream::operator>>(double& n)
istream& istream::operator>>(long double& n)
抽出した文字を n に格納します。
リターン値は*this です。

istream& istream::operator>>(void*& p)
抽出した文字を void*型に変換し、p の指す記憶領域に格納します。
リターン値は*this です。

istream& istream::operator>>(streambuf* sb)
文字を抽出し、sb の指す記憶領域に格納します。
抽出文字がない場合は、setstate(failbit)を呼び出します。
リターン値は*this です。

streamsize istream::gcount() const
chcount(抽出文字数)を参照します。
リターン値は chcount です。

int_type istream::get()
文字を抽出します。
リターン値は次のとおりです。
抽出可能の場合：抽出した文字
抽出不可の場合：setstate(failbat)を呼び出して、streambuf::eof

istream& istream::get (char& c)
istream& istream::get(signed char& c)
istream& istream::get(unsigned char& c)
文字を抽出し c に格納します。抽出した文字が streambuf::eof の場合は、failbit を設定します。
リターン値は*this です。

istream& istream::get(char* s, streamsize n)
istream& istream::get(signed char* s, streamsize n)
istream& istream::get(unsigned char* s, streamsize n)
サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
リターン値は*this です。

istream& istream::get(char* s, streamsize n, char delim)
istream& istream::get(signed char* s, streamsize n, char delim)
istream& istream::get(unsigned char* s, streamsize n, char delim)
サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
文字列内に‘delim’を検出したら、終了します。
ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
リターン値は*this です。

istream& istream::get(streambuf& sb)
文字列を抽出し、sb の指す記憶領域に格納します。
ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
リターン値は*this です。

istream& istream::get(streambuf& sb, char delim)
文字列を抽出し、sb の指す記憶領域に格納します。
途中で文字‘delim’を検出したら、終了します。
ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
リターン値は*this です。

istream& istream::getline(char* s, streamsize n)
istream& istream::getline(signed char* s, streamsize n)
istream& istream::getline(unsigned char* s, streamsize n)
サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
リターン値は*this です。

istream& istream::getline(char* s, streamsize n, char delim)
istream& istream::getline(signed char* s, streamsize n, char delim)
istream& istream::getline(unsigned char* s, streamsize n, char delim)
サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
途中で文字‘delim’を検出したら、終了します。
ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
リターン値は*this です。

istream& istream::ignore(streamsize n = 1, int_type delim = streambuf::eof)
n 個の文字を読み飛ばします。
途中で文字‘delim’を検出したら、読み飛ばし処理を中止します。
リターン値は*this です。

int_type istream::peek()
次の入力可能な入力文字を求めます。
リターン値は次のとおりです。
ok_==false の場合：streambuf::eof
ok_!=false の場合：rdbuf()->sgetc()

istream& istream::read(char* s, streamsize n)
istream& istream::read(signed char* s, streamsize n)
istream& istream::read(unsigned char* s, streamsize n)
ok_!=false の場合、サイズ n の文字列を抽出し、s の指す記憶領域に格納します。
抽出した文字数が n と異なる場合、eofbit を設定します。
リターン値は*this です。

streamsize istream::readsome(char* s, streamsize n)
streamsize istream::readsome (signed char* s, streamsize n)
streamsize istream::readsome (unsigned char* s, streamsize n)
サイズ n の文字列を抽出し、s の指す記憶領域に格納します。
文字数がストリームサイズより大きければ、ストリームサイズ分格納します。
リターン値は、抽出した文字数です。

istream& istream::putback(char c)
文字 c を入力ストリームに戻します。プットバックした文字が streambuf::eof の場合は、badbit を設定します。
リターン値は*this です。

istream& istream::unget()
入力ストリームのポインタをひとつ戻します。
抽出した文字が streambuf::eof の場合、badbit を設定します。
リターン値は*this です。

int istream::sync()
入力ストリームがあるかどうかを調べます。
この関数は streambuf::pubsync()を呼び出します。
リターン値は次のとおりです。
入力ストリームがない場合：streambuf::eof
入力ストリームがある場合：0

pos_type istream::tellg()
入力ストリームの位置を調べます。
この関数は streambuf::pubseekoff(0,cur,in)を呼び出します。
リターン値は次のとおりです。
ストリームの先頭からのオフセット
ただし、入力処理にエラーが発生した場合は-1

istream& istream::seekg(pos_type pos)
現在のストリームポインタから pos だけ移動します。
この関数は streambuf::pubseekpos(pos)を呼び出します。
リターン値は*this です。

istream& istream::seekg(off_type off, ios_base::seekdir dir)
dir で指定された方法で入力ストリームの読み込み位置を移動します。
この関数は streambuf::pubseekoff(off,dir)を呼び出します。
入力処理にエラーがある場合は処理は行いません。
リターン値は*this です。

(h) istream クラスマニピュレータ

種別	定義名	説明
関数	istream& ws(istream& is)	空白文字を読み飛ばします。

istream& ws(istream& is)

空白類を読み飛ばします。

リターン値は is です。

(i) istream メンバ外関数

種別	定義名	説明
関数	istream& operator>>(istream& in, char* s)	文字列を抽出し、s の指す記憶領域に格納します。
	istream& operator>>(istream& in, signed char* s)	
	istream& operator>>(istream& in, unsigned char* s)	
	istream& operator>>(istream& in, char& c)	文字を抽出し、c に格納します。
	istream& operator>>(istream& in, singed char& c)	
	istream& operator>>(istream& in, unsigned char& c)	

istream& operator>>(istream& in, char* s)
istream& operator>>(istream& in, signed char* s)
istream& operator>>(istream& in, unsigned char* s)
文字列を抽出し、s の指す記憶領域に格納します。
(フィールド幅-1)個の文字を格納したか、または入力ストリームに streambuf::eof が現れたか、または次の入力可能な文字 c が isspace(c)=1 の場合、処理は終了します。格納文字数が 0 の場合は failbit を設定します。
リターン値は in です。

istream& operator>>(istream& in, char& c)
istream& operator>>(istream& in, singed char& c)
istream& operator>>(istream& in, unsigned char& c)
文字を抽出し、c に格納します。
抽出入力がない場合、failbit を設定します。
リターン値は in です。

(j) ostream::sentry クラス

種別	定義名	説明
変数	ok_	出力可能状態か否かを意味します。
	__ec2p_os	ostream オブジェクトへのポインタです。
関数	sentry(ostream& os)	コンストラクタです。
	~sentry()	デストラクタです。
	operator bool()	ok_を参照します。

ostream::sentry::sentry (ostream& os)

内部クラス sentry のコンストラクタです。

good()が非 0 かつ tie()が非 0 なら flush()を呼び出します。__ec2p_os に os を設定します。

ostream::sentry::~sentry ()

内部クラス sentry のデストラクタです。

__ec2p_os->flags() & ios_base::unitbuf が真なら、flush()を呼び出します。

ostream::sentry::operator bool ()

ok_を参照します。

リターン値は ok_です。

(k) ostream クラス

種別	定義名	説明
関数	ostream(streambuf* sbptr)	コンストラクタです。
	virtual ~ostream()	デストラクタです。
	ostream& operator<<(bool n)	n を出力ストリームに挿入します。
	ostream& operator<<(short n)	
	ostream& operator<<(unsigned short n)	
	ostream& operator<<(int n)	
	ostream& operator<<(unsigned int n)	
	ostream& operator<<(long n)	
	ostream& operator<<(unsigned long n)	
	ostream& operator<<(float n)	
	ostream& operator<<(double n)	
	ostream& operator<<(long double n)	
	ostream& operator<<(void* n)	
	ostream& operator<<(streambuf* sbptr)	sbptr の出力列を出力ストリームに挿入します。
	ostream& put(char c)	文字 c を出力ストリームに挿入します。
	ostream& write(const char* s, streamsize n)	s の n 個の文字を出力ストリームに挿入します。
	ostream& write(const signed char* s, streamsize n)	
	ostream& write(const unsigned char* s, streamsize n)	
	ostream& flush()	出力ストリームをフラッシュします。この関数は streambuf::pubsync()を呼び出します。
	pos_type tellp()	現在の書き込み位置を求めます。この関数は streambuf::pubseekoff(0,cur,out)を呼び出します。
	ostream& seekp(pos_type pos)	ストリームの先頭から現在の位置までのオフセットを求めます。現在のストリームポインタから pos だけ移動します。この関数は streambuf::pubseekpos(pos)を呼び出します。
	ostream& seekp(off_type off, seekdir dir)	dir を基準として、ストリームの書き込み位置を off 分だけ移動します。この関数は streambuf::pubseekoff(off,dir)を呼び出します。

ostream::ostream(streambuf* sbptr)
　コンストラクタです。
　ios (sbptr)を呼び出します。

virtual ostream::~ostream()
　デストラクタです。

ostream& ostream::operator<<(bool n)
ostream& ostream::operator<<(short n)
ostream& ostream::operator<<(unsigned short n)
ostream& ostream::operator<<(int n)
ostream& ostream::operator<<(unsigned int n)
ostream& ostream::operator<<(long n)
ostream& ostream::operator<<(unsigned long n)
ostream& ostream::operator<<(float n)
ostream& ostream::operator<<(double n)
ostream& ostream::operator<<(long double n)
ostream& ostream::operator<<(void* n)
　sentry::ok_==true のとき、n を出力ストリームに挿入します。
　sentry::ok_==false のとき、failbit を設定します。
　リターン値は*this です。

ostream& ostream::operator<<(streambuf* sbptr)
　sentry::ok_==true のとき、sbptr の出力列を出力ストリームに挿入します。
　sentry::ok_==false のとき、failbit を設定します。
　リターン値は*this です。

ostream& ostream::put(char c)
　sentry::ok_==true かつ rdbuf()->sputc(c)!=streambuf::eof のとき、c を出力ストリームに挿入します。
　上記以外のとき、badbit を設定します。
　リターン値は*this です。

ostream& ostream::write(const char* s, streamsize n)
ostream& ostream::write(const signed char* s, streamsize n)
ostream& ostream::write(const unsigned char* s, streamsize n)
　sentry::ok_==true かつ rdbuf()->sputn(s, n)==n のとき、s の n 個の文字を出力ストリームに挿入します。
　上記以外のとき、badbit を設定します。
　リターン値は*this です。

ostream& ostream::flush()
　出力ストリームをフラッシュします。
　この関数は streambuf::pubsync()を呼び出します。
　リターン値は*this です。

pos_type ostream::tellp()
　現在の書き込み位置を求めます。

この関数は streambuf::pubseekoff(0,cur,out)を呼び出します。
リターン値は次のとおりです。
現在のストリームの位置
ただし、処理中にエラーが発生した場合は-1

ostream& ostream::seekp(pos_type pos)
エラーがないとき、ストリームの先頭から現在の位置までのオフセットを求めます。
また、現在のストリームポインタから pos だけ移動します。
この関数は streambuf::pubseekpos(pos)を呼び出します。
リターン値は*this です。

ostream& ostream::seekp(off_type off, seekdir dir)
エラーがないとき、dir を基準として off 分ストリームの位置を移動します。
この関数は streambuf::pubseekoff(off,dir)を呼び出します。
リターン値は*this です。

(l) ostream クラスマニピュレータ

種別	定義名	説明
関数	ostream& endl(ostream& os)	改行を挿入し、出力ストリームをフラッシュします。
	ostream& ends(ostream& os)	ヌルコードを挿入します。
	ostream& flush(ostream& os)	出力ストリームをフラッシュします。

ostream& endl(ostream& os)

ストリームに改行文字を挿入します。
出力ストリームをフラッシュします。この関数は flush()を呼び出します。
リターン値は os です。

ostream& ends(ostream& os)

出力ストリームにヌルコードを挿入します。
リターン値は os です。

ostream& flush(ostream& os)

出力ストリームをフラッシュします。この関数は streambuf::sync()を呼び出します。
リターン値は os です。

(m) ostream メンバ外関数

種別	定義名	説明
関数	ostream& operator<<(ostream& os, char s)	s を出力ストリームに挿入します。
	ostream& operator<<(ostream& os, signed char s)	
	ostream& operator<<(ostream& os, unsigned char s)	
	ostream& operator<<(ostream& os, const char* s)	
	ostream& operator<<(ostream& os, const singed char* s)	
	ostream& operator<<(ostream& os, const unsigned char* s)	

ostream& operator<<(ostream& os, char s)
ostream& operator<<(ostream& os, signed char s)
ostream& operator<<(ostream& os, unsigned char s)
ostream& operator<<(ostream& os, const char* s)
ostream& operator<<(ostream& os, const singed char* s)
ostream& operator<<(ostream& os, const unsigned char* s)

sentry::ok_= =true かつエラーがないとき、s を出力ストリームに挿入します。
上記以外のとき、failbit を設定します。
リターン値は os です。

(n) smanip クラスマニピュレータ

種別	定義名	説明
関数	smanip resetiosflags(ios_base::fmtflags mask)	mask 値で指定されたフラグをクリアします。
	smanip setiosflags(ios_base::fmtflags mask)	書式フラグ(fmtfl)を設定します。
	smanip setbase(int base)	出力時に用いる基数を設定します。
	smanip setfill(char c)	詰め文字(fillch)を設定します。
	smanip setprecision(int n)	精度(prec)を設定します。
	smanip setw(int n)	フィールド幅(wide)を設定します。

smanip resetiosflags(ios_base::fmtflags mask)
mask 値で指定されたフラグをクリアします。
リターン値は、入出力対象のオブジェクトです。

smanip setiosflags(ios_base::fmtflags mask)
書式フラグ(fmtfl)を設定します。
リターン値は、入出力対象のオブジェクトです。

smanip setbase(int base)
出力時に用いる基数を設定します。
リターン値は、入出力対象のオブジェクトです。

smanip setfill(char c)
詰め文字(fillch)を設定します。
リターン値は、入出力対象のオブジェクトです。

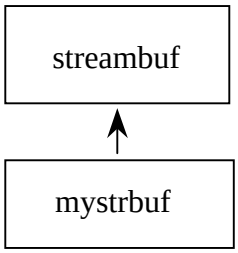
smanip setprecision(int n)
精度(prec) を設定します。
リターン値は、入出力対象のオブジェクトです。

smanip setw(int n)
フィールド幅(wide)を設定します。
リターン値は、入出力対象のオブジェクトです。

(o) EC++入出力ライブラリの使用例

istream, ostream のオブジェクトの初期化時に streambuf のかわりに mystrbuf クラスのオブジェクトへのポインタを使うことにより入出力ストリームが使用可能になります。

クラスの派生関係は次のようになります。矢印は、派生クラスから基底クラスを参照していることを示します。



種別	定義名	説明
変数	_file_Ptr	ファイルポインタです。
関数	mystrbuf()	コンストラクタです。streambuf バッファの初期化を行います。
	mystrbuf(void* ptr)	（同上）
	virtual ~mystrbuf()	デストラクタです。
	void* myfptr() const	FILE 型構造体へのポインタを返します。
	mystrbuf* open(const char* filename, int mode)	ファイル名とモードを指定して、ファイルをオープンします。
	mystrbuf* close()	ファイルのクローズを行います。
	virtual streambuf* setbuf(char* s, streamsize n)	ストリーム入出力用のバッファを確保します。
	virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))	ストリームポインタの位置を変えます。
	virtual pos_type seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))	ストリームポインタの位置を変えます。
	virtual int sync()	ストリームをフラッシュします。
	virtual int showmanyc()	入力ストリームの有効な文字数を返します。
	virtual int_type underflow()	ストリーム位置を動かさずに一文字読み込みます。
	virtual int_type pbackfail(int_type c = streambuf::eof)	c によって示される文字をプットバックします。
	virtual int_type overflow(int_type c = streambuf::eof)	c によって示される文字を挿入します。
	void _Init(_f_tye* fp)	初期処理です。

例：

```
#include <istream>
#include <ostream>
#include <mystrbuf>
#include <string>
#include <new>
void main(void)
{
    mystrbuf myfin(stdin);
    mystrbuf myfout(stdout);
    istream mycin(&myfin);
    ostream mycout(&myfout);

    int i;
    short s;
    long l;
    char c;
    string str;

    mycin >> i >> s >> l >> c >> str;
    mycout << "This is EC++ Liblary." << endl
           << i << s << l << c << str << endl;
    return;
}
```

(3) メモリ管理用ライブラリ

メモリの管理用ライブラリに対応するヘッダファイルは以下の通りです。

- <new>

メモリの確保・解放を行う関数を定義します。

_ec2p_new_handler 変数に例外処理関数のアドレスを設定することにより、メモリ確保に失敗した場合、例外処理を実行することができます。

_ec2p_new_handler は static 変数で、初期値は NULL です。このハンドラを使用することにより、リエントラント性は失われます。

例外処理関数に要求される動作：

- 割当可能な領域を作り出して返します。
- そうできない場合の動作は規定されていません。

種別	定義名	説明
型	new_handler	void 型を返す関数へのポインタ型です。
変数	_ec2p_new_handler	例外処理関数へのポインタです。
関数	void* operator new(size_t size)	size 分の領域を確保します。
	void* operator new[](size_t size)	size 分の配列領域を確保します。
	void* operator new(size_t size, void* ptr)	ptr の指している領域を記憶領域として割り当てます。
	void* operator new[](size_t size, void* ptr)	ptr の指している領域を配列領域として割り当てます。
	void operator delete(void* ptr)	領域を解放します。
	void operator delete[](void* ptr)	配列領域を解放します。
	new_handler set_new_handler(new_handler new_P)	_ec2p_new_handler に例外処理関数アドレス（new_P）を設定します。

void* operator new(size_t size)

size バイト分の領域を割り当てます。

領域割り当てに失敗し、かつ new_handler が設定されていれば、new_handler を呼び出します。

リターン値は次のとおりです。

領域確保に成功した場合：void 型へのポインタ

領域確保に失敗した場合：NULL

void* operator new [](size_t size)

size 分の配列領域を確保します。

領域割り当てに失敗し、かつ new_handler が設定されていれば、new_handler を呼び出します。

リターン値は次のとおりです。

領域確保に成功した場合：void 型へのポインタ

領域確保に失敗した場合：NULL

void* operator new(size_t size, void* ptr)

ptr の指している領域を記憶領域として割り当てます。

リターン値は ptr です。

void* operator new[](size_t size, void* ptr)
　ptr の指している領域を配列領域として割り当てます。
　リターン値は ptr です。

void operator delete(void* ptr)
　ptr が指す記憶領域を解放します。ptr が NULL のときは何もしません。

void operator delete [](void* ptr)
　ptr が指す配列領域を解放します。ptr が NULL のときは何もしません。

new_handler set_new_handler(new_handler new_P)
　_ec2p_new_handler に new_P を設定します
　リターン値は_ec2p_new_handler です。

(4) 複素数計算用クラスライブラリ

複素数計算用クラスライブラリに対応するヘッダファイルは以下の通りです。

- <complex>

float_complex クラス、double_complex クラスを定義します。

これらのクラスには派生関係はありません。

(a) float_complex クラス

種別	定義名	説明
型	value_type	float 型です。
変数	_re	float 精度の実数部を定義します。
	_im	float 精度の虚数部を定義します。
関数	float_complex(float re = 0.0f, float im = 0.0f)	コンストラクタです。
	float_complex(const double_complex& rhs)	
	float real() const	実数部(_re)を求めます。
	float imag() const	虚数部(_im)を求めます。
	float_complex& operator=(float rhs)	rhs を実数部にコピーします。虚数部は 0.0f を設定します。
	float_complex& operator+=(float rhs)	rhs を実数部に加算し、和を*this に格納します。
	float_complex& operator-=(float rhs)	rhs を実数部から減算し、差を*this に格納します。
	float_complex& operator*=(float rhs)	rhs を乗算し、積を*this に格納します。
	float_complex& operator/=(float rhs)	rhs で除算し、商を*this に格納します。
	float_complex& operator=(const float_complex& rhs)	rhs をコピーします。
	float_complex& operator+=(const float_complex& rhs)	rhs を加算し、和を*this に格納します。
	float_complex& operator-=(const float_complex& rhs)	rhs を減算し、差を*this に格納します。
	float_complex& operator*=(const float_complex& rhs)	rhs を乗算し、積を*this に格納します。
	float_complex& operator/=(const float_complex& rhs)	rhs で除算し、商を*this に格納します。

float_complex::float_complex (float re = 0.0f, float im = 0.0f)

クラス float_complex のコンストラクタです。

以下の値で初期化します。

_re = re;

_im = im;

float_complex::float_complex(const double_complex& rhs)

クラス float_complex のコンストラクタです。

以下の値で初期化します。

_re = (float)rhs.real();

_im = (float)rhs.imag();

float float_complex::real () const
実数部を求めます。
リターン値は、this->_re です。

float float_complex::imag () const
虚数部を求めます。
リターン値は、this->_im です。

float_complex& float_complex::operator= (float rhs)
rhs を実数部(_re)にコピーします。虚数部(_im)は 0.0f を設定します。
リターン値は*this です。

float_complex& float_complex::operator+= (float rhs)
rhs を実数部(_re)に加算し、結果を実数部(_re)に格納します。虚数部(_im)の値は変わりません。
リターン値は*this です。

float_complex& float_complex::operator-= (float rhs)
rhs を実数部(_re)から減算し、結果を実数部(_re)に格納します。虚数部(_im)の値は変わりません。
リターン値は*this です。

float_complex& float_complex::operator*= (float rhs)
rhs と乗算し、結果を*this に格納します。
(_re=_re*rhs, _im=_im*rhs)
リターン値は*this です。

float_complex& float_complex::operator/= (float rhs)
rhs で除算し、結果を*this に格納します。
(_re=_re/rhs, _im=_im/rhs)
リターン値は*this です。

float_complex& float_complex::operator= (const float_complex& rhs)
rhs をコピーします。
リターン値は*this です。

float_complex& float_complex::operator+= (const float_complex& rhs)
rhs を加算し、結果を*this に格納します。
リターン値は*this です。

float_complex& float_complex::operator-= (const float_complex& rhs)
rhs を減算し、結果を*this に格納します。
リターン値は*this です。

float_complex& float_complex::operator*= (const float_complex& rhs)
rhs と乗算し、結果を*this に格納します。
リターン値は*this です。

float_complex& float_complex::operator/= (const float_complex& rhs)
rhs で除算し、結果を*this に格納します。
リターン値は*this です。

(b) float_complex メンバ外関数

種別	定義名	説明
関数	float_complex operator+(const float_complex& lhs)	lhs の単項＋演算を行います。
	float_complex operator+(const float_complex& lhs, const float_complex& rhs)	lhs と rhs を加算し、和を lhs に格納します。
	float_complex operator+(const float_complex& lhs, const float& rhs)	
	float_complex operator+(const float& lhs, const float_complex& rhs)	
	float_complex operator-(const float_complex& lhs)	lhs の単項－演算を行います。
	float_complex operator-(const float_complex& lhs, const float_complex& rhs)	lhs から rhs を減算し、差を lhs に格納します。
	float_complex operator-(const float_complex& lhs, const float& rhs)	
	float_complex operator-(const float& lhs, const float_complex& rhs)	
	float_complex operator*(const float_complex& lhs, const float_complex& rhs)	lhs と rhs を乗算し、積を lhs に格納します。
	float_complex operator*(const float_complex& lhs, const float& rhs)	
	float_complex operator*(const float& lhs, const float_complex& rhs)	
	float_complex operator/(const float_complex& lhs, const float_complex& rhs)	lhs を rhs で除算し、商を lhs に格納します。
	float_complex operator/(const float_complex& lhs, const float& rhs)	
	float_complex operator/(const float& lhs, const float_complex& rhs)	
	bool operator==(const float_complex& lhs, const float_complex& rhs)	lhs と rhs の実数部どうし、虚数部どうしを比較します。
	bool operator==(const float_complex& lhs, const float& rhs)	
	bool operator==(const float& lhs, const float_complex& rhs)	

種別	定義名	説明
関数	bool operator!=(const float_complex& lhs, const float_complex& rhs)	lhs と rhs の実数部どうし、虚数部どうしを比較します。
	bool operator!=(const float_complex& lhs, const float& rhs)	
	bool operator!=(const float& lhs, const float_complex& rhs)	
	istream& operator>>(istream& is, float_complex& x)	u,(u),または(u,v)（u:実数部、v:虚数部）形式の x を入力します。
	ostream& operator<<(ostream& os, const float_complex& x)	x を u,(u)または (u,v)（u:実数部、v:虚数部）形式で出力します。
	float real(const float_complex& x)	実数部を求めます。
	float imag(const float_complex& x)	虚数部を求めます。
	float abs(const float_comlex& x)	絶対値を求めます。
	float arg(const float_complex& x)	位相角度を求めます。
	float norm(const float_complex& x)	２乗の絶対値を求めます。
	float_complex conj(const float_complex& x)	共役複素数を求めます。
	float_complex polar(const float& rho, const float& theta)	大きさが rho で位相角度が theta の複素数に対応する float_complex 値を求めます。
	float_complex cos(const float_complex& x)	複素余弦を求めます。
	float_complex cosh(const float_complex& x)	複素双曲余弦を求めます。
	float_complex exp(const float_complex& x)	指数関数を求めます。
	float_complex log(const float_complex& x)	自然対数を求めます。
	float_complex log10(const float_complex& x)	常用対数を求めます。
	float_complex pow(const float_complex& x, int y)	x の y 乗を求めます。
	float_complex pow(const float_complex& x, const float& y)	
	float_complex pow(const float_complex& x, const float_complex& y)	
	float_complex pow(const float& x, const float_complex& y)	
	float_complex sin(const float_complex& x)	複素正弦を求めます。
	float_complex sinh(const float_complex& x)	複素双曲正弦を求めます。
	float_complex sqrt(const float_complex& x)	右半空間における範囲での平方根を求めます。
	float_complex tan(const float_complex& x)	複素正接を求めます。
	float_complex tanh(const float_complex& x)	複素双曲正接を求めます。

float_complex operator+ (const float_complex& lhs)
　lhs の単項＋演算を行います。
　リターン値は lhs です。

float_complex operator+(const float_complex& lhs, const float_complex& rhs)
float_complex operator+(const float_complex& lhs, const float& rhs)
float_complex operator+(const float& lhs, const float_complex& rhs)
　lhs と rhs を加算し、結果を lhs に格納します。
　リターン値は、float_complex(lhs)+=rhs です。

float_complex operator-(const float_complex& lhs)
　lhs の単項－演算を行います。
　リターン値は、float_complex(-lhs.real(),-lhs.imag())です。

float_complex operator-(const float_complex& lhs, const float_complex& rhs)
float_complex operator-(const float_complex& lhs, const float& rhs)
float_complex operator-(const float& lhs, const float_complex& rhs)
　lhs から rhs を減算し、結果を lhs に格納します。
　リターン値は、float_complex(lhs)-=rhs です。

float_complex operator*(const float_complex& lhs, const float_complex& rhs)
float_complex operator*(const float_complex& lhs, const float& rhs)
float_complex operator*(const float& lhs, const float_complex& rhs)
　lhs と rhs を乗算し、結果を lhs に格納します。
　リターン値は、float_complex(lhs)*=rhs です。

float_complex operator/(const float_complex& lhs, const float_complex& rhs)
float_complex operator/(const float_complex& lhs, const float& rhs)
float_complex operator/(const float& lhs, const float_complex& rhs)
　lhs を rhs で除算し、結果を lhs に格納します。
　リターン値は、float_complex(lhs)/=rhs です。

bool operator==(const float_complex& lhs, const float_complex& rhs)
bool operator==(const float_complex& lhs, const float& rhs)
bool operator==(const float& lhs, const float_complex& rhs)
　lhs と rhs の実数部どうし、虚数部どうしを比較します。float 型引数の場合、虚数部は float 型の 0.0f と仮定されます。
　リターン値は、lhs.real()= =rhs.real() && lhs.imag()= =rhs.imag()です。

bool operator!=(const float_complex& lhs, const float_complex& rhs)
bool operator!=(const float_complex& lhs, const float& rhs)
bool operator!=(const float& lhs, const float_complex& rhs)
　lhs と rhs の実数部どうし、虚数部どうしを比較します。float 型引数の場合、虚数部は float 型の 0.0f と仮定されます。
　リターン値は、lhs.real()!=rhs.real() || lhs.imag()!=rhs.imag()です。

istream& operator>>(istream& is, float_complex& x)
u,(u), または(u,v)(u は実数部、v は虚数部)の形式の x を入力します。入力値は float_complex に変換されます。
u,(u),(u,v)形式以外が入力された場合は、is.setstate(ios_base::failbit) を呼びます。
リターン値は is です。

ostream& operator<<(ostream& os, const float_complex& x)
x を os に出力します。
出力形式は u,(u)または(u,v)(u は実数部、v は虚数部)です。
リターン値は os です。

float real(const float_complex& x)
実数部を求めます。
リターン値は x.real()です。

float imag(const float_complex& x)
虚数部を求めます。
リターン値は x.imag()です。

float abs(const float_complex& x)
絶対値を求めます。
リターン値は、|x.real()| + |x.imag()|です。

float arg(const float_complex& x)
位相角度を求めます。
リターン値は、atan2f(x.imag() , x.real())です。

float norm(const float_complex& x)
2 乗の絶対値を求めます。
リターン値は、x.real()^2+ x.imag()^2 です。

float_complex conj(const float_complex& x)
共役複素数を求めます。
リターン値は、float_complex(x.real(), (-1)*x.imag())です。

float_complex polar(const float& rho, const float& theta)
大きさが rho で位相角度(偏角)が theta の複素数に対応する float_complex 値を求めます。
リターン値は、float_complex(rho*cosf(theta), rho*sinf(theta))です。

float_complex cos(const float_complex& x)
複素余弦を求めます。
リターン値は、float_complex(cosf(x.real())*coshf(x.imag()), (-1)*sinf(x.real())*sinhf(x.imag()))です。

float_complex cosh(const float_complex& x)
複素双曲余弦を求めます。
リターン値は、cos(float_complex((-1)*x.imag(), x.real()))です。

float_complex exp(const float_complex& x)
　指数関数を求めます。
　リターン値は、expf(x.real())*cosf(x.imag()),expf(x.real())*sinf(x.imag())です。

float_complex log(const float_complex& x)
　(e を底とする)自然対数を求めます。
　リターン値は、float_complex(logf(abs(x)), arg(x))です。

float_complex log10(const float_complex& x)
　(10 を底とする)常用対数を求めます。
　リターン値は、float_complex(log10f(abs(x)), arg(x)/logf(10))です。

float_complex pow(const float_complex& x, int y)
float_complex pow(const float_complex& x, const float& y)
float_complex pow(const float_complex& x, const float_complex& y)
float_complex pow(const float& x, const float_complex& y)
　x の y 乗を求めます。
　pow(0,0)のとき、定義域エラーになります。
　リターン値は次のとおりです。
　　float_complex pow (const float_complex& x,const float_complex& y)の場合：exp(y* logf(x))
　　上記以外：exp(y*log(x))

float_complex sin(const float_complex& x)
　複素正弦を求めます。
　リターン値は、float_complex(sinf(x.real())*coshf(x.imag()), cosf(x.real())*sinhf(x.imag()))です。

float_complex sinh(const float_complex& x)
　複素双曲正弦を求めます。
　リターン値は、float_complex(0,-1)*sin(float_complex((-1)*x.imag(),x.real())) です。

float_complex sqrt(const float_complex& x)
　右半空間における範囲での平方根を求めます。
　リターン値は、float_complex(sqrtf(abs(x))*cosf(arg(x)/2), sqrtf(abs(x))*sinf(arg(x)/2))です。

float_complex tan(const float_complex& x)
　複素正接を求めます。
　リターン値は、sin(x) / cos(x) です。

float_complex tanh(const float_complex& x)
　複素双曲正接を求めます。
　リターン値は、sinh(x) / cosh(x)です。

(c) double_complex クラス

種別	定義名	説明
型	value_type	double 型です。
変数	_re	double 精度の実数部を定義します。
	_im	double 精度の虚数部を定義します。
関数	double_complex(double re = 0.0, double im = 0.0)	コンストラクタです。
	double_complex(const float_complex&)	
	double real() const	実数部を求めます。
	double imag() const	虚数部を求めます。
	double_complex& operator=(double rhs)	rhs を実数部にコピーします。虚数部は 0.0 を設定します。
	double_complex& operator+=(double rhs)	rhs を実数部に加算し、和を*this に格納します。
	double_complex& operator-=(double rhs)	rhs を実数部から減算し、差を*this に格納します。
	double_complex& operator*=(double rhs)	rhs を乗算し、積を*this に格納します。
	double_complex& operator/=(double rhs)	rhs で除算し、商を*this に格納します。
	double_complex& operator=(const double_complex& rhs)	rhs をコピーします。
	double_complex& operator+=(const double_complex& rhs)	rhs を加算し、和を*this に格納します。
	double_complex& operator+=(const double_complex& rhs)	rhs を加算し、和を*this に格納します。
	double_complex& operator*=(const double_complex& rhs)	rhs を乗算し、積を*this に格納します。
	double_complex& operator/=(const double_complex& rhs)	rhs で除算し、商を*this に格納します。

double_complex::double_complex(double re = 0.0, double im = 0.0)
クラス double_complex のコンストラクタです。
以下の値で初期化します。
_re = re;
_im = im;

double_complex::double_complex(const float_complex&)
クラス double_complex のコンストラクタです。
以下の値で初期化します。
_re = (double)rhs.real();
_im = (double)rhs.imag();

double double_complex::real () const
実数部を求めます。
リターン値は、this->_re です。

double double_complex::imag () const
虚数部を求めます。
リターン値は、this->_im です。

double_complex& double_complex::operator=(double rhs)
rhs を実数部(_re)にコピーします。虚数部(_im)は 0.0 を設定します。
リターン値は*this です。

double_complex& double_complex::operator+=(double rhs)
rhs を実数部(_re)に加算し、結果を実数部(_re)に格納します。虚数部(_im)の値は変わりません。
リターン値は*this です。

double_complex& double_complex::operator-=(double rhs)
rhs を実数部(_re)から減算し、結果を実数部(_re)に格納します。虚数部(_im)の値は変わりません。
リターン値は*this です。

double_complex& double_complex::operator*=(double rhs)
rhs と乗算し、結果を*this に格納します。
(_re=_re*rhs, _im=_im*rhs)
リターン値は*this です。

double_complex& double_complex::operator/=(double rhs)
rhs で除算し、結果を*this に格納します。
(_re=_re/rhs, _im=_im/rhs)
リターン値は*this です。

double_complex& double_complex::operator=(const double_complex& rhs)
rhs をコピーします。
リターン値は*this です。

double_complex& double_complex::operator+=(const double_complex& rhs)
rhs を加算し、結果を*this に格納します。
リターン値は*this です。

double_complex& double_complex::operator-=(const double_complex& rhs)
rhs を減算し、結果を*this に格納します。
リターン値は*this です。

double_complex& double_complex::operator*=(const double_complex& rhs)
rhs と乗算し、結果を*this に格納します。
リターン値は*this です。

double_complex& double_complex::operator/=(const double_complex& rhs)
rhs で除算し、結果を*this に格納します。
リターン値は*this です。

(d) double_complex メンバ外関数

種別	定義名	説明
関数	double_complex operator+(const double_complex& lhs)	lhs の単項＋演算を行います。
	double_complex operator+(const double_complex& lhs, const double_complex& rhs)	lhs と rhs を加算し、和を lhs に格納します。
	double_complex operator+(const double_complex& lhs, const double& rhs)	
	double_complex operator+(const double& lhs, const double_complex& rhs)	
	double_complex operator-(const double_complex& lhs)	lhs の単項－演算を行います。
	double_complex operator-(const double_complex& lhs, const double_complex& rhs)	lhs から rhs を減算し、差を lhs に格納します。
	double_complex operator-(const double_complex& lhs, const double& rhs)	
	double_complex operator-(const double& lhs, const double_complex& rhs)	
	double_complex operator*(const double_complex& lhs, const double_complex& rhs)	lhs と rhs を乗算し、積を lhs に格納します。
	double_complex operator*(const double_complex& lhs, const double& rhs)	
	double_complex operator*(const double& lhs, const double_complex& rhs)	
	double_complex operator/(const double_complex& lhs, const double_complex& rhs)	lhs を rhs で除算し、商を lhs に格納します。
	double_complex operator/(const double_complex& lhs, const double& rhs)	
	double_complex operator/(const double& lhs, const double_complex& rhs)	
	bool operator==(const double_complex& lhs, const double_complex& rhs)	lhs と rhs の実数部どうし、虚数部どうしを比較します。
	bool operator==(const double_complex& lhs, const double& rhs)	
	bool operator==(const double& lhs, const double_complex& rhs)	

種別	定義名	説明
関数	bool operator!=(const double_complex& lhs, const double_complex& rhs)	lhs と rhs の実数部どうし、虚数部どうしを比較します。
	bool operator!=(const double_complex& lhs, const double& rhs)	
	bool operator!=(const double& lhs, const double_complex& rhs)	
	istream& operator>>(istream& is, double_complex& x)	u,(u)または(u,v)（u:実数部、v:虚数部）形式の x を入力します。
	ostream& operator<<(ostream& os, double_complex& x)	x を u,(u)または (u,v)（u:実数部、v:虚数部）形式で出力します。
	double real(const double_complex& x)	実数部を求めます。
	double imag(const double_complex& x)	虚数部を求めます。
	double abs(const double_comlex& x)	絶対値を求めます。
	double arg(const double_complex& x)	位相角度を求めます。
	double norm(const double_complex& x)	２乗の絶対値を求めます。
	double_complex conj(const double_complex& x)	共役複素数を求めます。
	double_complex polar(const double& rho, const double& theta)	大きさが rho で位相角度が theta の複素数に対応する double_complex 値を求めます。
	double_complex cos(const double_complex& x)	複素余弦を求めます。
	double_complex cosh(const double_complex& x)	複素双曲余弦を求めます。
	double_complex exp(const double_complex&)	指数関数を求めます。
	double_complex log(const double_complex& x)	自然対数を求めます。
	double_complex log10(const double_complex& x)	常用対数を求めます。
	double_complex pow(const double_complex& x, int y)	x の y 乗を求めます。
	double_complex sin(const double_complex& x)	複素正弦を求めます。
	double_complex sinh(const double_complex& x)	複素双曲正弦を求めます。
	double_complex sqrt(const double_complex& x)	右半空間における範囲での平方根を求めます。
	double_complex tan(const double_complex& x)	複素正接を求めます。
	double_complex tanh(const double_complex& x)	複素双曲正接を求めます。

double_complex operator+(const double_complex& lhs)
　lhs の単項＋演算を行います。
　リターン値は lhs です。

double_complex operator+(const double_complex& lhs, const double_complex& rhs)
double_complex operator+(const double_complex& lhs, const double& rhs)
double_complex operator+(const double& lhs, const double_complex& rhs)
　lhs と rhs を加算し、結果を lhs に格納します。
　リターン値は、double_complex(lhs)+=rhs です。

double_complex operator-(const double_complex& lhs)
　lhs の単項－演算を行います。
　リターン値は、double_complex(-lhs.real(), -lhs.imag()) です。

double_complex operator-(const double_complex& lhs, const double_complex& rhs)
double_complex operator-(const double_complex& lhs, const double& rhs)
double_complex operator-(const double& lhs, const double_complex& rhs)
　lhs から rhs を減算し、結果を lhs に格納します。
　リターン値は、double_complex(lhs)-=rhs です。

double_complex operator*(const double_complex& lhs, const double_complex& rhs)
double_complex operator*(const double_complex& lhs, const double& rhs)
double_complex operator*(const double& lhs, const double_complex& rhs)
　lhs と rhs を乗算し、結果を lhs に格納します。
　リターン値は、double_complex(lhs)*=rhs です。

double_complex operator/(const double_complex& lhs, const double_complex& rhs)
double_complex operator/(const double_complex& lhs, const double& rhs)
double_complex operator/(const double& lhs, const double_complex& rhs)
　lhs を rhs で除算し、結果を lhs に格納します。
　リターン値は、double_complex(lhs)/=rhs です。

bool operator==(const double_complex& lhs, const double_complex& rhs)
bool operator==(const double_complex& lhs, const double& rhs)
bool operator==(const double& lhs, const double_complex& rhs)
　lhs と rhs の実数部どうし、虚数部どうしを比較します。double 型引数の場合、虚数部は double 型の 0.0 と仮定されます。
　リターン値は、lhs.real()= =rhs.real() && lhs.imag()= =rhs.imag() です。

bool operator!=(const double_complex& lhs, const double_complex& rhs)
bool operator!=(const double_complex& lhs, const double& rhs)
bool operator!=(const double& lhs, const double_complex& rhs)
　lhs と rhs の実数部どうし、虚数部どうしを比較します。double 型引数の場合、虚数部は double 型の 0.0 と仮定されます。
　リターン値は、lhs.real()!=rhs.real() || lhs.imag()!=rhs.imag() です。

istream& operator>>(istream& is, double_complex& x)
u,(u)または(u,v)(u は実数部、v は虚数部)の形式の複素数 x を入力します。入力値は double_complex に変換されます。
u,(u),(u,v)形式以外が入力された場合は、is.setstate(ios_base::failbit) を呼びます。
リターン値は is です。

ostream& operator<<(ostream& os, const double_complex& x)
x を os に出力します。
出力形式は u,(u)または(u,v)(u は実数部、v は虚数部)です。
リターン値は os です。

double real(const double_complex& x)
実数部を求めます。
リターン値は x.real() です。

double imag(const double_complex& x)
虚数部を求めます。
リターン値は x.imag() です。

double abs(const double_complex& x)
絶対値を求めます。
リターン値は、|x.real()| + |x.imag()|です。

double arg(const double_complex& x)
位相角度を求めます。
リターン値は、atan2(x.imag() , x.real()) です。

double norm(const double_complex& x)
2 乗の絶対値を求めます。
リターン値は、x.real()^2+ x.imag()^2 です。

double_complex conj(const double_complex& x)
共役複素数を求めます。
リターン値は、double_complex(x.real(), (-1)*x.imag()) です。

double_complex polar(const double& rho, const double& theta)
大きさが rho で位相角度(偏角)が theta の複素数に対応する double_complex 値を求めます。
リターン値は、double_complex(rho*cos(theta), rho*sin(theta)) です。

double_complex cos(const double_complex& x)
複素余弦を求めます。
リターン値は、double_complex(cos(x.real())*cosh(x.imag()), (-1)*sin(x.real())*sinh(x.imag())) です。

double_complex cosh(const double_complex& x)
複素双曲余弦を求めます。
リターン値は、cos(double_complex((-1)*x.imag(), x.real())) です。

double_complex exp(const double_complex& x)
指数関数を求めます。
リターン値は、exp(x.real())*cos(x.imag()),exp(x.real())*sin(x.imag()) です。

double_complex log(const double_complex& x)
(e を底とする)自然対数を求めます。
リターン値は、double_complex(log(abs(x)), arg(x))) です。

double_complex log10 (const double_complex& x)
(10 を底とする)常用対数を求めます。
リターン値は、double_complex(log10(abs(x)), arg(x)/log(10))) です。

double_complex pow(const double_complex& x, int y)
double_complex pow(const double_complex& x, const double& y)
double_complex pow(const double_complex& x, const double_complex& y)
double_complex pow(const double& x, const double_complex& y)
x の y 乗を求めます。
pow(0,0)のとき、定義域エラーになります。
リターン値は、exp(y*log(x)) です。

double_complex sin(const double_complex& x)
複素正弦を求めます。
リターン値は、double_complex(sin(x.real())*cosh(x.imag()), cos(x.real())*sinh(x.imag())) です。

double_complex sinh(const double_complex& x)
複素双曲正弦を求めます。
リターン値は、double_complex(0,-1)*sin(double_complex((-1)*x.imag(),x.real())) です。

double_complex sqrt(const double_complex& x)
右半空間における範囲での平方根を求めます。
リターン値は、double_complex(sqrt(abs(x))*cos(arg(x)/2), sqrt(abs(x))*sin(arg(x)/2))です。

double_complex tan(const double_complex& x)
複素正接を求めます。
リターン値は、sin(x) / cos(x) です。

double_complex tanh(const double_complex& x)
複素双曲正接を求めます。
リターン値は、sinh(x) / cosh(x) です。

(5) 文字列操作用クラスライブラリ

文字列操作用クラスライブラリに対応するヘッダファイルは以下の通りです。

- <string>

string クラスを定義します。

本クラスには派生関係はありません。

(a) string クラス

種別	定義名	説明
型	iterator	char*型です。
	const_iterator	const char*型です。
定数	npos	文字列の最大長(UINT_MAX 文字)です。
変数	s_ptr	オブジェクトが文字列を格納している領域へのポインタです。
	s_len	オブジェクトが格納している文字列長です。
	s_res	オブジェクトが文字列を格納するために確保している領域のサイズです。
関数	string(void)	コンストラクタです。
	string(const string& str, size_t pos = 0, size_t n = npos)	
	string(const char* str, size_t n)	
	string(const char* str)	
	string(size_t n, char c)	
	~string()	デストラクタです。
	string& operator=(const string& str)	str を代入します。
	string& operator=(const char* str)	str を代入します。
	string& operator=(char c)	c を代入します。
	iterator begin()	文字列の先頭ポインタを求めます。
	const_iterator begin() const	
	iterator end()	文字列の最後尾ポインタを求めます。
	const_iterator end() const	
	size_t size() const	格納されている文字列の文字列長を求めます。
	size_t length() const	
	size_t max_size() const	確保している領域のサイズを求めます。
	void resize(size_t n, char c)	格納可能な文字列の長さを n に変更します。
	void resize(size_t n)	格納可能な文字列の長さを n に変更します。
	size_t capacity() const	確保している領域のサイズを求めます。
	void reserve(size_t res_arg = 0)	領域の再割り当てを行います。
	void clear()	格納されている文字列を clear します。
	bool empty() const	格納している文字列の長さが 0 かチェックします。
	const char& operator[](size_t pos) const	s_ptr[pos]を参照します。
	char& operator[](size_t pos)	
	const char& at(size_t pos) const	
	char& at(size_t pos)	

種別	定義名	説明
関数	string& operator+=(const string& str)	str の文字列を追加します。
	string& operator+=(const char* str)	str の文字列を追加します。
	string& operator+=(char c)	c の文字を追加します。
	string& append(const string& str)	str の文字列を追加します。
	string& append(const char* str)	
	string& append(const string& str, size_t pos, size_t n)	オブジェクトの位置 pos に str の文字列を n 文字分追加します。
	string& append(const char* str, size_t n)	文字列 str の n 文字分を追加します。
	string& append(size_t n, char c)	n 個の文字 c を追加します。
	string& assign(const string& str)	str の文字列を代入します。
	string& assign(const char* str)	
	string& assign(const string& str, size_t pos, size_t n)	位置 pos に文字列 str の n 文字分を代入します。
	string& assign(const char* str, size_t n)	文字列 str の n 文字分を代入します。
	string& assign(size_t n, char c)	n 個の文字 c を代入します。
	string& insert(size_t pos1, const string& str)	位置 pos1 に str の文字列を挿入します。
	string& insert(size_t pos1, const string& str, size_t pos2, size_t n)	位置 pos1 に str の文字列の位置 pos2 から n 文字分を挿入します。
	string& insert(size_t pos, const char* str, size_t n)	pos の位置に文字列 str を n 文字分挿入します。
	string& insert(size_t pos, const char* str)	pos の位置に文字列 str を挿入します。
	string& insert(size_t pos, size_t n, char c)	位置 pos に n 個の文字 c の文字列を挿入します。
	iterator insert(iterator p, char c = char())	p が指す文字列の前に文字 c を挿入します。
	void insert(iterator p, size_t n, char c)	p が指す文字の前に、n 個の文字 c を挿入します。
	string& erase(size_t pos = 0, size_t n = npos)	位置 pos から n 個分取り除きます。
	iterator erase(iterator position)	position により参照された文字を取り除きます。
	iterator erase(iterator first, iterator last)	範囲[first, last]において文字を取り除きます。
	string& replace(size_t pos1, size_t n1, const string& str)	位置 pos1 から n1 文字分の文字列を、str の文字列で置き換えます。
	string& replace(size_t pos1, size_t n1, const char* str)	

種別	定義名	説明
関数	string& replace(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2)	位置 pos1 から n1 文字分の文字列を、str の位置 pos2 から n2 文字分の文字列で置き換えます。
	string& replace(size_t pos, size_t n1, const char* str, size_t n2)	位置 pos から n1 文字分の文字列を、n2 個の str の文字列で置き換えます。
	string& replace(size_t pos, size_t n1, size_t n2, char c)	位置 pos から n1 文字分の文字列を、n2 個の文字 c で置き換えます。
	string& replace(iterator i1, iterator i2, const string& str)	位置 i1 から i2 までの文字列を str の文字列で置き換えます。
	string& replace(iterator i1, iterator i2, const char* str)	
	string& replace(iterator i1, iterator i2, const char* str, size_t n)	位置 i1 から i2 までの文字列を str の文字列の n 文字分で置き換えます。
	string& replace(iterator i1, iterator i2, size_t n, char c)	位置 i1 から i2 までの文字列を n 個の文字 c で置き換えます。
	size_t copy(char* str, size_t n, size_t pos = 0) const	位置 pos に文字列 str の n 文字分の文字列をコピーします。
	void swap(string& str)	str の文字列と交換します。
	const char* c_str() const	文字列を格納している領域へのポインタを参照します。
	const char* date() const	
	size_t find(const string& str, size_t pos = 0) const	位置 pos 以降で str の文字列と同じ文字列が最初に現れる位置を検索します。
	size_t find(const char* str, size_t pos = 0) const	
	size_t find(const char* str, size_t pos, size_t n) const	位置 pos 以降で str の n 文字分と同じ文字列が最初に現れる位置を検索します。

種別	定義名	説明
関数	size_t find(char c, size_t pos = 0) const	位置 pos 以降で文字 c が最初に現れる位置を検索します。
	size_t rfind(const string& str, size_t pos = npos) const	位置 pos 以前で str の文字列と同じ文字列が最後に現れる位置を検索します。
	size_t rfind(const char* str, size_t pos = npos) const	
	size_t rfind(const char* str, size_t pos, size_t n) const	位置 pos 以前で str の n 文字分と同じ文字列が最後に現れる位置検索します。
	size_t rfind(char c, size_t pos = npos) const	位置 pos 以前で文字 c が最後に現れる位置を検索します。
	size_t find_first_of(const string& str, size_t pos = 0) const	位置 pos 以降で文字列 str に含まれる任意の文字が最初に現れる位置を検索します。
	size_t find_first_of(const char* str, size_t pos = 0) const	
	size_t find_first_of(const char* str, size_t pos, size_t n) const	位置 pos 以降で文字列 str の n 文字分に含まれる任意の文字が最初に現れる位置を検索します。
	size_t find_first_of(char c, size_t pos = 0) const	位置 pos 以降で文字 c が最初に現れる位置を検索します。
	size_t find_last_of(const string& str, size_t pos = npos) const	位置 pos 以前で文字列 str に含まれる任意の文字が最後に現れる位置を検索します。
	size_t find_last_of(const char* str, size_t pos = npos) const	
	size_t find_last_of(const char* str, size_t pos, size_t n) const	位置 pos 以前で文字列 str の n 文字分に含まれる任意の文字が最後に現れる位置を検索します。
	size_t find_last_of(char c, size_t pos = npos) const	位置 pos 以前で文字 c が最後に現れる位置を検索します。
	size_t find_first_not_of(const string& str, size_t pos = 0) const	位置 pos 以降で str 中の任意の文字と異なった文字が最初に現れる位置を検索します。
	size_t find_first_not_of(const char* str, size_t pos = 0) const	
	size_t find_first_not_of(const char* str, size_t pos, size_t n)	位置 pos 以降で str の先頭から n 文字までの任意の文字と異なった文字が最初に現れる位置を検索します。
	size_t find_first_not_of(char c, size_t pos = 0) const	位置 pos 以降で文字 c と異なった文字が最初に現れる位置を検索します。

種別	定義名	説明
関数	size_t find_last_not_of(const string& str, size_t pos = npos) const	位置 pos 以前で str 中の任意の文字と異なった文字が最後に現れる位置を検索します。
	size_t find_last_not_of(const char* str, size_t pos = npos) const	
	size_t find_last_not_of(const char* str, size_t pos, size_t n) const	位置 pos 以前で str の先頭から n 文字までの任意の文字と異なった文字が最後に現れる位置を検索します。
	size_t find_last_not_of(char c, size_t pos = npos) const	位置 pos 以前で文字 c と異なった文字が最後に現れる位置を検索します。
	string substr(size_t pos = 0, size_t n = npos) const	格納された文字列に対し、範囲[pos,n]の文字列を持つオブジェクトを生成します。
	int compare(const string& str) const	文字列と str の文字列を比較します。
	int compare(size_t pos1, size_t n1, const string& str) const	位置 pos1 から n1 文字分の文字列と str を比較します。
	int compare(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2) const	位置 pos1 から n1 文字分の文字列と str の位置 pos2 から n2 文字分の文字列を比較します。
	int compare(const char* str) const	str と比較します。
	int compare(size_t pos1, size_t n1, const char* str, size_t n2 = npos) const	位置 pos1 から n1 文字分の文字列と str の n2 文字分の文字列を比較します。

string::string(void)
　以下のように設定します。
　s_ptr = 0;
　s_len = 0;
　s_res = 1;

string::string(const string& str, size_t pos = 0, size_t n = npos)
　str をコピーします。ただし、s_len は、n と s_len の小さい方の値になります。

string::string(const char* str, size_t n)
　以下に設定します。
　s_ptr = str;
　s_len = n;
　s_res = n+1;

string::string(const char* str)
　以下に設定します。
　s_ptr = str;
　s_len = str の文字列長;
　s_res = str の文字列長+1;

string::string(size_t n, char c)
　以下に設定します。
　s_ptr =文字数 n で文字 c の文字列
　s_len = n
　s_res = n+1;

string::~string()
　クラス string のデストラクタです。
　文字列を格納している領域を解放します。

string& string::operator=(const string& str)
　str のデータを代入します。
　リターン値は*this です。

string& string::operator=(const char* str)
　str から string オブジェクトを生成し、そのデータを代入します。
　リターン値は*this です。

string& string::operator=(char c)
　c から string オブジェクトを生成し、そのデータを代入します。
　リターン値は*this です。

string::iterator string::begin()
string::const_iterator string::begin() const
　文字列の先頭ポインタを求めます。
　リターン値は、文字列の先頭ポインタです。

string::iterator string::end()
string::const_iterator string::end() const
　文字列の最後尾ポインタを求めます。
　リターン値は、文字列の最後尾ポインタです。

size_t string::size() const
size_t string::length() const
　格納されている文字列の文字列長を求めます。
　リターン値は、格納されている文字列の文字列長です。

size_t string::max_size() const
　確保している領域のサイズを求めます。
　リターン値は、確保している領域のサイズです。

void string::resize(size_t n, char c)
　オブジェクトが格納可能な文字列の長さを n に変更します
　n<=size()のとき、長さを n にしたもとの文字列と置き換えます。
　n>size()のとき、元の文字列の後ろに長さ n になるまで c をつめた文字列と置き換えます。
　n<=max_size()である必要があります
　n>max_size()の場合、n=max_size()として計算します。

void string::resize(size_t n)
　オブジェクトが格納可能な文字列の長さを n に変更します
　n<=size()のとき、長さを n にしたもとの文字列と置き換えます。
　n<=max_size()である必要があります。

size_t string::capacity() const
　確保している領域のサイズを求めます。
　リターン値は、確保している領域のサイズです。

void string::reserve(size_t res_arg = 0)
　記憶領域の再割り当てを行います。
　reserve()後、capacity()は reserve()の引数より大きいかまたは等しくなります
　再割り当てを行うと、すべての参照・ポインタ・この数列の中の要素の参照する iterator を無効にします。

void string::clear()
　格納されている文字列を clear します。

bool string::empty() const
　格納している文字列の長さが 0 かチェックします。
　リターン値は次のとおりです。
　　格納している文字列長が 0 の場合　：true
　　格納している文字列長が 0 以外の場合：false

const char& string::operator[] (size_t pos) const
char& string::operator[](size_t pos)
const char& string::at(size_t pos) const
char& string::at(size_t pos)
　s_ptr[pos]を参照します。
　リターン値は次のとおりです。
　　n< s_len の場合 ：s_ptr [pos]
　　n>= s_len の場合：'\0'

string& string::operator+=(const string& str)
　str が格納している文字列を追加します。
　リターン値は*this です。

string& string::operator+=(const char* str)
　str から string オブジェクトを生成し、その文字列を追加します。
　リターン値は*this です。

string& string::operator+=(char c)
　c から string オブジェクトを生成し、その文字列を追加します。
　リターン値は*this です。

string& string::append(const string& str)
string& string::append(const char* str)
　str の文字列をオブジェクトに追加します。
　リターン値は*this です。

string& string::append(const string& str, size_t pos, size_t n)
　オブジェクトの位置 pos に str の文字列を n 文字分追加します。
　リターン値は*this です。

string& string::append(const char* str, size_t n)
　文字列 str の n 文字分を追加します。
　リターン値は*this です。

string& string::append(size_t n, char c)
　n 個の文字 c を追加します。
　リターン値は*this です。

string& string::assign(const string& str)
string& string::assign(const char* str)
　str の文字列を代入します。
　リターン値は*this です。

string& string::assign(const string& str, size_t pos, size_t n)
　位置 pos に文字列 str の n 文字分を代入します。
　リターン値は*this です。

string& string::assign(const char* str, size_t n)
　文字列 str の n 文字分を代入します。
　リターン値は*this です。

string& string::assign(size_t n, char c)
　n 個の文字 c を代入します。
　リターン値は*this です。

string& string::insert(size_t pos1, const string& str)
　位置 pos1 に str の文字列を挿入します。
　リターン値は*this です。

string& string::insert(size_t pos1, const string& str, size_t pos2, size_t n)
　位置 pos1 に str の文字列の位置 pos2 から n 文字分を挿入します。
　リターン値は*this です。

string& string::insert(size_t pos, const char* str, size_t n)
　pos の位置に文字列 str を n 文字分挿入します。
　リターン値は*this です。

string& string::insert(size_t pos, const char* str)
　pos の位置に文字列 str を挿入します。
　リターン値は*this です。

string& string::insert(size_t pos, size_t n, char c)
　位置 pos に n 個の文字 c の文字列を挿入します。
　リターン値は*this です。

string::iterator string::insert(iterator p, char c = char())
　p が指す文字列の前に、文字 c を挿入します。
　リターン値は、挿入された文字です。

void string::insert(iterator p, size_t n, char c)
　p が指す文字の前に、n 個の文字 c を挿入します。

string& string::erase(size_t pos = 0, size_t n = npos)
　位置 pos から n 個分取り除きます。
　リターン値は*this です。

iterator string::erase(iterator position)
　position により参照された文字を取り除きます。
　リターン値は次のとおりです。
　　削除要素の次の iterator がある場合：削除要素の次の iterator
　　削除要素の次の iterator がない場合：end()

iterator string::erase(iterator first, iterator last)
　範囲[first, last]において文字を取り除きます。
　リターン値は次のとおりです。
　　last の次の iterator がある場合：last の次の iterator
　　last の次の iterator がない場合：end()

string& string::replace(size_t pos1, size_t n1, const string& str)
string& string::replace(size_t pos1, size_t n1, const char* str)
　位置 pos1 から n1 文字分の文字列を、str の文字列で置き換えます。
　リターン値は*this です。

string& string::replace(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2)
　位置 pos1 から n1 文字分の文字列を、str の位置 pos2 から n2 文字分の文字列で置き換えます。
　リターン値は*this です。

string& string::replace(size_t pos, size_t n1, const char* str, size_t n2)
　位置 pos から n1 文字分の文字列を、str の n2 文字分の文字列で置き換えます。
　リターン値は*this です。

string& string::replace(size_t pos, size_t n1, size_t n2, char c)
　位置 pos から n1 文字分の文字列を、n2 個の文字 c で置き換えます。
　リターン値は*this です。

string& string::replace(iterator i1, iterator i2, const string& str)
string& string::replace(iterator i1, iterator i2, const char* str)
　位置 i1 から i2 までの文字列を str の文字列で置き換えます。
　リターン値は*this です。

string& string::replace(iterator i1, iterator i2, const char* str, size_t n)
　位置 i1 から i2 までの文字列を、str の n 文字分の文字列で置き換えます。
　リターン値は*this です。

string& string::replace(iterator i1, iterator i2, size_t n, char c)
　位置 i1 から i2 までの文字列を、n 個の文字 c で置き換えます。
　リターン値は*this です。

size_t string::copy(char* str, size_t n, size_t pos = 0) const
　位置 pos に文字列 str の n 文字分の文字列をコピーします。
　リターン値は rlen です。

void string::swap(string& str)
　str の文字列と交換します。

const char* string::c_str() const
const char* string::data() const
　文字列を格納している領域へのポインタを参照します。
　リターン値は s_ptr です。

size_t string::find(const string& str, size_t pos = 0) const
size_t string::find(const char* str, size_t pos = 0) const
位置 pos 以降で str の文字列と同じ文字列が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find(const char* str, size_t pos, size_t n) const
位置 pos 以降で str の n 文字分と同じ文字列が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find(char c, size_t pos = 0) const
位置 pos 以降で文字 c が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::rfind(const string& str, size_t pos = npos) const
size_t string::rfind(const char* str, size_t pos = npos) const
位置 pos 以前で str の文字列と同じ文字列が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::rfind(const char* str, size_t pos, size_t n) const
位置 pos 以前で文字列 str の n 文字分と同じ文字列が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::rfind(char c, size_t pos = npos) const
位置 pos 以前で文字 c が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_first_of(const string& str, size_t pos = 0) const
size_t string::find_first_of(const char* str, size_t pos = 0) const
位置 pos 以降で文字列 str に含まれる任意の文字が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_first_of(const char* str, size_t pos, size_t n) const
位置 pos 以降で文字列 str の n 文字分に含まれる任意の文字が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_first_of(char c, size_t pos = 0) const
位置 pos 以降で文字 c が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_last_of(const string& str, size_t pos = npos) const
size_t string::find_last_of(const char* str, size_t pos = npos) const
位置 pos 以前で文字列 str に含まれる任意の文字が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_last_of(const char* str, size_t pos, size_t n) const
位置 pos 以前で文字列 str の n 文字分に含まれる任意の文字が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_last_of(char c, size_t pos = npos) const
位置 pos 以前で文字 c が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_first_not_of(const string& str, size_t pos = 0) const
size_t string::find_first_not_of(const char* str, size_t pos = 0) const
位置 pos 以降で str 中の任意の文字と異なった文字が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_first_not_of(const char* str, size_t pos, size_t n) const
位置 pos 以降で str の先頭から n 文字までの任意の文字と異なった文字が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_first_not_of(char c, size_t pos = 0) const
位置 pos 以降で文字 c と異なった文字が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_last_not_of (const string& str, size_t pos = npos) const
size_t string::find_last_not_of(const char* str, size_t pos = npos) const
位置 pos 以前で str 中の任意の文字と異なった文字が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_last_not_of(const char* str, size_t pos, size_t n) const
位置 pos 以前で str の先頭から n 文字までの任意の文字と異なった文字が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_last_not_of(char c, size_t pos = npos) const
位置 pos 以前で文字 c と異なった文字が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

string string::substr(size_t pos = 0, size_t n = npos) const
格納された文字列に対し、範囲[pos,n]の文字列を持つオブジェクトを生成します。
リターン値は、範囲[pos,n]の文字列を持つオブジェクトです。

int string::compare(const string& str) const
文字列と str の文字列を比較します。
リターン値は次のとおりです。
文字列が同一の場合：0
文字列が異なる場合：this->s_len>str.s_len のとき 1
this->s_len < str.s_len のとき-1

int string::compare(size_t pos1, size_t n1, const string& str) const
位置 pos1 から n1 文字分の文字列と str を比較します。
リターン値は次のとおりです。
文字列が同一の場合：0
文字列が異なる場合：this->s_len>str.s_len のとき 1
this->s_len < str.s_len のとき-1

int string::compare(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2) const
位置 pos1 から n1 文字分の文字列と str の位置 pos2 から n2 文字分の文字列を比較します。
リターン値は次のとおりです。
文字列が同一の場合：0
文字列が異なる場合：this->s_len>str.s_len のとき 1
this->s_len < str.s_len のとき-1

int string::compare(const char* str) const
str と比較します。
リターン値は次のとおりです。
文字列が同一の場合：0
文字列が異なる場合：this->s_len>str.s_len のとき 1
this->s_len < str.s_len のとき-1

int string::compare(size_t pos1, size_t n1, const char* str, size_t n2 = npos) const
位置 pos1 から n1 文字分の文字列と str の n2 文字分の文字列を比較します。
リターン値は次のとおりです。
文字列が同一の場合：0
文字列が異なる場合：this->s_len>str.s_len のとき 1
this->s_len < str.s_len のとき-1

(b) string クラスマニピュレータ

種別	定義名	説明
関数	string operator+(const string& lhs, const string& rhs)	lhs の文字列（または文字）に rhs の文字列（または文字）を追加し、オブジェクトを生成してその文字列を格納します。
	string operator+(const char* lhs, const string& rhs)	
	string operator+(char lhs, const string& rhs)	
	string operator+(const string& lhs, const char* rhs)	
	string operator+(const string& lhs, char rhs)	
	bool operator==(const string& lhs, const string& rhs)	lhs の文字列と rhs の文字列を比較します。
	bool operator==(const char* lhs, const string& rhs)	
	bool operator==(const string& lhs, const char* rhs)	
	bool operator!=(const string& lhs, const string& rhs)	lhs の文字列と rhs の文字列を比較します。
	bool operator!=(const char* lhs, const string& rhs)	
	bool operator!=(const string& lhs, const char* rhs)	
	bool operator<(const string& lhs, const string& rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator<(const char* lhs, const string& rhs)	
	bool operator<(const string& lhs, const char* rhs)	
	bool operator>(const string& lhs, const string& rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator>(const char* lhs, const string& rhs)	
	bool operator>(const string& lhs, const char* rhs)	
	bool operator<=(const string& lhs, const string& rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator<=(const char* lhs, const string& rhs)	
	bool operator<=(const string& lhs, const char* rhs)	
	bool operator>=(const string& lhs, const string& rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator>=(const char* lhs, const string& rhs)	
	bool operator>=(const string& lhs, const char* rhs)	
	void swap(string& lhs, string& rhs)	lhs の文字列と rhs の文字列を交換します。
	istream& operator>>(istream& is, string& str)	文字列を str に抽出します。
	ostream& operator<<(ostream& os, const string& str)	文字列を挿入します。
	istream& getline(istream& is, string& str, char delim)	is から文字列を抽出し，str に付加します。途中で文字'delim'を検出したら、入力を終了します。
	istream& getline(istream& is, string& str)	is から文字列を抽出し，str に付加します。途中で改行文字を検出したら、入力を終了します。

string operator+(const string& lhs, const string& rhs)
string operator+(const char* lhs, const string& rhs)
string operator+(char lhs, const string& rhs)
string operator+(const string& lhs, const char* rhs)
string operator+(const string& lhs, char rhs)
　lhs の文字列（または文字）に rhs の文字列（または文字）を追加し、オブジェクトを生成してその文字列を格納します。
　リターン値は、結合した文字列を格納するオブジェクトです。

bool operator==(const string& lhs, const string& rhs)
bool operator==(const char* lhs, const string& rhs)
bool operator==(const string& lhs, const char* rhs)
　lhs の文字列と rhs の文字列を比較します。
　リターン値は次のとおりです。
　　文字列が同一の場合：true
　　文字列が異なる場合：false

bool operator!=(const string& lhs, const string& rhs)
bool operator!=(const char* lhs, const string& rhs)
bool operator!=(const string& lhs, const char* rhs)
　lhs の文字列と rhs の文字列を比較します。
　リターン値は次のとおりです。
　　文字列が同一の場合：false
　　文字列が異なる場合：true

bool operator<(const string& lhs, const string& rhs)
bool operator<(const char* lhs, const string& rhs)
bool operator<(const string& lhs, const char* rhs)
　lhs の文字列長と rhs の文字列長を比較します。
　リターン値は次のとおりです。
　　lhs.s_len < rhs.s_len の場合 ：true
　　lhs.s_len >= rhs.s_len の場合：false

bool operator>(const string& lhs, const string& rhs)
bool operator>(const char* lhs, const string& rhs)
bool operator>(const string& lhs, const char* rhs)
　lhs の文字列長と rhs の文字列長を比較します。
　リターン値は次のとおりです。
　　lhs.s_len > rhs.s_len の場合 ：true
　　lhs.s_len <= rhs.s_len の場合：false

bool operator<=(const string& lhs, const string& rhs)
bool operator<=(const char* lhs, const string& rhs)
bool operator<=(const string& lhs, const char* rhs)
　lhs の文字列長と rhs の文字列長を比較します。
　リターン値は次のとおりです。

lhs.s_len <= rhs.s_len の場合：true
lhs.s_len > rhs.s_len の場合 ：false

bool operator>=(const string& lhs, const string& rhs)
bool operator>=(const char* lhs, const string& rhs)
bool operator>=(const string& lhs, const char* rhs)
lhs の文字列長と rhs の文字列長を比較します。
リターン値は次のとおりです。
lhs.s_len >= rhs.s_len の場合：true
lhs.s_len < rhs.s_len の場合 ：false
void swap(string& lhs, string& rhs)
lhs の文字列と rhs の文字列を交換します。

istream& operator>>(istream& is, string& str)
文字列を str に抽出します。
リターン値は is です。

ostream& operator<<(ostream& os, const string& str)
文字列を挿入します。
リターン値は os です。

istream& getline(istream& is, string& str, char delim)
is から文字列を抽出し、str に付加します。
途中で文字‘delim’を検出したら、入力を終了します
リターン値は is です。

istream& getline (istream& is, string& str)
is から文字列を抽出し、str に付加します。
途中で改行文字を検出したら、入力を終了します
リターン値は is です。

10.3.3 リエントラントライブラリ

表 10.42 にリエントラントライブラリ一覧を示します。 表中、△で示した関数は、errno 変数を設定しますので 、プログラム中で errno を参照していなければリエントラントに実行できます。

さらにセマフォを利用してリエントラント性を高めたい場合は標準ライブラリ構築ツールで **reent** オプションを指定してください。ライブラリは、rand、srand 関数を除いてすべてリエントラントに実行できます。但し、strtok 関数の同一文字列に対する連続的な呼び出しは保証しません。また、「9.2.2 (7) (b) **低水準インタフェースルーチンの仕様」**と「9.2.2 (7) (d) **リエントラントライブラリ用低水準インタフェースルーチン例」**を参照してください。

ﾘｴﾝﾄﾗﾝﾄ欄　○:ﾘｴﾝﾄﾗﾝﾄ　×:ﾉﾝﾘｴﾝﾄﾗﾝﾄ　△:errno を設定

表 10.42 リエントラントライブラリー覧

	標準 ｲﾝｸﾙｰﾄﾞﾌｧｲﾙ		関数名	ﾘｴﾝﾄﾗﾝﾄ
1	stddef.h	1	offsetof	○
2	assert.h	2	assert	×
3	ctype.h	3	isalnum	○
		4	isalpha	○
		5	iscntrl	○
		6	isdigit	○
		7	isgraph	○
		8	islower	○
		9	isprint	○
		10	ispunct	○
		11	isspace	○
		12	isupper	○
		13	isxdigit	○
		14	tolower	○
		15	toupper	○
4	math.h	16	acos	△
		17	asin	△
		18	atan	△
		19	atan2	△
		20	cos	△
		21	sin	△
		22	tan	△
		23	cosh	△
		24	sinh	△
		25	tanh	△
		26	exp	△
		27	frexp	△
		28	ldexp	△
		29	log	△
		30	log10	△
		31	modf	△
		32	pow	△
		33	sqrt	△
		34	ceil	△
		35	fabs	△
		36	floor	△
		37	fmod	△
5	mathf.h	38	acosf	△
		39	asinf	△
		40	atanf	△
		41	atan2f	△
		42	cosf	△
		43	sinf	△
		44	tanf	△
		45	coshf	△
		46	sinhf	△
		47	tanhf	△
		48	expf	△
		49	frexpf	△
		50	ldexpf	△
		51	logf	△
		52	log10f	△
		53	modff	△
		54	powf	△
		55	sqrtf	△
		56	ceilf	△
		57	fabsf	△
		58	floorf	△
		59	fmodf	△
6	setjmp.h	60	setjmp	○
		61	longjmp	○
7	stdarg.h	62	va_start	○
		63	va_arg	○
		64	va_end	○
8	stdio.h	65	fclose	×
		66	fflush	×
		67	fopen	×
		68	freopen	×
		69	setbuf	×
		70	setvbuf	×
		71	fprintf	×
		72	fscanf	×
		73	printf	×
		74	scanf	×

	標準 ｲﾝｸﾙｰﾄﾞﾌｧｲﾙ		関数名	ﾘｴﾝﾄﾗﾝﾄ
8	stdio.h	75	sprintf	△
		76	sscanf	△
		77	vfprintf	×
		78	vprintf	×
		79	vsprintf	△
		80	fgetc	×
		81	fgets	×
		82	fputc	×
		83	fputs	×
		84	getc	×
		85	getchar	×
		86	gets	×
		87	putc	×
		88	putchar	×
		89	puts	×
		90	ungetc	×
		91	fread	×
		92	fwrite	×
		93	fseek	×
		94	ftell	×
		95	rewind	×
		96	clearerr	×
		97	feof	×
		98	ferror	×
		99	perror	×
9	stdlib.h	100	atof	△
		101	atoi	△
		102	atol	△
		103	strtod	△
		104	strtol	△
		105	rand	×

	標準 ｲﾝｸﾙｰﾄﾞﾌｧｲﾙ		関数名	ﾘｴﾝﾄﾗﾝﾄ
9	stdlib.h	106	srand	×
		107	calloc	×
		108	free	×
		109	malloc	×
		110	realloc	×
		111	bsearch	○
		112	qsort	○
		113	abs	○
		114	div	△
		115	labs	○
		116	ldiv	△
10	string.	117	memcpy	○
		118	strcpy	○
		119	strncpy	○
		120	strcat	○
		121	strncat	○
		122	memcmp	○
		123	strcmp	○
		124	strncmp	○
		125	memchr	○
		126	strchr	○
		127	strcspn	○
		128	strpbrk	○
		129	strrchr	○
		130	strspn	○
		131	strstr	○
		132	strtok	×
		133	memset	○
		134	strerror	○
		135	strlen	○
		136	memmove	○

10.3.4 未サポートライブラリ

C 言語仕様で定義しているライブラリのうち、本コンパイラでサポートしていないライブラリを**表** 10.43 に示します。

表 10.43 **サポートしていないライブラリ**

	ヘッダファイル	ライブラリ名
1	locale.h*1	setlocale、localeconv
2	signal.h*1	signal、raise
3	stdio.h	remove、rename、tmpfile、tmpnam、fgetpos、fsetpos
4	stdlib.h	strtoul、abort、atexit、exit、getenv、system、 mblen、mbtowc、wctomb、mbstowcs、wcstombs
5	string.h	strcoll、strxfrm
6	time.h*1	clock、difftime、mktime、time、asctime、ctime、gmtime、localtime、 strftime

【注】*1 **ヘッダファイルをサポートしません。**

11. アセンブラ言語仕様

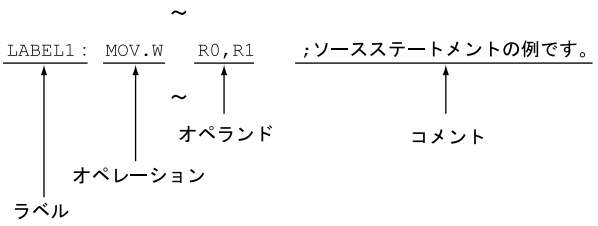
11.1 プログラムの要素

11.1.1 ソースステートメント

(1) ソースステートメントの構成

ソースステートメントの構成を以下に示します。
[ラベル][△オペレーション [△オペランド]][コメント]

例：



(a) ラベル

ソースステートメントにつける名札としてシンボルまたはローカルシンボルを書きます。
シンボルとはプログラマが定義する名前です。

(b) オペレーション

実行命令、アセンブラ制御命令、プリプロセッサ制御文のニーモニック(名称)を書きます。
実行命令はマイコンの命令です。
アセンブラ制御命令は、アセンブラが解釈して実行する命令です。
プリプロセッサ制御文には、ファイルインクルード機能、条件つきアセンブル機能、構造化アセンブリ機能、マクロ機能に関するものがあります。

(c) オペランド

オペレーションの実行**対象となるものを書きます。オペランドの個数と種類はオペレーションによって決まります。オペランド**を必要としないオペレーションもあります。

(d) コメント

プログラムを分かりやすくするための注釈を書きます。

(2) ソースステートメントの書き方

ソースステートメントは ASCII 文字で記述します。ただし、文字列またはコメントの中にはかな・漢字(シフト JIS コード、EUC コード)、欧州文字コードを記述できます。基本的に 1 つのソースステートメントは 1 行に納まるように書いてください。1 行の最大長は 8,192 文字です。

(a) ラベルの書き方

ラベルは次のように記述します。

・ 1 カラム目から記述する。

または

・ ラベル名の直後にコロン(:)をつける。

例：

良い例：

```
LABEL1                                  ; 1 カラム目から書き始めています。
  LABEL2:                               ; コロンで終わっています。
```

悪い例：

```
  LABEL3                                ; 1 カラム目から書き始めずコロンを
                                        ; つけてもいないので、アセンブラは
                                        ; ラベルと見なしません。
```

(b) オペレーションの書き方

オペレーションは次のように記述します。

・ ラベルを記述していない場合：2 カラム目以降から書き始める。

・ ラベルを記述している場合：ラベルとの間に 1 つ以上の空白またはタブを置いて書き始める。

例：

```
            ADD.W   R0,R1               ; ラベルを記述していない場合です。
LABEL1:     ADD.W   R1,R2               ; ラベルを記述してある場合です。
```

(c) オペランドの書き方

オペランドはオペレーションとの間に 1 つ以上の空白またはタブを置いて記述します。

例：

```
            ADD.W   R0,R1               ; ADD 命令のオペランドは 2 つです。
            SHAL.W  R1                  ; SHAL 命令のオペランドは 1 つです。
```

(d) コメントの書き方

セミコロン(;)の後に続けて記述します。アセンブラはセミコロンから行末までをコメントと見なします。

例：

```
            ADD.W R0,R1                 ; R1 に R0 を加えます。
```

(3) 複数行にわたるソースステートメントの書き方

次のようなときにはプログラムを見やすくするため、1 つのソースステートメントを複数の行に分けて記述することができます。

- ソースステートメントが長くなる場合
- オペランドの 1 つ 1 つにコメントをつけたい場合

複数行にわたるソースステートメントは次の(a)～(c)の手順で記述してください。

(a) オペランドとオペランドを区切るカンマ(,)を切れ目として改行します。
(b) すぐ次の行の 1 カラム目にプラス(+)を書きます。
(c) そのプラスの後ろにソースステートメントの続きを書きます。

プラスの後ろに空白またはタブを入れても構いません。各行の最後にコメントを書くこともできます。

例：

```
(a)
            .DATA.W         H'FF00,
 +                          H'FF00,
 +                          H'FFFF
                                        ; 1つのソースステートメントを3行に
                                        ; わたって書いた例です。

(b)
            .DATA.W         H'FF00,     ; 初期値1
 +                          H'FF00,     ; 初期値2
 +                          H'FFFF      ; 初期値3
                                        ; オペランド1つ1つに、コメントを
                                        ; つけた例です。
```

11.1.2 予約語

予約語は特別な意味を持つ語としてアセンブラが用意している名前です。予約語にはレジスタ名、演算子、ロケーションカウンタがあります。レジスタ名は CPU 種別ごとに異なりますので各 CPU の**プログラミングマニュアル**を参照してください。予約語はシンボルとしては使用できません。

・レジスタ名
ER0～ER7、E0～E7、R0～R7、R0H～R7H、R0L～R7L、SP*、CCR、EXR、MACH、MACL、PC、SBR、VBR
・演算子
STARTOF、SIZEOF、HIGH、LOW、HWORD、LWORD、
・ロケーションカウンタ
$

【注】* ER7(H8SX、H8S/2600、H8S/2000、H8/300H)**または** R7(H8/300、H8/300L)**と** SP **は同じレジスタを表します。**

11.1.3 シンボル

(1) シンボルの役割

シンボルはプログラマが定義する名前であり、次の役割を果たします。

- アドレスシンボル：データの格納場所、分岐先などのアドレスを表します。
- 定数シンボル　　：定数を表します。
- ビットデータ名　：ビット操作命令の対象となるメモリ上の 1 ビットデータを示します。
- レジスタ別名　　：汎用レジスタを表します。
- セクション名　　：セクションの名前を表します。

シンボルの使用例を以下に示します。

例：

(a)
```
        BRA     SUB1            ; BRA は分岐命令です。
        ～                      ; SUB1 は分岐先のアドレスシンボルを表します。
  SUB1:
        ～
```

(b)
```
  MAX:  .EQU    100             ; .EQU はシンボルに値を設定するアセンブラ制御命令です。
        MOV.B   #MAX,R0L        ; MAX は値 100 を表します。
        ～
```

(c)
```
  BSYM: .BEQU   1,SYM           ; .BEQU はビットデータに値を設定するアセンブラ制御命令です。
        BLD     BSYM            ; BSYM は SYM のビット 1 を表します。
  SYM:  .RES.B  1
        ～
```

(d)
```
  MIN:  .REG    R0              ; .REG はレジスタ別名を定義するアセンブラ制御命令です。
        MOV.W   #100,MIN        ; MIN は R0 の別名になります。
        ～
```

(e)
```
        .SECTION    CD,CODE,ALIGN=2
        ～                      ; .SECTION はセクションを宣言するアセンブラ制御命令です。
                                ; CD はこのセクションの名前となります。
```

(2) シンボルの名付け方

(a) シンボルに使用できる文字

次の ASCII 文字を使用できます。

- 英大文字、英小文字(A～Z, a～z)
- 数字(0～9)
- アンダースコア(_)
- ドル($)

アセンブラはシンボル中の英大文字と英小文字を区別します。

(b) **先頭の文字**

次のいずれかに限ります。

- 英大文字、英小文字(A～Z，a～z)
- アンダースコア(_)
- ドル($)

【注】 ドル($)1 文字はロケーションカウンタを表す予約語です。

(c) **最大文字数**

とくに制限はありません。

(d) **シンボルとして使用できない名前**

次の名称は、シンボルとして使用できません。

(i) 予約語

- レジスタニーモニック(ER0~ER7、E0~E7、R0~R7、R0H~R7H、R0L~R7L、SP、CCR、EXR、MACH、MACL、PC、SBR、VBR)
- 演算子 (STARTOF、SIZEOF、HIGH、LOW、HWORD、LWORD)
- ロケーションカウンタ ($)

(ii) アセンブラ生成シンボル

- 内部シンボル*
  _ $$*mmmmm*　　(*m* は数字(0~F)です)
- 構造化アセンブリ生成シンボル
  _$I*nnnnn*
  _$S*nnnnn*
  _$F*nnnnn*
  _$W*nnnnn*
  _$R*nnnnn*　　(*n* は数字(0~9)です)

【注】* 内部シンボルとはアセンブラの内部処理のため必要なシンボルです。内部シンボルはアセンブルリストやオブジェクトモジュールには出力されません。

(e) **シンボルの定義と参照**

シンボルはラベルとして記述することにより定義されます。参照する時はオペランドに記述します。例外として、.SECTION 制御命令と.MACRO 制御命令では定義するシンボルをオペランドに記述します。また、.MACRO 制御命令で定義したシンボル(マクロ名)はオペレーションに記述して参照します(マクロコール)。

シンボルを参照する場合、定義よりも参照が先に現れることがあります。このような参照を前方参照と呼びます。通常はこのような参照も可能ですが、一部で許されない場合がありますので注意が必要です。

複数のソースファイルでプログラムを構成する場合、ファイルをまたがるシンボル参照が必要になります。定義したシンボルを他のソースファイルから参照できるようにすることを外部定義と呼びます。逆に、他のソースファイルで定義したシンボルを参照することを外部参照と呼びます。

外部定義は.EXPORT、.GLOBAL、.BEXPORT 各制御命令で宣言します。

外部参照は.IMPORT、.GLOBAL、.BIMPORT 各制御命令で宣言します。

外部参照も前方参照と同様、許されないことがありますので注意が必要です。

11.1.4 定数

(1) 整数定数

整数定数は基数をつけて表現します。基数とはその整数定数が何進数であるかを示す記述です。

- 2 進数 ・・・・・・ 基数 B'*と 2 進表現の数値で記述
- 8 進数 ・・・・・・ 基数 Q'*と 8 進表現の数値で記述
- 10 進数 ・・・・・・ 基数 D'*と 10 進表現の数値で記述
- 16 進数 ・・・・・・ 基数 H'*と 16 進表現の数値で記述

【注】* B'：BINARY(2 **進)の意味です。**
Q'：OCTAL(8 **進)の意味です。**O **は数字のゼロと紛らわしいので** Q **を使います。**
D'：DECIMAL(10 **進)の意味です。**
H'：HEXADECIMAL(16 **進)の意味です。**

アセンブラは基数の英大文字と英小文字を区別しません。基数と数値は間を空けずに続けて書いてください。基数は省略しても構いません。基数のない整数定数は(通常は)10 進数として扱われます(.RADIX アセンブラ制御命令によって何進数にするかを指定できます)。

例：

```
.DATA.B B'10001000                    ;
.DATA.B Q'210                         ; これらのソースステートメントは、
.DATA.B D'136                         ; 全く同じ内容を表しています。
.DATA.B H'88                          ;
```

(2) 文字定数

文字定数は文字コードを値とする定数です。4 バイト以内の文字をダブルコーテーション(")で囲んで記述してください。ASCII 文字、シフト JIS コードもしくは EUC コードのかな・漢字、もしくは LATIN1 コードを記述することができます。ASCII 文字は H'09(タブ)、H'20(空白)～H'7E(~)が使用できます。ダブルコーテーション自身をデータとして使う場合は 2 つ続けて書いてください。かな・漢字を記述した場合、シフト JIS コードのときは sjis オプション、EUC コードのときは euc オプションを、LATIN1 コードの時は latin1 オプションを指定してください。なお、シフト JIS コードと EUC コード、LATIN1 コードを混在して使うことはできません。

例：

(a)

```
.DATA.L "ABC"           ;.DATA.L H'00414243 と同じ意味です。
.DATA.W "AB"            ;.DATA.W H'4142     と同じ意味です。
.DATA.B "A"             ;.DATA.B H'41       と同じ意味です。
                                            ; A の ASCII コード･･･ H'41
                                            ; B の ASCII コード･･･ H'42
                                            ; C の ASCII コード･･･ H'43
```

(b)

```
.DATA.B """"            ;ダブルコーテーション 1 文字の文字定数です。
.DATA.L "漢字"          ;漢字
```

11.1.5 ロケーションカウンタ

ロケーションカウンタはオブジェクトコード(実行命令やデータをコンピュータが理解できる形式に変換したコード)を配置するアドレス(ロケーション)を指し示します。ロケーションカウンタの値(以下､ロケーションカウンタ値と称します)はオブジェクトコードの出力に応じて自動的に変化します。また、アセンブラ制御命令により意図的にロケーションカウンタ値を変えることもできます。

例：

```
        ～
.ORG        H'1000          ; ロケーションカウンタに H'1000 を設定しています。
.DATA.W     H'FF            ; このアセンブラ制御命令のオブジェクトコードは長さ 2 バイトです。
                            ; ロケーションカウンタ値は H'1002 に変わります。
.DATA.W     H'F0            ; このアセンブラ制御命令のオブジェクトコードは長さ 2 バイトです。
                            ; ロケーションカウンタ値は H'1004 に変わります。
.DATA.W     H'10            ; このアセンブラ制御命令のオブジェクトコードは長さ 2 バイトです。
                            ; ロケーションカウンタ値は H'1006 に変わります。
                            ; .ORG はロケーションカウンタ値を設定するアセンブラ制御命令です。
                            ; .DATA はデータをメモリ上に確保するアセンブラ制御命令です。
                            ; .W はデータをワード(=2 バイト)単位で扱うとの指定です。
        ～
```

ロケーションカウンタはドル($)で参照できます。

例：

```
LABEL1：    .EQU $          ; LABEL1 というシンボルにこの時点でのロケーションカウンタ値を
                            ; 設定しています。
                            ; .EQU はシンボルに値を設定するアセンブラ制御命令です。
```

11.1.6 式

式は定数やシンボルと演算子を組み合わせて演算結果を求めるものであり、実行命令やアセンブラ制御命令のオペランドに使用します。

(1) 式の要素

式は項、演算子、カッコから構成されます。

(a) 項

項には次のものがあります。

- 定数
- ロケーションカウンタ ($)
- シンボル (レジスタ別名を除く)
- 上記の項と演算子による演算結果

単独の項も式の一種です。

(b) 演算子

表 11.1 に演算子の一覧を示します。

表 11.1 演算子一覧

演算区分	演算子	演算内容	書き方
算術演算	+	単項プラス	+ 項
	-	単項マイナス	- 項
	+	加算	項1 + 項2
	-	減算	項1 - 項2
	*	乗算	項1 * 項2
	/	除算	項1 / 項2
論理演算	~	単項否定	~ 項
	&	論理積	項1 & 項2
	|	論理和	項1 | 項2
	~	排他的論理和	項1 ~ 項2
シフト演算	<<	算術左シフト	項1 << 項2
	>>	算術右シフト	項1 >> 項2
セクション集合演算	STARTOF	セクション集合の先頭アドレスを求める	STARTOF セクション名
	SIZEOF	セクション集合のサイズをバイト単位で求める	SIZEOF セクション名
抽出演算	HIGH	上位バイト抽出	HIGH 項
	LOW	下位バイト抽出	LOW 項
	HWORD	上位ワード抽出	HWORD 項
	LWORD	下位ワード抽出	LWORD 項

【注】 HWORD、LWORD は、H8/300、H8/300L では使用できません。

(c) カッコ

丸カッコ ( ) によって演算の優先順位を変更できます。

(d) 演算の順序

1つの式の中に複数の演算が含まれる場合、演算子の優先順位とカッコ指定によって演算を処理する順序が決まります。アセンブラは次の規則にしたがって演算を処理します。

- 規則1
  カッコでくくられた演算から処理する。
  カッコが多重になっているときはより内側のカッコでくくられた演算を優先する。
- 規則2
  演算子の優先順位が高いものから処理する。
- 規則3
  演算子の優先順位が同じであるときは演算子の結合規則の向きに処理する。

表 11.2 に演算子の優先順位と結合規則を示します。

表 11.2 演算子の優先順位と結合規則

優先順位		演算子	結合規則
1	(高)	+ - ~ STARTOF SIZEOF HIGH LOW HWORD LWORD *1	右から左の順に演算を処理する。
2		* /	左から右の順に演算を処理する。
3		+ -	左から右の順に演算を処理する。
4		<< >>	左から右の順に演算を処理する。
5		&	左から右の順に演算を処理する。
6	(低)	| ~	左から右の順に演算を処理する。

【注】 **同一優先順位の演算子は、結合規則の方向に従って演算されます。**

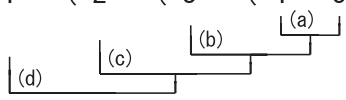
*1 **演算子は単項演算子を示します。**

式の記述例を以下に示します。

例：

(i)

```
1 + ( 2 - ( 3 + ( 4 - 5 ) ) )
```



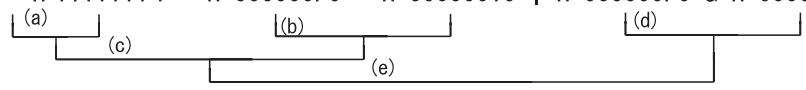
アセンブラは、(a)～(d)の順に計算します。

(a)の結果 ····· -1
(b)の結果 ····· 2
(c)の結果 ····· 0
(d)の結果 ····· 1

} 最終的な結果は、1になります。

(ii)

```
- H'FFFFFFF1 + H'000000F0 * H'00000010 | H'000000F0 & H'0000FFFF
```

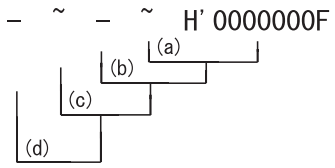


アセンブラは、(a)～(e)の順に計算します。

(a)の結果 ····· H'0000000F
(b)の結果 ····· H'00000F00
(c)の結果 ····· H'00000F0F
(d)の結果 ····· H'000000F0
(e)の結果 ····· H'00000FFF

} 最終的な結果は、H'0000FFFFになります。

(iii)



アセンブラは、(a)～(d)の順に計算します。

(a)の結果 ····· H'FFFFFFF0
(b)の結果 ····· H'00000010
(c)の結果 ····· H'FFFFFFEF
(d)の結果 ····· H'00000011

} 最終的な結果は，H'00000011になります。

(2) 演算の詳細

(a) STARTOF 演算

指定したセクションが最適化リンケージエディタで連結された後のセクション先頭アドレスを求めます。

(b) SIZEOF 演算

指定したセクションが最適化リンケージエディタで連結された後のセクションサイズを求めます。

例：

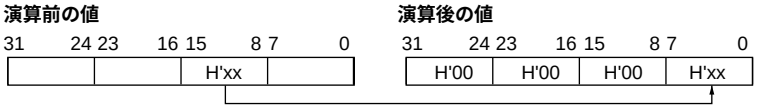
```
            .CPU     2600A
            .SECTION INIT_RAM,DATA,ALIGN=2
            .RES.B   H'100
;
            .SECTION INIT_DATA,DATA,ALIGN=2
INIT_BGN    .DATA.L  STARTOF INIT_RAM                          ; [1]
INIT_END    .DATA.L  STARTOF INIT_RAM + SIZEOF INIT_RAM        ; [2]
;
;
            .SECTION MAIN,CODE,ALIGN=2
INITIAL:
            MOV.L    @INIT_BGN,ER1
            MOV.L    @INIT_END,ER2
            MOV.W    #0,R3
LOOP:
            CMP.L    ER1,ER2
            BEQ      END
            MOV.W    R3,@ER1
            ADDS.L   #1,ER1
            BRA      LOOP
END:
            SLEEP
            .END
```

セクション (INIT_RAM)のデータ領域をゼロで初期化します。

[1]：**セクション** (INIT_RAM)**の先頭アドレスを求めます。**
[2]：**セクション** (INIT_RAM)**の最終アドレスを求めます。**

(c) HIGH 演算

4 バイト値の下位 2 バイトの上位バイトを抽出します。

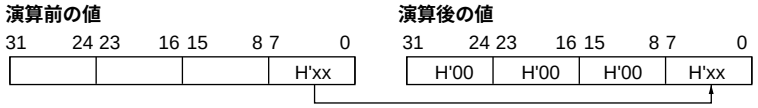


例：

```
LABEL:      .EQU            H'00007FFF
            MOV.W           #HIGH LABEL, R0   ; R0 に H'7F を代入します。
```

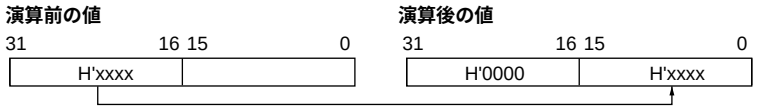
(d) LOW 演算

4 バイト値の最下位バイトを抽出します。



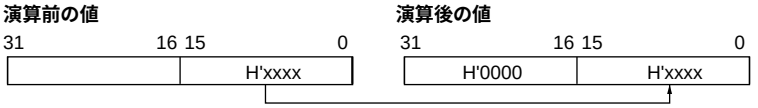
(e) HWORD 演算

4 バイト値の上位 2 バイトを抽出します。



(f) LWORD 演算

4 バイト値の下位 2 バイトを抽出します。



(3) 式に関する注意事項

(a) 内部処理

アセンブラは式の値を 32 ビット符号付きで処理します。

オペランドのサイズが、どのサイズ(8/16/32 ビット)であっても、演算は符号付き 32 ビット演算となります。このため、次の例のような場合には、エラーとなります。

例：

```
MOV.B  #~H'80:8,R0L
```

アセンブラは H'80 を H'00000080 と解釈します。したがって、~H'80 は H'FFFFFF7F となります。
また、H'FFFFF7F は、8 ビット値の範囲外なので、エラーとなります。
これを回避するためには、次のようにしてください。

```
MOV.B  #H'7F:8,R0L                ; 演算結果の値を直接記述する
MOV.B  #~H'80&H'FF:8,R0L          ; 論理積を用いて下位ビットを有効とします
MOV.W  #LOW ~H'80:8,R0L           ; 下位バイト抽出を用いて下位 8 ビットを
                                  ; 有効とします。
```

(b) 論理演算

論理演算で相対アドレスまたは外部参照シンボルを項とすることはできません。

(c) **算術演算**

値が確定しなければならない箇所では、乗算・除算で相対アドレスまたは外部参照シンボルを項とすることはできません。

また、除算で 0 を除数とすることはできません。

例：

```
        .IMPORT         SYM
        .DATA           SYM/10          ; 正常
        .ORG            SYM/10          ; エラーとなる
```

11.1.7 文字列

文字列は一連の文字をデータとして考えます。文字列には次の ASCII 文字を使用できます。

ASCIIコード ┌ H '09 (タブ)
　　　　　 └ H'20 (空白)～H'7E(~)

文字列中の 1 文字はその文字の ASCII コードを値としてもつバイトサイズのデータを表します。また、シフト JIS コードもしくは EUC コードのかな・漢字を記述することができます。文字としてかな・漢字を記述した場合、シフト JIS コードのときは sjis オプション、EUC コードのときは euc オプションを指定してください。欧州文字コード(LATIN1)を使用する場合は、latin1 オプションを指定してください。

指定しない場合はホストマシンに依存する日本語コードとします。

文字列は文字の並びをダブルコーテーション(")で囲んで記述してください。データを表す文字としてダブルコーテーションを使う場合はダブルコーテーションを 2 つ続けて書いてください。

例：

```
        .SDATA          "Hello!"        ;文字列データ Hello!を確保しています。
        .SDATA          "アセンブラ"    ;文字列データアセンブラを確保しています。
        .SDATA          """Hello!"""    ;文字列データ"Hello!"を確保しています。
;       .SDATA は文字列データをメモリ上に確保するアセンブラ制御命令です。
```

【注】 文字定数と文字列の違い

文字定数は数値です。データのサイズは 1 バイト、2 バイト、4 バイトのいずれかになります。文字列は数値として扱えません。データのサイズは 1 バイト以上 255 バイト以下です。

11.1.8 ローカルラベル

(1) **ローカルラベルの役割**

ローカルラベルはアドレスシンボル間で局所的に有効なラベルです。ローカルラベルは有効範囲外の他のシンボルと衝突することがありませんので他のシンボル名を意識せずに局所的な制御ができます。ローカルラベルは通常のアドレスシンボルと同様にラベルに記述することによって定義でき、オペランド内で参照できます。

なお、ローカルラベルはデバッグ時に参照できないほか、次の位置に指定できません。

- マクロ名
- セクション名
- オブジェクトモジュール名
- .ASSGINA、.ASSIGNC、.EQU、.BEQU、.ASSIGN、.REG、.DEFINE のラベル
- .EXPORT、.IMPORT、.GLOBAL、.BEXPORT、.BIMPORT のオペランド

ローカルラベルの使用例を以下に示します。
例：

```
LABEL1:                                          ; ローカルブロック 1 の開始
?0001:      CMP.W   R1､R2
            BEQ     ?0002
            BRA     ?0001
?0002:
LABEL2:                                          ; ローカルブロック 2 の開始
?0001:      CMP.W   R1､R2
            BGE     ?0002
            BRA     ?0001
?0002:
LABEL3:
```

(2) ローカルラベルの名付け方

(a) 文字の先頭

ローカルラベルは先頭が"？"で始まる文字列です。

(b) ローカルラベルに使用できる文字

ローカルラベルは先頭以外の文字が以下の ASCII 文字からなる文字列です。

- 英大文字、英小文字(A～Z, a～z)
- 数字(0～9)
- アンダースコア(__)
- ドル($)

アセンブラは英大文字と英小文字を区別しています。

(c) 最大文字数

2 文字以上 16 文字以内です。17 文字以上記述するとローカルシンボルとして扱われません。

(3) ローカルラベルの有効範囲

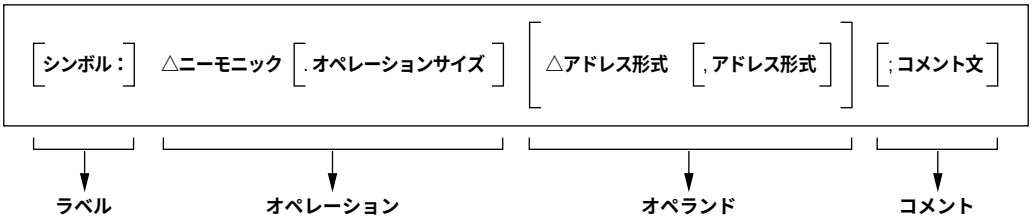
ローカルラベルの有効範囲をローカルブロックといいます。ローカルブロックの区切りはアドレスシンボルまたは.SECTION アセンブラ制御命令です。このローカルブロック内で定義されたローカルラベルは当該ローカルブロック内で参照できます。異なるローカルブロックに属するローカルラベルは、同じ名前であっても別のラベルと解釈し、エラーとはなりません。

【注】 .ASSIGNA、ASSIGNC、.EQU、.BEQU、.ASSIGN、.REG **のアセンブラ制御命令で定義したアドレスシンボルは有効範囲の区切りとはみなしません。**

11.2 実行命令

11.2.1 実行命令の概要

実行命令はマイコンの命令です。
マイコンはメモリ上にある実行命令のオブジェクトコードを解読して実行します。
実行命令のソースステートメントは基本的に次のように記述します。



本節ではニーモニック、オペレーションサイズ、アドレス形式について説明します。

(1) ニーモニック

ニーモニックは実行命令を表します。処理内容を連想させる英文字の名前がニーモニックとして用意されています。
アセンブラはニーモニック中の英大文字と英小文字を区別しません。

(2) オペレーションサイズ

オペレーションサイズはデータを操作する単位です。
指定できるオペレーションサイズは実行命令ごとに異なります。
アセンブラはオペレーションサイズの英大文字と英小文字を区別しません。

指定内容	データのサイズ
B	バイト (1 バイト)
W	ワード (2 バイト)
L	ロングワード (4 バイト)

(3) アドレス形式

アクセスの対象となるデータ領域や分岐先アドレスなどを表します。
指定できるアドレス形式は実行命令ごとに異なります。
表 11.3 に、アドレス形式の一覧を示します。

表 11.3 **アドレス形式一覧**

アドレス形式	名 称	解 説
ERn、Rn、En RnL、RnH	レジスタ直接	レジスタ上の領域です。
@ERn、 @Rn	レジスタ間接	メモリ上の領域です。(E)Rn の値が領域の先頭アドレスを表します。
@ERn+、 @Rn+、 @ERn-、 @Rn-	ポストインクリメント／ デクリメント レジスタ間接	メモリ上の領域です。インクリメント*1／デクリメント*2 する前の(E)Rn の値が領域の先頭アドレスを表します。 マイコンは(E)Rn を先に参照し、後からインクリメント／デクリメントします。
@-ERn、 @-Rn、 @+ERn、 @+Rn	プリデクリメント／ インクリメント レジスタ間接	メモリ上の領域です。デクリメント*2／インクリメント*1 した後の(E)Rn の値が、領域の先頭アドレスを表します。マイコンは(E)Rn を先にデクリメント／インクリメントし、後から参照します。
@(disp,ERn) @(disp,Rn)	ディスプレースメント付き レジスタ間接*3	メモリ上の領域です。領域の先頭アドレスは、 (E)Rn の値＋ディスプレースメント（disp）です。 (E)Rn の内容は変わりません。
@(disp,RnL.B) @(disp,Rn.W) @(disp,ERn.L)	ディスプレースメント付き インデックスレジスタ間接	メモリ上の領域です。領域の先頭アドレスは、 RnL.B/Rn.W/ERn.L の値＋ディスプレースメント（disp）です。 (E)Rn の内容は変わりません。
@abs	絶対アドレス	メモリ上の領域です。領域の先頭アドレスは、 指定した絶対アドレス(abs)です。
#imm	イミディエイトデータ	定数を表します。
@@abs	メモリ間接	メモリ上の領域です。 メモリ上に配置したオペランドを指定し、この内容を分岐アドレスとして分岐します。
@@vec:7	拡張メモリ間接	メモリ上の領域です。 メモリ上に配置したオペランドを指定し、この内容を分岐アドレスとして分岐します。
@(disp,PC)	ディスプレースメント付き PC 相対	メモリ上の領域です。領域の先頭アドレスは PC の値＋ディスプレースメント（disp）です。
@(RnL.B,PC) @(Rn.W,PC) @(ERn.L,PC)	インデックスレジスタ付き PC 相対	メモリ上の領域です。領域の先頭アドレスは PC の値＋RnL.B/Rn.W/ERn.L の値です。 (E)Rn の内容は変わりません。
<CCR> <EXR> <MACH> <MACL> <SBR> <VBR>	コントロールレジスタ	<CCR> ：CPU の内部状態を示します。 <EXR> ：トレース、割り込みを示します。 <MACH>,<MACL> ：積和演算結果を示します。 <SBR> ：ショートアドレスベース値を示します。 <VBR> ：ベクタベース値を示します。

【注】 *1 **インクリメント**
オペレーションサイズがバイトのとき 1、**ワード（**2 **バイト）のとき** 2、**ロングワード（**4 **バイト）のとき** 4 **を加えることです。**

*2 **デクリメント**
オペレーションサイズがバイトのとき 1、**ワード（**2 **バイト）のとき** 2、**ロングワード（**4 **バイト）のとき** 4 **を減じることです。**

*3 **ディスプレースメント**
2 **点間の距離です。本アセンブリ言語ではバイト単位で表現します。**

11.2.2 実行命令に関する注意事項

ニーモニックとアドレス形式の組み合わせにより指定できるオペレーションサイズが異なります。

(1) H8SX シリーズの実行命令のサイズとアドレス形式

(a) 実行命令のサイズ

H8SXマキシマムモード、アドバンストモード、ミドルモードおよび、ノーマルモードの実行命令とオペレーションサイズの組み合わせを**表**11.4に示します。

表 11.4 H8SX シリーズの実行命令とオペレーションサイズの組み合わせ

実行命令	オペレーションサイズ B	W	L	省略時	
ADD	○	○	○	B	
ADDS	×	×	○	L	
ADDX	○	○	○	B	
AND	○	○	○	B	
ANDC	○	×	×	B	
BAND	○	×	×	B	
Bcc	−	−	−	−	*1
BCLR	○	×	×	B	
BCLR/EQ	○	×	×	B	
BCLR/NE	○	×	×	B	
BFLD	○	×	×	B	
BFST	○	×	×	B	
BIAND	○	×	×	B	
BILD	○	×	×	B	
BIOR	○	×	×	B	
BIST	○	×	×	B	
BISTZ	○	×	×	B	
BIXOR	○	×	×	B	
BLD	○	×	×	B	
BNOT	○	×	×	B	
BOR	○	×	×	B	
BRA/BC	−	−	−	−	*1
BRA/BS	−	−	−	−	*1
BRA/S	−	−	−	−	*1
BSET	○	×	×	B	
BSET/EQ	○	×	×	B	
BSET/NE	○	×	×	B	
BSR	−	−	−	−	*1
BSR/BC	−	−	−	−	*1
BSR/BS	−	−	−	−	*1
BST	○	×	×	B	
BSTZ	○	×	×	B	
BTST	○	×	×	B	
BXOR	○	×	×	B	
CLRMAC	−	−	−	−	*1,3
CMP	○	○	○	B	
DAA	○	×	×	B	
DAS	○	×	×	B	
DEC	○	○	○	B	
DIVS	×	○	○	W	
DIVU	×	○	○	W	

実行命令	オペレーションサイズ B	W	L	省略時	
DIVXS	○	○	×	B	
DIVXU	○	○	×	B	
EEPMOV	○	○	×	B	
EXTS	×	○	○	W	
EXTU	×	○	○	W	
INC	○	○	○	B	
JMP	−	−	−	−	*1
JSR	−	−	−	−	*1
LDC	○	○	○	B/L	*4
LDM	×	×	○	L	
LDMAC	×	×	○	L	*3
MAC	−	−	−	−	*1,3
MOV	○	○	○	B	
MOVA	×	×	○	L	*5
MOVFPE	○	×	×	B	
MOVMD	○	○	○	B	
MOVSD	○	×	×	B	
MOVTPE	○	×	×	B	
MULS	×	○	○	W	
MULS/U	×	×	○	L	*3
MULU	×	○	○	W	
MULU/U	×	×	○	L	*3
MULXS	○	○	×	B	
MULXU	○	○	×	B	
NEG	○	○	○	B	
NOP	−	−	−	−	*1
NOT	○	○	○	B	
OR	○	○	○	B	
ORC	○	×	×	B	
POP	×	○	○	*2	
PUSH	×	○	○	*2	
ROTL	○	○	○	B	
ROTR	○	○	○	B	
ROTXL	○	○	○	B	
ROTXR	○	○	○	B	
RTE	−	−	−	−	*1
RTE/L	−	−	−	−	*1
RTS	−	−	−	−	*1
RTS/L	−	−	−	−	*1
SHAL	○	○	○	B	
SHAR	○	○	○	B	

SHLL	○	○	○	B	
SHLR	○	○	○	B	
SLEEP	−	−	−	−	*1
STC	○	○	○	B/L	*4
STM	×	×	○	L	
STMAC	×	×	○	L	*3
SUB	○	○	○	B	

SUBS	×	×	○	L	
SUBX	○	○	○	B	
TAS	○	×	×	B	
TRAPA	−	−	−	−	*1
XOR	○	○	○	B	
XORC	○	×	×	B	

【注】 *1 **サイズの指定はできません。**

*2 **マキシマムモード、アドバンストモード、ミドルモードの場合は** L(**ロングワードサイズ**)**、ノーマルモードの場合は** W(**ワードサイズ**)**になります。**

*3 **乗算器あり指定により有効となる命令です。**

*4 **コントロールレジスタが** CCR **または** EXR **の場合は** B(**バイトサイズ**)**および** W(**ワードサイズ**)**が指定可能であり、省略時は** B **になります。**

コントロールレジスタが SBR **または** VBR **の場合は** L(**ロングワードサイズ**)**のみになります。**

*5 **短縮形のオブジェクトコードを出力する場合は** C **を指定してください。この場合、ソースおよびディスティネーションのレジスタ番号が異なっていてもアセンブラはエラーを出力せず、ディスティネーションのレジスタ番号でオブジェクトコードを出力します。**

MOVA/B.L @(10:16,R1.W),ER1 ; **一般形　オブジェクトコード** H'78197A99000A

MOVA/B.C @(10:16,R1.W),ER1 ; **短縮形　オブジェクトコード** H'7A99000A

(b) **アドレス形式**

H8SXマキシマムモード、アドバンストモード、ミドルモードおよび、ノーマルモードのアドレス形式を**表**11.5に示します。

表 11.5　H8SX **シリーズのアドレス形式**

アドレス形式	**記述** *1
レジスタ直接形式	{ ERn | En | Rn | RnH | RnL }
レジスタ間接形式	@ERn
ポストインクリメントレジスタ間接形式	@ERn+
ポストデクリメントレジスタ間接形式	@ERn-
プリインクリメントレジスタ間接形式	@+ERn
プリデクリメントレジスタ間接形式	@-ERn
ディスプレースメント付きレジスタ間接形式	@(disp [:{2 | 16 | 32}], ERn)
ディスプレースメント付インデックスレジスタ間接形式	@(disp [:{16 | 32}], { RnL.B | Rn.W | ERn.L })
絶対アドレス形式	@abs[: { 8 | 16 | 24 | 32}]
イミディエイトデータ形式	#imm[: {3 | 4 | 5 | 8 | 16 | 32}]
メモリ間接形式	@@abs[:8]
拡張メモリ間接形式	@@vec:7
ディスプレースメント付きプログラムカウンタ相対形式	d[: { 8 | 16 }]
プログラムカウンタインデックス相対形式	{ RnL.B | Rn.W | ERn.L }
コントロールレジスタ	CCR、EXR、MACH、MACL、SBR、VBR

【注】 *1 n …… **レジスタ番号**(0 ～ 7 *2)

disp …… **ディスプレースメント**

abs …… **絶対アドレス**

imm …… **イミディエイトデータ**

vec …… **絶対アドレス**

*2 ER7 **は、**SP(**スタックポインタ**)**と同じです。**

(2) H8S/2600 シリーズの実行命令のサイズとアドレス形式

(a) 実行命令のサイズ

H8S/2600アドバンストモードおよび、ノーマルモードの実行命令とオペレーションサイズの組み合わせを**表**11.6に示します。

表 11.6 H8S/2600 シリーズの実行命令とオペレーションサイズの組み合わせ

実行命令	オペレーションサイズ				
	B	W	L	省略時	
ADD	○	○	○	B	
ADDS	×	×	○	L	
ADDX	○	×	×	B	
AND	○	○	○	B	
ANDC	○	×	×	B	
BAND	○	×	×	B	
Bcc	−	−	−	−	*1
BCLR	○	×	×	B	
BIAND	○	×	×	B	
BILD	○	×	×	B	
BIOR	○	×	×	B	
BIST	○	×	×	B	
BIXOR	○	×	×	B	
BLD	○	×	×	B	
BNOT	○	×	×	B	
BOR	○	×	×	B	
BSET	○	×	×	B	
BSR	−	−	−	−	*1
BST	○	×	×	B	
BTST	○	×	×	B	
BXOR	○	×	×	B	
CLRMAC	−	−	−	−	*1
CMP	○	○	○	B	
DAA	○	×	×	B	
DAS	○	×	×	B	
DEC	○	○	○	B	
DIVXS	○	○	×	B	
DIVXU	○	○	×	B	
EEPMOV	○	○	×	B	
EXTS	×	○	○	W	
EXTU	×	○	○	W	
INC	○	○	○	B	
JMP	−	−	−	−	*1
JSR	−	−	−	−	*1
LDC	○	○	×	B	
LDM	×	×	○	L	

実行命令	オペレーションサイズ				
	B	W	L	省略時	
LDMAC	×	×	○	L	
MAC	−	−	−	−	*1
MOV	○	○	○	B	
MOVFPE	○	×	×	B	
MOVTPE	○	×	×	B	
MULXS	○	○	×	B	
MULXU	○	○	×	B	
NEG	○	○	○	B	
NOP	−	−	−	−	*1
NOT	○	○	○	B	
OR	○	○	○	B	
ORC	○	×	×	B	
POP	×	○	○	*2	
PUSH	×	○	○	*2	
ROTL	○	○	○	B	
ROTR	○	○	○	B	
ROTXL	○	○	○	B	
ROTXR	○	○	○	B	
RTE	−	−	−	−	*1
RTS	−	−	−	−	*1
SHAL	○	○	○	B	
SHAR	○	○	○	B	
SHLL	○	○	○	B	
SHLR	○	○	○	B	
SLEEP	−	−	−	−	*1
STC	○	○	×	B	
STM	×	×	○	L	
STMAC	×	×	○	L	
SUB	○	○	○	B	
SUBS	×	×	○	L	
SUBX	○	×	×	B	
TAS	○	×	×	B	
TRAPA	−	−	−	−	*1
XOR	○	○	○	B	
XORC	○	×	×	B	

【注】 *1 **サイズの指定はできません。**

*2 **アドバンストモードの場合は** L(**ロングワードサイズ**)、**ノーマルモードの場合は** W(**ワードサイズ**) **になります。**

(b) **アドレス形式**

H8S/2600アドバンストモードおよび、ノーマルモードのアドレス形式を**表**11.7に示します。

表 11.7 H8S/2600 **シリーズのアドレス形式**

アドレス形式	記述 *1
レジスタ直接形式	{ ERn | En | Rn | RnH | RnL }
レジスタ間接形式	@ERn
ポストインクリメントレジスタ間接形式	@ERn+
プリデクリメントレジスタ間接形式	@-ERn
ディスプレースメント付きレジスタ間接形式	@(disp[:{16 | 32}], ERn)
絶対アドレス形式	@abs[: { 8 | 16 | 24 | 32}]
イミディエイトデータ形式	#imm[: { 8 | 16 | 32 }]
メモリ間接形式	@@abs[:8]
ディスプレースメント付きプログラムカウンタ相対形式	d[: { 8 | 16 }]
コントロールレジスタ	CCR、EXR、MACH、MACL

【注】*1 n …… **レジスタ番号**(0 ～ 7 *2)

disp …… **ディスプレースメント**

abs …… **絶対アドレス**

imm …… **イミディエイトデータ**

*2 ER7 **は、**SP(**スタックポインタ**)**と同じです。**

(3) H8S/2000 シリーズの実行命令のサイズとアドレス形式

(a) 実行命令のサイズ

H8S/2000アドバンストモードおよび、ノーマルモードの実行命令とオペレーションサイズの組み合わせを**表**11.8に示します。

表 11.8 H8S/2000 **シリーズの実行命令とオペレーションサイズの組み合わせ**

実行命令	オペレーションサイズ				
	B	W	L	省略時	
ADD	○	○	○	B	
ADDS	×	×	○	L	
ADDX	○	×	×	B	
AND	○	○	○	B	
ANDC	○	×	×	B	
BAND	○	×	×	B	
Bcc	−	−	−	−	*1
BCLR	○	×	×	B	
BIAND	○	×	×	B	
BILD	○	×	×	B	
BIOR	○	×	×	B	
BIST	○	×	×	B	
BIXOR	○	×	×	B	
BLD	○	×	×	B	
BNOT	○	×	×	B	
BOR	○	×	×	B	
BSET	○	×	×	B	
BSR	−	−	−	−	*1
BST	○	×	×	B	
BTST	○	×	×	B	
BXOR	○	×	×	B	
CMP	○	○	○	B	
DAA	○	×	×	B	
DAS	○	×	×	B	
DEC	○	○	○	B	
DIVXS	○	○	×	B	
DIVXU	○	○	×	B	
EEPMOV	○	○	×	B	
EXTS	×	○	○	W	
EXTU	×	○	○	W	
INC	○	○	○	B	
JMP	−	−	−	−	*1
JSR	−	−	−	−	*1
LDC	○	○	×	B	
LDM	×	×	○	L	
MOV	○	○	○	B	
MOVFPE	○	×	×	B	
MOVTPE	○	×	×	B	
MULXS	○	○	×	B	
MULXU	○	○	×	B	
NEG	○	○	○	B	
NOP	−	−	−	−	*1
NOT	○	○	○	B	
OR	○	○	○	B	
ORC	○	×	×	B	
POP	×	○	○	*2	
PUSH	×	○	○	*2	
ROTL	○	○	○	B	
ROTR	○	○	○	B	
ROTXL	○	○	○	B	
ROTXR	○	○	○	B	
RTE	−	−	−	−	*1
RTS	−	−	−	−	*1
SHAL	○	○	○	B	
SHAR	○	○	○	B	
SHLL	○	○	○	B	
SHLR	○	○	○	B	
SLEEP	−	−	−	−	*1
STC	○	○	×	B	
STM	×	×	○	L	
SUB	○	○	○	B	
SUBS	×	×	○	L	
SUBX	○	×	×	B	
TAS	○	×	×	B	
TRAPA	−	−	−	−	*1
XOR	○	○	○	B	
XORC	○	×	×	B	

【注】 *1 **サイズの指定はできません。**

*2 **アドバンストモードの場合は** L(**ロングワードサイズ**)**、ノーマルモードの場合は** W(**ワードサイズ**) **になります。**

(b) **アドレス形式**

H8S/2000アドバンストモードおよび、ノーマルモードのアドレス形式を**表**11.9に示します。

表 11.9 H8S/2000 **シリーズのアドレス形式**

アドレス形式	記述 *1
レジスタ直接形式	{ ERn | En | Rn | RnH | RnL }
レジスタ間接形式	@ERn
ポストインクリメントレジスタ間接形式	@ERn+
プリデクリメントレジスタ間接形式	@-ERn
ディスプレースメント付きレジスタ間接形式	@(disp[:{16 | 32}], ERn)
絶対アドレス形式	@abs[: { 8 | 16 | 24 | 32}]
イミディエイトデータ形式	#imm[: { 8 | 16 | 32 }]
メモリ間接形式	@@abs[:8]
ディスプレースメント付きプログラムカウンタ相対形式	d[: { 8 | 16 }]
コントロールレジスタ	CCR、EXR

【注】*1 n …… **レジスタ番号**(0 ～ 7 *2)

disp …… **ディスプレースメント**

abs …… **絶対アドレス**

imm …… **イミディエイトデータ**

*2 ER7 **は、**SP(**スタックポインタ**)**と同じです。**

(4) H8/300H シリーズの実行命令のサイズとアドレス形式

(a) 実行命令のサイズ

H8/300Hアドバンストモードおよび、ノーマルモードの実行命令とオペレーションサイズの組み合わせを**表**11.10に示します。

表 11.10 H8/300H **シリーズの実行命令とオペレーションサイズの組み合わせ**

実行命令	オペレーションサイズ				
	B	W	L	省略時	
ADD	○	○	○	B	
ADDS	×	×	○	L	
ADDX	○	×	×	B	
AND	○	○	○	B	
ANDC	○	×	×	B	
BAND	○	×	×	B	
Bcc	−	−	−	−	*1
BCLR	○	×	×	B	
BIAND	○	×	×	B	
BILD	○	×	×	B	
BIOR	○	×	×	B	
BIST	○	×	×	B	
BIXOR	○	×	×	B	
BLD	○	×	×	B	
BNOT	○	×	×	B	
BOR	○	×	×	B	
BSET	○	×	×	B	
BSR	−	−	−	−	*1
BST	○	×	×	B	
BTST	○	×	×	B	
BXOR	○	×	×	B	
CMP	○	○	○	B	
DAA	○	×	×	B	
DAS	○	×	×	B	
DEC	○	○	○	B	
DIVXS	○	○	×	B	
DIVXU	○	○	×	B	
EEPMOV	○	○	×	B	
EXTS	×	○	○	W	
EXTU	×	○	○	W	
INC	○	○	○	B	
JMP	−	−	−	−	*1
JSR	−	−	−	−	*1
LDC	○	○	×	B	
MOV	○	○	○	B	
MOVFPE	○	×	×	B	
MOVTPE	○	×	×	B	
MULXS	○	○	×	B	
MULXU	○	○	×	B	
NEG	○	○	○	B	
NOP	−	−	−	−	*1
NOT	○	○	○	B	
OR	○	○	○	B	
ORC	○	×	×	B	
POP	×	○	○	*2	
PUSH	×	○	○	*2	
ROTL	○	○	○	B	
ROTR	○	○	○	B	
ROTXL	○	○	○	B	
ROTXR	○	○	○	B	
RTE	−	−	−	−	*1
RTS	−	−	−	−	*1
SHAL	○	○	○	B	
SHAR	○	○	○	B	
SHLL	○	○	○	B	
SHLR	○	○	○	B	
SLEEP	−	−	−	−	*1
STC	○	○	×	B	
SUB	○	○	○	B	
SUBS	×	○	○	L	
SUBX	○	×	×	B	
TRAPA	−	−	−	−	*1
XOR	○	○	○	B	
XORC	○	×	×	B	

【注】 *1 **サイズの指定はできません。**

*2 **アドバンストモードの場合は** L(**ロングワードサイズ**)、**ノーマルモードの場合は** W(**ワードサイズ**) **になります。**

(b) **アドレス形式**

H8/300Hアドバンストモードおよび、ノーマルモードのアドレス形式を**表**11.11に示します。

表 11.11 H8/300H **シリーズのアドレス形式**

アドレス形式	記述 *1
レジスタ直接形式	{ ERn | En | Rn | RnH | RnL }
レジスタ間接形式	@ERn
ポストインクリメントレジスタ間接形式	@ERn+
プリデクリメントレジスタ間接形式	@-ERn
ディスプレースメント付きレジスタ間接形式	@(disp[:{16 | 24}], ERn)
絶対アドレス形式	@abs[: { 8 | 16 | 24 }]
イミディエイトデータ形式	#imm[: { 8 | 16 | 32 }]
メモリ間接形式	@@abs[:8]
ディスプレースメント付きプログラムカウンタ相対形式	d[: { 8 | 16 }]
コントロールレジスタ	CCR

【注】*1 n …… **レジスタ番号**(0 ～ 7 *2)

disp …… **ディスプレースメント**

abs …… **絶対アドレス**

imm …… **イミディエイトデータ**

*2 ER7 **は、**SP(**スタックポインタ**)**と同じです。**

(5) H8/300, H8/300L シリーズの実行命令のサイズとアドレス形式

(a) 実行命令のサイズ

H8/300、H8/300Lの実行命令とオペレーションサイズの組み合わせを**表**11.12に示します。

表 11.12 H8/300, H8/300L シリーズの実行命令とオペレーションサイズの組み合わせ

実行命令	オペレーションサイズ				
	B	W	L	省略時	
ADD	○	○	×	B	
ADDS	×	○	×	W	
ADDX	○	×	×	B	
AND	○	×	×	B	
ANDC	○	×	×	B	
BAND	○	×	×	B	
Bcc	−	−	−	−	*
BCLR	○	×	×	B	
BIAND	○	×	×	B	
BILD	○	×	×	B	
BIOR	○	×	×	B	
BIST	○	×	×	B	
BIXOR	○	×	×	B	
BLD	○	×	×	B	
BNOT	○	×	×	B	
BOR	○	×	×	B	
BSET	○	×	×	B	
BSR	−	−	−	−	*
BST	○	×	×	B	
BTST	○	×	×	B	
BXOR	○	×	×	B	
CMP	○	○	×	B	
DAA	○	×	×	B	
DAS	○	×	×	B	
DEC	○	×	×	B	
DIVXU	○	×	×	B	
EEPMOV	−	−	−	−	*
INC	○	×	×	B	
JMP	−	−	−	−	*
JSR	−	−	−	−	*

実行命令	オペレーションサイズ				
	B	W	L	省略時	
LDC	○	×	×	B	
MOV	○	○	×	B	
MOVFPE	○	×	×	B	
MOVTPE	○	×	×	B	
MULXU	○	×	×	B	
NEG	○	×	×	B	
NOP	−	−	−	−	*
NOT	○	×	×	B	
OR	○	×	×	B	
ORC	○	×	×	B	
POP	×	○	×	W	
PUSH	×	○	×	W	
ROTL	○	×	×	B	
ROTR	○	×	×	B	
ROTXL	○	×	×	B	
ROTXR	○	×	×	B	
RTE	−	−	−	−	*
RTS	−	−	−	−	*
SHAL	○	×	×	B	
SHAR	○	×	×	B	
SHLL	○	×	×	B	
SHLR	○	×	×	B	
SLEEP	−	−	−	−	*
STC	○	×	×	B	
SUB	○	○	×	B	
SUBS	×	○	×	W	
SUBX	○	×	×	B	
XOR	○	×	×	B	
XORC	○	×	×	B	

【注】 * **サイズの指定はできません。**

(b) **アドレス形式**

H8/300および、H8/300Lのアドレス形式を**表**11.13に示します。

表 11.13　H8/300, H8/300L **シリーズのアドレス形式**

アドレス形式	記述 *1
レジスタ直接形式	{ Rn | RnH | RnL }
レジスタ間接形式	@Rn
ポストインクリメントレジスタ間接形式	@Rn+
プリデクリメントレジスタ間接形式	@-Rn
ディスプレースメント付きレジスタ間接形式	@(disp[:16], Rn)
絶対アドレス形式	@abs[: { 8 | 16 }]
イミディエイトデータ形式	#imm[: { 8 | 16 }]
メモリ間接形式	@@abs[:8]
ディスプレースメント付きプログラムカウンタ相対形式	d[: 8]
コントロールレジスタ	CCR

【注】*1　n　……　**レジスタ番号**(0 ～ 7 *2)

disp　……　**ディスプレースメント**

abs　……　**絶対アドレス**

imm　……　**イミディエイトデータ**

*2　R7 **は、**SP(**スタックポインタ**)**と同じです。**

11.3 アセンブラ制御命令

アセンブラ制御命令はアセンブラが解釈、実行する命令です。
書式の下線は、省略時の解釈を示します。
表 11.14 にアセンブラ制御命令の一覧を示します。

表 11.14 アセンブラ制御命令一覧

分類	ニーモニック	機能
CPU に関するもの	.CPU	CPU 種別を指定する
8 ビット短絶対領域に関するもの	.SBR	8 ビット短絶対領域の基点を指定する。
セクションまたはロケーションカウンタに関するもの	.SECTION	セクションを宣言する
	.ORG	ロケーションカウンタ値を設定する
	.ALIGN	ロケーションカウンタ値を境界調整数の倍数に補正する
シンボルに関するもの	.EQU	シンボルに値を設定する
	.ASSIGN	シンボルに値を設定または再設定する
	.REG	レジスタ別名を設定する
	.BEQU	ビットデータ名を設定する
データまたはデータ領域を確保するもの	.DATA	整数データを確保する
	.DATAB	整数データブロックを確保する
	.SDATA	文字列データを確保する
	.SDATAB	文字列データブロックを確保する
	.SDATAC	計数付き文字列データを確保する
	.SDATAZ	ゼロ終端文字列データを確保する
	.RES	データ領域を確保する
	.SRES	文字列データ領域を確保する
	.SRESC	計数付き文字列データ領域を確保する
	.SRESZ	ゼロ終端文字列データ領域を確保する
外部定義または外部参照に関するもの	.EXPORT	外部定義シンボルを宣言する
	.IMPORT	外部参照シンボルを宣言する
	.GLOBAL	外部参照シンボルまたは外部定義シンボルを宣言する
	.BEXPORT	外部定義 BEQU シンボルを宣言する
	.BIMPORT	外部参照 BEQU シンボルを宣言する
	.ABS8	8 ビット短絶対アドレスシンボルを指定する。
	.NOABS8	8 ビット短絶対アドレスシンボルを指定を無効にする。
オブジェクトモジュールに関するもの	.OUTPUT	オブジェクトモジュール、デバッグ情報の出力を制御する
	.DEBUG	シンボルデバッグ情報の部分出力を制御する
	.LINE	デバッグ情報のファイル名、行番号を変更する
	.DISPSIZE	ディスプレースメントサイズを設定する
アセンブルリストに関するもの	.PRINT	アセンブルリストの出力を制御する
	.LIST	ソースプログラムリストの部分出力を制御する
	.FORM	アセンブルリストの行数と桁数を設定する
	.HEADING	ソースプログラムリストのヘッダを設定する
	.PAGE	ソースプログラムリストを改ページする
	.SPACE	ソースプログラムリストに空行を出力する

その他	.PROGRAM	**オブジェクトモジュール名を設定する**
	.RADIX	**基数指定のない整数定数の基数を指定する**
	.END	**ソースプログラムの終わりとエントリポイントを指定する**

.CPU

書　式　　△.CPU△<CPU 種別>

```
<CPU 種別>＝{ H8SXX [:<アドレス空間のビット幅>][:{M|D|MD}]   |
              H8SXA [:<アドレス空間のビット幅>][:{M|D|MD}]   |
              H8SXM [:<アドレス空間のビット幅>][:{M|D|MD}]   |
              H8SXN [:{M|D|MD}]                             |
              2600A [:<アドレス空間のビット幅>]              |
              2600N                                         |
              2000A [:<アドレス空間のビット幅>]              |
              2000N                                         |
              300HA [:<アドレス空間のビット幅>]              |
              300HN                                         |
              300｜300L                                    }
```

ラベルは記述できません

説　明　　.CPU は、作成するオブジェクトプログラムの CPU 種別と動作モード、アドレス空間のビット幅、乗除算器の有無を指定します。

CPU 種別がマキシマムモード、アドバンストモードおよびミドルモードの時のみ、アドレス空間のビット幅が指定できます。

CPU 種別とアドレス空間のビット幅は、次のようになります。

サブオプション名	意　味
H8SXX [：<アドレス空間のビット幅>] [:{M|D|MD}]	H8SX 用マキシマムモードのオブジェクトを作成します。 〈アドレス空間のビット幅〉は、28、32 のいずれかの数値で、それぞれ 256M バイト、4G バイトのアドレス空間を示します。 〈アドレス空間のビット幅〉の省略時解釈は 32 です。 乗除算器の指定ができます。
H8SXA [：<アドレス空間のビット幅>] [:{M|D|MD}]	H8SX 用アドバンストモードのオブジェクトを作成します。 〈アドレス空間のビット幅〉は、20、24、28、32 のいずれかの数値で、それぞれ 1M バイト、16M バイト、256M バイト、4G バイトのアドレス空間を示します。 〈アドレス空間のビット幅〉の省略時解釈は 24 です。 乗除算器の指定ができます。
H8SXM [：<アドレス空間のビット幅>] [:{M|D|MD}]	H8SX 用ミドルモードのオブジェクトを作成します。 〈アドレス空間のビット幅〉は、20、24 のいずれかの数値で、それぞれ 1M バイト、16M バイトのアドレス空間を示します。 〈アドレス空間のビット幅〉の省略時解釈は 24 です。 乗除算器の指定ができます。
H8SXN[:{M|D|MD}	H8SX 用ノーマルモードのオブジェクトを作成します。 乗除算器の指定ができます。
2600A [：<アドレス空間のビット幅>]	H8S/2600 用アドバンストモードのオブジェクトを作成します。 〈アドレス空間のビット幅〉は、20、24、28、32 のいずれかの数値で、それぞれ 1M バイト、16M バイト、256M バイト、4G バイトのアドレス空間を示します。 〈アドレス空間のビット幅〉の省略時解釈は 24 です。
2600N	H8S/2600 用ノーマルモードのオブジェクトを作成します。
2000A [：<アドレス空間のビット幅>]	H8S/2000 用アドバンストモードのオブジェクトを作成します。 〈アドレス空間のビット幅〉は、20、24、28、32 のいずれかの数値で、それぞれ 1M バイト、16M バイト、256M バイト、4G バイトのアドレス空間を示します。 〈アドレス空間のビット幅〉の省略時解釈は 24 です。
2000N	H8S/2000 用ノーマルモードのオブジェクトを作成します。

300HA [:<アドレス空間のビット幅>]	H8/300H 用アドバンストモードのオブジェクトを作成します。 〈アドレス空間のビット幅〉は、20 または 24 の数値で、それぞれ 1M バイト、16M バイトのアドレス空間を示します。 〈アドレス空間のビット幅〉の省略時解釈は 24 です。
300HN	H8/300H 用ノーマルモードのオブジェクトを作成します。
300	H8/300 のオブジェクトを作成します。
300L	H8/300L のオブジェクトを作成します。

乗算器／除算器の有無の指定は、次のようになります。

乗算器／除算器	指定方法
なし／なし	指定なし
あり／なし	M
なし／あり	D
あり／あり	MD

乗算器ありの場合の追加命令は、MAC, LDMAC, STMAC, CLRMAC, MULU/U, MULS/U です。
除算器ありの場合の追加命令は、ありません。

本制御命令は、ソースプログラムの最初に記述してください。アセンブルリストに関する制御命令を除いてソースプログラムの最初にない場合はエラーとなります。
また、この記述は 1 回限り有効です。
CPU 種別の優先順位は cpu オプション、.CPU 制御命令、H38CPU 環境変数の順となります。
指定を省略した場合、H38CPU 環境変数で設定した CPU 種別が有効となります。

例

```
.CPU        2600A:20          ; H8S/2600 アドバンストモードの 1Mbyte
.SECTION    A,CODE,ALIGN=2    ; モード用にアセンブルします。
MOV.L       ER0,ER1
MOV.L       ER0,ER2
```

8 ビット短絶対領域の基点の指定

.SBR

書　式　　△.SBR△[<定数>]
ラベルは記述できません。

説　明　　.SBR は、8 ビット短絶対領域の基点を宣言します。
.SBR <定数>と指定した場合、指定した定数値を 8 ビット短絶対領域の基点とします。基点の最下位 8 ビットは 0 でなければなりません。
.SBR のみ指定した場合、8 ビット短絶対領域の基点は、SBR オプション指定の有無により異なります。SBR オプションを指定した場合、SBR オプションで指定した値となります。SBR オプションを指定していない場合、アドレス空間のビット幅により以下の基点となります。

	CPU/**動作モード**	8 **ビット短絶対アドレス基点**
H8SX **マキシマムモード**	H8SXX[:32]	H'FFFFFF00
	H8SXX:28	H'0FFFFF00
H8SX **アドバンストモード**	H8SXA:32	H'FFFFFF00
	H8SXA:28	H'0FFFFF00
	H8SXA[:24]	H'00FFFF00
	H8SXA:20	H'000FFF00
H8SX **ミドルモード**	H8SXM[:24]	H'00FFFF00
	H8SXM:20	H'000FFF00
H8SX **ノーマルモード**	H8SXN	H'0000FF00

SBR 制御命令が指定できるのは、CPU が H8SXN,H8SXM,H8SXA,H8SXX の場合です。

また、実際に SBR(ショートアドレスベースレジスタ)へアドレスを設定するには、LDC 命令を記述する必要があります。

例

```
.CPU        H8SXA:32
.SECTION    A,CODE,ALIGN=2
.SBR     H'10000            ;H'00010000 番地を SBR として宣言します。
MOV.L    #H'10000,ER1
LDC.L    ER1,SBR            ; 実際に SBR にアドレスを設定します。
～
MOV.B    @H'FFFFFF00,R0L    ;16 ビットでアクセスします。
MOV.B    @H'00010050,R0H    ; 8 ビットでアクセスします。
～
.SBR                        ;SBR の宣言をクリアします。
MOV.L     #H'FFFFFF00,ER1
LDC.L     ER1,SBR           ; 実際に SBR をデフォルトに戻します。
～
MOV.B    @H'FFFFFF00,R0L    ; 8 ビットでアクセスします。
MOV.B    @H'00010050,R0H    ;32 ビットでアクセスします。
～
```

.SECTION

書　式　　△.SECTION△<セクション名>[,<セクション属性>[,<形式種別>]]

```
   <セクション属性>  ={ CODE   |
                       DATA   |
                       STACK  |
                       DUMMY  }
   <形式種別>        ={ LOCATE = <先頭アドレス> |
                       ALIGN  = <境界調整数>   }
```

ラベルは記述できません。

説　明　　.SECTION はセクションを宣言、再開を指定するアセンブラ制御命令です。
セクションとはプログラムの 1 区切りであり、リンケージの処理単位です。
(1) セクションの開始
セクション名の書き方は、シンボル名の書き方と同じです。
またセクションの大文字と小文字とは区別します。

セクション属性は以下のようになります。

- CODE …… コードセクション
- DATA …… データセクション
- STACK …… スタックセクション
- DUMMY …… ダミーセクション

locate=<先頭アドレス>を指定した場合、絶対アドレス形式でオブジェクトを出力します。
align=<境界調整数>を指定した場合、相対アドレス形式でオブジェクトを出力します。
最適化リンケージエディタはそのセクションの先頭が境界調整数の倍数にあたる絶対アドレスにくるように調整します。
形式種別の指定がない場合、align=2 が設定されます。

- 絶対アドレス形式
  絶対アドレス形式では、セクションの先頭アドレスを設定します。
  先頭アドレスの最大値は、次の通りです。

	CPU/**動作モード**	**最大値**
H8SX **マキシマムモード**	H8SXX[:32]	H'FFFFFFFF
	H8SXX:28	H'0FFFFFFF
H8SX **アドバンストモード**	H8SXA:32	H'FFFFFFFF
	H8SXA:28	H'0FFFFFFF
	H8SXA[:24]	H'00FFFFFF
	H8SXA:20	H'000FFFFF
H8SX **ミドルモード**	H8SXM[:24]	H'00FFFFFF
	H8SXM:20	H'000FFFFF
H8SX **ノーマルモード**	H8SXN	H'0000FFFF
H8S/2600 **アドバンストモード**	2600A:32	H'FFFFFFFF
	2600A:28	H'0FFFFFFF
	2600A[:24]	H'00FFFFFF
	2600A:20	H'000FFFFF
H8S/2600 **ノーマルモード**	2600N	H'0000FFFF
H8S/2000 **アドバンストモード**	2000A:32	H'FFFFFFFF
	2000A:28	H'0FFFFFFF
	2000A[:24]	H'00FFFFFF
	2000A:20	H'000FFFFF
H8S/2000 **ノーマルモード**	2000N	H'0000FFFF

H8/300H **アドバンストモード**	300HA[:24]	H'00FFFFFF
	300HA:20	H'000FFFFF
H8/300H **ノーマルモード**	300HN	H'0000FFFF
H8/300	300	H'0000FFFF
H8/300L	300L	H'0000FFFF

・ 相対アドレス形式
相対アドレス形式では、セクションの境界調整数を設定します。
最適化リンケージエディタでは、オブジェクトモジュールを連結する時に相対アドレスセクションの先頭アドレスを境界調整数の倍数に補正します。
境界調整数には、2^n の値が指定できます。

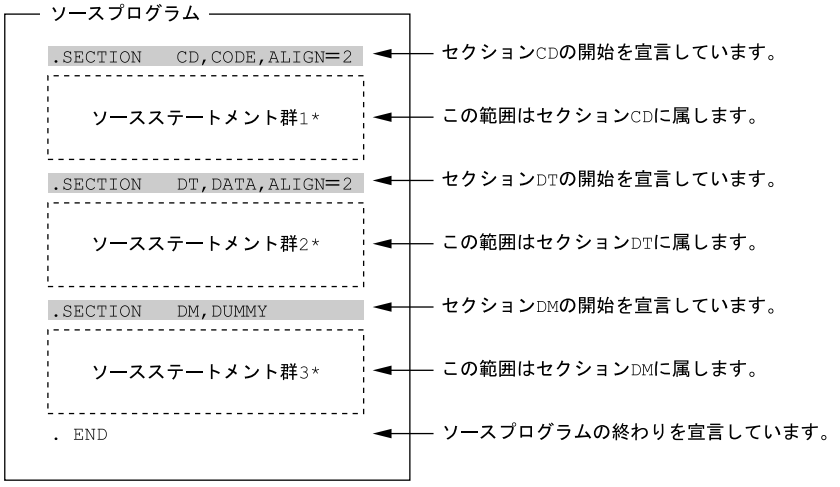
本制御命令で、セクションを宣言しなかった場合は、デフォルトセクションとして次のように設定します。

```
.SECTION  P,CODE,ALIGN=2
```

また、次のいずれかの場合にアセンブラはデフォルトセクションを用意します。
・セクションを宣言しないうちに実行命令を記述している。
・セクションを宣言しないうちにデータを確保するアセンブラ制御命令を記述している。
・セクションを宣言しないうちに.ALIGN アセンブラ制御命令を記述している。
・セクションを宣言しないうちに.ORG アセンブラ制御命令を記述している。
・セクションを宣言しないうちにロケーションカウンタを参照している。
・セクションを宣言しないうちにラベルだけの行を記述している。

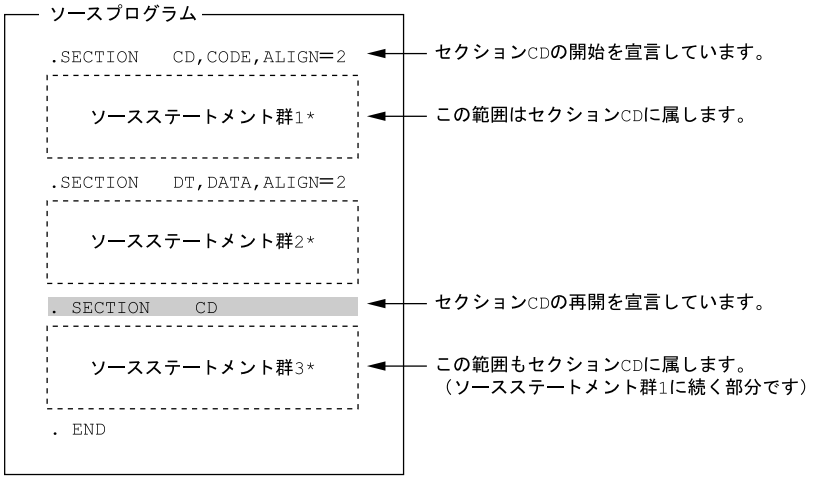
(2)セクションの再開
ソースプログラム中にすでに存在するセクションを再開します。
セクションの再開では、すでに存在するセクションのセクション名を指定します。
セクション属性と形式種別は、最初に宣言したものを継続します。

セクションの宣言について、例をあげて説明します。



【注】* この例では、「ソースステートメント群1～3」に.SECTIONが現れないことを仮定しています。

ソースプログラム中にすでに存在するセクションを再開することができます。
セクションの再開では、すでに存在するセクションのセクション名を指定します.
セクションの再開について、例をあげて説明します。



【注】* この例では、「ソースステートメント群1～3」に.SECTIONが現れないことを仮定しています。

例　この例では「～」の部分に.SECTION が現れないことを仮定しています。

```
            . ALIGN   2              ;  この範囲は、デフォルトセクションPに属します。
            . DATA.W  H'0102,H'0304  ;  デフォルトセクションPはコードセクションであり、
               ～                    ;  境界調整数＝2の相対アドレスセクションです。

        . SECTION    CD, CODE, ALIGN=2
            MOV.W     R0, R1         ;  この範囲は、セクションCDに属します。
            MOV.W     R0, R2         ;  セクションCDはコードセクションであり、
               ～                    ;  境界調整数＝2の相対アドレスセクションです。

        . SECTION    DT, DATA, LOCATE=H' 00001000
  X1 :      . DATA.W  H'2222         ;  この範囲は、セクションDTに属します。
            . DATA.W  H'3333         ;  セクションDTは、データセクションであり、
               ～                    ;  先頭アドレス＝ H' 00001000の絶対アドレス
                                     ;  セクションです。
            . END
```

ロケーションカウンタ値の設定

.ORG

書　式　　△.ORG△<ロケーションカウンタ値>
ラベルは記述できません。

説　明　　.ORG はセクション内のロケーションカウンタ値を指定した値に設定します。
.ORG によって実行命令やデータを特定のアドレスに配置できます。
ロケーションカウンタ値は次のように指定します。
・定数値またはセクション内のアドレスを指定する。
　かつ
・前方参照シンボルを使わずに指定する。

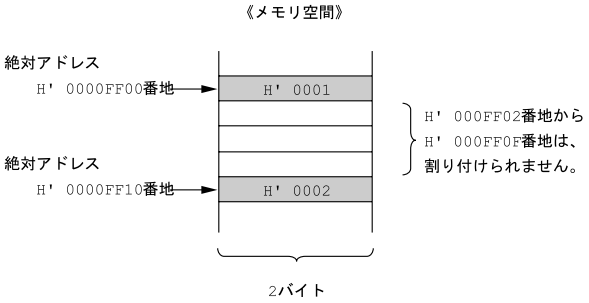
ロケーションカウンタの最大値は、次の通りです。

	CPU/動作モード	最大値
H8SX マキシマムモード	H8SXX[:32]	H'FFFFFFFF
	H8SXX:28	H'0FFFFFFF
H8SX アドバンストモード	H8SXA:32	H'FFFFFFFF
	H8SXA:28	H'0FFFFFFF
	H8SXA[:24]	H'00FFFFFF
	H8SXA:20	H'000FFFFF
H8SX ミドルモード	H8SXM[:24]	H'00FFFFFF
	H8SXM:20	H'000FFFFF
H8SX ノーマルモード	H8SXN	H'0000FFFF
H8S/2600 アドバンストモード	2600A:32	H'FFFFFFFF
	2600A:28	H'0FFFFFFF
	2600A[:24]	H'00FFFFFF
	2600A:20	H'000FFFFF
H8S/2600 ノーマルモード	2600N	H'0000FFFF
H8S/2000 アドバンストモード	2000A:32	H'FFFFFFFF
	2000A:28	H'0FFFFFFF
	2000A[:24]	H'00FFFFFF
	2000A:20	H'000FFFFF
H8S/2000 ノーマルモード	2000N	H'0000FFFF
H8/300H アドバンストモード	300HA[:24]	H'00FFFFFF
	300HA:20	H'000FFFFF
H8/300H ノーマルモード	300HN	H'0000FFFF
H8/300	300	H'0000FFFF
H8/300L	300L	H'0000FFFF

絶対アドレスセクションで指定する場合は、ロケーションカウンタ値はセクションの先頭アドレス以上の値で指定します。本制御命令を絶対アドレスセクションで指定した場合は、設定したロケーションカウンタ値は絶対アドレスとなり、相対アドレスセクションで指定した場合は、相対アドレスになります。

例

```
.SECTION  DT,DATA,LOCATE=H'0000FF00
.DATA.W    H'0001
.ORG  H'0000FF10                ;ロケーションカウンタ値を設定しています。
.DATA.L    H'0002               ;整数データ H'0002 を絶対アドレスの
～                              ;H'0000FF10 番地に確保しています。
```



ロケーションカウンタ値の補正

.ALIGN

書　式　　△.ALIGN△<境界調整数>
ラベルは記述できません。

説　明　　.ALIGN はセクション内のロケーションカウンタ値を境界調整数の倍数に補正します。
.ALIGN によって実行命令やデータを特定の境界（アドレスの区切り）に配置できます。
ロケーションカウンタ値は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
境界調整数には、2^nの値が指定できます。
境界調整数には、アドレス空間の指定により異なります。

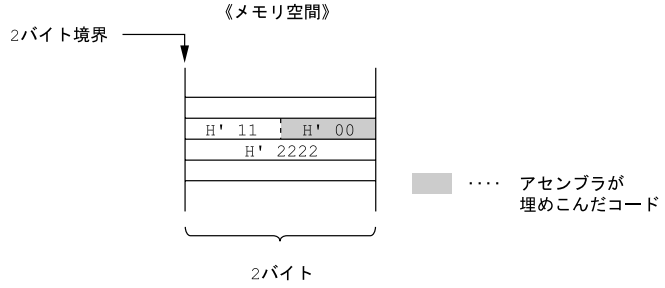
相対アドレスセクションで.ALIGN を使用する場合は
　.SECTION で指定する境界調整 ≧ .ALIGN で指定する境界調整数
となるようにしてください。
コードセクションに.ALIGN を記述すると、アセンブラは NOP 命令のオブジェクトコード*をメモリ上に埋めこみ、ロケーションカウンタ値を補正します。半端なバイトサイズの領域には H'00 を埋めこみます。
データセクション、ダミーセクション、スタックセクションに.ALIGN を記述すると、アセンブラは単にロケーションカウンタ値を補正するだけで、メモリ上にオブジェクトコードを埋めこみません。
【注】* このようなオブジェクトコードはアセンブルリスト上に表れません。

例

```
.CPU        2600A
.SECTION    P,DATA,ALIGN=2    ; …… [1]
.DATA.B     H'11              ; …… [2]
.ALIGN      2                 ; …… [3]
.DATA.W     H'2222
～
```

[1] オブジェクトモジュール結合時に、相対アドレスセクションの先頭アドレスを 2 の倍数番地に補正します。
[2] 1byte のデータ確保を確保するため、次のデータのロケーションカウンタ値が奇数番地になります。
[3] ロケーションカウンタ値を 2 の倍数番地(偶数番地)に補正しています。

バイトサイズの整数データ H'11 がもともと 2 バイト境界に位置するものと仮定します。
アセンブラは下図のようにオブジェクトコードを埋めこんで境界調整します。



シンボルに値を設定

.EQU

書　式　　<シンボル>[:]△.EQU△<シンボル値>

説　明　　.EQU はシンボルに値を設定します。
.EQU で定義したシンボルは再定義できません。
シンボル値は次のように指定します。
・定数値、アドレス、外部参照シンボルの値* を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
シンボル値として許される値は H'00000000～H'FFFFFFFF です。
【注】* 外部参照シンボル、外部参照シンボル＋定数、外部参照シンボル－定数が記述できます。

例

```
            ～
X1:     .EQU        10              ;X1 の値は 10 になります。
X2:     .EQU        20              ;X2 の値は 20 になります。
        CMP.W       #X1,R0          ;CMP.W #10,R0 と同じです。
        BNE         LABEL1
        CMP.W       #X2,R0          ;CMP.W #20,R0 と同じです。
        BEQ         LABEL2
        ～
```

シンボルに値を設定

.ASSIGN

書　式　　<シンボル>［:］△.ASSIGN△<シンボル値>

説　明　　.ASSIGN はシンボルに値を設定するアセンブラ制御命令です。
.ASSIGN で定義したシンボルは.ASSIGN で再定義できます。
シンボル値は次のように指定します。
・定数値またはアドレスを指定する。
　かつ
・前方参照シンボルを使わずに指定する。
シンボル値として許される値は H'00000000～H'FFFFFFFF です。
.ASSIGN による定義は定義した位置から有効です。
.ASSIGN で定義したシンボルには次の使用上の制限があります。
　・外部参照または外部定義できない。
　・デバッガで参照できない。

例

```
        ～
X1:     .ASSIGN     1
X2:     .ASSIGN     2
        CMP.W       #X1,R0          ;CMP.W #1,R0 と同じです。
        BNE         LABEL1
        CMP.W       #X2,R0          ;CMP.W #2,R0 と同じです。
        BEQ         LABEL2
        ～
X1:     .ASSIGN     3
X2:     .ASSIGN     4
        CMP.W       #X1,R0          ;CMP.W #3,R0 と同じです。
        BNE         LABEL3
        CMP.W       #X2,R0          ;CMP.W #4,R0 と同じです。
        BEQ         LABEL4
        ～
```

レジスタ別名の定義

.REG

書　式　　<シンボル>［:］△.REG△(<レジスタ名>)

説　明　　.REG はレジスタ名に別名をつけます。
レジスタ名は次の通りです。
・単一レジスタ … 1 つのレジスタにレジスタ別名をつけます。レジスタで使用できる箇所には全て指定できます。レジスタ名には汎用レジスタを指定できます。
・複数レジスタ … 2 つ以上のレジスタにレジスタ別名をつけます。ただし、CPU 種別が H8SX シリーズ、H8S/2600 シリーズ、H8S/2000 シリーズに限ります。LDM 命令、STM 命令と .REG 命令のオペランドに指定できます。H8SX シリーズではさらに RTS/L および RTE/L 命令のオペランドにも指定できます。レジスタ名には 32 ビット汎用レジスタを指定できます。

レジスタ名の書き方は次の通りです。

指定方法	説明	使用例
単一レジスタ	R0L~R7L, R0H~R7H, R0~R7, E0~E7, ER0~ER7 **のうち 1 つを指定します。**	SINGLEREG .REG (R0) **レジスタ** R0 **に** SINGLREG **という別名を指定します。**
複数レジスタ	**ハイフン(-)で区切って、範囲の形式で一度に複数のレジスタを指定します。** **左側のレジスタの番号より右側のレジスタの番号が小さいとエラーになり、**.REG **命令を無視します。**	RNG1 .REG (ER0-ER3) 4 **個のレジスタ** ER0,ER1,ER2,ER3 **に** RNG1 **と言う別名をつけています。** RNG2 .REG (ER3-ER0) **右側の番号の方が小さいのでエラーです。***
レジスタ別名の再設定	**あらかじめ定義したレジスタ別名をオペランドに指定します。**	ER00 .REG (ER0-ER3) ER01 .REG (ER00) 4 **個のレジスタ** ER0**～**ER3 **に** ER01 **という別名をつけています。**

【注】 .REG で定義したレジスタ別名は再定義できません。
.REG による定義は定義した位置から有効です。
.REG で定義したシンボルには次の使用上の制限があります。
・ 外部参照または外部定義できない。
・ デバッガで参照できない。
H8SX シリーズで指定できるレジスタの組み合わせは、次の通りです。
(ERn-ERn+1)ただし n は 0～6, (ERn-ERn+2) ただし n は 0～5,
(ERn-ERn+3) ただし n は 0～4
H8S/2600 シリーズ、H8S/2000 シリーズで指定できるレジスタの組み合わせは、次の通りです。
(ER0-ER1), (ER2-ER3), (ER4-ER5), (ER6-ER7), (ER0-ER2), (ER4-ER6), (ER0-ER3), (ER4-ER7)

例

```
        .CPU        2600A
RLST1: .REG         (R0)                ; …… [1]
RLST2: .REG         (ER0-ER2)           ; …… [2]
       MOV.W        RLST1,@ER6
       LDM.L        @SP+,(RLST2)
       STM.L        (RLST2),@-SP
```

[1] R0 のレジスタを RLST1 に指定します。
[2] ER0,ER1,ER2 の 3 個のレジスタを RLST2 に指定します。

ビットデータ名指定

.BEQU

書　式　　<シンボル>[:]△.BEQU△<ビット番号>, <置換シンボル名>

説　明　　.BEQU はビット操作命令の対象となるメモリ上の 1 ビットデータに名前をつけます。
ビットデータ名は、ビット操作命令のオペランドに指定できます。
指定されたビットデータ名は、#xx,@aa 形式に置換されます。

ビット番号は次のように指定します。
・定数値またはアドレスを指定する。
　かつ
・前方参照シンボルを使わずに指定する。
ビット番号には、0~7 の値が指定できます。

置換シンボル名は次のように指定します。

CPU 種別	置換シンボル名
H8SX シリーズ	8 ビット絶対アドレス形式 (@aa:8)
H8S/2600 シリーズ	16 ビット絶対アドレス形式(@aa:16)
H8S/2000 シリーズ	32 ビット絶対アドレス形式(@aa:32)
H8/300H シリーズ	8 ビット絶対アドレス形式 (@aa:8)
H8/300 シリーズ	
H8/300L シリーズ	

【注】　.BEQU による定義は定義した位置から有効です。
　　.BEQU で定義したシンボルは、.BEXPORT,.BIMPORT で外部定義/参照できます。

例

```
          .CPU    2600A:32
AD1       .EQU    H'FFFFFF00
AD2       .EQU    H'FFFF8000
AD1B0     .BEQU   0,AD1
AD1B1     .BEQU   1,AD1
AD2B2     .BEQU   2,AD2
AD2B3     .BEQU   3,AD2

          .SECTION   A,CODE,ALIGN=2
          BSET.B  AD1B0                ; BSET.B    #0,@AD1:8
          BSET.B  AD1B1                ; BSET.B    #1,@AD1:8
          BSET.B  AD2B2                ; BSET.B    #2,@AD2:16
          BSET.B  AD2B3                ; BSET.B    #3,@AD2:16
```

ビットデータ名の内容は次のようになります。

```
          AD1B0   ·····  H'FFFFFF00 番地のビット 0
          AD1B1   ·····  H'FFFFFF00 番地のビット 1
          AD2B2   ·····  H'FFFF8000 番地のビット 2
          AD2B3   ·····  H'FFFF8000 番地のビット 3
```

備　考　　ビットデータを指定できるビット操作命令は、次の通りです。
BSET, BCLR, BNOT, BTST, BAND, BIAND, BOR, BIOR, BXOR, BIXOR, BLD, BILD, BST, BIST

.DATA

書　式　　[<シンボル>[:]]△.DATA[.オペレーションサイズ]△整数データ[,...]
<オペレーションサイズ> = { B | W | L }

説　明　　.DATA は整数データを指定されたサイズに従って、メモリ上に確保します。

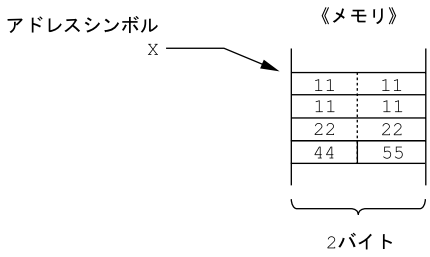
オペレーションサイズおよび整数データの範囲は、次のようになります。

サイズ	データのサイズ		整数データの範囲(10 進表現)	
B	バイト	(1 バイト)	H'00000000~H'000000FF	(0~255)
			H'FFFFFF80〜H'FFFFFFFF	(-128~-1)
W	ワード	(2 バイト)	H'00000000〜H'0000FFFF	(0~65,535)
			H'FFFF8000〜H'FFFFFFFF	(-32,768~-1)
L	ロングワード	(4 バイト)	H'00000000〜H'FFFFFFFF	(0〜4,294,967,295)
			H'80000000〜H'FFFFFFFF	(-2,147,483,648〜-1)

オペレーションサイズを省略すると、アセンブラは.DATA.B (バイトサイズ)と解釈します。
整数データには相対アドレス、外部参照シンボル、前方参照シンボルを含めて任意の値を指定できます。
オペレーションサイズによって指定できる整数データの範囲が異なります。

例

```
        .SECTION    A,DATA,ALIGN=2
X:
        .DATA.L     H'11111111          ;
        .DATA.W     H'2222              ; 整数データを確保しています。
        .DATA.B     H'44,H'55           ;
        ～
```



【注】データは16進数です。

整数データブロック確保

.DATAB

書　式　　[<シンボル>[:]]△.DATAB[.<オペレーションサイズ>]△<ブロック数>,<整数データ>
<オペレーションサイズ> = { B | W | L }

説　明　　.DATAB はブロック数分の整数データを指定したサイズに従ってメモリ上に確保します。

オペレーションサイズによって確保するデータのサイズが決まります。
オペレーションサイズを省略すると.DATAB.B (バイトサイズ)になります。
ブロック数は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。

オペレーションサイズおよびブロック数の範囲は、次のようになります。

サイズ	データのサイズ		ブロック数の範囲(10 進表現)	
B	バイト	(1 バイト)	H'00000001～H'FFFFFFFF	(1～4,294,967,295)
W	ワード	(2 バイト)	H'00000001～H'7FFFFFFF	(1～2,147,483,647)
L	ロングワード	(4 バイト)	H'00000001～H'3FFFFFFF	(1～1,073,741,823)

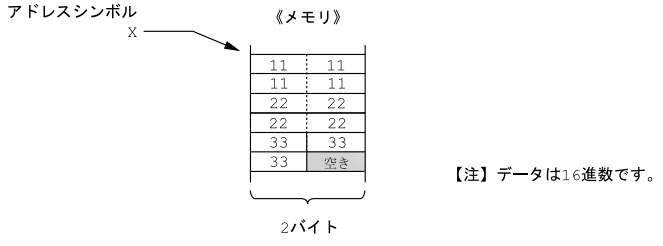
整数データには相対アドレス、外部参照シンボル、前方参照シンボルを含めて任意の値を指定できます。オペレーションサイズによって指定できるブロック数の範囲および整数データの範囲が異なります。

整数データの範囲は、次のようになります。

サイズ	整数データの範囲(10 進表現)	
B	H'00000000~H'000000FF	(0~255)
	H'FFFFFF80～H'FFFFFFFF	(-128~-1)
W	H'00000000～H'0000FFFF	(0~65,535)
	H'FFFF8000～H'FFFFFFFF	(-32,768~-1)
L	H'00000000～H'FFFFFFFF	(0～4,294,967,295)
	H'80000000～H'FFFFFFFF	(-2,147,483,648～-1)

例

```
        .SECTION    A,DATA,ALIGN=2
X:
        .DATAB.L    1,H'11111111        ;
        .DATAB.W    2,H'2222            ; 整数データブロックを確保しています。
        .DATAB.B    3,H'33              ;
        ～
```



.SDATA

書　式　　[<シンボル>[:]]△.SDATA△"<文字列>"[,...]

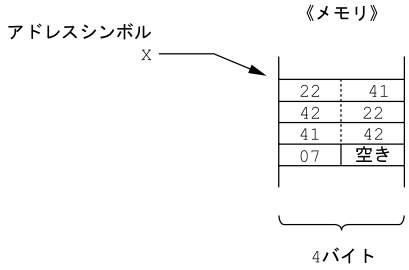
説　明　　.SDATA は文字列データをメモリ上に確保します。

文字列は、文字をダブルコーテーション(")で囲んで指定します。
ダブルコーテーション(")自体を文字として指定する場合は、2 つ続けて記述します。

文字列に制御文字を指定する場合は、ダブルコーテーション(")で囲んだ文字列の直後に制御コードをアングルブラケット(<>)で囲んで記述します。
　"文字列"<制御コード>
制御コードは次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。

例

```
        .SECTION    A,DATA,ALIGN=2
        .SDATA      """AB"""            ; ダブルコーテーションを含む例です。
        .SDATA      "AB"<H'07>          ; 制御文字をつけ加えた例です。
```



【注】1　データは16進数です。

【注】2　文字AのASCIIコード ···· H'41
　　文字BのASCIIコード ···· H'42
　　文字"のASCIIコード ···· H'22

文字列データブロック確保

.SDATAB

書　式　　[<シンボル>[:]]△.SDATAB△<ブロック数>,"<文字列>"

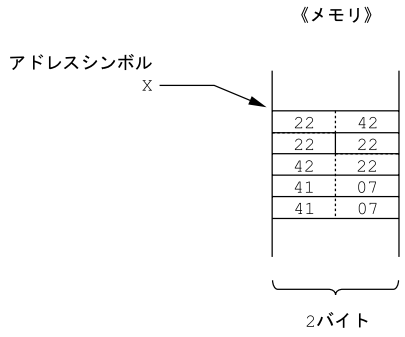
説　明　　.SDATAB はブロック数分の文字列データを連続してメモリ上に確保します。

ブロック数は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
ブロック数には１以上の値を指定してください。
ブロック数の上限値は文字列データの長さ×ブロック数が H'FFFFFFFF(4,294,967,295 バイト)以下になるように指定してください。

文字列は、文字をダブルコーテーション(")で囲んで指定します。
ダブルコーテーション(")自体を文字として指定する場合は、２つ続けて記述します。
文字列に制御文字を指定する場合は、ダブルコーテーション(")で囲んだ文字列の直後に制御コードをアングルブラケット(<>)で囲んで記述します。
　"文字列"<制御コード>
制御コードは次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。

例

```
        .SECTION    A,DATA,ALIGN=2
X:
        .SDATAB     2,"""B"""           ; ダブルコーテーションを含む例です。
        .SDATAB     2,"A"<H'07>         ; 制御文字をつけ加えた例です。
        ～
```



【注】1　データは16進数です。

【注】2　文字AのASCIIコード ···· H' 41
　　文字BのASCIIコード ···· H' 42
　　文字"のASCIIコード ···· H' 22

計数付き文字列データ確保

.SDATAC

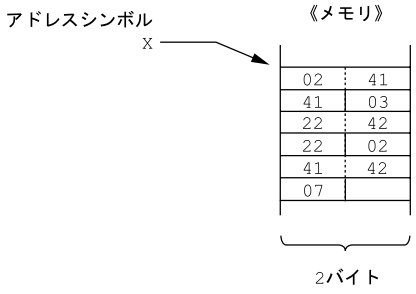
書　式　　[<シンボル>[:]]△.SDATAC△"<文字列>"［,...］

説　明　　.SDATAC は計数付き文字列データをメモリ上に確保します。
計数付き文字列とは文字列の先頭に１バイトの計数をつけ加えたものです。
文字列データを確保する際に、文字列データの先頭に１バイトの計数(文字列のバイト数を示すデータ)を付加して確保します。計数には、計数自体の１バイトは含みません。

文字列は、文字をダブルコーテーション(")で囲んで指定します。
ダブルコーテーション(")自体を文字として指定する場合は、２つ続けて記述します。
文字列に制御文字を指定する場合は、ダブルコーテーション(")で囲んだ文字列の直後に制御コードをアングルブラケット(<>)で囲んで記述します。
　"文字列"<制御コード>
制御文字の制御コードは次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。

例

```
        .SECTION    A,DATA,ALIGN=2
X:
        .SDATAC     "AA"            ; 計数付き文字列データを確保しています。
        .SDATAC     """B"""         ; ダブルコーテーションを含む例です。
        .SDATAC     "AB"<H'07>      ; 制御文字をつけ加えた例です。
```



【注】1　データは16進数です。

【注】2　文字AのASCIIコード　···· H' 41
　　文字BのASCIIコード　···· H' 42
　　文字 " のASCIIコード ···· H' 22

ゼロ終端文字列データ確保

.SDATAZ

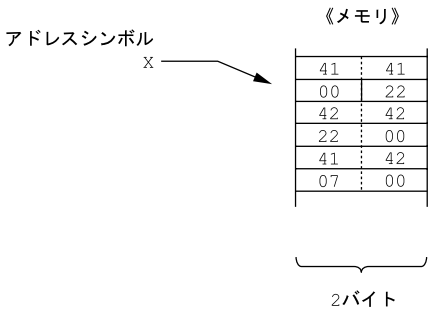
書　式　　[<シンボル>[:]]△.SDATAZ△"<文字列>"[,...]

説　明　　.SDATAZ はゼロ終端文字列データをメモリ上に確保します。
データを確保する際に、文字列データの末尾に１バイトの 0 を付加して確保します。

文字列は、文字をダブルコーテーション(")で囲んで指定します。
ダブルコーテーション(")自体を文字として指定する場合は、２つ続けて記述します。
文字列に制御文字を指定する場合は、ダブルコーテーション(")で囲んだ文字列の直後に制御コードをアングルブラケット(<>)で囲んで記述します。
　"文字列"<制御文字コード>
制御コードは次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。

例

```
        .SECTION    A,DATA,ALIGN=2
X:
        .SDATAZ     "AA"                ; ゼロ終端文字列データを確保しています。
        .SDATAZ     """BB"""            ; ダブルコーテーションを含む例です。
        .SDATAZ     "AB"<H'07>          ; 制御文字をつけ加えた例です。
        ～
```



【注】1　データは16進数です。

【注】2　文字AのASCIIコード ···· H' 41
　　文字BのASCIIコード ···· H' 42
　　文字"のASCIIコード ···· H' 22

データ領域確保

.RES

書　式　　[<シンボル>[:]]△.RES[.<オペレーションサイズ>]△<領域数>
　　<オペレーションサイズ> = { B | W | L }

説　明　　.RES は整数データをメモリ上に確保します。
指定したサイズの整数データを、<領域数>分だけ確保します。
オペレーションサイズによって確保するデータ領域の単位サイズが決まります。
データのサイズと領域数の範囲は、次のようになります。

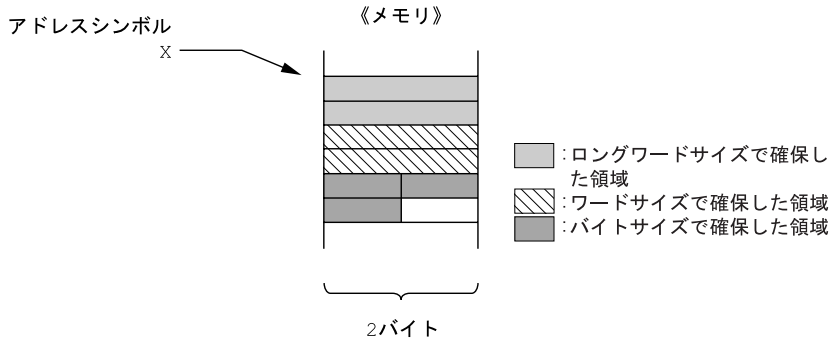
サイズ	データのサイズ		領域数の範囲(10 進表現)
B	バイト	(1 バイト)	H'00000001〜H'FFFFFFFF (1〜4,294,967,295)
W	ワード	(2 バイト)	H'00000001〜H'7FFFFFFF (1〜2,147,483,647)
L	ロングワード	(4 バイト)	H'00000001〜H'3FFFFFFF (1〜1,073,741,823)

オペレーションサイズを省略するとバイトサイズになります。

領域数は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。

例

```
        .SECTION   A,DATA,ALIGN=2
        .ALIGN  4
X:
        .RES.L     1        ; ロングワードサイズの領域１つ分を確保しています。
        .RES.W     2        ; ワードサイズの領域２つ分を確保しています。
        .RES.B     3        ; バイトサイズの領域３つ分を確保しています。
        〜
```



文字列データ領域確保

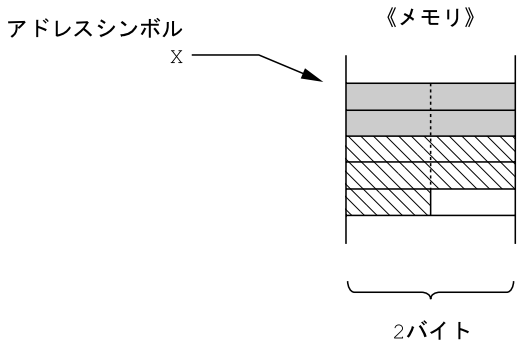
.SRES

書　式　　[<シンボル>[:]]△.SRES△<文字列領域サイズ>[,...]

説　明　　.SRES は文字列用のデータ領域を確保します。
指定した領域サイズ(バイト数)分の領域を確保します。
文字列領域サイズは次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
文字列領域サイズとして許される値は H'00000001～H'FFFFFFFF です。
（10 進表現では 1～4,294,967,295）

例

```
        .SECTION    A,DATA,ALIGN=2
X:
        .SRES       4                   ; 4 バイトの領域を確保しています。
        .SRES       5                   ; 5 バイトの領域を確保しています。
        ～
```



計数付き文字列データ領域確保

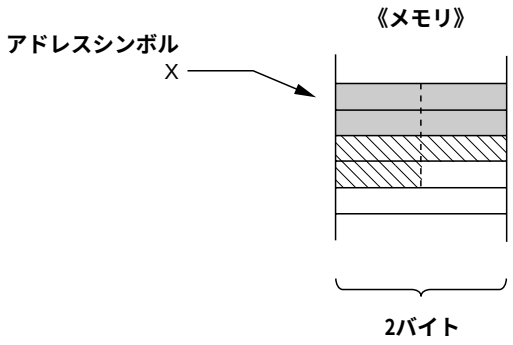
.SRESC

書　式　　[<シンボル>[:]]△.SRESC△<文字列領域サイズ>[,...]

説　明　　.SRESC は計数付き文字列用のデータ領域をメモリ上に確保します。
指定した領域サイズ(バイト数)に、計数の 1 バイトを加えた領域を確保します。
文字列領域サイズは次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
文字列領域サイズとして許される値は H'00000000～H'000000FF です。
（10 進表現では 0～255）
メモリ上に確保される領域のサイズは文字列領域サイズ＋計数用の 1 バイトです。

例

```
        .SECTION    A,DATA,ALIGN=2
X:
        .SRESC      3               ; 3 バイト+計数用の 1 バイトを確保しています。
        .SRESC      2               ; 2 バイト+計数用の 1 バイトを確保しています。
        ～
```



ゼロ終端文字列データ領域確保

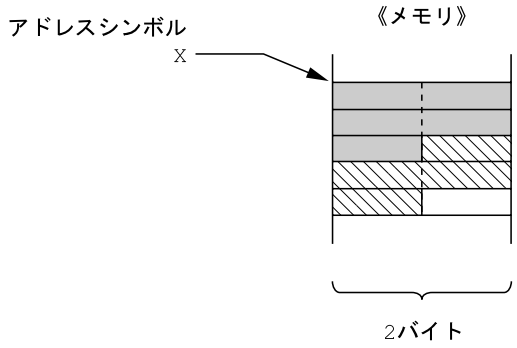
.SRESZ

書　式　　[<シンボル>[:]]△.SRESZ△<文字列領域サイズ>[,…]

説　明　　.SRESZ はゼロ終端文字列用のデータ領域をメモリ上に確保します。
指定した領域サイズ(バイト数)に、ゼロ終端の 1 バイトを加えた領域を確保します。
文字列領域サイズは次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
文字列領域サイズとして許される値は H'00000000～H'000000FF です。
（10 進表現では 0～255）
メモリ上に確保される領域のサイズは文字列領域サイズ＋終端ゼロ用の 1 バイトです。

例

```
        .SECTION    A,DATA,ALIGN=2
X:
        .SRESZ      4           ; 4 バイト+終端ゼロ用の 1 バイトを確保しています。
        .SRESZ      3           ; 3 バイト+終端ゼロ用の 1 バイトを確保しています。
        ～
```



外部定義シンボル宣言

.EXPORT

書　式　　△.EXPORT△<シンボル>[:{ 8 | 16}][,…]
ラベルは記述できません。

説　明　　.EXPORT は外部定義シンボルを宣言します。
ソースプログラム内で定義したシンボルが、別ソースプログラムから参照される場合に宣言します。
外部定義シンボルに指定できるシンボルは、次のものです。
・絶対値を持つシンボル
・アドレス値を持つシンボル
ただし、.ASSIGN 制御命令で定義したシンボル、ダミーセクションのアドレス値を持つシンボルは指定できません。
また、シンボル名にアクセスサイズ(:8 又は:16)を付けることにより、当該シンボルを 8 または 16 ビット絶対アドレス形式でアクセスします。ただし、前方参照シンボルはアクセスサイズ指定がないものをして扱います。本制御命令による宣言は、アクセスサイズの有無に関わらず、最初に設定したシンボルを有効とし、2 度目以降の宣言は無効になります。また、ABS8/NOABS8 制御命令が本制御命令より前方に指定された場合、ABS8/NOABS8 制御命令の指示に従います。
別プログラムソースからシンボルを外部参照するには、外部定義シンボル宣言に対応して、そのシンボルを参照する別ソースプログラムで外部参照シンボル宣言(.IMPORT)する必要があります。

例　　ファイルＡで定義しているシンボルをファイルＢで参照する例です。
・ファイルＡ

```
        .EXPORT    X                ; X を外部定義シンボルとして宣言します。
        ～
X:      .EQU  H'10000000            ; X を定義します。
        ～
```

・ファイルＢ

```
        .IMPORT    X                ; X を外部参照シンボルとして宣言します。
        ～
        .SECTION   A,DATA,ALIGN=2
        .DATA.L    X                ; X を参照します。
        ～
```

外部参照シンボル宣言

.IMPORT

書　式　　△.IMPORT△<シンボル>[:{ 8 | 16}][,…]
ラベルは記述できません。

説　明　　.IMPORT は外部参照シンボルを宣言します。
別ソースプログラム内で定義したシンボルを、参照する場合に宣言します。
ソースプログラム内で定義したシンボルは、外部参照シンボルの宣言はできません。
また、シンボル名にアクセスサイズ(:8 又は:16)を付けることにより、当該シンボルを 8 または 16 ビット絶対アドレス形式でアクセスします。ただし、前方参照シンボルはアクセスサイズ指定がないものをして扱います。

本制御命令による宣言は、アクセスサイズの有無に関わらず、最初に設定したシンボルを有効とし、2 度目以降の宣言は無効になります。また、ABS8/NOABS8 制御命令が本制御命令より前方に指定された場合、ABS8/NOABS8 制御命令の指示に従います。
シンボルを外部参照するには、外部定義シンボル宣言に対応して、そのシンボルを参照する別ソースプログラムで外部参照シンボル宣言(.EXPORT)する必要があります。

例　　ファイルＡで定義しているシンボルをファイルＢで参照する例です。
・ファイルＡ

```
        .CPU        2600A
        .EXPORT     X                   ; X を外部定義シンボルとして宣言します。
        ～
        .SECTION    A,CODE,ALIGN=2
X:      .EQU  H'10000000                ; X を定義します。
        ～
```

・ファイルＢ

```
        .IMPORT     X                   ; X を外部参照シンボルとして宣言します。
        ～
        .SECTION    A,DATA,ALIGN=2
        .DATA.L     X                   ; X を参照します。
        ～
```

外部定義シンボル、外部参照シンボル宣言

.GLOBAL

書　式　　△.GLOBAL△<シンボル>[:{ 8 | 16}][,...]
ラベルは記述できません。

説　明　　.GLOBAL は外部定義シンボルまたは外部参照シンボルを宣言します。
別ソースプログラム内で定義したシンボルを参照する場合と、ソースプログラム内で定義したシンボルが、別ソースプログラムから参照される場合に宣言します。
本制御命令では、ソースプログラム内で定義していないシンボルを外部参照シンボルとし、ソースプログラム内で定義しているシンボルを外部定義シンボルとします。
外部定義シンボルに指定できるシンボルは、次のものです。
・絶対値を持つシンボル
・アドレス値を持つシンボル
ただし、.ASSIGN 制御命令で定義したシンボル、ダミーセクションのアドレス値を持つシンボルは指定できません。
また、シンボル名にアクセスサイズ(:8 又は:16)を付けることにより、当該シンボルを 8 または 16 ビット絶対アドレス形式でアクセスします。ただし、前方参照シンボルはアクセスサイズ指定がないものをして扱います。
本制御命令による宣言は、アクセスサイズの有無に関わらず、最初に設定したシンボルを有効とし、2 度目以降の宣言は無効になります。また、ABS8/NOABS8 制御命令が本制御命令より前方に指定された場合、ABS8/NOABS8 制御命令の指示に従います。

例

```
        .CPU        2600A
        .GLOBAL     PROG1           ; PROG1 を外部定義シンボルとして宣言します。
        .GLOBAL     PROG2           ; PROG2 を外部参照シンボルとして宣言します。
;
        .SECTION    A,CODE,ALIGN=2
PROG1:
        MOV.L       ER0,ER1
        JSR         @PROG2:24
        MOV.L       ER1,ER2
        RTS
;
```

ビットデータ名の外部定義シンボル宣言

.BEXPORT

書　式　　△.BEXPORT△<シンボル>[,...]
ラベルは記述できません。

説　明　　.BEXPORT は.BEQU で指定されたビットデータ名の外部定義シンボルを宣言します。
ソースプログラム内で定義した.BEQU シンボルが、別ソースプログラムから参照される場合に宣言します。

例　　ファイルＡで定義しているシンボルをファイルＢで参照する例です。
・ファイルＡ

```
        .CPU        2600A:32
        .BEXPORT    AD1B0          ; AD1B0 を外部定義シンボルとして宣言します

AD1     .EQU        H'FFFFFF00
AD1B0   .BEQU       0,AD1
      ～
```

・ファイルＢ

```
        .BIMPORT    AD1B0          ; AD1B0 を外部参照シンボルとして宣言します。
      ～
        .SECTION    A,CODE,ALIGN=2
        BSET.B      AD1B0          ; AD1B0 を参照します。
      ～
```

ビットデータ名の外部参照シンボル宣言

.BIMPORT

書　式　　△.BIMPORT△<シンボル>[,...]
ラベルは記述できません。

説　明　　.BIMPORT は.BEQU で指定されたビットデータ名の外部参照シンボルを宣言します。
別ソースプログラム内で定義した.BEQU シンボルを参照する場合に宣言します。
.BIMPORT を定義後にシンボルを.BEQU 以外で定義した場合、ウォーニングを出力します。
同様に、シンボルを.BEQU の定義後に.BIMPORT 定義した場合も、同様にウォーニングを出力します。
.BEQU シンボルを外部参照するには、そのシンボルを定義するソースプログラムで外部定義シンボル宣言(.BEXPORT)する必要があります。

例　　ファイルAで定義しているシンボルをファイルBで参照する例です。
・ファイルA

```
          .BIMPORT   AD1B0          ;AD1B0 を外部参照シンボルとして宣言します。
        ～
          .SECTION   A,CODE,ALIGN=2
          BSET.B     AD1B0          ; AD1B0 を参照します。
        ～
```

・ファイルB

```
          .CPU       2600A:32
          .BEXPORT   AD1B0          ; AD1B0 を外部定義シンボルとして宣言します

AD1       .EQU       H'FFFFFF00
AD1B0     .BEQU      0,AD1
        ～
```

8 ビット短絶対アドレスシンボルの指定

.ABS8
.NOABS8

書　式　　△.ABS8△[<シンボル>[,…]]
ラベルは記述できません。
△.NOABS8
ラベルは記述できません。

説　明　　.ABS8 は、8 ビット絶対アドレス形式でアクセスするシンボルを指定します。.ABS8 のみ指定した場合、本制御命令より前方の全ての外部参照／定義シンボルを対象とします。
.NOABS8 を指定した場合、本制御命令より前方の全ての 8 ビット絶対アドレス形式によるアクセスを指定した外部参照／定義シンボルを対象外とします。

アクセスサイズの優先順位

優先順位		アクセスサイズの形式
高	1	**絶対アドレス形式の確保サイズ**
↑↓	2	.IMPORT／.EXPORT／.GLOBAL **制御命令のアクセスサイズ** .ABS8／.NOABS8 **制御命令**
低	3	abs8／abs16 **オプション**

例

```
.CPU     H8SXX:32
.IMPORT  sym1,sym3,sym5
.IMPORT  sym2:16
.IMPORT  sym4:8
MOV.B    @sym1  ,R1H        ;32 ビット    (指定なし)
MOV.B    @sym2  ,R1H        ;16 ビット    (.IMPORT のアクセスサイズ指定)
MOV.B    @sym3:8,R1H        ; 8 ビット    (確保サイズ指定)
MOV.B    @sym4  ,R1H        ; 8 ビット    (.IMPORT のアクセスサイズ指定)
MOV.B    @sym5  ,R1H        ;32 ビット    (指定なし)
MOV.B    @(sym1+sym2),R1H   ;16 ビット*   (指定なし、16 ビット混在指定)
.ABS8    sym1
MOV.B    @sym1  ,R1H        ; 8 ビット    (.abs8 指定)
MOV.B    @sym2  ,R1H        ;16 ビット    (.IMPORT のアクセスサイズ指定)
MOV.B    @sym3:8,R1H        ; 8 ビット    (確保サイズ指定)
MOV.B    @sym4  ,R1H        ; 8 ビット    (.IMPORT のアクセスサイズ指定)
MOV.B    @sym5  ,R1H        ;32 ビット    (指定なし)
MOV.B    @(sym1+sym2),R1H   ; 8 ビット*   (8 ビット、16 ビット混在指定)
.NOABS8
MOV.B    @sym1  ,R1H        ;32 ビット    (.noabs8 指定)
MOV.B    @sym2  ,R1H        ;16 ビット    (.IMPORT のアクセスサイズ指定)
MOV.B    @sym3:8,R1H        ; 8 ビット    (確保サイズ指定)
MOV.B    @sym4  ,R1H        ;32 ビット    (.noabs8 指定)
MOV.B    @sym5  ,R1H        ;32 ビット    (指定なし)
MOV.B    @(sym1+sym2),R1H   ;16 ビット*   (32 ビット、16 ビット混在指定)
```

備　考　　絶対アドレス形式に外部シンボルを複数記述した場合、アクセスサイズは最小のアクセスサイズを適用します。

オブジェクトモジュール／デバッグ情報出力制御

.OUTPUT

書　式　　△.OUTPUT△<出力指定> [,...]

```
<出力指定> = { obj | noobj |
               dbg | nodbg }
```

ラベルは記述できません。

説　明　　.OUTPUT はオブジェクトモジュールまたはデバッグ情報の出力を制御します。

(1) オブジェクトモジュールの出力

オブジェクトモジュールの出力を制御します。

出力種別は次のようになります。

出力種別	出力制御
obj	**出力**
noobj	**出力抑止**

(2) デバッグ情報の出力

デバッグ情報の出力を制御します。

出力種別は次のようになります。

出力種別	出力制御
dbg	**出力**
nodbg	**出力抑止**

本指定は、オブジェクトモジュールを出力時に有効です。

.OUTPUT を 2 回以上使用し、指定した内容が矛盾しているとエラーとなります。
オブジェクトモジュールとデバッグ情報の出力に関しては、アセンブラはオプションによる指定を優先します。
本制御命令の省略時解釈は、obj および nodbg です。

例　　オブジェクトモジュールとデバッグ情報の出力に関して、オプションによる指定がないことを仮定して結果を説明しています。

例 1：

```
        .OUTPUT     OBJ                 ; オブジェクトモジュールを出力します。
        ～                              ; デバッグ情報は出力しません。
```

例 2：

```
        .OUTPUT     OBJ,DBG             ; オブジェクトモジュールとデバッグ情報を
        ～                              ; 出力します。
```

例 3：

```
        .OUTPUT     OBJ,NODBG           ; オブジェクトモジュールを出力します。
        ～                              ; デバッグ情報は出力しません。
```

備　考　　デバッグ情報はデバッガでプログラムをデバッグするときに必要な情報であり、オブジェクトモジュールの一部となります。
デバッグ情報はソースステートメントの行に関する情報、シンボルに関する情報などを含みます。

シンボルデバッグ情報の部分出力制御

.DEBUG

書　式　　△.DEBUG△<出力指定>
　　<出力指定> = { ON | OFF }

ラベルは記述できません。

説　明　　.DEBUG はシンボルデバッグ情報の部分出力を制御します。
ソースプログラム中のシンボルのうち、デバッグに必要なものだけを出力したい場合に使用します。デバッグに必要なシンボルに限定してシンボルデバッグ情報を出力するとアセンブル時間を短縮できるなどの利点があります。
.DEBUG による指定はオブジェクトモジュールを出力し、かつ、デバッグ情報を出力する場合に限り有効です。
出力種別は、以下のようになります。

出力種別	出力制御
on	出力
off	出力抑止

本制御命令は、何度でも指定できます。また指定内容は本制御命令のソースステートメントに対して有効です。
本制御命令は、デバッグ情報の出力時のみ有効です。
本制御命令の省略時解釈は、on です。

例

```
.SECTION    A,CODE,ALIGN=2
.DEBUG      OFF             ; アセンブラは次のソースステートメント
                            ; からシンボルデバッグ情報を出力しません。
～
.DEBUG      ON              ; アセンブラは次のソースステートメント
                            ; からシンボルデバッグ情報を出力します。
～
```

補　足　　シンボルデバッグ情報は、デバッグ情報のうちのシンボルに関するものを示します。

行番号の変更

.LINE

書　式　　△.LINE△["<ファイル名>",]<行番号>

ラベルは記述できません。

説　明　　.LINE は、デバッグ情報のファイル名、行番号の変更を行います。
本制御命令は、C/C++ソースレベルデバッグを支援します。
C/C++コンパイラは、アセンブリソースプログラムを出力した際、本制御命令を埋め込みます。
これによりコンパイラが出力したアセンブリソースプログラムに対しても、C/C++ソースファイルの行情報を出力できます。

本制御命令を、指定した次の行からアセンブラが管理するファイル名、行番号は、本制御命令で指定した内容に切り替わります。
本制御命令で指定したファイル名、行番号は本制御命令が記述されているファイル内でのみ有効です。

例

```
ch38 -code=asmcode -debug test.c
```

Cソースプログラム（test.c）

```
int      func()         /*1*/
{                       /*2*/
  int  i, j;            /*3*/
                        /*4*/
  j=0;                  /*5*/
  for (i=1;i<=10;i++)   /*6*/
    j+=i;               /*7*/
  return(j);            /*8*/
}                       /*9*/
```

アセンブリソースプログラム（test.src）

```
            .CPU        2600A:24
            .EXPORT     _func
            .SECTION    P,CODE,ALIGN=2
            .LINE       "/asm/test.c",1
_func:                      ; function: func
            .LINE       2
            .LINE       5
            .LINE       6
            SUB.L       ER0,ER0
            MOV.B       #1,R0L
            .LINE       6
L5:
            .LINE       7
            ADD.W       R0,E0
            .LINE       6
            INC.W       #1,R0
            .LINE       6
            CMP.W       #10,R0
            BLE         L5:8
            .LINE       8
            MOV.W       E0,R0
            .LINE       9
            RTS
            .END
```

ディスプレースメントサイズの設定

.DISPSIZE

書　式　　△.DISPSIZE△<対象項目>=<ビット数> [,…]
<対象項目> = { FBR | XBR | FRG | XRG | FWD | XTN | ALL }

ラベルは記述できません。

説　明　　.DISPSIZE は分岐命令のディスプレースメント、ディスプレースメント付きレジスタ間接形式のディスプレースメントが前方参照値、外部参照値である場合のディスプレースメントのデフォルトサイズを設定します。
本制御命令の対象となるのは、ディスプレースメントサイズ (:8,:16,:24,:32)の指定がないディスプレースメントです。
対象項目は、次の通りです。

指定項目	内容
FBR	**前方参照の分岐命令**
XBR	**外部参照の分岐命令**
FRG	**前方参照のディスプレースメント付きレジスタ間接形式**
XRG	**外部参照のディスプレースメント付きレジスタ間接形式**
FWD	FBR, FRG **の同時指定**
XTN	XBR, XRG **の同時指定**
ALL	FBR, XBR, FRG, XRG **の同時指定**

ビット数は、以下のようになります。

CPU	**出力方法** *1
H8SX **マキシマムモード**	FBR=8, 16、XBR=8, 16、FRG=16, 32、XRG=16, 32、FWD=16、XTN=16、ALL=16
H8SX **アドバンストモード**	FBR=8, 16、XBR=8, 16、FRG=16, 32、XRG=16, 32、FWD=16、XTN=16、ALL=16
H8SX **ミドルモード**	FBR=8, 16、XBR=8, 16、FRG=16, 32、XRG=16, 32、FWD=16、XTN=16、ALL=16
H8SX **ノーマルモード**	FBR=8, 16、XBR=8, 16、FRG=16、XRG=16
H8S/2600 **アドバンストモード**	FBR=8, 16、XBR=8, 16、FRG=16, 32、XRG=16, 32、FWD=16、XTN=16、ALL=16
H8S/2600 **ノーマルモード**	FBR=8, 16、XBR=8, 16、FRG=16、XRG=16
H8S/2000 **アドバンストモード**	FBR=8, 16、XBR=8, 16、FRG=16, 32、XRG=16, 32、FWD=16、XTN=16、ALL=16
H8S/2000 **ノーマルモード**	FBR=8, 16、XBR=8, 16、FRG=16、XRG=16
H8/300H **アドバンストモード**	FBR=8, 16、XBR=8, 16、FRG=16, 24、XRG=16, 24、FWD=16、XTN=16、ALL=16
H8/300H **ノーマルモード**	FBR=8, 16、XBR=8, 16、FRG=16、XRG=16
H8/300, H8/300L	FBR=8、XBR=8、FRG=16、XRG=16

【注】下線部は、指定を省略した場合の設定です。

*1 H8/300、H8/300L **では、**FBR=8, XBR=8, FRG=16, XRG=16 **固定なので意味を持ちません。**

本制御命令は、何度でも指定することができます。
指定内容は本制御命令以降のソースステートメントに対して有効となります。
FBR は、optimize オプションと br_relative オプションがない場合に有効です。

例

```
        .CPU        2600A

        .SECTION    A,CODE,ALIGN=2
        .DISPSIZE   FBR=16              ; [1]
        BRA         sym                 ; BRA sym:16 と同じです。

        .DISPSIZE   FBR=8               ; [2]
        BRA         sym                 ; BRA sym:8 と同じです。
sym:
        MOV.W       R0,R1
```

[1] 前方参照の分岐命令のディスプレースメントサイズを 16 ビットに設定します。
[2] 前方参照の分岐命令のディスプレースメントサイズを 8 ビットに設定します。

アセンブルリストの出力制御

.PRINT

書　式　　△.PRINT△<出力指定> [,…]
　　<出力指定> = { LIST |NOLIST |SRC | NOSRC |
　　CREF |NOCREF |SCT | NOSCT }
ラベルは記述できません。

説　明　　.PRINT は出力指定により、
(1) アセンブルリスト
(2) ソースプログラムリスト
(3) クロスリファレンスリスト
(4) セクション情報リスト
　の各リストの出力/出力抑止をを制御します。
各出力指定により制御される内容は以下のとおりです。

項目	出力指定 *1		意味	制御内容
	出力	出力抑止		
(1)	list	nolist	アセンブルリストの出力制御 *2	アセンブルリストの出力/出力抑止
(2)	src	nosrc	ソースプログラムリストの出力制御 *3 *4	ソースプログラムリストの出力/出力抑止
(3)	cref	nocref	クロスリファレンスリストの出力制御 *3 *5	クロスリファレンスリストの出力/出力抑止
(4)	sct	nosct	セクション情報リストの出力制御 *3 *6	セクション情報リストの出力/出力抑止

【注】 *1　本指定は 1 度限り有効です。
*2　list/nolist オプションの指定がない場合に有効です。
*3　アセンブルリスト出力時のみ有効です。
*4　source/nosource オプションの指定がない場合に有効です。
*5　cross_reference/nocross_reference オプションの指定がない場合に有効です。
*6　section/nosection オプションの指定がない場合に有効です。

.PRINT を 2 回以上使用して指定内容が矛盾するとエラーとなります。

例
```
        .PRINT      LIST, SRC, NOCREF, NOSCT
;
        .SECTION    A,CODE,ALIGN=2
START
        MOV.W       R0,R1
        MOV.W       R0,R2
```

アセンブルリストにソースプログラムだけを出力します。

ソースプログラムリストの部分出力制御

.LIST

書　式　　△.LIST△<出力指定> [,...]

<出力指定> = { ON | OFFｴﾗｰ! ﾌﾞｯｸﾏｰｸが定義されていません。 |
COND | NOCOND | DEF | NODEF | CALL | NOCALL |
EXP | NOEXP | STR | NOSTR | CODE | NOCODE }

ラベルは記述できません。

説　明　　.LIST は出力指定により次のような働きをします。

(1) ソースステートメントの部分出力
(2) プリプロセッサ機能のソースステートメントの部分出力
(3) オブジェクトコード表示行の部分出力

各出力指定により制御される内容は以下のとおりです。

項目	出力指定		意味	制御内容
	出力	出力抑止		
(1)	on	off	**ソースステートメントの出力制御**	**本命令以降のソースステートメント**
(2)	cond	nocond	**条件つき不成立の出力制御** *1	.AIF, .AIFDEF **の不成立部分**
	def	nodef	**定義の出力制御** *1	**マクロ定義部分** .AREPEAT, .AWHILE **定義部分** .INCLUDE **制御文** .ASSIGNA, .ASSIGNC **制御文**
	call	nocall	**コールの出力制御** *1	**マクロコール文** .AIF, .AIFDEF, .AENDI **制御文**
	exp	noexp	**展開の出力制御** *1	**マクロ展開部分** .AREPEAT, .AWHILE **展開部分**
	str	nostr	**構造化の出力制御** *1	**構造化アセンブリ展開部分**
(3)	code	nocode	**オブジェクトコード表示行の出力制御** *1	**制御命令のオブジェクトコード表示が、ソースステートメントの行数を超える部分**

【注】*1　本指定は、show/noshow オプションの指定がない場合に有効です。

本制御命令は、ソースプログラムリスト上に表示しません。
本制御命令は何回でも指定でき、指定内容は本制御命令以降のソースステートメントに対して有効です。

例

```
        .PRINT      list
;
        .list       off                 ; ····· [1]
        .include    "bbb.h"             ;
        .list       on                  ; ····· [2]
        .section    A,CODE,ALIGN=2
START
      MOV.W         R0,R1
      MOV.W         R0,R2
```

ソースステートメントの出力を部分的に抑止しています。[1]～[2]の間のソースステートメントは、ソースプログラムリスト上に出力しません。

.FORM

書　式　　△.FORM△<サイズ指定>[,...]
　　<サイズ指定> = { LIN = <行数> | COL = <桁数> }
ラベルは記述できません。

説　明　　.FORM はアセンブルリストの１ページあたりの行数と１行あたりの桁数を設定します。
行数と桁数は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。

<行数>、<桁数>の許される値の範囲、および.FORM による指定もオプションによる指定もない場合の解釈は次の通りです。

指定内容	意味	許される値 *3	未指定時
LIN＝<**行数**>	１**ページあたりの行数** *1	20 ~ 255	60
COL＝<**桁数**>	１**行あたりの桁数** *2	79 ~ 255	132

【注】*1　lines オプションの指定がない場合、有効になります。
　　*2　columns オプションの指定がない場合、有効になります。
　　*3　範囲外の値を指定した時は、20 より小さい場合は 20、255 より大きい場合は 255 が指定されます。この場合、エラーは表示されません。

アセンブルリストの行数と桁数に関して、アセンブラはオプションによる指定を優先します。
.FORM は１つのソースプログラムで何回でも使えます。
設定したサイズ指定は、本制御命令以降のページに対して有効になります。

例　　アセンブルリストの行数と桁数の設定に関して、オプションによる指定がないことを仮定して結果を説明しています。

```
        ～
        .FORM  LIN=60,COL=200        ; このページからアセンブルリスト１ページ
                                     ; を 60 行にします。
                                     ; また、この行からアセンブルリスト１行を
                                     ; 200 桁にします。
        ～
        .FORM  LIN=55,COL=150        ; このページからアセンブルリスト１ページ
                                     ; を 55 行にします。
                                     ; また、この行からアセンブルリスト１行を
                                     ; 150 桁にします。
        ～
```

ソースプログラムリストヘッダ設定

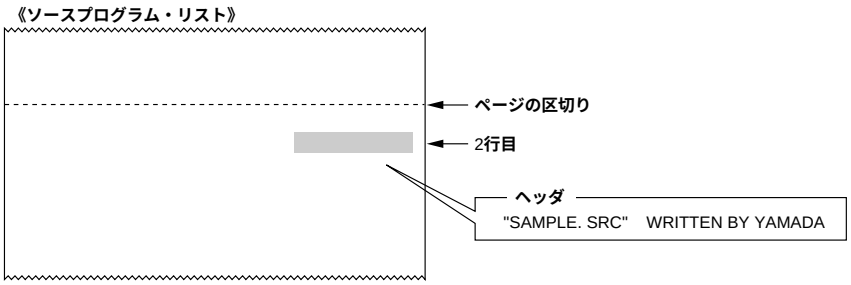
.HEADING

書　式　　△.HEADING△"<文字列>"
ラベルは記述できません。

説　明　　.HEADING はソースプログラムリストのページヘッダに表示するタイトルを設定します。
ヘッダとして設定できるのは 60 文字以内の文字列です。ただし、60 文字を超えた場合でもエラーメッセージは出力しません。
文字列は、文字をダブルコーテーション(")で囲んで指定します。
ダブルコーテーション自体を文字として指定する場合は、2 つ続けて記述します。
.HEADING は 1 つのソースプログラムの中で何回でも使えます。
ページヘッダのタイトルは、本制御命令がリストの 1 行目にある場合はそのページから、ページの 2 行目以降で設定している場合は次のページから表示します。

例

```
～
.HEADING   """SAMPLE.SRC""  WRITTEN BY YAMADA"
～
```



ソースプログラムリストの改ページ

.PAGE

書　式　　△.PAGE
ラベルは記述できません。

説　明　　.PAGE はソースプログラムリストを改ページします。
本制御命令がリストの１行目にある場合、その改ページ指定は無効になります。
また、ソースプログラム・リスト上に表示されません。
本制御命令は、ソースプログラムリスト出力時に有効です。

例

```
        ～
        .PRINT      LIST
        .SECTION    A,CODE,ALIGN=2
START
        MOV.W       R0,R1
        MOV.W       R0,R2

;
        .PAGE
        .SECTION    B,DATA,ALIGN=2
DAT
        .DATA.W     H'0001
        .DATA.W     H'0002
        ～
```

《ソースプログラム・リスト》

```
    4  00000000  0D01                4             MOV.W      R0,R1
    5  00000002  0D02                5             MOV.W      R0,R2
```

← 改ページ

```
*** H8S,H8/300 ASSEMBLER Ver. 4.0 ***   07/18/00 21:28:14
PROGRAM NAME =

    9  00000000                      9             .SECTION   B,DATA,ALIGN=2
   10  00000000                     10     DAT
   11  00000000  0001               11             .DATA.W    H'0001
   12  00000002  0002               12             .DATA.W    H'0002
```

ソースプログラムリストの空行出力

.SPACE

書　式　　△.SPACE[△<行数>]
ラベルは記述できません。

説　明　　.SPACE は空行を指定の行数分ソースプログラムリストに出力します。
オペランドを省略すると空行を１行出力します。

行数は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
行数として許される値は 1～50 です。
1 より小さい場合は 1 に、50 より大きい場合には 50 に設定されます。この場合、エラーは表示しません。

本制御命令で出力する空行には行番号などの表示がありません。
また、空行を出力して改ページが生じる場合、アセンブラは改ページ以降の空行を出力しません。

本制御命令は、ソースプログラムリスト上に表示されません。
本制御命令は、ソースプログラムリストの出力時に有効です。

例

```
        .SECTION   A,DATA,ALIGN=2
        .DATA.W    H'1111
        .DATA.W    H'2222
        .DATA.W    H'3333
        .DATA.W    H'4444              ; セクションが切り替わる箇所で、
        .SPACE  5                      ; 5 行の空行を挿入しています。
        .SECTION   B,DATA,ALIGN=2
        ～
```

《ソースプログラム・リスト》

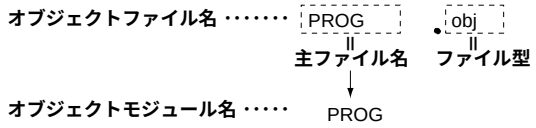
```
 *** H8S,H8/300 ASSEMBLER Ver. 4.0 ***   07/18/00 13:35:58
PROGRAM NAME=

   1  00000000                    1          .SECTION  A,DATA,ALIGN=2
   2  00000000 1111               2          .DATA.W   H'1111
   3  00000002 2222               3          .DATA.W   H'2222
   4  00000004 3333               4          .DATA.W   H'3333
   5  00000006 4444               5          .DATA.W   H'4444









   7                              7          .SECTION  B,DATA,ALIGN=2
          ～
```

オブジェクトモジュール名設定

.PROGRAM

書　式　　△.PROGRAM△<オブジェクトモジュール名>
ラベルは記述できません。

説　明　　.PROGRAM はオブジェクトモジュール名を設定します。
オブジェクトモジュール名とは最適化リンケージエディタがオブジェクトモジュールを識別するために必要とする名前です。
オブジェクトモジュール名のつけ方はシンボルの名付け方と同じです。
アセンブラはオブジェクトモジュール名の英大文字と英小文字を区別します。
本制御命令による設定は最初の１回だけが有効です。アセンブラは２回め以降の指定を無視します。
本制御命令による設定がない場合、デフォルト（暗黙）のオブジェクトモジュール名を設定します。デフォルトはオブジェクトファイル（オブジェクトモジュールの出力先）の主ファイル名です。



オブジェクトモジュール名はプログラムの中で使用しているシンボル名と重複しても構いません。

例

```
.PROGRAM    PROG1          ; オブジェクトモジュール名として PROG1 を
                           ; 設定しています。
～
```

.RADIX

書　式　　△.RADIX△<基数指定>
　　<基数指定>={B | Q | D | H}
ラベルは記述できません。

説　明　　.RADIX は基数指定のない整数定数の基数を設定します。
基数指定の内容によって基数のない整数定数が何進数になるかが決まります。
基数のない整数定数が 16 進数になるよう指定した場合（基数指定 H）、整数定数の一番上位の桁が A～F であるときはその上に 0 をつけ加えてください。
（アセンブラは A～F で始まる記述をシンボルと見なします）
本制御命令による指定は指定した位置から有効です。

指定内容	基数のない整数定数
B	2 進数
Q	8 進数
D	10 進数
H	16 進数

本制御命令を省略した場合、基数のない整数定数は 10 進数となります。

例　　・例 1

```
        ～
        .RADIX  D
X:      .EQU        100             ; 100 は 10 進数です。
        ～
        .RADIX  H
Y:      .EQU        64              ; 64 は 16 進数です。
        ～
```

・例 2

```
        ～
        .RADIX  H
Z:      .EQU        0F              ; F と書くとシンボルと見なされるので
                                    ; 先頭に 0 を付けています。
        ～
```

ソースプログラム終端／エントリポイントの指定

.END

書　式　　△.END［△<実行開始アドレス>]
ラベルは記述できません。

説　明　　.END はソースプログラムの終わりを示します。
本制御命令が出現した時点で、アセンブラはアセンブル処理を終了します。
オペランドに指定したシンボルをエントリポイントとします。
シンボルには外部定義シンボルを指定します。

例

```
        .EXPORT    START
        .SECTION   P,CODE,ALIGN=4
START:
        ～
        .END       START        ; ソースプログラムの終了を宣言しています。
                                ; シンボル START がエントリポイントになります。
```

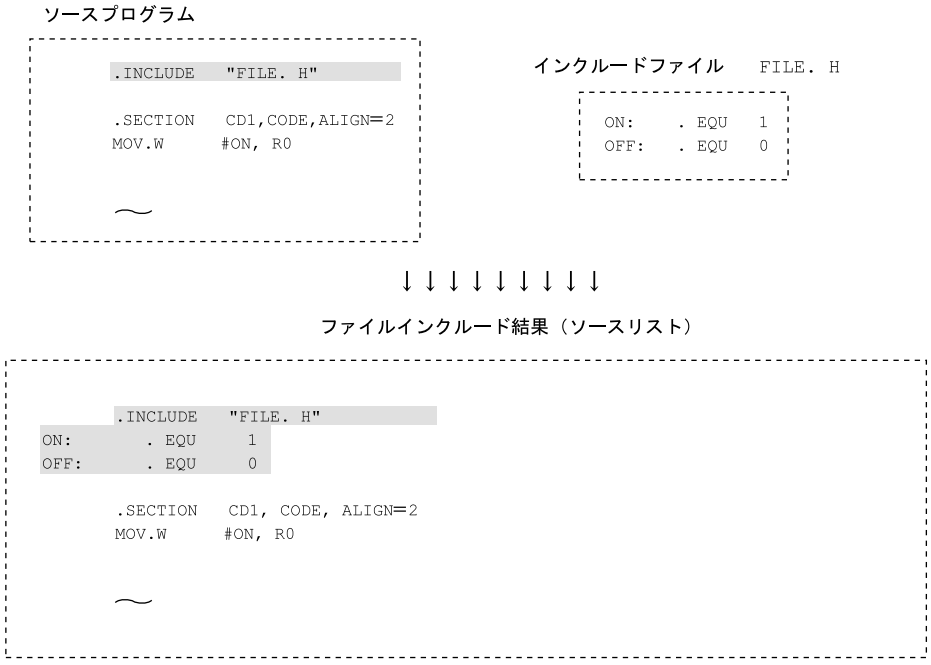
11.4 ファイルインクルード機能

ファイルインクルードとはアセンブルするソースファイルに他のソースファイルを取り込む機能です（以下、取り込まれる側のソースファイルをインクルードファイルといいます）。

ファイルインクルード機能に関する制御文として.INCLUDE 制御文があります。

プログラマが.INCLUDE 制御文を記述した位置に指定のインクルードファイルが取り込まれます。

例：



指定インクルードファイルの取り込み

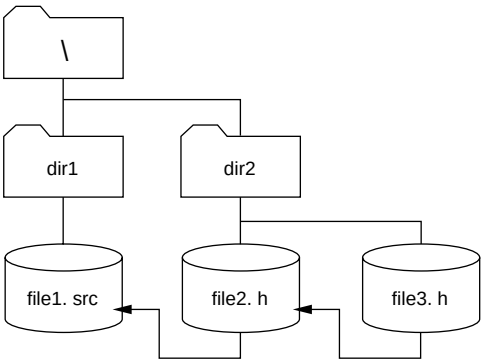
.INCLUDE

書　式　　△.INCLUDE△"<ファイル名>"

ラベルは記述できません。

説　明　　.INCLUDE は指定したインクルードファイルを取り込みます。
ファイル名として主ファイル名だけを指定した場合、ファイル型なしのファイル名が有効となります。（アセンブラによるファイル型の仮定なし）
ファイル名はディレクトリを含めた形で指定することができます。
ディレクトリは絶対パス（ルートディレクトリからの経路）または相対パス（カレントディレクトリからの経路）で指定します。
インクルードファイルの中へさらに別のファイルを取り込むこともできます。インクルードは 30 段階までネストすることができます。
.INCLUDE で指定したディレクトリ名は include オプションで変更することができます。

例　　ディレクトリが下図のような構造になっているとき、以下のことを実行するとします。



ルートディレクトリ（¥）からアセンブラを起動
入力ソースファイルは¥dir1¥file1.src
file1.src に file2.h をインクルード
file2.h に file3.h をインクルード
起動コマンドは次のようになります。

```
 >asm38 ¥dir1¥file1.src （RET）
```

file1.src にはつぎのインクルード制御文が必要になります。

```
 .INCLUDE "dir2¥file2.h"  ; ¥がカレントディレクトリです。（相対パス指定）
 または
 .INCLUDE "¥dir2¥file2.h" ; 絶対パス指定
```

file2.h にはつぎのインクルード制御文が必要になります。

```
 .INCLUDE "file3.h"   ; ¥dir2 がカレントディレクトリです。（相対パス指定）
 または
 .INCLUDE "¥dir2¥file3.h" ; 絶対パス指定
```

【注】UNIX の場合、円マーク(¥)をスラッシュ（/）に替えてください。

11.5 条件つきアセンブリ機能

11.5.1 条件つきアセンブリ機能の概要

条件つきアセンブリ機能は次のようなアセンブルを簡単に実現します。

- ソースプログラムの文字列を他の文字列に置き換える
- ソースプログラムの一部分をアセンブルするか否か条件によって切り替える
- ソースプログラムの一部分を繰り返し展開してアセンブルする

(1) プリプロセッサ変数

アセンブル条件を記述するための変数をプリプロセッサ変数といいます。
プリプロセッサ変数の型には整数型と文字型があります。

(a) 整数型プリプロセッサ変数

.ASSIGNA 制御文または、assigna オプションで整数値を定義します(.ASSIGNA 制御命令では再定義が可能)。

参照する時は、プリプロセッサ変数の先頭にバックスラッシュ(円記号)とアンパサンド<¥&>を付けます。

例：

```
FLAG: .ASSIGNA  1                          ; FLAG に整数値 1 を設定しています。
       ～
      .AIF ¥&FLAG EQ 1                     ; .AIF 1 EQ 1 と同じです。
      MOV.W    R0,R1                       ; MOV.W R0,R1 をアセンブルします。
      .AENDI
       ～
```

(b) 文字型プリプロセッサ変数

.ASSIGNC 制御文または、assignc オプションで文字列を定義します(.ASSIGNC 制御命令では再定義が可能)。

参照する時は、プリプロセッサ変数の先頭にバックスラッシュ(円記号)とアンパサンド<¥&>を付けます。

例：

```
FLAG: .ASSIGNC  "ON"                       ; FLAG に文字列 ON を設定しています。
       ～
      .AIF "¥&FLAG" EQ "ON"                ; .AIF "ON" EQ "ON" と同じです
      MOV.W    R0,R1                       ; MOV.W R0,R1 をアセンブルします。
      .AENDI
       ～
```

(2) 置換シンボル

.DEFINE 制御文で定義します。
ソースプログラムの一部分を指定によって置き換えることができます。
コーディングは次のようになります。

例：

```
SYM1: .DEFINE  "R1"
       ～
      MOV.W    SYM1,R0                     ; MOV.W R1,R0 に置き換えられます。
       ～
```

(3) 条件つきアセンブル

ソースプログラムの一部分をアセンブルするか否か条件によって切り替えることができます。条件つきアセンブリの条件には関係演算子で判別する比較型条件つきアセンブルと置換シンボルで判別する定義型条件つきアセンブルがあります。

(a) 比較型条件つきアセンブル

比較型条件つきアセンブルは条件の成立か不成立かによりアセンブルする範囲を切り替えます。コーディングは次のようになります。

```
          ～
.AIF      比較型条件
    条件が成立したときアセンブルする部分
.AELIF    比較型条件
    条件が成立したときアセンブルする部分          この部分は省略可能
.AELSE
    全ての条件が成立しないときアセンブルする部分
.AENDI
          ～
```

例：

```
  ～
.AIF  "¥&FLAG" EQ "ON"
MOV.W    R0,R2                          ; FLAG が"ON"のときアセンブルします。
MOV.W    R1,R3                          ;
.AELSE
MOV.W    R2,R0                          ; FLAG が"ON"でないときアセンブルします。
MOV.W    R3,R1                          ;
.AENDI
  ～
```

(b) 定義型条件つきアセンブル

定義型条件つきアセンブルは置換シンボルが定義されているか否かによりアセンブルする範囲を切り替えます。コーディングは次のようになります。

```
          ～
.AIFDEF   定義型条件
    条件の置換シンボルが定義されているとき
    アセンブルする部分
.AELSE
    条件の置換シンボルが定義されていないとき          この部分は省略可能
    アセンブルする部分
.AENDI
          ～
```

例：

```
  ～
.AIFDEF  FLAG
MOV.W    R0,R3                    ; .AIFDEF 制御文で参照するより前に
MOV.W    R1,R4                    ; FLAG が.DEFINE 制御文で定義されて
MOV.W    R2,R5                    ; いるときアセンブルします。
.AELSE
MOV.W    R3,R0                    ; .AIFDEF 制御文で参照するより前に
MOV.W    R4,R1                    ; FLAG が.DEFINE 制御文で定義されて
MOV.W    R5,R2                    ; いないときアセンブルします。
.AENDI
  ～
```

(4) 繰り返し展開

ソースプログラムの一部分を指定の回数だけ繰り返し展開してアセンブルすることができます。コーディングは次のようになります。

```
            ～
.AREPEAT    繰り返し回数
    繰り返しの対象となる部分
.AENDR
            ～
```

例：

```
MOV.B  R1L,R1H
.AREPEAT 2                                  ; 繰り返しの回数を指定します。
ADD.B  R0L,R1L
ADD.B  R2L,R3L
.AENDR
ADD.B  R3L,R1H
```

[展開後]

```
MOV.B  R1L,R1H
ADD.B  R0L,R1L  ┐
ADD.B  R2L,R3L  ├ 展開した部分
ADD.B  R0L,R1L  │
ADD.B  R2L,R3L  ┘
ADD.B  R3L,R1H
```

.AREPEAT～.AENDR 間のソースステートメントを、2 回繰り返して展開し、展開した部分をアセンブルします。

(5) 条件つき繰り返し展開

ソースプログラムの一部分を条件が成立している間、繰り返し展開してアセンブルすることができます。コーディングは次のようになります。

```
                〜
      .AWHILE     繰り返し条件
         繰り返しの対象となる部分
      .AENDW
                〜
```

例：

```
        MOV.B   R0H,R0L
 COUNT  .ASSIGNA  2             ; COUNT に 2 を設定します。
        .AWHILE   ¥&COUNT NE 0 ; COUNT≠0 の間、展開を行います。
        ADD.B   R0L,R1L
        ADD.B   R0L,R2L
        INC.B   R0L
 COUNT  .ASSIGNA ¥&COUNT-1     ; COUNT から 1 を減算しています。
        .AENDW
        MOV.B   R0L,@SP
```

[展開後]

```
        MOV.B   R0H,R0L
        ADD.B   R0L,R1L
        ADD.B   R0L,R2L
        INC.B   R0L
 COUNT  .ASSIGNA ¥&COUNT-1
        ADD.B   R0L,R1L
        ADD.B   R0L,R2L
        INC.B   R0L
 COUNT  .ASSIGNA ¥&COUNT-1
        MOV.B   R0L,@SP
```

展開した部分

.AWHILE〜.AENDW 間のソースステートメントを、COUNT≠0 の間だけ繰り返して展開し、展開した部分をアセンブルします。

11.5.2 条件つきアセンブリ機能に関する制御文

表 11.15 に条件つきアセンブリ機能の制御文の一覧を示します。

表 11.15 **条件付アセンブリ機能一覧**

分類	ニーモニック	機能
変数定義に関するもの	.ASSIGNA	整数型プリプロセッサ変数を定義します。再定義が可能です。
	.ASSIGNC	文字型プリプロセッサ変数を定義します。再定義が可能です。
	.DEFINE	プリプロセッサ置換文字列を定義します。再定義できません。
条件分岐に関するもの	.AIF .AELIF .AELSE .AENDI	ソースプログラムの一部分をアセンブルするか否か、条件によって切り替えます。 条件成立の場合は.AIF 以降、不成立の場合は.AELIF または.AELSE 以降のソースプログラムをアセンブルします。
	.AIFDEF .AELSE .AENDI	ソースプログラムの一部分をアセンブルするか否か、置換シンボルの定義によって切り替えます。 置換シンボルが定義されている場合は.AIFDEF 以降、定義されていない場合は.AELSE 以降のソースプログラムをアセンブルします。
繰り返し展開に関するもの	.AREPEAT .AENDR	ソースプログラムの一部分(.AREPEAT と.AENDR の間)を指定の回数だけ繰り返し展開してアセンブルします。
	.AWHILE .AENDW	ソースプログラムの一部分(.AWHILE と.AENDW の間)を条件が成立している間、繰り返し展開してアセンブルします。
	.EXITM	.AREPEAT、.AWHILE による繰り返し展開を中断します。
その他	.AERROR	プリプロセッサ展開時のエラー処理をします。
	.ALIMIT	プリプロセッサでの.AWHILE の展開の上限値を設定します。

.ASSIGNA

書　式　　<プリプロセッサ変数名>[:]△.ASSIGNA△<値>

説　明　　.ASSIGNA 制御文は、プリプロセッサ変数を定義します。
プリプロセッサ変数名の付け方はシンボルの名付け方と同じです。
また、プリプロセッサ変数名の最大文字数は 32 文字で英大文字と英小文字を区別します。
.ASSIGNA で定義したプリプロセッサ変数は.ASSIGNA によって再定義できます。
プリプロセッサ変数の値は次の形式で指定します。
・定数（整数定数、文字定数）
・既に定義したプリプロセッサ変数
・上記を項とする式
定義したプリプロセッサ変数は本制御文以降のソースステートメントに対して有効です。
プリプロセッサ変数は以下の箇所で参照できます。
・.ASSIGNA、.ASSIGNC 制御文
・.AIF、.AELIF、.AREPEAT、.AWHILE 制御文
・マクロ本体（.MACRO～.ENDM 間のソースステートメント）
プリプロセッサ変数を参照する場合、前にバックスラッシュ(円記号)とアンパサンド<¥&>を付けて記述してください。
　¥&プリプロセッサ変数名[']
アポストロフィ（'）はプリプロセッサ変数名とソースステートメントの区別を明確にしたい場合に記述します。
オプションでプリプロセッサ文字列が定義されている場合、同名のプリプロセッサ変数に対する.ASSIGNA は無効となります。

例

```
FLAG    .ASSIGNA   1                ; FLAG に 1 を設定します。
;
        .SECTION   A,CODE,ALIGN=2
START
        .AIF       ¥&FLAG EQ 1      ; .AIF 1 EQ 1 と同じです。
        MOV.W      R0,R2
        .AENDI
        .AIF       ¥&FLAG EQ 2      ; .AIF 1 EQ 2 と同じです。
        MOV.W      R1,R2
        .AENDI
;
FLAG    .ASSIGNA   2                ; FLAG の値を 2 に変更します。
;
        .AIF       ¥&FLAG EQ 1      ; .AIF 2 EQ 1 と同じです。
        MOV.W      R0,R2
        .AENDI
        .AIF       ¥&FLAG EQ 2      ; .AIF 2 EQ 2 と同じです。
        MOV.W      R1,R2
        .AENDI
```

整数型プリプロセッサ変数 FLAG を.AIF で参照しています。

文字型プリプロセッサ変数定義

.ASSIGNC

書　式　　<プリプロセッサ変数名>[:]△.ASSIGNC△"<文字列>"

説　明　　.ASSIGNC 制御文は、文字型プリプロセッサ変数を定義します。
プリプロセッサ変数名の付け方はシンボルの名付け方と同じです。
また、プリプロセッサ変数名の最大文字数は 32 文字で英大文字と英小文字を区別します。
.ASSIGNC で定義したプリプロセッサ変数は.ASSIGNC によって再定義できます。
文字列は文字および既に定義したプリプロセッサ変数をダブルコーテーション（"）で囲んで指定します。
定義したプリプロセッサ変数は本制御文以降のソースステートメントに対して有効です。

プリプロセッサ変数は以下の箇所で参照できます。
・.ASSIGNA、.ASSIGNC 制御文
・.AIF、.AELIF、.AREPEAT、.AWHILE 制御文
・マクロ本体（.MACRO～.ENDM 間のソースステートメント）
プリプロセッサ変数を参照する場合、前にバックスラッシュ(円記号)とアンパサンド<¥&>を付けて記述してください。
　¥&プリプロセッサ変数名[']
アポストロフィ(')はプリプロセッサ変数名とソースステートメントの区別を明確にしたい場合に記述します。
コマンドライン・オプションでプリプロセッサ文字列が定義されている場合、同名のプリプロセッサ変数に対する.ASSIGNC は無効となります。

例

```
FLAG1 .ASSIGNC   "ON"                  ; FLAG1 に"ON"を設定します。
;
      .SECTION   A,CODE,ALIGN=2
START
      .AIF       "¥&FLAG1" EQ "ON"     ; .AIF "ON" EQ "ON"と同じです。
      MOV.W      R0,R2
      .AENDI
      .AIF       "¥&FLAG1" EQ "OFF"    ; .AIF "ON" EQ "OFF"と同じです。
;
FLAG2 .ASSIGNC   "OFF"                 ; FLAG を文字列 OFF に変更します。
;
      .AIF       "¥&FLAG2" EQ "ON"     ; .AIF "OFF" EQ "ON"と同じです。
      MOV.W      R3,R5
      .AENDI

      .AIF       "¥&FLAG2" EQ "OFF"    ; .AIF "OFF" EQ "OFF"と同じです。
      MOV.W      R4,R5
      .AENDI
;
FLAG  .ASSIGNC   "¥&FLAG1' AND ¥&FLAG2" ; FLAG と AND の区別を明確にするため
                                        ; "'"を使います。
                                        ; FLAG は結果的に"ON AND OFF"に
                                        ; なります。
```

文字型プリプロセッサ変数 FLAG を.AIF で参照しています。

プリプロセッサ置換文字列定義

.DEFINE

書　式　　<シンボル>[:]△.DEFINE△"<置換文字列>"

説　明　　.DEFINE 制御文はシンボルの対応する置換文字列に置き換えることを指定します。
.DEFINE と .ASSIGNC との違いは以下の点です。
(a) .ASSIGNC で定義したシンボルはプリプロセッサ文でしか使用できませんが、.DEFINE で定義したシンボルは任意のステートメントで使用できます。
(b) .ASSIGNA、.ASSIGNC で定義したシンボルは「¥&シンボル」の形式で参照しますが、.DEFINE で定義したシンボルは「シンボル」の形式で参照します。
(c) .DEFINE で定義したシンボルは再定義できません。
(d) オプションで置換シンボルが定義されている場合、同名のシンボルに対する .DEFINE は無効となります。

例

```
SYM1: .DEFINE    "R1"
      MOV.W      SYM1,R0                ;MOV.W R1,R0 に置き換えられます。
```

先頭が a～f または A～F で始まる 16 進数は.DEFINE で同名のシンボルが定義された場合、置換対象になります。置換対象外にするには先頭に 0 を付加してください。

```
A0:   .DEFINE    "0"
      MOV.W      #H'A0,R0               ;MOV.W #H'0,R0 に置き換えられます。
      MOV.W      #H'0A0,R0              ;置き換えられません。
```

基数（B'、Q'、D'、H'）は.DEFINE で同名のシンボルが定義された場合、置換対象になります。B、Q、D、H、b、q、d、h 一文字のシンボルを定義するときは注意してください。

```
B:    .DEFINE    "H"
      MOV.W      #B'10,R0               ;MOV.H #H'10,R0 に置き換えられます。
```

.AIF, .AELIF, .AELSE, .AENDI

書　式　　△.AIF△<項 1>△<関係演算子>△<項 2>
　　<.AIF の条件成立時にアセンブルするソースステートメント>
[△.AELIF△<項 1>△<関係演算子>△<項 2>
　　<.AELIF の条件成立時にアセンブルするソースステートメント>]
[△.AELSE
　　<全ての条件不成立時にアセンブルするソースステートメント>]
△.AENDI

ラベルは記述できません。

説　明　　.AIF～.AELIF～.AELSE～.AENDI 間に記述したソースステートメントのうち、条件が成立した部分をアセンブルします。
.AELIF と .AELSE は省略できます。
また、.AELIF は.AIF と .AELSE の間ならば繰り返し指定できます。
各オペレーションで指定できるオペランドは次のようになります。

オペレーション	オペランド
.AIF	**比較型条件**
.AELIF	**比較型条件**
.AELSE	**記述不可能**
.AENDI	

<項 1>、<項 2>には値または文字列を記述します。ただし、値と文字列を比較すると常に条件不成立となります。
値は定数またはプリプロセッサ変数で指定します。
文字列は文字またはプリプロセッサ変数をダブルコーテーション（"）で囲んで指定します。
ダブルコーテーション（"）自体を文字として指定する場合はダブルコーテーションを 2 つ続けて記述（""）します。

関係演算子の条件は以下のとおりです。

EQ	**項 1 ＝ 項 2**
NE	**項 1 ≠ 項 2**
GT	**項 1 ＞ 項 2**
LT	**項 1 ＜ 項 2**
GE	**項 1 ≧ 項 2**
LE	**項 1 ≦ 項 2**

【注】文字列の比較は EQ、NE のみ有効です。

例

```
.AIF      ¥&TYPE EQ 1
MOV.W     R0,R3                ; TYPE が 1 の時、アセンブルします。
MOV.W     R1,R4
.AELIF    ¥&TYPE EQ 2
MOV.W     R0,R2                ; TYPE が 2 の時、アセンブルします。
MOV.W     R1,R3
.AELSE
MOV.W     R0,R4                ; TYPE が 1 でも 2 でもない時、
MOV.W     R1,R5                ; アセンブルします。
.AENDI
```

.AIFDEF, .AELSE, .AENDI

書　式

△.AIFDEF△<置換シンボル>
　<置換シンボルが定義されていた時にアセンブルするソースステートメント>
[△.AELSE
　<置換シンボルが定義されていない時にアセンブルするソースステートメント>]
△.AENDI

ラベルは記述できません。

説　明

.AIFDIF、.AELSE、.AENDI はアセンブルするか否かを置換シンボルの定義によって切り替えます。.AELSE は省略できます。
各オペレーションで指定できるオペランドは次のようになります。

オペレーション	オペランド
.AIFDEF	**定義型条件**
.AELSE	**記述不可能**
.AENDI	

置換シンボルは.DEFINE 制御文または、define オプションで定義します。
記述した置換シンボルがオプションで定義されている、または本制御文で参照するより前に定義している場合、条件成立となります。
また、記述した置換シンボルが本制御文で参照した後に定義している、または定義がない場合は、条件不成立となります。

例

```
        .AIFDEF   FLAG
        MOV.W     R0,R3              ;FLAG が.DEFINE 制御文で定義されて
        MOV.W     R1,R4              ;いるときアセンブルします。
        .AELSE
        MOV.W     R0,R2              ;FLAG が.DEFINE 制御文で定義されて
        MOV.W     R1,R3              ;いないときアセンブルします。
        .AENDI
```

.AREPEAT, .AENDR

書　式　　△.AREPEAT△<回数>
　　<繰り返し展開してアセンブルするソースステートメント>
△.AENDR

ラベルは記述できません。

説　明　　.AREPEAT、.AENDR は指定された回数だけ繰り返し展開してアセンブルする制御文です。
各オペレーションで指定できるオペランドは次のようになります。

オペレーション	オペランド
.AREPEAT	**繰り返し展開する回数**
.AENDR	**記述不可能**

.AREPEAT で指定された回数だけ .AREPEAT～.AENDR の間に記述したソースステートメントを繰り返し展開してアセンブルします（ソースステートメントを繰り返しコピーするのと同じで実行時のループにはなりません）。
回数は定数またはプリプロセッサ変数で指定します。
回数に 0 以下の値を指定した場合は展開しません。

例

```
        MOV.B       @SP,R0L
        .AREPEAT    3
        SHAL.B      R0L
        .AENDR
        MOV.B       R0L,@SP
```

[展開後]

```
        MOV.B       @SP,R0L
        SHAL.B      R0L
        SHAL.B      R0L
        SHAL.B      R0L
        MOV.B       R0L,@SP
```

.AWHILE, .AENDW

書　式　　△.AWHILE△<項 1>△<関係演算子>△<項 2>
　　<繰り返し展開してアセンブルするソースステートメント>
△.AENDW

ラベルは記述できません。

説　明　　.AWHILE、.AENDW は条件が成立している間だけ繰り返し展開します。
.AWHILE で指定した条件が成立している間、.AWHILE～.AENDW の間に記述したソースステートメントを繰り返し展開してアセンブルします（ソースステートメントを繰り返しコピーするのと同じで実行時のループにはなりません）。

<項 1>、<項 2>は値または文字列を記述します。ただし、値と文字列を比較すると常に条件不成立となります。
値は定数またはプリプロセッサ変数で指定します。
文字列は文字またはプリプロセッサ変数をダブルコーテーション（"）で囲んで指定します。
ダブルコーテーション（"）自体を文字として指定する場合は、ダブルコーテーションを 2 つ続けて記述（""）します。
条件つき繰り返し展開は最終的に条件を不成立にして展開を終了します。
条件が不成立にならない場合は 65,535 回または.ALIMIT 制御文で指定した展開回数を繰り返しますので条件の指定にはよく注意してください。
関係演算子の条件は以下のとおりです。

関係演算子	条件
EQ	項 1 ＝ 項 2
NE	項 1 ≠ 項 2
GT	項 1 ＞ 項 2
LT	項 1 ＜ 項 2
GE	項 1 ≧ 項 2
LE	項 1 ≦ 項 2

【注】文字列の比較は EQ、NE のみ有効です。

例

```
; COUNT≠0 の間だけ繰り返し展開します。
COUNT   .ASSIGNA   2                    ; COUNT に 2 を設定しています。
        .AWHILE    ¥&COUNT NE 0         ; 条件は COUNT≠0 です。
        ADD.B      R0L,R1L
        ADD.B      R0L,R2L
        INC.B      R0L
COUNT   .ASSIGNA   ¥&COUNT-1            ; COUNT から 1 を減算しています。
        .AENDW

; STOP≦10 の間だけ繰り返し展開します。
STOP    .ASSIGNA   0                    ; STOP に 0 を設定しています。
        .AWHILE    ¥&STOP LE 10         ; 条件は STOP≦10 です。
        ADD.B      R0L,R1L
        ADD.B      R0L,R2L
        INC.B      R0L
STOP    .ASSIGNA   ¥&STOP+3             ; STOP に 3 を加算しています。
        .AENDW
```

展開の中断終了

.EXITM

書　式　　△.EXITM
ラベルは記述できません。

説　明　　.EXITM は繰り返し展開（.AREPEAT～.AENDR）および条件つき繰り返し展開（.AWHILE～.AENDW）の展開を中断させます。
各展開では本制御文が出現した時点で展開を中断します。
本制御文はマクロ展開の中断終了にも使用します。マクロ命令と繰り返し展開を組み合わせて使用する場合は本制御文の位置に注意してください。

例

```
COUNT .ASSIGNA    0              ; COUNT に 0 を設定しています。
      .AWHILE     1 EQ 1         ; 無限展開（常に条件成立）を指定しています。
      ADD.W       R0,R1
      ADD.W       R2,R3

COUNT .ASSIGNA    ¥&COUNT+1      ; COUNT に 1 を加えます。
      .AIF        ¥&COUNT EQ 2   ; 条件は COUNT=2 です。
      .EXITM                     ; 条件成立で.AWHILE を中断終了します。
      .AENDI
      .AENDW
```

COUNT が更新され、.AIF の条件が成立すると.EXITM がアセンブルされます。
.EXITM がアセンブルされた時点で.AWHILE の展開を中断終了します。
展開結果は以下のようになります。

```
       ADD.W      R0,R1    … COUNT が 0 のとき
       ADD.W      R2,R3
       ADD.W      R0,R1    … COUNT が 1 のとき
       ADD.W      R2,R3
```

この後、COUNT は 2 となり、展開は中断終了します。

プリプロセッサ展開時のエラー処理

.AERROR

書　式　　△.AERROR
ラベルは記述できません。

説　明　　.AERROR をアセンブルするとエラー670 を発生し、アセンブラをエラー終了します。
.AERROR はプリプロセッサ変数の値のチェック等に使用することができます。

例

```
～
.AIF  ¥&FLAG  EQ 1
ADD.W       R0,R1
INC.W       R0
.AELSE
.AERROR                    ;¥&FLAG が 1 以外の場合エラーとします。
.AENDI
～
```

展開の上限値設定

.ALIMIT

書　式　　△.ALIMIT△<回数>
ラベルは記述できません。

説　明　　条件つき繰り返し展開（.AWHILE～.AENDW）で、ステートメントの展開回数の上限値を設定します。
<回数>の値は次の形式で指定します。
・定数（整数定数、文字定数）
・既に定義したプリプロセッサ変数
・上記を項とする式

.ALIMIT で指定した上限値を越えるとウォーニング 854 となり、展開を打ち切ります。
展開回数の限界値は.ALIMIT を指定しないとき、65,535 です。
繰り返し展開回数の上限値は、本制御命令で再指定することで値を変更できます。
上限値の再指定は、本制御命令以降のソースステートメントに対して有効です。

例

```
COUNT .ASSIGNA    3              ; COUNT に 3 を設定しています。
      .ALIMIT     10             ; 繰り返しの上限値に、10 を指定しています。
      .AWHILE     ¥&COUNT NE 4
      ADD.W       R0,R1          ; [1]
      ADD.W       R0,R1          ; [1]
      INC.W       R0             ; [1]
COUNT .ASSIGNA    ¥&COUNT-1      ; [1]
      .AENDW
```

COUNT≠4 の間だけ[1]を展開します。
10 回展開した後、ウォーニング 854 を出力し、繰り返し展開を中断終了します。

11.6 マクロ機能

11.6.1 マクロ機能の概要

本アセンブリ言語ではプログラム中でよく使用する一連の処理に名前をつけ、1つの命令（マクロ命令）として定義することができます。このような定義をマクロ定義といいます。マクロ定義の方法は次のとおりです。

```
          ～
.MACRO    マクロ名
   マクロ本体
.ENDM
          ～
```

マクロ名はマクロ命令につける名前、マクロ本体はマクロ命令の内容です。

定義したマクロ命令を呼び出して使用することをマクロコールといいます。マクロコールの方法は次のとおりです。

```
          ～
定義済みのマクロ名
          ～
```

マクロ定義とマクロコールの例を以下に示します。

例：

```
～
.MACRO  SUM     ; R1,R2,R3 の合計を求める処理を
ADD.W   R2,R1   ; マクロ命令 SUM として定義します。
ADD.W   R3,R1
.ENDM
～

SUM             ; マクロ命令 SUM を呼び出します。
                ; マクロ本体
                ;   ADD.W   R2,R1
                ;   ADD.W   R3,R1
                ; が展開されます。
```

マクロ命令は、パラメータを使用することで、マクロ本体の一部変更して展開することも可能です。

手順は次のとおりです。

(1) マクロ定義
　　.MACRO文で仮引数を定義(マクロ名につづいて記述)します。
　　マクロ本体の記述に仮引数を使います(仮引数の先頭にバックスラッシュ(円記号 "¥")を付けます)。

(2) マクロコール
　　マクロパラメータを付けてマクロ命令を呼出します。

マクロ命令展開の際、仮引数は対応するマクロパラメータに置き換えられます。

例：

```
～
.MACRO    SUM ARG1      ; 仮引数 ARG1 を定義します。
MOV.W     R1,¥ARG1      ; ARG1 を使ってマクロ本体を記述しています。
ADD.W     R2,¥ARG1
ADD.W     R3,¥ARG1
.ENDM
～
SUM       R0            ;  マクロパラメータ R0 を付けてマクロ SUM を呼び出します。
                        ;  マクロ本体中の仮引数がマクロパラメータで置き換えられ、
                        ;    ADD.W    R1,R0
                        ;    ADD.W    R2,R0
                        ;    ADD.W    R3,R0
                        ;  が展開されます。
```

11.6.2 マクロ機能に関する制御文

表 11.16 にマクロ機能の制御文の一覧を示します。

表 11.16 マクロ機能の制御文一覧

ニーモニック	機能
.MACRO	**マクロ命令を定義します。**
.ENDM	
.EXITM	**マクロ命令の展開を中断します。** 11.5.2 .EXITM **を参照してください。**

.MACRO, .ENDM

書　式　　△.MACRO△<マクロ名>[△<仮引数>[,...]]
△.ENDM
<仮引数>：<仮引数名>[=<仮引数のデフォルト>]

ラベルは記述できません。

説　明　　.MACRO、.ENDM はマクロを定義します。
.MACRO～.ENDM 間のソースステートメント（マクロ本体）をマクロ命令として名前を付けることをマクロ命令を定義する（マクロ定義）といいます。

各オペレーションで指定できるオペランドは次のようになります。

オペレーション	オペランド
.MACRO	**・マクロ命令** **・仮引数** **デフォルトを記述(省略可)**
.ENDM	**記述不可能**

(1) マクロ名
マクロ名はマクロ命令に付ける名前です。
マクロ命令を展開する際にマクロ本体の一部を置換して展開したい場合に指定します。
仮引数はマクロ展開を行なう（マクロコール）時に指定された文字列（マクロパラメータ）に置換されます。

マクロ本体では置換したい部分に仮引数名を記述します。マクロ本体での仮引数の参照方法は次のとおりです。
¥仮引数名[']
アポストロフィー(')は、仮引数名とソースステートメントの区別を明確にしたい場合に記述します。

(2) 仮引数
仮引数には仮引数のデフォルトを設定できます。仮引数のデフォルトにはマクロコール時にマクロパラメータを省略した場合に置換する文字列を指定します。
仮引数の書き方はシンボル名の書き方と同じです。
仮引数名の最大文字数は 32 文字で英大文字と英小文字を区別します。

(3) 仮引数のデフォルト
仮引数のデフォルトに次の文字を含む場合は文字列をダブルコーテーション（"）またはアングルブラケット（<>）で囲んでください。
・スペース
・タブ
・カンマ（,）
・セミコロン（;）
・ダブルコーテーション（"）
・アングルブラケット（<>）

マクロ展開では文字列を囲んだダブルコーテーション（"）やアングルブラケット（<>）は取り除いて置換します。

(4) 制限

マクロ命令は次の場所では定義できません。

・マクロ本体（.MACRO～.ENDM）

・.AREPEAT～.AENDR の間

・.AWHILE～.AENDW の間

マクロ本体には.END を記述できません。

.ENDM のラベルにはシンボルを記述できません。

.ENDM のラベルにシンボルを記述した場合は.ENDM を無視します。この場合、エラーは表示しません。

例

```
; R3,R4,R5 の合計を求める処理をマクロ命令 SUM として定義します。
        ～
        .MACRO    SUM
        MOV.W     R3,R1
        ADD.W     R4,R1
        ADD.W     R5,R1
        .ENDM
        ～
        SUM                         ; マクロ命令 SUM を呼び出します。
                                    ; マクロ本体
                                    ;    MOV.W   R3,R1
                                    ;    ADD.W   R4,R1
                                    ;    ADD.W   R5,R1
                                    ; が展開されます。


; 仮引数 P1,P2,P3 の加算結果を R0 に出力する処理を、マクロ命令 TOTAL として
; 定義します。
        ～
        .MACRO    TOTAL P1,P2,P3
        MOV.W     ¥P1,R0
        ADD.W     ¥P2,R0
        ADD.W     ¥P3,R0
        .ENDM
        ～
        TOTAL     R1,R2,R3          ; マクロ命令 TOTAL を呼び出します。
                                    ; マクロ本体
                                    ;    MOV.W   R1,R0
                                    ;    ADD.W   R2,R0
                                    ;    ADD.W   R3,R0
                                    ; が展開されます。
```

11.6.3 マクロ本体

.MACRO と.ENDM の間に記述した一連のソースステートメントをマクロ本体と呼びます。マクロ本体はマクロコール（マクロ命令を呼び出すこと）により、展開してアセンブルされます。本節ではマクロ本体が持つ機能と記述方法を説明します。

(1) 仮引数の参照

マクロ展開でマクロパラメータと置換したい部分に仮引数を記述します。仮引数の参照方法は以下のとおりです。

¥仮引数［'］

アポストロフィ（'）は仮引数名とソースステートメントの区別を明確にしたい場合に記述します。

例：

```
            .MACRO    PLUS1 P,P1          ; P,P1 は仮引数です。
            ADD.W     #1,¥P1              ; 仮引数 P1 を参照しています。
            .SDATA    "¥P'1"              ; 仮引数 P を参照しています。
            .ENDM
            PLUS1     R,R1                ; PLUS1 を展開します。
[展開結果]
            ADD.W     #1,R1               ; 仮引数 P1 を参照しています。
            .SDATA    "R1"                ; 仮引数 P を参照しています。
```

(2) プリプロセッサ変数 (.ASSIGNA, .ASSIGNC)の参照

マクロ本体ではプリプロセッサ変数を参照できます。プリプロセッサ変数の参照方法は次のとおりです。

¥&プリプロセッサ変数名［'］

アポストロフィ（'）はプリプロセッサ変数とソースステートメントの区別を明確にしたい場合に記述します。

例：

```
            .MACRO    PLUS1
            ADD.W     #1,R¥&V1            ; プリプロセッサ変数 V1 を参照しています。
            .SDATA    "¥&V'1"             ; プリプロセッサ変数 V を参照しています。
            .ENDM
      V:    .ASSIGNC  "R"                 ; プリプロセッサ変数 V を定義しています。
      V1:   .ASSIGNA  1                   ; プリプロセッサ変数 V1 を定義しています。
            PLUS1                         ; PLUS1 を展開します。
[展開結果]
            ADD.W     #1,R1               ; プリプロセッサ変数 V1 を参照しています。
            .SDATA    "R1"                ; プリプロセッサ変数 V を参照しています。
```

(3) マクロ生成番号

マクロ本体にラベルがある場合などは、複数回マクロコールをするとシンボル名が重複してしまいます。このような事態を回避するためにはマクロ生成番号を使用してください。マクロ生成番号はマクロ展開で固有の 5 桁の 10 進数（00000～99999）を展開します。マクロ生成番号は次のように記述してください。

```
¥@
```

シンボル名の一部としてマクロ生成番号を記述しておくと、マクロコールのたびに固有のシンボル名となり、重複を避けることができます。１つのマクロ本体に２つ以上のマクロ生成番号を記述できますが、１回のマクロコールでは同じマクロ生成番号が展開されます。また、マクロ生成番号は数字に展開されるのでシンボル名の先頭には記述しないでください。

例：

```
            .MACRO MCO Rn
            MOV.W  ¥Rn,¥Rn
            BEQ    LAB¥@:8
            MOV.W  #H'0,¥Rn
   LAB¥@:
            INC.W  ¥Rn
            .ENDM

            MCO    R1                              ; MCO を展開するたびに異なる
   ;
            MCO    R2                              ; シンボルを生成します。

   [展開結果]
            MOV.W  R1,R1
            BEQ    LAB00000:8
            MOV.W  #H'0,R1
   LAB00000:
            INC.W  R1
   ;
            MOV.W  R2,R2
            BEQ    LAB00001:8
            MOV.W  #H'0,R2
   LAB00001:
            INC.W  R2
```

(4) マクロ処理除外

マクロ本体内にバックスラッシュ（円記号 "¥"）があるとマクロ置換処理の対象になります。したがって、バックスラッシュを ASCII 文字として記述したい場合はマクロ置換処理から除外する必要があります。マクロ処理除外の書き方は次のとおりです。

¥(マクロ処理除外文字列)

マクロ展開ではバックスラッシュ（¥）とカッコは取り除きます。

例：

```
.MACRO BACK_SLASH_SET
¥(MOV.W  #"¥",R0)                    ; ¥は ASCII 文字として展開されます。
.ENDM

BACK_SLASH_SET
```

[展開結果]

```
MOV.W    #"¥",R0                     ; ¥は ASCII 文字として展開されます。
```

(5) マクロ内コメント

マクロ本体のコメントをマクロ展開では展開したくない場合に、マクロ内コメントを記述します。マクロ内コメントの書き方は次のとおりです。

¥;コメント

例：

```
.MACRO COMMENT_IGNORE  Rn
MOV.W  ¥Rn,@-SP                      ¥; ¥Rn のデータを退避します。
.ENDM

COMMENT_IGNORE R1
```

[展開結果]

```
MOV.W  R1,@-SP
```

(6) 文字列操作関数

マクロ本体には文字列操作関数を記述できます。文字列操作関数には次のものがあります。

.LEN 関数　　：　文字列の長さ
.INSTR 関数　：　文字列の検索
.SUBSTR 関数　：　文字列の切り出し

11.6.4 マクロコール

マクロ定義により定義されたマクロ命令を展開することをマクロコールといいます。
マクロコールは以下の書式で記述します。

[<シンボル>[:]]△<マクロ名> [△<マクロパラメータ>[,…]]
<マクロパラメータ>[=<仮引数名>]=<文字列>

マクロ名は、マクロコールする以前にマクロ定義(.MACRO)します。
マクロパラメータには、マクロ展開で置換する文字列を指定します。
この場合、マクロ名に対応するマクロ定義(.MACRO)で、仮引数を宣言しておく必要があります。

(1) マクロパラメータの指定方法

マクロパラメータの指定方法には、位置指定とキーワード指定があります。

(2) 位置指定

マクロ定義(.MACRO)で宣言した仮引数の並び順と、マクロパラメータの並び順を一致させて指定する方法です。

(3) キーワード指定

マクロ定義(.MACRO)で宣言した仮引数の仮引数名にイコール(=)で区切って指定する方法です。

(4) マクロパラメータの書き方

マクロパラメータに次の文字を含む場合は、文字列をダブルコーテーション(")または、アングルブラケット(<>)で囲んでください。

・スペース
・タブ
・カンマ(,)
・セミコロン(;)
・ダブルコーテーション(")
・アングルブラケット(<>)

マクロ展開では、文字列を囲んだダブルコーテーションや、アングルブラケットは取り除いて置換します。

例：

```
          .MACRO    SUM    FROM=0,TO=6          ; マクロ命令 SUM、仮引数 FROM,TO を
                                                ; 定義します。
          MOV.W     R¥FROM,R0
  COUNT   .ASSIGNA  ¥FROM+1
          .AWHILE   ¥&COUNT LE ¥TO
          AND.W     R¥&COUNT,R0                 ; 仮引数を用いてマクロ本体を記述して
  COUNT   .ASSIGNA  ¥&COUNT+1                   ; います。
          .AENDW
          .ENDW

          SUM       0,3                         ; どちらも同じ結果になります。
          SUM       TO=3                        ;
```

マクロ本体中の仮引数がマクロパラメータで置き換えられ、展開結果は次のようになります。

[展開結果]

```
        MOV.W   R0,R0
        AND.W   R1,R0
        AND.W   R2,R0
        AND.W   R3,R0
```

11.6.5 文字列操作関数

表 11.17 にマクロ本体で使用できる文字列操作関数の一覧を示します。

表 11.17 **文字列操作関数一覧**

.LEN	**文字列の長さを返します。**
.INSTR	**文字列の検索を行ないます。**
.SUBSTR	**文字列の切り出しを行ないます。**

文字列の長さ

.LEN

書　式　　.LEN[△]("<文字列>")

説　明　　.LEN は文字列の長さを数え、基数を省略した 10 進数に置換します。
文字列は文字をダブルコーテーション（"）で囲んで指定します。
ダブルコーテーション自体を文字として指定する場合は 2 つ続けて記述します。
文字列にはマクロの仮引数、プリプロセッサ変数を指定できます。
.LEN("¥仮引数名")
.LEN("¥&プリプロセッサ変数名")
本関数を記述できるのはマクロ本体（.MACRO～.ENDM）だけです。

例

```
        .MACRO    RESERVE_LENGTH P1
        .SRES     .LEN("¥P1")
        .ENDM

       RESERVE_LENGTH  ABCDEF
       RESERVE_LENGTH  ABC
```

[展開結果]

```
        .SRES     6                   ; "ABCDEF"の字数は 6 です。
        .SRES     3                   ; "ABC"の字数は 3 です。
```

.INSTR

書　式　　.INSTR[△]("<文字列１>","<文字列２>"[,<検索開始位置>])

説　明　　.INSTR は文字列１に文字列２が含まれているかを検索し、文字列の先頭を０とした検索位置を基数を省略した 10 進数に置換します。
文字列１に文字列２が含まれていない場合は-1 に置換します。

文字列は文字をダブルコーテーション（"）で囲んで指定します。
ダブルコーテーション（"）自体を文字として指定する場合は 2 つ続けて記述します。

検索開始位置は文字列１の先頭を 0 とした数値で指定します。
省略した場合は 0 を設定します。

文字列、検索開始位置にはマクロの仮引数、プリプロセッサ変数を指定できます。
```
  .INSTR("¥仮引数名", .... )
  .INSTR("¥&プリプロセッサ変数名", .... )
```
本関数を記述できるのはマクロ本体（.MACRO～.ENDM）だけです。

例
```
        .MACRO    FIND_STR   P1
        .DATA.W   .INSTR("ABCDEFG","¥P1",0)
        .ENDM

       FIND_STR  CDE
       FIND_STR  H
```

[展開結果]
```
       .DATA.W   2                    ; "ABCDEFG"の 2 文字目（先頭を 0 と
                                      ; します）に"CDE"があります。
       .DATA.W   -1                   ; "ABCDEFG"の中に"H"はありません。
```

文字列の切り出し

.SUBSTR

書　式　　.SUBSTR[△]("<文字列>",<切り出しの開始位置>,<切り出しの長さ>)

説　明　　.SUBSTR は文字列の先頭を 0 とした切り出しの開始位置から、切り出しの長さ分の文字列を切り出し、ダブルコーテーション(")で囲んだ文字列に置換します。
文字列は文字をダブルコーテーションで囲んで指定します。
ダブルコーテーション自体を文字として指定する場合は 2 つ続けて記述します。

切り出しの開始位置は 0 以上が指定できます。
切り出しの開始位置、切り出しの長さが不適当な場合には空文字("")に置換します。
また、切り出しの長さは 1 以上が指定できます。

切り出しの開始位置、切り出しの長さにはマクロの仮引数、プリプロセッサ変数を指定できます。

```
  .SUBSTR("¥仮引数名", .... )
  .SUBSTR("¥&プリプロセッサ変数名", .... )
```

本関数を記述できるのはマクロ本体（.MACRO～.ENDM）だけです。

例

```
        .MACRO    RESERVE_STR  P1=0,P2
        .SDATA    .SUBSTR("ABCDEFG",¥P1,¥P2)
        .ENDM

       RESERVE_STR     2,2
       RESERVE_STR     ,3            ; マクロパラメータ P1 を省略しています。
```

[展開結果]

```
       .SDATA    "CD"
       .SDATA    "ABC"
```

11.7 構造化アセンブリ機能

構造化アセンブリ機能は、処理を選択したり、繰り返したりする命令を展開する機能です。

表 11.18 に、後述の構造化アセンブリ制御文で使用するコンディションコードの条件を示しています。各構造化アセンブリ制御文の説明とあわせて参照してください。

表 11.18 コンディションコード一覧

コンディションコード	比較実行型の時の条件	コンディションコード指定型の時の条件
< EQ >	**項**1 ＝ **項**2	Z = 1
< NE >	**項**1 ≠ **項**2	Z = 0
< GT >	**項**1 ＞ **項**2 (**符号付き比較**)	Z v (N ⊕ V) = 0
< LT >	**項**1 ＜ **項**2 (**符号付き比較**)	N ⊕ V = 1
< GE >	**項**1 ≧ **項**2 (**符号付き比較**)	N ⊕ V = 0
< LE >	**項**1 ≦ **項**2 (**符号付き比較**)	Z v (N ⊕ V) = 1
< HI >	**項**1 ＞ **項**2 (**符号なし比較**)	C v Z = 0
< LO > < CS >	**項**1 ＜ **項**2 (**符号なし比較**)	C = 1
< HS > < CC >	**項**1 ≧ **項**2 (**符号なし比較**)	C = 0
< LS >	**項**1 ≦ **項**2 (**符号なし比較**)	C v Z = 1
< VC >		V = 0
< VS >		V = 1
< PL >		N = 0
< MI >		N = 1
< T >		**常に条件成立**
< F >		**常に条件不成立**

【注】 N …… CCR(**コンディションコードレジスタ**)**の** N(**ネガティブ**)**フラグ**
Z …… CCR **の** Z(**ゼロ**)**フラグ**
V …… CCR **の** V(**オーバフロー**)**フラグ**
C …… CCR **の** C(**キャリ**)**フラグ**
v …… **論理和**
⊕ …… **排他的論理和**

11.7.1 構造化アセンブリ機能に関する注意事項

構造化アセンブリ機能は、各制御文に対して一定の形式の命令、シンボルを展開する機能であり、最適化を行うものではありません。
したがって、各制御文で指定できる内容は、展開する命令の仕様により限定されます。
また、効率の悪い命令や不必要なシンボルを展開する場合があります。

(1) 命令の展開

コンディションコード条件で判定を行う時、展開する命令の仕様によって記述方法が限定されます。
例：

```
.IF.B (R0L<LT>#10)                ;展開した命令がエラーとなります。
      MOV.W    R1,R2
.ENDI
```

.IF 制御命令は、CMP 命令に展開されます。
しかしこの記述では、CMP R0L,#10 と展開されますので、エラーとなります。
これを回避するためには、以下のように記述してください。

```
.IF.B (#10<LT>R0L)                ; CMP #10,R0L と展開されます。
      MOV.W    R1,R2
.ENDI
```

(2) シンボルの展開

各制御文では、以下に示す形式のシンボルを展開します。

```
.IF ･･･････ _$I00000 ～ _$I99999
.SWITCH ････ _$S00000 ～ _$S99999
.FOR[U] ････ _$F00000 ～ _$F99999
.WHILE ･････ _$W00000 ～ _$W99999
.REPEAT ････ _$R00000 ～ _$R99999
```

したがって、同じ名称のシンボルは使用できません。

11.7.2 構造化アセンブリ機能に関する制御文

表 11.19 に構造化アセンブリ機能の制御文の一覧を示します。

表 11.19 **構造化アセンブリ機能の制御文一覧**

.IF	**処理の選択**
.SWITCH	**条件に従って命令を選択実行します**
.FOR	**処理の繰り返し**
.WHILE	**条件が成立している間、繰り返し実行します**
.REPEAT	
.BREAK	**処理の繰り返しの中断、処理は終了します**
.CONTINUE	**処理の繰り返しの中断、処理は継続します**

.IF

書　式　　△.IF[.<サイズ>][:<分岐サイズ>]△<条件>
[△.ELSE [:<分岐サイズ>]]
△.ENDI

```
<サイズ>     = { B | W | L }
<分岐サイズ> = { 8 | 16 }
<条件>       = { 項1<CC>項2 | <CC> }
<CC>         = { EQ | NE | GT | LT | GE | LE | HI | LO | HS |
                 LS | CC | CS | VC | VS | PL | MI | T  | F  }
```

ラベルは記述できません。

説　明　　.IF で条件判定した結果に従って、ソースステートメントを選択実行します。
条件が成立した場合は.IF~.ELSE 間のソースステートメントを、条件が成立しない場合は.ELSE~.ENDI 間のソースステートメントを実行します。
.ELSE の文は、省略することができます。この場合は、条件が成立した場合に.IF~.ENDI 間のソースステートメントを実行します。

サイズ、分岐サイズは以下のようになります。

(1)　サイズ

サイズには、比較実行型で比較する項のサイズを指定します。
オペレーションサイズを省略すると、.IF.B (バイトサイズ)となります。
本指定は、コンディションコード指定型では意味を持ちません。
サイズは以下のようになります。

サイズ		サイズ
B	**バイト**	(1 **バイト**)
W	**ワード**	(2 **バイト**)
L	**ロングワード**	(4 **バイト**)

(2)　分岐サイズ

分岐サイズには、.IF と.ELSE の分岐サイズがあります。
.IF の分岐サイズには、.IF から.ELSE まで、または.IF から.ENDI まで(.ELSE 省略時)の分岐サイズを指定します。
.ELSE の分岐サイズには、.ELSE から.ENDI までの分岐サイズを指定します。
分岐サイズは次のようになります。

オペレーション	分岐サイズ
8	8 **ビット**
16	16 **ビット**

指定を省略した場合の設定は、
11.3 .DISPSIZE **の** FBR、3.2.2 br_relative、3.2.2 [no]optimize を参照してください。

コンディションコードの条件については、**表** 11.18 **コンディションコード一覧**を参照してください。

また、条件判定には、次の 2 つがあります。

(1)　比較実行型

比較実行型は、2 つの項をコンディションコードの条件で判定します。
項には、CMP 命令に指定できるアドレス形式を指定します。

(2) コンディションコード指定型
コンディションコード指定型は、CCR(コンディションコードレジスタ)の状態で条件判定します。

制限事項
(1) H8/300, H8/300L では、サイズに L を指定できません。
(2) H8/300, H8/300L では、分岐サイズに 16 を指定できません。
(3) .IF~.ELSE 間、.ELSE~.ENDI 間、.IF~.ENDI 間(.ELSE 省略時)のソースステートメントのサイズは、指定した分岐サイズを超えることはできません。
分岐サイズに対するソースステートメントの最大サイズは、次の通りです。
8 ‥‥ 100 バイト程度
16 ‥‥ 32,700 バイト程度
(4) 本制御命令を使用すると、_$I00000～_$I99999 のシンボルを展開します。
従って、同じ名称のシンボルは使用することができません。

例
(1)
```
        .IF.W (R0<EQ>R1)
               ADD.W     #1,R0          ; [1]
               MOV.W     R0,R2          ; [1]
        .ELSE
               ADD.W     #1,R1          ; [2]
               MOV.W     R1,R2          ; [2]
        .ENDI
```
比較実行型の例です。
R0＝R1 の場合は[1]を、R0≠R1 の場合は[2]を実行します。

(2)
```
        .IF.B (#H'10<LT>R0L)
               SUB.W     R1,R1          ; [3]
               MOV.W     R1,R2          ; [3]
        .ENDI
```
比較実行型の例です。
H'10＜R0L(符号付き比較)の場合に[3]を実行します。

(3)
```
        .IF (<NE>)
               ADD.B     #5:8,R0L       ; [4]
        .ELSE
               MOV.B     R0L,R1L        ; [5]
        .ENDI
```
コンディションコード指定型の例です。
CCR(コンディションコードレジスタ)の Z(ゼロ)フラグが 0 の時は[4]を、1 の場合は[5]を実行します。

(4)
```
        .IF.B (#0<LE>R0L)
           .IF.B (#50<GE>R0L)
               MOV.W     R2,R1          ; [6]
               ADD.W     R3,R1          ; [6]
           .ENDI
        .ENDI
```
.IF を組み合わせた例です。
0≦R0L≦50(符号付き比較)の場合に、[6]を実行します。

分岐命令

.SWITCH

書　式　　△.SWITCH[.<サイズ>]△<条件 1>
(△.CASE [:<分岐サイズ>]△<条件 2>
[△.BREAK [:<分岐サイズ>]])[,…]
[△.OTHERS]
△.ENDS

```
<サイズ>     = { B | W | L }
<分岐サイズ> = { 8 | 16 }
<条件 1>     = { <レジスタ> | <CC> }
<条件 2>     = { <項> | <CC> }
<CC>         = { EQ | NE | GT | LT | GE | LE | HI | LO | HS |
                 LS | CC | CS | VC | VS | PL | MI | T  | F  }
```

ラベルは記述できません。

説　明　　.SWITCH と .CASE で条件判定した結果に従って、ソースステートメントを選択実行します。
.SWITCH と .CASE で条件が成立した場合は、該当する .CASE~.BREAK 間のソースステートメントを実行します。
.SWITCH と .CASE の条件判定は、上から順番に判定します。
.BREAK を省略した場合には、直後の .CASE~.BREAK 間、 .OTHERS~.ENDS 間のソースステートメントまでを実行します。

サイズ、分岐サイズは以下のようになります。

(1)　サイズ

サイズには、比較実行型で比較するレジスタと項のサイズを指定します。
オペレーションサイズを省略すると、.SWITCH.B (バイトサイズ)となります。
本指定は、コンディションコード指定型では意味を持ちません。
サイズは以下のようになります。

サイズ	サイズ	
B	**バイト**	(1 **バイト**)
W	**ワード**	(2 **バイト**)
L	**ロングワード**	(4 **バイト**)

(2)　分岐サイズ

分岐サイズには、 .CASE と .BREAK の分岐サイズがあります。
.CASE の分岐サイズには、 .CASE から次に現れる .CASE、 .OTHERS、 .ENDS までの分岐サイズを指定します。
.BREAK の分岐サイズには、 .BREAK から .ENDS までの分岐サイズを指定します。
分岐サイズは次のようになります。

オペレーション	分岐サイズ
8	8 **ビット**
16	16 **ビット**

指定を省略した場合の設定は、
11.3 .DISPSIZE **の** FBR、3.2.2 br_relative、3.2.2 [no]optimize を参照してください。

コンディションコードの条件については、**表** 11.18 **コンディションコード一覧**を参照してください。

また、条件判定には、次のようなものがあります。

(1) 比較実行型

比較実行型は、レジスタと項が等しいかを判定します。

.SWITCH には、レジスタを指定します。

.CASE には、CMP 命令のソースオペランドに指定できるアドレス形式を指定します。

(2) コンディションコード指定型

コンディションコード指定型は、CCR(コンディションコードレジスタ)の状態で条件判定します。

.SWITCH には、CCR を指定します。

.CASE には、コンディションコードを指定します。

制限事項

(1) H8/300, H8/300L では、サイズに L を指定できません。

(2) H8/300, H8/300L では、分岐サイズに 16 を指定できません。

(3) 各.CASE に対応するソースステートメント、各.BREAK~.ENDS 間のソースステートメントのサイズは、指定した分岐サイズを超えることはできません。

分岐サイズに対するソースステートメントの最大サイズは、次の通りです。

8 ‥‥ 100 バイト程度

16 ‥‥ 32,700 バイト程度

(4) 本制御命令を使用すると、_$S00000～_$S99999 のシンボルを展開します。

従って、同じ名称のシンボルは使用することができません。

例

(1)

```
        .SWITCH.B  (R0L)
        .CASE   #0
                MOV.W     R1,R4          ; [1]
        .BREAK
        .CASE   #1
                MOV.W     R2,R4          ; [2]
        .BREAK
        .OTHERS
                MOV.W     R3,R4          ; [3]
        .ENDS
```

比較実行型の例です。

R0L=0 の場合は[1]を、R0L=1 の場合は[2]を、それ以外の場合は[3]を実行します。

(2)

```
        .SWITCH.B  (CCR)
        .CASE   <CS>
                MOV.W     R0,R3          ; [4]
        .BREAK
        .CASE   <MI>
                MOV.W     R1,R3          ; [5]
        .ENDS
```

コンディションコード指定型の例です。

CCR(コンディションコードレジスタ)の C(キャリ)フラグが 1 の場合は「4」を、N(ネガティブ)フラグが 1 の場合は「5」を実行します。

(3)

```
        .SWITCH.B  (R0L)
        .CASE   #0
        .CASE   #1
        .CASE   #2
                MOV.W     R1,R3          ; [6]
        .BREAK
        .CASE   #3
                MOV.W     R2,R3          ; [7]
        .ENDS
```

.CASE に対する.BREAK を省略した例です。
R0L=0、R0L=1、R0L=2 の場合は[6]を、R0L=3 の場合は[7]を実行します。

.FOR[U]

書　式　　△.FOR[U][.<サイズ>][:<分岐サイズ>]△<条件>
△.ENDF
<サイズ>　　= { B | W | L }
<分岐サイズ> = { 8 | 16 }
<条件>　　= { <ループカウンタ>=<初期値>, <終値> [,[{+ | -}]<増分値>]}
ラベルは記述できません。

説　明　　.FOR[U]のループカウンタと終値で条件判定を行い、条件が成立している間だけ.FOR[U]~.ENDF 間のソーステートメントを繰り返し実行します。
本制御命令には、符号付き範囲で繰り返しの条件を判定する.FOR と、符号なし範囲で繰り返しの条件を判定する.FORU があります。

サイズ、分岐サイズは以下のようになります。

(1)　サイズ

サイズには、比較実行型で比較するレジスタと項のサイズを指定します。
オペレーションサイズを省略すると、.FOR[U].B (バイトサイズ)となります。
サイズは以下のようになります。

サイズ		サイズ
B	**バイト**	(1 **バイト**)
W	**ワード**	(2 **バイト**)
L	**ロングワード**	(4 **バイト**)

(2)　分岐サイズ

分岐サイズには、.FOR[U]から.ENDF までの分岐サイズを指定します。
分岐サイズは次のようになります。

オペレーション	分岐サイズ
8	8 **ビット**
16	16 **ビット**

指定を省略した場合の設定は、
11.3 .DISPSIZE **の** FBR、3.2.2 br_relative、3.2.2 [no]optimize を参照してください。

条件は次のようになります。

(1)　ループカウンタ=初期値

ループカウンタの初期値を指定します。
ループカウンタには、レジスタを指定します。
初期値には、MOV 命令のソースオペランドに指定できるアドレス形式を指定します。

(2)　終値

ループカウンタと比較する終値を指定します。
処理の繰り返しの条件は、以下の通りです。
　増分方向が増加方向　・・・・・・　ループカウンタ≦終値
　増分方向が減少方向　・・・・・・　ループカウンタ≧終値
終値には、CMP 命令のソースオペランドに指定できるアドレス形式を指定します。

(3) 増分値

1 回のループごとにループカウンタを加算、または減算する増分値を指定します。
増分方向には、増加方向の場合にはプラス(+)、減少方向の場合にはマイナス(-)を指定します。
指定を省略した場合には、プラス(+)を指定します。
増分値には、ADD、SUB 命令のソースオペランドに指定できるアドレス形式を指定します。指定を省略した場合、+#1 を指定します。

以下にループカウンタの数値範囲を示します。無限ループとなる可能性がありますので、数値範囲には十分注意してください。

制御文	増分方向	サイズ	ループカウンタの数値範囲（初期値〜終値）		
.FOR	+	B	-128	〜	126
		W	-32,768	〜	32,766
		L	-2,147,483,647	〜	2,147,483,646
	-	B	127	〜	-127
		W	32,767	〜	-32,767
		L	2,147,483,647	〜	-2,147,483,647
.FORU	+	B	0	〜	254
		W	0	〜	65,534
		L	0	〜	4,294,967,294
	-	B	255	〜	1
		W	65,535	〜	1
		L	4,294,967,295	〜	1

制限事項

(1) H8/300, H8/300L では、サイズに L を指定できません。
(2) H8/300, H8/300L では、分岐サイズに 16 を指定できません。
(3) .FOR[U]~.ENDF 間のソースステートメントのサイズは、指定した分岐サイズを超えることはできません。
分岐サイズに対するソースステートメントの最大サイズは次の通りです。
8 ‥‥ 100 バイト程度
16 ‥‥ 32,700 バイト程度
(4) 本制御命令を使用すると、_$F00000〜_$F99999 のシンボルを展開します。
従って、同じ名称のシンボルは使用することができません。

例

(1)

```
        .FOR.B (R0L=#1,#10,+#1)         ; [1]
               ADD.B     R0L,R1L
        .ENDF
```

.FOR の例です。
ループカウンタは R0L、初期値は#1、終値は#10、増分値は+#1 です。
R0L≦10(符号付き比較)の間だけ[1]を繰り返し実行します。

(2)

```
        .FOR.W (R0=R1,R2,-R3)
               ADD.B     #1:8,R5L        ; [2]
        .ENDF
```

.FOR の例です。
ループカウンタは R0、初期値は R1、終値は R2、増分値は-R3 です。
R0≧R2(符号付き比較)の間だけ[2]を繰り返します。

(3)

```
        .FORU.B(R0L=#1,#200,+#1)
                ADD.W     R1,R2           ; [3]
                ADD.W     R3,R4           ; [3]
        .ENDF
```

.FORU の例です。
ループカウンタは R0L、初期値は#1、終値は#200、増分値は+#1 です。
R0L≦200(符号なし比較)の間だけ[3]を繰り返し実行します。

(4)

```
        .FORU.L(ER0=#H'00000100,#H'000001FC,+#4)
                MOV.L     @ER0,ER2                      ; [4]
                MOV.L     ER2,@(H'00001100:32,ER1)      ; [4]
                ADDS.L    #4,ER1                        ; [4]
        .ENDF
```

.FORU の例です。
ループカウンタは ER0、初期値は#H'00000100、終値は＃H'000001FC、増分値は+#4 です。
ER0≦H'000001FC(符号なし比較)のときだけ[4]を繰り返し実行します。

ループ命令

.WHILE

書　式　　△.WHILE[.<サイズ>][:<分岐サイズ>]△<条件>
△.ENDW

```
<サイズ>     = { B | W | L }
<分岐サイズ> = { 8 | 16 }
<条件>       = { 項1<CC>項2 | <CC> }
<CC>         = { EQ | NE | GT | LT | GE | LE | HI | LO | HS |
                 LS | CC | CS | VC | VS | PL | MI | T  | F  }
```

ラベルは記述できません。

説　明　　.WHILE で条件判定を行い、条件が成立している間だけ.WHILE~.ENDW 間のソースステートメントを繰り返し実行します。

サイズ、分岐サイズは以下のようになります。

(1)　サイズ

サイズには、比較実行型で比較するレジスタと項のサイズを指定します。
オペレーションサイズを省略すると、.WHILE.B (バイトサイズ)となります。
本指定は、コンディションコード指定型では意味を持ちません。
サイズは以下のようになります。

サイズ	サイズ	
B	**バイト**	(1 **バイト**)
W	**ワード**	(2 **バイト**)
L	**ロングワード**	(4 **バイト**)

(2)　分岐サイズ

分岐サイズには、.WHILE から.ENDW までの分岐サイズを指定します。
分岐サイズは次のようになります。

オペレーション	分岐サイズ
8	8 **ビット**
16	16 **ビット**

指定を省略した場合の設定は、
11.3 .DISPSIZE **の** FBR、3.2.2 br_relative、3.2.2 [no]optimize を参照してください。

コンディションコードの条件については、**表** 11.18 **コンディションコード一覧**を参照してください。

また、条件判定には、次のようなものがあります。

(1)　比較実行型

比較実行型は、2 つの項をコンディションコードの条件で判定します。
項には、CMP 命令に指定できるアドレス形式を指定します。

(2)　コンディションコード指定型

コンディションコード指定型は、CCR(コンディションコードレジスタ)の状態で条件判定します。

制限事項

(1) H8/300, H8/300L では、サイズに L を指定できません。
(2) H8/300, H8/300L では、分岐サイズに 16 を指定できません。
(3) .WHILE~.ENDW 間のソースステートメントのサイズは、指定した分岐サイズを超えることはできません。
分岐サイズに対するソースステートメントの最大サイズは次の通りです。
8 …… 100 バイト程度
16 …… 32,700 バイト程度
(4) 本制御命令を使用すると、_$W00000～_$W99999 のシンボルを展開します。
従って、同じ名称のシンボルは使用することができません。

例

(1)

```
        .WHILE.B  (#50<GT>R0L)
                ADD.W     R1,R2          ; [1]
                ADD.B     #1:8,R0L       ; [1]
        .ENDW
```

比較実行型の例です。
50>R0L(符号付き比較)の間だけ[1]を繰り返します。

(2)

```
        .WHILE.W  (R0<LS>R1)
                MOV.B     R2L,R3L        ; [2]
                SUB.W     R5,R1          ; [2]
        .ENDF
```

比較実行型の例です。
R0≦R1(符号なし比較)の間だけ[2]を繰り返します。

(3)

```
        .WHILE (<NE>)
                MOV.L     @ER2,ER4       ; [3]
                MOV.L     ER4,@ER3       ; [3]
                ADDS.L    #4,ER2         ; [3]
                ADDS.L    #4,ER3         ; [3]
                SUB.B     R1L,R0L        ; [3]
        .ENDW
```

コンディションコード指定型の例です。
CCR(コンディションコードレジスタ)の Z(ゼロ)フラグが 0 の間だけ[3]を繰り返します。

(4)

```
        .WHILE (<PL>)
                MOV.L     ER2,@ER1       ; [4]
                ADDS.L    #4,ER1         ; [4]
                MOV.L     ER3,@ER1       ; [4]
                ADDS.L    #4,ER1         ; [4]
                ADD.W     #-1,R0         ; [4]
        .ENDW
```

コンディションコード指定型の例です。
CCR(コンディションコードレジスタ)の N(ネガティブ)フラグが 0 の間だけ[4]を繰り返し実行します。

.REPEAT

書　式　　△.REPEAT
△.UNTIL [.<サイズ>]△<条件>
<サイズ>　　= { B | W | L }
<条件>　　= { 項1<CC>項2 | <CC> }
<CC>　　= { EQ | NE | GT | LT | GE | LE | HI | LO | HS |
LS | CC | CS | VC | VS | PL | MI | T | F }
ラベルは記述できません。

説　明　　.UNTIL で条件判定し、条件が成立するまで、.REPEAT~.UNTIL 間のソースステートメントを繰り返し実行します。
.UNTIL で条件判定するため、必ず一回は.REPEAT~.UNTIL 間のソースステートメントを実行します。

サイズは以下のようになります。
サイズには、比較実行型で比較するレジスタと項のサイズを指定します。
オペレーションサイズを省略すると、.UNTIL.B (バイトサイズ)となります。
本指定は、コンディションコード指定型では意味を持ちません。
サイズは以下のようになります。

サイズ	サイズ	
B	**バイト**	(1 **バイト**)
W	**ワード**	(2 **バイト**)
L	**ロングワード**	(4 **バイト**)

コンディションコードの条件については、**表** 11.18 **コンディションコード一覧**を参照してください。

また、条件判定には、次のようなものがあります。
(1)　比較実行型
比較実行型は、2 つの項をコンディションコードの条件で判定します。
項には、CMP 命令に指定できるアドレス形式を指定します。

(2)　コンディションコード指定型
コンディションコード指定型は、CCR(コンディションコードレジスタ)の状態で条件判定します。

制限事項　　(1) H8/300, H8/300L では、サイズに L を指定できません。
(2) .REPEAT~.UNTIL 間のソースステートメントのサイズは、CPU 種別が 300、300L のとき、100 バイト程度、その他のとき、32,700 バイト程度となります。
(3) 本制御命令を使用すると、_$R00000～_$R99999 のシンボルを展開します。
従って、同じ名称のシンボルは使用することができません。

例　　(1)

```
        .REPEAT
                MOV.L     @ER0,ER2        ; [1]
                MOV.L     ER2,@ER1        ; [1]
                ADDS.L    #4,ER0          ; [1]
                ADDS.L    #4,ER1          ; [1]
        .UNTIL (#H'001000<LS>ER0)
```

比較実行型の例です。
H'001000≦ER0(符号なし比較)が成立するまで[1]を繰り返し実行します。

(2)

```
        .REPEAT
                ADD.W     R2,R3           ; [2]
                ADD.W     R2,R4           ; [2]
                SUB.B     R1L,R0L         ; [2]
        .UNTIL (<EQ>)
```

コンディションコードの指定型の例です。
CCR(コンディションコードレジスタ)の Z(ゼロ)フラグが 1 になるまで[2]を実行します。

処理を中断 (終了)

.BREAK

書　式　　△.BREAK[:<分岐サイズ>]
　　<分岐サイズ> = { 8 | 16 }
ラベルは記述できません。

説　明　　.FOR[U]、.WHILE、.REPEAT の繰り返し処理で、.BREAK 以下のソースステートメントを実行しないで繰り返し処理を終了します。
具体的には、.FOR[U]、.WHILE、.REPEAT の繰り返し処理で、それぞれ.ENDF、.ENDW、.UNTIL へ分岐して繰り返し処理を終了します。

分岐サイズには、.BREAK から.ENDF、.ENDW、.UNTIL までの分岐サイズを指定します。
分岐サイズは次のようになります。

オペレーション	分岐サイズ
8	8 ビット
16	16 ビット

指定を省略した場合の設定は、
11.3 .DISPSIZE の FBR、3.2.2 br_relative、3.2.2 [no]optimize を参照してください。

本制御文は、.SWITCH でも使用します。
.SWITCH での使用方法については、11.7 .SWITCH を参照してください。

制限事項　　H8/300, H8/300L では、分岐サイズに 16 を指定できません。

例

```
        .WHILE (<T>)
           .IF.B  (#10<LE>R0L)
               .BREAK
           .ENDI
           ADD.W   R1,R2
           INC.B   R0L
        .ENDW
```

10≦R0L の場合、繰り返し処理を中断終了します。

.CONTINUE

書　式　　△.CONTINUE[:<分岐サイズ>]
　　<分岐サイズ> = { 8 | 16 }
ラベルは記述できません。

説　明　　.FOR[U]、.WHILE、.REPEAT の繰り返し処理で、.CONTINUE 以下のソースステートメントを実行しないで繰り返し処理を継続します。
具体的には、.FOR[U]、.WHILE、.REPEAT の繰り返し処理で、それぞれの繰り返し処理の条件判定に分岐します。

分岐サイズには、.CONTINUE から.ENDF、.ENDW、.UNTIL までの分岐サイズを指定します。
分岐サイズは次のようになります。

オペレーション	分岐サイズ
8	8 ビット
16	16 ビット

指定を省略した場合の設定は、
11.3 .DISPSIZE の FBR、3.2.2 br_relative、3.2.2 [no]optimize を参照してください。

制限事項　　H8/300, H8/300L では、分岐サイズに 16 を指定できません。

例

```
.WHILE.B  (#10<GT>R0L)
   INC.B    R0L
   INC.B    R1L
   .IF.B    (#10<LT>R1L)
       .CONTINUE
   .ENDI
   ADD.W    R2,R3              ; [1]
.ENDW
```

10＜R1L の場合には[1]を実行しません。

12. コンパイラのエラーメッセージ

12.1 エラー形式とエラーレベル

本章では、以下の形式で出力するエラーメッセージとエラー内容を説明します。

エラー番号　(エラーレベル) エラーメッセージ
　　エラー内容

エラーレベルは、エラーの重要度に従い、5 種類に分類されます。

エラーレベル		動作
(I)	インフォメーション	処理を継続し、オブジェクトプログラムを出力します。
(W)	ウォーニング	処理を継続し、オブジェクトプログラムを出力します。
(E)	エラー	処理を継続します。オブジェクトプログラムは出力しません。
(F)	フェータル	エラーメッセージの出力と同時に、処理を中断します。
(-)	インターナル	エラーメッセージの出力と同時に、処理を中断します。

12.2 メッセージ一覧

C0002 (I) No declarator
　　宣言子のない宣言があります。

C0003 (I) Unreachable statement
　　実行されることのない文があります。

C0004 (I) Constant as condition
　　if 文または switch 文の条件を示す式として、定数式を指定しています。

C0005 (I) Precision lost
　　代入式において、右辺の式の値を左辺の型へ変換する時に、精度が失われる可能性があります。

C0006 (I) Conversion in argument
　　関数の引数の式が、原型宣言で指定した引数の型に変換されます。

C0008 (I) Conversion in return
　　リターン文の式が、関数の返す値の型に変換されます。

C0010 (I) Elimination of needless expression
　　不要な式があります。

C0011 (I) Used before set symbol "変数名"
　　値の設定されていない局所変数を参照しています。

C0015 (I) No return value
　　void 型以外の型を返す関数の中で、リターン文が値を返していないか、またはリターン文がありません。

C0100 (I) Function "関数名" not optimized
　　関数のサイズが大きすぎるため、最適化できません。

C0200 (I) No prototype function
　　関数の原型宣言がありません。

C0300 (I) #pragma interrupt has no effect
　　#pragma interrupt で指定された関数が存在しません。

C0301 (I) #pragma abs8 has no effect
　　#pragma abs8 で指定された変数が存在しません。

C0302 (I) #pragma abs16 has no effect
　　#pragma abs16 で指定された変数が存在しません。

C0303 (I) #pragma indirect has no effect
　　#pragma indirect で指定された関数が存在しません。

C0304 (I) #pragma regsave/noregsave has no effect
　　#pragma regsave/noregsave で指定された関数が存在しません。

C0305 (I) #pragma inline/inline_asm has no effect
　　#pragma inline/inline_asm で指定された関数が存在しません。

C0306 (I) #pragma global_register has no effect
　　#pragma global_register で指定された変数が存在しません。

C0307 (I) #pragma entry has no effect
　　#pragma entry で指定された宣言が存在しません。

C1000 (W) Illegal pointer assignment
　　ポインタ型どうしの代入で、それぞれのポインタ型の指す型が異なっています。

C1001 (W) Illegal comparison in "演算子"
　　二項演算子 == または != の被演算子が、一方がポインタ型で他方が値 0 以外の汎整数型を指しています。

C1002 (W) Illegal pointer for "演算子"
　　二項演算子 ==、!=、>、<、>= または <= の被演算子が、同じ型へのポインタ型を指していません。

C1005 (W) Undefined escape sequence
文字定数または文字列の中で、文法上定義していない拡張表記（バックスラッシュとそれに続く文字）を用いています。

C1007 (W) Long character constant
文字定数の長さが 2 文字になっています。

C1008 (W) Identifier too long
識別子の長さが 8189 文字を超えています。8190 文字以降は無効となります。

C1010 (W) Character constant too long
文字定数の長さが 2 文字を超えています。3 文字目以降は無効となります。

C1012 (W) Floating point constant overflow
浮動小数点定数の値が値の範囲を超えています。符号にしたがって＋∞ または－∞ に対応する内部表現の値を仮定します。

C1013 (W) Integer constant overflow
整定数の値が unsigned long 型のとり得る値の範囲を超えています。オーバフローした上位ビットを無視した値を仮定します。

C1014 (W) Escape sequence overflow
文字定数あるいは文字列の中でのビットパターンを示す拡張表記の値が 255 を超えています。下位 1 バイトの値を有効とします。

C1015 (W) Floating point constant underflow
浮動小数点定数の値の絶対値が表現できる最小値よりも小さな値となっています。定数の値を 0.0 と仮定します。

C1016 (W) Argument mismatch
原型宣言の中の引数と関数呼び出しの対応する引数の型がポインタ型で、それぞれの指す型が異なっています。関数呼び出しにおける引数のポインタの内部表現をそのまま設定します。

C1017 (W) Return type mismatch
関数の返す型とリターン文の式の型がポインタ型で、それぞれの指す型が異なっています。リターン文の式のポインタの内部表現をそのまま設定します。

C1019 (W) Illegal constant expression
定数式において関係演算子 <、>、<= または >= の被演算子が、同じ型へのポインタ型を指していません。結果の値を 0 と仮定します。

C1020 (W) Illegal constant expression of "-"
定数式において二項演算子 - の被演算子が、同じ型へのポインタ型を指していません。結果の値を 0 と仮定します。

C1021 (W) Convert to sjis-space
日本語コードで指定の出力コードに変換できないものがあります。シフト JIS のスペースに変換します。

C1022 (W) Convert to euc-space
日本語コードで指定の出力コードに変換できないものがあります。EUC のスペースに変換します。

C1023 (W) Can not convert japanese code "文字" to output type
日本語コードで指定の出力コードに変換できないものがあります。スペースに変換します。

C1024 (W) First operand of "演算子" is not lvalue
演算子の第一オペランドに左辺値以外を指定しています。

C1025 (W) Out of float
浮動小数点定数の有効桁数が 17 桁を超えています。18 桁以降は無効となります。

C1200 (W) Division by floating point zero
定数式の中で浮動小数点数 0.0 を除数とする割り算を行っています。符号にしたがって、＋∞または－∞に対応する内部表現の値を仮定します。

C1201 (W) Ineffective floating point operation
定数式の中で ∞－∞、0.0/0.0 等の無効演算を行っています。無効演算の結果を表わす非数に対応する内部表現の値を仮定します。

C1300 (W) Command parameter specified twice
同じコンパイラオプションを 2 度以上指定しています。同じコンパイラオプションの中で最後に指定したものを有効とします。

C1302 (W) 'frame' or 'noframe' option ignored
最適化指定ありのときに frame オプション、または最適化指定なしのときに noframe オプションを指定しています。オプションの指定を無視します。

C1305 (W) 'show=object' option ignored
アセンブリソースプログラムの出力指定時に、show=object オプションを指定しています。オプションの指定を無視します。

C1306 (W) 'speed=inline' option ignored
最適化指定なしのときに speed=inline オプションを指定しています。オプションの指定を無視します。

C1307 (W) Section name too long
セクション名称の長さが 8192 文字を超えています。8192 文字までを有効とし、それ以降の文字を無視します。

C1308 (W) 'speed=loop' option ignored
最適化指定なしのときに speed=loop オプションを指定しています。オプションの指定を無視します。

C1310 (W) 'goptimize' option ignored
アセンブリソースプログラムの出力指定時に goptimize オプションを指定しています。オプションの指定を無視します。

C1311 (W) 'cmncode' option ignored
最適化指定なしのときに cmncode オプションを指定しています。オプションの指定を無視します。

C1313 (W) Invalid SBR value
sbr オプションの下位 8bit に 0 以外を指定しています。下位 8bit の指定を無視します。

C1314 (W) 'ecpp' option ignored
C++例外処理機能の有効指定時に ecpp オプションを指定しています。オプションの指定を無視します。

C1315 (W) 'noregexpansion' option ignored
CPU 種別が H8SX のときに、noregexpansion オプションを指定しています。オプションの指定を無視します。

C1316 (W) 'cmncode' option ignored
CPU 種別が H8SX のときに、cmncode オプションを指定しています。オプションの指定を無視します。

C1318 (W) 'align=4' option ignored
CPU 種別が H8SX 以外のときに、align=4 オプションを指定しています。オプションの指定を無視します。

C1319 (W) 'speed=intrinsic' option ignored
CPU 種別が H8SX 以外のときに、speed=intrinsic オプションを指定しています。オプションの指定を無視します。

C1321 (W) 'sbr' option ignored
CPU 種別が H8SX 以外のときに、sbr オプションを指定しています。オプションの指定を無視します。

C1322 (W) 'volatile_loop' option ignored
CPU 種別が H8SX 以外のときに、volatile_loop オプションを指定しています。オプションの指定を無視します。

C1323 (W) 'infinite_loop' option ignored
CPU 種別が H8SX 以外のときに、infinite_loop オプションを指定しています。オプションの指定を無視します。

C1324 (W) 'ptr16' option ignored
CPU 種別が H8SX 以外のときに、ptr16 オプションを指定しています。オプションの指定を無視します。

C1325 (W) 'del_vacant_loop' option ignored
CPU 種別が H8SX 以外のときに、del_vacant_loop オプションを指定しています。オプションの指定を無視します。

C1326 (W) 'global_alloc' option ignored
CPU 種別が H8SX 以外のときに、global_alloc オプションを指定しています。オプションの指定を無視します。

C1327 (W) 'struct_alloc' option ignored
CPU 種別が H8SX 以外のときに、struct_alloc オプションを指定しています。オプションの指定を無視します。

C1328 (W) 'const_var_propagate' option ignored
CPU 種別が H8SX 以外のときに、const_var_propagate オプションを指定しています。オプションの指定を無視します。

C1329 (W) 'opt_range' option ignored
CPU 種別が H8SX 以外のときに、opt_range オプションを指定しています。オプションの指定を無視します。

C1330 (W) 'max_unroll' option ignored
CPU 種別が H8SX 以外のときに、max_unroll オプションを指定しています。オプションの指定を無視します。

C1331 (W) Section name "S" specified
セクション名に S を指定しています。
コンパイラが生成したスタック領域のセクション名と重なる場合があります。

C1332 (W) 'indirect=extended' option ignored
CPU 種別が H8SX 以外のとき、indirect=extended オプションを指定しています。オプションの指定を無視します。

C1400 (W) Function "関数名" in #pragma inline is not expanded
#pragma inline で指定した関数がインライン展開されませんでした。#pragma inline 指定を無視します。

C1401 (W) #pragma abs16 ignored
CPU/動作モードが H8SXN、H8SXM、2600n、2000n、300hn および 300 のときに、#pragma abs16 を指定しています。#pragma abs16 指定を無視します。

C1403 (W) #pragma asm ignored
オブジェクト形式がリロケータブルオブジェクトプログラムのときに、#pragma asm を指定しています。#pragma asm 指定を無視します。

C1404 (W) 'case=table' option ignored by switch
switch 文をテーブル方式に展開することができません。switch 文を if-then 方式で展開します。

C1405 (W) Illegal #pragma syntax
認識できない#pragma 文を指定しています。#pragma 指定を無視します。

C1406 (W) Abs8 attribute ignored
abs8 指定を無視しました。

C1510 (W) Illegal bit width
CPU オプションでのビットサイズ指定が不正です。

C1511 (W) Illegal value in operand
オペランドに範囲外の値が指定されました。

C2000 (E) Illegal preprocessor keyword
プリプロセッサ文で、誤ったキーワードを使用しています。

C2001 (E) Illegal preprocessor syntax
プリプロセッサ文またはマクロ呼び出しの指定方法に誤りがあります。

C2007 (E) Invalid include file name "ファイル名"
インクルードファイル名の指定が不正です。

C2016 (E) Preprocessor constant expression too complex
#if 文、#elif 文で指定した定数式の演算子と被演算子の合計が 512 個を超えています。

C2019 (E) File name too long
ファイル名の長さが 4096 文字を超えています。

C2020 (E) System identifier "名前" redefined
組み込み関数と同名のシンボルを定義しています。

C2021 (E) System identifier "名前" mismatch
指定された CPU/動作モードに存在しない組み込み関数を使用しています。

C2100 (E) Multiple storage classes
宣言の中で二つ以上の記憶クラス指定子を指定しています。

C2101 (E) Address of register
レジスタ記憶クラスを持つ変数に対して、単項演算子&を適用しています。

C2102 (E) Illegal type combination
型指定子の組み合わせが誤っています。

`C2103 (E) Bad self reference structure`
構造体、共用体のメンバの型を、親の構造体または共用体と同じ型で宣言しています。

`C2104 (E) Illegal bit field width`
ビットフィールド幅を示す定数式が整数型でありません。あるいはビットフィールド幅として負の整数を指定しています。

`C2105 (E) Incomplete tag used in declaration`
構造体または共用体で仮宣言されたタグ名、または未宣言のタグ名を `typedef` 宣言、ポインタを指す型あるいは関数の返す型以外の宣言で使用しています。

`C2106 (E) Extern variable initialized`
複文内で `extern` 記憶クラスを指定した宣言に対して初期値を指定しています。

`C2107 (E) Array of function`
要素の型が関数型となる配列型を指定しています。

`C2108 (E) Function returning array`
リターン値の型が配列型となる関数型を指定しています。

`C2109 (E) Illegal function declaration`
複文内の関数型の変数宣言において、`extern` 以外の記憶クラスを指定しています。

`C2110 (E) Illegal storage class`
外部定義の中で記憶クラスとして `auto` または `register` を指定しています。

`C2111 (E) Function as a member`
構造体または共用体のメンバの型に関数型を指定しています。

`C2112 (E) Illegal bit field`
ビットフィールドに誤った型を指定しています。ビットフィールドに許される型指定子は、`char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, enum, bool` のいずれか、これらの型に `const` または `volatile` を組み合わせた型です。

`C2113 (E) Bit field too wide`
ビットフィールド幅が型指定子で指定したサイズ（8、16、32 ビット）を超えています。

`C2114 (E) Multiple variable declarations`
変数名を同じ有効範囲の中で重複して宣言しています。

`C2115 (E) Multiple tag declarations`
構造体、共用体、列挙型のタグ名を同じ有効範囲の中で重複して宣言しています。

`C2117 (E) Empty source program`
ソースプログラム内に外部定義が含まれていません。

C2118 (E) Prototype mismatch "関数名"
関数の型が以前になされている宣言で指定した型と一致しません。

C2119 (E) Not a parameter name "引数名"
関数の引数宣言列にない識別子に対して引数宣言を行っています。

C2120 (E) Illegal parameter storage class
関数の引数宣言で register 以外の記憶クラスを指定しています。

C2121 (E) Illegal tag name
構造体、共用体または列挙型とタグ名の組み合わせが、以前に宣言した型とタグ名の組み合わせと異なっています。

C2122 (E) Bit field width 0
メンバ名を指定しているビットフィールドの幅が 0 になっています。

C2123 (E) Undefined tag name
列挙型の宣言で未定義のタグ名を使用しています。

C2124 (E) Illegal enum value
列挙型のメンバに整数でない定数式を指定しています。

C2125 (E) Function returning function
リターン値の型が関数型となる関数型を指定しています。

C2126 (E) Illegal array size
配列の要素数を指定する値が制限値を超えています。配列要素数の制限値は、CPU/動作モードが
H8SXN,2600n,2000n,300hn,300 のとき 65535、
H8SXA と H8SXX とで ptr16 オプション指定ありのとき、または、H8SXM のとき 32767、
H8SXA:20,2600a:20,2000a:20,300ha:20 のとき 1048575、
H8SXA:24,2600a:24,2000a:24,300ha:24 のとき 16777215、
H8SXA:28,H8SXX:28,2600a:28,2000a:28 のとき 268435455、
H8SXA:32,H8SXX:32,2600a:32,2000a:32 のとき 4294967295 です。

C2127 (E) Missing array size
配列の要素数の指定がありません。

C2129 (E) Illegal initializer type
変数の初期値指定において初期値の型が変数に代入可能な型でありません。

C2130 (E) Initializer should be constant
構造体型、共用体型、配列型の変数の初期値、または静的に割り付けられる変数の初期値に定数式でないものを指定しています。

C2131 (E) No type nor storage class
外部データ定義において記憶クラスまたは型の指定がありません。

C2132 (E) No parameter name
関数の引数宣言列が空であるにもかかわらず引数の宣言を行っています。

C2133 (E) Multiple parameter declarations
（マクロ）関数定義の引数宣言列の中で同一名の引数を重複して宣言しているか、または引数宣言が関数宣言子の中と外の 2 箇所で行われています。

C2134 (E) Initializer for parameter
引数の宣言において初期値を指定しています。

C2135 (E) Multiple initialization
同一の変数に対して、初期化を重複して行っています。

C2136 (E) Type mismatch
extern あるいは static 記憶クラスを持つ変数あるいは関数を 2 度以上宣言しており、その型が一致していません。

C2137 (E) Null declaration for parameter
関数の引数宣言で識別子を指定していません。

C2138 (E) Too many initializers
構造体、共用体または配列の初期値指定において、構造体のメンバ数または配列の要素数より多く初期値の数を指定しています。あるいは、共用体の最初のメンバがスカラ型のときに 2 個以上の初期値を指定しています。

C2139 (E) No parameter type
関数宣言の引数宣言に型指定がありません。

C2140 (E) Illegal bit field
共用体にビットフィールドを指定しています。

C2141 (E) Struct has no member name
構造体の先頭のメンバに無名のビットフィールドを指定しています。

C2142 (E) Illegal void type
void 型の指定方法に誤りがあります。void 型を指定できるのは以下の三つの場合です。
（1）ポインタの指す先の型として指定する場合。
（2）関数の返す型として指定する場合。
（3）原型宣言の関数が引数を持たないことを明示的に指定する場合。

C2143 (E) Illegal static function
ソースファイル内に定義のない static 記憶クラスを持つ関数宣言があります。

C2150 (E) Multiple function qualifiers
複数の関数修飾子を指定しています。

C2151 (E) "名前" must be qualified for function type
関数型以外を修飾できません。

C2152 (E) Illegal attribute combination
キーワードの組み合わせ方が許されていません。
組み合わせとして許されているのは、以下の"○"で示されたキーワードです。

	__near8	__near16	__abs8	__abs16	__ptr16	__interrupt	__inline	__indirect	__indirect_ex	__regsave	__noregsave
__near8	×	×	○	○	×	×	×	×	×	×	×
__near16	×	×	○	○	×	×	×	×	×	×	×
__abs8	○	○	×	×	×	×	×	×	×	×	×
__abs16	○	○	×	×	○	×	×	×	×	×	×
__ptr16	×	×	×	○	×	×	×	×	×	×	×
__interrupt	×	×	×	×	×	×	×	×	×	○	○
__inline	×	×	×	×	×	×	×	○	○	×	×
__indirect	×	×	×	×	×	×	○	×	×	○	○
__indirect_ex	×	×	×	×	×	×	○	×	×	○	○
__regsave	×	×	×	×	×	○	×	○	○	×	×
__noregsave	×	×	×	×	×	○	×	○	○	×	×

C2153 (E) Illegal "名前" specifier
属性指定子の記述方法に誤りがあります。

C2154 (E) "名前" must be specified for variables
この属性指定子は変数のみ指定できます。

C2155 (E) "名前" must be specified for functions
この属性指定子は関数のみ指定できます。

C2157 (E) Attribute keyword and pragma cannot be specified for one symbol
属性キーワードと#pragma 宣言の同時指定はできません。

C2158 (E) Attribute mismatch
宣言間で属性が一致していません。

C2159 (E) Multiple entry functions
エントリ関数が複数指定されています。

C2160 (E) Illegal '__near8/__near16' variable size
__near8/__near16 を指定した変数のサイズが範囲を超えています。

C2161 (E) Illegal '__abs8' variable type
__abs8 変数の型が不正です。

C2162 (E) Illegal '__global_register' variable type
__global_register 変数の型が不正です。

C2163 (E) Illegal '__interrupt' function type
割り込み関数の型が不正です。

C2164 (E) Cannot specify '名前' to local storage class
ここでは指定できない属性を使用しています。

C2165 (E) Multiple pointer qualifiers
__ptr16 が複数指定されました。

C2166 (E) '__ptr16' must be qualified for data pointer type
__ptr16 がデータポインタ型以外に指定されました。

C2190 (E) Multiple functions on vector "ベクタ番号"
同じベクタ番号に複数の関数が指定されています。

C2200 (E) Index not integer
配列の添字の式が整数型ではありません。

C2201 (E) Cannot convert parameter "n"
関数呼び出しにおける n 番目の引数を対応する原型宣言の引数の型に変換できません。

C2202 (E) Number of parameters mismatch
関数呼び出しにおける引数の数が原型宣言の引数の数と一致しません。

C2203 (E) Illegal member reference for "."
演算子 . の左側の式の型が構造体型、共用体型ではありません。

C2204 (E) Illegal member reference for "->"
演算子 -> の左側の式の型が構造体型または共用体型へのポインタではありません。

C2205 (E) Undefined member name
構造体、共用体への参照で宣言していないメンバ名を使用しています。

C2206 (E) Modifiable lvalue required for "演算子"
前置または後置演算子 ++、-- を代入可能な左辺値(配列型、const 型を除く左辺値)でない式に使用しています。

C2207 (E) Scalar required for "!"
単項演算子 ! をスカラ型でない式に使用しています。

C2208 (E) Pointer required for "*"
単項演算子 * をポインタ型でない式か、または void 型へのポインタ型の式に使用しています。

C2209 (E) Arithmetic type required for "演算子"
単項演算子 + または - を算術型でない式に使用しています。

C2210 (E) Integer required for "~"
単項演算子 ~ を汎整数型でない式に使用しています。

C2211 (E) Illegal sizeof
sizeof 演算子をビットフィールドの指定のあるメンバ、関数型、void 型またはサイズの指定していない配列に使用しています。

C2212 (E) Illegal cast
キャスト演算子で指定している型が配列型、構造体型または共用体型です。あるいはキャスト演算子の被演算子が void 型、構造体型か共用体型で型変換できません。

C2213 (E) Arithmetic type required for "演算子"
二項演算子 *、/、*= または /= を算術型でない式に適用しています。

C2214 (E) Integer required for "演算子"
二項演算子 <<、>>、&、|、^、%、<<=、>>=、&=、|=、^= または %= を汎整数型でない式に適用しています。

C2215 (E) Illegal type for "+"
二項演算子 + の被演算子の型の組み合わせが許されていません。二項演算子 + の型の組み合わせで許されるのは、両辺とも算術型の場合と、一方がポインタ型で他方が汎整数型の場合だけです。

C2216 (E) Illegal type for parameter
関数呼び出しの引数の型に void 型を指定しています。

C2217 (E) Illegal type for "-"
二項演算子 - の被演算子の型の組み合わせが許されていません。二項演算子 - の型の組み合わせで許されるのは、以下の三つの場合です。
（1）両方の被演算子が算術型の場合。
（2）両方の被演算子が同じ型へのポインタ型の場合。
（3）第 1 被演算子がポインタ型で、第 2 被演算子が汎整数型の場合。

C2218 (E) Scalar required in "?:"
条件演算子 ?：の第 1 被演算子の型がスカラ型でありません。

C2219 (E) Type not compatible in "?:"
条件演算子 ? : の第 2 被演算子と第 3 被演算子の型が合っていません。条件演算子 ? : の第 2 被演算子と第 3 被演算子の組み合わせで許されるのは、以下の六つの場合です。
（1）両方とも算術型の場合。
（2）両方とも void 型の場合。
（3）両方とも同じ型へのポインタ型の場合。
（4）一方がポインタ型で、他方が値 0 の整定数または値 0 の整定数を void 型へのポインタ型に変換したものである場合。
（5）一方がポインタ型で、他方が void 型へのポインタ型の場合。
（6）両方とも同じ型の構造体または共用体の場合。

C2220 (E) Modifiable lvalue required for "演算子"
代入演算子 =、*=、/=、%=、+=、-=、<<=、>>=、&=、^= または |= の左辺の式に代入可能な左辺値(配列型、const 型を除く左辺値)以外の式を指定しています。

C2221 (E) Illegal type for "演算子"
後置演算子 ++ または -- の被演算子にスカラ型以外の型、関数型または void 型へのポインタ型を指定しています。

C2222 (E) Type not compatible for "="
代入演算子 = の両辺の式の型が合っていません。代入演算子 = の両辺の式の組み合わせで許されるのは、以下の五つの場合です。
（1）両方とも算術型の場合。
（2）両方とも同じ型へのポインタ型の場合。
（3）左辺がポインタ型で、右辺が値 0 の整定数または値 0 の整定数を void 型へのポインタ型に変換したものである場合。
（4）一方がポインタ型で、他方が void 型へのポインタ型の場合。
（5）両方とも同じ型の構造体または共用体の場合。

C2223 (E) Incomplete tag used in expression
構造体または共用体で仮宣言されたタグ名を式中で使用しています。

C2224 (E) Illegal type for assign
代入演算子 += または -= の両辺の型が正しくありません。

C2225 (E) Undeclared name "名前"
宣言していない名前を式の中で用いています。

C2226 (E) Scalar required for "演算子"
二項演算子 && または || をスカラ型でない式に適用しています。

C2227 (E) Illegal type for equality
等値演算子 == または != の被演算子の型の組み合わせが許されていません。等値演算の被演算子の組み合わせで許されるのは、以下の三つの場合です。
（1）両方とも算術型の場合。
（2）両方とも同じ型へのポインタ型の場合。
（3）一方がポインタ型で、他方が値 0 の整定数または void 型へのポインタ型である場合。

C2228 (E) Illegal type for comparison
関係演算子 >、<、>= または <= の被演算子の型の組み合わせが許されていません。関係演算子の被演算子の組み合わせで許されるのは、以下の二つの場合です。
（1）両方とも算術型の場合。
（2）両方とも同じ型へのポインタ型の場合。

C2230 (E) Illegal function call
関数呼び出しにおいて、関数型あるいは関数型へのポインタ型でない式を用いています。

C2231 (E) Address of bit field
単項演算子 & をビットフィールドに適用しています。

C2232 (E) Illegal type for "演算子"
前置演算子 ++ または -- の被演算子にスカラ型以外の型、関数型または void 型へのポインタ型を指定しています。

C2233 (E) Illegal array reference
配列型、関数型または void 型を除くポインタ型以外の式を配列として使用しています。

C2234 (E) Illegal typedef name reference
typedef 宣言された名前を式の中で変数として使用しています。

C2235 (E) Illegal cast
ポインタを浮動小数点型にキャストしています。

C2237 (E) Illegal constant expression
定数式の中でポインタ型の定数を整数型へキャストした結果に対して演算を行っています。

C2238 (E) Lvalue or function type required for "&"
単項演算子&を左辺値あるいは関数型以外の式に適用しています。

C2239 (E) Illegal section name
セクション名に使用できない文字があります。

C2240 (E) Illegal section naming
セクションの命名に誤りがあります。用途の異なるセクションに同じ名前を付けています。

C2300 (E) Case not in switch
case ラベルを switch 文以外に指定しています。

C2301 (E) Default not in switch
default ラベルを switch 文以外に指定しています。

C2302 (E) Multiple labels
一つの関数内にラベル名を重複して定義しています。

C2303 (E) Illegal continue
continue 文を while 文、for 文または do 文以外に指定しています。

C2304 (E) Illegal break
break 文を while 文、for 文、do 文または switch 文以外に指定しています。

C2305 (E) Void function returns value
void 型を返す関数の中の return 文でリターン値を指定しています。

C2306 (E) Case label not constant
case ラベルの式が汎整数型の定数式ではありません。

C2307 (E) Multiple case labels
同一の値を持つ case ラベルを一つの switch 文の中に重複して指定しています。

C2308 (E) Multiple default labels
default ラベルを一つの switch 文の中に重複して指定しています。

C2309 (E) No label for goto
goto 文で指定した行き先のラベルがありません。

C2310 (E) Scalar required “while, for, do”
while 文、for 文または do 文の制御式（文の実行を判定する式）がスカラ型ではありません。

C2311 (E) Integer required
switch 文の制御式（文の実行を判定する式）が汎整数型ではありません。

C2312 (E) Missing (
if 文、while 文、for 文、do 文または switch 文の制御式（文の実行を判定する式）の左括弧"("がありません。

C2313 (E) Missing ;
do 文の最後のセミコロン";"がありません。

C2314 (E) Scalar required "if"
if 文の制御式（文の実行を判定する式）がスカラ型ではありません。

C2316 (E) Illegal type for return value
return 文の式の型を関数の返す型に変換することができません。

C2320 (E) Illegal asm position
#pragma asm の指定位置が適切ではありません。

C2330 (E) Illegal #pragma interrupt declaration
割り込み関数宣言に誤りがあります。

C2331 (E) Illegal interrupt function call
割り込み関数宣言のある関数をプログラム中で呼び出しまたは参照しています。

C2332 (E) Function "関数名" in #pragma interrupt already declared
割り込み関数宣言#pragma interrupt で指定した関数を既に通常の関数として宣言しています。

C2333 (E) Multiple interrupt for one function
同一関数に対して割り込み関数宣言#pragma interrupt を重複して宣言しています。

C2334 (E) Illegal parameter in #pragma interrupt function
割り込み関数で使用する引数の型が一致していません。引数の型として指定できるのは void 型だけです。

C2335 (E) Missing parameter declaration in #pragma interrupt function
割り込み関数宣言#pragma interrupt のスタック切り替え指定 (sp) または割り込み終了の指定 (sy) に、宣言していない変数または関数を使用しています。

C2336 (E) Parameter out of range in #pragma interrupt function
割り込み関数宣言#pragma interrupt の引数 tn の値が 3 を超えています。

C2337 (E) Illegal #pragma interrupt function type
割り込み関数宣言に誤りがあります。

C2340 (E) Illegal #pragma abs8 declaration
短絶対アドレス変数宣言に誤りがあります。

C2341 (E) Variable "変数名" in #pragma abs8 already declared
短絶対アドレス変数宣言#pragma abs8 で指定した変数を既に変数として宣言しています。

C2342 (E) Illegal #pragma abs8 symbol type
短絶対アドレス変数宣言#pragma abs8 で指定した変数を変数名以外の型で宣言しています。

C2345 (E) Illegal #pragma abs16 declaration
短絶対アドレス変数宣言に誤りがあります。

C2346 (E) Variable "変数名" in #pragma abs16 already declared
短絶対アドレス変数宣言#pragma abs16 で指定した変数を既に変数として宣言しています。

C2347 (E) Illegal #pragma abs16 symbol type
短絶対アドレス変数宣言#pragma abs16 で指定した変数を変数名以外の型で宣言しています。

C2350 (E) Illegal section name declaration
#pragma section の指定に誤りがあります。

C2352 (E) Section name table overflow
セクション数が全体で 65280 個を超えました。

C2353 (E) Section size overflow regarding "セクション名"
セクションサイズが 32KB を超えました。

C2360 (E) Illegal #pragma indirect declaration
メモリ間接関数宣言に誤りがあります。

C2361 (E) Function "関数名" in #pragma indirect already declared
メモリ間接関数宣言#pragma indirect で指定した関数を既に関数として宣言しています。

C2362 (E) Illegal #pragma indirect function type
メモリ間接関数宣言#pragma indirect で指定した関数を関数以外の型で宣言または定義しています。

C2363 (E) Too many indirect identifiers
1 ファイルで指定できるメモリ間接関数の数が制限を超えました。1 ファイルで指定できるメモリ間接関数の数は、256 個です。

C2370 (E) Illegal #pragma regsave/noregsave declaration
#pragma regsave または#pragma noregsave 宣言に誤りがあります。

C2371 (E) Function "関数名" in #pragma regsave/noregsave already declared
#pragma regsave または#pragma noregsave で指定した関数を既に関数として宣言しています。

C2372 (E) Illegal #pragma regsave/noregsave function type
#pragma regsave または#pragma noregsave で指定した関数を関数以外の型で宣言または定義しています。

C2380 (E) Illegal #pragma inline/inline_asm declaration
#pragma inline または#pragma inline_asm 宣言に誤りがあります。

C2381 (E) Function "関数名" in #pragma inline/inline_asm already declared
#pragma inline または#pragma inline_asm で指定した関数を既に関数として宣言しています。

C2382 (E) Illegal #pragma inline/inline_asm function type
#pragma inline または#pragma inline_asm で指定した関数を関数以外の型で宣言または定義しています。

C2383 (E) #pragma inline_asm ignored
オブジェクト形式がリロケータブルオブジェクトプログラムのときに、#pragma inline_asm を指定しています。

C2390 (E) Illegal #pragma global_register declaration
#pragma global_register 宣言に誤りがあります。

C2391 (E) Variable "変数名" in #pragma global_register already declared
#pragma global_register で指定した変数を既に変数として宣言しています。

C2392 (E) Illegal #pragma global_register symbol type
#pragma global_register で指定した変数を変数以外の型で宣言しています。

C2393 (E) Illegal register
#pragma global_register で指定したレジスタ名に誤りがあります。または、重複指定しています。

C2400 (E) Illegal character "文字"
不正な文字があります。

C2401 (E) Incomplete character constant
文字定数の途中に改行があります。

C2402 (E) Incomplete string
文字列の途中に改行があります。

C2403 (E) EOF in comment
コメントの途中でファイルが終了しました。

C2404 (E) Illegal character code "文字コード"
不正な文字コードがあります。

C2405 (E) Null character constant
文字定数の中に文字を指定していません。すなわち ' ' という形式の文字定数を指定しています。

C2407 (E) Incomplete logical line
空でないソースファイルの最後の文字にバックスラッシュ "＼" 、またはバックスラッシュのあとに改行文字"＼(RET)"を指定しています。

C2408 (E) Comment nest too deep
コメントのネストが 255 レベルを超えています。

C2410 (E) Illegal #pragma entry declaration
#pragma entry 宣言に構文エラーがあります。

C2411 (E) Function "関数名" in #pragma entry already declared
#pragma entry 宣言の前に同名のシンボルがあるか、または既にプラグマ指定されています。

C2412 (E) Illegal #pragma entry function type
指定されたシンボルが関数ではありません。

C2413 (E) Multiple #pragma entry declaration
#pragma entry 宣言が複数存在しています。

C2420 (E) Illegal #pragma pack/unpack declaration
#pragma pack, #pragma unpack 宣言に構文エラーがあります。

C2440 (E) Illegal #pragma stacksize declaration
#pragma stacksize 宣言に構文エラーがあります。

C2441 (E) Multiple #pragma stacksize declaration
#pragma stacksize 宣言が複数存在しています。

C2442 (E) Stack size overflow
#pragma stacksize で指定したスタックサイズが大きすぎます。

C2450 (E) Illegal #pragma option declaration
#pragma option 宣言に誤りがあります。

C2460 (E) Pragma kind mismatch
宣言間で#pragma 種別が一致していません。

C2470 (E) Illegal #pragma bit_order declaration
#pragma bit_order 宣言に誤りがあります。

C2500 (E) Illegal token "語句"
語句の並びが文法に合っていません。

C2501 (E) Division by zero
定数式中で整数型データのゼロ除算が行われました。

C2510 (E) Missing {
__asm ブロック開始の"{"がありません。

C2511 (E) Illegal mnemonic
ニーモニックが不正です。

C2512 (E) Member reference required for "offset"
offset 演算子をメンバ参照式以外に使用しています。

C2513 (E) Number of operands mismatch
オペランド数が不正です。

C2514 (E) Illegal addressing mode
オペランドのアドレッシングモードが不正です。

C2515 (E) Illegal register list
レジスタリストの指定が不正です。

C2516 (E) Illegal immediate data
即値データが不正です。

C2517 (E) Illegal value in operand
オペランドに範囲外の値が指定されました。

C2518 (E) Invalid delay slot instruction
遅延スロットに不正な命令が配置されています。

C2600 (E) #error : "文字列"
nolist オプションが指定されていなければ、#error の文字列で指定されたエラーメッセージをリストファイルに表示します。

C2801 (E) Illegal parameter type in intrinsic function
組み込み関数で引数の型が一致しません。

C2802 (E) Parameter out of range in intrinsic function
組み込み関数で引数の大きさが指定可能範囲を超えています。

C2803 (E) Usage for intrinsic function is wrong
組み込み関数の使い方に間違いがあります。

C3000 (F) Statement nest too deep
if 文、while 文、for 文、do 文および switch 文のネストが、256 レベルを超えています。

C3006 (F) Too many parameters
関数の宣言または呼び出しにおいて引数の数が 63 個を超えています。

C3007 (F) Too many macro parameters
マクロの定義または呼び出しにおいて、引数の数が 63 個を超えています。

C3008 (F) Line too long
マクロ展開後の 1 行の長さが 16384 文字を超えています。

C3009 (F) String literal too long
文字列の長さが 32767 文字を超えています。文字列の長さは、連続して指定した文字列を連結した後のバイト数です。ここでいう文字列の長さとは、ソースプログラム上の長さではなく文字列のデータに含まれるバイト数で、拡張表記も 1 文字に数えます。

C3013 (F) Too many switches
switch 文の数が 2048 個を超えています。

C3014 (F) For nest too deep
for 文のネストが 128 レベルを超えています。

C3017 (F) Too many case labels
一つの switch 文における case ラベルの数が 511 個を超えています。

C3018 (F) Too many goto labels
一つの関数の中で定義している goto ラベルの数が 511 個を超えています。

C3019 (F) Cannot open source file "ファイル名"
ソースファイルをオープンすることができません。

C3020 (F) Source file input error
ソースファイルまたはインクルードファイルを読み込むことができません。

C3021 (F) Memory overflow
コンパイラが内部で使用するメモリ領域を割り当てることができません。

C3022 (F) Switch nest too deep
switch 文のネストが 128 レベルを超えています。

C3023 (F) Type nest too deep
基本型を修飾する型（ポインタ型、配列型、関数型）の数が 16 個を超えています。

C3024 (F) Array dimension too deep
配列の次元数が 6 次元を超えています。

C3025 (F) Source file not found
コマンドラインの中にソースファイル名の指定がありません。

C3026 (F) Expression too complex
式が複雑すぎます。

C3027 (F) Source file too complex
プログラムの文のネストが深いかあるいは、式が複雑すぎます。

C3030 (F) Too many compound statements
1 関数における複文の数が、2048 を超えています。

C3031 (F) Data size overflow
配列または構造体の大きさが、制限値を超えています。配列または構造体の制限値は、CPU/動作モードが
H8SXN,2600n,2000n,300hn,300 のとき 65535、
H8SXA と H8SXX とで ptr16 オプション指定ありのとき、または、H8SXM のとき 32767、
H8SXA:20,2600a:20,2000a:20,300ha:20 のとき 1048575、
H8SXA:24,2600a:24,2000a:24,300ha:24 のとき 16777215、
H8SXA:28,H8SXX:28,2600a:28,2000a:28 のとき 268435455、
H8SXX:32,H8SXX:32,2600a:32,2000a:32 のとき 4294967295 です。

C3034 (F) Invalid file name "ファイル名"
ファイル名の指定が不正です。

C3200 (F) Object size overflow
オブジェクトサイズがメモリの制限を超えています。
オブジェクトサイズの制限値は、CPU/動作モードが
H8SXN,2600n,2000n,300hn,300 のとき 65535、
H8SXM:20,H8SXA:20,2600a:20,2000a:20,300ha:20 のとき 1048575、
H8SXM:24,H8SXA:24,2600a:24,2000a:24,300ha:24 のとき 16777215、
H8SXA:28,H8SXX:28,2600a:28,2000a:28 のとき 268435455、
H8SXA:32,H8SXX:32,2600a:32,2000a:32 のとき 4294967295 です。

C3201 (F) Object data size overflow
データサイズが制限値を超えています。データサイズの制限値は、CPU/動作モードが
H8SXN,H8SXM,2600n,2000n,300hn,300 のとき 65535、
H8SXA と H8SXX とで ptr16 オプション指定ありのとき 65535、
H8SXA:20,2600a:20,2000a:20,300ha:20 のとき 1048575、
H8SXA:24,2600a:24,2000a:24,300ha:24 のとき 16777215、
H8SXA:28,H8SXX:28,2600a:28,2000a:28 のとき 268435455、
H8SXX:32,H8SXX:32,2600a:32,2000a:32 のとき 4294967295 です。

C3202 (F) Illegal stack access
局所変数・テンポラリ領域およびレジスタ退避領域がスタックポインタ(SP)またはフレームポインタ(FP)からのオフセットが制限値より離れています。または、引数領域が SP または FP からのオフセットが制限値より離れています。
SP,FP からのオフセットの制限値は、CPU/動作モードが
H8SXN,H8SXM,2600n,2000n,300hn,300 のとき 32767、
H8SXA と H8SXX とで ptr16 オプション指定ありのとき 32767、
H8SXA:20,2600a:20,2000a:20,300ha:20 のとき 524287、
H8SXA:24,2600a:24,2000a:24,300ha:24 のとき 8388607、
H8SXA:28,H8SXX:28,2600a:28,2000a:28 のとき 134217727、
H8SXA:32,H8SXX:32,2600a:32,2000a:32 のとき 2147483647 です。

C3300 (F) Cannot open internal file
コンパイラが内部で使用する中間ファイルをオープンすることができません。

C3301 (F) Cannot close internal file
コンパイラが内部で生成する中間ファイルをクローズすることができません。コンパイラが生成する中間ファイルをアクセスしていないかを確認してください。

C3302 (F) Cannot input internal file
コンパイラが内部で生成する中間ファイルを読み込むことができません。コンパイラが生成する中間ファイルをアクセスしていないかを確認してください。

C3303 (F) Cannot output internal file
コンパイラが内部で生成する中間ファイルに書き込むことができません。ディスク容量を増やしてください。

C3304 (F) Cannot delete internal file
コンパイラが内部で生成する中間ファイルを削除することができません。コンパイラが生成する中間ファイルをアクセスしていないかを確認してください。

C3305 (F) Invalid command parameter "オプション"
コンパイラオプションの指定方法が誤っています。

C3306 (F) Interrupt in compilation
コンパイル処理中に標準入力端末から 「(cntl)＋C」キーによる割り込みを検出しました。

C3307 (F) Compiler version mismatch in "ファイル名"
コンパイラを構成するファイル間のバージョンが一致していません。インストールガイドの組み込み方法を参照し、コンパイラ本体を再インストールしてください。

C3320 (F) Command parameter buffer overflow
コマンドラインの指定が 4096 文字を超えています。

C3322 (F) Lacking cpu specification
CPU/動作モードの指定がされていません。cpu オプションまたは環境変数 H38CPU で CPU/動作モードを指定してください。

C3323 (F) Illegal environment specified "環境変数"
コンパイラで使用する環境変数（CH38TMP、H38CPU）の指定に誤りがあります。

C3324 (F) Cannot open subcommand file "ファイル名"
サブコマンドファイルをオープンすることができません。

C3325 (F) Cannot close subcommand file
サブコマンドファイルをクローズできません。サブコマンドファイルを使用していないか確認してください。

C3326 (F) Cannot input subcommand file
サブコマンドファイルが読み込めません。

C3327 (F) Cannot find "ファイル名"
コンパイラの実行ファイルが見つかりません。ファイル名またはパス名をもう一度確認してください。

C4000 (-) Internal error
コンパイラの内部処理で何らかの障害が生じました。本コンパイラをお求めになった営業所あるいは代理店にエラーの発生状況をご連絡ください。

C5003 (F) #include file "ファイル名" includes itself
自分自身のファイル"ファイル名"をインクルードしています。

C5004 (F) Out of memory
コンパイルに必要なメモリが不足しています。システムのメモリを増やすか、他のアプリケーションを終了してください。

C5005 (F) Could not open source file "名前"
ファイル"名前"をオープンできませんでした。ファイル名が正しいか確認してください。

C5006 (E) Comment unclosed at end of file
コメントの終了指定 */がありません。

C5007 (E) (I) Unrecognized token
認識できない字句があります(マクロの場合は(I)となります)。

C5008 (E) (I) Missing closing quote
文字列の終了指定 " がありません(マクロの場合は(I)となります)。

C5009 (I) Nested comment is not allowed
/* */コメントがネストしています。

C5010 (E) "#" not expected here
#が行の先頭、プリプロセッサ以外に指定されています。

C5011 (E) Unrecognized preprocessing directive
認識できないプリプロセッサのキーワードがあります。

C5012 (E) Parsing restarts here after previous syntax error
字句の解析を再開しました。

C5013 (E)(F) Expected a file name
ファイル名が必要です。
#include 文では(F)、#line 文では(E)となります。

C5014 (E) Extra text after expected end of preprocessing directive
プリプロセッサ文の後にさらにテキストが記述されています。

C5016 (F) "名前" is not a valid source file name
ファイル"名前"が有効でありません。

C5017 (E) Expected a "]"
"]"がありません。

C5018 (E) Expected a ")"
")"がありません。

C5019 (E) Extra text after expected end of number
数値の後ろにさらにテキストが記述されています。

C5020 (E) Identifier "名前" is undefined
シンボル"名前"の定義がありません。

C5021 (W) Type qualifiers are meaningless in this declaration
意味のない型限定子を指定しています。型限定子を無効にします。

C5022 (E) Invalid hexadecimal number
16 進数の記述に誤りがあります。

C5024 (E) Invalid octal digit
8 進数の記述に誤りがあります。

C5025 (E) Quoted string should contain at least one character
文字定数が空です。

C5026 (E) Too many characters in character constant
文字定数中の文字数が多すぎます。

C5027 (W) Character value is out of range
文字の値が範囲を超えています。超えた値は切り捨てられます。

C5028 (E) Expression must have a constant value
式の値が定数でありません。

C5029 (E) Expected an expression
式が必要です。

C5030 (E) Floating constant is out of range
浮動小数点数の値が範囲を超えています。

C5031 (E) Expression must have integral type
式の型は汎整数型でなければなりません。

C5032 (E) Expression must have arithmetic type
式の型は算術型でなければなりません。

C5033 (E) Expected a line number
#line 文には行番号が必要です。

C5034 (E) Invalid line number
#line 文の行番号が有効でありません。

C5035 (F) #error directive: "行番号"
#error 文が適用されました。

C5036 (E) The #if for this directive is missing
#if 文の指定方法に誤りがあります。

C5037 (E) The #endif for this directive is missing
#endif 文の指定方法に誤りがあります。

C5038 (W) Directive is not allowed -- an #else has already appeared
#else 文はすでに出現しました。本指定を読み飛ばします。

C5039 (E) Division by zero
ゼロ除算が発生しました。

C5040 (E) Expected an identifier
識別子が必要です。

C5041 (E) Expression must have arithmetic or pointer type
式の型は算術型またはポインタ型でなければなりません。

C5042 (E) Operand types are incompatible ("型 1" and "型 2")
"型 1"と"型 2"のオペランドの型が適合しません。

C5044 (E) Expression must have pointer type
式の型はポインタ型でなければなりません。

C5045 (W) #undef may not be used on this predefined name
システムで定義しているマクロ名を取り消すことはできません。#undef 指定を無効にします。

C5046 (W) This predefined name may not be redefined
システムで定義しているマクロ名を再定義することはできません。#define 指定を無効にします。

C5047 (W) Incompatible redefinition of macro "名前" (declared at line "行番号")
マクロ"名前"の再定義が以前の定義の形式と異なります。本マクロ定義を無効にします。

C5049 (E) Duplicate macro parameter name
マクロのパラメタ名を 2 重定義しています。

C5050 (E) "##" may not be first in a macro definition
#define マクロの最初に##が指定されています。

C5051 (E) "##" may not be last in a macro definition
#define マクロの最後に##が指定されています。

C5052 (E) Expected a macro parameter name
#に続くマクロ引数がありません。

C5053 (E) Expected a ":"
":"が必要です。

C5054 (W) Too few arguments in macro invocation
マクロ展開時の実引数が足りません。

C5055 (W) Too many arguments in macro invocation
マクロ展開時の実引数が多すぎます。

C5056 (E) Operand of sizeof may not be a function
sizeof 演算のオペランドに関数を指定できません。

C5057 (E) This operator is not allowed in a constant expression
この演算子は定数式中に指定できません。

C5058 (E) This operator is not allowed in a preprocessing expression
この演算子はプリプロセッサの式中で指定できません。

C5059 (E) Function call is not allowed in a constant expression
定数式中で関数呼び出しはできません。

C5060 (E) This operator is not allowed in an integral constant expression
この演算子は汎整数型定数式中で指定できません。

C5061 (W) Integer operation result is out of range
整数演算の結果が値の範囲を超えました。オーバフローした上位ビットを無視した値を仮定します。

C5062 (W) Shift count is negative
シフトカウントが負の値です。指定された通りに演算します。

C5063 (W) Shift count is too large
シフトカウントが有効ビット数を超えています。指定された通りに演算します。

C5064 (W) Declaration does not declare anything
宣言を指定するシンボルがありません。宣言を無視します。

C5065 (E) Expected a ";"
";"が必要です。

C5066 (E) Enumeration value is out of "int" range
enum 型メンバの値が int 型の範囲を超えました。

C5067 (E) Expected a "}"
"}"が必要です。

C5068 (W) Integer conversion resulted in a change of sign
符号変換を伴った整数型変換が実施されました。ビット列をそのまま設定します。

C5069 (W) Integer conversion resulted in truncation
上位バイト側を切り捨てる整数型変換が実施されました。切り捨て後の値を設定します。

C5070 (E) Incomplete type is not allowed
不完全型が指定されています。

C5071 (E) Operand of sizeof may not be a bit field
sizeof 演算子のオペランドにビットフィールドが指定されています。

C5075 (E) Operand of "*" must be a pointer
*演算子のオペランドの型がポインタ型ではありません。

C5077 (E) This declaration has no storage class or type specifier
記憶クラスまたは型の指定がありません。

C5079 (E) Expected a type specifier
型指定子が必要です。

C5080 (E) A storage class may not be specified here
ここでは記憶クラスを指定することはできません。

C5081 (E) More than one storage class may not be specified
記憶クラスを複数指定することはできません。

C5083 (W) Type qualifier specified more than once
const/volatile 限定子を複数指定しています。余分な指定を無視します。

C5084 (E) Invalid combination of type specifiers
型の組み合わせが正しくありません。

C5085 (E) Invalid storage class for a parameter
仮引数に不当な記憶クラスを指定しています。

C5086 (E) Invalid storage class for a function
関数に不当な記憶クラスを指定しています。

C5087 (E) A type specifier may not be used here
型を指定することはできません。

C5088 (E) Array of functions is not allowed
関数を要素とする配列は指定できません。

C5089 (E) Array of void is not allowed
void 型を要素とする配列は指定できません。

C5090 (E) Function returning function is not allowed
関数型をリターン型とする関数は指定できません。

C5091 (E) Function returning array is not allowed
配列をリターン型とする関数は指定できません。

C5093 (E) Function type may not come from a typedef
typedef 宣言された関数型を使用することはできません。

C5094 (E) The size of an array must be greater than zero
配列のサイズは0より大きな値でなければなりません。

C5095 (E) Array is too large
配列のサイズが大きすぎます。

C5097 (E) A function may not return a value of this type
関数はこの型の値を返すことができません。

C5098 (E) An array may not have elements of this type
配列はこの型を要素とすることができません。

C5100 (E) Duplicate parameter name
仮引数の名前が重複しています。

C5101 (E) "名前" has already been declared in the current scope
同一スコープ内にすでに"名前"の宣言が存在します。

C5103 (E) Class is too large
クラスのサイズが大きすぎます。

C5105 (E) Invalid size for bit field
ビットフィールドのサイズが不正です。

C5106 (E) Invalid type for a bit field
ビットフィールドの型が不正です。

C5107 (E) Zero-length bit field must be unnamed
長さ0のビットフィールドには名前をつけられません。

C5108 (W) Signed bit field of length 1
符号付整数型の長さ 1 のビットフィールドが指定されています。指定された型で処理します。

C5109 (E) Expression must have (pointer-to-) function type
式は関数型へのポインタ型でなければなりません。

C5110 (E) Expected either a definition or a tag name
宣言の定義またはタグ名が必要です。

C5111 (I) Statement is unreachable
実行されない文です。最適化により削除される可能性があります。

C5112 (E) Expected "while"
while キーワードが必要です。

C5114 (E) Entity-kind "名前" was referenced but not defined
参照される"名前"の定義がありません。

C5115 (E) A continue statement may only be used within a loop
continue 文はループの中で有効です。

C5116 (E) A break statement may only be used within a loop or switch
break 文はループまたは switch 文の中で有効です。

C5117 (W) Non-void entity-kind "名前" should return a value
void 型でない関数がリターン値を返しません。リターン値は不定です。

C5118 (E) A void function may not return a value
void 型を返す関数はリターン値を返すことはできません。

C5119 (E) Cast to type "型" is not allowed
"型"へのキャストは指定できません。

C5120 (E) Return value type does not match the function type
リターン値と関数の型が合いません。

C5121 (E) A case label may only be used within a switch
case ラベルを switch 文以外で使用しています。

C5122 (E) A default label may only be used within a switch
default ラベルを switch 文以外で使用しています。

C5123 (E) Case label value has already appeared in this switch
case ラベルの値がすでに switch 文の中に存在します。

C5124 (E) Default label has already appeared in this switch
default ラベルの値がすでに switch 文の中に存在します。

C5125 (E) Expected a "("
"("が必要です。

C5126 (E) Expression must be an lvalue
式は左辺値でなければなりません。

C5127 (E) Expected a statement
文が必要です。

C5128 (I) Loop is not reachable from preceding code
実行されないループ文です。

C5129 (E) A block-scope function may only have extern storage class
ブロック内で宣言された関数は extern 記憶クラスでなければなりません。

C5130 (E) Expected a "{"
"{"が必要です。

C5131 (E) Expression must have pointer-to-class type
式はクラスへのポインタ型でなければなりません。

C5132 (E) Expression must have pointer-to-struct-or-union type
式は構造体または共用体へのポインタ型でなければなりません。

C5133 (E) Expected a member name
メンバ名が必要です。

C5134 (E) Expected a field name
フィールド名が必要です。

C5135 (E) Entity-kind "名前" has no member "メンバ名"
"名前"は"メンバ名"を持ちません。

C5136 (E) Entity-kind "名前" has no field "フィールド名"
"名前"は"フィールド名"を持ちません。

C5137 (E) Expression must be a modifiable lvalue
式は修正可能な左辺値でなければなりません。

C5139 (E) Taking the address of a bit field is not allowed
ビットフィールドのアドレスを参照することはできません。

C5140 (E) Too many arguments in function call
関数呼び出しの実引数の数が多すぎます。

C5142 (E) Expression must have pointer-to-object type
式はオブジェクトへのポインタ型でなければなりません。

C5143 (F) Program too large or complicated to compile
プログラムが大きすぎるかまたは複雑すぎます。

C5144 (E) A value of type "型 1" cannot be used to initialize an entity of type "型 2"
初期値の"型 1"と変数の"型 2"が異なります。

C5145 (E) Entity-kind "名前" may not be initialized
"名前"を初期化することはできません。

C5146 (E) Too many initializer values
初期値の数が多すぎます。

C5147 (E) Declarationis incompatible with "名前"(declared at line "行番号")
前に宣言した"名前"の型が合致しません。

C5148 (E) Entity-kind "名前" has already been initialized
すでに"名前"の初期値が設定されています。

C5149 (E) A global-scope declaration may not have this storage class
大域的なスコープでの宣言にはこの記憶クラスを指定できません。

C5150 (E) A type name may not be redeclared as a parameter
型名を仮引数で再宣言することはできません。

C5151 (E) A typedef name may not be redeclared as a parameter
型名を仮引数で再宣言することはできません。

C5153 (E) Expression must have class type
式はクラス型でなければなりません。

C5154 (E) Expression must have struct or union type
式は構造体または共用体型でなければなりません。

C5157 (E) Expression must be an integral constant expression
式は整数型の定数式でなければなりません。

C5158 (E) Expression must be an lvalue or a function designator
式は左辺値または関数名でなければなりません。

C5159 (E) Declaration is incompatible with previous "名前" (declared at line "行番号")
前に使用した"名前"の型と合致しません。

C5160 (E) Name conflicts with previously used external name "名前"
前に使用した外部名"名前"と名前が重なります。

C5161 (I) Unrecognized #pragma
認識できない#pragma 指定があります。#pragma 指定を無視します。

C5163 (F) Could not open temporary file "名前"
テンポラリファイル"名前"をオープンできませんでした。コンパイラの環境設定やホスト環境のファイルシステム異常がないか確認ください。

C5164 (F) Name of directory for temporary files is too long ("名前")
テンポラリファイルの"名前"が長すぎます。

C5165 (E) Too few arguments in function call
関数呼び出しの実引数の数が足りません。

C5166 (E) Invalid floating constant
浮動小数点定数の指定が不正です。

C5167 (E) Argument of type "型1" is incompatible with parameter of type "型2"
実引数の型"型1"と仮引数の型"型2"とが合致しません。

C5168 (E) A function type is not allowed here
関数型は許されません。

C5169 (E) Expected a declaration
宣言が必要です。

C5170 (W) Pointer points outside of underlying object
ポインタが指している領域がオブジェクトの範囲を超えています。

C5171 (E) Invalid type conversion
キャストの型が不正です。

C5172 (I) External/internal linkage conflict with previous declaration
前の宣言と外部/内部リンケージが異なります。内部リンケージが仮定されます

C5173 (E) Floating-point value does not fit in required integral type
浮動小数点数型の値を汎整数型に変換するときに値の範囲を超えました。

C5174 (I) Expression has no effect
効果のない式です。最適化で削除される可能性があります。

C5175 (W) Subscript out of range
配列のインデックスが範囲を超えています。指定されたインデックスで処理を継続します。

C5177 (W) Entity-kind "entity" was declared but never referenced
参照されない宣言があります。

C5179 (W) Right operand of "%" is zero
%演算子の右辺が値０です。指定された式で評価します。

C5182 (F) Could not open source file "名前" (no directories in search list)
ファイル"名前"をオープンできませんでした。ディレクトリが存在するかどうか確認ください。

C5183 (E) Type of cast must be integral
キャストの型は汎整数型でなければなりません。

C5184 (E) Type of cast must be arithmetic or pointer
キャストの型は算術型またはポインタ型でなければなりません。

C5185 (I) Dynamic initialization in unreachable code
初期化式は実行されません。実行時に初期値は設定されません。

C5186 (W) Pointless comparison of unsigned integer with zero
0 と符号無し整数の無意味な比較をしています。指定された通りに式を評価します。

C5187 (I) Use of "=" where "==" may have been intended
"=="が意図される式で"="が使われています。指定された通りに式を評価します。

C5189 (F) Error while writing "ファイル名" file
ファイルの書き込みに失敗しました。

C5191 (W) Type qualifier is meaningless on cast type
キャストの型に意味のない型限定子を指定しています。指定された型を無視します。

C5192 (W) Unrecognized character escape sequence
認識できないエスケープシーケンス文字を指定しています。値をそのまま使用します。

C5193 (I) Zero used for undefined preprocessing identifier
プリプロセッサ文の式評価に値０が使われました。指定された通りに式を評価します。

C5219 (F) Error while deleting file "ファイル名"
ファイル"ファイル名"を削除することができません。

C5221 (W) Floating-point value does not fit in required floating-point type
浮動小数点型を要求された浮動小数点型に変換できません。無限大の値とみなします。

C5224 (W) The format string requires additional arguments
フォーマット文字列で要求する引数より実引数の数が足りません。

C5225 (W) The format string ends before this argument
フォーマット文字列が要求する引数より実引数の数が多すぎます。

C5226 (W) Invalid format string conversion
フォーマット変換の形式が実引数の型と異なります。

C5229 (W) Bit field cannot contain all values of the enumerated type
ビットフィールドが enum 型全ての値を保持できません。結果は切り捨てられます。

C5235 (E) Variable "名前" was declared with a never-completed type
変数"名前"が不完全型のまま宣言されました。

C5236 (W)(I) Controlling expression is constant
制御式が定数です(I)。制御式がアドレス定数です(W)。指定された通りに式を評価します。

C5237 (I) Selector expression is constant
switch 文の制御式が定数です。

C5238 (E) Invalid specifier on a parameter
引数宣言で不正な指定子を使用しています。

C5239 (E) Invalid specifier outside a class declaration
クラス宣言外で不正な指定子を使用しています。

C5240 (E) Duplicate specifier in declaration
一つの宣言内で指定子を重複して使用しています。

C5241 (E) A union is not allowed to have a base class
union 型は基底クラスを持つことはできません。

C5242 (E) Multiple access control specifiers are not allowed
アクセス指定子が重複して使われています。

C5243 (E) Class or struct definition is missing
class 定義の括弧の対応がとれません。

C5244 (E) Qualified name is not a member of class "型" or its base classes
限定名がクラスまたは基底クラスのメンバの"型"でありません。

C5245 (E) A nonstatic member reference must be relative to a specific object
非静的メンバの参照がオブジェクトに対応していません。

C5246 (E) A nonstatic data member may not be defined outside its class
非静的データメンバはクラス外で定義できません。

C5247 (E) Entity-kind "名前" has already been defined
"名前"はすでに定義されています。

C5248 (E) Pointer to reference is not allowed
リファレンス型へのポインタ型は許されません

C5249 (E) Reference to reference is not allowed
リファレンス型へのリファレンス型は許されません。

C5250 (E) Reference to void is not allowed
void 型へのリファレンス型は許されません。

C5251 (E) Array of reference is not allowed
リファレンス型の配列は許されません。

C5252 (E) Reference entity-kind "名前" requires an initializer
リファレンス型の定義"名前"には初期値が必要です。

C5253 (E) Expected a ","
カンマ","が必要です。

C5254 (E) Type name is not allowed
型名は許されません。

C5255 (E) Type definition is not allowed
型の定義は許されません。

C5256 (E) Invalid redeclaration of type name "名前" (declared at line
"行番号")
型名"名前"を再定義することはできません。

C5257 (E) Const entity-kind "名前" requires an initializer
const 型の定義"名前"には初期値が必要です。

C5258 (E) "this" may only be used inside a nonstatic member function
"this"が非静的メンバ関数以外で使われています。

C5259 (E) Constant value is not known
const 型の値が不明です。

C5261 (I) Access control not specified ("名前" by default)
基底クラスのアクセス制御指定がありません。アクセス制御指定"名前"が仮定されます。

C5262 (E) Not a class or struct name
基底クラスで指定されたクラスまたは構造体がありません。

C5263 (E) Duplicate base class name
基底クラスを二重に指定しています。

C5264 (E) Invalid base class
基底クラスが不正です。

C5265 (E) Entity-kind "名前" is inaccessible
"名前"をアクセスすることはできません。

C5266 (E) "名前" is ambiguous
指定された"名前"があいまいです。

C5269 (E) Implicit conversion to inaccessible base class "型" is not allowed
アクセス不可能なクラスへの暗黙の型変換は許されません。

C5274 (E) Improperly terminated macro invocation
マクロ呼び出しの途中でファイルが終了しました。

C5276 (E) Name followed by "::" must be a class or namespace name
::に続く名前はクラス名または namespace 名でなければなりません。

C5277 (E) Invalid friend declaration
フレンド宣言の指定が正しくありません。

C5278 (E) A constructor or destructor may not return a value
コンストラクタやデストラクタはリターン値を持てません。

C5279 (E) Invalid destructor declaration
デストラクタの宣言が正しくありません。

C5280 (E)(W) Declaration of a member with the same name as its class
クラス名と同じ名前のメンバ名を宣言しています。
(W) 非 static 変数名
(E) static 変数名,typedef 名,enum メンバなど

C5281 (E) Global-scope qualifier (leading "::") is not allowed
グローバルなスコープ決定演算子は許されません。

C5282 (E) The global scope has no "名前"
"名前"がグローバルなスコープに宣言されていません。

C5283 (E) Qualified name is not allowed
限定名は許されません。

C5284 (W) NULL reference is not allowed
NULL へのリファレンスは許されません。指定された通りに式を評価します。

C5285 (E) Initialization with "{...}" is not allowed for object of type "型"
"型"のオブジェクトに{ }形式の初期化は許されません。

C5286 (E) Base class "type" is ambiguous
基底クラスの型があいまいです。

C5287 (E) Derived class "type" contains more than one instance of class "型"
派生型が複数の同一クラス"型"を含みます。

C5288 (E) Cannot convert pointer to base class "型1" to pointer to derived class "型2" -- base class is virtual
仮想基底クラス"型1"のポインタ型を派生クラス"型2"のポインタ型に変換することはできません。

C5289 (E) No instance of constructor "名前" matches the argument list
コンストラクタ"名前"の引数が一致しません。

C5290 (E) Copy constructor for class "型" is ambiguous
クラス"型"のコピーコンストラクタがあいまいです。

C5291 (E) No default constructor exists for class "型"
クラス"型"のデフォルトコンストラクタは存在しません。

C5292 (E) "名前" is not a nonstatic data member or base class of class "型"
"名前" が非静的データメンバまたは基底クラス"型"でありません。

C5293 (E) Indirect nonvirtual base class is not allowed
仮想でない間接基底クラスは許されません。

C5294 (E) Invalid union member -- class "型" has a disallowed member function
union メンバに指定できないクラス"型"のメンバ関数があります。

C5297 (E) Expected an operator
演算子が必要です。

C5298 (E) Inherited member is not allowed
継承されたメンバを使用することはできません。

C5299 (E) Cannot determine which instance of entity-kind "名前" is intended
オーバーロード関数の"名前"を決定できません。

C5300 (E) A pointer to a bound function may only be used to call the function
メンバ関数へのポインタを関数呼び出し以外に使用しています。

C5302 (E) Entity-kind "名前" has already been defined
関数"名前"はすでに定義されています。

C5304 (E) No instance of entity-kind "名前" matches the argument list
関数"名前"の引数が一致しません。

C5305 (E) Type definition is not allowed in function return type declaration
関数のリターン型の宣言で型の定義をすることはできません。

C5306 (E) Default argument not at end of parameter list
デフォルト引数の宣言がパラメタリストの最後ではありません。

C5307 (E) Redefinition of default argument
デフォルト引数を再定義しています。

C5308 (E) More than one instance of entity-kind "名前" matches the argument list:
引数リストが一致するためオーバーロード関数"名前"があいまいです。

C5309 (E) More than one instance of constructor "名前" matches the argument list:
引数リストが一致するためコンストラクタ"名前"があいまいです。

C5310 (E) Default argument of type "型1" is incompatible with parameter of type "型2"
デフォルト値の"型1"が引数の"型2"に合致しません。

C5311 (E) Cannot overload functions distinguished by return type alone
リターン型が異なる関数をオーバーロードすることはできません。

C5312 (E) No suitable user-defined conversion from "型1" to "型2" exists
適切な利用者定義変換"型1"から"型2"が存在しません。

C5313 (E) Type qualifier is not allowed on this function
関数に型限定子(const, volatile)を指定することはできません。

C5314 (E) Only nonstatic member functions may be virtual
静的メンバ関数に virtual を指定しています。

C5315 (E) The object has type qualifiers that are not compatible with the member function
オブジェクトの型限定子(const, volatile)がメンバ関数の型限定子と合致しません。

C5316 (E) Program too large to compile (too many virtual functions)
仮想関数の数が多すぎます。

C5317 (E) Return type is not identical to nor covariant with return type
"型" of overridden virtual function entity-kind "名前"
仮想関数"名前"のリターン型"型"が異なります。

C5318 (E) Override of virtual entity-kind "名前" is ambiguous
仮想関数"名前"の置き換えがあいまいです。

C5319 (E) Pure specifier ("= 0") allowed only on virtual functions
純粋指定子"=0" を仮想関数以外に指定しています。

C5320 (E) Badly-formed pure specifier (only "= 0" is allowed)
純粋指定子の形式が正しくありません。"=0"だけが許されます。

C5321 (E) Data member initializer is not allowed
データメンバの初期化指定が正しくありません。

C5322 (E) Object of abstract class type "型" is not allowed:
抽象クラス"型"のオブジェクトは定義できません。

C5323 (E) Function returning abstract class "型" is not allowed:
抽象クラス"型"を返す関数は定義できません。

C5324 (I) Duplicate friend declaration
フレンド宣言が重複して指定されています。

C5325 (E) Inline specifier allowed on function declarations only
inline 指定子は関数宣言でのみ有効です。

C5326 (E) "inline" is not allowed
inline 指定は許されません。

C5327 (E) Invalid storage class for an inline function
inline 関数の記憶クラスが不正です。

C5328 (E) Invalid storage class for a class member
クラスメンバの記憶クラスが不正です。

C5329 (E) Local class member entity-kind "名前" requires a definition
局所クラスメンバ"名前"の定義がありません。

C5330 (E) Entity-kind "名前" is inaccessible
"名前"をアクセスできません。

C5332 (E) Class "type" has no copy constructor to copy a const object
クラス"型"に const 型オブジェクトをコピーするコピーコンストラクタがありません。

C5333 (E) Defining an implicitly declared member function is not allowed
暗黙宣言されたメンバ関数を定義することはできません。

C5334 (E) Class "型" has no suitable copy constructor
クラス"型"に適切なコピーコンストラクタが存在しません。

C5335 (E) Linkage specification is not allowed
リンケージ指定子を指定することはできません。

C5336 (E) Unknown external linkage specification
認識できないリンケージ指定が指定されました。

C5337 (E) Linkage specification is incompatible with previous "名前"
(declared at line "行番号")
前に指定されたリンケージ指定子"名前"と合致しません。

C5338 (E) More than one instance of overloaded function "名前" has "C" linkage
C リンケージを持ったオーバーロード関数"名前"が複数あります。

C5339 (E) Class "型" has more than one default constructor
クラス"型"は複数のデフォルトコンストラクタを持っています。

C5341 (E) "operator 演算子" must be a member function
演算子関数"演算子"はメンバ関数でなければなりません。

C5342 (E) Operator may not be a static member function
静的メンバ関数の演算子関数は許されません。

C5343 (E) No arguments allowed on user-defined conversion
利用者定義変換に引数は許されません。

C5344 (E) Too many parameters for this operator function
演算子関数の引数の数が多すぎます。

C5345 (E) Too few parameters for this operator function
演算子関数の引数の数が足りません。

C5346 (E) Nonmember operator requires a parameter with class type
メンバ関数でない演算子関数はクラス型を引数に持つ必要があります。

C5347 (E) Default argument is not allowed
デフォルト引数は許されません。

C5348 (E) More than one user-defined conversion from "型 1" to "型 2" applies:
"型 1"から"型 2"への利用者定義型変換があいまいです。

C5349 (E) No operator "演算子" matches these operands
演算子関数"演算子"のオペランドが一致しません。

C5350 (E) More than one operator "演算子" matches these operands:
演算子関数"演算子"のオペランドがあいまいです。

C5351 (E) First parameter of allocation function must be of type "size_t"
operator new の第一引数は size_t 型でなければなりません。

C5352 (E) Allocation function requires "void *" return type
operator new のリターン型は void *型でなければなりません。

C5353 (E) Deallocation function requires "void" return type
operator delete のリターン型は void 型でなければなりません。

C5354 (E) First parameter of deallocation function must be of type "void *"
operator delete の第一引数は void *型でなければなりません。

C5356 (E) Type must be an object type
型はオブジェクト型でなければなりません。

C5357 (E) Base class "type" has already been initialized
基底クラスはすでに初期化されています。

C5359 (E) Entity-kind "名前" has already been initialized
"名前"はすでに初期化されています。

C5360 (E) Name of member or base class is missing
メンバ名または基底クラスに誤りがあります。

C5363 (E) Invalid anonymous union -- nonpublic member is not allowed
無名 union のメンバが公開メンバでありません。

C5364 (E) Invalid anonymous union -- member function is not allowed
無名 union にメンバ関数は許されません。

C5365 (E) Anonymous union at global or namespace scope must be declared static
グローバルまたは namespace スコープの無名 union は static 宣言が必要です。

C5366 (E) Entity-kind "名前" provides no initializer for:
"名前"に初期化指定はできません。

C5367 (E) Implicitly generated constructor for class "型" cannot initialize:
暗黙に生成されたクラス"型"のコンストラクタを初期化することはできません。

```
C5368 (E) Entity-kind "名前" defines no constructor to initialize the
          following:
```
"名前"は初期化のためのコンストラクタを定義していません。

```
C5369 (E) Entity-kind "名前" has an uninitialized const or reference member
```
"名前"の const またはリファレンスメンバが初期化されていません。

```
C5370 (W) Entity-kind "名前" has an uninitialized const field
```
"名前"の const フィールドが初期化されていません。

```
C5371 (E) Class "型" has no assignment operator to copy a const object
```
const オブジェクトをコピーするクラス"型"の代入演算子関数が定義されていません。

```
C5372 (E) Class "型" has no suitable assignment operator
```
クラス"型"に適当な代入演算が定義されていません。

```
C5373 (E) Ambiguous assignment operator for class "型"
```
クラス"型"の代入演算子関数があいまいです。

```
C5375 (E) Declaration requires a typedef name
```
typedef 名の宣言が必要です。

```
C5377 (E) "virtual" is not allowed
```
virtual を指定することはできません。

```
C5378 (E) "static" is not allowed
```
static を指定することはできません。

```
C5380 (E) Expression must have pointer-to-member type
```
式はメンバへのポインタ型でなければなりません。

```
C5381 (I) Extra ";" ignored
```
余分な";"を無視します。

```
C5382 (W) Nonstandard member constant declaration (standard form is a static
          const integral member)
```
const メンバの宣言が標準形式でありません。初期化は無効です。

```
C5384 (E) No instance of overloaded "名前" matches the argument list
```
オーバーロード関数"名前"の引数リストが一致しません。

```
C5386 (E) No instance of entity-kind "名前" matches the required type
```
要求される型のオーバーロード関数"名前"がありません。

```
C5388 (E) "operator->" for class "型 1" returns invalid type "型 2"
```
クラス"型 1"の operator->演算関数のリターン型"型 2"が正しくありません。

C5389 (E) A cast to abstract class "型" is not allowed:
抽象クラス"型"へのキャストは許されません。

C5391 (E) A new-initializer may not be specified for an array
配列を new によって初期化することはできません。

C5392 (E) Member function "名前" may not be redeclared outside its class
メンバ関数"名前"がクラスの外側で再宣言されました。

C5393 (E) Pointer to incomplete class type is not allowed
不完全クラスへのポインタ型は許されません。

C5394 (E) Reference to local variable of enclosing function is not allowed
ローカルクラスを囲む関数のローカル変数へのリファレンスは許されません。

C5397 (E) Implicitly generated assignment operator cannot copy:
暗黙に生成された代入演算子関数がオブジェクトを正しくコピーすることができません。

C5399 (I) Entity-kind "名前" has an operator newxxxx() but no default operator deletexxxx()
"名前"が operator new を持ちますがデフォルトの operator delete を持ちません。

C5400 (I) Entity-kind "名前" has a default operator deletexxxx() but no operator newxxxx()
"名前"がデフォルトの operator delete を持ちますが operator new を持ちません。

C5401 (E) Destructor for base class "型" is not virtual
基底クラス"型"のデストラクタが virtual でありません。

C5403 (E) Entity-kind "名前" has already been declared
メンバ関数"名前"が再宣言されています。

C5404 (E) Function "main" may not be declared inline
main 関数を inline 宣言することはできません。

C5405 (E) Member function with the same name as its class must be a constructor
クラス名と同じ名前のメンバ関数はコンストラクタでなければなりません。

C5407 (E) A destructor may not have parameters
デストラクタは引数を持つことができません。

C5408 (E) Copy constructor for class "型1" may not have a parameter of type "型2"
クラス"型1"のコピーコンストラクタは"型2"の引数を持つことはできません。

C5409 (E) Entity-kind "名前" returns incomplete type "型"
関数"名前"のリターン型が不完全型"型"です。

C5410 (E) Protected entity-kind "名前" is not accessible through a "型" pointer or object
限定公開名"名前"は"型"へのポインタやオブジェクトを経由してアクセスすることはできません。

C5411 (E) A parameter is not allowed
仮引数は許されません。

C5412 (E) An "asm" declaration is not allowed here
asm 宣言は許されません。

C5413 (E) No suitable conversion function from "型1" to "型2" exists
"型1"から"型2"への適切な変換関数が存在しません。

C5414 (W) Delete of pointer to incomplete class
不完全型クラスへのポインタは削除されました。

C5415 (E) No suitable constructor exists to convert from "型1" to "型2"
"型1"から"型2"へ変換する適切なコンストラクタが存在しません。

C5416 (E) More than one constructor applies to convert from "型1" to "型2":
"型1"から"型2"へ変換するコンストラクタがあいまいです。

C5417 (E) More than one conversion function from "型1" to "型2" applies:
"型1"から"型2"への変換関数があいまいです。

C5418 (E) More than one conversion function from "型" to a built-in type applies:
"型"からビルトイン型への変換関数があいまいです。

C5424 (E) A constructor or destructor may not have its address taken
コンストラクタまたはデストラクタのアドレスを参照することはできません。

C5427 (E) Qualified name is not allowed in member declaration
限定名をメンバ宣言のなかで使用できません。

C5429 (E) The size of an array in "new" must be non-negative
new で指定された配列のサイズに負の値は許されません。

C5430 (W) Returning reference to local temporary
関数内にローカルな領域のリファレンスをリターン値にしています。

C5432 (E) "enum" declaration is not allowed
enum 型宣言は許されません。

C5433 (E) Qualifiers dropped in binding reference of type "型 1" to initializer of type "型 2"
const/volatile 限定の型"型 2"が参照型"型 1"の初期値に指定されました。

C5434 (E) A reference of type "型 1" (not const-qualified) cannot be initialized with a value of type "型 2"
const 型修飾されない型"型 1"へのリファレンスを"型 2"の値で初期化できません。

C5435 (E) A pointer to function may not be deleted
関数へのポインタを削除することはできません。

C5436 (E) Conversion function must be a nonstatic member function
変換関数は非静的メンバ関数でなければなりません。

C5437 (E) Template declaration is not allowed here
このスコープ内でテンプレート宣言は許されません。

C5438 (E) Expected a "<"
"<"が必要です。

C5439 (E) Expected a ">"
">"が必要です。

C5440 (E) Template parameter declaration is missing
テンプレートの引数宣言が正しくありません。

C5441 (E) Argument list for entity-kind "名前" is missing
テンプレート"名前"の実引数リストが正しくありません。

C5442 (E) Too few arguments for entity-kind "名前"
テンプレート"名前"の実引数が足りません。

C5443 (E) Too many arguments for entity-kind "名前"
テンプレートの実引数が多すぎます。

C5445 (E) Entity-kind "名前 1" is not used in declaring the parameter types of entity-kind "名前 2"
テンプレート"名前 1"の引数"名前 2"が使用されません。

C5449 (E) More than one instance of entity-kind "名前" matches the required type
オーバーロード関数"名前"があいまいです。

C5452 (E) Return type may not be specified on a conversion function
変換関数のリターン型が指定されていません。

C5456 (E) Excessive recursion at instantiation of entity-kind "名前"
テンプレート"名前"のインスタンスが再帰的に生成されます。

C5457 (E) "名前" is not a function or static data member
"名前"が関数または静的データメンバではありません。

C5458 (E) Argument of type "型1" is incompatible with template parameter
of type "型2"
実引数の型"型1"がテンプレートの引数"型2"に合致しません。

C5459 (E) Initialization requiring a temporary or conversion is not allowed
初期化にテンポラリや変換を要求することは許されません。

C5461 (E) Initial value of reference to non-const must be an lvalue
const 型を持たないリファレンスの初期値は左辺値でなければなりません。

C5463 (E) "template" is not allowed
"template"指定は許されません。

C5464 (E) "型" is not a class template
"型"がクラステンプレートではありません。

C5466 (E) "main" is not a valid name for a function template
"main"は関数テンプレートの名前に使用できません。

C5467 (E) Invalid reference to entity-kind "名前" (union/nonunion mismatch)
"名前"の参照が不正です。

C5468 (E) A template argument may not reference a local type
テンプレートの実引数はローカルな型を参照できません。

C5469 (E) Tag kind of "名前1" is incompatible with declaration of
entity-kind "名前2" (declared at line "行番号")
タグ名"名前1"の種類と"名前2"の宣言が合致しません。

C5470 (E) The global scope has no tag named "名前"
グローバルスコープにタグ名"名前"がありません。

C5471 (E) Entity-kind "名前1" has no tag member named "名前2"
"名前1"はタグメンバ"名前2"を持ちません。

C5473 (E) Entity-kind "名前" may be used only in pointer-to-member
declaration
typedef 名"名前"はメンバへのポインタ型の宣言の中で使用されなければなりません。

C5475 (E) A template argument may not reference a non-external entity
テンプレートの実引数は外部名以外を参照できません。

C5476 (E) Name followed by "::~" must be a class name or a type name
::~に続く名前はクラス名または型名でなければなりません。

C5477 (E) Destructor name does not match name of class "型"
クラス名"型"とデストラクタ名が合致しません。

C5478 (E) Type used as destructor name does not match type "型"
デストラクタ名で使われた型と"型"が合致しません。

C5479 (I) Entity-kind "名前" redeclared "inline" after being called
関数が呼ばれたあとに inline"名前"を宣言しています。以降 inline 指定を有効にします。

C5481 (E) Invalid storage class for a template declaration
テンプレート宣言の記憶クラス指定が正しくありません。

C5484 (E) Invalid explicit instantiation declaration
テンプレートの実引数が不正です。

C5485 (E) Entity-kind "名前" is not an entity that can be instantiated
テンプレート"名前"を実体化できません。

C5486 (E) Compiler generated entity-kind "entity" cannot be explicitly instantiated
コンパイラが生成した関数を実体化することはできません。

C5487 (E) Inline entity-kind "名前" cannot be explicitly instantiated
インライン関数"名前"を実体化することはできません。

C5488 (E) Pure virtual entity-kind "名前" cannot be explicitly instantiated
純粋仮想関数"名前"を実体化することはできません。

C5489 (E) Entity-kind "名前" cannot be instantiated -- no template definition was supplied
テンプレート定義がないため "名前"を実体化することはできません。

C5490 (E) Entity-kind "名前" cannot be instantiated -- it has been explicitly specialized
"名前"を実体化することはできません。

C5493 (E) No instance of entity-kind "名前" matches the specified type
オーバーロード関数"名前"と指定された型が合致しません。

C5496 (E) Template parameter "名前" may not be redeclared in this scope
テンプレート引数"名前"がスコープ内で再宣言されています。

C5497 (W) Declaration of "名前" hides template parameter
"名前"の宣言はテンプレート引数を隠蔽します。

C5498 (E) Template argument list must match the parameter list
テンプレート実引数と仮引数が合致しません。

C5499 (E) Conversion function to convert from "型1" to "型2" is not allowed
"型1"から"型2"への変換関数は許されません。

C5500 (E) Extra parameter of postfix "operatorxxxx" must be of type "int"
後置演算関数の第2引数の型は int 型でなければなりません。

C5501 (E) An operator name must be declared as a function
演算子名は関数として宣言しなければなりません。

C5502 (E) Operator name is not allowed
演算子名は許されません。

C5503 (E) Entity-kind "名前" cannot be specialized in the current scope
スコープ内で"名前"があいまいです。

C5505 (E) Too few template parameters -- does not match previous declaration
テンプレートの引数が足りません。

C5506 (E) Too many template parameters -- does not match previous declaration
テンプレートの引数が多すぎます。

C5507 (E) Function template for operator delete(void *) is not allowed
operator delete(void *)の関数テンプレートは許されません。

C5508 (E) Class template and template parameter may not have the same name
クラステンプレートとテンプレートの引数が同じ名前です。

C5510 (E) A template argument may not reference an unnamed type
テンプレートの実引数が名前付けされていない型を参照しています。

C5511 (E) Enumerated type is not allowed
enum 型は許されません。

C5512 (W) Type qualifier on a reference type is not allowed
リファレンス型に const/volatile 修飾を指定することはできません。

C5513 (E) A value of type "型1" cannot be assigned to an entity of type "型2"
型不一致のため"型1"の値を"型2"の実体に代入することができません。

C5514 (W) Pointless comparison of unsigned integer with a negative constant
負の定数と符号無し整数を比較しています。

C5515 (E) Cannot convert to incomplete class "型"
不完全型"型"への型変換はできません。

C5516 (E) Const object requires an initializer
const 型のオブジェクトには初期値が必要です。

C5517 (E) Object has an uninitialized const or reference member
オブジェクトが未初期化の const 型メンバあるいはリファレンス型メンバを持ちます。

C5519 (E) Entity-kind "名前" may not have a template argument list
"名前"はテンプレート実引数を持つことができません。

C5520 (E) Initialization with "{...}" expected for aggregate object
集成型のオブジェクトは{...}の形式で初期化しなければなりません。

C5521 (E) Pointer-to-member selection class types are incompatible ("型1" and "型2")
メンバへのポインタ型のクラスの型が"型1"と"型2"で合致しません。

C5522 (W) Pointless friend declaration
自分自身へのフレンド宣言をしています。

C5526 (E) A parameter may not have void type
void 型の引数は指定できません。

C5529 (E) This operator is not allowed in a template argument expression
テンプレートの実引数式に指定された演算は許されません。

C5530 (E) Try block requires at least one handler
try 文に対応する catch 文がありません。

C5531 (E) Handler requires an exception declaration
catch 文の(...)には例外宣言が必要です。

C5532 (E) Handler is masked by default handler
デフォルトハンドラによってハンドラがマスクされました。

C5533 (E) Handler is potentially masked by previous handler for type "型"
"型"を持つ前のハンドラによってハンドラがマスクされる可能性があります。

C5534 (I) Use of a local type to specify an exception
ローカルな型を使用した例外処理が指定されています。

C5535 (I) Redundant type in exception specification
例外処理中に冗長な型の指定があります。

C5536 (E) Exception specification is incompatible with that of previous entity-kind "名前" (declared at line "行番号"):
例外処理指定が前の指定"名前"と合致しません。

C5540 (E) Support for exception handling is disabled
例外処理を行うオプション(exception)が指定されていません。

C5541 (W) Omission of exception specification is incompatible with previous entity-kind "名前" (declared at line "行番号")
例外処理の省略形が前の"名前"と合致しません。

C5542 (F) Could not create instantiation request file "名前"
テンプレートを実体化するのに使用するファイル"名前"を作成することができませんでした。

C5543 (E) Non-arithmetic operation not allowed in nontype template argument
対応するテンプレートの実引数に非算術型変換は許されません。

C5544 (E) Use of a local type to declare a nonlocal variable
ローカルでない変数にローカルな型を指定しています。

C5545 (E) Use of a local type to declare a function
関数宣言にローカルな型を指定しています。

C5546 (E) Transfer of control bypasses initialization of:
初期化処理が行われません。

C5548 (E) Transfer of control into an exception handler
例外ハンドラ処理が実行されます。

C5549 (I) Entity-kind "名前" is used before its value is set
"名前"に値を設定する前に使用しています。

C5550 (W) Entity-kind "名前" was set but never used
"名前"が使用されませんでした。

C5551 (E) Entity-kind "名前" cannot be defined in the current scope
"名前"はこのスコープ内で定義できません。

C5552 (W) Exception specification is not allowed
例外処理指定は許されません。例外処理を無効にします。

C5553 (W) External/internal linkage conflict for entity-kind "名前"
(declared at line "行番号")
"名前"の外部/内部リンケージ指定が衝突します。外部リンケージを設定します。

C5554 (W) Entity-kind "名前" will not be called for implicit or explicit conversions
変換関数"名前"は暗黙的にも明示的にも呼ばれることはありません。

C5555 (E) Tag kind of "名前" is incompatible with template parameter of type
"型"タグ"名前"の種類とテンプレートの引数の"型"が合致しません。

C5556 (E) Function template for operator new(size_t) is not allowed
operator new(size_t)の関数テンプレートは許されません。

C5558 (E) Pointer to member of type "型" is not allowed
メンバへのポインタ型"型"が誤っています。

C5559 (E) Ellipsis is not allowed in operator function parameter list
省略指定(...)は演算子関数の引数リストに指定できません。

C5598 (E) A template parameter may not have void type
テンプレートの引数に void 型は指定できません。

C5601 (E) A throw expression may not have void type
throw 式に void 型は指定できません。

C5603 (E) Parameter of abstract class type "型" is not allowed:
抽象クラス"型"の引数は許されません。

C5604 (E) Array of abstract class "型" is not allowed:
抽象クラス"型"の配列は許されません。

C5610 (W) Entity-kind "名前 1" does not match "名前 2" -- virtual function override intended?
"名前 1"と"名前 2"が一致しません。指定された通りに処理を継続します。

C5611 (W) Overloaded virtual function "名前 1" is only partially overridden in entity-kind "名前 2"
"名前 1"のオーバーロード仮想関数は"名前 2"の中で一部の仮想関数だけが置き換えの対象になります。指定された通りに処理を継続します。

C5612 (E) Specific definition of inline template function must precede its first use
インライン指定されたテンプレート関数は呼び出しの前に定義しなければなりません。

C5624 (E) "名前" is not a type name
"名前"は型の名前ではありません。

C5641 (F) "名前" is not a valid directory
"名前"が正しいディレクトリでありません。

C5642 (F) Cannot build temporary file name
コンパイラが使用するテンポラリファイルを作成できません。

C5656 (E) Transfer of control into a try block
外側のブロックから try ブロックに制御が移ります。

C5657 (W) Inline specification is incompatible with previous "名前"
(declared at line "行番号")
インライン指定が前の宣言"名前"と合致しません。

C5658 (E) Closing brace of template definition not found
テンプレート定義の閉じ括弧がありません。

C5660 (E) Invalid packing alignment value
pack の値が不正です。指定を無視します。

C5662 (W) Call of pure virtual function
純粋仮想関数が関数を呼び出しています。

C5663 (E) Invalid source file identifier string
#pragma 指定の構文に誤りがあります。

C5664 (E) A class template cannot be defined in a friend declaration
フレンド宣言内でクラステンプレートを定義することはできません。

C5673 (E) A reference of type "型1" cannot be initialized with a value of
type "型2"
const/volatile 型"型1"のリファレンスは"型2"の値で初期化できません。

C5674 (E) Initial value of reference to const volatile must be an lvalue
const/volatile 型のリファレンスの初期値は左辺値でなければなりません。

C5678 (I) Call of entity-kind "名前" (declared at line "行番号") cannot be
inlined
関数呼び出し"名前"がインライン展開されませんでした。

C5679 (I) Entity-kind "名前" cannot be inlined
関数"名前"はインライン展開されません。

C5693 (E) <typeinfo> must be included before typeid is used
typeid を使うためには<typeinfo>をインクルードしなければなりません。

C5694 (E) "名前" cannot cast away const or other type qualifiers
"名前"のキャストの結果 const などの属性がなくなります。

C5695 (E) The type in a dynamic_cast must be a pointer or reference to a complete class type, or void *
dynamic_cast の型は完全クラス型へのポインタ型またはリファレンス型か void *型でなければなりません。

C5696 (E) The operand of a pointer dynamic_cast must be a pointer to a complete class type
dynamic_cast ポインタのオペランドは完全クラス型へのポインタ型でなければなりません。

C5697 (E) The operand of a reference dynamic_cast must be an lvalue of a complete class type
dynamic_cast のリファレンスのオペランドは完全クラス型の左辺値でなければなりません。

C5698 (E) The operand of a runtime dynamic_cast must have a polymorphic class type
実行時 dynamic_cast のオペランドはポリモフィックなクラス型でなければなりません。

C5701 (E) An array type is not allowed here
配列型は許されません。

C5702 (E) Expected an "="
代入式が必要です。

C5703 (E) Expected a declarator in condition declaration
宣言子が必要です。

C5704 (E) "名前", declared in condition, may not be redeclared in this scope
このスコープ内で"名前"を再宣言することはできません。

C5705 (E) Default template arguments are not allowed for function templates
関数テンプレートにデフォルトの実引数を指定することはできません。

C5706 (E) Expected a "," or ">"
"," または ">" が必要です。

C5707 (E) Expected a template parameter list
テンプレートの引数リストが必要です。

C5708 (W) Incrementing a bool value is deprecated
bool 型の値をインクリメントしています。値をインクリメントして処理を継続します。

C5709 (E) bool type is not allowed
bool 型の値をデクリメントすることはできません。

C5710 (E) Offset of base class "名前1" within class "名前2" is too large
クラス"名前2"内の基底クラス"名前1"のサイズが大きすぎます。

C5711 (E) Expression must have bool type (or be convertible to bool)
式の型は bool 型か bool 型へ変換可能な型でなければなりません。

C5717 (E) The type in a const_cast must be a pointer, reference, or pointer to member to an object type
const_cast の型はポインタ型、リファレンス型またはメンバへのポインタ型でなければなりません。

C5718 (E) A const_cast can only adjust type qualifiers; it cannot change the underlying type
const_cast は const/volatile 以外の型を調整することはできません。

C5719 (E) mutable is not allowed
mutable の指定は許されません。

C5720 (W) Redeclaration of entity-kind "名前" is not allowed to alter its access
"名前"の再宣言でアクセス指定を変更することはできません。前の宣言のアクセス指定を有効にします。

C5722 (W) Use of alternative token "<:" appears to be unintended
2文字表記 "<:" が使用されました。"[" と解釈します。

C5723 (W) Use of alternative token "%:" appears to be unintended
2文字表記 "%:" が使用されました。"#" と解釈します。

C5724 (E) namespace definition is not allowed
namespace の定義はファイルスコープまたは namespace スコープ内で許されます。

C5725 (E) Name must be a namespace name
namespace の名前が正しくありません。

C5726 (E) Namespace alias definition is not allowed
namespace の別名定義はここでは許されません。

C5727 (E) namespace-qualified name is required
namespace の限定名が要求されます。

C5728 (E) A namespace name is not allowed
namespace 名は許されません。

C5730 (E) Entity-kind "名前" is not a class template
"名前"はクラステンプレートのメンバではありません。

C5732 (E) Allocation operator may not be declared in a namespace
operator new 関数が namespace 内で宣言されています。

C5733 (E) Deallocation operator may not be declared in a namespace
operator delete 関数が namespace 内で宣言されています。

C5734 (E) Entity-kind "名前 1" conflicts with using-declaration of
entity-kind "名前 2"
名前"名前 1"が using 宣言名"名前 2"と衝突します。

C5735 (E) Using-declaration of entity-kind "名前 1" conflicts with
entity-kind "名前 2" (declared at line "行番号")
using 宣言の名前が衝突します。

C5737 (W) Using-declaration ignored -- it refers to the current namespace
現在の namespace スコープの名前を using 宣言しています。using 宣言を無視します。

C5738 (E) A class-qualified name is required
クラスの限定名が要求されます。

C5741 (W) Using-declaration of entity-kind "名前" ignored
using 宣言"名前"は無効です。

C5742 (E) Entity-kind "名前 1" has no actual member "名前 2"
"名前 1"に"名前 2"のメンバは存在しません。

C5750 (E) Entity-kind "名前" (declared at line "行番号") was used before its
template was declared
"名前"はテンプレートが宣言される前に使われました。

C5751 (E) Static and nonstatic member functions with same parameter types
cannot be overloaded
同じ引数の型を持つ静的メンバ関数と非静的メンバ関数はオーバーロードすることはできません。

C5752 (E) No prior declaration of entity-kind "名前"
namespace テンプレート関数"名前"の宣言がありません。

C5753 (E) A template-id is not allowed
ここではテンプレート(template 名<template 実引数>)は許されません。

C5754 (E) A class-qualified name is not allowed
ここではクラス限定名は許されません。

C5755 (E) Entity-kind "名前" may not be redeclared in the current scope
このスコープ内で"名前"を再宣言することはできません。

C5756 (E) Qualified name is not allowed in namespace member declaration
namespace メンバの宣言で指定された限定名は許されません。

C5757 (E) Entity-kind "名前" is not a type name
"名前"は型名ではありません。

C5761 (E) Typename may only be used within a template
typename キーワードはテンプレート内でのみ使用できます。

C5766 (W) Exception specification for virtual entity-kind "名前 1" is incompatible with that of overridden entity-kind "名前 2"
仮想関数の例外指定"名前 1"が"名前 2"に合致しません。

C5767 (W) Conversion from pointer to smaller integer
ポインタをポインタサイズより小さい型に変換しています。

C5768 (W) Exception specification for implicitly declared virtual entity-kind "名前 1" is incompatible with that of overridden entity-kind "名前 2"
コンパイラが生成する暗黙の仮想関数"名前 1"の例外指定が"名前 2"に合致しません。

C5771 (E) "explicit" is not allowed
explicit はクラス宣言内のコンストラクタにのみ指定できます。

C5772 (E) Declaration conflicts with "名前" (reserved class name)
予約されたクラス名 type_info と衝突します。

C5773 (E) Only "()" is allowed as initializer for array entity-kind "名前"
配列"名前"の初期化指定が正しくありません。

C5774 (E) "virtual" is not allowed in a function template declaration
関数テンプレートに virtual 指定はできません。

C5775 (E) Invalid anonymous union -- class member template is not allowed
無名 union の指定が正しくありません。

C5776 (E) Template nesting depth does not match the previous declaration of entity-kind "名前"
テンプレートのパラメタのネストが前の宣言"名前"と合致しません。

C5777 (E) This declaration cannot have multiple "template <...>" clauses
この宣言に複数のテンプレート宣言はできません。

C5779 (E) "名前", declared in for-loop initialization, may not be redeclared in this scope
for 文の初期化式で宣言された"名前"をこのスコープ内で再宣言できません。

C5782 (E) Definition of virtual entity-kind "名前" is required here
ここに仮想関数の定義"名前"が要求されます。

C5784 (E) A storage class is not allowed in a friend declaration
フレンド宣言に記憶クラスを指定することはできません。

C5785 (E) Template parameter list for "名前" is not allowed in this declaration
この宣言内に"名前"のテンプレートの引数並びは許されません。

C5786 (E) entity-kind "名前" is not a valid member class or function template
"名前"は有効なメンバまたは関数テンプレートではありません。

C5787 (E) Not a valid member class or function template declaration
有効なメンバまたは関数テンプレート宣言ではありません。

C5788 (E) A template declaration containing a template parameter list may not be followed by an explicit specialization declaration
テンプレート関数の定義の後にテンプレート引数並びを含むテンプレート宣言は指定できません。

C5789 (E) Explicit specialization of entity-kind "名前 1" must precede the first use of entity-kind "名前 2"
明示的なテンプレートの実体の定義"名前 1"は最初のテンプレート"名前 2"を使用する前になければなりません。

C5790 (E) Explicit specialization is not allowed in the current scope
明示的なテンプレートの実体の定義はこのスコープでは許されません。

C5791 (E) Partial specialization of entity-kind "名前" is not allowed
テンプレート"名前"の部分的な定義は許されません。

C5792 (E) Entity-kind "名前" is not an entity that can be explicitly specialized
"名前"はテンプレートのインスタンスではありません。

C5793 (E) Explicit specialization of entity-kind "名前" must precede its first use
明示的なテンプレートの実体"名前"の定義は最初の使用より前になければなりません。

C5794 (W) Template parameter "テンプレート引数" may not be used in an elaborated type specifier
class 指定にテンプレート引数を使用することはできません。class 指定を無効にしてテンプレートを有効にします。

C5795 (E) Specializing entity-kind "名前" requires "template<>" syntax
"名前"のテンプレートの実体定義は tempalte<>形式が要求されます。

C5800 (E) This declaration may not have extern "C" linkage
この宣言は extern "C" リンケージを持つことはできません。

C5801 (E) "名前" is not a class or function template name in the current scope
"名前"はこのスコープ内ではクラステンプレートまたは関数テンプレートではありません。

C5802 (W) Specifying a default argument when redeclaring an unreferenced function template is nonstandard
未参照の関数テンプレートを再宣言するときにデフォルト引数を指定しています。
デフォルト引数を無視します。

C5803 (E) Specifying a default argument when redeclaring an already referenced function template is not allowed
すでに参照された関数テンプレートを再宣言するときにデフォルト引数を指定しています。

C5804 (E) Cannot convert pointer to member of base class "型1" to pointer to member of derived class "型2" -- base class is virtual
仮想基底クラス"型1"のメンバポインタを派生クラス"型2"のメンバポインタに変換することはできません。

C5805 (E) Exception specification is incompatible with that of entity-kind "名前" (declared at line "行番号"):
throw 例外指定は"名前"の例外指定と合致しません。

C5806 (W) Omission of exception specification is incompatible with entity-kind "名前" (declared at line "行番号")
throw 例外指定の省略は"名前"の例外指定と合致しません。"名前"を有効にします。

C5807 (E) The parse of this expression has changed between the point at which it appeared in the program and the point at which the expression was evaluated -- "typename" may be required to resolve the ambiguity
デフォルト実引数の式の評価が途中で終了しました。

C5808 (E) Default-initialization of reference is not allowed
リファレンス型のデフォルトの初期化は許されません。

C5809 (E) Uninitialized entity-kind "名前" has a const member
未初期化の"名前"が const 型メンバを持ちます。

C5810 (E) Uninitialized base class "型" has a const member
未初期化の基底クラス"型"が const 型メンバを持ちます。

C5811 (E) Const entity-kind "名前" requires an initializer -- class "型" has no explicitly declared default constructor
const 型の"名前"には初期化指定が必要です。クラス"型"が明示的に宣言されたデフォルトコンストラクタを持ちません。

C5812 (W) Const object requires an initializer -- class "型" has no explicitly declared default constructor
const 型オブジェクトには初期化指定が必要です。クラス"型"が明示的に宣言されたデフォルトコンストラクタを持ちません。

C5815 (I) Type qualifier on return type is meaningless
テンプレートで実体化されるリターン型に意味のない修飾型を指定しています。修飾型を有効にします。

C5817 (E) Static data member declaration is not allowed in this class
局所クラスは静的データメンバを持つことはできません。

C5818 (E) Template instantiation resulted in an invalid function declaration
テンプレートで実体化された関数宣言が正しくありません。

C5822 (E) Invalid destructor name for type "型"
"型"のデストラクタ名が正しくありません。

C5824 (E) Destructor reference is ambiguous -- both entity-kind "名前 1" and entity-kind "名前 2" could be used
"名前 1"と"名前 2"が使われました。デストラクタの参照があいまいです。

C5825 (W) Virtual inline entity-kind "名前" was never defined
仮想インラインメンバ関数"名前"の定義がありません。

C5826 (W) Entity-kind "名前" was never referenced
関数の引数"名前"は参照されません。

C5827 (E) Only one member of a union may be specified in a constructor initializer list
共用体の一つのメンバのみをコンストラクタの初期化で指定できます。

C5831 (I) Support for placement delete is disabled
operator delete 関数の型が正しくありません。処理を継続します。

C5832 (E) No appropriate operator delete is visible
適当な operator delete 関数が見つかりません。

C5833 (E) Pointer or reference to incomplete type is not allowed
不完全型へのポインタまたはリファレンス型は許されません。

C5834 (E) Invalid partial specialization -- entity-kind "名前" is already fully specialized
すでに特別化された"名前"を部分特別化しています。

C5835 (E) Incompatible exception specifications
例外指定の型が合致しません。

C5836 (W) Returning reference to local variable
ローカル変数のリファレンスをリターン値に指定しています。指定された処理を継続します。

C5837 (W) Omission of explicit type is nonstandard ("int" assumed)
型指定がありません。int 型を仮定します。

C5838 (E) More than one partial specialization matches the template argument list of entity-kind "名前"
部分特別化テンプレート"名前"のテンプレート実引数があいまいです。

C5840 (E) A template argument list is not allowed in a declaration of a primary template
プライマリテンプレート宣言にテンプレート実引数は指定できません。

C5841 (E) Partial specializations may not have default template arguments
部分特別化テンプレートはデフォルトのテンプレート引数を持つことはできません。

C5842 (E) Entity-kind "名前 1" is not used in template argument list of entity-kind "名前 2"
部分特別化テンプレート"名前 1"は"名前 2"のテンプレート実引数に使用されません。

C5843 (E) The type of partial specialization template parameter entity-kind "名前" depends on another template parameter
部分特別化テンプレート"名前"のテンプレート仮引数が別のテンプレート仮引数に依存しています。

C5844 (E) The template argument list of the partial specialization includes a nontype argument whose type depends on a template parameter
部分特別化テンプレートのテンプレート実引数がテンプレート仮引数に依存する非型の実引数を含んでいます。

C5845 (E) This partial specialization would have been used to instantiate entity-kind "名前"
この部分特別化テンプレートはプライマリテンプレート"名前"を実体化しようとしています。

C5846 (E) This partial specialization would have been made the instantiation of entity-kind "名前" ambiguous
この部分特別化テンプレートは"名前"の実体化があいまいになります。

C5847 (E) Expression must have integral or enum type
式の型は汎整数型か enum 型でなければなりません。

C5848 (E) Expression must have arithmetic or enum type
式の型は算術型か enum 型でなければなりません。

C5849 (E) Expression must have arithmetic, enum, or pointer type
式の型は算術型、enum 型もしくはポインタ型でなければなりません。

C5850 (E) Type of cast must be integral or enum
キャストの型は汎整数型か enum 型でなければなりません。

C5851 (E) Type of cast must be arithmetic, enum, or pointer
キャストの型は算術型、enum 型もしくはポインタ型でなければなりません。

C5852 (E) Expression must be a pointer to a complete object type
式の型は完全オブジェクト型へのポインタ型でなければなりません。

C5853 (E) A partial specialization of a member class template must be declared in the class of which it is a member
メンバクラスの部分特別化テンプレートはそのメンバを含むクラスの中で宣言しなければなりません。

C5854 (E) A partial specialization nontype argument must be the name of a nontype parameter or a constant
部分特別化テンプレートの非型テンプレート実引数は非型の仮引数名か定数でなければなりません。

C5855 (E) Return type is not identical to return type "型" of overridden virtual function entity-kind "名前"
関数のリターン型がオーバーライドされた仮想関数"名前"のリターン型"型"と同一でありません。

C5857 (E) A partial specialization of a class template must be declared in the namespace of which it is a member
部分特別化テンプレートはそのメンバを含む namespace の中で宣言しなければなりません。

C5858 (E) Entity-kind "名前" is a pure virtual function
"名前"は純粋仮想関数です。

C5859 (E) Pure virtual entity-kind "名前" has no overrider
純粋仮想関数"名前"はオーバーライドされません。

C5861 (E) Invalid character in input line
行中に不正な文字が現れました。

C5862 (E) Function returns incomplete type "型"
関数のリターン型"型"が不完全型です。

C5864 (E) "名前" is not a template
"名前"はテンプレートでありません。

C5865 (E) A friend declaration may not declare a partial specialization
部分特別化テンプレートはフレンド宣言内で指定できません。

C5867 (W) Declaration of "size_t" does not match the expected type "型"
size_t 型が期待する"型"と異なります。

C5868 (E) Space required between adjacent ">" delimiters of nested template argument lists (">>" is the right shift operator)
２つのテンプレート実引数リストの最後に指定する">>"は間に空白が必要です。

C5870 (E) Invalid multibyte character sequence
不正な 2 バイト文字があります。

C5871 (E) Template instantiation resulted in unexpected function type of "型 1" (the meaning of a name may have changed since the template declaration -- the type of the template is "型 2")
"型 2"を持つテンプレートの実体化の結果、期待されない型"型 1"の関数が作られました。

C5873 (E) Non-integral operation not allowed in nontype template argument
非型のテンプレート実引数に非汎整数型の演算は許されません。

C5875 (W) Embedded C++ does not support templates
Embedded C++仕様はテンプレート機能をサポートしません。

C5876 (W) Embedded C++ does not support exception handling
Embedded C++仕様は例外処理機能をサポートしません。

C5877 (W) Embedded C++ does not support namespaces
Embedded C++仕様は namespace 機能をサポートしません。

C5878 (W) Embedded C++ does not support run-time type information
Embedded C++仕様はランタイム型情報機能をサポートしません。

C5879 (W) Embedded C++ does not support the new cast syntax
Embedded C++仕様は new のキャスト機能をサポートしません。

C5880 (W) Embedded C++ does not support using-declarations
Embedded C++仕様は using 宣言機能をサポートしません。

C5881 (W) Embedded C++ does not support "mutable"
Embedded C++仕様は mutable 機能をサポートしません。

C5882 (W) Embedded C++ does not support multiple or virtual inheritance
Embedded C++仕様は多重継承/仮想継承機能をサポートしません。

C5885 (E) "型1" cannot be used to designate constructor for "型2"
"型1"はコンストラクタの"型2"で使用することはできません。

C5891 (E) An explicit template argument list is not allowed on this declaration
この宣言内では明示的なテンプレート実引数は許されません。

C5894 (E) Entity-kind "名前" is not a template
"名前"はテンプレートではありません。

C5896 (E) Expected a template argument
テンプレートの実引数が期待されます。

C5898 (E) Nonmember operator requires a parameter with class or enum type
非メンバ演算子関数にはクラスまたはenum型の仮引数が要求されます。

C5900 (E) Using-declaration of entity-kind "名前" is not allowed
"名前"のusing宣言は許されません。

C5901 (E) Qualifier of destructor name "型1" does not match type "型2"
"型1"のデストラクタの限定名が"型2"に一致しません。

C5902 (W) Type qualifier ignored
型限定名が不正です。型限定名を無効にします。

C5916 (E) Cannot convert pointer to member of derived class "型1" to pointer to member of base class "型2" -- base class is virtual
派生クラス"型1"のメンバへのポインタ型を仮想基底クラス"型2"のメンバへのポインタ型に変換できません。

C5919 (F) Invalid output file: "名前"
テンプレート情報ファイルの"名前"が不正です。
コンパイラの環境設定やホスト環境のファイルシステム異常がないか確認ください。

C5920 (F) Cannot open output file: "名前"
テンプレート情報ファイル"名前"をオープンすることができません。
コンパイラの環境設定やホスト環境のファイルシステム異常がないか確認ください。

C5926 (F) Cannot open definition list file: "名前"
ファイル"名前"をオープンすることができません。
コンパイラの環境設定やホスト環境のファイルシステム異常がないか確認ください。

C5928 (E) Incorrect use of va_start
va_startの使用方法に誤りがあります。

C5929 (E) Incorrect use of va_arg
va_arg の使用方法に誤りがあります。

C5930 (E) Incorrect use of va_end
va_end の使用方法に誤りがあります。

C5935 (E) "typedef" may not be specified here
typedef を指定することはできません。

C5936 (W) Redeclaration of entity-kind "名前" alters its access
"名前"の再宣言でアクセス指定を変更しています。
再定義されたアクセス指定を有効にします。

C5937 (E) A class or namespace qualified name is required
クラスまたは namespace の限定名が要求されます。

C5940 (W) Missing return statement at end of non-void entity-kind "名前"
void 型以外をリターンする関数"名前"が return 文を持ちません。
return 値は不定になります。

C5941 (W) Duplicate using-declaration of "名前" ignored
using 宣言"名前"を重複指定しています。重複した using 宣言を無効にします。

C5946 (E) Name following "template" must be a member template
"template"に続く名前はメンバテンプレートでなければなりません。

C5947 (E) Name following "template" must have a template argument list
"template"に続く名前はテンプレート実引数でなければなりません。

C5952 (E) A template parameter may not have class type
テンプレート仮引数にクラス型名は指定できません。

C5953 (E) A default template argument cannot be specified on the declaration of a member of a class template
クラステンプレートのメンバ宣言にデフォルトのテンプレート実引数を指定できません。

C5954 (E) A return statement is not allowed in a handler of a function try block of a constructor
コンストラクタの try ブロックのハンドラ内にリターン文は許されません。

C5959 (W) Declared size for bit field is larger than the size of the bit field type; truncated to "サイズ" bits
指定されたビット数がビットフィールドの型の"サイズ"を超えています。
ビット数をビットフィールドの型のサイズに合わせて処理を継続します。

C5960 (E) Type used as constructor name does not match type "型"
コンストラクタ名として使用された型が"型"と一致しません。

C5961 (W) Use of a type with no linkage to declare a variable with linkage
リンケージを持たない型を使用してリンケージを持つ変数として宣言しています。
リンケージを持つものとします。

C5962 (W) Use of a type with no linkage to declare a function
リンケージを持たない型を使用してリンケージを持つ関数として宣言しています。
リンケージを持つものとします。

C5963 (E) Return type may not be specified on a constructor
コンストラクタにリターン型を指定できません。

C5964 (E) Return type may not be specified on a destructor
デストラクタにリターン型を指定できません。

C5965 (E) Incorrectly formed universal character name
universal character の形式が正しくありません。

C5966 (E) Universal character name specifies an invalid character
universal character で指定された文字が不正です。

C5967 (E) A universal character name cannot designate a character in the basic character set
基本文字集合内で universal character を文字として指定することはできません。

C5968 (E) This universal character is not allowed in an identifier
識別子にこの universal character は許されません。

C5978 (E) A template friend declaration cannot be declared in a local class
テンプレートのフレンド関数は局所クラスで宣言できません。

C5979 (E) Ambiguous "?" operation: second operand of type "型1" can be converted to third operand type "型2", and vice versa
三項演算子"?:"の第２式の"型 1"と第３式の"型 2"が互いに変換可能な型であいまいです。

C5980 (E) Call of an object of a class type without appropriate operator() or conversion functions to pointer-to-function type
オブジェクトを呼び出していますが operator()関数または関数へのポインタ型変換関数が定義されていません。

C5982 (E) There is more than one way an object of type "型" can be called for the argument list
実引数リストから呼ぶことができる"型"のオブジェクトが２つ以上あります。

C5984 (W) Operator new and operator delete cannot be given internal linkage
delete が static で定義されています。

C5985 (E) Storage class "mutable" is not allowed for anonymous unions
mutable を無名共用体に指定することはできません。

C5987 (E) Abstract class type "型" is not allowed as catch type:
抽象クラスを catch で受けることはできません。

C5988 (E) A qualified function type cannot be used to declare a nonmember function or a static member function
修飾付き関数型を非メンバ関数や static メンバ関数の宣言に使用することはできません。

C5989 (E) A qualified function type cannot be used to declare a parameter
修飾付き関数型を関数パラメータ指定に使用することはできません。

C5990 (E) Cannot create a pointer or reference to qualified function type
修飾付き関数型へのポインタ型や参照型を作ることはできません。

C5991 (W) Extra braces are nonstandard
集合型の初期化子リストに余分な'{'があります。

C5994 (E) An empty template parameter list is not allowed in a template template parameter declaration
空テンプレートパラメータを持つテンプレートをテンプレートパラメータに指定することはできません。

C5995 (E) Expected "class"
テンプレートパラメータに指定するクラステンプレートはクラスを必要とします。

C5996 (E) The "class" keyword must be used when declaring a template template parameter
テンプレートパラメータに指定するクラステンプレートは構造体ではいけません。

C5998 (E) A qualified name is not allowed for a friend declaration that is a function definition
friend 指定付き関数定義において、名前空間の名前付き関数名を指定することはできません。

C5999 (E) "型" is not compatible with "型"
指定したクラステンプレートはテンプレートパラメータと形式が一致しません。

C6000 (W) A storage class may not be specified here
ここには記憶クラス指定子を指定することはできません。

C6006 (E) A template template parameter cannot have the same name as one of its template parameters
テンプレートパラメータに指定するクラステンプレート名が、それ自身のテンプレートパラメータ名と同じです。

C6008 (E) A parameter of a template template parameter cannot depend on the type of another template parameter
テンプレートパラメータに指定するテンプレートが持つ非型パラメータの型は、他の型パラメータに依存してはいけません。

C6009 (E) "インスタンス名" is not an entity that can be defined
実体のないインスタンスを生成しようとしています。

C6010 (E)Destructor name must be qualified
デストラクタ名は限定する必要があります。

C6013 (E) A qualified friend template declaration must refer to a specific previously declared template
限定フレンドテンプレートは参照前に定義しておく必要があります。

C6018 (E) "クラス名" has no member class "メンバ名"
クラスにないメンバを使っています。

C6019 (E) The global scope has no class named "クラス名"
クラス内の名前にファイルスコープ演算子を使っています。

C6020 (E) Recursive instantiation of template default argument
テンプレートのデフォルト引数で再帰的にインスタンスを生成します

C6021 (E) Access declarations and using-declarations cannot appear in unions
union で using 指定は使えません。

C6022 (E) "名前" is not a class member
クラスのメンバではありません。

C6028 (W) Invalid redeclaration of nested class
クラス内でクラスを 2 重定義しています。

C6035 (E) "テンプレート名" cannot be declared in this scope
このスコープでは template を宣言することができません。

C6057 (E) __evenaccess qualifier is applied to only integer type
__evenaccess 修飾子は整数型以外を修飾できません。

C6058 (E) Expected a section name string
__sectop/__secend にセクション名がありません。

C6059 (E) Expected a section name
#pragma section のセクション名が不正です。

C6060 (E) Invalid pragma declaration
#pragma の構文が不正です。

C6061 (E) "名前" has already been specified by other pragma
このシンボルは既に他の#pragma 指定がされています。

C6062 (E) Pragma may not be specified after definition
シンボル定義後の宣言にのみ#pragma 指定することはできません。

C6063 (E) Invalid kind of pragma is specified to this symbol
不正な#pragma を指定しました。

C6064 (I) This pragma has no effect
この#pragma で指定されたシンボルが存在しません。

C6065 (E) __regparam? must be qualified for function type
__regparam2/__regparam3 は関数型以外を修飾できません。

C6066 (E) Illegal 属性指定子 specifier
属性指定子の記述方法に誤りがあります。

C6067 (E) Multiple pointer qualifiers
__ptr16 が複数指定されました。

C6068 (E) __ptr16 must be qualified for data pointer type
__ptr16 がデータポインタ型以外に指定されました。

12.3 C ライブラリ関数のエラーメッセージ

ライブラリ関数の中には，ライブラリ関数を実行中にエラーが発生した場合，標準ライブラリのヘッダファイル <stddef.h> で定義しているマクロ errno にエラー番号を設定するものがあります。エラー番号には, 対応するエラーメッセージが定義してあり, エラーメッセージを出力することができます。エラーメッセージを出力するプログラム例を以下に示します。

例

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

void main(void)
{
            FILE *fp;

            fp=fopen(“file”, “w”);
            fp=NULL;

            fclose(fp);                         /* error occurred  */

            printf(“%s\n”, strerror(errno));    /* print error message */
}
```

説明

fclose 関数に値 NULL のファイルポインタを実引数として渡しているので，エラーとなります。このとき errno に対応するエラー番号が設定されます。

strerror 関数は, エラー番号を実引数として渡すと, 対応するエラーメッセージの文字列のポインタを返します。printf 関数の文字列出力指定によりエラーメッセージを出力します。

表 12.1　C ライブラリ関数のエラーメッセージ一覧

エラー番号	エラーメッセージ／説明	エラー番号を設定する関数
1100 (ERANGE)	DATA OUT OF RANGE オーバフローが発生しました。	frexp, ldexp, modf, ceil, floor, fmod, strtol, atoi, atol, perror, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, acos, acosf, asin, asinf, atan, atan2, atan2f, atanf, ceilf, cos, cosf, cosh, coshf, exp, expf, floorf, fmodf, ldexpf, log, log10, log10f, logf, modff, pow, powf, sin, sinf, sinh, sinhf, sqrt, sqrtf, tan, tanf, tanh, tanhf, fabs, fabsf, frexpf
1101 (EDOM)	DATA OUT OF DOMAIN 数学関数の引数に対する結果の値が定義されていません。	acos, acosf, asin, asinf, atan, atan2, atan2f, atanf, ceil, ceilf, cos, cosf, cosh, coshf, exp, expf, floor, floorf, fmod, fmodf, ldexp, ldexpf, log, log10, log10f, logf, modf, modff, pow, powf, sin, sinf, sinh, sinhf, sqrt, sqrtf, tan, tanf, tanh, tanhf, fabs, fabsf, frexp, frexpf
1102 (EDIV)	DIVISION BY ZERO ゼロによる除算を行っています。	div, ldiv
1104 (ESTRN)	TOO LONG STRING 文字列の長さが 32767 文字を超えています。	strtol, strtod, atoi, atol, atof
1106 (PTRERR)	INVALID FILE POINTER ファイルポインタの値に NULL ポインタ定数を指定しています。	fclose, fflush, freopen, setbuf, setvbuf, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, ungetc, fread, fwrite, fseek, ftell, rewind, perror
1200 (ECBASE)	INVALID RADIX 基数の指定が誤っています。	strtol, atoi, atol
1202 (ETLN)	NUMBER TOO LONG 数値を表現する文字列の長さが 17 桁を超えています。	strtod, fscanf, scanf, sscanf, atof
1204 (EEXP)	EXPONENT TOO LARGE 指数部の桁数が 3 桁を超えています。	strtod, fscanf, scanf, sscanf, atof
1206 (EEXPN)	NORMALIZED EXPONENT TOO LARGE 文字列を一度 IEEE 規格の 10 進形式に正規化したとき指数部の桁数が 3 桁を超えています。	strtod, fscanf, scanf, sscanf, atof
1210 (EFLOATO)	OVERFLOW OUT OF FLOAT float 型の 10 進数値が, float 型の範囲を超えています(オーバフロー)。	strtod, fscanf, scanf, sscanf, atof
1220 (EFLOATU)	UNDERFLOW OUT OF FLOAT Float 型の 10 進数値が, float 型の範囲を超えています(アンダフロー)。	strtod, fscanf, scanf, sscanf, atof
1230 (EOVER)	FLOATING POINT OVERFLOW 数値定数が，double 型の範囲を超えています(オーバフロー)。	strtod, fscanf, scanf, sscanf, atof
1240 (EUNDER)	FLOATING POINT UNDERFLOW 数値定数が，double 型の範囲を超えています(アンダフロー)。	strtod, fscanf, scanf, sscanf, atof

エラー番号	エラーメッセージ／説明	エラー番号を設定する関数
1300 (NOTOPN)	FILE NOT OPEN **ファイルがオープンされていません。**	fclose, fflush, setbuf, setvbuf, fprintf, fscanf, printf, scanf, vfprintf, vprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, fread, fwrite, fseek, ftell, rewind, perror, freopen
1302 (EBADF)	BAD FILE NUMBER **入力専用ファイルに対して出力関数, あるいは出力専用ファイルに対して入力関数を発行しています。**	fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, perror, fread, fwrite
1304 (ECSPEC)	ERROR IN FORMAT **書式付き入出力関数で指定している書式が誤っています。**	fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, perror

13. アセンブラのエラーメッセージ

13.1 エラー形式とエラーレベル

本章では、以下の形式で出力するエラーメッセージとエラー内容を説明します。

エラー番号　(エラーレベル)　エラーメッセージ
　　エラー内容

エラーレベルは、エラーの重要度に従い、3 種類に分類されます。

エラーレベル		動作
(W)	ウォーニング	処理を継続します。
(E)	エラー	処理を継続します。
(F)	フェータル	処理を中断します。

13.2 メッセージ一覧

10 (E) NO INPUT FILE SPECIFIED
　　入力ソースファイルの指定がありません。
　　入力ソースファイルを指定してください。

20 (E) CANNOT OPEN FILE ファイル名
　　指定のファイルをオープンできません。
　　ファイル名、ディレクトリ名などを見直してください。

30 (E) INVALID COMMAND PARAMETER
　　オプションに誤りがあります。
　　オプションを見直してください。

40 (E) CANNOT ALLOCATE MEMORY
　　処理中にメモリが足りなくなりました。
　　ユーザが使用できるメモリ量が極端に少ない場合に発生します。
　　他に実行中の処理があればその処理を終了してからアセンブラを再起動してください。
　　それでも本エラーが発生する場合はホストシステムのメモリ管理の方法を見直してください。

50 (E) INVALID FILE NAME ファイル名
　　ディレクトリを含めたファイル名が長すぎるか、ファイル名に誤りがあります。
　　ファイル名を見直してください。
　　このときアセンブラが出力するオブジェクトモジュールはデバッガで扱えない可能性があります。

60 (W) INVALID VALUE ファイル名
SBR オプションの定数値の最下位８ビットに 0 以外を指定しました。
定数値を見直してください。
アセンブラは定数値の最下位８ビットを 0 に変更します。

101 (E) SYNTAX ERROR IN SOURCE STATEMENT
ソースステートメントに構文上の誤りがあります。
ソースステートメント全体を見直してください。

102 (E) SYNTAX ERROR IN DIRECTIVE
アセンブラ制御命令のソースステートメントに構文上の誤りがあります。
ソースステートメント全体を見直してください。

103 (E) .END NOT FOUND
プログラムに.END がありません。
.END を記述してください。

104 (E) LOCATION COUNTER OVERFLOW
ロケーションカウンタ値が最大値を超えています。
プログラムを縮小してください。

105 (E) ILLEGAL INSTRUCTION IN STACK SECTION
スタックセクション内に実行命令、データを確保するアセンブラ制御命令を記述しています。
実行命令、データを確保するアセンブラ制御命令を削除してください。

106 (E) TOO MANY ERRORS
エラーの数が多いので表示を打ち切りました。
ソースステートメント全体を見直してください。

108 (E) ILLEGAL CONTINUATION LINE
複数行にわたって記述したソースステートメントに誤りがあります。
記述方法を見直してください。

150 (E) INVALID DELAY SLOT INSTRUCTION
ディレイスロット命令（遅延分岐命令の直後にくる実行命令）が不当です。
実行命令の記述順序を変更するなどして、ディレイスロット命令が不当とならないようにしてください。

200 (E) UNDEFINED SYMBOL REFERENCE
参照しているシンボルが定義されていません。
シンボルを定義してください。

201 (E) ILLEGAL SYMBOL OR SECTION NAME
シンボル（セクション名を含む）として予約語(レジスタ名、演算子、ロケーションカウンタ)を指定しています。
シンボル（セクション名を含む）を訂正してください。

202 (E) ILLEGAL SYMBOL OR SECTION NAME
シンボル（セクション名を含む）に誤りがあります。
シンボル（セクション名を含む）を訂正してください。

203 (E) ILLEGAL LOCAL LABEL
ローカルラベルの指定に誤りがあります。
ローカルラベルの指定を訂正してください。

300 (E) ILLEGAL MNEMONIC
オペレーションに誤りがあります。
オペレーションを訂正してください。

301 (E) TOO MANY OPERANDS OR ILLEGAL COMMENT
アセンブラ制御命令のオペランドが多すぎるか、コメントに誤りがあります。
オペランドまたはコメントを訂正してください。

304 (E) LACKING OPERANDS
オペランドが足りません。
オペランドを訂正してください。

306 (E) SYNTAX ERROR IN REGISTER LIST
複数レジスタの指定方法に誤りがあります。
複数レジスタの指定を訂正してください。

307 (E) ILLEGAL ADDRESSING MODE OR OBJECT CODE SIZE
オペランドに許されないアドレス形式を指定しているか、
確保サイズ(:8、:16、:24、:32)に誤りがあります。
オペランドまたは確保サイズを訂正してください。

308 (E) SYNTAX ERROR IN OPERAND
オペランドに文法上の誤りがあります。
オペランドを訂正してください。

400 (E) CHARACTER CONSTANT TOO LONG
文字定数が 4 文字を超えています。
文字定数を訂正してください。

402 (E) ILLEGAL VALUE IN OPERAND
オペランドとして範囲外の値です。
値を変更してください。

403 (E) ILLEGAL OPERATION FOR RELATIVE VALUE
相対アドレスに対して乗除算または論理演算を指定しています。
演算内容を訂正してください。

404 (E) ILLEGAL IMMEDIATE DATA
#1,2,4 、#0～3、#0～7 のオペランドで値に相対値を指定しています。
相対値は使用できませんので、値を訂正してください。

407 (E) MEMORY OVERFLOW
式の計算中、計算用のメモリが足りなくなりました。
演算内容を簡単にしてください。

408 (E) DIVISION BY ZERO
0 除算を指定しています。
演算内容を変更してください。

409 (E) REGISTER IN EXPRESSION
式の中にレジスタ名が現れました。
演算内容を訂正してください。

411 (E) INVALID STARTOF/SIZEOF OPERAND
STARTOF 演算または SIZEOF 演算でセクション名以外を指定しています。
演算の内容を訂正してください。

412 (E) ILLEGAL SYMBOL IN EXPRESSION
シフト数に相対値または、相対シンボルを指定しています。
演算内容を訂正してください。

413 (E) ILLEGAL DISPLACEMENT VALUE
ディスプレースメント値が不当です。
ディスプレースメントを偶数にしてください。

500 (E) SYMBOL NOT FOUND
ラベルが必要なアセンブラ制御命令のソースステートメントにラベルがありません。
ラベルを記述してください。

501 (E) ILLEGAL ADDRESS VALUE IN OPERAND
セクションの先頭アドレスの指定が誤っているか、ロケーションカウンタ値の指定が誤っています。
先頭アドレスまたはロケーションカウンタ値を訂正してください。

502 (E) ILLEGAL SYMBOL IN OPERAND
オペランドに不当な値（前方参照シンボル、外部参照シンボル、相対アドレスシンボル、未定義シンボル）を指定しています。
オペランドを訂正してください。

503 (E) UNDEFINED EXPORT SYMBOL
ファイル内で定義していないシンボルを外部定義シンボルとして宣言しています。
シンボルを定義するか、外部定義シンボルとしての宣言を取りやめてください。

504 (E) INVALID RELATIVE SYMBOL IN OPERAND
オペランドに不当な値（前方参照シンボル、外部参照シンボル）を指定しています。
オペランドを訂正してください。

505 (E) ILLEGAL OPERAND
オペランドの名称に誤りがあります。
オペランドを訂正してください。

506 (E) ILLEGAL OPERAND
オペランドとして許されない要素を指定しています。
オペランドを訂正してください。

508 (E) ILLEGAL VALUE IN OPERAND
オペランドに範囲外の値を指定しています。
オペランドを訂正してください。

510 (E) ILLEGAL BOUNDARY VALUE
境界調整数の指定に誤りがあります。
境界調整数を訂正してください。

511 (E) ILLEGAL DISPLACEMENT SIZE
.DISPSIZE のビット数に誤りがあります。
ビット数を訂正してください。

512 (E) ILLEGAL EXECUTION START ADDRESS
実行開始アドレスに誤りがあります。
実行開始アドレスを訂正してください。

513 (E) ILLEGAL REGISTER NAME
レジスタ名に誤りがあります。
レジスタ名を訂正してください。

514 (E) INVALID EXPORT SYMBOL
外部定義できないシンボルを外部定義シンボルとして宣言しています。
外部定義シンボルとしての宣言を取りやめてください。

516 (E) EXCLUSIVE DIRECTIVES
制御命令の指定内容が矛盾しています。
関連する制御命令を含めて見直してください。

517 (E) INVALID VALUE IN OPERAND
オペランドに不当な値（前方参照シンボル、外部参照シンボル、他セクションの相対アドレスシンボル）を指定しています。
オペランドを訂正してください。

518 (E) INVALID IMPORT SYMBOL
ファイル内で定義しているシンボルを外部参照シンボルとして宣言しています。
外部参照シンボルとしての宣言を取りやめてください。

520 (E) ILLEGAL .CPU DIRECTIVE POSITION
.CPU 制御命令がプログラムの先頭にないか、複数回指定しています。
.CPU 制御命令はプログラムの先頭に 1 回だけ指定してください。

521 (E) ILLEGAL SYMBOL IN OPERAND
optimize オプション指定時において、定数値を指定するオペランドにアドレスを値に持つシンボル、またはロケーションカウンタ値を指定しています。
アドレスを値に持つシンボル、ロケーションカウンタ値を指定する場合には、optimize オプションを指定しないでください。

523 (E) ILLEGAL OPERAND
.LINE 制御命令のオペランドに誤りがあります。
.LINE 制御命令のオペランドを訂正してください。

524 (E) ILLEGAL ADDRESSING SPACE SIZE
.CPU 制御命令のオペランドに、許されないアドレス空間のビット幅を指定しています。
アドレス空間のビット幅を訂正してください。

525 (E) ILLEGAL .LINE DIRECTIVE POSITION
.LINE 制御命令をマクロ展開または条件付き繰り返し展開内に指定しています。
.LINE 制御命令の指定位置を変えてください。

526 (E) STRING TOO LONG
オペランドの文字列が 255 文字を超えています。
.SDATA、.SDATAB、.SDATAC、.SDATAZ 制御命令のオペランドに指定する文字列は 255 文字以内としてください。

527 (E) CANNOT SUPPORT COMMON SECTION SINCE VERSION 4
セクション属性に COMMON を指定しています。
コモンセクションは使用できなくなりました。
最適化リンケージエディタの start オプションでコロン(:)を用いて複数セクションを同一アドレスに配置できます。

528 (E) SPECIFICATION OF THE ADDRESS OVERLAPS
セクション内のアドレス割付けが重複しています。
.SECTION 制御命令、.ORG 制御命令の指定内容を見直してください。

529 (E) THE ADDRESS BETWEEN SECTIONS OVERLAPS
セクション間のアドレス割付けが重複しています。
.SECTION 制御命令、.ORG 制御命令の指定内容を見直してください。

600 (E) INVALID CHARACTER
ソースプログラムに不当な文字があります。
不当な文字を訂正してください。

601 (E) INVALID DELIMITER
区切り文字が不当です。
区切り文字を訂正してください。

602 (E) INVALID CHARACTER STRING FORMAT
文字列に誤りがあります。
文字列を訂正してください。

603 (E) SYNTAX ERROR IN SOURCE STATEMENT
ソースステートメントに構文上の誤りがあります。
ソースステートメント全体を見直してください。

604 (E) ILLEGAL SYMBOL IN OPERAND
絶対値を指定するオペランドに、相対値、前方参照値および、未定義シンボルを指定しています。
値を訂正してください。

610 (E) MULTIPLE MACRO NAMES
.MACRO で定義しようとしているマクロ名は既に定義されています。
マクロ名を訂正してください。

611 (E) MACRO NAME NOT FOUND
.MACRO のオペランドにマクロ名がありません。
マクロ名を記述してください。

612 (E) ILLEGAL MACRO NAME
.MACRO のマクロ名に誤りがあります。
マクロ名を訂正してください。
マクロ名には、実行命令、制御命令(ピリオド(.)を除く)、制御文(ピリオド(.)を除く)のニーモニックは指定できません。

613 (E) ILLEGAL .MACRO DIRECTVE POSITION
マクロ本体（.MACRO～.ENDM 間）、.AREPEAT～.AENDR 間、.AWHILE～.AENDW 間に.MACRO があります。
.MACRO を削除してください。

614 (E) MULTIPLE MACRO PARAMETERS
マクロ定義（.MACRO）の仮引数の宣言で仮引数名が重複しています。
仮引数名を訂正してください。

615 (E) ILLEGAL .END DIRECTIVE POSITION
マクロ本体（.MACRO～.ENDM 間）に.END があります。
.END を削除してください。

616 (E) MACRO DIRECTIVES MISMATCH
.ENDM が.MACRO に対応していないか、.EXITM がマクロ本体（.MACRO～.ENDM 間）、.AREPEAT～.AENDR 間、.AWHILE～.AENDW 間以外にあります。
.ENDM または.EXITM を削除してください。

618 (E) MACRO EXPANSION TOO LONG
マクロ展開で 1 行の文字数が 8,192 文字を超えています。
8,192 文字以下になるように訂正してください。

619 (E) ILLEGAL MACRO PARAMETER
マクロコールでマクロパラメータの仮引数名に誤りがあるか、マクロ本体（.MACRO～.ENDM 間）の仮引数名に誤りがあります。
仮引数名を訂正してください。
マクロ本体の仮引数名が誤りの場合はマクロ展開時にエラーになります。

620 (E) UNDEFINED PREPROCESSOR VARIABLE
参照しているプリプロセッサ変数が定義されていません。
プリプロセッサ変数を定義してください。

621 (E) ILLEGAL .END DIRECTIVE POSITION
マクロ展開中に.END があります。
.END を削除してください。

622 (E) ')' NOT FOUND
マクロ処理除外の閉じカッコがありません。
マクロ処理除外の閉じカッコを記述してください。

623 (E) SYNTAX ERROR IN STRING FUNCTION
文字列操作関数に構文上の誤りがあります。
文字列操作関数を見直してください。

624 (E) MACRO PARAMETERS MISMATCH
マクロコールで位置指定のマクロパラメータの数が多すぎます。
マクロパラメータの数を訂正してください。

630 (E) SYNTAX ERROR IN OPERAND
構造化アセンブリ制御文のオペランドに構文上の誤りがあります。
ソースステートメント全体を見直してください。

631 (E) END DIRECTIVE MISMATCH
対になる制御文で終了の制御文が一致しません。
制御文を見直してください。

632 (E) SYNTAX ERROR IN OPERAND
構造化アセンブリ制御文のオペランドのコンディションコードに誤りがあります。
コンディションコードを訂正してください。

633 (E) ILLEGAL .BREAK OR .CONTINUE DIRECTIVE POSITION
.BREAK、.CONTINUEが.FOR[U]～ .ENDF間、.WHILE～.ENDW間、.REPEAT～.UNTIL間以外にあります。
.BREAK、.CONTINUEを削除してください。

634 (E) EXPANSION TOO LONG
構造化アセンブリ展開で、1行の文字数が8,192文字を超えました。
8,192文字以下になるように訂正してください。

640 (E) SYNTAX ERROR IN OPERAND
条件つきアセンブリ制御文のオペランドに構文上の誤りがあります。
ソースステートメント全体を見直してください。

641 (E) INVALID RELATIONAL OPERATOR
条件つきアセンブリ制御文のオペランドの関係演算子に誤りがあります。
関係演算子を訂正してください。

642 (E) ILLEGAL .END DIRECTIVE POSITION
.AREPEAT～.AENDR間、.AWHILE～.AENDW間に.ENDがあります。
.ENDを削除してください。

643 (E) DIRECTIVE MISMATCH
.AREPEAT、.AWHILEに対する.AENDR、.AENDWが対になっていません。
制御文を見直してください。

644 (E) ILLEGAL .AENDW OR .AENDR DIRECTIVE POSITION
.AIF～.AENDI間に.AENDR、.AENDWがあります。
.AENDR、.AENDWを削除してください。

645 (E) EXPANSION TOO LONG
.AREPEAT、.AWHILE展開で1行の文字数が8,192文字を超えています。
8,192文字以下になるように訂正してください。

650 (E) INVALID INCLUDE FILE
.INCLUDEのファイル名に誤りがあります。
ファイル名を訂正してください。

651 (E) CANNOT OPEN INCLUDE FILE
.INCLUDEのファイルをオープンできません。
ファイル名を訂正してください。

652 (E) INCLUDE NEST TOO DEEP
ファイルインクルードのネストが30レベルを超えています。
ネストを30レベル以下にしてください。

653 (E) SYNTAX ERROR IN OPERAND
.INCLUDE のオペランドに構文上の誤りがあります。
オペランドを訂正してください。

660 (E) .ENDM NOT FOUND
.MACRO に対する.ENDM がありません。
.ENDM を記述してください。

661 (E) END DIRECTIVE NOT FOUND
構造化アセンブリ制御文で終了の制御文が足りません。
終了の制御文を記述してください。

662 (E) ILLEGAL .END DIRECTIVE POSITION
.AIF～.AENDI 間に.END があります。
.END を削除してください。

663 (E) ILLEGAL .END DIRECTIVE POSITION
インクルードファイル中に.END があります。
.END を削除してください。

664 (E) ILLEGAL .END DIRECTIVE POSITION
.AIF～.AENDI 間に.END があります。
.END を削除してください。

665 (E) ILLEGAL SYMBOL IN OPERAND
optimize オプション指定時において、プリプロセッサ制御命令にプリプロセッサ変数以外のシンボルを指定しました。
シンボルを訂正してください。
また、プリプロセッサ変数以外のシンボルを指定する場合には、optimize オプションを指定しないでください。

667 (E) EXPANSION TOO LONG
.DEFINE 制御文で 1 行の文字数が 8,192 文字を超えています。
8,192 文字以下になるように訂正してください。

668 (E) ILLEGAL VALUE IN OPERAND
.AIFDEF 制御命令のオペランドに誤りがあります。
本制御命令のオペランドは.DEFINE 制御文のシンボルで指定してください。

669 (E) STRING TOO LONG
オペランドの文字列が 255 文字を超えています。
.ASSIGNC 制御命令、.DEFINE 制御命令、文字列操作関数（.LEN、.INSTR、.SUBSTR）のオペランドに指定する文字列は 255 文字以内としてください。

670 (E) SUCCESSFUL CONDITION .AERROR
.AERROR 制御文を含むステートメントが.AIF の条件によって処理されています。
.AERROR を処理しないように条件式を見直してください。

800 (W) SYMBOL NAME TOO LONG
プリプロセッサ変数名、または define 置換シンボル名が 32 文字を超えています。
シンボル名を訂正してください。
アセンブラは 33 文字目以降を無視します。

801 (W) MULTIPLE SYMBOLS
定義済みのシンボルを再び定義しています。
シンボルの再定義を取りやめてください。
アセンブラは 2 度目以降の定義を無視します。

805 (W) ILLEGAL OPERATION SIZE
構造化アセンブリ制御文の分岐サイズ(:8、:16)に誤りがあります。
分岐サイズを訂正してください。

807 (W) ILLEGAL OPERATION SIZE
オペレーションサイズに誤りがあります。
オペレーションサイズを訂正してください。
アセンブラはオペレーションサイズの指定を無視します。

808 (W) ILLEGAL CONSTANT SIZE
整数定数の記述の一部に誤りがあります。
記述を訂正してください。
解釈サイズには、バイト(.B)、ワード(.W)があり、それぞれ１バイト、２バイトの符号付きの値として解釈します。

810 (W) TOO MANY OPERANDS
実行命令のオペランドが多すぎるか、コメントに誤りがあります。
オペランドまたはコメントを訂正してください。
アセンブラは余分なオペランドの指定を無視します。

811 (W) ILLEGAL SYMBOL DEFINITION
ラベルを記述できないアセンブラ制御命令のソースステートメントにラベルを記述しています。
ラベルを削除してください。
アセンブラはラベルを無視します。

813 (W) SECTION ATTRIBUTE MISMATCH
セクションの再開で異なる種類のセクションを指定しているか、絶対アドレスセクションの再開でセクションの先頭アドレスを再び指定しています。
セクションを再開する場合はセクションの種類や先頭アドレスを指定しないでください。
セクションを開始したときの指定がそのまま有効です。

814 (W) ILLEGAL OBJECT CODE SIZE
確保サイズ(:8、:16、:24、:32)に誤りがあります。
確保サイズを訂正してください。
#xx:2、#xx:3 はマニュアル記述上の記号です。アセンブラでは記述できません。

815 (W) MULTIPLE MODULE NAMES
オブジェクトモジュール名を再設定しています。
オブジェクトモジュールの設定は 1 度だけにしてください。
アセンブラは 2 度目以降の設定を無視します。

816 (W) START ODD ADDRESS
偶数バイトのデータまたはデータ領域を奇数アドレスから確保しました。
偶数アドレスに訂正してください。

817 (W) OPERATION SIZE MISMATCH
バイトサイズ(.B)に対して、@-SP、@SP+を指定しています。
そのままオブジェクトコードを出力しますが、SP(スタックポインタ)が奇数値となるため、使用しないでください。

818 (W) ILLEGAL ACCESS SIZE
アクセスサイズ(:8、:16)に誤りがあります。
アクセスサイズを訂正してください。

819 (W) @Rn+, @-Rn, @+Rn, @Rn-, @(d,Rn) OR @Rn USED
H8/300H、H8S、H8SX CPU では、@Rn+、 @-Rn、 @+Rn、@Rn-、@(d,Rn)、@Rn で指定している Rn を ERn に訂正してください。

825 (W) ILLEGAL INSTRUCTION IN DUMMY SECTION
ダミーセクションに実行命令、データを確保するアセンブラ制御命令を記述しています。
実行命令、データを確保するアセンブラ制御命令を削除してください。
アセンブラは実行命令、データを確保するアセンブラ制御命令の記述を無視します。

830 (W) OPERATION SIZE MISMATCH
バイトサイズ(.B)に対して、ERn、Rn を指定しました。または、ワードサイズ(.W)に対して、ERn を指定しています。
レジスタを訂正してください。
バイトサイズでは RnL、ワードサイズでは Rn としてオブジェクトコードを生成します。

832 (W) MULTIPLE 'P' DEFINITIONS
デフォルトセクション名としての P が他のシンボルである P と重複しています。
P が重複しないようにしてください。
アセンブラは P をデフォルトセクション名とみなし、他のシンボル P の定義を無効とします。

835 (W) ILLEGAL VALUE IN OPERAND
実行命令のオペランドに範囲外の値を指定しています。
値を訂正してください。
アセンブラは値を範囲内に補正してオブジェクトコードを生成します。

836 (W) CONSTANT SIZE OVERFLOW
整数定数の値が整数定数の解釈サイズ(.B、.W)の範囲を超えています。
整数定数の値を訂正してください。
解釈サイズには、バイト(.B)、ワード(.W)があり、それぞれ１バイト、２バイトの符号付きの値として解釈します。

837 (W) SOURCE STATEMENT TOO LONG
ソースステートメントの１行の長さが 8,192 バイトを超えています。
コメント文を短くするなどして１行を 8,192 バイト以内に納めてください。
または、ソースステートメントを複数行に分けて記述してください。

838 (W) ILLEGAL CHARACTER CODE
コメント、文字列以外にシフト JIS、EUC または LATIN1 コードを指定したか、sjis、euc または、latin1 オプションの指定がありません。
シフト JIS コード、EUC または LATIN1 コードはコメント、文字列内に指定してください。または、sjis、euc、latin1 オプションを指定してください。

850 (W) ILLEGAL SYMBOL DEFINITION
ラベルフィールドにシンボルを指定しました。
シンボルを削除してください。

851 (W) MACRO SERIAL NUMBER OVERFLOW
マクロ生成番号が 99,999 を超えています。
マクロコールの回数を減らしてください。

852 (W) UNNECESSARY CHARACTER
オペランドの終了後に文字があります。
オペランドを訂正してください。

853 (W) NEGATIVE IMMEDIATE VALUE
.FOR[U]の増分値に、#-xx を記述しています。
-#xx に訂正してください。
そのまま、.FOR[U]を展開します。

854 (W) .AWHILE ABORTED BY .ALIMIT
展開回数が.ALIMIT 制御文で設定した上限値に達したため展開を中断しました。
繰り返しを展開する条件を見直してください。

855 (W) ILLEGAL VALUE IN OPERAND
SBR 制御命令の定数値の最下位８ビットに 0 以外を指定しました。
定数値を見直してください。
アセンブラは定数値の最下位８ビットを 0 に変更します。

870 (W) ILLEGAL DISPLACEMENT VALUE
ディスプレースメント値が不当です。
ディスプレースメントを偶数にしてください。
アセンブラは指定どおりにオブジェクトコードを生成します。

871 (W) MISSING DELAY SLOT INSTRUCTION
ディレイスロット命令（遅延分岐命令の直後にくる実行命令）が不明です。
実行命令の記述順序を変更するなどして、ディレイスロット命令を追加してください。
アセンブラはそのまま、オブジェクトコードを出力します。

901 (F) SOURCE FILE INPUT ERROR
ソースファイルの入力時にエラーが発生しました。
ディスクの空き容量を確認してください。
ディスク上の不要なファイルを削除するなどして必要な空き容量を確保してください。

902 (F) MEMORY OVERFLOW
メモリ不足です(中間語に関する情報を処理できません)。
プログラムを分割してください。

903 (F) LISTING FILE OUTPUT ERROR
リストファイルの出力時にエラーが発生しました。
ディスクの空き容量を確認してください。
ディスク上の不要なファイルを削除するなどして必要な空き容量を確保してください。

904 (F) OBJECT FILE OUTPUT ERROR
オブジェクトファイルの出力時にエラーが発生しました。
ディスクの空き容量を確認してください。
ディスク上の不要なファイルを削除するなどして必要な空き容量を確保してください。

905 (F) MEMORY OVERFLOW
メモリ不足です(ソースプログラムの行に関する情報を処理できません)。
プログラムを分割してください。

906 (F) MEMORY OVERFLOW
メモリ不足です(シンボルに関する情報を処理できません)。
プログラムを分割してください。

907 (F) MEMORY OVERFLOW
メモリ不足です(セクションに関する情報を処理できません)。
プログラムを分割してください。

908 (F) SECTION OVERFLOW
セクションの個数が多すぎます。
プログラムを分割してください。
セクションの上限は、goptimize オプションを指定している時で、デバッグ情報を出力するときは62,265 個です。デバッグ情報を出力しないときは 65,274 個までです。

933 (F) LACKING CPU SPECIFICATION
CPU 種別が設定されていません、
CPU 種別を CPU オプション、CPU 制御命令、H38CPU 環境変数のいずれかで指定してください。

935 (F) SUBCOMMAND FILE INPUT ERROR
サブコマンドファイル入力時にエラーが発生しました。
ディスクの空き容量を確認してください。
ディスク上の不要なファイルを削除するなどして必要な空き容量を確保してください。

954 (F) MEMORY OVERFLOW
メモリ不足です。
ソースプログラムを分割してください。

955 (F) LOCAL BLOCK NUMBER OVERFLOW
ローカルラベルの有効範囲であるローカルブロックの個数が 100,000 個を超えています。
ソースプログラムを分割してください。

956 (F) EXPAND FILE INPUT/OUTPUT ERROR
プリプロセッサ展開出力のファイル出力時にエラーが発生しました。
ディスクの空き容量を確認してください。
ディスク上の不要なファイルを削除するなどして必要な空き容量を確保してください。

957 (F) MEMORY OVERFLOW
メモリ不足です。
ソースプログラムを分割してください。

964 (F) MEMORY OVERFLOW
メモリ不足です(シンボルに関する情報を処理できません)。
ソースプログラムを分割してください。

970 (F) MEMORY OVERFLOW
メモリ不足です(セクションのサイズが大きすぎます)。
.ORG 制御命令でロケーションカウンタに大きなオフセットを与えたり、.DATAB 制御命令等で大きなデータ領域を確保した可能性があります。
セクションを分割するか、データ領域を小さくしてください。

14. 最適化リンケージエディタのエラーメッセージ

14.1 エラー形式とエラーレベル

本章では、以下の形式で出力するエラーメッセージとエラー内容を説明します。

エラー番号　　(エラーレベル) エラーメッセージ
エラー内容

エラーレベルは、エラーの重要度に従い、5 種類に分類されます。

	エラーレベル		動作
L0000 - L0999 P0000 - P0999	(I)	**インフォメーション**	**処理を継続します。**
L1000 - L1999 P1000 - P1999	(W)	**ウォーニング**	**処理を継続します。**
L2000 - L2999 P2000 - P2999	(E)	**エラー**	**オプション解析処理を継続し、処理を中断します。**
L3000 - L3999 P3000 - P3999	(F)	**フェータル**	**処理を中断します。**
L4000 - P4000 -	(-)	**インターナル**	**処理を中断します。**

L で始まるエラー番号は、最適化リンケージエディタ出力メッセージです。

P で始まるエラー番号は、プレリンカ出力メッセージです。P で始まるエラー番号は、nomessage オプションや change_message オプションで指定できません。

14.2 メッセージ一覧

L0001 (I)Section "セクション" created by optimization "最適化"
"最適化"の最適化によって、"セクション"を作成しました。

L0002 (I)Symbol "シンボル" created by optimization "最適化"
"最適化"の最適化によって、"シンボル"を作成しました。

L0003 (I)"ファイル"-"シンボル" moved to "セクション" by optimization
variable_access の最適化によって、"ファイル"内の"シンボル"を移動しました。

L0004 (I)"ファイル"-"シンボル" deleted by optimization
symbol_delete の最適化によって、"ファイル"内の"シンボル"を削除しました。

L0005 (I)The offset value from the symbol location has been changed by optimization : "ファイル"-"セクション"-"シンボル±offset"
"シンボル±offset"の範囲で最適化によるサイズ変更があったため offset 値を変更しました。問題ないか確認してください。offset 値の変更を抑止したい場合は、"ファイル"のアセンブル時に goptimize オプション指定を外してください。

L0100 (I)No inter-module optimization information in "ファイル"
"ファイル"内にモジュール間最適化情報がありません。"ファイル"をモジュール間最適化の対象外にします。モジュール間最適化の対象にする場合は、コンパイル、アセンブル時に goptimize オプションを指定してください。ただし、asmsh には goptimize オプションはありません。

L0101 (I)No stack information in "ファイル"
"ファイル"内にスタック情報がありません。"ファイル"はアセンブラ出力ファイルまたは SYSROF->ELF コンバートファイルの可能性があります。最適化リンケージエディタが出力するスタック情報ファイルに当該ファイルの内容は含まれません。

P0200 (I)"インスタンス" no longer needed in "ファイル"
使用しない"インスタンス"が"ファイル"内にあります。

P0201 (I)"インスタンス" assigned to file "ファイル"
"インスタンス"を"ファイル"に割り当てます。

P0202 (I)Executing: "コマンド"
インスタンス生成のために"コマンド"を実行しています。

P0203 (I)"インスタンス" adopted by file "ファイル"
"インスタンス"が"ファイル"に割り当てられました。

L0300 (I) Mode type "モード種別１" in "ファイル" differ from "モード種別２"
異なるモード種別のファイルを入力しました。

L1000 (W)Option "オプション" ignored
"オプション"は無効です。"オプション"を無視します。

L1001 (W)Option "オプション１" is ineffective without option "オプション２"
"オプション１"は"オプション２"が必要です。"オプション１"を無視します。

L1002 (W)Option "オプション１" cannot be combined with option "オプション２"
"オプション１"と"オプション２"は同時に指定できません。"オプション１"を無視します。

L1003 (W)Divided output file cannot be combined with option "オプション"
"オプション"指定時、出力ファイルの分割指定はできません。オプションの指定を無視します。先頭入力ファイル名を出力ファイル名として使用します。

L1004 (W)Fatal level message cannot be changed to other level : "番号"
Fatal レベルメッセージはレベル変更できません。"番号"の指定を無視します。
change_message オプションで変更できるエラーは、Information/Warning/Error レベルです。

L1005 (W)Subcommand file terminated with end option instead of exit option
end オプションの後に処理指定がありません。exit オプションを仮定して処理します。

L1006 (W)Options following exit option ignored
exit オプションの後のオプションを無視しました。

L1007 (W)Duplicate option : "オプション"
"オプション"が重複しています。最後に指定した方を有効にします。

L1007 (W)Duplicate option : "オプション"
"オプション"が重複しています。最後に指定した方を有効にします。

L1008 (W)Option "オプション" is effective only in cpu type "CPU 種別"
"オプション"は"CPU 種別"以外では無効です。"オプション"を無視します。

L1011 (W)Duplicate module specified in option "オプション" : "モジュール"
"オプション"で同じモジュールを 2 度指定しました。2 度目の指定を無視します。

L1012 (W)Duplicate symbol/section specified in option "オプション" : "名前"
"オプション"で同じシンボル名またはセクション名を 2 度指定しました。2 度目の指定を無視します。

L1013 (W)Duplicate number specified in option "オプション" : "番号"
"オプション"で同じエラー番号を指定しました。最後に指定した方を有効にします。

L1100 (W)Cannot find "名前" specified in option "オプション"
"オプション"で指定したシンボル名またはセクション名が見つかりません。"名前"の指定を無視します。

L1101 (W)"名前" in rename option conflicts between symbol and section
rename オプションで指定した"名前"がセクション名とシンボル名の両方に存在します。
シンボル名を変更の対象にします。

L1102 (W)Symbol "シンボル" redefined in option "オプション"
"オプション"で指定したシンボルはすでに定義されています。そのまま処理を続けます。

L1103 (W)Invalid address value specified in option "オプション" : "アドレス"
"オプション"で指定した"アドレス"は無効な値です。"アドレス"の指定を無視します。

L1104 (W)Invalid section specified in option "オプション" : "セクション"
"オプション"に初期値のないセクションは、指定できません。
"セクション"の指定を無視します。

L1110 (W)Entry symbol "シンボル" in entry option conflicts
entry オプションで指定した"シンボル"以外のシンボルがコンパイル、アセンブル時にエントリシンボルとして指定されています。オプション指定を優先します。

L1120 (W)Section address is not assigned to "セクション"
"セクション"のアドレス指定がありません。"セクション"を最後尾に配置します。

L1121 (W)Address cannot be assigned to absolute section "セクション" in start option
"セクション"は絶対アドレスセクションです。絶対アドレスセクションに対するアドレス指定を無視します。

L1122 (W)Section address in start option is incompatible with alignment : "セクション"
start オプションで指定した"セクション"のアドレスは境界調整数と矛盾しています。境界調整数に合わせてセクションアドレスを補正します。

L1130 (W)Section attribute mismatch in rom option : "セクション 1,セクション 2"
rom オプションで指定した"セクション 1"と"セクション 2"の属性、境界調整数が異なります。"セクション 2"の境界調整数はどちらか大きい方を有効とします。

L1140 (W)Load address overflowed out of record-type in option "オプション"
アドレス値よりも小さい record 形式を指定しました。指定した record 形式を超える範囲は、別の record 形式で出力します。

L1141 (W)Cannot fill unused area from "アドレス" with the specified value
空きエリアのサイズが space オプションで指定された値の倍数となっていないため、"アドレス"以降に指定データを出力できませんでした。

L1150 (W)Sections in fsymbol option have no symbol
fsymbol オプションで指定したセクションは外部定義シンボルがありません。fsymbol オプションを無視します。

L1160 (W)Undefined external symbol "シンボル"
未定義の"シンボル"を参照しています。

L1170 (W)Specified SBR addresses conflict
異なる複数のSBRアドレスが指定されました。SBR=USERとして処理します。

L1171 (W)Least significant byte in SBR="定数" ignored
SBR オプションで指定されたアドレス"定数"の下位 8bit は無効です。

L1200 (W)Backed up file "ファイル 1" into "ファイル 2"
"ファイル 1"を"ファイル 2"にバックアップしました。

L1300 (W)No debug information in input files
入力ファイル内にデバッグ情報がありません。debug,sdebug,compress オプション指定を無視します。コンパイル、アセンブル時に該当するオプションを指定しているか確認してください。

L1301 (W)No inter-module optimization information in input files
入力ファイル内にモジュール間最適化情報がありません。optimize オプションを無視します。コンパイル、アセンブル時に goptimize オプションを指定してください。

L1302 (W)No stack information in input files
入力ファイル内にスタック情報がありません。stack オプションを無視します。入力ファイルがアセンブラ出力ファイルまたは SYSROF->ELF コンバートファイルの場合は、stack オプションは無効です。

L1310 (W)"セクション" in "ファイル" is not supported in this tool
"ファイル"内に非サポートセクションがありました。"セクション"を無視します。

L1311 (W)Invalid debug information format in "ファイル"
"ファイル"内のデバッグ情報は dwarf2 ではありません。debug 情報を削除します。

L1320 (W)Duplicate symbol "シンボル" in "ファイル"
"シンボル"は重複しています。先に入力したファイル内シンボルを優先します。

L1321 (W)Entry symbol "シンボル" in "ファイル" conflicts
エントリシンボル定義のあるオブジェクトファイルを複数入力しました。先に入力したファイル内のエントリシンボルを有効にします。

L1322 (W)Section alignment mismatch : "セクション"
境界調整数の異なる同名セクションを入力しました。境界調整数は最大の指定を有効にします。

L1323 (W)Section attribute mismatch : "セクション"
属性の異なる同名セクションを入力しました。絶対セクションと相対セクションの場合は、絶対セクションとして扱います。read/write 属性が異なる場合は、どちらも許可します。

L1324 (W)Symbol size mismatch : "シンボル" in "ファイル"
サイズの異なるコモンシンボルまたは定義シンボルが入力されました。定義シンボルを優先します。コモンシンボル同士の場合は、先に入力したファイル内シンボルを優先します。

L1330 (W)Cpu type "CPU 種別１" in "ファイル" differ from "CPU 種別２"
異なる CPU 種別のファイルを入力しました。CPU 種別を H8SX として処理を継続します。

L1400 (W)Stack size overflow in register optimization
レジスタ最適化で、スタックアクセスコードがコンパイラのスタック量制限値を超えました。レジスタ最適化指定を無視します。

L1401 (W)Function call nest too deep
関数の呼び出しネストが深すぎるため、レジスタ最適化を実施できません。

L1410 (W)Cannot optimize "ファイル"-"セクション" due to
multi label relocation operation
複数ラベルのリロケーション演算を持つセクションは最適化できません。"ファイル"内の"セクション"を最適化対象外にします。

L1420 (W)"ファイル" is newer than "プロファイル"
"ファイル"は"プロファイル"より後に更新されました。プロファイル情報を無視します。

L1500 (W)Cannot check stack size
スタックセクションがないため、コンパイル時の stack オプションで指定したスタックサイズの整合性をチェックできません。コンパイル時の stack オプションの整合性をチェックするためにはコンパイル時、アセンブル時に goptimize オプション指定が必要です。

L1501 (W)Stack size overflow : "スタックサイズ"
スタックセクションサイズが、コンパイル時に stack オプションで指定した"スタックサイズ"を超えました。コンパイル時のオプションを変更するか、スタック量を削減できるようにプログラムを変更してください。

L1502 (W)Stack size in "ファイル" conflicts with that in another file
複数のファイルで異なるスタックサイズを指定されています。コンパイル時のオプションを確認してください。

P1600 (W)An error occurred during name decoding of "インスタンス"
"インスタンス"はデコードできませんでした。エンコード名でメッセージ出力します。

L2000 (E)Invalid option : "オプション"
"オプション"はサポートしていません。

L2001 (E)Option "オプション" cannot be specified on command line
"オプション"はコマンドライン上では指定できません。サブコマンドファイル内で指定してください。

L2002 (E)Input option cannot be specified on command line
コマンドライン上で input オプションを指定しました。コマンドライン上での入力ファイル指定は input オプション無しで指定してください。

L2003 (E)Subcommand option cannot be specified in subcommand file
サブコマンドファイル内に subcommand オプションを指定しました。subcommand オプションはネストできません。

L2004 (E)Option "オプション 1" cannot be combined with option "オプション 2"
"オプション 1"と"オプション 2"は同時に指定できません。

L2005 (E)Option "オプション" cannot be specified while processing "プロセス"
"プロセス"処理に対して"オプション"は指定できません。

L2006 (E)Option "オプション 1" is ineffective without option "オプション 2"
"オプション 1"は"オプション 2"が必要です。

L2010 (E)Option "オプション" requires parameter
"オプション"はパラメタ指定が必要です。

L2011 (E)Invalid parameter specified in option "オプション" : "パラメタ"
"オプション"で無効なパラメタを指定しました。

L2012 (E)Invalid number specified in option "オプション" : "値"
"オプション"指定で無効な値を指定しました。値の範囲を確認してください。

L2013 (E)Invalid address value specified in option "オプション" : "アドレス"
"オプション"で指定した"アドレス"は無効な値です。0～FFFFFFFF の間の 16 進数で指定してください。

L2014 (E)Illegal symbol/section name specified in "オプション" : "名前"
"オプション"で指定したセクションまたはシンボル名に不正文字が使用されています。セクション/シンボル名で使用できるのは数字、英字、_、$(先頭は数字以外)です。

L2020 (E)Duplicate file specified in option "オプション" : "ファイル"
"オプション"指定で同じファイルを 2 度指定しました。

L2021 (E)Duplicate symbol/section specified in option "オプション" : "名前"
"オプション"指定で同じシンボル名またはセクション名を 2 度指定しました。

L2022 (E)Address ranges overlap in option "オプション" : "アドレス範囲"
"オプション"で指定した"アドレス範囲"が重複しています。

L2100 (E)Invalid address specified in cpu option : "アドレス"
cpu オプションで cpu では指定できないアドレスを指定しました。

L2101 (E)Invalid address specified in option "オプション" : "アドレス"
"オプション"で指定した"アドレス"は cpu で指定できるアドレス範囲、または cpu オプションで指定した範囲を超えました。

L2110 (E)Section size of second parameter in rom option is not 0 :
"セクション"
rom オプションの第 2 パラメタにサイズが 0 でない"セクション"を指定しました。

L2111 (E)Absolute section cannot be specified in rom option : "セクション"
rom オプションで絶対アドレスセクションを指定しました。

L2120 (E)Library "ファイル" without module name specified as input file
入力ファイルとしてモジュール名なしのライブラリファイルを指定しました。

L2121 (E)Input file is not library file : "ファイル(モジュール)"
入力ファイルで指定した"ファイル(モジュール)"はライブラリファイルではありません。

L2130 (E)Cannot find file specified in option "オプション" : "ファイル"
"オプション"で指定したファイルが見つかりません。

L2131 (E)Cannot find module specified in option "オプション" : "モジュール"
"オプション"で指定したモジュールがありません。

L2132 (E)Cannot find "名前" specified in option "オプション"
"オプション"で指定したシンボルまたはセクションが存在しません。

L2133 (E)Cannot find defined symbol "名前" in option "オプション"
"オプション"で指定した外部定義シンボルが存在しません。

L2140 (E)Symbol/section "名前" redefined in option "オプション"
"オプション"で指定したシンボル、セクションはすでに定義されています。

L2141 (E)Module "モジュール" redefined in option "オプション"
"オプション"で指定したモジュールはすでに登録されています。

L2200 (E)Illegal object file : "ファイル"
P2200 ELF フォーマット以外を入力しました。

L2201 (E)Illegal library file : "ファイル"
"ファイル"はライブラリファイルではありません。

L2202 (E)Illegal cpu information file : "ファイル"
"ファイル"は cpu 情報ファイルではありません。

L2203 (E)Illegal profile information file : "ファイル"
"ファイル"はプロファイル情報ファイルではありません。

L2210 (E)Invalid input file type specified for option "オプション" :
"ファイル(種別)"
"オプション"指定時に処理できないファイル(種別)を入力しました。

L2211 (E)Invalid input file type specified while processing "プロセス" :
"ファイル(種別)"
"プロセス"処理に対して処理できないファイル(種別)を入力しました。

L2220 (E)Illegal mode type "モード種別" in "ファイル"
異なるモード種別のファイルを入力しました。

L2221 (E)Section type mismatch : "セクション"
属性(初期値有無)の異なる同名セクションを入力しました。

L2300 (E)Duplicate symbol "シンボル" in "ファイル"
"シンボル"は重複しています。

L2301 (E)Duplicate module "モジュール" in "ファイル"
"モジュール"は重複しています。

L2310 (E)Undefined external symbol "シンボル" referenced in "ファイル"
"ファイル"内で未定義の"シンボル"を参照しています。

L2311 (E)Section "セクション1" cannot refer to overlaid section :
"セクション2"-"シンボル"
同一アドレスを指定したオーバレイセクション間でシンボル参照がありました。
"セクション1"と"セクション2"を同じアドレスに割り付けないでください。

L2320 (E)Section address overflowed out of range : "セクション"
"セクション"のアドレスが使用可能なアドレス範囲を超えました。

L2321 (E)Section "セクション1" overlaps section "セクション2"
"セクション1"と"セクション2"のアドレスが重複しました。start オプションのアドレス指定を変更してください。

L2330 (E)Relocation size overflow : "ファイル"-"セクション"-"オフセット"
リロケーション演算結果がリロケーションサイズを超えました。ブランチ先が届かない、特定のアドレスに配置しなければならないシンボルを参照しているなどが考えられます。コンパイル、アセンブルリストで、"セクション"の"オフセット"位置の参照シンボルが正しい位置に配置されているか確認してください。

L2331 (E)Division by zero in relocation value calculation :
"ファイル"-"セクション"-"オフセット"
リロケーション演算に0除算が発生しました。コンパイル、アセンブルリストで、"セクション"の"オフセット"位置の演算に問題がないか確認してください。

L2332 (E)Relocation value is odd number :
"ファイル"-"セクション"-"オフセット"
リロケーション演算結果が奇数になりました。コンパイル、アセンブルリストで、"セクション"の"オフセット"位置の演算に問題がないか確認してください。

L2340 (E)Symbol name in section "セクション" is too long
fsymbol で指定した"セクション"内のシンボルの文字数が8174文字を超えました。

L2400 (E)Global register in "ファイル" conflicts : "シンボル","レジスタ"
"ファイル"内で指定したグローバルレジスタにはすでに別のシンボルが割りついています。

L2401 (E)__near8, __near16 symbol "シンボル" is outside near memory area
"シンボル"は__near8、__near16 の範囲に割りついていません。start 指定を変更するか、コンパイル時の__near 指定を外して、正しいアドレス計算ができるようにしてください。

L2402 (E)Number of register parameter conflicts with that in another file : "関数"
"関数"は複数のファイルで異なるレジスタパラメタ数を指定されています。

L2410 (E)Address value specified by map file differs from one after linkage as to "シンボル"
"シンボル"のアドレス値がコンパイル時に使用した外部シンボル割り付け情報ファイル内のアドレスとリンク後のアドレスで異なっています。
コンパイル時の map オプション指定前後でプログラムを変更していないか確認してください。または、optlnk の最適化によって、コンパイル時の map オプション指定前後のシンボル並び順が変ることがあります。コンパイル時 map オプションを無効にするか、optlnk の最適化オプションを無効にしてください。

L2411 (E)Map file in "ファイル" conflicts with that in another file
入力ファイル間でコンパイル時に異なる外部シンボル割り付け情報ファイルを使用しています。

L2412 (E)Cannot open file : "ファイル"
"ファイル"(外部シンボル割り付け情報ファイル)がオープンできません。ファイル名およびアクセス権が正しいか確認してください。

L2413 (E)Cannot close file : "ファイル"
"ファイル"(外部シンボル割り付け情報ファイル)がクローズできません。ディスク容量に空きがない可能性があります。

L2414 (E)Cannot read file : "ファイル"
"ファイル"(外部シンボル割り付け情報ファイル)が読みこめません。ディスク容量に空きがない可能性があります。

L2415 (E)Illegal map file : "ファイル"
"ファイル"(外部シンボル割り付け情報ファイル)のフォーマットが不正です。ファイル名が正しいか確認してください。

L2416 (E)Order of functions specified by map file differs from one after linkage as to "関数名"
関数"関数名"は、コンパイル時に使用した外部シンボル割り付け情報ファイル内の情報とリンク後の配置とで、他の関数との並び順が異なっています。関数内 static 変数のアドレスが、外部シンボル割り付け情報ファイルとリンク後の結果とで異なっている可能性があります。

P2500 (E)Cannot find library file : "ファイル"
ライブラリとして指定した"ファイル"がありません。

P2501 (E)"インスタンス" has been referenced as both an explicit specialization and a generated instantiation
すでに定義が存在しているインスタンスに対して、インスタンス生成を要求しています。"インスタンス"を使用しているファイルに対して、form=relocate でリロケータブルオブジェクトファイルを作成していないか確認してください。

P2502 (E)"インスタンス" assigned to "ファイル 1" and "ファイル 2"
"ファイル 1"と"ファイル 2"に"インスタンス"定義が重複しています。"インスタンス"を使用しているファイルに対して、form=relocate でリロケータブルオブジェクトファイルを作成していないか確認してください。

L3000 (F)No input file
入力ファイルがありません。

L3001 (F)No module in library
ライブラリ内のモジュール数が 0 になりました。

L3002 (F)Option "オプション 1" is ineffective without option "オプション 2"
"オプション 1"は"オプション 2"が必要です。

L3100 (F)Section address overflow out of range : "セクション"
"セクション"のアドレスが使用可能な上限の領域をを超えました。
start オプションのアドレス指定を変更してください。
アドレス空間の詳細については各 CPU のハードウェアマニュアルを参照してください。

L3101 (F)Section "セクション 1" overlaps section "セクション 2"
"セクション 1"と"セクション 2"のアドレスが重複しました。start オプションのアドレス指定を変更してください。

L3102 (F)Section contents overlap in absolute section "セクション"
絶対アドレスセクションのセクション内データアドレスが重複しています。ソースプログラムを修正してください。

L3110 (F)Illegal cpu type "cpu 種別" in "ファイル"
異なる cpu 種別のファイルを入力しました。

L3111 (F)Illegal encode type "エンディアン種別" in "ファイル"
異なるエンディアン種別のファイルを入力しました。

L3112 (F)Invalid relocation type in "ファイル"
"ファイル"内にサポートしていないリロケーションタイプがありました。コンパイラ、アセンブラのバージョンが正しいか確認してください。

L3200 (F)Too many sections
セクション数が限界値を超えました。複数ファイル出力を指定すると解決できる可能性があります。

L3201 (F)Too many symbols
シンボル数が限界値を超えました。複数ファイル出力を指定すると解決できる可能性があります。

L3202 (F)Too many modules
モジュール数が限界値を超えました。ライブラリを分けて作成してください。

L3300 (F)Cannot open file : "ファイル"
P3300 "ファイル"をオープンできません。ファイル名およびアクセス権が正しいか、確認してください。

L3301 (F)Cannot close file : "ファイル"
"ファイル"をクローズできません。ディスク容量に空きがない可能性があります。

L3302 (F)Cannot write file : "ファイル"
"ファイル"に書きこめません。ディスク容量に空きがない可能性があります。

L3303 (F)Cannot read file : "ファイル"
P3303 "ファイル"を読めません。空ファイルを入力したか、ディスク容量に空きがない可能性があります。

L3310 (F)Cannot open temporary file
P3310 中間ファイルをオープンできません。HLNK_TMP 指定が正しいか確認してください。またはディスク容量に空きがない可能性があります。

L3311 (F)Cannot close temporary file
中間ファイルをクローズできません。ディスク容量に空きがない可能性があります。

L3312 (F)Cannot write temporary file
中間ファイルに書きこめません。ディスク容量に空きがない可能性があります。

L3313 (F)Cannot read temporary file
中間ファイルを読めません。ディスク容量に空きがない可能性があります。

L3314 (F)Cannot delete temporary file
中間ファイルを削除できません。ディスク容量に空きがない可能性があります。

L3320 (F)Memory overflow
P3320 最適化リンケージエディタが内部で使用するメモリが不足しています。メモリを増やしてください。

L3400 (F)Cannot execute "ロードモジュール"
"ロードモジュール"を起動できません。"ロードモジュール"のパスが設定されているか確認してください。

L3410 (F)Interrupt by user
標準入力端末から「(cntl)+C」キーによる割り込みを検出しました。

L3420 (F)Error occurred in "ロードモジュール"
"ロードモジュール"実行中にエラーが発生しました。

P3500 (F)Bad instantiation request file -- instantiation assigned to more than one file
インスタンス生成指定ファイルに誤りがあります。
リンク対象ファイルを再コンパイルしてください。

P3501 (F)Instantiation loop
インスタンス生成処理がループしています。
入力ファイル名が別ファイルのインスタンス生成要求ファイルと一致している可能性があります。拡張子を除いたファイル名が一致しないようにファイル名を変更してください。

P3502 (F)Cannot create instantiation request file "ファイル"
インスタンス生成指定ファイルを作成できません。
オブジェクト作成ディレクトリ以下のアクセス権が正しいか確認してください。

P3503 (F)Cannot change to directory "ディレクトリ"
"ディレクトリ"に移動できません。
"ディレクトリ"が存在するか確認してください。

P3504 (F)File "ファイル" is read-only
"ファイル"は読み取り専用です。
アクセス権を変更してください。

L4000 (-)Internal error : ("内部エラー番号") "ファイル 行番号" / "コメント"
P4000 最適化リンケージエディタの処理中に内部的な問題が発生しました。
メッセージ内の内部エラー番号、ファイル、行番号、コメントを添えて、販売元のサポートセンターまでご連絡ください。

15. 標準ライブラリ構築ツール・フォーマットコンバータのエラーメッセージ

15.1 エラー形式とエラーレベル

本章では、以下の形式で出力するエラーメッセージとエラー内容を説明します。

エラー番号　　(エラーレベル）エラーメッセージ
エラー内容

エラーレベルは、エラーの重要度に従い、3 種類に分類されます。

		エラーレベル	動作
G1000 - G1999	(W)	ウォーニング	処理を継続します。
G2000 - G2999	(E)	エラー	オプション解析処理を継続し、処理を中断します。
G3000 - G3999	(F)	フェータル	処理を中断します。

15.2 メッセージ一覧

G1001 (W)Debug information ignored

変換対象ファイルに#pragma option により、最適化あり指定の関数と最適化なし指定の関数が混在しました。デバッグ情報を削除して変換します。

G1002 (W)Command parameter specified twice

同じオプションを二回以上指定しています。同じオプションの中で、最後に指定したものを有効とします。オプションの指定に誤りが無いかどうか確認してください。

G2001 (E)Cannot open file "ファイル"

ファイルをオープンできません。ファイル名およびアクセス権が正しいか確認して下さい。

G2002 (E)Illegal file type "ファイル"

SYSROF から ELF への変換の場合、オブジェクトファイルまたはライブラリファイル以外のファイルが指定されました。ELF から SYSROF への変換の場合、ロードモジュールファイル以外のファイルが指定されました。ファイルの種別を確認の上、再実行して下さい。

G2003 (E)Illegal file format "ファイル"
ファイルのフォーマットが不正です。ファイルの内容を確認の上、再実行して下さい。

G3001 (F)Invalid command parameter "パラメタ"
不正なコマンドパラメタが指定されました。コマンドパラメタの内容を確認の上、再実行して下さい。

G3002 (F)No input file
入力ファイルがありません。

G3003 (F)Command parameter buffer overflow
コマンドラインの指定が 32767 文字をこえています。

G3101 (F)Cannot open file "ファイル"
ファイルをオープンできません。ファイル名およびアクセス権が正しいか確認してください。

G3102 (F)Cannot input file "ファイル"
指定されたファイルから入力できません。変換対象ファイルをアクセスしていないか確認してください。

G3103 (F)Cannot create file "ファイル"
ファイルを生成できません。ディスク容量に空きがあるか確認してください。

G3104 (F)Cannot output file "ファイル"
ファイルに書き込めません。書き込み禁止を解除してください。

G3105 (F)Cannot open internal file
内部で生成する中間ファイルをオープンすることができません。中間ファイルをアクセスしていないか確認してください。

G3106 (F)Cannot output internal file
内部で生成する中間ファイルに出力できません。ディスク容量に空きがないか、ディスクに物理的なエラーが有る場合があります。

G3107 (F)Memory overflow
内部で使用するメモリ領域を割り当てることができません。必要なメモリを確保して再実行してください。

G3108 (F)Illegal format in archive "ファイル"
　　指定されたファイルはアーカイブのフォーマットではありません。

G3109 (F)Cannot fined "ファイル名"
　　ファイルを見つけることができません。環境変数 PATH の設定を確認してください。

G3201 (F)Cannot execute compiler
　　コンパイラを起動できません。コンパイラのパス名および環境変数を確認してください。

G3202 (F)Cannot execute optlinker
　　最適化リンケージエディタを起動できません。最適化リンケージエディタのパス名を確認してください。

G3203 (F)Interrupt by user
　　実行中に割り込みを検出しました。

G3204 (F)Cannot execute assembler
　　アセンブラを起動できません。アセンブラのパス名を確認してください。

G3300 (F)Already existent file "ファイル"
　　ファイルは既に存在しています。

16. 限界値

16.1 コンパイラの限界値

コンパイラの限界値を**表** 16.1 に示します。
ソースプログラムを作成する際は、この限界値の範囲で作成してください。

表 16.1 **コンパイラの限界値**

	分類	項目			限界値
1	起動	define **オプションが指定可能なマクロ名総数**			**制限なし**
2		**ファイル名長**			**制限なし**(OS **に依存**)
3	ソース	1 **行の文字数**			16384 **文字**
4	プログラム	1 **ファイルあたりのソースプログラムの行数**			**制限なし**
5		**コンパイル可能なソースプログラムの総行数**			**制限なし**
6	プリプロセッサ	#include **文のネストの深さ**			**制限なし**
7		#define **文のマクロ名総数**			**制限なし**
8		**マクロ定義、マクロ呼び出しの指定可能引数**			**制限なし**
9		**マクロ名の再置き換えの数**			**制限なし**
10		#if, #ifdef, #ifndef, #else, #elif **文のネストの深さ**			**制限なし**
11		#if, #elif **文で指定可能な演算子、非演算子の合計数**			**制限なし**
12	宣言	**関数定義数**			**制限なし**
13		**外部結合となる識別子(外部名)の数**			**制限なし**
14		1 **関数内で有効な識別子(内部名)の数**			**制限なし**
15		**基本型を修飾するポインタ型、配列型、関数型の合計数**			16 **個**
16		**配列の次元数**			6 **次元**
17		**配列・構造体のサイズ** *1	H8SX H8/2600 H8/2000 H8/300H H8/300	**ノーマルモード**	65,535 **バイト**
			H8SX	**ミドルモード**	32,767 **バイト**
			H8SX	**アドバンストモード**(ptr16 **有**)	
			H8SX	**マキシマムモード**(ptr16 **有**)	
			H8/300H	**アドバンストモード**	16,777,215 **バイト**
			H8SX	**アドバンストモード**(ptr16 **無**)	4,294,967,295 **バイト**
			H8SX	**マキシマムモード**(ptr16 **無**)	
			H8/2600	**アドバンストモード**	
			H8/2000	**アドバンストモード**	
18	文	**複文のネストの深さ**			**制限なし**
19		**繰り返し文**(while **文**、do **文**、for **文**)、**選択文**(if **文**、switch **文**)**の組み合わせによるネストの深さ**			256 **レベル**
20		1 **関数内で指定可能な** goto **ラベルの数**			511 **個**
21		switch **文の数**			2048 **個** *4
22		switch **文のネストの深さ**			128 **レベル**
23		**１つの** switch **文内で指定可能な** case **ラベルの数**			511 **個**
24		for **文のネストの深さ**			128 **レベル**

	分類	項目	限界値
25	式	文字列の長さ	32766 文字
26		関数定義、関数呼び出しで指定可能引数	63 個 *2
27		1 つの式で指定可能な演算子と非演算子の合計数	約 500 個
28	標準ライブラリ	open 関数で一度にオープンできるファイルの数	可変 *3

【注】 *1 アドバンスト、ミドル、マキシマムモードの場合、アドレス空間のビット幅を指定すると、アドレス空間のビット幅に対応するアドレス空間サイズが優先します。H8SX のアドバンスモードおよびマキシマムモードの時 ptr16 オプション有無により値が変わります。

*2 非静的関数メンバの場合は 62 個になります。

*3 詳細は「9.2.2(5) C/C++ライブラリ関数の初期設定(_INITLIB)」を参照して下さい。

*4 本バージョンの変更点。

16.2 アセンブラの限界値

アセンブラの限界値を表 16.2 に示します。

表 16.2 アセンブラの限界値

	項目	限界値		
1	1 行文字数	8192 文字		
2	文字定数	4 文字まで		
3	シンボル長	制限なし *1		
4	シンボル数	制限なし		
5	外部参照シンボル数	制限なし		
6	外部定義シンボル数	制限なし		
7	セクションの最大サイズ *2	H8SX	キシマムモード	H'FFFFFFFF バイトまで
		H8SX	ドバンストモード	H'FFFFFFFF バイトまで
		H8SX	ドルモード	H'00FFFFFF バイトまで
		H8SX	ーマルモード	H'0000FFFF バイトまで
		H8S/2600	ドバンストモード	H'FFFFFFFF バイトまで
		H8S/2600	ーマルモード	H'0000FFFF バイトまで
		H8S/2000	ドバンストモード	H'FFFFFFFF バイトまで
		H8S/2000	ーマルモード	H'0000FFFF バイトまで
		H8/300H	ドバンストモード	H'00FFFFFF バイトまで
		H8/300H	ーマルモード	H'0000FFFF バイトまで
		H8/300		H'0000FFFF バイトまで
		H8/300L		H'0000FFFF バイトまで
8	セクション数	goptimize オプション指定時	デバッグあり	H'FEF1 個
			デバッグなし	H'FEFA 個
		goptimize オプション未指定時	デバッグあり	H'FEF2 個
			デバッグなし	H'FEFB 個
9	ファイルインクルード	ネストは 30 レベルまで		
10	文字列長	255 文字まで		
11	ファイル名長	制限なし(OS に依存)		

【注】 *1 プリプロセッサ変数名、DEFINE 置換シンボル名、マクロ名および、マクロ仮引数名は、32 文字までです。

*2 セクションの最大サイズは、アドレス空間の指定により異なります。

17. バージョンアップにおける注意事項

17.1 バージョンアップ時の注意事項

旧バージョン(H8S,H8/300 シリーズ C/C++コンパイラパッケージ Ver.3 台とそれ以前)からバージョンアップして使用する場合の注意事項を説明します。

17.1.1 プログラムの動作保証

コンパイラをバージョンアップしてプログラム開発する場合、プログラムの動作が変わることがあります。プログラムを作成する際は、以下の点に注意して、お客様のプログラムを十分にテストしてください。

(1) プログラムの実行時間やタイミングに依存するプログラム

言語仕様は、プログラムの実行時間については何も規定していません。したがってコンパイラのバージョンの違いによりプログラムの実行時間と I/O 等周辺機器のタイミングのずれ、あるいは割り込み処理のような非同期処理の時間の差等により、プログラムの動作が変わる場合があります。

(2) 一つの式に 2 個以上の副作用が含まれているプログラム

一つの式に 2 個以上の副作用が含まれている場合、コンパイラのバージョンにより、動作が変わる可能性があります。

例：

```
a[i++]=b[i++];              /* i のインクリメント順序は不定です。*/
f(i++, i++);                /* インクリメントの順序でパラメタの値が変わります。*/
                            /* i の値が 3 の時 f(3, 4) または f(4, 3) になります。*/
```

(3) 結果がオーバフローや不当演算に依存するプログラム

オーバフローが生じた場合や、不当演算を実施した場合、結果の値は保証しません。したがって、バージョンが変わると動作が変わる可能性があります。

例：

```
int a, b;
x=(a*b)/10;   /* a と b の値の範囲によってはオーバフローする可能性があります */
```

(4) 変数の初期化抜け、型の不一致

変数が初期化されていない場合や、パラメタやリターン値の型が呼び出し側と呼び出される側で対応していない場合、不正な値をアクセスすることになります。したがって、コンパイラのバージョンによって動作が変わる場合があります。

例：

file 1:

```
int f(double d)
{
        :
}
```

file 2:

```
int g(void)
{
        f(1);
}
```

関数呼び出し側のパラメタは int型ですが、関数定義側のパラメタは、double型のため、値を正しく参照できません。

上記に記載された情報が全ての起こりうる状況を示したわけではありません。したがって、お客様の責任で本コンパイラを正しくご使用の上、お客様のプログラムを十分にテストしてください。

17.1.2 旧バージョンとの互換性

旧バージョン(Ver.3 台とそれ以前)のコンパイラ、およびそれに付随するアセンブラおよびリンケージエディタ出力のオブジェクトファイル、ライブラリファイルとリンクする場合、または旧バージョンで使用していたデバッガをそのまま使用する場合に注意すべき点を説明します。

(1) オブジェクト形式 (Ver.4 から)

形式は、従来の SYSROF から標準フォーマットの ELF 形式に変更しました。また、デバッグ情報形式も、標準フォーマットの DWARF2 形式に変更しました。

旧バージョン(Ver.3 台とそれ以前）のコンパイラ、アセンブラ出力オブジェクトファイル(SYSROF)を最適化リンケージエディタに入力する場合は、ファイルコンバータを使用して ELF 形式に変換してください。但し、リンケージエディタ出力リロケータブルファイル（拡張子 rel)およびリロケータブルファイルを含むライブラリファイルは変換できません。

また、SYSROF 形式および ELF/DWARF1 形式ロードモジュールをサポートするデバッガを使用する場合は、ファイルコンバータを使用して ELF/DWARF2 形式ロードモジュールを、SYSROF または ELF/DWARF1 形式に変換してください。但し、#pragma option(Ver.4 の追加機能）を用いて同一ファイル内で最適化有り、無しの関数を混在した場合には、デバッグ情報部分は変換できません。オブジェクト部分のみの変換になります。

(2) 関数インタフェース変更オプションの追加 (Ver.4 から)

関数インタフェース規則を変更するオプションとして、structreg、longreg が追加になりました。このオプションを指定する場合は、全てリコンパイルしてください。また、アセンブラルーチンとのインタフェースも修正してください。

(3) スタック領域 (Ver.4 から)

stack オプションで、スタック計算サイズを指定できるようになりました。

オプション省略時は stack=medium（2byte でスタック計算を行う）が有効になりますので、変更が必要な場合は、stack オプションで指定してください。

(4) const データ出力セクション (Ver.4 から

Ver3 台では volatile オプション指定時、const 宣言のある変数を D セクションに出力していましたが、Ver4 では C セクションに出力するよう変更しました。

(5) データ配置 (Ver.4 から)

align/noalign オプションで、データを境界調整毎に並べ替えることができるようになりました。

オプション省略時は align(境界調整毎に並べ替えを行う）が有効になりますので、並べ替えを抑止する場合は、noalign オプションを指定してください。

(6) $ABS8C, $ABS8D, $ABS8B セクションの境界調整 (Ver.4 から)

#pragma abs8 や_ _abs8、abs8 オプション指定時に出力される$ABS8C, $ABS8D, $ABS8B セクションの境界調整数を従来の２から１に変更しました。

これに伴い、#pragma abs8 や_ _abs8、abs8 オプション指定時に有効となる変数の条件が、「char, unsigned char 型変数／配列または char, unsigned char 型変数／配列をメンバに持つ構造体／クラス」から「境界調整１の変数／配列／構造体／クラス」に変更になりました。

(7) インクルードファイルの基点 (Ver.4 から)

chgincpath オプションを廃止し、ディレクトリ相対形式で指定されたインクルードファイル検索時、常にソースファイルのあるディレクトリを基点に検索するように変更しました。

(8) C++プログラム (Ver.4 から)

エンコード規則、実行方式を変更しましたので、旧バージョンコンパイラで作成した C++オブジェクトファイルはリンクできません。必ずリコンパイルしてから使用してください。

また実行環境の設定で用いる、グローバルクラスオブジェクト初期処理／後処理のライブラリ関数名も変更になりました。**「9.2.2 実行環境の設定」**を参照し、修正してください。

(9) コモンセクションの廃止（アセンブリプログラム Ver.4 から）

オブジェクトフォーマットの変更に伴い、コモンセクションのサポートを廃止しました。

(10) .end 制御命令のエントリ指定（アセンブリプログラム Ver.4 から）

.end 制御命令でエントリ指定できるシンボルは外部定義シンボルだけになりました。

(11) モジュール間最適化 (Ver.4 から)

旧バージョンのコンパイラ、アセンブラ出力オブジェクトファイルは、モジュール間最適化の対象になりません。モジュール間最適化の対象にしたいファイルについては、必ずリコンパイル、リアセンブルしてください。

(12) オプションの統一

次に示すコンパイラオプションは旧バージョンと Ver.6 とで同じにしてください。

cpu, regparam, pack, structreg/nostructreg, longreg/nolongreg, stack, double=float, rtti, exception

但し、旧バージョンのオブジェクトファイルとVer.6のオブジェクトファイルとをリンクする場合、次に示すVer.6のオプションを使用しないでください。

byteenum, bit_order=right, indirect=extended, ptr16, sbr

また、Ver.4 より前のバージョンのオブジェクトファイルと Ver.6 のオブジェクトファイルとをリンクする場合、Ver.4 からサポートされた以下のオプションを使用しないでください。

structreg, longreg, stack=small/medium, double=float, rtti, exception

また、Ver.3 より前のバージョンのオブジェクトファイルと Ver.6 のオブジェクトファイルとをリンクする場合、Ver.3 からサポートされた以下のオプションを使用しないでください。

regparam=3, pack=1

17.1.3 コマンドラインインタフェース

(1) アセンブラ(Ver.4)、最適化リンケージエディタ(Ver.7)コマンドライン指定方法

ファイル名、オプション間に、スペースが必須になりました。
また、ファイル名、オプションの指定順序に制限がなくなりました。

(2) 最適化リンケージエディタオプション (Ver.7 から)

会話形式のオプション指定サポートを廃止しました。また、旧バージョンのモジュール間最適化ツール(optlnk38)とリンケージエディタ (lnk)、ライブラリアン(lbr)、オブジェクトコンバータ (cnvs) を最適化リンケージエディタ(optlnk)に統合しました。これに伴い、コマンドライン仕様が大幅に変更になりました。変更したコマンド一覧を**表** 17.1、**表** 17.2 に示します。

表 17.1 リンケージコマンド変更一覧

No.	コマンド名	V6	V7	備考
1	start	start= セクション(アドレス) 短縮形 st	start= セクション/アドレス 短縮形 star	—
2	rom	rom=(rom セクション, ram セクション)	rom=rom セクション/ ram セクション	—
3	define	define=外部名(定義値)	define=外部名=定義値	—
4	rename	rename= ed=変更前(変更後), er=変更前(変更後), un=変更前(変更後) 短縮形 re	rename= (変更前=変更後), (変更前=変更後), — 短縮形 ren	オブジェクト形式変更によりユニットの概念廃止
5	delete	delete= ed=ユニット.シンボル un=ユニット	delete=(シンボル) —	オブジェクト形式変更によりユニットの概念廃止
6	print / noprint	print noprint	list —	ファイル名省略可
7	mlist	mlist	list	—
8	information	information	message	—
9	directory	directory	HLNK_DIR(環境変数)	—
10	form	短縮形 f	短縮形 fo	—
11	output / nooutput	短縮形 o nooutput 指定可	短縮形 ou nooutput 指定不可	output のみ指定可
12	cpu	短縮形 c	短縮形 cp	直接範囲指定可
13	elf / sysrof / sysrofplus	elf / sysrof / sysrofplus	廃止	常に ELF
14	exclude / noexclude	exclude / noexclude	廃止	常に exclude
15	align_section	align_section	廃止	常に有効*
16	check_section	check_section	廃止	常に有効*
17	cpucheck	cpucheck	廃止	常に有効*
18	udf / noudf	udf / noudf	廃止	常に出力*
19	udfcheck	udfcheck	廃止	常に有効*
20	echo / noecho	echo / noecho	廃止	常に抑止
21	exchange	exchange	廃止	オブジェクト形式変更によりユニットの概念廃止
22	autopage	autopage	廃止	対象 cpu なし
23	abort	abort	廃止	会話形式廃止
24	list	list	廃止	V7 の list オプションとは別
25	library / nolibrary	nolibrary 指定可	nolibrary 指定不可	library のみ指定可
26	exit	省略不可	省略可	—
27	debug / nodebug	省略時：nodebug	省略時： 入力ファイルの debug 情報有無に依存	—

【注】* change_message オプションで無効にすることができます。

表 17.2 ライブラリアンコマンド変更一覧

No.	コマンド名	V2	V7	備考
1	add	add	input	—
2	directory	directory	HLNK_DIR(環境変数)	—
3	slist	slist	list show	—
4	list	list (s)	list show	—
5	delete	短縮形 d	短縮形 del	—
6	create	create (s | u)	library form=library(s | u)	—
7	output	output (s | u)	output form=library(s | u)	—
		短縮形 o	短縮形 ou	
8	replace	短縮形 r	短縮形 rep	—
9	abort	abort	廃止	会話形式廃止
10	exit	省略不可	省略可	—

17.1.4 提供内容

「H8S,H8/300 シリーズ C/C++コンパイラパッケージ」の提供内容のうち、以下のファイルが Ver.4 パッケージで変更になりました。

(1) CPU 情報ファイル作成ツール

optlnk の cpu オプションで、アドレス範囲を直接指定できるようになりました。

旧バージョンの CPU 情報ファイル作成ツールで作成した cpu 情報ファイルはそのまま使用できます。CPU 情報を変更・作成する場合は、cpu オプションで直接アドレス範囲を指定してください。

(2) 標準ライブラリファイル

関数インタフェースや最適化オプションを任意に指定できるようにするため、従来の標準ライブラリファイル提供から、標準ライブラリ構築ツール提供に変更しました。

(3) ヘッダファイル

bool 型サポートに伴い、defbool.h を削除しました。

17.1.5 リストファイル仕様

(1) コンパイルリスト (Ver.4 から)

カラム数を見やすく変更しました。また、タブのカラム数を選択できるようにしました。

(2) 最適化リンケージエディタ (Ver.7 から)

従来のリンケージマップリスト、ライブラリリストのフォーマットを一新しました。

17.2 追加・改善内容

17.2.1 共通の追加・改善

(1) コンパイラ Ver.4,アセンブラ Ver.4,最適化リンケージエディタ Ver.7 の主な追加・改善機能

(a) 制限値の緩和

ソースプログラムやコマンドラインの制限を大幅に緩和しました。

- **ファイル名長：251 バイト → 無制限**
- **シンボル長：251 バイト → 無制限**
- **シンボル数：65,535 個 → 無制限**
- **ソースプログラム行数：C/C++:32,767 行、ASM:65,535 行 → 無制限**
- **C プログラム行長：8,192 文字 → 16,384 文字**
- **C プログラム文字列長：512 文字 → 16,384 文字**
- **サブコマンドファイル行長：ASM:300 バイト、optlnk:512 バイト →無制限**
- **最適化リンケージエディタ ROM オプションのパラメタ数:64 個 → 無制限**

(b) ディレクトリ名、ファイル名のハイフン(-)

ディレクトリ名、ファイル名にハイフン(-)を指定できるようになりました。

(c) コピーライト表示抑止

logo/nologo オプション指定により、コピーライト表示有無を指定できるようになりました。

(d) エラーメッセージのプリフィックス

統合開発環境でのエラーヘルプ機能サポートに伴い、コンパイラ、最適化リンケージエディタのエラーメッセージの先頭にプリフィックスを付与しました。

17.2.2 コンパイラの追加・改善

(1) Ver.4 の主な追加・改善機能

(a) キーワードサポート

キーワード(__interrupt, __indirect, __entry, __abs8, __abs16, __regsave,__noregsave, __inline, __global_register)を用いて、関数または変数の宣言および定義に対して属性を指定できるようになりました。

(b) ベクタテーブル生成機能

#pragma interrupt, #pragma indirect, #pragma entry および__interrupt, __indirect, __entry の vect 指定を用いて、自動的に関数のベクタテーブルを生成できます。

(c) __evenaccess サポート

__evenaccess で指定した定数、変数の偶数バイトアクセスを保証します。

(d) レジスタパラメタ指定拡張

__regparam2, __regparam3 を用いて、関数毎にレジスタパラメタ数を指定できます。

(e) 関数単位オプションの指定

#pragma option を用いて、関数単位でオプション指定ができます。

(f) データの near 配置サポート

__near8, __near16 を用いて、配列、構造体のアドレス計算コードを最適化することができます。但し、ポインタサイズは変わりません。

(g) スタックの near 配置サポート

stack オプションを用いて、スタック領域のスタックアドレス計算コードを最適化することができます。

(h) 組み込み関数の追加

次の組み込み関数を追加しました。

- **符号なしオーバフロー演算**

(i) double=float サポート

double=float オプションにより、double 型宣言データや浮動小数点定数を float 型として扱います。

(j) noregsave 関数のサポート強化

#pragma noregsave, __noregsave 宣言関数を呼び出す場合、呼び出し関数側でレジスタを保証するよう変更しました。

(k) 環境変数の複数指定

インクルードディレクトリ用環境変数（CH38)で、複数のディレクトリ指定ができます。

(l) 構造体パラメタ／リターン値のレジスタ渡し

structreg オプションを用いて、サイズの小さい構造体パラメタ／リターン値をレジスタで渡すことができます。

(m) 4byte パラメタ／リターン値のレジスタ渡し(cpu=300)

longreg オプションを用いて、4byte パラメタ／リターン値をレジスタで渡すことができます。

(n) 非 volatile 変数のループ外移動条件

ループ判定式にある非 volatile の外部変数は、ループ内で副作用（関数呼び出し、代入等）がなくても、常にループ外移動最適化を抑止します。

(o) speed=loop=1 | 2 のサポート

speed=loop=1|2 オプションにより、ループ展開最適化の実行を制御できます。

(p) アラインによるデータ割り付け変更

align オプションにより、データを境界調整毎に再配置し、境界調整による空きを最小限にできるようになりました。

(q) 暗黙の宣言の追加

_ _HITACHI_ _、_ _HITACHI_VERSION_ _などが暗黙に#define 宣言されます。

(r) static ラベル名

#pragma asm～#pragma endasm および#pragma inline_asm 関数内でファイルスコープの static ラベルを参照できるように、ラベル名を_ _$(名前)に変更しました。
ただし、リンケージリストでは_(名前)と表示されます。

(s) 言語仕様拡張・変更

- **union 初期化時のエラーを抑止します。**

例：

```
union{
    char c[4];
}uu={ {'a','b','c'} };
```

- **ビットフィールドに enum の記述ができるようになりました。**

例：

```
struct{
    enum E1{a,b,c} m1:2;
    enum E1        m2:2;
};
```

- **列挙子の最後の","に対して、エラー出力を抑止します。**

例：

```
enum E1{a,b,c,}m1;
```

- **共用体の代入と宣言を同時にできるようになりました。**

例：

```
union U{
    int a,b;
}u1;
void test(){
    union U u2 = u1;
}
```

- **C コンパイル時のアドレスに対するキャストのエラーチェックを緩和しました。**
  アドレスに対するキャストを記述する場合は、必ず C コンパイル(lang=c オプション)を指定してください。

例：

```
int x;
short addr1=(short)&x;
```

- **C プログラムの関数・変数宣言と#pragma 宣言の出現順の制約を緩和しました。**

例：

```
void f(void);
#pragma interrupt f
void f(void){}       // 関数宣言後の#pragma 宣言は有効になります。
                     // (Ver.3 台ではエラー)
```

- **C++プログラムの関数・変数宣言と#pragma 宣言の出現順の規約を変更しました。**

例：

```
void f(void){}
#pragma interrupt f  // 関数定義後の#pragma 宣言は無効になります。
void f(void);        // 関数定義後の#pragma 宣言がかかる関数宣言はエラーにな
                        ります。
```

- **C++言語仕様として、例外処理やテンプレート機能もサポートしました。**

(2) Ver.4 → Ver.6 **の主な追加・改善機能**

(※ Ver.5 は存在せず、欠番となります)

(a) **新** CPU **のサポート**

CPU 種別が H8SX のオブジェクトファイルの生成をサポートしました。

(b) 2byte **サイズポインタのサポート** (H8SX **のみ**)

_ _ptr16 キーワード指定か ptr16 オプション指定により 2byte サイズポインタが使用できます。H8SX のアドバンストモードとマキシマムモードで有効です。

(c) **ビットフィールド並び順指定**

#pragma bit_order 指定か bit_order オプション指定により、メモリに対するビットフィールドメンバの詰め込み方を指定できます。

(d) **拡張メモリ間接方式の関数呼び出し** (H8SX **のみ**)

_ _indirect_ex キーワード指定か indirect=extended オプション指定により、拡張メモリ間接呼び出しとなる関数を宣言できます。また、#pragma indirect section はメモリ間接(@@aa:8)呼び出し用関数アドレス領域の $INDIRECT セクションだけでなく、拡張メモリ間接(@@aa:7)呼び出し用関数アドレス領域の $EXINDIRECT セクションのセクション名を切り替えることができます。

(e) **アセンブル機能** (H8SX **のみ**)

_ _asm キーワード指定により、C/C++ソースプログラムの中にアセンブリ言語を記述することが可能です。

(f) #line 出力抑止指定

noline オプション指定により、プリプロセッサ展開時の#line の出力を抑止することができます。

(g) memcpy 関数、strcpy 関数のインライン展開指定 (H8SX のみ)

library オプション指定により、memcpy と strcpy の二つのライブラリ関数をインライン展開することができます。

(h) エラーレベルの変更

change_message オプション指定により、インフォメーションレベルとウォーニングレベルのメッセージは個別にエラーレベルの変更が可能です。

(i) 8bit 絶対領域のアドレス指定 (H8SX のみ)

sbr オプション指定により、8bit 絶対アドレス領域の配置アドレスを指定することが可能です。

(j) 最適化機能の強化 (H8SX のみ)

以下のオプションが追加されたことにより、最適化の内容をさらに詳細に指定することが可能です。

opt_range、del_vacant_loop、max_unroll、infinite_loop、global_alloc、struct_alloc、const_var_propagate、volatile_loop

(k) 組み込み関数の追加

以下の組み込み関数が追加されました。

- H8SX の 64bit 乗算 (mulsu, muluu)
- H8SX のブロック転送命令 (movmdb, movmdw, movmdl, movsd)
- ブロック転送命令組み込み関数強化 (eepmovb, eepmovw, eepmovi)
- MOVFPE 命令組み込み関数見直し (_movfpe)

(l) ワイルドカードのサポート

入力ファイルをワイルドカードで指定することが可能です。

(m) コンパイラ限界値の変更

1 つのファイルに記述できる switch 文を 256 から 2048 にしました。

(n) インフォメーションメッセージ表示の仕様変更

Ver.4 では message や nomessage オプションをコマンドラインに複数指定すると最後に指定したオプションだけが有効になりました。本バージョンではコマンドライン内の各 nomessage オプションが指定する番号の和集合の番号のメッセージが抑止されます。

(o) 列挙型データの型

byteenum オプション指定時に enum の値の範囲が 0～255 の場合に当該 enum 宣言した列挙型データを unsigned char 型として扱う仕様を追加しました。

(p) インライン展開

CPU 種別が H8SX の場合、speed=inline=<数値>オプションの<数値>の意味が、その他の CPU 種別と異なります。H8SX の場合、<数値>はプログラムサイズの増加率を表し、その他の CPU の場合、インライン展開できる関数のノード最大数を表します。

(q) 1 バイトアラインデータセクション、4 バイトアラインデータセクション (H8SX のみ)

align=4 オプションを指定するとサイズが奇数のデータを 1 バイトアラインのセクションへ、サイズが 4 の倍数のデータを 4 バイトアラインのセクションへ出力することができます。

(r) セクション名

section オプションで P、C、B、D セクションをセクション名 S に変更するとウォーニングエラーを発生します。S はスタック用に予約されたセクション名です。

(s) 暗黙の宣言の追加

_ _H8SXN_ _、_ _H8SXM_ _、_ _H8SXA_ _、_ _H8SXX_ _、_ _HAS _MULTIPLIER_ _、_ _HAS _DIVIDER_ _、_ _INTRINSIC_LIB_ _、_ _DATA_ADDRESS_SIZE_ _、_ _H8_ _、_ _RENESAS_VERSION_ _、_ _RENESAS_ _が新たに暗黙に#define 宣言されます。

(t) リエントラントライブラリサポート

ライブラリ構築ツールで reent オプションを指定した場合、リエントラントライブラリが生成されます。

(u) リトルエンディアン空間サポート (H8SX のみ)

H8SX は製品によりリトルエンディアン空間をサポートしています。リトルエンディアン空間の 2, 4 バイトのデータはデータサイズで書き込み、読み込みます。このために_ _evenaccess キーワードの機能を拡張しました。

(3) Ver.6 の最適化機能に関する注意事項

この最適化に関する注意事項は、V6 で H8SX のオブジェクトを生成するケースが対象です。それ以外のケースにおける最適化処理は V4 以前と同様となります。

V6 における最適化処理は最新のコンパイラ最適化技術を適用し、従来(V4)では実現できなかった、ポインタや外部変数の別名解析や、制御フローを含めたデータ生存区間解析を実現しています。これにより、言語仕様で許されている範囲で V4 よりも広範囲の最適化をしています。

一方、従来最適化対象にならなかったために動作していたプログラムが、最適化対象になったために動作しなくなるという場合もあります

従来最適化されなかったもので、V6 で最適化対象になる例を以下に示します。

(a) volatile 宣言のない外部変数、ポインタ変数のアクセスの扱い

volatile 宣言はプログラム上の逐次処理以外でデータの値が変更されるため、参照されるたびに必ず実際にデータが割り付けられた領域をアクセスすることを保証します。例えば、割り込み処理やハードウェア処理によりデータの値が変更される場合などが挙げられます。

コンパイラ処理では、volatile 宣言のない変数については、プログラム内の逐次処理および関数呼び出しによる変更以外は変更されることはないと解釈します。
V4 までは、以下のような例で volatile 宣言がない外部変数に対する最適化を行っていました。

例：

```
    int a;
    f() {
        int *ptr=&a;
        *ptr=1;          // <------ この代入式のみを削除
        *ptr=2;
    }
```

V6 では更に以下のケースの最適化を行います。
これらの最適化を抑止したい場合は当該変数を volatile 宣言してください。

例 1：

```
    int a;
    f() {
        int *ptr=&a;
        *ptr &= ~( (0x0080) );          // <-- (1)
        while( !( *ptr & (0x0080) ) )  // <-- (2)
        {
           :
        }
    }
```

この例では、最適化の結果(2)の while 文が無限ループになります。
・ポインタの別名解析により、(1)の*ptr と(2)の*ptr を同一値として扱います。
・(1)の式を(2)に伝播します。その結果、(2)式は以下のように変換されます。

```
     while( !( (*ptr & ~( (0x0080) )) & (0x0080) ) )  // <-- (2)
  -> while(!(*ptr & 0))
  -> while(!(0))
  -> while(1)
```

従って、当該式は常に真となり、判定文が削除され、上記 while 文は無限ループになります。

例 2：

```
    int a,b;
    f() {
        a=1;        // <-- (1)
        if(a)       // <-- (2)
        {
            b=1;    // <-- (3)
        }
 }
```

この例では、最適化の結果(2)の if 文判定式が削除され、常に(3)が実行されます。
・外部変数の別名解析により、(1)と(2)の a を同一値として扱います。
・(1)の定数値を(2)に伝播します。その結果、(2)式は以下のように変換されます。

```
  -> if(1)
```

従って、当該式は常に真となり、判定文が削除され、上記(3)式が常に実行されます。
例 3：

```
int a,b,c;
f() {
    a=1;          // <-- (1)
    if(c)         // <-- (2)
    {
        b=1;      // <-- (3)
    }
    a=2;          // <-- (4)
}
```

この例では、最適化の結果(1)の式が削除されます。
・if 文の制御式を含めた制御フローを求めます。
・制御フローと外部変数の別名解析により、a に(1)で設定した値は使用されていない
ことがわかります。従って、上記(1)式は参照されない冗長式となり、削除されます。

例 4：

```
int a;
int b[10];
f() {
    int i;                  // <-- (1)
    for(i=0; i<10; i++)     // <-- (2)
    {
        b[i]=a;             // <-- (3)
    }
}
```

この例では、最適化の結果(3)の a はループの前に一度だけ参照され、ループ内では常に一定値として扱われます。
・for ループ制御式を含めた制御フローを求めます。
・制御フローと外部変数の別名解析により、(3)の a がループ内で一定値として扱います。
・(3)の a の参照式を(2)の for ループ外に移動します。

```
temp=a;
for(i=0; i<10; i++)   // <-- (2)
{
    b[i]=temp;        // <-- (3)
}
```

従って、(3)式の変数 a はループ中は一定値となります。

例 5：

```
int a;
f() {
    a = 0;        // <-- (1)
    while (1);    // <-- (2)
}
```

この例では、最適化の結果(1)の文は不要とみなされ削除されます。
・ (2)は無限ループなので本関数は出口がないと判断します。
・ 無限ループ内で a の参照はないので、(1)の設定は不要コードとみなされ削除します。

(b) volatile_loop オプション

volatile_loop オプションは、ループ制御変数が非 volatile 外部変数でかつ判定式が単純な場合に、ループ制御変数を volatile として扱い、無限ループ化を抑止します。しかし、ループ制御変数がループ内不変でない場合は volatile 化の対象外になります。

V6 では、このような場合は当該変数を volatile 宣言してください。

以下に例を示します。

例：

```
struct {
    unsigned char a:1;
} ST;
int a;
extern void f();
void func() {
    while (ST.a) {       // <-- (1)
        if (a) {         // <-- (2)
            f();         // <-- (3)
        }
    }
}
```

この例の場合、ST.a は f()で書き換わる可能性があり、ループ内で不変とはみなされません。よって、volatile_loop オプションを指定しても ST は volatile 化されません。

・(2)の条件が成立した場合は、(3)が実行され f()で ST.a の値が書き換わる可能性があるため、呼び出し後 ST.a を再ロードします。

・(2)の条件が不成立の場合は、ST.a が書き換わらないので前回(1)の判定で使用した ST.a の値をそのまま使用します。

(4) V4 オブジェクトと V6 オブジェクトの互換性について

V4 オブジェクトと V6 オブジェクトをリンクするには、以下の条件を満たす必要があります。

(1) C ソースプログラム

以下の関数インタフェースに影響するオプションが同一であること

・reparam
・longreg/nolongreg
・double=float
・structreg/nostructreg
・stack
・byteenum
・pack/unpack

(2) アセンブリプログラム

「9.3.2 **関数呼び出しのインタフェース**」の関数呼び出し規約に準拠していること

【注１】

マニュアルに記述のない内容については、バージョンアップによる互換性は保証していません。レジスタの退避・回復順序等、コンパイラの出力コードに依存するアセンブリコードを記述している場合は、V4 オブジェクトと V6 オブジェクトをリンクできません。

【注２】

OS やミドルウェア等とのリンクについては、購入元にお問い合わせください。

17.2.3 アセンブラの追加・改善機能

(1) Ver.4 の主な追加・改善機能

(a) BEQU の外部定義・参照

.BIMPORT, .BEXPORT を用いて、.BEQU シンボルの外部定義、参照が可能になりました。

(2) Ver.4 → Ver.6 の主な追加・改善機能

(※ Ver.5 は存在せず、欠番となります)

(a) 新 CPU のサポート

CPU 種別が H8SX のオブジェクトファイルの生成をサポートしました。

(b) レジスタ使用方法のチェック強化

CPU 種別が H8SX、 H8S、H8/300H の場合に@Rn+, @-Rn, @+Rn, @Rn-, @(d,Rn)または@Rn とプログラムに記述すると以下のウォーニングが発生するように変更しました。

```
819  (W) @Rn+, @-Rn, @+Rn, @Rn-, @(d,Rn) OR @Rn USED
```

これらのアドレッシングモードでは Rn を ERn と書き換えてください。

17.2.4 最適化リンケージエディタの追加・改善機能

(1) Ver.7, Ver.7.1 での主な追加・改善機能

(a) ワイルドカードのサポート

入力ファイルや start オプションのセクション名でワイルドカードを指定できます。

(b) サーチパス

環境変数(HLNK_DIR)により、複数の入力ファイル、ライブラリファイルのサーチパスを指定できます。

(c) ロードモジュール分割出力

アブソリュートロードモジュールファイルを分割出力できます。

(d) エラーレベルの変更

インフォメーション、ウォーニング、エラーレベルのメッセージは、個別にエラーレベルや出力有無を変更できます。

(e) バイナリ、HEX サポート

バイナリファイルを入出力できるようになりました。
また、インテル HEX タイプの出力も選択できるようになりました。

(f) stack 使用量情報の出力

stack オプションにより、スタック解析ツール用情報ファイルを出力できます。

(g) optimize=variable_access 最適化の改善

16bit 絶対アドレス空間に配置した変数も、最適化により 8bit アドレス空間に移動できるようになりました。

(h) optimize=register 最適化の改善

optimize=speed オプションの指定がないとき、関数間のレジスタ退避・回復最適化後に、複数レジスタの退避・回復を関数呼び出しに置換して、サイズ圧縮を行います。

(i) アセンブリプログラム最適化の改善

.org、.align、.data 制御命令を含むセクションも最適化できるようになりました。

(j) デバッグ情報削除機能

strip オプションにより、ロードモジュールファイルやライブラリファイル内のデバッグ情報だけを削除できます。

(k) デバッグ情報圧縮機能

compress オプション指定により、デバッグ情報の圧縮が可能になりました。

(2) Ver.7 -> Ver.8 での主な追加・改善機能

(a) 新 CPU のサポート

CPU 種別が H8SX のオブジェクトファイルの入力をサポートしました。

(b) 空きエリア出力指定

space オプション指定により、空きエリアへ指定値を埋め込むことができます。

(c) メモリ使用量指定

memory オプション指定により、内部のメモリ使用量を指定することができます。

(d) 8bit 絶対領域のアドレス指定

sbr オプション指定により、8bit 絶対領域の配置アドレスを指定することが可能です。

(e) セクションアドレス重複時のエラーレベル変更

リンク時にセクションのアドレスが重なった場合のエラーレベルが Ver.7 では Fatal だったのを、Ver.8 では Error に変更しました。これにより、セクションアドレスが重複するような状況でも change_message オプション指定により、ユーザ責任において処理の継続が可能です。

17.3 フォーマットコンバータ操作方法

17.3.1 オブジェクトファイル形式

オブジェクトファイル形式は、標準フォーマットの ELF 形式に準拠しています。また、デバッグ情報形式は、標準フォーマットの DWARF2 形式に準拠しています。

17.3.2 旧バージョンとの互換性

(1) オブジェクトファイル、ライブラリファイル

Ver.3 台以前のコンパイラ、アセンブラ出力オブジェクトファイルおよびライブラリファイルを最適化リンケージエディタに入力する場合は、フォーマットコンバータを使用して ELF 形式に変換してください。但し、デバッグ情報は変換時に削除されます。

また、リンケージエディタ出力リロケータブルファイル（拡張子 rel)およびリロケータブルファイルを含むライブラリファイルは変換できません。

フォーマットコンバータは、オブジェクト形式を変換したファイルを、入力ファイル名と同じ名前で出力します。入力ファイルは、<入力ファイル名.拡張子>.bak として保存します。

ELF 形式のオブジェクトファイルおよびライブラリファイルを、旧バージョン(Ver3 台以前）のコンパイラ、アセンブラ出力オブジェクト形式に変換することはできません。

(2) ロードモジュールファイル

ELF 形式のロードモジュールファイルは、フォーマットコンバータを使用してリンケージエディタの Ver.6 台とそれ以前の出力のフォーマットに変換することができます。変換可能なオブジェクトファイル形式を、**表** 17.3 に示します。H8SX の ELF 形式のロードモジュールファイルはサポートしていません。

表 17.3 ELF **形式から変換可能なオブジェクトファイル形式**

	コンパイラ、アセンブラ バージョン	リンケージエディタ 指定オプション		オブジェクトファイル形式 オブジェクト	オブジェクトファイル形式 デバッグ情報	変換可否
1	1.0 台	debug		SYSROF	SYSROF	○
2	2.0 台	sdebug		SYSROF	SYSROF	×
3	3.0 台	sysrof	debug	SYSROF	SYSROF	○
4			sdebug	SYSROF	DWARF1	×
5		elf	debug	ELF	DWARF1	○
6			sdebug	ELF	DWARF1	×

フォーマットコンバータは、オブジェクト形式を変換したファイルを、入力ファイル名と同じ名前で出力します。入力ファイルは、<入力ファイル名.拡張子>.bak として保存します。

リンケージエディタの Ver.6 台とそれ以前の出力ロードモジュールファイルを、ELF 形式に変換することはできません。

Ver.4 パッケージ以降のバージョンのコンパイラ、アセンブラ、最適化リンケージエディタで新規追加した機能を使用した場合は、ELF 形式から旧形式へ変換できません。

17.3.3 オプション指定規則

コマンドラインの形式は以下の通りです。

```
helfcnv [△<オプション> … ]△<ファイル名>[…] [△<オプション> …]
```

<オプション>：- <オプション>[=<サブオプション>]

<ファイル名>：ワイルドカード(*,?)も指定できます。

17.3.4 オプション解説

コマンドライン形式の場合は、英大文字は短縮形指定時の文字を示します。下線は省略時解釈を示します。統合開発環境を使用する場合は、最適化リンケージエディタのオプションウィンドウで指定します。対応するダイアログメニューを、タブ名<カテゴリ名>[項目]…で示します。

変換対象ファイルの種別（オブジェクトファイル、ライブラリファイル、ロードモジュール）は、フォーマットコンバータが自動判定します。

(1) オブジェクトファイル、ライブラリファイルの変換

Ver.3 台以前のコンパイラ、アセンブラで作成したオブジェクトファイル、ライブラリファイルを、ELF 形式に変換します。オブジェクトファイル、ライブラリファイル内に含まれているデバッグ情報は削除されます。

本機能は、統合開発環境ではサポートしていません。コマンドラインで使用してください。

表 17.4 オブジェクトファイル、ライブラリファイル変換用オプション一覧

	項目	オプション	指定内容
1	**アドレス空間の指定**	Address_space=<**アドレス空間サイズ**> <**アドレス空間サイズ**>:{ 20 | 24 | 28 | 32 }	**アドレス空間の指定**
2	fpu	Fpu	FPU **あり***1
3	dsp	Dsp	DSP **あり***1

*1 SuperH **用のオプションです。**H8S,H8/300 **シリーズでは無効です。**

アドレス空間の指定

Address_space

—

書　式　　Address_space=<アドレス空間サイズ>
<アドレス空間サイズ>:{ 20 | 24 | 28 | 32 }

説　明　　cpu=300ha, 2000a, 2600a の場合のアドレス空間サイズを指定します。
本オプションの省略時解釈は、24 です。

例　　helfcnv -a=20 *.obj ; ディレクトリ内の全ての*.obj をアドレス空間サイズが
; 20bit の elf 形式に変換します。

備　考　　リンケージエディタ出力リロケータブルファイル（拡張子 rel)およびリロケータブルファイルを含むライブラリファイルは変換できません。

(2) ロードモジュールファイルの変換

ELF 形式ロードモジュールを、リンケージエディタの旧バージョン（Ver.6 台とそれ以前）の出力

オブジェクトファイル形式に変換します。ロードモジュールファイルにデバッグ情報が含まれている場合は、デバッグ情報も含めて変換します。H8SX のロードモジュールは変換しないでください。

表 17.5 ロードモジュールファイル変換用オプション一覧

	項目	オプション	ダイアログメニュー	指定内容
1	**変換形式指定**	Sysrof	**コンパイラ** <**アウトプット**> [**出力形式**:] [**アブソリュート** (SYSROF)]	SYSROF **形式に変換**
		Dwarf1		ELF/DWARF1 **形式に変換**

変換形式指定

Sysrof
Dwarf1

コンパイラ<アウトプット>[出力形式:][アブソリュート(SYSROF)]

書　式　　Sysrof

Dwarf1

説　明　　変換後のオブジェクト形式を指定します。

sysrof 指定時は、ELF/DWARF2 形式のロードモジュールファイルを、SYSROF 形式に変換します。

dwarf1 指定時は、ELF/DWARF2 形式のロードモジュールファイルを 、ELF/DWARF1 形式に変換します。

最適化リンケージエディタで、sdebug オプションを指定したときは、変換後のファイルにはデバッグ情報は含まれません。

例

```
helfcnv  test.abs                ; test.abs を SYSROF 形式に変換
helfcnv  -d test.abs             ; test.abs を ELF/DWARF1 形式に変換
```

備　考　　#pragma option で、同一ファイル内に最適化有り、無しの関数を混在した場合、変換後のファイルにはデバッグ情報は含まれません。

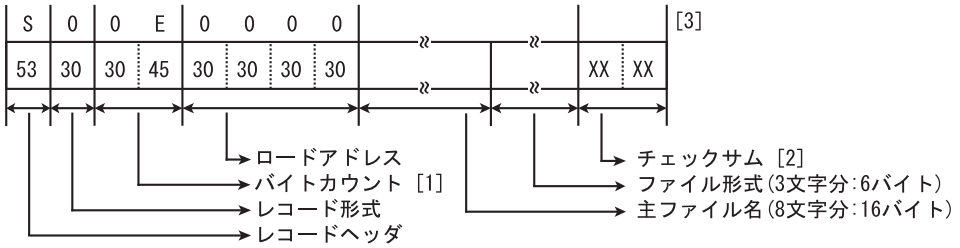
18. 付録

18.1 S タイプ、HEX ファイル形式

本節では、最適化リンケージエディタによって出力される S タイプファイルおよび、HEX ファイルについて説明します。

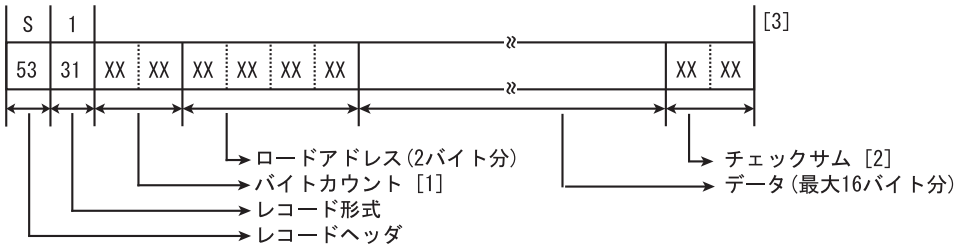
18.1.1 S タイプファイル形式

(a) ヘッダレコード(S0レコード)

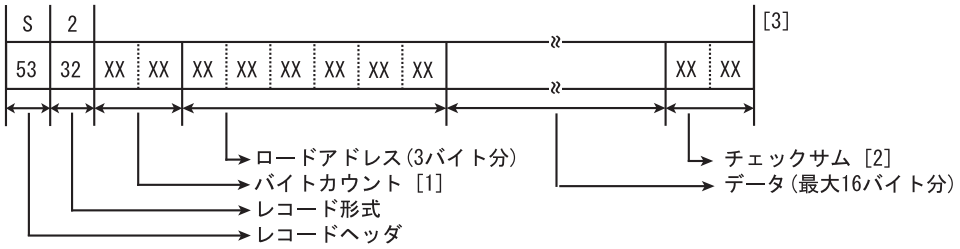


(b) データレコード(S1, S2, S3レコード)

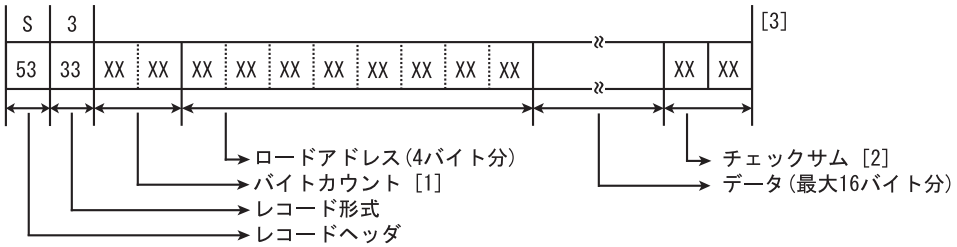
(i) ロードアドレスが0 ～ FFFFの場合



(ii) ロードアドレスが10000 ～ FFFFFFの場合

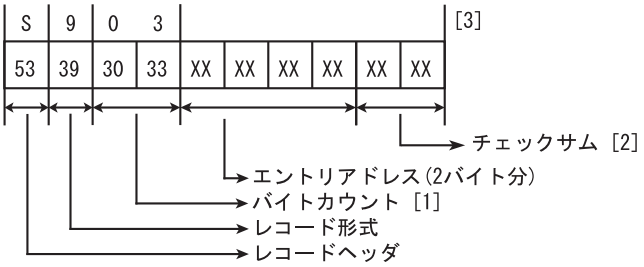


（iii）ロードアドレスが1000000 ～ FFFFFFFFの場合

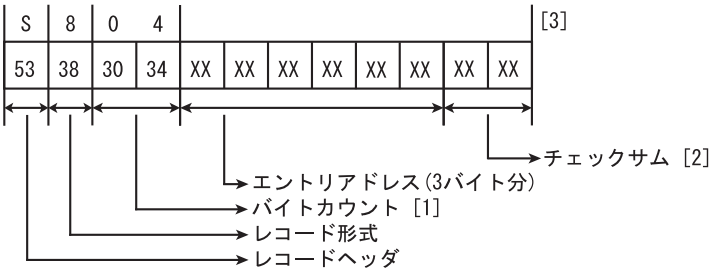


(c) エンドレコード(S9, S8, S7レコード)

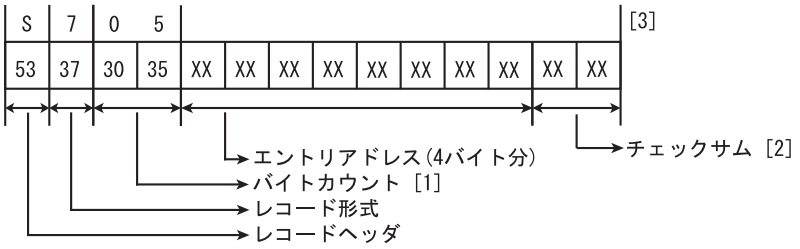
（i） エントリアドレスが0 ～ FFFFの場合



（ii）エントリアドレスが10000 ～ FFFFFFの場合



（iii）エントリアドレスが1000000 ～ FFFFFFFFの場合



【注】 [1] ロードアドレス(またはエントリアドレス)からチェックサムまでのバイト数
[2] バイトカウンタからチェックサムの前までのデータ値をバイト単位に加算した結果の1の補数
[3] チェックサムの直後に改行コードが付加される

18.1.2 HEX ファイル形式

各データレコードの実行アドレスは以下のように求めます。

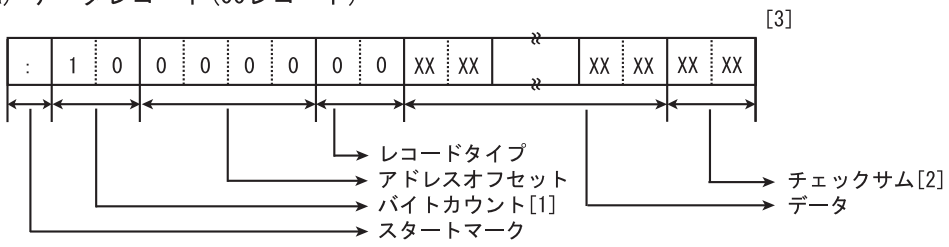
(1) **セグメントアドレスの場合**

(セグメントベースアドレス << 4) + (データレコードのアドレスオフセット)

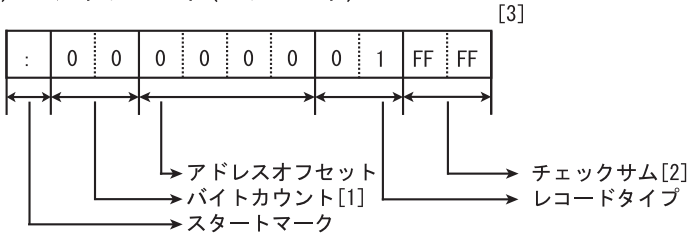
(2) **リニアアドレスの場合**

(リニアベースアドレス << 16) + (データレコードのアドレスオフセット)

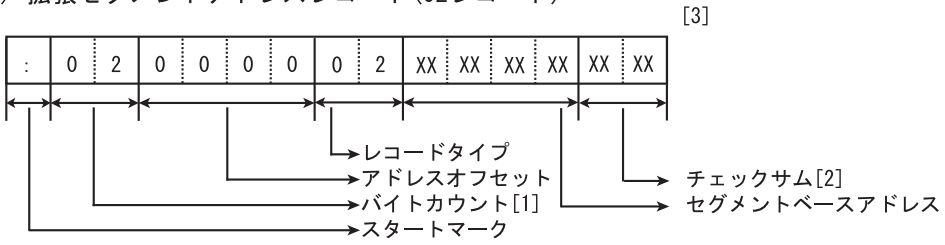
(a) データレコード(00レコード)



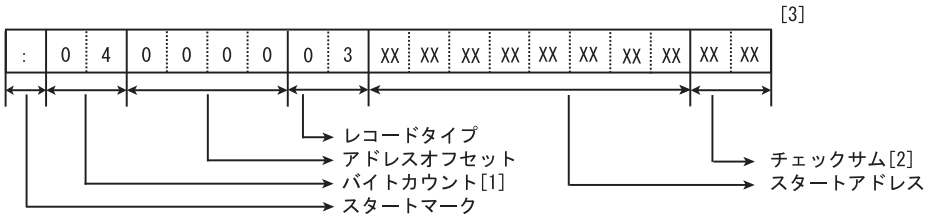
(b) エンドレコード(01レコード)



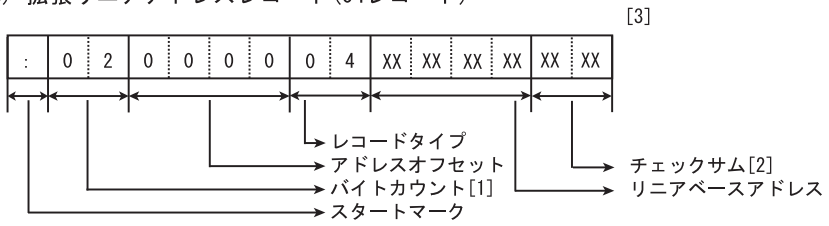
(c) 拡張セグメントアドレスレコード(02レコード)



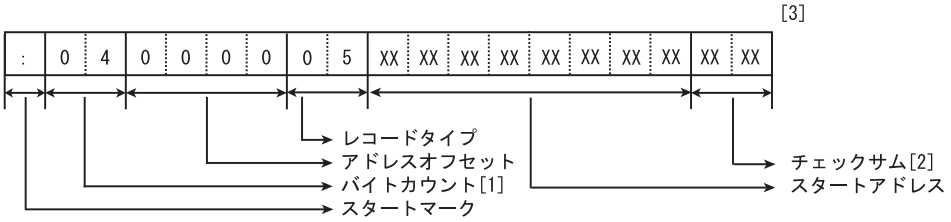
(d) スタートアドレスレコード(03レコード)



(e) 拡張リニアアドレスレコード(04レコード)



(f) 32bitスタートリニアアドレスレコード(05レコード)



【注】 [1] レコードタイプからチェックサムの前までのバイト数
[2] バイトカウンタからチェックサムの前までのデータを16進数で加算した結果の2の補数(下位8bitが有効)
[3] チェックサムの直後に改行コードが付加される

18.2 ASCII コード一覧表

表 18.1 ASCII コード一覧表

下位 4 ビット	上位 4 ビット							
	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	‘	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	＼	l	|
D	CR	GS	−	=	M	]	m	}
E	SO	RS	・	>	N	^	n	～
F	SI	US	／	?	O	_	o	DEL

18.3 短絶対アドレスのアクセス範囲

各 CPU／動作モードにおける 8 ビット絶対アドレスおよび 16 ビット絶対アドレスのアクセス範囲を**表** 18.2 に示します。

表 18.2 **短絶対アドレスのアクセス範囲**

CPU／**動作モード**	8 **ビット絶対アドレス（**@aa:8**）のアクセス範囲**	16 **ビット絶対アドレス（**@aa:16**）のアクセス範囲**
H8SXA:32 H8SXX [:32] 2600a:32 2000a:32	0xFFFFFF00～0xFFFFFFFF	0x0～0x7FFF、 0xFFFF8000～0xFFFFFFFF
H8SXA:28 H8SXX:28 2600a:28 2000a:28	0xFFFFF00～0xFFFFFFF	0x0～0x7FFF、 0xFFF8000～0xFFFFFFF
H8SXA[:24] H8SXM[:24] 2600a[:24] 2000a[:24] 300ha[:24]	0xFFFF00～0xFFFFFF	0x0～0x7FFF、 0xFF8000～0xFFFFFF
H8SXA:20 H8SXM:20 2600a:20 2000a:20 300ha:20	0xFFF00～0xFFFFF	0x0～0x7FFF、 0xF8000～0xFFFFF
H8SXN 2600n 2000n 300hn 300 300l	0xFF00～0xFFFF	—

[注] CPU種別がH8SXの場合、8ビット絶対アドレス領域はsbrオプション指定により変更が可能です。

索引

記号

数字

英語

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

日本語

H8S、H8/300シリーズ C/C++コンパイラ、アセンブラ、最適化
リンケージエディタ　ユーザーズマニュアル

発行年月　2003 年 9 月 12 日　Rev.1.00
発　　行　株式会社ルネサス テクノロジ 営業企画統括部
〒100-0004 東京都千代田区大手町 2-6-2
編　　集　株式会社ルネサス小平セミコン 技術ドキュメント部

株式会社ルネサス テクノロジ 営業企画統括部 〒100-0004 東京都千代田区大手町2-6-2 日本ビル

営業お問合せ窓口
株式会社ルネサス販売

http://www.renesas.com

本社	〒100-0004	千代田区大手町2-6-2 (日本ビル)	(03) 5201-5350
京浜支社	〒212-0058	川崎市幸区鹿島田890-12 (新川崎三井ビル)	(044) 549-1662
西東京支社	〒190-0023	立川市柴崎町2-2-23 (第二高島ビル2F)	(042) 524-8701
札幌支店	〒060-0002	札幌市中央区北二条西4-1 (札幌三井ビル5F)	(011) 210-8717
東北支社	〒980-0013	仙台市青葉区花京院1-1-20 (花京院スクエア13F)	(022) 221-1351
いわき支店	〒970-8026	いわき市平小太郎町4-9 (損保ジャパンいわき第二ビル3F)	(0246) 22-3222
茨城支社	〒312-0034	ひたちなか市堀口832-2 (日立システムプラザ勝田1F)	(029) 271-9411
新潟支店	〒950-0087	新潟市東大通1-4-2 (新潟三井物産ビル3F)	(025) 241-4361
松本支社	〒390-0815	松本市深志1-2-11 (昭和ビル7F)	(0263) 33-6622
中部営業本部	〒460-0008	名古屋市中区栄3-13-20 (栄センタービル4F)	(052) 261-3000
浜松支店	〒430-7710	浜松市板屋町111-2 (浜松アクトタワー10F)	(053) 451-2131
西部営業本部	〒541-0044	大阪市中央区伏見町4-1-1 (大阪明治生命館ランドアクシスタワー10F)	(06) 6233-9500
北陸支社	〒920-0031	金沢市広岡3-1-1 (金沢パークビル8F)	(076) 233-5980
中国支社	〒730-0036	広島市中区袋町5-25 (広島袋町ビルディング8F)	(082) 244-2570
松山支店	〒790-0003	松山市三番町4-4-6 (GEエジソンビル松山2号館3F)	(089) 933-9595
鳥取支店	〒680-0822	鳥取市今町2-251 (日本生命鳥取駅前ビル)	(0857) 21-1915
九州支社	〒812-0011	福岡市博多区博多駅前2-17-1 (ヒロカネビル本館5F)	(092) 481-7695
鹿児島支店	〒890-0053	鹿児島市中央町12-2 (明治生命西鹿児島ビル2F)	(099) 284-1748

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：カスタマサポートセンタ E-Mail: csc@renesas.com

Colophon 1.0

H8S、H8/300 シリーズ C/C++ コンパイラ、アセンブラ、最適化リンケージエディタ
HSS008CLCS6S ユーザーズマニュアル

ルネサス エレクトロニクス株式会社
神奈川県川崎市中原区下沼部1753 〒211-8668