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

RENESAS

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。

お客様各位

資料中の「日立製作所」、「日立XX」等名称の株式会社ルネサス テクノロジへの変更について

2003年4月1日を以って三菱電機株式会社及び株式会社日立製作所のマイコン、ロジック、アナログ、ディスクリート半導体、及びDRAMを除くメモリ（フラッシュメモリ・SRAM等）を含む半導体事業は株式会社ルネサス テクノロジに承継されました。従いまして、本資料中には「日立製作所」、「株式会社日立製作所」、「日立半導体」、「日立XX」といった表記が残っておりますが、これらの表記は全て「株式会社ルネサス テクノロジ」に変更されておりますのでご理解の程お願い致します。尚、会社商標・ロゴ・コーポレートステートメント以外の内容については一切変更しておりませんので資料としての内容更新ではありません。

ルネサステクノロジ ホームページ（http://www.renesas.com）

2003年4月1日
株式会社ルネサス テクノロジ
カスタマサポート部

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。

User's Manual

RENESAS

H8S,H8/300シリーズ C/C++コンパイラ

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

HSS008CLCS3S

ルネサス エレクトロニクス
www.renesas.com

Rev.5.00　1998.12

ご注意

1. 本書に記載の製品及び技術のうち「外国為替及び外国貿易法」に基づき安全保障貿易管理関連貨物・技術に該当するものを輸出する場合，または国外に持ち出す場合は日本国政府の許可が必要です。
2. 本書に記載された情報の使用に際して，弊社もしくは第三者の特許権，著作権，商標権，その他の知的所有権等の権利に対する保証または実施権の許諾を行うものではありません。また本書に記載された情報を使用した事により第三者の知的所有権等の権利に関わる問題が生じた場合，弊社はその責を負いませんので予めご了承ください。
3. 製品及び製品仕様は予告無く変更する場合がありますので，最終的な設計，ご購入，ご使用に際しましては，事前に最新の製品規格または仕様書をお求めになりご確認ください。
4. 弊社は品質・信頼性の向上に努めておりますが，宇宙，航空，原子力，燃焼制御，運輸，交通，各種安全装置， ライフサポート関連の医療機器等のように，特別な品質・信頼性が要求され，その故障や誤動作が直接人命を脅かしたり，人体に危害を及ぼす恐れのある用途にご使用をお考えのお客様は，事前に弊社営業担当迄ご相談をお願い致します。
5. 設計に際しては，特に最大定格，動作電源電圧範囲，放熱特性，実装条件及びその他諸条件につきましては，弊社保証範囲内でご使用いただきますようお願い致します。
   保証値を越えてご使用された場合の故障及び事故につきましては，弊社はその責を負いません。
   また保証値内のご使用であっても半導体製品について通常予測される故障発生率，故障モードをご考慮の上，弊社製品の動作が原因でご使用機器が人身事故，火災事故，その他の拡大損害を生じないようにフェールセーフ等のシステム上の対策を講じて頂きますようお願い致します。
6. 本製品は耐放射線設計をしておりません。
7. 本書の一部または全部を弊社の文書による承認なしに転載または複製することを堅くお断り致します。
8. 本書をはじめ弊社半導体についてのお問い合わせ，ご相談は弊社営業担当迄お願い致します。

はじめに

本マニュアルは、**「H8S,H8/300 シリーズ C/C++コンパイラ」**の使用方法を述べたものです。

本コンパイラは C 言語および C++言語で記述したソースプログラム、日立シングルチップマイクロコンピュータ H8S/2600 シリーズ、H8S/2000 シリーズ、H8/300H 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。

本マニュアルは本文 11 章と付録で構成されています。各章の内容を以下に示します。

第 1 章　概要

機能概要と開発の流れを説明します。

第 2 章　チュートリアル

コンパイルからロードモジュール作成までの一連の操作手順について説明します。

第 3 章　ファイル仕様

ソースプログラムやリストファイル等のファイル仕様について説明します。

第 4 章　コンパイラのオプション・環境変数

コンパイラのオプション機能・環境変数について説明します。

第 5 章　プログラミング

コンパイラの限界値、オブジェクトプログラムの実行方式などプログラム開発時に考慮すべき事項について説明します。また、組み込みプログラム向けのコンパイラ拡張機能について説明します。

第 6 章　標準 C ライブラリ

C/C++言語の中で標準的に利用できる C ライブラリ関数の仕様について説明します。

第 7 章　EC++クラスライブラリ

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。

第 8 章　システム組み込み

コンパイラの生成したオブジェクトプログラムをシステムに組み込むための、メモリ割り付け手法や ROM 化の方法について説明します。また、標準入出力ライブラリ、メモリ管理ライブラリを使用する場合に作成しなければならない低水準インタフェースルーチンの仕様について説明します。

第 9 章　モジュール間最適化ツールのオプション・環境変数

モジュール間最適化ツールのオプション機能・環境変数について説明します。

第 10 章　コンパイラのエラーメッセージ

コンパイル時に発生するエラーメッセージとエラー内容を説明します。

第 11 章　モジュール間最適化のエラーメッセージ

モジュール間最適化ツール実行時に発生するエラーメッセージとエラー内容を説明します。

■注意

H8S, H8/300 シリーズ C コンパイラ（Ver. 2.0）から使用している場合は、**「付録Ｉ　旧バージョンとの相違点」**を参照してください。

■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®95operating system *[4]、Microsoft®WindowsNT®operating system に対応するように書かれています。UNIX 上で動作するコンパイラを以下 UNIX 版と称します。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コンパイラを以下 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®95 operating system, Microsoft® WindowsNT® operating system **は、米国** Microsoft Corporation **の米国及びその他の国における登録商標です。**

コンパイラバージョンアップにおける注意事項

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。

C++言語仕様の制限事項

C++言語仕様は、「Annotated C++Reference Manual(Addison-Wesley,1990)」Margaret A.Ellis、Bjarne Stroustrup 著に準拠しています。未サポート機能として、以下の項目があります。

(1) テンプレート
(2) 例外処理
(3) bool*

注意 EC++**ライブラリでは、**defbool.h **で以下のように定義しています。**
enum bool{_FALSE=0,_TRUE=1};

目次

第 1 章　概要

1.1　プログラムの開発手順 1
1.2　コンパイラの概要 2
1.3　標準ライブラリの概要 2
1.4　モジュール間最適化ツールの概要 3
1.5　CPU 情報ファイル作成ツールの概要 3

第 2 章　チュートリアル

2.1　チュートリアルの概要 5
2.2　コマンドライン指定による実行 5
2.2.1　環境変数の設定 5
2.2.2　CPU 固有の環境設定 6
2.2.3　コンパイラの実行 7
2.2.4　CPU 情報ファイルの作成 9
2.2.5　モジュール間最適化ツールの実行 10

第 3 章　ファイル仕様

3.1　ファイル名の付け方 13
3.2　コンパイルリストの見方 14
3.2.1　コンパイルリストの構成 14
3.2.2　ソースリスト情報 14
3.2.3　エラー情報 16
3.2.4　シンボル割り付け情報 17
3.2.5　オブジェクト情報 19
3.2.6　統計情報 21
3.3　最適化情報リストの見方 22
3.3.1　最適化情報リストの構成 22
3.3.2　入力情報 22
3.3.3　シンボル最適化情報 23
3.3.4　シンボル参照回数情報 24
3.4　リンケージリストの見方 25
3.4.1　リンケージリストの構成 25
3.4.2　入力情報 25
3.4.3　リンケージマップリスト 26
3.4.4　外部定義シンボルリスト 27
3.4.5　未定義シンボルリスト 27

3.4.6　変更／削除シンボルリスト........28
3.4.7　強制定義シンボルリスト........28

第 4 章　コンパイラのオプション・環境変数

4.1　コマンドラインの形式........29
4.2　オプション一覧........29
4.3　オプションの内容........32
4.4　コンパイラの環境変数........52

第 5 章　プログラミング

5.1　コンパイラの限界値........53
5.2　オブジェクトプログラムの構造........55
5.3　データの内部表現........58
5.3.1　スカラ型（C 言語）、基本型（C++言語）........58
5.3.2　複合型（C 言語）、クラス型（C++言語）........60
5.3.3　ビットフィールド........64
5.4　C++プログラムと C プログラムとの結合........66
5.5　アセンブリプログラムとの結合........66
5.5.1　外部名の相互参照方法........66
5.5.2　関数呼び出しのインタフェース........68
5.6　拡張機能........75
5.6.1　#pragma 拡張子........75
5.6.2　セクションアドレス演算子........90
5.6.3　組み込み関数........91
5.7　プログラム作成上の注意事項........105
5.7.1　コーディング上の注意事項........105
5.7.2　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........108
5.7.3　プログラム開発上の注意事項........109
5.7.4　トラブル発生時の対処方法........110

第６章　標準Ｃライブラリ

6.1　ライブラリの概要 ... 111
6.2　<stddef.h> ... 118
6.3　<assert.h> ... 119
6.4　<ctype.h> ... 120
6.5　<float.h> ... 129
6.6　<limits.h> ... 131
6.7　<errno.h> ... 132
6.8　<math.h> ... 133
6.9　<mathf.h> ... 149
6.10　<setjmp.h> ... 165
6.11　<stdarg.h> ... 168
6.12　<stdio.h> ... 172
6.13　<no_float.h> ... 217
6.14　<stdlib.h> ... 218
6.15　<string.h> ... 233

第７章　EC++クラスライブラリ

7.1　ライブラリの概要 ... 255
7.2　ストリーム入出力用クラスライブラリ ... 256
7.2.1　ios_base::Init クラス ... 257
7.2.2　ios_base クラス ... 258
7.2.3　ios クラス ... 263
7.2.4　ios クラスマニピュレータ ... 267
7.2.5　streambuf クラス ... 272
7.2.6　istream::sentry クラス ... 282
7.2.7　istream クラス ... 283
7.2.8　istream クラスマニピュレータ ... 292
7.2.9　istream メンバ外関数 ... 293
7.2.10　ostream::sentry クラス ... 294
7.2.11　ostream クラス ... 295
7.2.12　ostream クラスマニピュレータ ... 299
7.2.13　ostream メンバ外関数 ... 300
7.2.14　smanip クラスマニピュレータ ... 301
7.2.15　EC++入出力ライブラリの使用例 ... 303
7.3　メモリ管理用ライブラリ ... 305
7.4　複素数計算用クラスライブラリ ... 307
7.4.1　float_complex クラス ... 307
7.4.2　float_complex メンバ外関数 ... 311
7.4.3　double_complex クラス ... 319
7.4.4　double_complex メンバ外関数 ... 323

7.5 文字列操作用クラスライブラリ 331
7.5.1 string クラス 331
7.5.2 string クラスマニピュレータ 351

第 8 章　システム組み込み

8.1 システム組み込みの概要 357
8.2 メモリ領域の割り付け 358
8.2.1 静的領域の割り付け 358
8.2.2 動的領域の割り付け 361
8.3 実行環境の設定 364
8.4 ライブラリ関数使用時の実行環境の設定 367

第 9 章　モジュール間最適化のオプション・環境変数

9.1 コマンドラインの形式 383
9.2 サブコマンドファイルの書式 383
9.3 最適化機能オプション／サブコマンド 384
9.4 リンケージ機能サブコマンド 391
9.5 モジュール間最適化の環境変数 404

第 10 章　コンパイラのエラーメッセージ

10.1 コンパイラのエラーメッセージ 405
10.2 C ライブラリ関数のエラーメッセージ 428

第 11 章　モジュール間最適化のエラーメッセージ

11.1 モジュール間最適化のエラーメッセージ 431

付録

A. コンパイラが規定する言語仕様 439
A.1 言語仕様 439
A.2 サポートしていないライブラリ 443
A.3 浮動小数点の仕様 444
B. 引数割り付けの具体例 450
B.1 H8S/2600 用、H8S/2000 用、H8/300H 用（cpu＝2600a、cpu＝2600n、cpu＝2000a、cpu＝2000n、cpu＝300ha、cpu＝300hn） 450
B.2 H8/300 用（cpu＝300） 453
C. レジスタとスタック領域の使用法 456
C.1 H8S/2600 用、H8S/2000 用、H8/300H 用アドバンストモード（cpu＝2600a、cpu＝2000a、cpu＝300ha） 456
C.2 H8S/2600 用、H8S/2000 用、H8/300H 用ノーマルモード（cpu＝2600n、cpu＝2000n、cpu＝300hn） 457

C.3 H8/300 用（cpu＝300） 458
D. 終了処理関数の作成例 459
D.1 終了処理の登録と実行（onexit）ルーチンの作成例 459
D.2 プログラムの終了（exit）ルーチンの作成例 460
D.3 異常終了（abort）ルーチンの作成例 461
E. 低水準インタフェースルーチンの作成例 462
F. 短絶対アドレスのアクセス範囲 469
G. エンコード規則 470
H. リエントラントライブラリ 472
I. 旧バージョンとの相違点 474
I.1 組み込み用拡張機能の追加 474
I.2 追加・改善機能 474
I.3 言語仕様上の変更 475
J. CPU 情報ファイルの作成 477
J.1 CIA の機能 477
J.2 CIA の起動方法 477
J.3 CIA の使用手順と選択メニュー 478
J.4 CIA の使用例 479
J.5 制限事項一覧 481
J.6 CIA のエラーメッセージ 482
K. ASCII コード一覧表 483

索引 485

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、標準ライブラリ、モジュール間最適化ツール、CPU 情報ファイル作成ツールについて説明します。

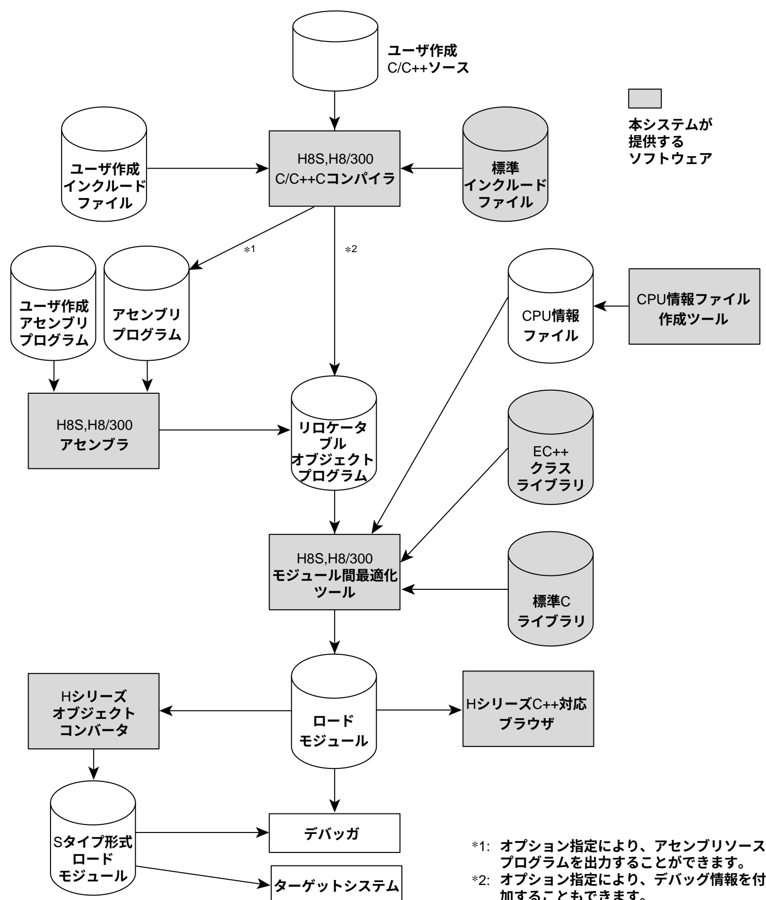


図 1.1 **プログラムの開発手順**

以下、本マニュアルで説明するコンパイラ、標準ライブラリ、モジュール間最適化ツール、CPU情報ファイル作成ツールの概要を述べます。

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。

本コンパイラは、以下の CPU をサポートしています。

- H8S/2600 シリーズ（以下、H8S/2600 と略す）
- H8S/2000 シリーズ（以下、H8S/2000 と略す）
- H8/300H シリーズ（以下、H8/300H と略す）
- H8/300 シリーズ（以下、H8/300 と略す）
- H8/300L シリーズ（以下、H8/300 と略す）*

また、H8S/2600、H8S/2000、H8/300H では、ノーマルモードとアドバンストモードの各 CPU／動作モードもサポートしています。

【注】* **H8/300L シリーズは、H8/300 シリーズと同一の命令仕様なので、コンパイラでは H8/300 シリーズと同一に扱います。**

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、ブラウザによるC++ソース解析を行うためのブラウザ情報出力オプションをサポートしています。

(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) モジュール間最適化ツールによるリンク時最適化を行うために必要な付加情報を出力する、モジュール間最適化情報出力オプションをサポートしています。

1.3 標準ライブラリの概要

本コンパイラが提供する標準ライブラリには、C 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）を含んだ標準 C ライブラリファイルと、組み込み向け C++クラスライブラリを含んだ EC++クラスライブラリファイルがあります。また、コードサイズ優先／スピード優先を選択できます。

標準ライブラリには、次の 40 種類があります。リンク時に、cpu オプション、regparam オプションおよびコードサイズ優先／スピード優先の選択により**表** 1.1 に示すライブラリを指定してください。

表 1.1 標準ライブラリとコンパイルオプションの関係

CPU/動作モード	オプション	regparam 指定	標準 C ライブラリ		EC++クラスライブラリ	
			サイズ優先	スピード優先	サイズ優先	スピード優先
H8/300	300	2	c38reg.lib	c38regs.lib	ec2reg.lib	ec2regs.lib
		3	c38reg3.lib	c38regs3.lib	ec2reg3.lib	ec2regs3.lib
H8/300H ノーマルモード	300hn	2	c38hn.lib	c38hns.lib	ec2hn.lib	ec2hns.lib
		3	c38hn3.lib	c38hns3.lib	ec2hn3.lib	ec2hns3.lib
H8/300H アドバンストモード	300ha	2	c38ha.lib	c38has.lib	ec2ha.lib	ec2has.lib
		3	c38ha3.lib	c38has3.lib	ec2ha3.lib	ec2has3.lib
H8S/2000,H8S/2600 ノーマルモード	2000n 2600n	2	c8s26n.lib	c8s26ns.lib	ec226n.lib	ec226ns.lib
		3	c8s26n3.lib	c8s26ns3.lib	ec226n3.lib	ec226ns3.lib
H8S/2000,H8S/2600 アドバンストモード	2000a 2600a	2	c8s26a.lib	c8s26as.lib	ec226a.lib	ec226as.lib
		3	c8s26a3.lib	c8s26as3.lib	ec226a3.lib	ec226as3.lib

注意　ライブラリのコードサイズ優先、スピード優先の選択は、コンパイラオプションの speed とは関係なく指定することができます

1.4　モジュール間最適化ツールの概要

「H8S,H8/300 シリーズモジュール間最適化ツール」（以下、モジュール間最適化ツールと略す）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、最適化を行い、結合してロードモジュールを作成するソフトウェアシステムです。従来コンパイラでは最適化できなかった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。

本モジュール間最適化ツールは、H シリーズリンケージエディタ（以下、リンケージエディタと略す）と同じリンケージ機能をサポートしており、さらに次の３つの機能が追加されています。

(1)　オブジェクトプログラムの最適化機能
(2)　DWARF情報（デバッグ情報フォーマット）の出力機能
(3)　シンボル参照情報リストの出力機能

1.5　CPU 情報ファイル作成ツールの概要

「H8S,H8/300 シリーズ CPU 情報ファイル作成ツール」（以下、CPU 情報ファイル作成ツールと略す）は、使用可能なメモリの配置情報を含む CPU 情報ファイルを作成するソフトウェアシステムです。モジュール間最適化ツール実行時に、CPU 情報ファイルを指定することにより、短絶対アドレス領域などのメモリ特性を活用した最適化やリンク結果によるメモリ割り付け範囲チェックの機能が有効になります。

2. チュートリアル

2.1 チュートリアルの概要

本章では、コンパイラやモジュール間最適化ツールの起動方法、オプションの指定方法およびコンパイルからロードモジュール作成にいたる一連のプログラム作成作業について説明します。

なお、説明は H8S,H8/300 シリーズ C/C++コンパイラパッケージに含まれる全てのツールをインストールした状態で進めます。

PC 版の場合は、日立統合開発環境によるウィンドウインタフェースと、DOS プロンプトによるコマンドラインインタフェースの２通りの実行方法があります。

ウィンドウインタフェースをご使用の場合は、**日立統合開発環境のユーザーズマニュアルにあるチュートリアル**を参照してください。

2.2 コマンドライン指定による実行

2.2.1 環境変数の設定

(1) 実行ファイルのパス設定(path)

コンパイラ、モジュール間最適化ツール、CPU 情報ファイル作成ツールのパスを設定します。

■指定フォーマット

UNIX 版　%set path=(<コンパイラパス名> $path)

PC 版　C:¥> path=<コンパイラパス名>; <最適化ツールパス名>; <CPU 情報ファイル作成ツールパス名>; [<既存パス名>;...]

■指定例

UNIX 版　%set path=(/usr/CH38 $path)

PC 版　C:¥> path=c:¥him¥toolchains¥hitachi¥ch38¥v30¥bin;c:¥him¥toolchains¥hitachi¥optlnk38¥v20;c:¥him¥toolchains¥hitachi¥utilities;%path%

(2) インクルードファイルの格納ディレクトリの指定(CH38)

コンパイラが提供するシステムインクルード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

UNIX 版　%setenv CH38 <インクルードファイルパス名>

PC 版　C:¥> set CH38=<インクルードファイルパス名>

■指定例

UNIX 版　%setenv CH38 /usr/CH38

PC 版　C:¥> set CH38=c:¥him¥toolchains¥hitachi¥ch38¥v30¥include

(3) テンポラリファイルの格納ディレクトリの指定(CH38TMP)

コンパイラが使用するテンポラリファイル格納ディレクトリを指定します。

■指定フォーマット

```
UNIX版   %setenv CH38TMP <テンポラリファイルパス名>
PC版     C:¥> set CH38TMP=<テンポラリファイルパス名>
```

■指定例

```
UNIX版   %setenv CH38TMP /usr/tmp
PC版     C:¥> set CH38TMP=c:¥
```

2.2.2 CPU 固有の環境設定

(1) CPU 種別の指定(H38CPU)

使用する CPU 種別（CPU 名、動作モード）、アドレス空間のビット幅を指定します。
指定可能な CPU 種別、アドレス空間のビット幅については、**「4.4 コンパイラの環境変数」**を参照してください。

■指定フォーマット

```
UNIX版   %setenv H38CPU <CPU種別>[:<アドレス空間のビット幅>]
PC版     C:¥> set H38CPU=<CPU種別>[:<アドレス空間のビット幅>]
```

■指定例

```
UNIX版   %setenv H38CPU 2600a:24
PC版     C:¥> set H38CPU=2600a:24
```

(2) 標準ライブラリの指定(HLNK_LIBRARY1、HLNK_LIBRARY2)

コンパイラが提供する標準ライブラリファイル（C ライブラリ関数・実行時ルーチンのライブラリファイルおよび C++組み込み用クラスライブラリのライブラリファイル）を指定します。
指定可能なライブラリ名については、**「1.3 標準ライブラリの概要」**を参照してください。

■指定フォーマット

```
UNIX版  %setenv HLNK_LIBRARY1 <標準ライブラリ１>
        %setenv HLNK_LIBRARY2 <標準ライブラリ２>
PC版    C:¥> set HLNK_LIBRARY1=<標準ライブラリ１>
        C:¥> set HLNK_LIBRARY2=<標準ライブラリ２>
```

■指定例

```
UNIX版  %setenv HLNK_LIBRARY1 $CH38/c8s26a.lib
        %setenv HLNK_LIBRARY2 $CH38/ec226a.lib
PC版    C:¥>set HLNK_LIBRARY1=C:¥him¥toolchains¥hitachi¥ch38¥v30¥lib¥c8s26a.lib
        C:¥>set HLNK_LIBRARY2=C:¥him¥toolchains¥hitachi¥ch38¥v30¥lib¥ec226a.lib
```

2.2.3 コンパイラの実行

(1) コンパイラインストールディレクトリ内のサンプルディレクトリに移動します。

■指定例

```
UNIX版   %cd /usr/CH38/sample
PC版     C:¥> cd c:¥him¥toolchains¥hitachi¥ch38¥v30¥sample
```

(2) コンパイラを起動してみましょう。

```
ch38 (RET)
```

コンパイルは行わず、コマンドライン形式、オプション一覧を出力します。

(3) C プログラムをコンパイルします。

```
ch38△test1.c (RET)
```

ソースプログラム「test1.c」を C プログラムとしてコンパイルします。

(4) C++プログラムをコンパイルします。

```
ch38△test2.cpp (RET)
```

ソースプログラム「test2.cpp」を C++プログラムとしてコンパイルします。

(5) 複数のソースプログラムを一度にコンパイルできます。

```
ch38△test1.c△test2.cpp (RET)
```

ソースプログラム「test1.c」を C プログラムとして、「test2.cpp」を C++プログラムとしてコンパイルします。

(6) オプションを指定します。

```
ch38△-goptimize△-debug△-show=object,allocation△test1.c (RET)
```

オプション goptimize、debug、show の前に、－を付加します。
複数のオプションを指定するときは、スペース（△）で区切ります。
また、複数のサブオプションを指定するときはカンマ（, ）で区切ります。

(7) オプション指定は短縮形を用いることができます。

```
ch38△-g△-deb△-sh=o,a△test1.c (RET)
```

オプション-goptimize△-debug△-show=object,allocation と同じ指定です。

(8) 複数のプログラムをコンパイルするときはオプション位置で有効範囲が異なります。

■例 1：ソースプログラムすべてに有効なオプション指定例

```
ch38△-g△-deb△-sh=o,a△test1.c△test2.cpp (RET)
```

最初のソースプログラムより前に指定したオプションは、全てのソースプログラムに有効です。
test1.c、test2.cpp ともに goptimize、debug、show オプションが有効になります。

■例 2：各ソースプログラムごとに有効なオプション指定例

```
ch38△test1.c△test2.cpp△-deb△-sh=o,a (RET)
```

ソースプログラム直後のオプション指定は、直前のソースプログラムだけに有効です。

debug、show オプションは test2.cpp のみ有効です。ソースプログラムごとのオプション指定は、ソースプログラム全体に対するオプション指定よりも優先します。

(9) **ロードモジュールを作成するために初期設定プログラムをコンパイルします。**

```
ch38△-g△-deb△-sh=o,a△init.c△vectbl.c△scttbl.c (RET)
```

初期設定プログラムは、C プログラム、C++プログラム共通に使用します。

(10) **サンプルプログラムの main 関数をコンパイルします。**

```
ch38△-g△-deb△-sh=o,a△cmain.c(RET)（C プログラムの場合）
ch38△-g△-deb△-sh=o,a△cppmain.cpp(RET)（C++プログラムの場合）
```

C プログラムの場合は「cmain.c」、C++プログラムの場合は「cppmain.cpp」をコンパイルします。

2.2.4　CPU 情報ファイルの作成

(1)　CPU 情報ファイル作成を作成します。

ここでは sample ディレクトリ内の CPU 情報ファイルを使用します。使用する CPU/動作モードに合わせて選択してください。

cpu/動作モード	対応する CPU 情報ファイル(サンプル)
H8/300L	300l.cpu
H8/300	300.cpu
H8/300H　ノーマルモード	300hn.cpu
H8/300H　アドバンストモード	300ha.cpu
H8S/2000 ノーマルモード	2000n.cpu
H8S/2000 アドバンストモード	2000a.cpu
H8S/2600 ノーマルモード	2600n.cpu
H8S/2600 アドバンストモード	2600a.cpu

(2)　CPU 情報ファイル作成ツールを起動してみましょう。

■指定例

```
cia38△2600n.cpu (RET)
```

(3)　ファイル名を変更するかきいてきます。(ret)すると変更しません。

```
*** OLD FILE ***
  NEW CPU FILE NAME?  .(RET)
```

(4)　MAP の情報が表示されます。

```
***** CPU INFORMATION *****
CPU : H8S/2600
;
PROGRAM       AREA        BITSIZE : 16
DATA          AREA        BITSIZE  : 16
   No         Device      Start       End      State   Bus
    1 :       ROM AREA    :000000 - 0027FF       2      16
    2 :       EEPROM *1   :006000 - 007FFF       2      16
    3 :       RAM AREA    :00FEC0 - 00FFBF       2      16
    4 :       I/O AREA    :00FFF8 - 00FFFF       3       8
```

【注】モジュール間最適化ツールでは、メモリの属性(ROM,RAM)を参照します。
ROM、RAM または I/O のいずれかを指定してください。

(5)　cia プログラムを終了します。

```
        ** EDIT MENU **
        1:ADD    2:DELETE    3:COMMENT   4:CIA ABORT   .:CIA END
        ?  .(RET)
*** CIA COMPLETED ***
```

2.2.5 モジュール間最適化ツールの実行

(1) モジュール間最適化ツールを起動してみましょう。

```
optlnk38 (RET)
```

最適化、リンク処理は行わずに、コマンドライン形式、オプションの一覧を表示します。

(2) サブコマンドファイルを編集します。

ここでは sample ディレクトリ内のサブコマンドファイルを使用します。使用する CPU/動作モードに合わせて選択してください。

cpu/動作モード	対応するサブコマンドファイル(サンプル)	
	C プログラム	C++プログラム
H8/300L	c300l.sub	cpp300l.sub
H8/300	c300.sub	cpp300.sub
H8/300H ノーマルモード	c300hn.sub	cpp300hn.sub
H8/300H アドバンストモード	c300ha.sub	cpp300ha.sub
H8S/2000 ノーマルモード	c2000n.sub	cpp2000n.sub
H8S/2000 アドバンストモード	c2000a.sub	cpp2000a.sub
H8S/2600 ノーマルモード	c2600n.sub	cpp2600n.sub
H8S/2600 アドバンストモード	c2600a.sub	cpp2600a.sub

■c2600a.sub の内容

```
input          cmain,init             ; 入力ファイル名を指定します
input          vectbl,scttbl          ; 入力ファイル名を指定します
form           a                      ; 出力フォーマットを absolute に指定します
output         sample                 ; 出力ファイル名を sample.abs に指定します
optimize                              ; 全最適化を行います
sysrof                                ; オブジェクトフォーマットを sysrof に指定します
debug                                 ; デバッグ情報ファイルを出力します
start          Cvect1(0),Cvect2       ; 各セクションの開始アドレスを指定します
start          P,C,D,C$BSEC,C$DSEC,$ABS8D,$ABS16D(0200)
start          R,B(0ED00),S(0FE00)    ;
start          $ABS8B,$ABS8R(0FFFFFF00),$ABS16B,$ABS16R(0FFFF8000)
ROM            (D,R)
ROM            ($ABS8D,$ABS8R)
ROM            ($ABS16D,$ABS16R)
check_section                         ; start 指定のないセクションをチェックします
cpu            2600a.cpu              ; cpu 情報ファイル名を指定します
cpucheck                              ; メモリ割り付けが CPU 情報外のときロードモジュールを作成
                                      ; しません
udfcheck                              ; 未定義シンボルがあるときロードモジュールを作成しません
print          sample.map             ; sample.map にメモリ割り付け情報を出力します
mlist          sample.lop             ; sample.lop に最適化情報を出力します
symbol_forbid _vec_table1,_vec_table2
exit                                  ; 処理を終了します
```

■cpp2600a.sub の内容

```
input         cppmain,init          ; 入力ファイル名を指定します
input         vectbl,scttbl         ; 入力ファイル名を指定します
form          a                     ; 出力フォーマットを absolute に指定します
output        sample                ; 出力ファイル名を sample.abs に指定します
optimize                            ; 最適化を行います
sysrofplus                          ; オブジェクトフォーマットを sysrofplus に指定します
sdebug                              ; デバッグ情報ファイルを出力します
start         Cvect1(0),Cvect2      ; 各セクションの開始アドレスを指定します
start         P,C,D,C$INIT,C$END,C$BSEC,C$DSEC,$ABS8D,$ABS16D(0200)
start         R,B(0ED00),S(0FE00)   ;
start         $ABS8B,$ABS8R(0FFFFFF00),$ABS16B,$ABS16R(0FFFF8000)
ROM           (D,R)
ROM           ($ABS8D,$ABS8R)
ROM           ($ABS16D,$ABS16R)
check_section                       ; start 指定のないセクションをチェックします
cpu           2600a.cpu             ; cpu 情報ファイル名を指定します
cpucheck                            ; メモリ割り付けが CPU 情報外のときロードモジュールを作成
                                    ; しません
udfcheck                            ; 未定義シンボルがあるときロードモジュールを作成しません
print         sample.map            ; sample.map にメモリ割り付け情報を出力します
mlist         sample.lop            ; sample.lop に最適化情報を出力します
symbol_forbid _vec_table1,_vec_table2
exit                                ; 処理を終了します
```

(3) モジュール間最適化ツールを実行します。

```
optlnk38 -subcommand=c2600a.sub (RET)
```

または

```
optlnk38 -subcommand=cpp2600a.sub (RET)
```

ロードモジュールファイル「sample.abs」を出力します。また、リンケージマップリスト「sample.map」にメモリ割り付け情報を、最適化情報リスト「sample.lop」にシンボル最適化情報を出力します。

3. ファイル仕様

3.1 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、コンパイラは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。コンパイラおよび関連ソフトウェアで使用する標準のファイル拡張子を**表** 3.1 に示します。

表 3.1 **コンパイラで使用する標準のファイル拡張子**

No.	拡張子	意味
1	c	C 言語で記述されたソースプログラムファイル
2	cpp,cc,cp	C++言語で記述された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後のファイル
7	pp	C++プログラム用プリプロセッサ展開後のファイル
8	obj	リロケータブルオブジェクトプログラムファイル
9	src	アセンブリソースプログラムファイル
10	dtb	ブラウザ情報ファイル
11	iop	モジュール間最適化用情報ファイル
12	dwi	dwarf フォーマット変換用情報ファイル
13	lib	ライブラリファイル
14	lct	ライブラリファイル用 dwarf フォーマット付加情報ファイル
15	abs	アブソリュートロードモジュールファイル
16	ctr	アブソリュートロードモジュール用 dwarf フォーマット付加情報ファイル
17	rel	リロケータブルロードモジュールファイル
18	rct	リロケータブルロードモジュール用 dwarf フォーマット付加情報ファイル
19	lop	モジュール間最適化情報リストファイル
20	map	リンケージマップリストファイル
21	dbg	sysrof フォーマットデバッグ情報ファイル
22	dwf	dwarf フォーマットデバッグ情報ファイル

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

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。

3.2 コンパイルリストの見方

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。

3.2.1 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 3.2** に示します。

表 3.2 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。

3.2.2 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まえのソース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あと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 3.1（a）、（b）**に示します。また、**図 3.1（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 ************

FILE NAME: m0260.c

   Seq File      Line Pi 0----+----1----+----2----+----3----+----4----+----5--
     1 m0260.c      1     #include "header.h"
     4 m0260.c      2
     7 m0260.c      5     int sum2(void)
     8 m0260.c      6     {   int j;
    10 m0260.c      8
    11 m0260.c      9     #ifdef SMALL
    12 m0260.c     10         j=SML_INT;
    13 m0260.c     11     #else
    14 m0260.c     12         j=LRG_INT;
    15 m0260.c     13     #endif
    16 m0260.c     14
    17 m0260.c     15         return j; /* continue12345678901234567890123456 7
    [1]  [2]       [3]    +2345678901234567890  */
    18 m0260.c     16    [6]
                              }
```

(b) show=expansionのソースリスト情報

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

FILE NAME: m0260.c

  Seq File         Line Pi 0----+----1----+----2----+----3----+----4----+----5--
    1 m0260.c         1     #include "header.h"
    2 header.h        1     #define SML_INT        1
    3 header.h        2     #define LRG_INT      100
    4 m0260.c         2
    7 m0260.c         5     int sum2(void)
    8 m0260.c         6     {   int j;
   10 m0260.c         8
   11 m0260.c         9     #ifdef SMALL
   12 m0260.c        10 X       j=SML_INT;
   13 m0260.c        11[4]  #else
   14 m0260.c        12  E      j=100;
   15 m0260.c        13 [5] #endif
   16 m0260.c        14
   17 m0260.c        15         return j; /* continue1234567890123456789012345 67
   [1]  [2]         [3]    +2345678901234567890  */
   18 m0260.c        16    [6]
                              }
```

【注】

[1] リスト上の行番号
[2] 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
[3] 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
[4] 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
[5] show＝expansion指定時、＃define文によるマクロ置換のあったソース行
[6] ソースプログラムの一行がコンパイルリストの一行に入りきらず、複数行にまたがって表示されたソース行

図 3.1 ソースリスト情報の出力形式

3.2.3 エラー情報

エラー情報の出力例を**図** 3.2 に示します。

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

FILE NAME: m0260.c

  Seq File        Line Pi 0----+----1----+----2----+----3----+----4----+----5---
    1 m0260.c        1     #include "header.h"
    4 m0260.c        2
    5 m0260.c        3     extern int sum3(int);
    6 m0260.c        4
    7 m0260.c        5     sum3(int x)
    8 m0260.c        6     {
    9 m0260.c        7         int i;
   10 m0260.c        8         int j;
   11 m0260.c        9
   12 m0260.c       10         j=0;
   13 m0260.c       11         for (i=0; i<=x; i++){
   14 m0260.c       12             j+=k;          <------------ エラー発生行
   15 m0260.c       13         }
   16 m0260.c       14         return j;
   17 m0260.c       15      }
   18 m0260.c       16
   19 m0260.c       17
   20 m0260.c       18


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

FILE NAME: m0260.c


File         Line Erno Lvl  Message
m0260.c        12 2225 (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 3.2 **エラーを含んだソースエラーリストとエラー情報**

3.2.4 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 3.3** に示します。

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

FILE NAME: m0280.c

Seq File       Line Pi 0----+----1----+----2----+----3----+----4----+----5---
  1 m0280.c       1    extern int h(char, char *, double );
  2 m0280.c       2
  3 m0280.c       3    int
  4 m0280.c       4    h(char a, register char *b, double c)
  5 m0280.c       5    {
  6 m0280.c       6       char    *d;
  7 m0280.c       7
  8 m0280.c       8       d= &a;
  9 m0280.c       9       h(*d,b,c);
 10 m0280.c      10       {
 11 m0280.c      11            register int i;
 12 m0280.c      12
 13 m0280.c      13            i= *d;
 14 m0280.c      14            return i;
 15 m0280.c      15       }
 16 m0280.c      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)
Temporary Size          : 0x00000000 Byte(s)   [4]
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**

A saved from B	**Bで渡された引数を関数入口でA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アドレス領域のサイズ**
Linkage Area Size	**：リンケージ領域（リターンアドレス領域＋フレームポインタ退避領域）の合計サイズ**
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 3.3　シンボル割り付け情報（cpu=2600a）

■注意

最適化オプション optimize＝1 が指定されていると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出力します。

Optimize Option Specified : No Allocation Information Available

図 3.3 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** 3.4 に示します。

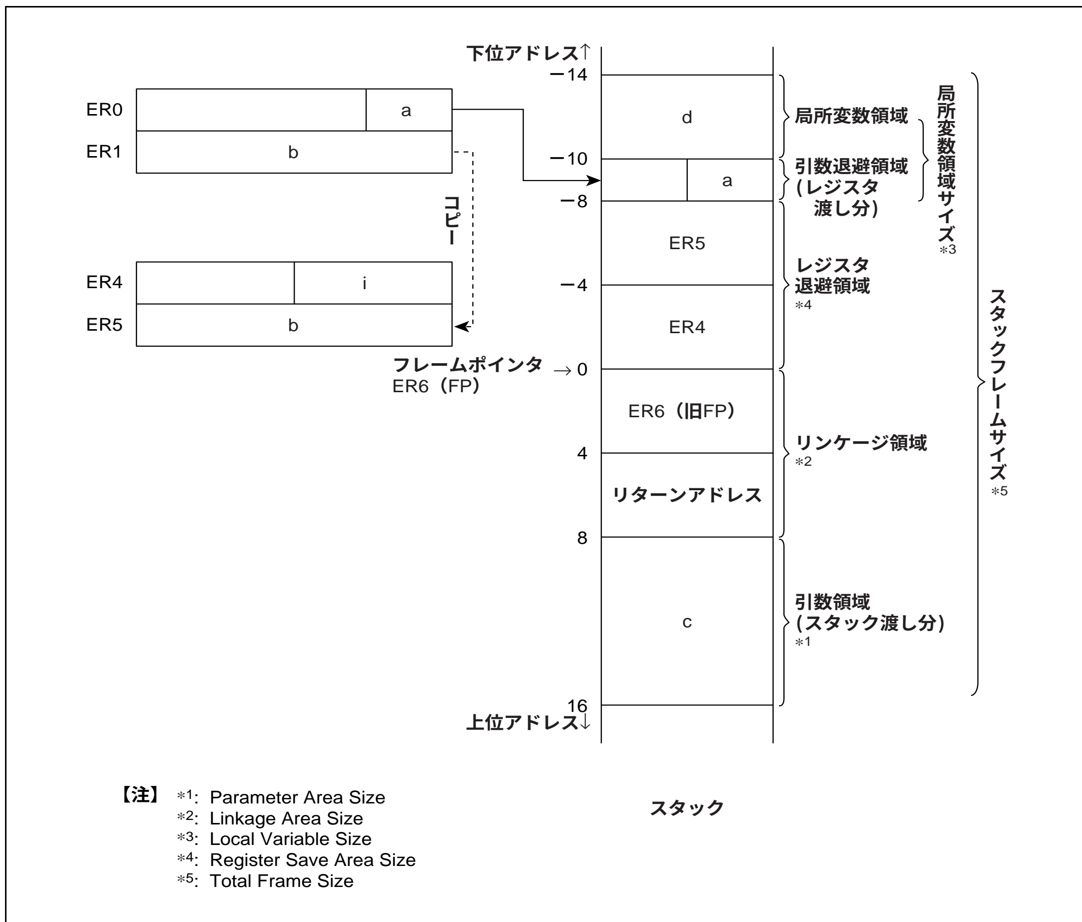


図 3.4　**スタック上の割り付け例**（cpu=2600a）

3.2.5 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** 3.5、**図** 3.6 に示します。

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

FILE NAME: m0251.c

SCT OFFSET CODE        C LABEL     INSTRUCTION OPERAND    COMMENT
[1]  [2]  [3]                [4]
P                                                        ; section
                     1:    extern int sum(int);
                                [5]
                     2:
                     3:    int
                     4:    sum(int x)
   00000000             _sum:                           ; function: sum
                     5:    {
                     6:    int i;
                     7:    int j;
                     8:
                     9:    j=0;
   00000000 1911             SUB.W       R1,R1
                    10:    for (i=0; i<=x; i++){
   00000002 1988             SUB.W       E0,E0
   00000004 4004             BRA         L8:8
   00000006             L7:
                    11:      j+=i;
   00000006 0981             ADD.W       E0,R1
   00000008 0B58             INC.W       #1,E0
   0000000A             L8:
   0000000A 1D08             CMP.W       R0,E0
   0000000C 4FF8             BLE         L7:8
                    12:        }
                    13:        return j;
   0000000E 0D10             MOV.W       R1,R0
                    14:    }
   00000010 5470             RTS
```

【注】
- [1] 各セクションのセクション属性（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 3.5 **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）

■注意

show＝expansion オプションを指定した場合は、常に**図** 3.6 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属性（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**

図 3.6 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）

3.2.6 統計情報

統計情報の出力例を**図** 3.7 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)
                                                                       [1]
TOTAL PROGRAM  SECTION: 0x00000012 Byte(s)
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=alloacation -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 3.7 **統計情報**

■注意

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。

3.3 最適化情報リストの見方

本節では、モジュール間最適化ツールが出力する最適化情報リストの内容と形式について説明します。

3.3.1 最適化情報リストの構成

コンパイルリストの構成と内容を**表** 3.3 に示します。

表 3.3 **最適化情報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	サブオプション省略時
1	入力情報	コマンドライン、サブコマンド、エラーメッセージを表示	—	出力する
2	シンボル最適化情報	最適化対象となったシンボルを表示	show＝symbol	出力しない
3	シンボル参照回数情報	シンボルの参照回数を表示	show＝reference	出力しない

3.3.2 入力情報

コマンドライン、サブコマンドファイルで指定した内容を出力します。また、エラーメッセージも出力します。入力情報の出力例を図 3.8 に示します。

```
LINK COMMAND LINE                                   } [1]

optlnk38 -optimize -sub=a.sub

LINK SUBCOMMANDS                                    } [2]

INPUT a.obj
OUTPUT a.abs
FORM A
CPU sd38.cpu
START P,C,D,B(0400),$INDIRECT(0)
ENTRY _main
DEBUG
PRINT a.map
INF
EXIT
** 1210 CANNOT FIND SECTION($INDIRECT)              } [3]
** 1240 NO DEBUG INFORMATION
```

【注】
[1] **コマンドラインで指定した文字列**
[2] **サブコマンドファイルで指定したサブコマンド列**
[3] **エラーメッセージ**

図 3.8 **入力情報**

3.3.3 シンボル最適化情報

最適化によって新規に生成されたシンボル、サイズが変更された関数シンボル、別セクション（短絶対アドレス領域）に移動された変数シンボル、削除されたシンボルの情報を出力します。シンボル最適化情報の出力例を図 3.9 に示します。

```
***  INTER-MODULE OPTIMIZER Symbol Optimize List  ***

* Created Symbols *  ········································ [1]
Symbol Name Optimized Addr Optimized Size  (Inc Size)
_sub3
          H'00000000    H'00000000     (        +2Bytes)

* Changed Symbols *  ········································ [2]
Symbol Name Optimized Addr Original Size Optimized Size  (Inc/Dec Size)
_sub1
          H'00000000    H'00000000 --> H'00000000     (        -2Bytes)
_sub2
          H'00000000    H'00000000 --> H'00000000     (        -2Bytes)


* Moved Symbols *  ·········································· [3]
Symbol Name Original Addr Optimized Addr Symbol Size
_const_1
          H'00000000 -->H'00000000     H'00000000

* Deleted Symbols *  ········································ [4]

Symbol Name Original Addr Original Size  (Dec Size)
_bss_2
          H'00000000    H'00000000     (        -2Bytes)
 _a34567890123456789012345678901234567890123456789012345678901234567890
2345678901234567890123456789012345678901234567890123456789012345678901
3456789012345678901234567890123456789012345678901234567890123456789012
45678END
           H'00000000     H'00000000    (-2147483648Bytes)
```

【注】
[1] **共通コードの最適化等により、新規に作成されたシンボル**
[2] **最適化によってサイズが変更になった関数シンボル**
[3] **変数アクセス最適化により短絶対アドレス領域に移動された変数シンボル**
[4] **未参照シンボル削除により削除されたシンボル**

図 3.9 **シンボル最適化情報**

3.3.4 シンボル参照回数情報

短絶対アドレッシングモード活用最適化(optimize=variable_access)および間接アドレッシングモードの活用最適化(optimize=function_call)の対象となるシンボルの参照回数を、最適化項目ごとに出力します。シンボル参照回数情報の出力例を**図** 3.10 に示します。

```
***  INTER-MODULE OPTIMIZER Symbol Reference List  ***

* ABS8 Reference Variable *  ································· [1]

Symbol Name                                 Addr     Reference Counts

_const_1
                                          H'00000000              1

_bss_1
                                          H'00000000            123*············[2]

 _a34567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901
2345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
345678END
                                          H'00000000            12

* ABS16 Reference variable *  ································ [3]

Symbol Name                                 Addr     Reference Counts

_const_2
                                          H'00000000    1234567890


* INDIRECT Reference Function *  ····························· [4]

_sub1
```

【注】
[1] **8/16ビット絶対アドレス領域に割り付け可能な変数シンボルとその参照回数**
[2] **最適化済シンボルに*を付加**
[3] **16ビット絶対アドレス領域に割り付け可能な変数シンボルとその参照回数**
[4] **関数シンボルの呼び出し回数(間接関数呼び出しによる最適化対象)**

図 3.10 **シンボル参照回数情報**

3.4 リンケージリストの見方

本節では、モジュール間最適化ツール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。

3.4.1 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** 3.4 に示します。

表 3.4　**リンケージリストの構成と内容**

No.	リストの作成	内容
1	入力情報	コマンドライン、サブコマンド、エラーメッセージを表示
2	リンケージマップ情報	セクションの種別、先頭／最終アドレス、サイズを表示
3	外部定義シンボル情報	外部定義シンボル名、値、種別をアルファベット順に表示
4	未定義シンボル情報	未定義シンボル名とその属性、参照ファイル名をアルファベット順に表示
5	変更／削除シンボル情報	rename／delete で指定されたユニット名／シンボル名とその定義ファイル名を入力順に表示
6	強制定義シンボル情報	define により強制定義された外部参照シンボルを表示

3.4.2 入力情報

```
LINK COMMAND LINE

LNK -sub=test1.sub
        (1)
LINK SUBCOMMANDS

input test1                                                    ┐
entry _main                                                    │
debug                                                          │
rom (D,R)                                                      │
start P,c(200),D,B(08000)                                      │
cpu 300ha.cpu                                                  │
Print test.map                                                 ├ (2)
exit                                                           │
** 105 UNDEFINED EXTERNAL SYMBOL(TEST1.$MVN$3)                 │
** 105 UNDEFINED EXTERNAL SYMBOL(TEST1.$sp_regsv$3)            │
** 105 UNDEFINED EXTERNAL SYMBOL(TEST1.$spregld2$3)            │
** 105 UNDEFINED EXTERNAL SYMBOL(TEST1._strcpy)                │
** 105 UNDEFINED EXTERNAL SYMBOL(TEST1._strcmp)                ┘
```

図 3.11 **入力情報**

(1)　コマンドラインで入力した文字列を出力します。

(2)　サブコマンドファイルから入力した文字列を出力します。また、それに対するエラーメッセージおよびインフォメーションメッセージも出力します。

3.4.3 リンケージマップリスト

```
                    ***     LINKAGE EDITOR LINK MAP LIST  ***

SECTION NAME                            START    -     END    LENGTH
              UNIT NAME                           MODULE NAME


ATTRIBUTE   :   DATA  NOSHR ROM
               (2)    (3)   (4)
D                                  H'00008000  -  H'00008007  H'00000008
(1)              TEST1                    (5)      TEST1            (6)
                  (7)                               (8)
* TOTAL ADDRESS *                  H'00008000  -  H'00008007  H'00000008
                                                (9)                 (10)
```

図 3.12 PRINT **で出力するマップリスト例**

【注】 (1) **セクション名を結合順に表示します。**

(2) **内容種別を次のように表示します。**＊[1]

DATA	データまたはコモン
CODE	コード
DUMMY	ダミー
STACK	スタック
RESV	特殊
UNDEF	未定義
*****	未設定

(3) **結合属性を次のように表示します。**＊[1]

SHR	共有結合
NOSHR	単純結合
DUMMY	ダミー結合
UNDEF	結合属性を定義していない
*****	結合属性が設定されていない

(4) ROM **化支援機能に該当するセクションの場合に表示します。**

ROM ROM用セクション（ROMオプション／サブコマンドのセクション1に相当するもの）

RAM RAM用セクション（ROMオプション／サブコマンドのセクション2に相当するもの）

(5) **オブジェクトの先頭アドレスと最終アドレスを** 16 **進数で表示します。**

(6) **オブジェクトのアドレス長を** 16 **進数で表示します。**

(7) **ユニット名を表示します。**

(8) **モジュール名を表示します。**

(9) **当該セクションの先頭アドレスと最終アドレスを表示します。**

(10) **当該セクションの総アドレス長を表示します。**

＊1 **内容種別、結合属性の詳細は、「**H **シリーズリンケージエディタ、ライブラリアン、オブジェクトコンバータユーザーズマニュアル」を参照して下さい。**

3.4.4 外部定義シンボルリスト

外部定義シンボルがあった場合に**図** 3.13 の形式で出力します。

```
                 *** LINKAGE EDITOR EXTERNALLY DEFINED SYMBOLS LIST ***

      SYMBOL NAME                        ADDR                TYPE

      _Array1Glob                     H'0000800E              DAT
      _Array2Glob                     H'00008074              DAT
      _Func1                          H'000005CE              ENT
      _Func2                          H'000005DC              ENT
       (1)                               (2)                  (3)
```

図 3.13 PRINT **で出力する外部定義シンボルリスト例**

【注】 (1) **外部シンボル名をアルファベット順に表示します。**

(2) **外部シンボルの値を** 16 **進数で表示します。**

(3) **シンボル種別を次のように表示します。**

DAT	データ／変数名
EQU	定数値として定義されたシンボル名
ENT	入口名
***	未定義／未設定

3.4.5 未定義シンボルリスト

未定義シンボルがあった場合に**図** 3.14 の形式で出力します。

```
                *** LINKAGE EDITOR UNRESOLVED EXTERNAL REFERENCE LIST ***

      FILE NAME   :test1.OBJ
                       (1)
      MODULE NAME:TEST1 (2)
      UNIT NAME   :TEST1 (3)
                           SYMBOL NAME                        TYPE

                           $sp_regsv$3                         ***
                           $spregld2$3                         ***
                           _strcpy                             ***
                             (4)                               (5)
```

図 3.14 PRINT **で出力する未定義シンボルリスト例**

【注】 (1) **未定義シンボルを含むファイル名を表示します。**

(2) **未定義シンボルを含むモジュール名を表示します。**

(3) **未定義シンボルを含むユニット名を表示します。**

(4) **未定義シンボル名をアルファベット順に表示します。**

(5) **未定義シンボルの属性を次のように表示します。**

DAT	データ／変数名
ENT	入口名
***	未定義／未設定

3.4.6 変更／削除シンボルリスト

RENAME または DELETE サブコマンド指定により、ユニット名またはシンボル名の変更および削除処理を行った場合、PRINT オプション／サブコマンドを指定すると、**図 3.15** に示す形式でリストを出力します。

```
                    *** LINKAGE EDITOR RENAME/DELETE LIST ***

        FILE NAME   :test1.obj
                      (1)
        UNIT NAME   :TEST1
                      (2)
        FROM NAME                   TO NAME                      TYPE REN/DEL

        _Func1                                                    ED DELETE
        _strcpy                     strcpy                        ER RENAME
          (3)                         (4)                        (5)  (6)
```

図 3.15　変更／削除シンボルリスト例

【注】 (1) **変更／削除の対象となるユニット名またはシンボル名を含むファイル名を入力順に表示します。**

(2) **ユニット名を表示します。ユニット名を変更または削除した場合は、旧ユニット名を表示します。**

(3) **変更前の名前を表示します。**

(4) **変更後の名前を表示します。削除の場合は表示しません。**

(5) **サブコマンドで指定した種別を表示します。**

UN	ユニット名
ED	外部定義シンボル名
ER	外部参照シンボル名

(6) **変更／削除の別を示します。**

RENAME	変更
DELETE	削除

3.4.7 強制定義シンボルリスト

DEFINE オプション／サブコマンド指定により、外部参照シンボルの強制定義処理を行った場合、PRINT オプション／サブコマンドを指定すると、**図 3.16** に示す形式でリストを出力します。

```
                    *** LINKAGE EDITOR DEFINE LIST ***

      UNDEFINED SYMBOL                 DEFINED SYMBOL            DEFINED VALUE

      _strcpy                          _Func1                      H'000005CE
        (1)                              (2)                          (3)
```

図 3.16　強制定義シンボルリスト例

【注】 (1) **強制的に定義したシンボル名を表示します。**

(2) **外部定義シンボルを指定した場合は、そのシンボル名を表示します。**

(3) **定義したシンボルの値を 16 進数で表示します。**

4. コンパイラのオプション・環境変数

4.1 コマンドラインの形式

コンパイラを起動するコマンドラインの形式は次のとおりです。

```
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>］［，<サブオプション>］．．．］
```

4.2 オプション一覧

コンパイラオプション一覧を**表 4.1** に示します。

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。また、下線は省略時解釈を示します。

また、日立統合化開発環境の対応するダイアログメニューを、タブ名[項目]で示します。ダイアログメニューの詳細は、日立統合化開発環境の**ユーザーズマニュアル**を参照して下さい。

表 4.1　コンパイラオプション一覧

No	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	Source [Language]	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	組み込み向け C++言語	ECpp	Source [Check under EC++ Language Spec.]	EC++言語仕様に基づいて シンタックスチェック
3	インクルードファイルディレクトリ	Include = <パス名>,…	Source[Show Entries For:] [Include File Directories]	インクルードファイルの 取り込み先パス名を指定
4	デフォルトインクルードファイル	PREInclude = <ファイル名>,…	Source[Show Entries For:] [Preinclude Files]	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
5	インクルードファイルの基点	CHgincpath	- (常に chgincpath が有効)	相対パス形式のインクルード ファイルの検索基点ディレクトリ を変更
6	マクロ名の定義	DEFine = <マクロ名>=<名前> | <マクロ名>=<定数> | <マクロ名>	Source[Show Entries For:] [Defines]	<名前>を<マクロ名>として定義 <定数>を<マクロ名>として定義 <マクロ名>を定義したものと仮定
7	文字列内の文字コード	EUc SJis LATin1	Source [Caracter Code]	euc コードを選択 sjis コードを選択 latin1 コードを選択
8	コメントのネスト	COMment	Source [Allow Comment Nest]	コメント(/* */)のネストを許す

No	項目	コマンドライン形式	ダイアログメニュー	指定内容
9	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],…]	Source[Show Entries For:] [Messages]	出力あり 出力なし （エラー番号、範囲を指定可能）
10	オブジェクト形式	Code = Machinecode | Asmcode	Object [Output File Type]	機械語プログラムを出力 アセンブリプログラムを出力
11	オブジェクトコード内漢字変換	OUtcode = Euc | Sjis	Object [Out Character Code]	euc コード sjis コード
12	文字列出力領域	STring = Const | Data	Object [Store String Data in]	定数領域へ出力 初期化データ領域へ出力
13	オブジェクトファイル	OBject [= <ファイル名>] NOOBject	Object [Output File path]	出力あり 出力なし
14	プリプロセッサ展開	PREProcessor [= <ファイル名>]	Source [Preprocessed Source File Path]	プリプロセッサ展開後のソースログラムを出力
15	デバッグ情報	DEBug NODEBug	Object [Debug Information]	出力あり 出力なし
16	ブラウザ情報	BRowser	Object [Browser Information]	C++プログラムのブラウザ情報を出力
17	Mac レジスタ保証	MAcsave	Object [Interrupt handler saves/restores MACH and MACL registers if used]	割り込み関数の前後で mac レジスタを常に保証
18	引数格納レジスタ	REGParam = 2 | 3	Tuning [Change number of parameter registers from 2(default) to 3]	(E)R0,(E)R1 を使用 (E)R0,(E)R1,(E)R2 を使用
19	構造体、共用体、クラスメンバの境界調整数	PAck = 1 | 2	Object [Pack struct, union and class]	1 データの境界調整に従う
20	リストファイル	List [= <ファイル名>] NOList	List [Generate List File]	出力あり 出力なし
21	リスト内容と形式	SHow = SOurce | NOSOurce | Object | NOObject | STatistics | NOSTatistics | Allocation | NOAllocation | Expansion|NOExpansion | Width = <数値> | Length = <数値> 省略時：(w = 0, l = 0)	List [List File Contents] [List File Format]	 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

No	項目	コマンドライン形式	ダイアログメニュー	指定内容
22	セクション名	SEction = 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> 省略時： (p=P,c=C,d=D,b=B)	Section [Program Section (P)] [Const Section (C)] [Data Section (D)] [Uninitialized Data Section (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
23	最適化レベル	OPtimize = 0 | 1	Optimize [Optimization]	最適化なし 最適化あり
24	スピード優先最適化	SPeed [=Register | SHift | Loop | SWitch | Inline [=<数値>] | STruct | Expression]	Optimize [Optimize for speed] [Speed Sub-options]	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 ループ文をループ展開 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
25	モジュール間最適化	Goptimize	Optimize [Generate file for inter module optimization]	モジュール間最適化用付加情報出力
26	外部変数の最適化	Volatile NOVolatile	Optimize [Avoid optimizing external symbols treating them as volatile]	外部変数の最適化抑止 外部変数を最適化
27	列挙型サイズ	Byteenum	Optimize [Treat enum as char if it is in the range of char]	宣言した列挙型のデータを char 型で扱う
28	変数割り付けレジスタ数の拡張	Regexpansion NORegexpansion	Object [Increase a register for register variable]	(E)R3～(E)R6 を使用 (E)R4～(E)R6 を使用
29	共通式の最適化	CMncode	Optimize [Put common subexpression on a register temporary]	共通式削除の最適化強化
30	switch 文展開方式	CAse = Auto | Ifthen | Table	Optimize [Switch statement]	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
31	メモリ間接形式	INDirect	Tuning [Function Call]	関数呼び出しをメモリ間接形式で展開
32	短絶対アドレス	ABS8 ABS16	Tuning [Data Access]	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

No	項目	コマンドライン形式	ダイアログメニュー	指定内容
33	乗除算仕の拡張解釈	CPUExpand NOCPUExpand	Tuning [Mul/Div Operation Specification]	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
34	ブロック転送命令	EEpmov	Tuning [Use EEPMOVE in Block Copy]	構造体の代入式を eepmov 命令で展開
35	CPU／動作モード	CPu = 2600N | 2600A [:<ビット幅>] | 2000N | 2000A [:<ビット幅>] | 300HN | 300HA[:<ビット幅>] | 300 | 300L | 300Reg	CPU [CPU Selection]	 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
36	サブコマンドファイルの選択	SUbcommand = <ファイル名>	—	<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

4.3 オプションの内容

(1) C/C++言語の選択

■コマンドライン形式

```
LANg = C | CPp
```

■ダイアログメニュー

```
Source[Language]
```

■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。

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

(2) 組み込み向け C++言語

■コマンドライン形式

```
ECpp
```

■ダイアログメニュー

```
Source[Check under EC++ Language Spec.]
```

■説明

Embedded C++ 言語仕様に基づいて、C++プログラムをシンタックスチェックします。

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”6515 (E) EC++ useless keyword “キーワード” ” を出力します。

また 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、、ウォーニングメッセージ”1500 (W) EC++ does not support “クラス名” ”を出力します。ウォーニングメッセージ 1500 出力時のコンパイラ生成オブジェクトプログラムは、EC++オプションを指定しない場合と変わりません。

(3) インクルードファイルディレクトリ

■コマンドライン形式

```
Include = <パス名>,…
```

■ダイアログメニュー

```
Source[Show Entries For:][Include File Directories]
```

■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。

(4) デフォルトインクルードファイル

■コマンドライン形式

```
PREInclude = <ファイル名>,…
```

■ダイアログメニュー

```
Source[Show Entries For:][Preinclude Files]
```

■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(){}
```

(5) インクルードファイルの基点

■コマンドライン形式

```
CHgincpath
```

■ダイアログメニュー

なし（常に chgincpath が有効）

■説明

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を検索するとき、ソースファイルのあるディレクトリを基点のディレクトリとします。指定のない場合は、コンパイラを起動したカレントディレクトリを基点ディレクトリとします。**図** 4.1 に例を示します。

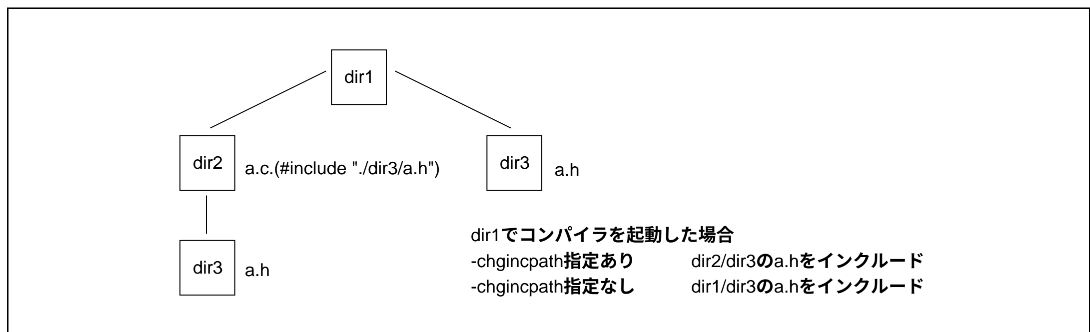


図 4.1 **基点ディレクトリ**

(6) マクロ名の定義

■コマンドライン形式

DEFine = <マクロ名> = <名前> |
<マクロ名> = <定数> |
<マクロ名>

■ダイアログメニュー

```
Source[Show Entries For:][Defines]
```

■説明

マクロ名を定義します。

サブオプションで指定できるマクロ名、名前および定数の定義を**表** 4.2 に示します。マクロ名、名前および定数の長さはそれぞれ先頭から 31 文字までが有効となります。サブオプションの<マクロ名>＝<名前>および<マクロ名>＝<定数>の形式では、それぞれ名前または定数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>を単独で指定した場合は、そのマクロ名が定義されたものと仮定することができます。サブオプションは 16 個まで指定できます。

表 4.2 define **オプションで指定できるマクロ名、名前、定数**

No.	項目	説明
1	**マクロ名**	**英字またはアンダラインで始まり、そのあとに** 0 **個以上の英字、アンダラインまたは数字が続く文字列です。**
2	**名前**	**英字またはアンダラインで始まり、そのあとに** 0 **個以上の英字、アンダラインまたは数字が続く文字列です。**
3	**定数**	10 **進定数：**1 **個以上の数字（**0〜9**）の繰り返し、または** 1 **個以上の数字の繰り返しにピリオドが続き、そのあとに** 0 **個以上の数字が続く文字列です。** 8 **進定数：数字０で始まり、その後に１個以上の０〜７の数字が続く文字列です。** 16 **進定数：数字０に** x **が続き、その後に１個以上の数字、または** A〜F, a〜f **が続く文字列です。**

(7) **文字列内の文字コード**

■コマンドライン形式

```
EUc
SJis
LATin1
```

■ダイアログメニュー

```
Source[Character Code]
```

■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** 4.3 に示します。

表 4.3 **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

注意 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。**

(8) コメントのネスト

■コマンドライン形式

```
COMment
```

■ダイアログメニュー

```
Source[Allow Comment Nest]
```

■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。

(9) インフォメーションメッセージ

■コマンドライン形式

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

■ダイアログメニュー

```
Source[Show Entries For:][Messages]
```

■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、サブオプションでエラー番号を指定すると、指定したインフォメーションレベル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
```

0005 および 300～306 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。

(10) オブジェクト形式

■コマンドライン形式

```
Code ＝ Machinecode ｜ Asmcode
```

■ダイアログメニュー

```
Object[Output File Type]
```

■説明

オブジェクトプログラムの形式を指定します。

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。

code＝asmcode と debug オプションを同時に指定すると、アセンブリプログラム中に.LINE 制御命令が出力されます。アセンブル時にも debug オプションを指定することにより、C/C++ソースプログラムレベルでのステップ実行が可能になります。

本オプションを省略した場合は、code＝machinecode が指定されたものと解釈します。

注意 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。**

---

(11) オブジェクトコード内漢字変換

■コマンドライン形式

```
OUtcode = Euc | Sjis
```

■ダイアログメニュー

```
Object[Out Character Code]
```

■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。

(12) 文字列出力領域

■コマンドライン形式

```
STring ＝ Const | Data
```

■ダイアログメニュー

```
Object[Store String Data in]
```

■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、**「8. システム組み込み」**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。

(13) オブジェクトファイル

■コマンドライン形式

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

■ダイアログメニュー

```
Object[Output File Path]
```

■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>指定をしない場合には、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。

注意 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。**
code、outcode、debug、browser、pack、string、show=object,statistics,allocation、section、optimize、speed、goptimize、byteenum、volatile、regexpansion、cmncode、case、indirect、abs8、abs16、cpuexpand、eepmov、regparam、macsave

(14) プリプロセッサ展開

■コマンドライン形式

```
PREProcessor [= <ファイル名>]
```

■ダイアログメニュー

```
Source[Preprocessed Source File Path]
```

■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＞指定をしない場合は、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。

注意 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、browser、pack、string、show=object,statistics,allocation、section、optimize、speed、goptimize、byteenum、volatile、regexpansion、cmncode、case、indirect、abs8、abs16、cpuexpand、eepmov、regparam、macsave

(15) デバッグ情報

■コマンドライン形式

```
DEBug
NODEBug
```

■ダイアログメニュー

```
Object[Debug Information]
```

■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。

object=machinecode を同時に指定した場合は、直接デバッグ情報が出力されます。

object=asmcode を同時に指定した場合は、.LINE 制御命令がアセンブリプログラム中に組み込まれます。アセンブル時にも debug オプションを指定することにより、C/C++ソースプログラムレベルでのステップ実行が可能になります。

本オプションは、最適化オプションを指定した場合でも有効です。

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。

注意　ソースレベルのデバッグを行うためには、リンク時に debug または sdebug オプションを指定する必要があります。

---

(16) ブラウザ情報

■コマンドライン形式

```
BRowser
```

■ダイアログメニュー

```
Object[Browser Information]
```

■説明

ブラウザツールに必要なブラウザ情報を出力します。コンパイル時にオブジェクトプログラムの出力ディレクトリに cppdtb というディレクトリがなければ作成し、その下にブラウザ情報ファイルを生成します。本オプションは C++プログラムをコンパイルするときのみ有効です。

注意　C++プログラムのコンパイル時に browser オプションを指定すると nodebug オプションは無効になります。

---

(17) mac レジスタ保証

■コマンドライン形式

```
MAcsave
```

■ダイアログメニュー

```
Object[Interrupt handler saves/restores MACH and MACL registers if used]
```

■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たとき、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。

(18) **引数格納レジスタ**

■コマンドライン形式

```
REGParam = 2 | 3
```

■ダイアログメニュー

```
Object[Change number of parameter registers from 2(default) to 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。

(19) **構造体、共用体、クラスメンバの境界調整数**

■コマンドライン形式

```
PAck = 1 | 2
```

■ダイアログメニュー

```
Object[Pack struct, union and class]
```

■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。

pack オプション指定時の構造体メンバの境界調整数を**表** 4.4 に示します。

表 4.4 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

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**「**5.2.2 **データの内部表現** (2)**複合型」**を参照してください。

(20)　リストファイル

■コマンドライン形式

```
List [= <リストファイル名>]
NOList
```

■ダイアログメニュー

```
List[Generate List File]
```

■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、**「3.1　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。

(21)　リスト内容と形式

■コマンドライン形式

```
SHow=     SOurce｜NOSOurce｜
          Object｜NOObject｜
          STatistics｜NOSTatistics｜
          Allocation｜NOAllocation｜
          Expansion｜NOExpansion
          Width=〈数値〉｜
          Length=〈数値〉
```

■ダイアログメニュー

```
List[List File Contents][List File Format]
```

■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。サブオプションの一覧を**表** 4.5 に示します。

表 4.5　show オプションのサブオプション一覧

項番	サブオプション名	意味
1	source	ソースプログラムのリストを出力します。
2	nosource	ソースプログラムのリストを出力しません。
3	object	オブジェクトプログラムのリストを出力します。
4	noobject	オブジェクトプログラムのリストを出力しません。
5	statistics	統計情報のリストを出力します。
6	nostatistics	統計情報のリストを出力しません。
7	allocation	シンボル割り付け情報のリストを出力します。

項番	サブオプション名	意味
8	noallocation	シンボル割り付け情報のリストを出力しません。
9	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。
10	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には、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。
11	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。
12	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。

本項で記した各リストの具体例については「**3.2　コンパイルリストの見方**」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
です。

(22)　セクション名

■コマンドライン形式

```
SEction=   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>
```

■ダイアログメニュー

```
Section[Program Section (P)][Const Section (C)]
       [Data Section (D)][Uninitialized Data Section (B)]
```

■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。
プログラムとセクション名の対応についての詳細は、「**5.2　オブジェクトプログラムの構造**」を参照してください。
<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、32 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。

(23) 最適化レベル

■コマンドライン形式

```
OPtimize ＝ 0 ｜ 1
```

■ダイアログメニュー

```
Optimize[Optimize]
```

■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 とみなします。

注意　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。

(24) スピード優先最適化

■コマンドライン形式

```
SPeed ［＝   Register｜
             SHift｜
             Loop｜
             SWitch｜
             Inline [ = <数値>]｜
             STruct｜
             Expression ］
```

■ダイアログメニュー

```
Optimize[Optimize for speed][Speed Sub-options]
```

■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／出口でレジスタを退避／回復するコードを実行時ルーチンを使用せずに PUSH､POP 命令で展開します。

speed＝shift オプションは、シフト演算を実行時ルーチンを使わないコードで展開します。

speed＝loop オプションは、ループ展開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。インライン展開の条件については、**「5.6.1（5）関数のインライン展開」**を参照してください。

speed＝struct オプションは、構造体型や double 型の代入文を実行時ルーチンを使わないコードで展開します。

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。

注意　最適化なし（optimize＝0）を指定したとき、speed＝loop, inline は無効となります。

(25) モジュール間最適化

■コマンドライン形式

```
Goptimize
```

■ダイアログメニュー

```
Optimize[Generate file for inter-Module Optimization]
```

■説明

モジュール間最適化用付加情報を出力します。

オブジェクトプログラム出力ディレクトリに、「ch38iop」という名前のディレクトリを自動作成し、ディレクトリ「ch38iop」の下にソースプログラムと同じファイル名で拡張子「iop」の付加情報ファイルを出力します。

(26) 外部変数の最適化

■コマンドライン形式

```
Volatile
NOVolatile
```

■ダイアログメニュー

```
Optimize[Avoid optimizing external symbols treating them as volatile]
```

■説明

volatile オプションは、全ての外部変数に対して最適化を行いません。

novolatile オプションは、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;
}
```

（a）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
```

（b）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
```

(27) 列挙型サイズ

■コマンドライン形式

```
Byteenum
```

■ダイアログメニュー

```
Optimize[Treat enum as char if it is register variable]
```

■説明

enum 宣言した列挙型のデータを char 型として扱います。
本オプションが指定された場合で、かつ、enum 宣言した列挙型のメンバの値が全て-128～127 の範囲のとき、列挙型データを char 型として扱います。
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 1 つでも-128～127 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;}
```

（a）byteenum指定時

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

（b）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 2byte 領域に割り当てます
```

(28) 変数割り付けレジスタ数の拡張

■コマンドライン形式

```
Regexpansion
NORegexpansion
```

■ダイアログメニュー

```
Optimize[Increase a Register for Register Variable]
```

■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、**「付録 A.1（8）レジスタ」**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。

(29) 共通式の最適化

■コマンドライン形式

CMncode

■ダイアログメニュー

Optimize[Put Common Subexpression on a register temporary]

■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。

(30) switch 文展開方式

■コマンドライン形式

CAse = Auto | Ifthen | Table

■ダイアログメニュー

Optimize[Switch statement]

■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。この展開方式は、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;
          default:          b=0; 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）

```
MOV.W  @_a,R0                 MOV.W  @_a,R0
MOV.B  R0H,R0H                SUB.W  #1,R0
BNE    Ld                     CMP.W  #2,R0
CMP.B  #1,R0L                 BHI    Ld
BEQ    L1                     ADD.W  R0,R0
CMP.B  #2,R0L                 MOV.W  @(L,ER0),R0
BEQ    L2                     JMP    @ER0
CMP.B  #3,ROL                 ：
BEQ    L3                     L:（ジャンプテーブル）
BRA    Ld
case=ifthen 時                case=table 時
```

表 4.6　switch 文　展開方式の比較

case 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	22 バイト	18	28（22+6）バイト	28
3		30		

(31)　メモリ間接形式

■コマンドライン形式

INDirect

■ダイアログメニュー

Tuning[Function Call]

■説明

ソースプログラム内で呼び出す関数を全てメモリ間接（@@aa:8）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は、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が出力されます。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、**「5.6.1（2）セクション切り替え機能」**を参照してください。

注意　アドレステーブルを格納できるエリアは、0x0000～0x00FF 番地に制限されています。リンク時には、start コマンドで”$INDIRECT”セクションのアドレスを 0x0000～0x00FF 番地の範囲内に指定してください。

(32)　短絶対アドレスの指定

■コマンドライン形式

ABS8

ABS16

■ダイアログメニュー

Tuning[Data Access]

■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 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 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 および#pragma abs16 拡張子でも指定できます。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。

注意　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、「付録 F　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、「5.6.1（4）セクション切り替え機能」を参照してください。

---

(33)　乗除算仕様の拡張解釈

■コマンドライン形式

```
CPUExpand
NOCPUExpand
```

■ダイアログメニュー

```
Tuning[Mul/Div Operation Specification]
```

■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。本オプションの指定による乗除算のコード展開を**表** 4.7 に示します。

表 4.7　cpuexpand オプションの演算仕様

<table>
<tr><th rowspan="2">対象演算</th><th colspan="2">us1*us2 の演算サイズ（H8S/2600 の例）</th></tr>
<tr><th>cpuexpand 指定時</th><th>nocpuexpand 指定時</th></tr>
<tr><td>unsigned
short us1, us2;
unsigned long ul;
ul＝us1*us2;</td><td>us1*us2 は unsigned long で演算します。
出力例： 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。</td><td>us1*us2 は unsigned short で演算します。
出力例： 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。</td></tr>
<tr><td>unsigned
short us1, us2, us3;
unsigned short us;
us＝us1*us2/us3;</td><td>us1*us2 は unsigned long で演算します。
出力例：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。</td><td>us1*us2 は unsigned short で演算します。
出力例： 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。</td></tr>
</table>

注意　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。

(34)　ブロック転送命令

■コマンドライン形式

```
EEpmov
```

■ダイアログメニュー

```
Tuning[Use EEPMOVE in Block Copy]
```

■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、ブロック転送命令 EEPMOV でコード展開します。

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。

注意　EEPMOV 命令実行中に NMI 割り込みが発生すると、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。NMI 割り込みが発生す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。

(35) CPU／動作モード

■コマンドライン形式

```
CPu＝        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
```

■ダイアログメニュー

```
CPU[CPU Selection]
```

■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。サブオプションの一覧を**表 4.8** に示します。

表 4.8 cpu オプションのサブオプション一覧

項番	サブオプション名	意　味
1	2600n	H8S/2600 用ノーマルモードのオブジェクトを作成します。
2	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。
3	2000n	H8S/2000 用ノーマルモードのオブジェクトを作成します。
4	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。
5	300hn	H8/300H 用ノーマルモードのオブジェクトを作成します。
6	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。
7	300	H8/300 のオブジェクトを作成します。
8	300l	H8/300 のオブジェクトを作成します。クロスアセンブラとの互換のために用意しています。
9	300reg	H8/300 のオブジェクトを作成します。旧バージョンとの互換のために用意しています。

注意
1. 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。
2. 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。

(36) サブコマンドファイルの選択

■コマンドライン形式

SUbcommand = <サブコマンドファイル名>

■ダイアログメニュー

なし

■説明

subcommand オプションは、C 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,length＝0 -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,length＝0 -debug -byteenum test.c
```

4.4　コンパイラの環境変数

コンパイラで使用する環境変数の使用方法を**表 4.9** に示します。

表 4.9　コンパイラの環境変数

No.	環境変数	説明
1	path	コンパイラ本体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= <コンパイラ本体パス名>;[<既存パス名>;...] unix 版 C シェル　%set path =(<コンパイラ本体パス名> $path) ボーンシェル　%PATH=:<コンパイラ本体パス名>[:<既存パス名>...] %export PATH
2	H38CPU	コンパイラの -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　省略時解釈 2600n　—　— 2600a　20 | 24 | 28 | 32　24 2000n　—　— 2000a　20 | 24 | 28 | 32　24 300hn　—　— 300ha　20 | 24　24 300　—　— 300l　—　— 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>[:<アドレス空間のビット幅>] unix 版 C シェル　%setenv H38CPU <CPU／動作モード>[:<アドレス空間のビット幅>] ボーン　%H38CPU=<CPU／動作モード>[:<アドレス空間のビット幅>] シェル　%export　H38CPU
3	CH38	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 版には省略時解釈はありません。 指定フォーマット： PC 版　C> set CH38= <インクルードパス名> unix 版 C シェル　%setenv CH38 <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> %export　CH38
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. プログラミング

5.1 コンパイラの限界値

コンパイラがコンパイルできるソースプログラムの限界値を**表 5.1** に示します。ソースプログラムを作成する場合は、この限界値の範囲内で作成してください。

表 5.1 コンパイラの限界値

No.	分類	項目		限界値
1	コンパイラの起動	一度にコンパイルできるソースプログラムの数		制限なし*1
2		define オプションで指定できるマクロ名の総数		制限なし
3		ファイル名の長さ		128 文字
4	ソースプログラムの行数	1 行の長さ		8192 文字
5		1ファイルあたりのソースプログラムの行数		65535 行
6		コンパイル可能なソースプログラムの行数		制限なし
7	プリプロセッサ	#include 文によるファイルのネストの深さ		30 レベル
8		#define 文によるマクロ名の総数		制限なし
9		マクロ定義、マクロ呼び出しで指定できる引数の数		63 個
10		マクロ名の再置き換えの数		32 回
11		#if、#ifdef、#ifndef、#else、#elif 文のネストの深さ		32 レベル
12		#if、#elif 文で指定できる演算子と被演算子の合計数		512 個
13	宣言	関数定義の数		制限なし
14		外部結合となる識別子（外部名）の数		32767 個
15		一つの関数内で有効な識別子（内部名）の数		32767 個
16		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
17		配列の次元数		6 次元
18		配列、構造体のサイズ*2	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	65535 バイト
			H8/300H アドバンストモード	16777215 バイト
			H8S/2600 アドバンストモード、 H8S/2000 アドバンストモード	4294967295 バイト
19	文	複文のネストの深さ		256 レベル
20		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文のネストの深さ		256 レベル
21		一つの関数内で指定できる goto ラベルの数		511 個
22		switch 文の数		256 個
23		switch 文のネストの深さ		128 レベル
24		case ラベルの数		511 個
25		for 文のネストの深さ		128 レベル
26	式	文字列の長さ		512 文字
27		関数定義、関数呼び出しで指定できる引数の数		63 個*3

No.	分類	項目	限界値
28	式	一つの式で指定できる演算子と被演算子の合計数	約500個
29	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*4

【注】 *1 PC版はコマンドラインの制約により127文字までの入力となります。

*2 アドバンストモードの場合、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。

*3 非静的関数メンバのときは、62個になります。

*4 open関数で一度にオープンできるファイルの数を指定できます。
詳細は「8.4（4）（a）標準入出力の初期設定ルーチン（_INIT_IOLIB）の作成例」を参照してください。

5.2 オブジェクトプログラムの構造

ここでは、C/C++プログラム、標準ライブラリが使用するメモリ領域の性質について述べます。メモリ領域の性質には、以下の項目があります。

- セクション
  メモリ領域のうち、コンパイラが静的に割り付ける領域は、セクションを構成します。セクションにはセクション名とセクション種別があります。
- 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。
- 初期値の有無
  プログラム実行開始時の初期値の有無です。
- 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。

C/C++プログラム、標準ライブラリが使用するメモリ領域の種類とその性質の概要を**表 5.2** に示します。

表 5.2　メモリ領域の種類とその性質の概要

No	名称	セクション		書き込み操作	初期値の有無	境界調整数	内容
		名称	種別				
1	プログラム領域	P*[1]	code	不可	有	2byte	機械語を格納
2	定数領域	C*[1]	data	不可	有	2byte	const 型のデータを格納
3	初期化データ領域	D*[1]	data	可	有	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	可	無	2byte	初期値のないデータを格納
5	定数領域 (8bit アドレス空間)	$ABS8C*[1]	data	不可	有	2byte	abs8 オプション指定時、または #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	可	有	2byte	abs8 オプション指定時、または #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	可	無	2byte	abs8 オプション指定時、または #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	不可	有	2byte	abs16 オプション指定時、または#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	可	有	2byte	abs16 オプション指定時、または#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	可	無	2byte	abs16 オプション指定時、または#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)	$INDIRECT *[1]	data	不可	有	2byte	indirect オプション指定時、または#pragma indirect で指定された関数のアドレスを格納
12	初期化データセクションのアドレス領域	C$DSEC	data	不可	有	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

No	名称	セクション		書き込み操作	初期値の有無	境界調整数	内容
		名称	種別				
13	未初期化データセクションのアドレス領域	C$BSEC	data	不可	有	2byte	未初期値化データ領域セクションのアドレス、最終アドレスを格納
14	C++初期処理データ領域	C$INIT	data	不可	有	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のアドレスを格納
15	C++後処理データ領域	C$END	data	不可	有	2byte	グローバルクラスオブジェクトに対して呼び出されるデストラクタのアドレスを格納
16	C++仮想関数表領域	C$VTBL	data	不可	有	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
17	スタック領域	S	stack	可	無	2byte	プログラム実行に必要な領域。「8.2.2 動的領域の割り付け」参照。
18	ヒープ領域	—	—	可	無	—	ライブラリ関数malloc、realloc、calloc、new で使用する領域。「8.2.2 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。

■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
プログラム領域(main(){...})
定数領域(c)
初期化データ領域(a)
未初期化データ領域(b)

■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;
  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
プログラム領域(f(){...})
定数領域(c)
初期化データ領域(d)
未初期化データ領域(a, b)
初期処理データ領域(&A::A)
後処理データ領域(&A::~A)

コンパイラが生成する領域と格納されるデータ

5.3 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。

5.3.1 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** 5.3 に示します。

表 5.3 **スカラ型・基本型の内部表現**

No	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（上記以外）	2	2	有	-2^{15}（-32768）	$2^{15}-1$（32767）
12	float	4	2	有	-∞	+∞
13	double, long double	8	2	有	-∞	+∞
14	ポインタ (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）
15	ポインタ ＊[1] (H8/300H アドバンストモード)	4	2	無	0	$2^{24}-1$ (16777215)
16	ポインタ (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)

No	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
17	リファレンス (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）
18	リファレンス *1 (H8/300H アドバンストモード)	4	2	無	0	2^{24}-1 (16777215)
19	リファレンス (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)
20	データメンバへのポインタ (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）
21	データメンバへのポインタ *1 (H8/300H アドバンストモード)	4	2	無	0	2^{24}-1 (16777215)
22	データメンバへのポインタ (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)
23	関数メンバへのポインタ *2 (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)	10	2	－	－	－
24	関数メンバへのポインタ *2 (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)	6	2	－	－	－

【注】 *1 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。
*2 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
    };
};
```

5.3.2 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 5.4 に複合型、クラス型の内部表現を示します。

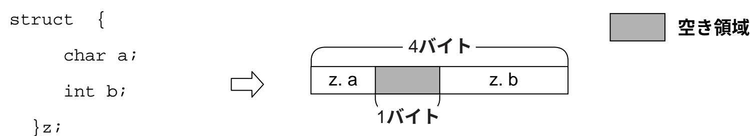
表 5.4 **複合型、クラス型の内部表現**

No.	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** 「(1)**構造体データの割り付け方**」**参照**	struct { **境界調整数** 1byte char a,b; **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** 「(2)**共用体データの割り付け方**」**参照**	union { **境界調整数** 1byte char a,b; **サイズ** 1byte };

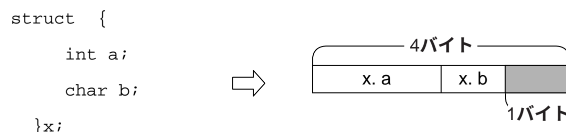
4	**クラス型**	(1)**基底クラス、仮想関数がある場合**:
		常に2
		(2)**上記以外**:
		データメンバの境界調整数のうち最大値
			データメンバ、仮想関数表へのポインタ、仮想基底クラスへのポインタの和
			「(3)**クラスデータの割り付け方**」**参照**
				(H8S/2600 **アドバンストモード時**)
				class A{
				char a; :**境界調整数** 1byte
				}; :**サイズ** 1byte
				
				class B: public A{
				virtual void f(); :**境界調整数** 2byte
				}; :**サイズ** 6byte

(1) 構造体データの割り付け方

(a) 構造体型の各メンバを割り付ける時、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。



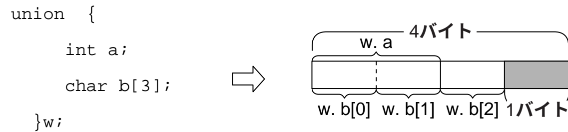
(b) 構造体が2バイトの境界調整数を持つ場合で、最後のメンバが奇数バイト目で終わっているとき、その次のバイトも含めて構造体型の領域として扱います。



(2) 共用体データの割り付け方

(a) 共用体が2バイトの境界調整数を持つ場合で、メンバのサイズの最大値が奇数バイトのとき、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。

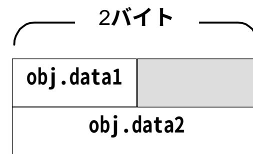
```
union  {
    int a;
    char b[3];
  }w;
```



(3) クラスデータの割り付け方

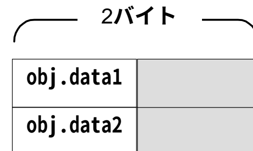
(a) 基底クラス、仮想関数がないクラスの場合、構造体データの割り付け規則に従ってデータメンバを割り付けます。

```
class A{
   char data1;
   short data2;
 public:
   A();
   int getData1()
       {return data1;}
}obj;
```



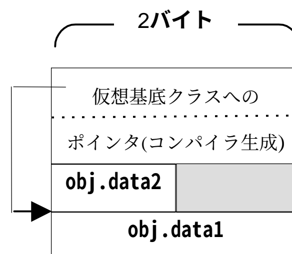
(b) 基底クラスがある場合、クラスの境界調整数は2バイトになります。

```
class A{
   char data1;
};
class B:public A{
   char data2;
}obj;
```



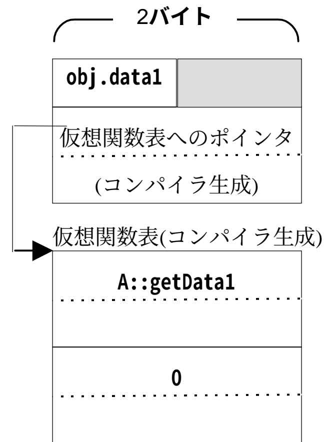
(c) クラスに仮想基底クラスがある場合、仮想基底クラスへのポインタを割り付けます。

```
class A{
   short data1;
};
class B: virtual protected A{
   char data2;
}obj;
```



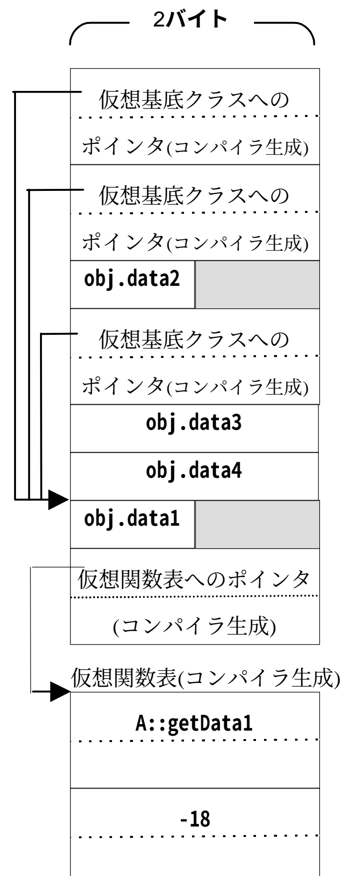
(d) クラスに仮想関数がある場合、コンパイラは仮想関数表を生成し、仮想関数表へのポインタを割り付けます。

```
class A{
    char data1;
  public:
    virtual int getData1();
}obj;
```



(e) 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。

```
class A{
    char data1 ;
    virtual short getData1();
};
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;
```



(f)　空クラスの場合、1バイトのダミー領域を割り付けます。

```
class A{
  void fun();
}obj;
```


(g)　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。

```
class A{
  void fun();
};
class B: A{
  void sub();
};
```


(h)　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。

```
class A{
  void fun();
};
class B: A{
char data1;
}obj;
```

5.3.3 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。

(1) ビットフィールドのメンバ

表 5.5 にビットフィールドメンバの仕様を示します。

表 5.5 ビットフィールドメンバの仕様

No.	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。

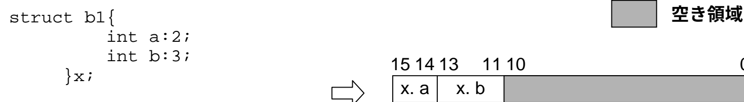
注意 符号付き（signed）で宣言されたサイズが 1 ビットのビットフィールドのデータは、データそのものを符号として解釈します。したがって、表現できる値は 0 と−1 だけになります。0 と 1 を表現する場合には、必ず符号なし（unsigned）で宣言してください。

---

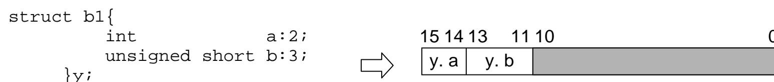
(2) ビットフィールドの割り付け方

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。

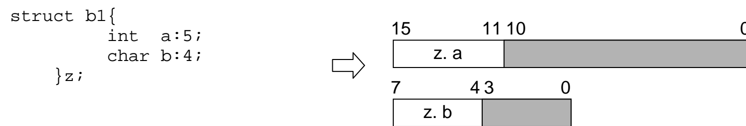
(a) ビットフィールドのメンバは領域内で左（上位ビット側）から順に詰め込みます。



(b) 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。



(c)　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。



(d)　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：

(e)　無名のビットフィールドあるいはビット幅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：

5.4 C++プログラムと C プログラムとの結合

C++プログラムから C プログラムの関数を参照する場合は、リンケージ指定子 extern “C”を使用します。この機能を用いて C++プログラムで既存の C プログラム資産を活用することができます。

また、標準インクルードファイルは C と C++で共用できるようになっていますので、C++プログラムから C ライブラリ関数を使用することができます。

C++プログラムから C プログラムの関数を呼び出す例を以下に示します。

■例

```
#include <stdio.h>            //標準インクルードファイルをインクルードします。
extern "C" int f(int);        //関数 f が C プログラムの関数であることを指定します。
extern int data;              //C プログラムのグローバル変数を extern 宣言します。

void g()
{
...
printf("test start¥n");       //C ライブラリ関数を使用します。
f(1);                         //C の関数 f を呼び出します。
data = 2;                     //C プログラムのグローバル変数に設定します。
...
}
```

5.5 アセンブリプログラムとの結合

本コンパイラでは、#pragma 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

5.5.1 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

C++プログラムでは、関数名を外部定義します。

アセンブリプログラムでは、「.IMPORT」制御命令を用いて関数名を C++エンコード規則に従った名前で外部参照宣言します。

C++エンコード規則は**「付録 G エンコード規則」**を参照してください。また、コンパイルリストのオブジェクト情報から参照することもできます。

また、「extern “C”」を用いて宣言することにより、(2)の変数名と同じ規則（先頭に下線(_)を付与）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)

```
int f(int a)
{
    …
}
```

アセンブリプログラム(参照する側)

```
.IMPORT   __f__Fi
.SECTION P,CODE,ALIGN=2
     :
JSR       @__f__Fi
     :
.END
```

5.5.2 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。スタックポインタより下位アドレスに格納されたデータは、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。

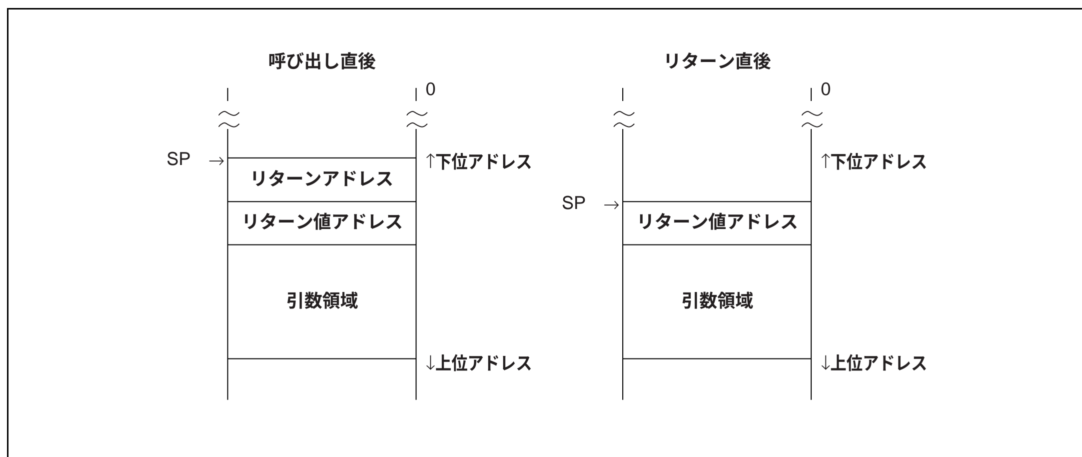


図 5.1 スタックフレームの割り付け、解放に関する規則

(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 5.6 に示します。

表 5.6　関数呼び出し前後のレジスタ保証規則

No	項目	引数格納レジスタ数 *1	CPU 種類と対象レジスタ		プログラミングにおける留意点
			H8S/2600、H8S/2000、H8/300H	H8/300	
1	保証しないレジスタ	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,ER1,ER2	R0,R1,R2	
2	保証するレジスタ	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、リターン時に回復する。
		3	ER3～ER6	R3～R6	

注意　*1：引数格納レジスタ数は、regparam オプションで指定できます。

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
```

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
```

- MOV.L ER1,@(4,SP) / MOV.L ER0,ER6：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
- JSR @_sub：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)

```
void sub(void)
{
    …
}
```

アセンブリプログラム(呼び出す側)

```
    .IMPORT   __sub__Fv
    .SECTION P,CODE,ALIGN=2
             :
    MOV.L    ER1,@(4,SP)
    MOV.L    ER0,ER6
    JSR      @__sub__Fv
              :
    RTS
    .END
```

- MOV.L ER1,@(4,SP) / MOV.L ER0,ER6：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
- JSR @__sub__Fv：関数名は**「付録 G エンコード規則」**＊1に従って参照

【注】＊1 C++関数のエンコード名は、show=object オプション指定により、オブジェクトリストで参照することもできます。また、「extern “C”」を用いて宣言することにより、C プログラムの関数と同じ規則（先頭に下線(_)を付与）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。

(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;
 }
```
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 → 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 → 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**図** 5.2 に、引数割り付け領域の一般規則を**表** 5.7 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。

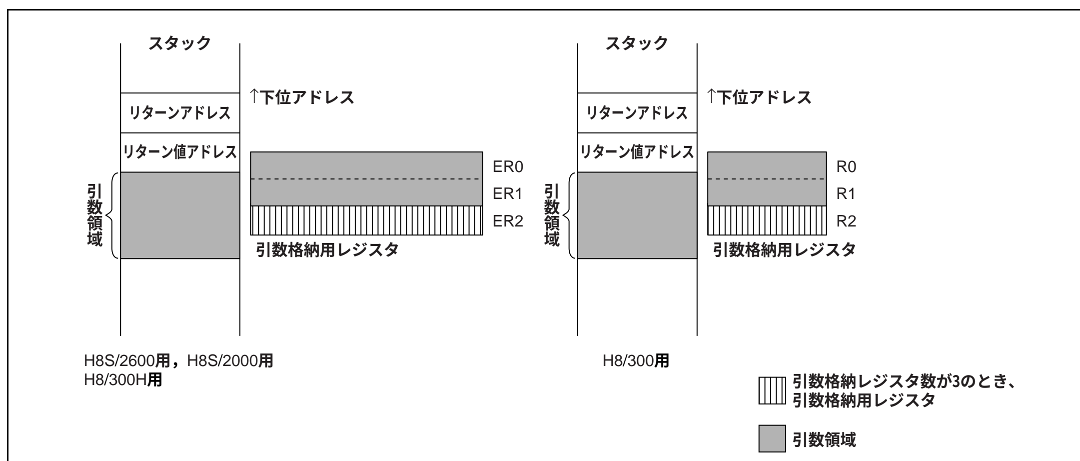


図 5.2　**引数の割り付け領域**

表 5.7 引数割り付け領域の一般規則

No	CPU 種類	引数格納レジスタ数 *1	割り付け規則		
			レジスタに割り付ける引数		スタックに割り付ける引数
			引数格納用レジスタ	対象の型	
1	H8S/2600, H8S/2000, H8/300H	2	ER0、ER1	char、unsigned char、short、unsigned short、int、unsigned int、long、unsigned long、float、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		
2	H8/300	2	R0、R1	char、unsigned char、short、unsigned short、int、unsigned int、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、宣言の中で対応する型のない引数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。
        :
```

(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 5.3** に示します。

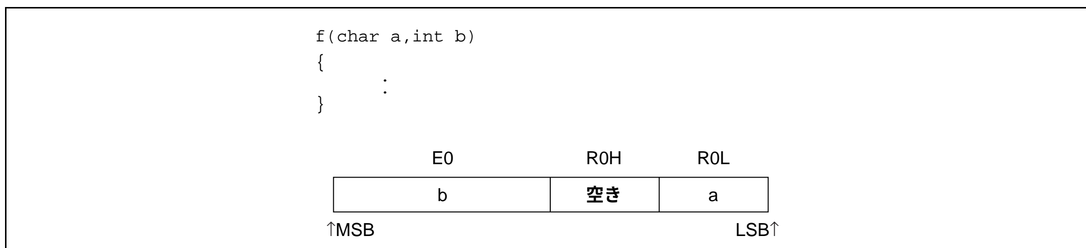


図 5.3 引数格納用レジスタの割り付け例（H8S/2600 用）

■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、H8S、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。

「付録 B．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 5.8** 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 5.4 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 5.8 リターン値の型と設定場所

No.	リターン値の型	リターン値の設定場所	
		H8S/2600 用、H8S/2000 用、H8/300H 用	H8/300 用
1	char、unsigned char	レジスタ（R0L）	レジスタ（R0L）
2	short、unsigned short、int、unsigned int	レジスタ（R0）	レジスタ（R0）
3	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 ノーマルモード：（R0） アドバンストモード：（ER0）	レジスタ（R0）
4	long、unsigned long、float	レジスタ（ER0）	リターン値設定領域（メモリ）
5	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）

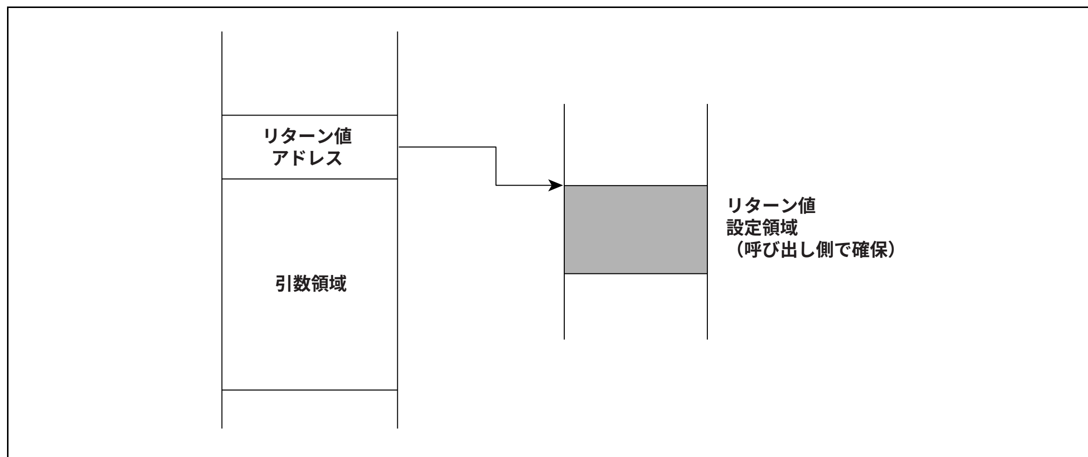


図 5.4 リターン値をメモリに設定する場合の設定領域

5.6 拡張機能

コンパイラの拡張機能として、次の機能をサポートしています。

- #pragma 拡張子
- セクションアドレス演算子
- 組み込み関数

以下、各々の機能について説明します。

5.6.1 #pragma 拡張子

#pragma 拡張子の一覧を表 5.9 に示します。

表 5.9 #pragma 拡張子一覧

No.	#pragma 拡張子	機能
1	#pragma interrupt	割り込み関数の作成
2	#pragma stacksize	スタックセクションの作成
3	#pragma entry	エントリ関数の作成
4	#pragma section、 #pragma abs8 section、 #pragma abs16 section、 #pragma indirect section	セクションの切り替え指定
5	#pragma abs8、 #pragma abs16	短絶対アドレス形式でアクセスする変数を指定
6	#pragma indirect	メモリ間接で関数呼び出しを行う関数を指定
7	#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
8	#pragma inline	関数のインライン展開を指定
9	#pragma inline_asm	アセンブリ記述関数をインライン展開
10	#pragma asm、 #pragma endasm	アセンブリ命令列の埋め込み
11	#pragma regsave、 #pragma noregsave	関数の入口／出口でレジスタの退避／回復コード出力を制御
12	#pragma global_register	グローバル変数にレジスタを割り付け

(1)　割り込み関数の作成機能

■記述方法

#pragma interrupt（関数名［（割り込み仕様）］［，関数名［（割り込み仕様）］...］）

<関数名>:グローバル関数、C++静的メンバ関数

割り込み仕様の一覧を**表** 5.10 に示します。

表 5.10　**割り込み仕様の一覧**

No.	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＞　：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

■説明

#pragma interrupt を用いて外部（ハードウェア）割り込み関数となる関数を宣言します。

(a) #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ER0、ER1(H8/300時はR0、R1)を含むレジスタを保証し、RTE命令でリターンします。
(b) 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
```

*1 STK＋100を割り込み関数「f」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**5.5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：H8S/2000、H8S/2600時のみ）を回復し、割り込み関数から復帰してください。

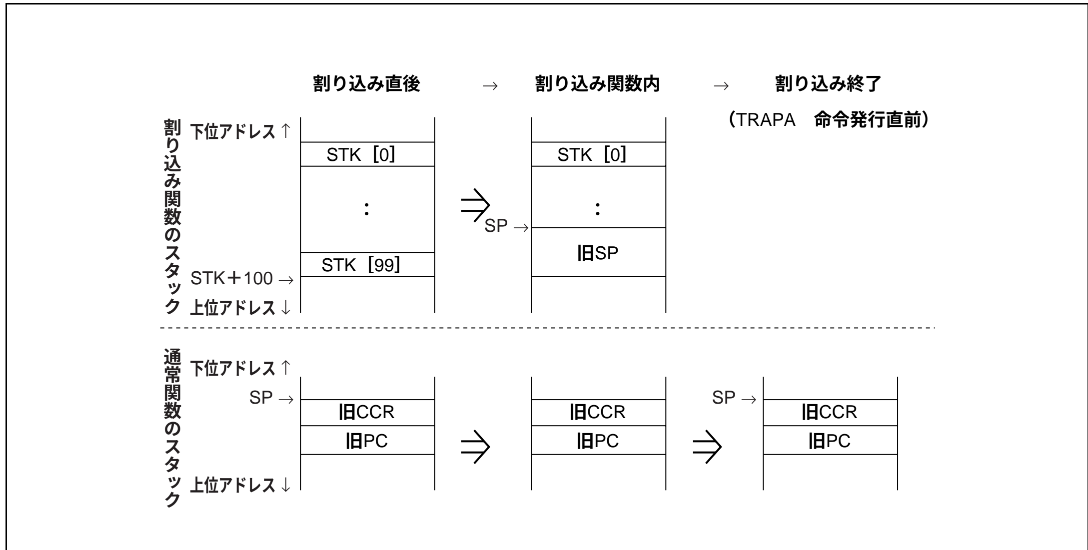


図 5.5 割り込み関数によるスタック使用例

(c) 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 f1(void)                         /* JMP @f2:24   ;JMP @f2 でリターン */
{
    :
}
```

(d) 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。

■使用上の注意事項

(a) #pragma interrupt宣言後の関数定義または関数宣言が対象になります。

例：

```
#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)
{    …   }
```

(b) 割り込み関数として定義できるのは、グローバル関数と静的メンバ関数だけです。グローバル関数にstaticを指定してもexternとして処理します。
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。

例：

```
#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 でないのでエラーになります */
```

(c) 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。

例：

```
#pragma interrupt(f1)
void f1(void){…}
int f2(void){ f1( );}  /* 関数 f1 は割り込み関数なので、エラーになります */
```

(d) 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。

例：

```
#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 */
```

(e) #pragma interrupt文1行に宣言できる関数の数は63個までです。
(f) 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。
(g) 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。

(2) スタックセクションの作成機能

■記述方法

#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。

■使用上の注意事項

(a) 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。
(b) #pragma stacksizeはファイル内で一回しか指定できません。

■使用例

```
#pragma stacksize 100              .SECTION S,STACK
                                   .RES.W   50
```

(3) エントリ関数の作成機能

■記述方法

#pragma entry <関数名>[(sp=<定数>)]

■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。

(a) 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：

```
#pragma entry INIT(sp=0x8000)           .SECTION P,CODE
void INIT()                         _INIT:
{                                       MOV.W    #H'32768,SP
         :                                       :
}
```

(b) 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：

```
#pragma stacksize 100        .SECTION S,STACK
#pragma entry INIT           .RES.W   50
void INIT()                  .SECTION P,CODE
{                      _INIT:
  :                          MOV.W    #STARTOF S + SIZEOF S,SP
}                                        :
```

(c) sp指定も#pragma stacksize宣言もない場合、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コマンドで指定してください。

例：

```
#pragma entry INIT            .SECTION S,STACK  ;サイズ０のＳセクションを生成
void INIT()                   .SECTION P,CODE
{                        _INIT:
   :                          MOV.W    #STARTOF S + SIZEOF S,SP
}                                           :
```

(d) 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。

■使用上の注意事項

(a) #pragma entry <関数名>指定は、<関数名>の宣言前に行ってください。
(b) 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。

(4) セクション切り替え機能

■記述方法

```
#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。

表 5.11 セクション切り替え機能とセクション名

No	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	8bit 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	16bit 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	$INDIRECT<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なります。

■使用上の注意事項

(a) #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。
(b) 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。

■使用例

```
#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){ a=c;}       /* fは,セクションPabcに割り付きます */
#pragma section
int b;                    /* bは,セクションBに割り付きます */
void g(void){ b=c;}       /* gは,セクションPに割り付きます */
```

(5) 短絶対アドレス機能

■記述方法

```
#pragma abs8 （＜変数名＞［，＜変数名＞...］）
#pragma abs16（＜変数名＞［，＜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。

(a) #pragma 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。
(b) #pragma 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。
(c) セクション名の切り替え方法については、**「（4）セクション切り替え機能」**の#pragma abs8 sectionおよび#pragma abs16 sectionを参照してください。

■使用上の注意事項

(a) #pragma abs8、#pragma abs16宣言後の変数定義・変数宣言が対象になります。
(b) #pragma abs8、#pragma 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。
(c) #pragma abs8、#pragma abs16文1行に宣言できる変数の数は63個までです。
(d) #pragma abs8、#pragma abs16で宣言した変数は、セクション切り替え機能を使用しない場合、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。
(e) #pragma abs8で宣言した変数が1byteアクセス対象でない場合は、エラーになります。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、クラスを宣言してください。ただし、pack=1オプションまたは#pragma pack 1拡張子により境界調整を１に指定した構造体、クラスは全て対象になります。

■使用例

```
#pragma abs8(c)
#pragma abs16(i)
char c;                  /* c はセクション名$ABS8B に割り付きます */
int i;                   /* i はセクション名$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){
    c=10;                /* c を 8 ビット絶対アドレスでアクセス     */
    i=100;               /* i を 16 ビット絶対アドレスでアクセス    */
    l=1000;              /* l を 32 ビット絶対アドレスでアクセス    */
}
```

(6) メモリ間接の関数呼び出し

■記述方法

#pragma indirect（関数名［, 関数名...］）

<関数名>:グローバル関数、C++メンバ関数

■説明

#pragma 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。

(a) #pragma indirectを用いて宣言された関数は、JSR　@@“$＋関数名”：8の形で呼び出します。また、メモリ間接呼び出し宣言された関数の本体が定義されている場合は、“$＋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。
(b) セクション名の切り替え方法については、**「（4）セクション切り替え機能」**の#pragma indirect sectionを参照してください。

■使用上の注意事項

(a) #pragma indirect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。
(b) #pragma indirect文1行に宣言できる関数の数は63個までです。
(c) #pragma indirectで宣言できる関数の数は、アドバンストモードのとき64個、またノーマルモードのとき128個までです。
リンク時にはアドレステーブルのセクションを0x0000～0x00FF番地に割り付けてください。
(d) #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。

例：

```
#include <indirect.h>  /* 実行時ルーチンをメモリ間接呼び出しする場合、*/
void main(void)         /* <indirect.h>をインクルード                */
{   …   }
```

■使用例

```
#pragma indirect (f)
unsigned char f(void)          /* アドレステーブルに$f を生成し、関数 f のアドレスを */
{   …   }                      /* 格納します                                       */
sub( )
{                              /* @@$f:8 メモリ間接で関数呼び出し                  */
    f( );
}
```

(7) 構造体、共用体、クラスメンバの境界調整数の指定

■記述方法

```
#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** 5.12 に示します。

表 5.12 #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

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「5.3 データの内部表現 (2)複合型」を参照してください。

■使用例

```
#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 S2 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              */
```

(8) **関数のインライン展開**

■記述方法

#pragma inline（関数名［，関数名...］）

<関数名>：グローバル関数、メンバ関数

■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。

■使用上の注意事項

(a) #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。
(b) #pragma inline文1行に宣言できる関数の数は63個までです。
(c) #pragma 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 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。
(d) #pragma inline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。

■**使用例**

```
#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。*/
}
```

(9) **アセンブラ埋め込みインライン展開**

■**記述方法**

```
#pragma inline_asm (<関数名>[,<関数名>...])
<関数名>:グローバル関数
```

■**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。

■**使用上の注意事項**

(a) #pragma inline_asm宣言後に出現する関数定義を対象にします。オーバロード関数は指定できません。
(b) #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。
(c) 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、**「H8S, H8/300シリーズクロスアセンブラユーザーズマニュアル」**を参照してください。
(d) 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。
(e) 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。
(f) 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。
(g) 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:
```

■使用例

```
#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
}
```

(10) アセンブラ埋め込み機能

■記述方法

```
#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。

■使用上の注意事項

(a) 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。

(b) 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。

(c) #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。

(d) 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。
```

■使用例

```
void func(void)
{
#pragma asm
    CLRMAC                  ;MAC レジスタを０設定します。
#pragma endasm
          :
}
```

(11) レジスタ退避／回復コード制御機能

■記述方法

```
#pragma regsave (<関数名> [, <関数名>...] )
#pragma noregsave (<関数名> [, <関数名>...] )
<関数名>:グローバル関数、C++メンバ関数
```

■説明

#pragma regsave、#pragma noregsave を用いて、レジスタ退避／回復コードを制御します。

(a) #pragma 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。

(b) #pragma 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。

■使用上の注意事項

(a) #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。

(b) #pragma regsave／noregsave文1行に宣言できる関数の数は63個までです。

■使用例(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  */
#pragma interrupt g          /* 関数 g は割り込み関数               */

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         */
```

(12) グローバル変数のレジスタ割り付け

■記述方法

```
#pragma global_register (<変数名>=<レジスタ名>[,<変数名>=<レジスタ名>...])
<変数名>:グローバル変数、C++静的メンバデータ
```

■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。

■使用上の注意事項

(a) #pragma global_register宣言後の変数定義・変数宣言が対象になります。

(b) 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。

(c) 指定可能なレジスタは、ER4、ER5（H8/300時はR4、R5）です。

(d) 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。
(e) 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。
(f) 割り込み関数内での設定・参照は保証されません。
(g) 変数、レジスタの重複指定、#pragma abs8、#pragma abs16との二重指定はできません。

■使用例

```
#pragma global_register(x=R4,y=R5L)
int      x;
char     y;
void func1(void)
{    x++; }
void func2(void)
{    y=0; }
void func(int  a)
{
    x = a;
    func1();
    func2();
}
```

5.6.2 セクションアドレス演算子

■記述方法

```
__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。

■使用例

セクション初期化で使用する例を示します。セクション初期化方法の詳細は、**「8.システム組み込み」**を参照してください。

```
#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 0x0        /* 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       */
}
```

5.6.3 組み込み関数

システム制御命令等、以下に示すC/C++言語で記述できない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
- 特殊命令（TRAPA、SLEEP、MOVFPE、MOVTPE、TAS、EEPMOV、NOP）
- ローテート演算
- コンディションコード反映演算
- 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。

■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     */
```

組み込み関数の一覧を**表 5.11** に示します。

表 5.11 組み込み関数の一覧

No	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）

No	項目	仕様	機能
15	積和演算	long mac(long val,int* ptr1, int* ptr2, unsigend long count)	MAC 命令を用いて val+Σi=0,count-1(ptr1[i]∗ptr2[i])を演算 またはリングバッファ機能を用いて、 val+ Σi=0,count-1(ptr1[i]∗*((ptr2+i)&mask))
		long macl(long val,int* ptr1,int* ptr2, unsigned long count,unsigned long mask)	
16	特殊命令	void trapa(unsigned int trap_no)	TRAPA #trap_no に展開
17		void sleep(void)	SLEEP 命令に展開
18		void movfpe(char* addr,char data)	MOVFPE 命令を用いて、*addr を data に設定
19		void movtpe(char data,char* addr)	MOVTPE 命令を用いて、data を*addr に設定
20		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
21		void eepmov(char* dst,char* src, unsigned char size)	EEPMOV 命令を用いて size バイト分 *src を*dst に転送
		void eepmov(char* dst,char* src, unsigned int size)	
22		void nop(void)	NOP 命令に展開
23	ローテート演算	char rotlc(int count,char data)	data を count ビット分左ローテート
		int rotlw(int count,int data)	
		long rotll(int count,long data)	
24		char rotrc(int count,char data)	data を count ビット分右ローテート
		int rotrw(int count,int data)	
		long rotrl(int count,long data)	
25	コンディションコード反映演算	int ovfaddc(char dst,char src, char* 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
		int ovfaddw(int dst,int src, int* rst)	
		int ovfaddl(long dst,long src, long* rst)	
26		int ovfsubc(char dst,char src, char* 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
		int ovfsubw(int dst,int src, int* rst)	
		int ovfsubl(long dst,long src, long* rst)	
27		int ovfshalc(char dst,char* 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
		int ovfshalw(int dst,int* rst)	
		int ovfshall(long dst,long* rst)	
28		int ovfnegc(char dst,char* rst)	-dst を*rst に設定し、 演算結果のコンディションコードを反映
		int ovfnegw(int dst,int* rst)	
		int ovfnegl(long dst,long* rst)	
29	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
30		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

(1) set_imask_ccr 関数

■呼び出しインタフェース

```
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。

■使用例

```
#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    */
}
```

(2) get_imask_ccr 関数

■呼び出しインタフェース

```
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。

■使用例

```
#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      */
      :
}
```

(3) set_ ccr 関数

■呼び出しインタフェース

```
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。

■使用例

```
#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                    */
}
```

(4) get_ccr 関数

■呼び出しインタフェース

```
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。

■使用例

```
#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 */
       :
}
```

(5) and_ccr 関数

■呼び出しインタフェース

```
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。

■使用例

```
#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        */
}
```

(6) or_ccr 関数

■呼び出しインタフェース

```
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。

■使用例

```
#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      */
}
```

(7) xor_ccr 関数

■呼び出しインタフェース

```
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。

■使用例

```
#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          */
}
```

(8) set_imask_exr 関数

■呼び出しインタフェース

```
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 2600a、2000a、2600n、2000n のときのみ有効です。

■使用例

```
#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。              */
}
```

(9) get_imask_exr 関数

■呼び出しインタフェース

```
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 2600a、2000a、2600n、2000n のときのみ有効です。

■使用例

```
#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。                            */
}
```

(10) set_exr 関数

■呼び出しインタフェース

```
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 2600a、2000a、2600n、2000n のときのみ有効です。

■使用例

```
#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        */
}
```

(11) get_exr 関数

■呼び出しインタフェース

```
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 2600a、2000a、2600n、2000n のときのみ有効です。

■使用例

```
#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          */
        :
}
```

(12) and_exr 関数

■呼び出しインタフェース

```
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 2600a、2000a、2600n、2000n のときのみ有効です。

■使用例

```
#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           */
}
```

(13) or_exr 関数

■呼び出しインタフェース

```
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 2600a、2000a、2600n、2000n のときのみ有効です。

■使用例

```
#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           */
}
```

(14) xor_exr 関数

■呼び出しインタフェース

```
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 2600a、2000a、2600n、2000n のときのみ有効です。

■使用例

```
#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           */
}
```

(15) mac、macl 関数

■呼び出しインタフェース

```
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 2600a、2600n のときのみ有効です。

注意 **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。下記使用例の場合、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。**

■使用例

```
#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 を実行します          */
l2=macl(100,ptr1,ptr2,4,~4);   /* l2=100+0*9+1*8+2*9+3*8 を実行します          */
}                              /* ptr2[0],pre2[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                         */
```

(16) trapa 関数

■呼び出しインタフェース

```
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。

■使用例

```
#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           */
}
```

(17) sleep 関数

■呼び出しインタフェース

```
void sleep(void);
```

■説明

低消費電力状態命令 SLEEP に展開します。

■使用例

```
#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              */
}
```

(18) movfpe 関数

■呼び出しインタフェース

```
void movfpe(char* addr,char data);
```

■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dataへ取り出します。*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。

■使用例

```
#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アクセスできるように*/
char a,data;                  /* #pragma abs16で宣言します                        */
void f(void)
{
    movfpe(&a,data);          /* Eクロックに同期したタイミングでaをdataに転送します */
}
```

(19) movtpe 関数

■呼び出しインタフェース

```
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。

■使用例

```
#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アクセスできるように*/
char a,data;                  /* #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 */
```

(20) tas 関数

■呼び出しインタフェース

```
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2600a、2000a、2600n、2000nでのみ有効です。

■使用例

```
#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   */
```

(21) eepmov 関数

■呼び出しインタフェース

```
void eepmov(char* dst,char* src,unsigned char size);
```

または、

```
void eepmov(char* dst,char* 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。

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。

■使用例

```
#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 */
}
```

(22) nop 関数

■呼び出しインタフェース

```
void nop(void);
```

■説明

nop 命令に展開します。

■使用例

```
#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       */
}
```

(23) rotlc、rotlw、rotll 関数

■呼び出しインタフェース

```
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 ビットローテート（回転）し、その結果を返します。

■使用例

```
#include <machine.h>            /* 必ず<machine.h>をインクルードします */
int i,data;
void f(void)
{
    i=rotlw(5,data);            /* data を 5bit 左ローテートします       */
}
```

(24)　rotrc、rotrw、rotrl 関数

■呼び出しインタフェース

```
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 ビットローテート（回転）し、その結果を返します。

■使用例

```
#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       */
```

(25)　ovfaddc、ovfaddw、ovfaddl 関数

■呼び出しインタフェース

```
int ovfaddc(char dst,char src,char * rst);
int ovfaddw(int dst,int src,int * rst);
int ovfaddl(long dst, long src,long * rst);
```

■説明

ovfaddc、ovfaddw、ovfaddl 関数は、それぞれ 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、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。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 関数は、CPU／動作モードが 300 以外のときに有効です。

■使用例

```
#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;
}
```

(26) ovfsubc、ovfsubw、ovfsubl 関数

■呼び出しインタフェース

```
int ovfsubc(char dst,char src,char * rst);

int ovfsubw(int dst,int src,int * rst);

int ovfsubl(long dst, long src,long * rst);
```

■説明

ovfsubc、ovfsubw、ovfsubl 関数は、それぞれ 1 バイト、2 バイトおよび 4 バイト同士の dst と src の減算を行い、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 関数は、CPU／動作モードが 300 以外のときに有効です。

■使用例

```
#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;
}
```

(27) ovfshalc、ovfshalw、ovfshall 関数

■呼び出しインタフェース

```
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／動作モードが 300 以外のときに有効です。

■使用例

```
#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;
 }
```

(28) ovfnegc、ovfnegw、ovfnegl 関数

■呼び出しインタフェース

```
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／動作モードが 300 以外のときに有効です。

■使用例

```
#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 の結果のボローで*/
   dst=0;                     /* 分岐します                                       */
 }
```

(29) dadd 関数

■呼び出しインタフェース

```
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。

■使用例

```
#include <machine.h>           /* 必ず<machine.h>をインクルードします      */
char ptr1[10]={0,1,2,3,4,5,6,7,8,9};
char ptr2[10]={0,1,2,3,4,5,6,7,8,9};
char rst[10];
void main(void)
{
dadd((char)10,ptr1,ptr2,rst); /* ptr1,ptr2 を 10 桁の 10 進数として加算します*/
}                              /* rst=0,2,4,6,9,1,3,5,7,8 */
```

(30) dsub 関数

■呼び出しインタフェース

```
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。

■使用例

```
#include <machine.h>           /* 必ず<machine.h>をインクルードします      */
char ptr1[10]={1,0,0,0,0,0,0,0,0,0};
char ptr2[10]={0,1,2,3,4,5,6,7,8,9};
char rst[10];
void main(void)
{
dsub((char)10,ptr1,ptr2,rst); /* ptr1,ptr2 を 10 桁の 10 進数として減算します*/
}                              /* rst=0,8,7,6,5,4,3,2,1,1              */
```

5.7 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。

5.7.1 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。

■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*/
            fa=1.0/0.0;       /*（W）浮動小数点の 0 除算を検出します*/

/* 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 C 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 0xFF 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 SS{
        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"
void sub(void)
{
  unsigned char DDR=P4DDR.Schar;
  DDR &=~P0;
  P4DDR.Schar=DDR;
}
```

5.7.2 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); // 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; // エラー

const cvalue2 = 1; // 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; //エラー
}
```

```
void func(void *ptrv, int *ptri)
{
    ptri = (int *)ptrv; //OK
}
```

5.7.3 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) リンク時には CPU／動作モードに合わせた標準ライブラリをリンクしてください。

コンパイラが提供する標準ライブラリには 40 種類のライブラリがあります。必ず CPU／動作モードに対応するライブラリを指定してください。それ以外のライブラリを指定した場合の動作は保証しません。参照：**「1.3 標準ライブラリの概要」**

(d) コンパイル時に指定する regparam オプションは統一して下さい。

コンパイル時に regparam オプションを用いて指定する引数格納レジスタ数は、必ず統一して下さい。regparam 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。

また、リンク時に標準ライブラリを指定する場合、必ず regparam オプションに対応するライブラリを指定してください。それ以外のライブラリを指定した場合の動作は保証しません。参照：**「1.3 標準ライブラリの概要」**

5.7.4 トラブル発生時の対処方法

表 5.12　トラブル発生時の対処方法（1）

No.	現象	確認内容	対処方法	参照
1	コンパイラ出力のオブジェクトプログラムにアセンブラ埋め込み部分が出力されない。	コンパイラオプションで code＝asmcode を指定したか。	コンパイル時に、code＝asmcode を指定してください。	「5.6.1（10）アセンブラ埋め込み機能」
2	リンク時に、エラー No.314 cannot found section が出力される。	リンケージエディタの start オプションにおいて、コンパイラ出力のセクション名は大文字で指定しているか。	正しいセクション名を指定してください。	「5.2 オブジェクトプログラムの構造」 「4.コンパイラのオプション・環境変数」
3	リンク時に、エラー No.105 undefined external symbol が出力される。	C/C++プログラムとアセンブリプログラム間で変数を相互参照している場合、アセンブリプログラム内で下線を付加しているか。またはエンコード名を指定しているか。	正しい変数名で参照してください。	「5.5.1 外部名の相互参照方法」
		プログラムで C ライブラリ関数を使用していないか。	リンク時に入力ライブラリとして標準ライブラリを指定してください。	「1.3 標準ライブラリの概要」
		未定義参照シンボル名が＄で始まっていないか。（標準ライブラリ中の実行時ルーチンを使用しています）		
		C ライブラリ関数の標準入出力ライブラリを使用していないか。	低水準インタフェースルーチンを作成してリンクしてください。	「8.4（6）低水準インタフェースルーチン」
4	リンク時に、エラー No.108 relocation size overflow が出力される。	abs8、abs16 オプションを指定しているか。または、#pragma abs8、#pragma abs16 を使用しているか。	8 ビット、16 ビット短絶対アドレス領域を正しい領域に割り付けてください。	「4.　コンパイラのオプション・環境変数」 「5.6.1（5）短絶対アドレス機能」
5	デバッグ時にソースレベルのデバッグができない。	コンパイル、アセンブル、リンク時とも debug、sdebug オプションを指定したか。	コンパイル時、アセンブル時、およびリンク時のそれぞれに debug、sdebug オプションを指定してください。	「4.　コンパイラのオプション・環境変数」

6. 標準 C ライブラリ

6.1 ライブラリの概要

本章では、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。**「6.1 ライブラリの概要」**では、ライブラリの構成を概説し、本編の読み方および用語について説明します。以下の節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。

(1) 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 6.1 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 6.1 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

項番	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** 6.2 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 6.2　マクロ名定義からなる標準インクルードファイル

項番	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。

(2)　ライブラリ編の説明形式

次にライブラリ編の説明形式について説明します。

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 6.1 参照**）、その後、各関数ごとの説明を行う（**図 6.2 参照**）という形式で構成されています。

図 6.1 に標準インクルードファイル説明の凡例、**図 6.2** に関数説明の凡例を示します。

項番＜標準インクルードファイル名＞

■機能概要

本標準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。

■定義名一覧

本標準インクルードファイル内で定義されるマクロ名、あるいは関数宣言のひとつひとつに対する説明を行います。

定義名	種類	説明
標準インクルードファイル内で定義されるマクロ名、または宣言される関数名を示します。	定義名の分類を示します。定義名の分類は以下のとおりです。 定義名の分類 1. マクロ名　：引数を持たない形式のマクロ名称であることを意味します。 2. マクロ　：引数を持つ形式のマクロ名称であることを意味します。 3. 関数　：関数により実現されているライブラリの関数宣言であることを意味します。 4. タグ名　：構造体宣言における構造体宣言名であることを示しています。	定義名に対する説明を行ないます。

その他、本標準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などを説明します。

図 6.1　標準インクルードファイル説明の凡例

項番　関数名

関数名を示します。ライブラリが関数として実現されているかマクロとして実現されているかを示します。*

■機能

ライブラリ関数の機能概要を説明します。

■呼び出し手順

ライブラリ関数の書式、およびライブラリ関数使用時に必要となるデータの宣言を示します。

■パラメータ

No.	名前	型	意味
項番です。	パラメタの名称を示します。	パラメタの型を示します。	パラメタの意味を説明します。

■リターン値

型　　：リターン値の型を示します。
正常　：ライブラリ関数が正常終了したときのリターン値を示します。
異常　：ライブラリ関数が異常終了したときのリターン値を示します。

ライブラリ関数の詳細な使用を説明します。

【注意】

ライブラリ関数の使用上の注意事項を説明します。

【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。（処理系定義の値はユーザーズマニュアルを参照してください。）

図 6.2　関数説明の凡例

【注】*　関数とマクロとの相違は、「（3）（c）関数とマクロ」を参照してください。
**　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「6.2<stddef.h>」を参照してください。

(3) ライブラリ関数の説明で使用する用語

(a) 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。

(b) FILE 構造体およびファイルポインタ

（a）で述べた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、成功したか、失敗したか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。

(c) 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。

(d)　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 という名前は、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。

(e)　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。

(f)　ヌル文字

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。

(g)　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。

(h)　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。テキストファイルの実現法についてはユーザーズマニュアルを参照してください。
- 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。

(i)　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。

(j)　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 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、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精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。

(k) 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 6.3** に示す 12 種類があります。

表 6.3 ファイルアクセスモードの種類

項番	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。

(l) 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。

(m) 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、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有効です。

(n) 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いいます。

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。

(4) ライブラリ使用時の注意事項

(a) 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。

(b) 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。2 章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。

6.2　＜stddef.h＞

■機能概要

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。

■定義名一覧

定義名	種類	説明
ptrdiff_t	マクロ名	二つのポインタを減算した結果の型を示します。
size_t	マクロ名	sizeof 演算子による演算結果の型を示します。
NULL	マクロ名	ポインタが何も指していない時の値を示します。 この値は、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。
errno	マクロ名	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。

上記のマクロ名は、すべて処理系定義です。

■処理系定義仕様

No.	項目	コンパイラの仕様
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 H8S/2600 用ノーマルモード H8S/2000 用ノーマルモード H8S/300H 用ノーマルモード H8/300 用
		long 型　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

6.3 ＜assert.h＞

■機能概要

プログラム中に診断機能を付け加えます。

■定義名一覧

定義名	種類	説明
assert	マクロ	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）。

assert 関数はマクロとして実現されています。

【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。

(1) assert マクロ

■機能

プログラム中に診断機能を付け加えます。

■呼び出し手順

```
#include <assert.h>
int expression;


          assert (expression);
```

■パラメタ

No.	名前	型	意味
1	expression	int	評価する式

■リターン値

型　： void
正常： ―
異常： ―

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ΔFILEΔ<ファイル名>,lineΔ<行番号>

6.4 ＜ctype.h＞

■機能概要

文字に対して、その種類の判定や変換を行います。

■定義名一覧

定義名	種類	説明
isalnum	関数	英字または 10 進数字かどうかを判定します。
isalpha	関数	英字かどうかを判定します。
iscntrl	関数	制御文字かどうかを判定します。
isdigit	関数	10 進数字かどうかを判定します。
isgraph	関数	空白を除く印字文字かどうかを判定します。
islower	関数	英小文字かどうかを判定します。
isprint	関数	空白を含む印字文字かどうかを判定します。
ispunct	関数	特殊文字かどうかを判定します。
isspace	関数	空白類文字かどうかを判定します。
isupper	関数	英大文字かどうかを判定します。
isxdigit	関数	16 進数字かどうかを判定します。
tolower	関数	英大文字を英小文字に変換します。
toupper	関数	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** 6.4 に示します。

表 6.4　文字の種類

項番	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 0 x20～0 x7E 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、及び 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

No.	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 6.4 に示します。

表 6.5　真となる文字の集合

No.	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’

(1) isalnum 関数

■機能

文字が英字または 10 進数字であるかどうか判定します。

■呼び出し手順

```
#include <ctype.h>
int c, ret;

        ret=isalnum(c);
```

■パラメタ

No.	名前	型	意味
1	c	int	判定する文字

■リターン値

型　：　int
正常：　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
異常：　―

(2) isalpha 関数

■機能

文字が英字であるかどうか判定します。

■呼び出し手順

```
#include <ctype.h>
int c , ret;

        ret=isalpha(c);
```

■パラメタ

No.	名前	型	意味
1	c	int	判定する文字

■リターン値

型　：　int
正常：　文字 c が英字の時：0 以外
　　　　文字 c が英字以外の時：0
異常：　―

(3) iscntrl 関数

■機能

文字が制御文字であるかどうか判定します。

■呼び出し手順

```
#include <ctype.h>
int c, ret;

         ret=iscntrl(c);
```

■パラメタ

No.	名前	型	意味
1	c	int	判定する文字

■リターン値

型　：　int
正常：　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
異常：　―

(4) isdigit 関数

■機能

文字が 10 進数字であるかどうか判定します。

■呼び出し手順

```
#include <ctype.h>
int c, ret;

         ret=isdigit(c);
```

■パラメタ

No.	名前	型	意味
1	c	int	判定する文字

■リターン値

型　：　int
正常：　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
異常：　―

(5) isgraph 関数

■機能

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。

■呼び出し手順

```
#include <ctype.h>
int c, ret;

         ret=isgraph(c);
```

■パラメタ

No.	名前	型	意味
1	c	int	判定する文字

■リターン値

型　：　int
正常：　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
異常：　―

(6) islower 関数

■機能

文字が英小文字であるかどうか判定します。

■呼び出し手順

```
#include <ctype.h>
int c, ret;

         ret=islower(c);
```

■パラメタ

No.	名前	型	意味
1	c	int	判定する文字

■リターン値

型　：　int
正常：　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
異常：　―

(7) isprint 関数

■機能

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。

■呼び出し手順

```
#include <ctype.h>
int c, ret;

         ret=isprint(c);
```

■パラメタ

No.	名前	型	意味
1	c	int	**判定する文字**

■リターン値

型　：　int
正常：　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
異常：　—

(8) ispunct 関数

■機能

文字が特殊文字であるかどうか判定します。

■呼び出し手順

```
#include <ctype.h>
int c, ret;

         ret=ispunct(c);
```

■パラメタ

No.	名前	型	意味
1	c	int	**判定する文字**

■リターン値

型　：　int
正常：　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
異常：　—

(9) isspace 関数

■機能

文字が空白類文字であるかどうか判定します。

■呼び出し手順

```
#include <ctype.h>
int c, ret;

        ret=isspace(c);
```

■パラメタ

No.	名前	型	意味
1	c	int	判定する文字

■リターン値

型　：　int
正常：　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
異常：　―

(10) isupper 関数

■機能

文字が英大文字であるかどうか判定します。

■呼び出し手順

```
#include <ctype.h>
int c, ret;

        ret=isupper(c);
```

■パラメタ

No.	名前	型	意味
1	c	int	判定する文字

■リターン値

型　：　int
正常：　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
異常：　―

(11) isxdigit 関数

■機能

文字が 16 進数字かどうか判定します。

■呼び出し手順

```
#include <ctype.h>
int c, ret;

        ret=isxdigit(c);
```

■パラメタ

No.	名前	型	意味
1	c	int	判定する文字

■リターン値

型　：　int
正常：　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
異常：　—

(12) tolower 関数

■機能

英大文字を対応する英小文字に変換します。

■呼び出し手順

```
#include <ctype.h>
int c, ret;

        ret=tolower(c);
```

■パラメタ

No.	名前	型	意味
1	c	int	変換する文字

■リターン値

型　：　int
正常：　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
異常：　—

(13) toupper 関数

■機能

英小文字を対応する英大文字に変換します。

■呼び出し手順

```
#include <ctype.h>
int c, ret;

         ret=toupper(c);
```

■パラメタ

No.	名前	型	意味
1	c	int	変換する文字

■リターン値

型　：　int
正常：　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
異常：　—

6.5 <float.h>

■機能概要

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。

■定義名一覧

定義名	種類	説明
FLT_RADIX	マクロ名	指数部表現における基数を示します。
FLT_ROUNDS	マクロ名	加算演算結果が丸められ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。 ・加算演算結果を丸める場合：正の値 ・加算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	マクロ名	乗算演算結果においてガードビットが用いられ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。 ・ガードビットが用いられる場合：1 ・ガードビットが用いられない場合：0
FLT_NORMALIZE	マクロ名	浮動小数点数の値が正規化されて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。 ・正規化されている場合：1 ・正規化されていない場合：0
FLT_MAX	マクロ名	float 型の浮動小数点数値として表現できる最大値を示します。
DBL_MAX	マクロ名	double 型の浮動小数点数値として表現できる最大値を示します。
LDBL_MAX	マクロ名	long double 型の浮動小数点数値として表現できる最大値を示します。
FLT_MIN	マクロ名	float 型の浮動小数点数値として表現できる正の値での最小値を示します。
DBL_MIN	マクロ名	double 型の浮動小数点数値として表現できる正の値での最小値を示します。
LDBL_MIN	マクロ名	long double 型の浮動小数点数値として表現できる正の値での最小値を示します。
FLT_MAX_EXP	マクロ名	float 型の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を示します。
DBL_MAX_EXP	マクロ名	double 型の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を示します。
LDBL_MAX_EXP	マクロ名	long double 型の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を示します。
FLT_MIN_EXP	マクロ名	float 型の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を示します。
DBL_MIN_EXP	マクロ名	double 型の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を示します。
LDBL_MIN_EXP	マクロ名	long double 型の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を示します。
FLT_MAX_10_EXP	マクロ名	float 型の浮動小数点数値として表現できる 10 のべき乗の最大値を示します。
DBL_MAX_10_EXP	マクロ名	double 型の浮動小数点数値として表現できる 10 のべき乗の最大値を示します。
LDBL_MAX_10_EXP	マクロ名	long double 型の浮動小数点数値として表現できる 10 のべき乗の最大値を示します。

定義名	種類	説明
FLT_MIN_10_EXP	マクロ名	float 型の正の値として表現できる浮動小数点数値の 10 のべき乗の最小値を示します。
DBL_MIN_10_EXP	マクロ名	double 型の正の値として表現できる浮動小数点数値の 10 のべき乗の最小値を示します。
LDBL_MIN_10_EXP	マクロ名	long double 型の正の値として表現できる浮動小数点数値の 10 のべき乗の最小値を示します。
FLT_DIG	マクロ名	float 型の浮動小数点数値の 10 進精度の最大桁数を示します。
DBL_DIG	マクロ名	double 型の浮動小数点数値の 10 進精度の最大桁数を示します。
LDBL_DIG	マクロ名	long double 型の浮動小数点数値の 10 進精度の最大桁数を示します。
FLT_MANT_DIG	マクロ名	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を示します。
DBL_MANT_DIG	マクロ名	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を示します。
LDBL_MANT_DIG	マクロ名	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を示します。
FLT_EXP_DIG	マクロ名	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を示します。
DBL_EXP_DIG	マクロ名	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を示します。
LDBL_EXP_DIG	マクロ名	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を示します。
FLT_POS_EPS	マクロ名	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	マクロ名	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	マクロ名	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	マクロ名	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	マクロ名	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	マクロ名	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	マクロ名	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	マクロ名	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	マクロ名	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。
FLT_NEG_EPS_EXP	マクロ名	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	マクロ名	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	マクロ名	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。

上記のマクロ名はすべて処理系定義です。

6.6 ＜limits.h＞

■機能概要

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。

■定義名一覧

定義名	種類	説明
CHAR_BIT	マクロ名	char 型が何ビットから構成されるかを示します。
CHAR_MAX	マクロ名	char 型の変数が値として持つことのできる最大値を示します。
CHAR_MIN	マクロ名	char 型の変数が値として持つことのできる最小値を示します。
SCHAR_MAX	マクロ名	signed char 型の変数が値として持つことのできる最大値を示します。
SCHAR_MIN	マクロ名	signed char 型の変数が値として持つことのできる最小値を示します。
UCHAR_MAX	マクロ名	unsigned char 型の変数が値として持つことのできる最大値を示します。
SHRT_MAX	マクロ名	short 型の変数が値として持つことのできる最大値を示します。
SHRT_MIN	マクロ名	short 型の変数が値として持つことのできる最小値を示します。
USHRT_MAX	マクロ名	unsigned short 型の変数が値として持つことのできる最大値を示します。
INT_MAX	マクロ名	int 型の変数が値として持つことのできる最大値を示します。
INT_MIN	マクロ名	int 型の変数が値として持つことのできる最小値を示します。
UINT_MAX	マクロ名	unsigned int 型の変数が値として持つことのできる最大値を示します。
LONG_MAX	マクロ名	long 型の変数が値として持つことのできる最大値を示します。
LONG_MIN	マクロ名	long 型の変数が値として持つことのできる最小値を示します。
ULONG_MAX	マクロ名	unsigned long 型の変数が値として持つことのできる最大値を示します。

上記のマクロ名はすべて処理系定義です。

6.7　＜errno.h＞

■機能概要

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。

■定義名一覧

定義名	種類	本コンパイラの仕様
errno	マクロ	int 型変数、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る。
ERANGE	マクロ	「10.2　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	マクロ	
EDIV	マクロ	
ESTRN	マクロ	
PTRERR	マクロ	
ECBASE	マクロ	
ETLN	マクロ	
EEXP	マクロ	
EEXPN	マクロ	
EFLOATO	マクロ	
EFLOATU	マクロ	
EDBLO	マクロ	
EDBLU	マクロ	
ELDBLO	マクロ	
ELDBLU	マクロ	
NOTOPN	マクロ	
EBADF	マクロ	
ECSPEC	マクロ	

上記のマクロ名は、すべて処理系定義です。

6.8 <math.h>

■機能概要

各種の数値計算を行います。

■定義名一覧

定義名	種類	説明
EDOM	マクロ名	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 に設定する値を示しています。
ERANGE	マクロ名	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 に設定する値を示しています。
HUGE_VAL	マクロ名	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を示しています。
acos	関数	浮動小数点数の逆余弦を計算します。
asin	関数	浮動小数点数の逆正弦を計算します。
atan	関数	浮動小数点数の逆正接を計算します。
atan2	関数	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	関数	浮動小数点数のラディアン値の余弦を計算します。
sin	関数	浮動小数点数のラディアン値の正弦を計算します。
tan	関数	浮動小数点数のラディアン値の正接を計算します。
cosh	関数	浮動小数点数の双曲線余弦を計算します。
sinh	関数	浮動小数点数の双曲線正弦を計算します。
tanh	関数	浮動小数点数の双曲線正接を計算します。
exp	関数	浮動小数点数の指数関数を計算します。
frexp	関数	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	関数	浮動小数点数と 2 のべき乗の乗算を計算します。
log	関数	浮動小数点数の自然対数を計算します。
log10	関数	浮動小数点数の 10 を底とする対数を計算します。
modf	関数	浮動小数点数を整数部分と小数部分に分解します。
pow	関数	浮動小数点数のべき乗を計算します。
sqrt	関数	浮動小数点数の正の平方根を計算します。
ceil	関数	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	関数	浮動小数点数の絶対値を計算します。
floor	関数	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	関数	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。

上記のマクロ名はすべて処理系定義です。

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

No.	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「付録 A.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。

(1) acos関数

■機能

浮動小数点数の逆余弦を計算します。

■呼び出し手順

```
#include <math.h>
double d, ret;

        ret=acos(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	逆余弦を求める浮動小数点数

■リターン値

型　：　double
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。

acos関数のリターン値の範囲は[0, π]です。

【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。

(2) asin関数

■機能

浮動小数点数の逆正弦を計算します。

■呼び出し手順

```
#include <math.h>
double d, ret;

        ret=asin(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	逆正弦を求める浮動小数点数

■リターン値

型　：　double
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。

asin関数のリターン値の範囲は[－π／2, π／2]です。

【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。

(3)　atan 関数

■機能

浮動小数点数の逆正接を計算します。

■呼び出し手順

```
#include <math.h>
double d, ret;

         ret=atan(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	**逆正接を求める浮動小数点数**

■リターン値

型　：　double
正常：　d の逆正接値
異常：　—

atan 関数のリターン値の範囲は（－π／2，π／2）です。

(4) atan2 関数

■機能

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。

■呼び出し手順

```
#include <math.h>
double x, y, ret;

         ret=atan2(y, x);
```

■パラメタ

No.	名前	型	意味
1	x	double	除数
2	y	double	被除数

■リターン値

型　：　double
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。

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 6.3 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。

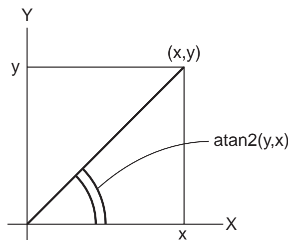


図 6.3　atan2 関数の意味

【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。

(5) cos 関数

■機能

浮動小数点数のラディアン値の余弦を計算します。

■呼び出し手順

```
#include <math.h>
double d, ret;

          ret=cos(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	余弦を求めるラディアン値

■リターン値

型 ： double
正常： d の余弦値
異常： —

(6) sin 関数

■機能

浮動小数点数のラディアン値の正弦を計算します。

■呼び出し手順

```
#include <math.h>
double d, ret;

          ret=sin(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	正弦を求めるラディアン値

■リターン値

型 ： double
正常： d の正弦値
異常： —

(7) tan関数

■機能

浮動小数点数のラディアン値の正接を計算します。

■呼び出し手順

```
#include <math.h>
double d, ret;

        ret=tan(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	**正接を求めるラディアン値**

■リターン値

型　：　double
正常：　dの正接値
異常：　―

(8) cosh関数

■機能

浮動小数点数の双曲線余弦を計算します。

■呼び出し手順

```
#include <math.h>
double d, ret;

        ret=cosh(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	**双曲線余弦を求める浮動小数点数**

■リターン値

型　：　double
正常：　dの双曲線余弦値
異常：　―

(9) sinh 関数

■機能

浮動小数点数の双曲線正弦を計算します。

■呼び出し手順

```
#include <math.h>
double d, ret;

        ret=sinh(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	**双曲線正弦を求める浮動小数点数**

■リターン値

型　： double
正常： d の双曲線正弦値
異常： —

(10) tanh 関数

■機能

浮動小数点数の双曲線正接を計算します。

■呼び出し手順

```
#include <math.h>
double d, ret;

        ret=tanh(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	**双曲線正接を求める浮動小数点数**

■リターン値

型　： double
正常： d の双曲線正接値
異常： —

(11) exp 関数

■機能

浮動小数点数の指数関数を計算します。

■呼び出し手順

```
#include <math.h>
double d, ret;

         ret=exp(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	指数関数を求める浮動小数点数

■リターン値

型　：　double
正常：　d の指数関数値
異常：　―

(12) frexp 関数

■機能

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。

■呼び出し手順

```
#include <math.h>
double ret, value;
int *e;

        ret=frexp(value, e);
```

■パラメタ

No.	名前	型	意味
1	value	double	［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
2	e	int 型を指すポインタ	2 のべき乗値を格納する記憶域へのポインタ

■リターン値

型　：　double
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
異常：　—

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。

(13) ldexp 関数

■機能

浮動小数点数と 2 のべき乗の積を計算します。

■呼び出し手順

```
#include <math.h>
double ret, e;
int f;

        ret=ldexp(e, f);
```

■パラメタ

No.	名前	型	意味
1	e	double	2 のべき乗値を求める浮動小数点数
2	f	int	2 のべき乗値

■リターン値

型　：　double
正常：　$e*2^f$ の演算結果の値
異常：　―

(14) log 関数

■機能

浮動小数点数の自然対数を計算します。

■呼び出し手順

```
#include <math.h>
double d, ret;

        ret=log(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	自然対数を求める浮動小数点数

■リターン値

型　：　double
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。

【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。

(15) log10 関数

■機能

浮動小数点数の 10 を底とする対数を計算します。

■呼び出し手順

```
#include <math.h>
double d, ret;

        ret=log10(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	10 を底とする対数を求める浮動小数点数

■リターン値

型　：　double
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。

【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。

(16) modf 関数

■機能

浮動小数点数を整数部分と小数部分に分解します。

■呼び出し手順

```
#include <math.h>
double a, *b, ret;

        ret=modf(a, b);
```

■パラメタ

No.	名前	型	意味
1	a	double	整数部分と小数部分に分解する浮動小数点数
2	b	double 型を指すポインタ	整数部分を格納する記憶域を指すポインタ

■リターン値

型　：　double
正常：　a の小数部分
異常：　―

(17) pow 関数

■機能

浮動小数点数のべき乗を計算します。

■呼び出し手順

```
#include <math.h>
double x, y, ret;

        ret=pow(x, y);
```

■パラメタ

No.	名前	型	意味
1	x	double	べき乗される値
2	y	double	べき乗する値

■リターン値

型　：　double
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。

【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。

(18) sqrt 関数

■機能

浮動小数点数の正の平方根を計算します。

■呼び出し手順

```
#include <math.h>
double d, ret;

        ret=sqrt(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	正の平方根を求める浮動小数点数

■リターン値

型　：　double
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。

【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。

(19) ceil 関数

■機能

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。

■呼び出し手順

```
#include <math.h>
double d, ret;

         ret=ceil(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	**小数点以下を切り上げる浮動小数点数**

■リターン値

型　：　double
正常：　d の小数点以下を切り上げた整数値
異常：　―

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。

(20) fabs 関数

■機能

浮動小数点数の絶対値を計算します。

■呼び出し手順

```
#include <math.h>
double d, ret;

         ret=fabs(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	**絶対値を求める浮動小数点数**

■リターン値

型　：　double
正常：　d の絶対値
異常：　―

(21) floor 関数

■機能

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。

■呼び出し手順

```
#include <math.h>
double d, ret;

        ret=floor(d);
```

■パラメタ

No.	名前	型	意味
1	d	double	**小数点以下を切り捨てる浮動小数点数**

■リターン値

型　：　double
正常：　d の小数点以下を切り捨てた整数値
異常：　—

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。

(22) fmod 関数

■機能

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。

■呼び出し手順

```
#include <math.h>
double x, y, ret;

        ret=fmod(x, y);
```

■パラメタ

No.	名前	型	意味
1	x	double	被除数
2	y	double	除数

■リターン値

型 ： double
正常： 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
異常： ―

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。

6.9 ＜mathf.h＞

■機能概要

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。

■定義名一覧

定義名	種類	説明
EDOM	マクロ名	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 に設定する値を示しています。
ERANGE	マクロ名	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 に設定する値を示しています。
HUGE_VAL	マクロ名	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を示しています。
acosf	関数	浮動小数点数の逆余弦を計算します。
asinf	関数	浮動小数点数の逆正弦を計算します。
atanf	関数	浮動小数点数の逆正接を計算します。
atan2f	関数	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	関数	浮動小数点数のラディアン値の余弦を計算します。
sinf	関数	浮動小数点数のラディアン値の正弦を計算します。
tanf	関数	浮動小数点数のラディアン値の正接を計算します。
coshf	関数	浮動小数点数の双曲線余弦を計算します。
sinhf	関数	浮動小数点数の双曲線正弦を計算します。
tanhf	関数	浮動小数点数の双曲線正接を計算します。
expf	関数	浮動小数点数の指数関数を計算します。
frexpf	関数	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	関数	浮動小数点数と 2 のべき乗の乗算を計算します。
logf	関数	浮動小数点数の自然対数を計算します。
log10f	関数	浮動小数点数の 10 を底とする対数を計算します。
modff	関数	浮動小数点数を整数部分と小数部分に分解します。
powf	関数	浮動小数点数のべき乗を計算します。
sqrtf	関数	浮動小数点数の正の平方根を計算します。
ceilf	関数	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	関数	浮動小数点数の絶対値を計算します。
floorf	関数	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	関数	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。

上記のマクロ名はすべて処理系定義です。

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

No.	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「付録 A.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。

(1) acosf 関数

■機能

浮動小数点数の逆余弦を計算します。

■呼び出し手順

```
#include <mathf.h>
float f, ret;

        ret=acosf(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	逆余弦を求める浮動小数点数

■リターン値

型 ： float
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。

acosf 関数のリターン値の範囲は[0, π]です。

【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。

(2) asinf 関数

■機能

浮動小数点数の逆正弦を計算します。

■呼び出し手順

```
#include <mathf.h>
float f, ret;

        ret=asinf(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	逆正弦を求める浮動小数点数

■リターン値

型　： float
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。

asinf 関数のリターン値の範囲は[－π／2, π／2]です。

【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。

(3) atanf 関数

■機能

浮動小数点数の逆正接を計算します。

■呼び出し手順

```
#include <mathf.h>
float f, ret;

        ret=atanf(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	逆正接を求める浮動小数点数

■リターン値

型　： float
正常： f の逆正接値
異常： ―

atanf 関数のリターン値の範囲は（－π／2, π／2）です。

(4) atan2f 関数

■機能

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。

■呼び出し手順

```
#include <mathf.h>
float x, y, ret;

          ret=atan2f(y, x);
```

■パラメタ

No.	名前	型	意味
1	x	float	除数
2	y	float	被除数

■リターン値

型 ： float
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。

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 6.4 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。

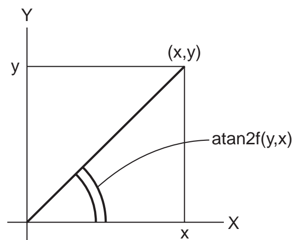


図 6.4 atan2f 関数の意味

【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。

(5) cosf 関数

■機能

浮動小数点数のラディアン値の余弦を計算します。

■呼び出し手順

```
#include <mathf.h>
float f, ret;

        ret=cosf(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	余弦を求めるラディアン値

■リターン値

型 ： float
正常： f の余弦値
異常： —

(6) sinf 関数

■機能

浮動小数点数のラディアン値の正弦を計算します。

■呼び出し手順

```
#include <mathf.h>
float f, ret;

        ret=sinf(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	正弦を求めるラディアン値

■リターン値

型 ： float
正常： f の正弦値
異常： —

(7) tanf 関数

■機能

浮動小数点数のラディアン値の正接を計算します。

■呼び出し手順

```
#include <mathf.h>
float f, ret;

         ret=tanf(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	**正接を求めるラディアン値**

■リターン値

型　：　float
正常：　f の正接値
異常：　―

(8) coshf 関数

■機能

浮動小数点数の双曲線余弦を計算します。

■呼び出し手順

```
#include <mathf.h>
float f, ret;

         ret=coshf(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	**双曲線余弦を求める浮動小数点数**

■リターン値

型　：　float
正常：　f の双曲線余弦値
異常：　―

(9) sinhf 関数

■機能

浮動小数点数の双曲線正弦を計算します。

■呼び出し手順

```
#include <mathf.h>
float f, ret;

         ret=sinhf(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	**双曲線正弦を求める浮動小数点数**

■リターン値

型 ： float
正常： f の双曲線正弦値
異常： —

(10) tanhf 関数

■機能

浮動小数点数の双曲線正接を計算します。

■呼び出し手順

```
#include <mathf.h>
float f, ret;

         ret=tanhf(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	**双曲線正接を求める浮動小数点数**

■リターン値

型 ： float
正常： f の双曲線正接値
異常： —

(11) expf 関数

■機能

浮動小数点数の指数関数を計算します。

■呼び出し手順

```
#include <mathf.h>
float f, ret;

        ret=expf(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	指数関数を求める浮動小数点数

■リターン値

型 ： float
正常： f の指数関数値
異常： —

(12) frexpf 関数

■機能

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。

■呼び出し手順

```
#include <mathf.h>
float ret, value;
int *e;

         ret=frexpf(value, e);
```

■パラメタ

No.	名前	型	意味
1	value	float	［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
2	e	int 型を指すポインタ	2 のべき乗値を格納する記憶域へのポインタ

■リターン値

型　：　float
正常：　value が 0.0 の時：0.0
　　　　value が 0.0 でない時：$ret*2^{\text{e の指している領域の値}}$=value で定義される ret の値
異常：　—

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。

(13) ldexpf 関数

■機能

浮動小数点数と 2 のべき乗の積を計算します。

■呼び出し手順

```
#include <mathf.h>
float ret, e;
int f;

          ret=ldexpf(e, f);
```

■パラメタ

No.	名前	型	意味
1	e	float	2 のべき乗値を求める浮動小数点数
2	f	int	2 のべき乗値

■リターン値

型 ： float
正常： $e*2^f$ の演算結果の値
異常： —

(14) logf 関数

■機能

浮動小数点数の自然対数を計算します。

■呼び出し手順

```
#include <mathf.h>
float f, ret;

          ret=logf(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	自然対数を求める浮動小数点数

■リターン値

型 ： float
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。
【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。

(15) log10f 関数

■機能

浮動小数点数の 10 を底とする対数を計算します。

■呼び出し手順

```
#include <mathf.h>
float f, ret;

        ret=log10f(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	10 を底とする対数を求める浮動小数点数

■リターン値

型　：　float
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。
【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。

(16) modff 関数

■機能

浮動小数点数を整数部分と小数部分に分解します。

■呼び出し手順

```
#include <mathf.h>
float a, *b, ret;

        ret=modff(a, b);
```

■パラメタ

No.	名前	型	意味
1	a	float	整数部分と小数部分に分解する浮動小数点数
2	b	float 型を指すポインタ	整数部分を格納する記憶域を指すポインタ

■リターン値

型　：　float
正常：　a の小数部分
異常：　—

(17) powf 関数

■機能

浮動小数点数のべき乗を計算します。

■呼び出し手順

```
#include <mathf.h>
float x, y, ret;

        ret=powf(x, y);
```

■パラメタ

No.	名前	型	意味
1	x	float	**べき乗される値**
2	y	float	**べき乗する値**

■リターン値

型 ： float
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。
【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。

(18) sqrtf 関数

■機能

浮動小数点数の正の平方根を計算します。

■呼び出し手順

```
#include <mathf.h>
float d, ret;

        ret=sqrtf(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	**正の平方根を求める浮動小数点数**

■リターン値

型 ： float
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。
【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。

(19) ceilf 関数

■機能

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。

■呼び出し手順

```
#include <mathf.h>
float f, ret;

         ret=ceilf(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	小数点以下を切り上げる浮動小数点数

■リターン値

型　：　float
正常：　f の小数点以下を切り上げた整数値
異常：　―

ceilf 関数は f の値より大きいか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。

(20) fabsf 関数

■機能

浮動小数点数の絶対値を計算します。

■呼び出し手順

```
#include <mathf.h>
float f, ret;

         ret=fabsf(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	絶対値を求める浮動小数点数

■リターン値

型　：　float
正常：　f の絶対値
異常：　―

(21) floorf関数

■機能

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。

■呼び出し手順

```
#include <mathf.h>
float f, ret;

        ret=floorf(f);
```

■パラメタ

No.	名前	型	意味
1	f	float	小数点以下を切り捨てる浮動小数点数

■リターン値

型　：　float
正常：　fの小数点以下を切り捨てた整数値
異常：　―

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。

(22) fmodf 関数

■機能

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。

■呼び出し手順

```
#include <mathf.h>
float x, y, ret;

          ret=fmodf(x, y);
```

■パラメタ

No.	名前	型	意味
1	x	float	**被除数**
2	y	float	**除数**

■リターン値

型　：　float
正常：　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
異常：　—

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。

6.10 ＜setjmp.h＞

■機能概要

関数間の制御の移動をサポートします。

■定義名一覧

定義名	種類	説明
jmp_buf	**マクロ名**	**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を示しています。**
setjmp	**関数**	**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	**関数**	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。**

上記のマクロ名は、処理系定義です。

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。

■例

```
 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。

(1) setjmp 関数

■機能

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。

■呼び出し手順

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

        ret=setjmp(env);
```

■パラメタ

No.	名前	型	意味
1	env	jmp_buf	実行環境を退避する記憶域へのポインタ

■リターン値

型　：　int
正常：　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
異常：　—

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。

【注意】

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。

(2) longjmp 関数

■機能

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。

■呼び出し手順

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

        longjmp(env, ret);
```

■パラメタ

No.	名前	型	意味
1	env	jmp_buf	実行環境を退避した記憶域へのポインタ
2	ret	int	setjmp 関数へのリターンコード

■リターン値

型　：　void
正常：　—
異常：　—

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。

【注意】

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。

6.11 ＜stdarg.h＞

■機能概要

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。

■定義名一覧

定義名	種類	説明
va_list	マクロ名	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を示しています。
va_start	マクロ	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	マクロ	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	マクロ	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。

上記のマクロ名はすべて処理系定義です。

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。

■例

```
 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);
      }
```

【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）。

(1) va_start マクロ

■機能

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。

■呼び出し手順

```
#include <stdarg.h>
va_list ap;

        va_start(ap, parmN);
```

■パラメータ

No.	名前	型	意味
1	ap	va_list	可変個のパラメタにアクセスするための変数
2	parmN	parmN の型	最右端の引数の識別子

■リターン値

型 ： void
正常： —
異常： —

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。

【注意】

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。

(2) va_arg マクロ

■機能

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。

■呼び出し手順

```
#include <stdarg.h>
va_list ap;
type     ret;


         ret=va_arg(ap, type);
```

■パラメータ

No.	名前	型	意味
1	ap	va_list	可変個のパラメタにアクセスするための変数
2	type	型名	アクセスするパラメタの型

■リターン値

型　：　パラメタの第2引数で指定した型 type
正常：　パラメタの値
異常：　—

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。

【注意】

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。

(3) va_end マクロ

■機能

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。

■呼び出し手順

```
#include <stdarg.h>
va_list ap;


        va_end(ap);
```

■パラメータ

No.	名前	型	意味
1	ap	va_list	**可変個の引数を参照するための変数**

■リターン値

型 ： —
正常： —
異常： —

【注意】

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。

6.12 ＜stdio.h＞

■機能概要

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。

また、<no_float.h>は浮動小数点変換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 サイズを削減できます。

■定義名一覧

定義名	種類	説明
FILE	マクロ名	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を示しています。
_IOFBF	マクロ名	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	マクロ名	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	マクロ名	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	マクロ名	入出力処理において必要とするバッファの大きさを示しています。
EOF	マクロ名	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	マクロ名	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を示しています。
SEEK_CUR	マクロ名	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	マクロ名	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	マクロ名	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	マクロ名	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を示しています。
TMP_MAX	マクロ名	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を示します。
stderr	マクロ名	標準エラーファイルに対するファイルポインタを示します。
stdin	マクロ名	標準入力ファイルに対するファイルポインタを示します。
stdout	マクロ名	標準出力ファイルに対するファイルポインタを示します。
fclose	関数	ストリーム入出力用ファイルをクローズします。
fflush	関数	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	関数	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	関数	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	関数	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	関数	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	関数	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	関数	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	関数	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	関数	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。

定義名	種類	説明
sprintf	関数	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	関数	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	関数	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	関数	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	関数	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	関数	ストリーム入出力用ファイルから 1 文字入力します。
fgets	関数	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	関数	ストリーム入出力用ファイルへ 1 文字出力します。
fputs	関数	ストリーム入出力用ファイルへ文字列を出力します。
getc	マクロ名	ストリーム入出力用ファイルから 1 文字入力します。
getchar	マクロ名	標準入力ファイルから 1 文字入力します。
gets	関数	標準入力ファイルから文字列を入力します。
putc	マクロ名	ストリーム入出力用ファイルへ 1 文字出力します。
putchar	マクロ名	標準出力ファイルへ 1 文字出力します。
puts	関数	標準出力ファイルへ文字列を出力します。
ungetc	関数	ストリーム入出力用ファイルへ 1 文字をもどします。
fread	関数	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	関数	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	関数	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	関数	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	関数	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	関数	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	関数	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	関数	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	関数	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。

上記のマクロ名は全て処理系定義です。

■処理系定義仕様

No.	項目	コンパイラの仕様
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 6.6 に示します。

表 6.6　無限大および非数の表示形式

No.	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。

■例

```
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。

(1) fclose 関数

■機能

ストリーム入出力用ファイルをクローズします。

■呼び出し手順

```
#include <stdio.h>
FILE  *fp;
int  ret;

         ret=fclose(fp);
```

■パラメタ

No.	名前	型	意味
1	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型　：　int
正常：　0
異常：　0 以外

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。

(2) fflush 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
FILE  *fp;
int   ret;

          ret=fflush(fp);
```

■パラメタ

No.	名前	型	意味
1	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型 ： int
正常： 0
異常： 0 以外

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。

(3) fopen 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
FILE  *ret;
const char  *fname,  *mode;

         ret=fopen(fname,  mode);
```

■パラメタ

No.	名前	型	意味
1	fname	const char 型を指すポインタ	ファイル名を示す文字列へのポインタ
2	mode	const char 型を指すポイン	ファイルアクセスモードを示す文字列へのポインタ

■リターン値

型　：　FILE 型へのポインタ
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

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。

(4) freopen 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
const char  *fname, *mode;
FILE  *ret, *fp;

        ret=freopen(fname, mode, fp);
```

■パラメタ

No.	名前	型	意味
1	fname	const char 型を指すポインタ	新しいファイル名を示す文字列へのポインタ
2	mode	const char 型を指すポインタ	ファイルアクセスモードを示す文字列へのポインタ
3	fp	FILE 型を指すポインタ	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

■リターン値

型　：　FILE 型へのポインタ
正常：　fp
異常：　NULL

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。

(5)　setbuf 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
FILE  *fp;
char buf [BUFSIZ];

         setbuf(fp, buf);
```

■パラメタ

No.	名前	型	意味
1	fp	FILE 型を指すポインタ	ファイルポインタ
2	buf	char 型の配列を指すポインタ	バッファ領域へのポインタ

■リターン値

型　：　void
正常：　__
異常：　__

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。

(6) setvbuf 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
FILE  *fp;
char  *buf;
int   type, ret;
size_t size;

         ret=setvbuf(fp, buf, type, size);
```

■パラメタ

No.	名前	型	意味
1	fp	FILE 型を指すポインタ	ファイルポインタ
2	buf	char 型を指すポインタ	バッファ領域へのポインタ
3	type	int	バッファの管理方式
4	size	size_t	バッファ領域の大きさ

■リターン値

型　：　int
正常：　0
異常：　0 以外

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。

【注意】

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。

(7) fprintf 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
FILE *fp;
const char *control;
int  ret;

        ret=fprintf(fp, control 〔,arg〕…);
```

■パラメタ

No.	名前	型	意味
1	fp	FILE 型を指すポインタ	ファイルポインタ
2	control	const char 型を指すポインタ	書式を示す文字列へのポインタ
3	arg, …	特に規定せず	書式に従って出力されるデータの並び

■リターン値

型 ： int
正常： 変換し出力した文字数
異常： 負の値

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。

（a） 通常の文字
（b）に示す%で始まる指定以外の文字はそのまま出力されます。

（b） 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。

```
% [フラグ…] { [*]          }
            { [フィールド幅] }
            [ . { [*]   } ] [パラメタのサイズ指定] 変換文字
                { [精度] }
```

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**6.7に示します。

表 6.7 フラグの種類と意味

項番	種類	意味
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 6.9 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6.9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6.9参照）**

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**6.8にサイズ指定の種類とその意味を示します。

表 6.8 パラメタのサイズ指定の種類とその意味

項番	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6.9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 6.9　変換文字と変換の方式

項番	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。
	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	

項番	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
9	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。
10	G			double 型	
11	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。
12	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ふくまれません。）	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ふくまれません。）
13	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。
14	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	
15	%	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。

(8) fscanf 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
FILE  *fp;
const char  *control;
int ret;

         ret=fscanf(fp, control〔,ptr〕…);
```

■パラメタ

No.	名前	型	意味
1	fp	FILE 型を指すポインタ	ファイルポインタ
2	control	const char 型を指すポインタ	書式を示す文字列へのポインタ
3	ptr	データ型を指すポインタ	入力したデータを格納する記憶域へのポインタ

■リターン値

型 ： int
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

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。

(a) 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。

(b) 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 6.11 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 6.10 にサイズ指定の種類とその意味を示します。

表 6.10 変換後のデータのサイズ指定の種類とその意味

項番	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6.11　変換文字と変換の内容

項番	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。 x 変換と X 変換に意味の違いはありません。	整数型
6	X	X 変換		
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** 6.11 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。

(9)　printf 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
const char *control;
int ret;

          ret=printf(control〔 ,arg〕…);
```

■パラメタ

No.	名前	型	意味
1	control	const char 型を指すポインタ	書式を示す文字列へのポインタ
2	arg	書式に従った型	書式に従って出力されるデータ

■リターン値

型　：　int
正常：　変換し出力した文字数
異常：　負の値

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「6.12 (7)　fprintf 関数」を参照してください。

(10) scanf 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
const char *control;
int ret;

        ret=scanf(control〔 ,ptr〕…);
```

■パラメタ

No.	名前	型	意味
1	control	const char 型を指すポインタ	書式を示す文字列へのポインタ
2	ptr	任意のデータを指すポインタ	入力変換したデータを格納する記憶域へのポインタ

■リターン値

型　：　int
正常：　入力変換に成功したデータの個数
異常：　EOF

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「6.12 (8)fscanf 関数」を参照してください。

[注意]　%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。

(11) sprintf 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
char *s;
const char *control;
int ret;
        ret=sprintf(s, control〔, arg〕…);
```

■パラメタ

No.	名前	型	意味
1	s	char 型を指すポインタ	データを出力する記憶域へのポインタ
2	control	const char 型を指すポインタ	書式を示す文字列へのポインタ
3	arg	書式に従った型	書式に従って出力されるデータ

■リターン値

型　： int
正常： 変換した文字数
異常： ―

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「6.12 (7)　fprintf 関数」を参照してください。

(12) sscanf 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
const char *s, *control;
int ret;

        ret=sscanf(s, control〔, ptr〕…);
```

■パラメタ

No.	名前	型	意味
1	s	const char 型を指すポインタ	入力するデータがある記憶域
2	control	const char 型を指すポインタ	書式を示す文字列へのポインタ
3	ptr	データ型を指すポインタ	入力変換したデータを格納する記憶域へのポインタ

■リターン値

型 ： int
正常： 入力変換に成功したデータの個数
異常： EOF

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「6.12 (8) fscanf 関数」を参照してください。

(13) vfprintf関数

■機能

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。

■呼び出し手順

```
#include <stdarg.h>
#include <stdio.h>
FILE  *fp;
const char *control;
va_list arg;
int      ret;
         ret=vfprintf(fp, control, arg);
```

■パラメタ

No.	名前	型	意味
1	fp	FILE型を指すポインタ	ファイルポインタ
2	control	const char型を指すポインタ	書式を示す文字列へのポインタ
3	arg	va_list	引数リスト

■リターン値

型　：　int
正常：　変換し出力した文字数
異常：　負の値

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「6.12 (7)　fprintf関数」を参照してください。

【注意】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。

(14) vprintf 関数

■機能

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。

■呼び出し手順

```
#include <stdarg.h>
#include <stdio.h>
const char *control;
va_list    arg;
int          ret;
          ret=vprintf(control, arg);
```

■パラメタ

No.	名前	型	意味
1	control	const char 型を指すポインタ	書式を示す文字列へのポインタ
2	arg	va_list	引数リスト

■リターン値

型 ： int
正常： 変換し出力した文字数
異常： 負の値

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「6.12 (7) fprintf 関数」を参照してください。

【注意】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。

(15) vsprintf 関数

■機能

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。

■呼び出し手順

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

■パラメタ

No.	名前	型	意味
1	s	char 型を指すポインタ	データを出力する記憶域へのポインタ
2	control	const char 型を指すポインタ	書式を示す文字列へのポインタ
3	arg	va_list	引数リスト

■リターン値

型 ： int
正常： 変換した文字数
異常： 負の数

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「6.12 (7) fprintf 関数」を参照してください。

【注意】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。

(16) fgetc 関数

■機能

ストリーム入出力用ファイルから 1 文字入力します。

■呼び出し手順

```
#include <stdio.h>
FILE  *fp;
int   ret;

        ret=fgetc(fp);
```

■パラメタ

No.	名前	型	意味
1	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型　：　int
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

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。

【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。

(17)　　fgets 関数

■機能

ストリーム入出力用ファイルから文字列を入力します。

■呼び出し手順

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

■パラメタ

No.	名前	型	意味
1	s	char 型を指すポインタ	文字列を入力する記憶域へのポインタ
2	n	int	文字列を入力する記憶域のバイト数
3	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型　：　char 型へのポインタ
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

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。

【注意】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。

(18) fputc 関数

■機能

ストリーム入出力用ファイルへ 1 文字出力します。

■呼び出し手順

```
#include <stdio.h>
FILE  *fp;
int   c, ret;

        ret=fputc(c, fp)
```

■パラメタ

No.	名前	型	意味
1	c	int	出力する文字
2	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型　： int
正常： 出力した文字
異常： EOF

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。

【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。

(19) fputs 関数

■機能

ストリーム入出力用ファイルへ文字列を出力します。

■呼び出し手順

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

■パラメタ

No.	名前	型	意味
1	s	const char 型を指すポインタ	出力する文字列へのポインタ
2	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型 ： int
正常： 0
異常： 0 以外

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。

(20)　getc 関数

■機能

ストリーム入出力用ファイルから 1 文字入力します。

■呼び出し手順

```
#include <stdio.h>
FILE  *fp;
int   ret;

        ret=getc(fp);
```

■パラメタ

No.	名前	型	意味
1	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型　：　int
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

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。

【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。

(21) getchar関数

■機能

標準入力ファイル（stdin）から、1文字入力します。

■呼び出し手順

```
#include <stdio.h>
int ret ;

        ret=getchar( );
```

■パラメタ

No.	名前	型	意味
—	—	—	—

■リターン値

型　：　int
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

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。

【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。

(22) gets 関数

■機能

標準入力ファイル（stdin）から文字列を入力します。

■呼び出し手順

```
#include <stdio.h>
char  *ret, *s;

        ret=gets(s);
```

■パラメタ

No.	名前	型	意味
1	s	char 型を指すポインタ	文字列を入力する記憶域へのポインタ

■リターン値

型　：　char 型へのポインタ
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

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。

【注意】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。

(23) putc 関数

■機能

ストリーム入出力用ファイルへ 1 文字出力します。

■呼び出し手順

```
#include <stdio.h>
FILE  *fp;
int   c, ret;

          ret=putc(c, fp);
```

■パラメタ

No.	名前	型	意味
1	c	int	出力する文字
2	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型　： int
正常： 出力した文字
異常： EOF

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。

【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。

(24) putchar 関数

■機能

標準出力ファイル（stdout）へ 1 文字出力します。

■呼び出し手順

```
#include <stdio.h>
int   c, ret;

         ret=putchar(c);
```

■パラメタ

No.	名前	型	意味
1	c	int	出力する文字

■リターン値

型　：　int
正常：　出力した文字
異常：　EOF

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。

【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。

(25)　puts 関数

■機能

標準出力ファイル（stdout）へ文字列を出力します。

■呼び出し手順

```
#include <stdio.h>
const  char *s;
int   ret;

         ret=puts(s);
```

■パラメタ

No.	名前	型	意味
1	s	const char 型を指すポインタ	出力する文字列へのポインタ

■リターン値

型　：　int
正常：　0
異常：　0 以外

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。

(26) ungetc 関数

■機能

ストリーム入出力用ファイルへ 1 文字をもどします。

■呼び出し手順

```
#include <stdio.h>
int    c, ret;
FILE   *fp;

         ret=ungetc(c, fp);
```

■パラメタ

No.	名前	型	意味
1	c	int	もどす文字
2	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型 ： int
正常： もどした文字
異常： EOF

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もどします。また、ここでもどされた文字は、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、通常もどした文字 c を返しますが、エラー発生の時は、EOF を返します。

【注意】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もどされますが、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。

(27) fread 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
void  *ptr;
size_t  size;
size_t  n, ret;
FILE  *fp;

        ret=fread(ptr, size, n, fp);
```

■パラメタ

No.	名前	型	意味
1	ptr	void 型を指すポインタ	データを入力する記憶域へのポインタ
2	size	size_t	1 メンバのバイト数
3	n	size_t	入力するメンバの数
4	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型　：　size_t
正常：　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
異常：　－

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。

【注意】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。

(28) fwrite 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
const void  *ptr;
size_t  size;
size_t  n, ret;
FILE  *fp  ;


        ret=fwrite(ptr, size, n, fp);
```

■パラメタ

No.	名前	型	意味
1	ptr	const void 型を指すポインタ	出力するデータを格納している記憶域へのポインタ
2	size	size_t	1 メンバのバイト数
3	n	size_t	出力するメンバの数
4	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型 ： size_t
正常： 出力に成功したメンバ数
異常： ―

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。

【注意】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。

(29)　fseek 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
FILE  *fp;
long  offset;
int   type, ret;

        ret=fseek(fp, offset, type);
```

■パラメタ

No.	名前	型	意味
1	fp	FILE 型を指すポインタ	ファイルポインタ
2	offset	long	オフセットの種類で指定された位置からのオフセット
3	type	int	オフセットの種類

■リターン値

型　：　int
正常：　0
異常：　0 以外

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** 6.12 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 6.12　オフセットの種類

項番	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。

【注意】　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。

(30) ftell 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
FILE  *fp;
long  ret;

         ret=ftell(fp);
```

■パラメタ

No.	名前	型	意味
1	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型 ： long
正常： 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）
異常： ―

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。

【注意】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。

(31) rewind 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
FILE  *fp;

      rewind(fp);
```

■パラメタ

No.	名前	型	意味
1	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型 ： void
正常： ―
異常： ―

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。

【注意】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。

(32) clearerr 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
FILE  *fp;

      clearerr(fp);
```

■パラメタ

No.	名前	型	意味
1	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型　：　void
正常：　－
異常：　－

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。

(33) feof 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
FILE  *fp ;
int   ret ;

      ret=feof(fp);
```

■パラメタ

No.	名前	型	意味
1	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型　：　int
正常：　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
異常：　__

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 をかえします。

(34) ferror 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
FILE  *fp;
int   ret;

      ret=ferror(fp);
```

■パラメタ

No.	名前	型	意味
1	fp	FILE 型を指すポインタ	ファイルポインタ

■リターン値

型　：　int
正常：　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
異常：　－

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。

(35) perror 関数

■機能

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。

■呼び出し手順

```
#include <stdio.h>
const char  *s;

     perror(s);
```

■パラメタ

No.	名前	型	意味
1	s	const char 型を指すポインタ	エラーメッセージへのポインタ

■リターン値

型 ： void
正常： －
異常： －

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。

6.13 ＜no_float.h＞

■簡易入出力関数のサポート

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。

(1) 対象関数

fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf

(2) 使用方法

#include <stdio.h>の指定行の前に、#include <no_float.h>をマクロ宣言します。

■例

```
#include <no_float.h>
#include <stdio.h>
void main(void)
{
  printf("Hello¥n");
}
```

■注意

#include <no_float.h>を指定したときに、上記対象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。

6.14 <stdlib.h>

■機能概要

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。

■定義名一覧

定義名	種類	説明
onexit_t	マクロ名	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を示しています。
div_t	マクロ名	div 関数のリターン値の構造体の型を示しています。
ldiv_t	マクロ名	ldiv 関数のリターン値の構造体の型を示しています。
RAND_MAX	マクロ名	rand 関数において生成する擬似乱数整数の最大値を示しています。
atof	関数	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	関数	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	関数	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	関数	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	関数	数を表現する文字列を long 型の整数値に変換します。
rand	関数	0 から RAND_MAX の間の擬似乱数整数を生成します。
srand	関数	rand 関数で生成する擬似乱数列の初期値を設定します。
calloc	関数	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	関数	指定された記憶域を解放します。
malloc	関数	記憶域を割り当てます。
realloc	関数	記憶域の大きさを指定された大きさに変更します。
bsearch	関数	2 分割検索を行います。
qsort	関数	ソートを行います。
abs	関数	int 型整数の絶対値を計算します。
div	関数	int 型整数の除算の商と余りを計算します。
labs	関数	long 型整数の絶対値を計算します。
ldiv	関数	long 型整数の除算の商と余りを計算します。

上記のマクロ名は、処理系定義です。

(1) atof 関数

■機能

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。

■呼び出し手順

```
#include <stdlib.h>
const char *nptr;
double  ret;

        ret=atof(nptr);
```

■パラメタ

No.	名前	型	意味
1	nptr	const char 型を指すポインタ	変換する数を表現する文字列のポインタ

■リターン値

型　：　double
正常：　変換された double 型の浮動小数点数値
異常：　—

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。

【注意】

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。

(2) atoi 関数

■機能

10 進数を表現する文字列を、int 型の整数値に変換します。

■呼び出し手順

```
#include <stdlib.h>
const char *nptr;
int  ret;

        ret=atoi(nptr);
```

■パラメタ

No.	名前	型	意味
1	nptr	const char 型を指すポインタ	変換する数を表現する文字列のポインタ

■リターン値

型　：　int
正常：　変換された int 型の整数値
異常：　—

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。

【注意】

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。

(3) atol 関数

■機能

10 進数を表現する文字列を、long 型の整数値に変換します。

■呼び出し手順

```
#include <stdlib.h>
const char *nptr ;
long ret;

        ret=atol(nptr);
```

■パラメタ

No.	名前	型	意味
1	nptr	const char 型を指すポインタ	変換する数を表現する文字列のポインタ

■リターン値

型　：　long
正常：　変換された long 型の整数値
異常：　―

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。

【注意】

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。

(4) strtod 関数

■機能

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。

■呼び出し手順

```
#include <stdlib.h>
const char *nptr;
char      **endptr;
double     ret;

          ret=strtod(nptr, endptr);
```

■パラメタ

No.	名前	型	意味
1	nptr	const char 型を指すポインタ	変換する数を表現する文字列へのポインタ
2	endptr	char 型を指すポインタへのポインタ	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

■リターン値

型 ： double
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

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「C 言語仕様」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。

【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。

(5) strtol 関数

■機能

数を表現する文字列を long 型の整数値に変換します。

■呼び出し手順

```
#include <stdlib.h>
long  ret;
const  char *nptr;
char   **endptr;
int    base;

         ret=strtol(nptr, endptr, base);
```

■パラメタ

No.	名前	型	意味
1	nptr	const char 型を指すポインタ	変換する数を表現する文字列へのポインタ
2	endptr	char 型を指すポインタへのポインタ	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
3	base	int	変換の基数（0 又は 2〜36）

■リターン値

型　：　long
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

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、**「C 言語仕様」**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。

【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。

(6) rand 関数

■機能

0 から RAND_MAX の間の擬似乱数整数を生成します。

■呼び出し手順

```
#include <stdlib.h>
int ret;

        ret=rand(  );
```

■パラメタ

No.	名前	型	意味
—	—	—	—

■リターン値

型 ： int
正常： 擬似乱数整数値
異常： —

(7) srand 関数

■機能

rand 関数で生成する擬似乱数列の初期値を設定します。

■呼び出し手順

```
#include <stdlib.h>
unsigned int  seed;

        srand(seed);
```

■パラメタ

No.	名前	型	意味
1	seed	unsigned int	**擬似乱数列生成の初期値**

■リターン値

型 ： void
正常： —
異常： —

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。

【注意】

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。

(8) calloc 関数

■機能

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。

■呼び出し手順

```
#include <stdlib.h>
size_t  nelem, elsize;
void  *ret;

        ret=calloc(nelem, elsize);
```

■パラメタ

No.	名前	型	意味
1	nelem	size_t	**要素の数**
2	elsize	size_t	**ひとつの要素の占めるバイト数**

■リターン値

型　：　void 型へのポインタ
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

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。

(9) free 関数

■機能

指定された記憶域を解放します。

■呼び出し手順

```
#include <stdlib.h>
void  *ptr;

        free(ptr);
```

■パラメタ

No.	名前	型	意味
1	ptr	void 型を指すポインタ	解放する記憶域のアドレス

■リターン値

型　：　void
正常：　__
異常：　__

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。

【注意】

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。

(10) malloc 関数

■機能

記憶域を割り当てます。

■呼び出し手順

```
#include <stdlib.h>
size_t  size;
void  *ret;

        ret=malloc(size);
```

■パラメタ

No.	名前	型	意味
1	size	size_t	割り当てる記憶域のバイト数

■リターン値

型 ： void 型へのポインタ
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

size で示されるバイトの分だけ記憶域を割り当てます。

(11) realloc 関数

■機能

記憶域の大きさを指定された大きさに変更します。

■呼び出し手順

```
#include <stdlib.h>
size_t  size;
void  *ptr, *ret;

       ret=realloc(ptr, size);
```

■パラメタ

No.	名前	型	意味
1	ptr	void 型を指すポインタ	変更する記憶域の先頭アドレス
2	size	size_t	変更後の記憶域のバイト数

■リターン値

型　：　void 型へのポインタ
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

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。

【注意】

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。

(12) bsearch 関数

■機能

二分割検索を行います。

■呼び出し手順

```
#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);
```

■パラメタ

No.	名前	型	意味
1	key	const void 型を指すポインタ	検索するデータへのポインタ
2	base	const void を指すポインタ	検索対象となるテーブルへのポインタ
3	nmemb	size_t	検索対象のメンバの数
4	size	size_t	検索対象のメンバのバイト数
5	compar	int 型を返す関数へのポインタ	比較を行う関数へのポインタ

■リターン値

型　：　void 型へのポインタ
正常：　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
異常：　－

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。

【注意】

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。

(13) qsort 関数

■機能

ソートを行います。

■呼び出し手順

```
#include <stdlib.h>
const  void  *base;
size_t  nmemb,  size;
int (*compar)(const void *, const  void *);

              qsort(base, nmemb, size, compar);
```

■パラメタ

No.	名前	型	意味
1	base	const void を指すポインタ	ソート対象となるテーブルへのポインタ
2	nmemb	size_t	ソート対象のメンバの数
3	size	size_t	ソート対象のメンバのバイト数
4	compar	int 型を返す関数へのポインタ	比較を行う関数へのポインタ

■リターン値

型　：　void
正常：　—
異常：　—

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。

(14) abs 関数

■機能

絶対値を求めます。

■呼び出し手順

```
#include <stdlib.h>
int  i,  ret ;

        ret=abs(i);
```

■パラメタ

No.	名前	型	意味
1	i	int	絶対値を求める整数

■リターン値

型　：　int
正常：　i の絶対値
異常：　__

【注意】

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。

(15) div 関数

■機能

int 型整数の除算の商と余りを計算します。

■呼び出し手順

```
#include <stdlib.h>
int  numer,  denom;
div_t  ret;

        ret=div(numer, denom);
```

■パラメタ

No.	名前	型	意味
1	numer	int	被除数
2	denom	int	除数

■リターン値

型　：　div_t
正常：　numer を denom で除算した結果の商と余り。
異常：　__

(16) labs 関数

■機能

long 型整数の絶対値を計算します。

■呼び出し手順

```
#include <stdlib.h>
long  j;
long  ret;
        ret=labs(j);
```

■パラメタ

No.	名前	型	意味
1	j	long	絶対値を求める整数

■リターン値

型 ： long
正常： j の絶対値
異常： ―

【注意】

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。

(17) ldiv 関数

■機能

long 型整数の除算の商と余りを計算します。

■呼び出し手順

```
#include <stdlib.h>
long  numer, denom;
ldiv_t  ret;

         ret=ldiv(numer, denom);
```

■パラメタ

No.	名前	型	意味
1	numer	long	被除数
2	denom	long	除数

■リターン値

型 ： ldiv_t
正常： numer を denom で除算した結果の商と余り。
異常： ―

6.15　<string.h>

■機能概要

文字配列の操作に必要な種々の関数を定義します。

■定義名一覧

定義名	種類	説明
memcpy	関数	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	関数	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	関数	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	関数	文字列の後に、文字列を連結します。
strncat	関数	文字列に文字列を指定した文字数分、連結します。
memcmp	関数	指定された二つの記憶域の比較を行います。
strcmp	関数	指定された二つの文字列を比較します。
strncmp	関数	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	関数	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	関数	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	関数	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	関数	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	関数	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	関数	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	関数	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	関数	指定した文字列をいくつかの字句に切り分けます。
memset	関数	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	関数	エラーメッセージを設定します。
strlen	関数	文字列の長さを計算します。
memmove	関数	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。

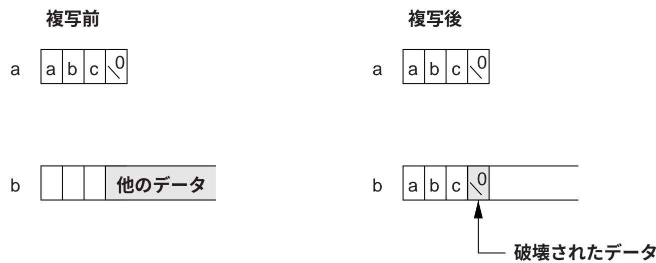
■処理系定義仕様

No.	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	「10.2　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。



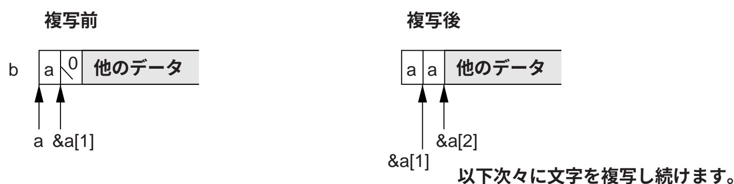
（2） 文字列の複写を行う時、複写元の領域と複写先の領域が重なっていると正しい動作が保証されませんので注意が必要です。

■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。



(1) memcpy関数

■機能

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。

■呼び出し手順

```
#include <string.h>
void    *ret, *s1;
const   void  *s2;
size_t   n;

         ret=memcpy(s1, s2, n);
```

■パラメタ

No.	名前	型	意味
1	s1	void型を指すポインタ	複写先の記憶域へのポインタ
2	s2	const void型を指すポインタ	複写元の記憶域へのポインタ
3	n	size_t	複写する文字数

■リターン値

型 ： void型へのポインタ
正常： s1の値
異常： ―

(2) strcpy関数

■機能

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。

■呼び出し手順

```
#include <string.h>
char    *s1, *ret;
const   char *s2;

          ret=strcpy(s1, s2);
```

■パラメタ

No.	名前	型	意味
1	s1	char型を指すポインタ	複写先の記憶域へのポインタ
2	s2	const char型を指すポインタ	複写元の文字列へのポインタ

■リターン値

型 ： char型へのポインタ
正常： s1の値
異常： ―

(3) strncpy 関数

■機能

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。

■呼び出し手順

```
#include <string.h>
char    *s1, *ret;
const   char *s2;
size_t  n;
          ret=strncpy(s1, s2, n );
```

■パラメタ

No.	名前	型	意味
1	s1	char 型を指すポインタ	複写先の記憶域へのポインタ
2	s2	const char 型を指すポインタ	複写元の文字列へのポインタ
3	n	size_t	複写する文字数

■リターン値

型 ： char 型へのポインタ
正常： s1 の値
異常： ―

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。

【注意】

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。

(4) strcat 関数

■機能

文字列の後に、文字列を連結します。

■呼び出し手順

```
#include <string.h>
char    *s1, *ret;
const   char *s2;

          ret=strcat(s1, s2);
```

■パラメタ

No.	名前	型	意味
1	s1	char 型を指すポインタ	連結される文字列へのポインタ
2	s2	const char 型を指すポインタ	連結する文字列へのポインタ

■リターン値

型 ： char 型へのポインタ
正常： s1 の値
異常： —

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。

(5) strncat 関数

■機能

文字列に文字列を指定した文字数分連結します。

■呼び出し手順

```
#include <string.h>
char    *s1, *ret;
const   char *s2;
size_t  n;

         ret=strncat(s1, s2, n);
```

■パラメタ

No.	名前	型	意味
1	s1	char 型を指すポインタ	連結される文字列へのポインタ
2	s2	const char 型を指すポインタ	連結する文字列へのポインタ
3	n	size_t	連結する文字数

■リターン値

型　：　char 型へのポインタ
正常：　s1 の値
異常：　―

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。

(6) memcmp 関数

■機能

指定された二つの記憶域の内容を比較します。

■呼び出し手順

```
#include <string.h>
const   void  *s1, *s2;
size_t  n;
int     ret;

        ret=memcmp(s1, s2, n);
```

■パラメタ

No.	名前	型	意味
1	s1	const void 型を指すポインタ	比較される記憶域へのポインタ
2	s2	const void 型を指すポインタ	比較する記憶域へのポインタ
3	n	size_t	比較する記憶域の文字数

■リターン値

型 ： int
正常： 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
異常： ―

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。比較するための基準は処理系定義です。

(7) strcmp 関数

■機能

指定された二つの文字列の内容を比較します。

■呼び出し手順

```
#include <string.h>
const   char  *s1, *s2;
int   ret;

        ret=strcmp(s1, s2);
```

■パラメタ

No.	名前	型	意味
1	s1	const char 型を指すポインタ	比較される文字列へのポインタ
2	s2	const char 型を指すポインタ	比較する文字列へのポインタ

■リターン値

型　：　int
正常：　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
異常：　－

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。比較するための基準は処理系定義です。

(8) strncmp 関数

■機能

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。

■呼び出し手順

```
#include <string.h>
const   char *s1, *s2;
size_t  n;
int   ret;

        ret=strncmp(s1, s2, n);
```

■パラメタ

No.	名前	型	意味
1	s1	const char 型を指すポインタ	比較される文字列へのポインタ
2	s2	const char 型を指すポインタ	比較する文字列へのポインタ
3	n	size_t	比較する文字数の最大値

■リターン値

型 ： int
正常： 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
異常： ―

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。比較するための基準は処理系定義です。

(9) memchr 関数

■機能

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。

■呼び出し手順

```
#include <string.h>
const  void   *s;
int   c;
size_t n ;
void  *ret;

        ret=memchr(s, c, n);
```

■パラメタ

No.	名前	型	意味
1	s	const void 型を指すポインタ	検索を行う記憶域へのポインタ
2	c	int	検索する文字
3	n	size_t	検索を行う文字数

■リターン値

型　：　void 型へのポインタ
正常：　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
異常：　—

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。

(10) strchr 関数

■機能

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。

■呼び出し手順

```
#include <string.h>
const  char *s;
int   c;
char   *ret;

         ret=strchr(s, c);
```

■パラメタ

No.	名前	型	意味
1	s	const char 型を指すポインタ	検索を行う文字列へのポインタ
2	c	int	検索する文字

■リターン値

型　：　char 型へのポインタ
正常：　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
異常：　―

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。

【注意】

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。

(11) strcspn 関数

■機能

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。

■呼び出し手順

```
#include <string.h>
const   char  *s1, *s2;
size_t   ret;

         ret=strcspn(s1, s2);
```

■パラメタ

No.	名前	型	意味
1	s1	const char 型を指すポインタ	調べられる文字列へのポインタ
2	s2	const char 型を指すポインタ	s1 を調べるための文字列へのポインタ

■リターン値

型 ： size_t
正常： 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
異常： ―

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。

【注意】

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。

(12) strpbrk 関数

■機能

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。

■呼び出し手順

```
#include <string.h>
const   char  *s1, *s2;
char    *ret;

         ret=strpbrk(s1, s2);
```

■パラメタ

No.	名前	型	意味
1	s1	const char 型を指すポインタ	検索を行う文字列へのポインタ
2	s2	const char 型を指すポインタ	s1 内で検索する文字を示す文字列へのポインタ

■リターン値

型　： char 型へのポインタ
正常： 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
異常： ―

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。

(13) strrchr関数

■機能

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。

■呼び出し手順

```
#include <string.h>
const  char  *s;
int    c;
char  *ret;

         ret=strrchr(s, c) ;
```

■パラメタ

No.	名前	型	意味
1	s	const char 型を指すポインタ	検索を行う文字列へのポインタ
2	c	int	検索する文字

■リターン値

型　：　char 型へのポインタ
正常：　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
異常：　―

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。

【注意】

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。

(14)　strspn 関数

■機能

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。

■呼び出し手順

```
#include <string.h>
const   char  *s1, *s2;
size_t      ret;

         ret=strspn(s1, s2);
```

■パラメタ

No.	名前	型	意味
1	s1	const char 型を指すポインタ	調べられる文字列へのポインタ
2	s2	const char 型を指すポインタ	s1 を調べるための文字列へのポインタ

■リターン値

型　：　size_t
正常：　s1 の先頭から、s2 で指定した文字が続いている文字数
異常：　―

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。

(15) strstr 関数

■機能

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。

■呼び出し手順

```
#include <string.h>
const   char  *s1, *s2;
char    *ret;

         ret=strstr(s1, s2);
```

■パラメタ

No.	名前	型	意味
1	s1	const char 型を指すポインタ	検索を行う文字列へのポインタ
2	s2	const char 型を指すポインタ	検索する文字列へのポインタ

■リターン値

型 ： char 型へのポインタ
正常： 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
異常： —

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。

(16) strtok 関数

■機能

指定した文字列をいくつかの字句に切り分けます。

■呼び出し手順

```
#include <string.h>
char   *s1, *ret;
const   char  *s2;

          ret=strtok(s1, s2);
```

■パラメタ

No.	名前	型	意味
1	s1	char 型を指すポインタ	いくつかの字句に切り分ける文字列へのポインタ
2	s2	const char 型を指すポインタ	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

■リターン値

型　：　char 型へのポインタ
正常：　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
異常：　—

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。

(17) memset 関数

■機能

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。

■呼び出し手順

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

         ret=memset(s, c, n);
```

■パラメタ

No.	名前	型	意味
1	s	vold 型を指すポインタ	文字が設定される記憶域へのポインタ
2	c	int	設定する文字
3	n	size_t	設定する文字数

■リターン値

型　：　vold 型へのポインタ
正常：　s の値
異常：　―

s で指された記憶域に n 文字分、文字 c を設定します。

(18) strerror 関数

■機能

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。

■呼び出し手順

```
#include <string.h>
char   *ret;

int   s;
         ret=strerror(s);
```

■パラメタ

No.	名前	型	意味
1	s	int	エラー番号

■リターン値

型 ： char 型へのポインタ
正常： 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
異常： ―

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。

【注意】

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。

(19) strlen 関数

■機能

文字列の長さを計算します。

■呼び出し手順

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

■パラメタ

No.	名前	型	意味
1	s	const char 型を返すポインタ	長さを求める文字列へのポインタ

■リターン値

型　：　size_t
正常：　文字列の文字数
異常：　—

【注意】

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。

(20) memmove

■機能

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。

■呼び出し手順

```
#include  <string.h>
void  *ret,*s1;
const  void  *s2;
size_t  n;
            ret=memmove(s1,s2,n);
```

■パラメタ

No.	名前	型	意味
1	s1	void 型を指すポインタ	複写先の記憶域へのポインタ
2	s2	const void 型を指すポインタ	複写元の記憶域へのポインタ
3	n	size_t	複写する文字数

■リターン値

型　：　void 型へのポインタ
正常：　s1 の値
異常：　—

7. EC++クラスライブラリ

7.1 ライブラリの概要

本章では、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。**「7.1 ライブラリの概要」**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。以下の節では、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。

(1) ライブラリの種類

表 7.1 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 7.1 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

No	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>

7.2 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。

(1) <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。

(2) <streambuf>

ストリームバッファに対する関数を定義します。

(3) <istream>

入力ストリームからの入力関数を定義します。

(4) <ostream>

出力ストリームへの出力関数を定義します。

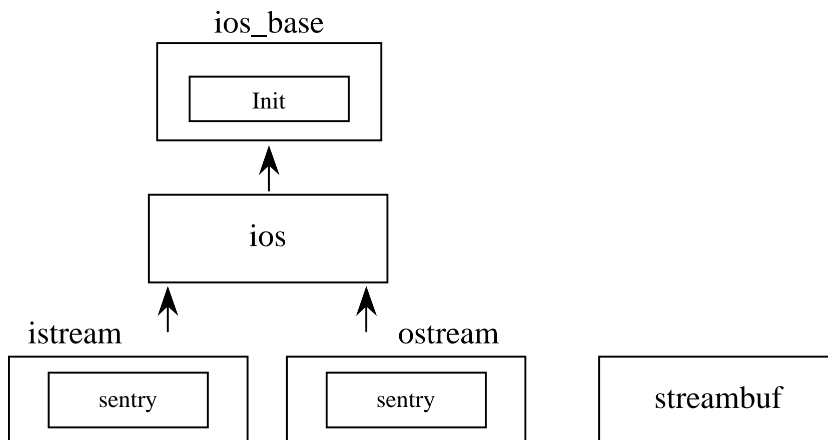
(5) <iostream>

入出力関数を定義します。

(6) <iomanip>

引数を持つマニピュレータを定義します。

また、これらのクラスの階層図は次のようになります。



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マクロ名を示します。

■定義名一覧

定義名	種類	説明
streamoff	マクロ	long 型で定義された型です。
streamsize	マクロ	size_t 型で定義された型です。
int_type	マクロ	int 型で定義された型です。
pos_type	マクロ	long 型で定義された型です。
off_type	マクロ	long 型で定義された型です。

7.2.1 ios_base::Init クラス

■定義名一覧

定義名	種類	説明
init_cnt	データ	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。
Init ()	関数	コンストラクタです。
~ Init ()	関数	デストラクタです。

(1) ios_base:: Init::Init()

■機能

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。

■リターン値

なし

(2) ios_base:: Init::~ Init ()

■機能

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。

■リターン値

なし

7.2.2 ios_base クラス

■定義名一覧

定義名	種類	説明
fmtflags	マクロ	フォーマット制御情報を表す型です。
iostate	マクロ	ストリームバッファの入出力状態を表す型です。
openmode	マクロ	ファイルのオープンモードを表す型です。
seekdir	マクロ	ストリームバッファのシーク状態を表す型です。
fmtfl	データ	書式フラグです。
wide	データ	出力データの桁です。
sb	データ	streambuf オブジェクトへのポインタです。
prec	データ	出力時の精度(小数点以下の桁数)です。
fillch	データ	詰め文字です。
tiestr	データ	出力列へのポインタです。
void _ec2p_init_base()	関数	初期化します。
void _ec2p_copy_base(ios_base& ios_base_dt)	関数	ios_base_dt をコピーします。
ios_base()	関数	コンストラクタです。
~ios_base()	関数	デストラクタです。
fmtflags flags() const	関数	書式フラグ(fmtfl)を参照します。
fmtflags flags(fmtflags fmtflg)	関数	書式フラグ(fmtfl)と fmtflg の論理積を書式フラグ(fmtfl)に設定します。
fmtflags setf(fmtflags fmtflg)	関数	fmtflg を書式フラグ(fmtfl)に設定します。
fmtflags setf(fmtflags fmtflg, fmtflags mask)	関数	mask&fmtflg を書式フラグ(fmtfl)に設定します。
void unsetf(fmtflags mask)	関数	~mask&書式フラグ(fmtfl)を書式フラグ(fmtfl)に設定します。
char fill() const	関数	詰め文字(fillch)を参照します。
char fill(char ch)	関数	ch を詰め文字(fillch)に設定します。
int precision() const	関数	精度(prec)を参照します。
streamsize precision(streamsize preci)	関数	preci を精度(prec)に設定します。
streamsize width() const	関数	幅値(wide)を参照します。
streamsize width(streamsize wd)	関数	wd を幅値(wide)に設定します。

(1)　　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;
```

(2)　　ios_base::iostate

ストリームバッファの入出力状態を定義します。
state 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;
```

(3) 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	= 0x1;	**入力用のファイルを** open **します。**
const ios_base::openmode ios_base::out	= 0x2;	**出力用のファイルを** open **します。**
const ios_base::openmode ios_base::ate	= 0x4;	**オープン後一度だけ** eof **に** seek **します。**
const ios_base::openmode ios_base::app	= 0x8;	**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。**

(4) ios_base::seekdir

ストリームバッファの seek 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列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;

(5) void _ec2p_init_base()

■機能

以下の値で初期設定します。
fmtfl = skipws | dec;
wide = 0;
prec = 6;
fillch = ' ';

■リターン値

なし

(6) void _ec2p_copy_base(ios_base & ios_base_dt)

■機能

ios_base_dt をコピーします。

■リターン値

なし

(7) ios_base::ios_base()

■機能

クラス ios_base のコンストラクタです。
Init::Init()を呼び出します。

■リターン値

なし

(8) ios_base::~ios_base()

■機能

クラス ios_base のデストラクタです。

■リターン値

なし

(9) ios_base::fmtflags ios_base::flags() const

■機能

書式フラグ(fmtfl)を参照します。

■リターン値

書式フラグ(fmtfl)

(10) ios_base::fmtflags ios_base::flags(fmtflags fmtflg)

■機能

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

設定前の書式フラグ(fmtfl)

(11) ios_base::fmtflags ios_base::setf(fmtflags fmtflg)

■機能

fmtflg を書式フラグ(fmtfl)に設定します。

■リターン値

設定前の書式フラグ(fmtfl)

(12) ios_base::fmtflags ios_base::setf(fmtflags fmtflg, fmtflags mask)

■機能

mask&fmtflg の値を書式フラグ(fmtfl)に設定します。

■リターン値

設定前の書式フラグ(fmtfl)

(13) void ios_base::unsetf(fmtflags mask)

■機能

~mask&書式フラグ(fmtfl)を書式フラグ(fmtfl)に設定します。

■リターン値

なし

(14) char ios_base::fill() const

■機能

詰め文字(fillch)を参照します。

■リターン値

詰め文字(fillch)

(15) char ios_base::fill(char ch)

■機能

ch を詰め文字として設定します。

■リターン値

設定前の詰め文字(fillch)

(16) int ios_base::precision() const

■機能

精度(prec)を参照します。

■リターン値

精度(prec)

(17) streamsize ios_base::precision(streamsize preci)

■機能

preci を精度(prec)に設定します。

■リターン値

設定前の精度(prec)

(18) streamsize ios_base::width() const

■機能

フィールドの幅(wide)を参照します。

■リターン値

幅(wide)

(19) streamsize ios_base::width(streamsize wd)

■機能

wd をフィールド幅(wide)に設定します。

■リターン値

設定前の幅(wide)

7.2.3 ios クラス

■定義名一覧

定義名	種類	説明
sb	データ	streambuf へのポインタです。
tiestr	データ	ostream へのポインタです。
state	データ	streambuf の状態フラグです。
ios()	関数	コンストラクタです。
ios(streambuf *sbptr)		
void init(streambuf *sbptr)	関数	初期設定を行います。
virtual ~ios()	関数	デストラクタです。
operator void*() const	関数	エラー有無(!state&(badbit|failbit))を判定します。
bool operator !() const	関数	エラー有無(state&(badbit|failbit))を判定します。
iostate rdstate() const	関数	状態フラグ(state)を参照します。
void clear(iostate st=goodbit)	関数	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)	関数	st を状態フラグ(state)に設定します。
bool good() const	関数	エラー有無(state==goodbit)を判定します。
bool eof() const	関数	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	関数	エラー有無(state&badbit)を判定します。
bool fail() const	関数	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	関数	ostream オブジェクトへのポインタ(tiestr)を参照します。
ostream* tie(ostream* tstrptr)	関数	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	関数	ストリームバッファへのポインタ(sb)を参照します。
streambuf* rdbuf(streambuf* sbptr)	関数	sbptr をストリームバッファへのポインタ(sb)に設定します。
ios & copyfmt(const ios & rhs)	関数	rhs の状態フラグ(state)をコピーします。

(1) ios::ios()

■機能

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。

■リターン値

なし

(2) ios::ios(streambuf *sbptr)

■機能

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。

■リターン値

なし

(3) void ios::init(streambuf *sbptr)

■機能

sbptr を sb に設定します。
state、tiestr を 0 に設定します。

■リターン値

なし

(4) virtual ios::~ios()

■機能

クラス ios のデストラクタです。

■リターン値

なし

(5) bool ios::operator void*() const

■機能

エラー有無(!state&(badbit|failbit))を判定します。

■リターン値

エラー有の場合：FALSE
エラー無の場合：TRUE

(6) bool ios::operator!() const

■機能

エラー有無(state&(badbit|failbit))を判定します。

■リターン値

エラー有の場合：TRUE
エラー無の場合：FALSE

(7) iostate ios::rdstate() const

■機能

状態フラグ(state) を参照します。

■リターン値

状態フラグ(state)

(8) void ios::clear(iostate st=goodbit)

■機能

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 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。

■リターン値

なし

(9) void ios::setstate(iostate st)

■機能

st を状態フラグ(state)に設定します。

■リターン値

なし

(10) bool ios::good() const

■機能

エラー有無(state==goodbit)を判定します。

■リターン値

エラー有の場合：FALSE
エラー無の場合：TRUE

(11) bool ios::eof() const

■機能

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

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

(12) bool ios::bad() const

■機能

エラー有無(state&badbit)を判定します。

■リターン値

エラー有の場合：TRUE
エラー無の場合：FALSE

(13) bool ios::fail() const

■機能

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

不一致の場合：TRUE
一致の場合　：FALSE

(14) ostream* ios::tie() const

■機能

ostream オブジェクトポインタ(tiestr)を参照します。

■リターン値

オブジェクトポインタ(tiestr)

(15) ostream* ios::tie(ostream* tstrptr)

■機能

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

設定前のオブジェクトポインタ(tiestr)

(16) streambuf* ios::rdbuf() const

■機能

streambuf へのポインタ(sb) を参照します。

■リターン値

streambuf へのポインタ(sb)

(17) streambuf* ios::rdbuf(streambuf* sbptr)

■機能

sbptr をストリームバッファへのポインタ(sb)に設定します。

■リターン値

設定前の streambuf へのポインタ(sb)

(18) ios & ios::copyfmt(const ios & rhs)

■機能

rhs の状態フラグ(state)をコピーします。

■リターン値

*this

7.2.4　　ios クラスマニピュレータ

■定義名一覧

マニピュレータ	説明
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& noshowbase(　ios_base& str)	基数表示接頭辞モードをクリアします。
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)	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。

(1) ios_base& boolalpha(ios_base& str)

■機能

bool 型の書式に設定します。

■リターン値

str

(2) ios_base& noboolalpha(ios_base& str)

■機能

bool 型の書式をクリアします。

■リターン値

str

(3) ios_base& showbase(ios_base& str)

■機能

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

str

(4) ios_base& noshowbase(ios_base &str)

■機能

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

str

(5) ios_base& showpoint(ios_base & str)

■機能

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

str

(6) ios_base& noshowpoint(ios_base& str)

■機能

小数点を出力するモードをクリアします。

■リターン値

str

(7) ios_base& showpos(ios_base& str)

■機能

+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

str

(8) ios_base& noshowpos(ios_base & str)

■機能

+記号生成出力モードをクリアします。

■リターン値

str

(9) ios_base& skipws(ios_base& str)

■機能

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

str

(10) ios_base& noskipws(ios_base& str)

■機能

空白読み飛ばし入力モードをクリアします。

■リターン値

str

(11) ios_base& uppercase(ios_base& str)

■機能

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

str

(12) ios_base nouppercase(ios_base & str)

■機能

大文字変換出力モードをクリアします。

■リターン値

str

(13) ios_base& internal(ios_base & str)

■**機能**

フィールド幅(wide)の範囲で出力時に
１．符号、基数
２．詰め文字(fill)
３．数値
の順で出力します。

■リターン値

str

(14) ios_base& left(ios_base & str)

■**機能**

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**

str

(15) ios_base& right(ios_base & str)

■**機能**

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**

str

(16) ios_base& dec(ios_base & str)

■**機能**

変換基数を 10 進モードに設定します。

■**リターン値**

str

(17) ios_base& hex(ios_base & str)

■**機能**

変換基数を 16 進モードに設定します。

■リターン値

str

(18) ios_base& oct(ios_base & str)

■**機能**

変換基数を 8 進モードに設定します。

■**リターン値**

str

(19) ios_base& fixed(ios_base & str)

■機能

固定小数点出力モードに設定します。

■リターン値

str

(20) ios_base& scientific(ios_base & str)

■機能

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

str

7.2.5 streambuf クラス

■定義名一覧

定義名	種類	説明
_B_cnt_ptr	データ	バッファの有効データ長へのポインタです。
B_beg_ptr	データ	バッファのベースポインタへのポインタです。
_B_len_ptr	データ	バッファの長さへのポインタです。
B_next_ptr	データ	バッファの次の読み出し位置へのポインタです。
B_end_ptr	データ	バッファの終端位置へのポインタです。
B_beg_pptr	データ	制御バッファの先頭位置へのポインタです。
B_next_pptr	データ	バッファの次の読み出し位置へのポインタです。
C_flg_ptr	データ	ファイルの入出力制御フラグへのポインタです。
char* _ec2p_getflag() const	関数	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()	関数	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()	関数	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()	関数	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()	関数	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)	関数	streambuf のポインタをセットします。
streambuf()	関数	コンストラクタです。
virtual ~streambuf()	関数	デストラクタです。
streambuf* pubsetbuf(char* s, streamsize n)	関数	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)	関数	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)	関数	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()	関数	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。

定義名	種類	説明
streamsize in_avail()	関数	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()	関数	次の一文字を読み込みます。
int_type sbumpc()	関数	一文字読み込みポインタを次に設定します。
int_type sgetc()	関数	一文字読み込みます。
int sgetn(char* s, streamsize n)	関数	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)	関数	読み込み位置をプットバックします。
int sungetc()	関数	読み込み位置をプットバックします。
int sputc(char c)	関数	文字 c を挿入します。
int_type sputn(const char* s, streamsize n)	関数	s の指す n 個の文字を挿入します。
char* eback() const	関数	入力列の先頭ポインタを求めます。
char* gptr() const	関数	入力列の次ポインタを求めます。
char* egptr() const	関数	入力列の最後尾ポインタを求めます。
void gbump(int n)	関数	入力列の次ポインタを n 進めます。
void setg(char* gbeg, char* gnext, char* gend)	関数	入力列の各ポインタを代入します。
char* pbase() const	関数	出力列の先頭ポインタを求めます。
char* pptr() const	関数	出力列の次ポインタを求めます。
char* epptr() const	関数	出力列の最後尾ポインタを求めます。
void pbump(int n)	関数	出力列の次ポインタを n 進めます。
void setp(char* pbeg, char* pend)	関数	出力列の各ポインタを設定します。
virtual streambuf *setbuf(char* s, streamsize n)	関数 [*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))	関数 [*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))	関数 [*1]	ストリーム位置を変更します。

定義名	種類	説明
virtual int sync()	関数 *1	出力ストリームをフラッシュします。
virtual int showmanyc()	関数 *1	入力列の有効な文字数を求めます。
virtual streamsize xsgetn(char* s, streamsize n)	関数	s の指す記憶領域に n 個の文字を設定します。
virtual int_type underflow()	関数 *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()	関数 *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)	関数 *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)	関数	s の指す n 個の文字を挿入します。
virtual int_type overflow(int_type c=eof)	関数 *1	c を出力列に挿入します。

注 *1 このクラスでは処理を定義していません。

(1) streambuf ::streambuf()

■機能

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 =_Blen_ptr = 0
B_beg_pptr = &B_beg_ptr
B_next_pptr = &B_next_ptr

■リターン値

なし

(2) virtual streambuf ::~streambuf()

■機能

デストラクタです。

■リターン値

なし

(3) streambuf* streambuf ::pubsetbuf(char* s, streamsize n)

■機能

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

setbuf(s,n)

(4) pos_type streambuf ::pubseekoff(off_type off,ios _base::seekdir way, ios_base::openmode which=(ios_base::openmode)(ios_base::in|ios_base::out))

■機能

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

新たに設定されたストリームの位置

(5) pos_type streambuf ::pubseekpos(pos_type sp, ios_base::openmode which=(ios_base::openmode) (ios_base::in | ios_base::out))

■機能

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

先頭からのオフセット

(6) int streambuf ::pubsync()

■機能

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

0

(7) streamsize streambuf ::in_avail()

■機能

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

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)

(8) int_type streambuf ::snextc()

■機能

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

EOF でない場合：読み込んだ文字
EOF の場合　　：EOF

(9) int_type streambuf::sbumpc()

■機能

一文字読み込みポインタを次に設定します。

■リターン値

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

(10) int_type streambuf ::sgetc()

■機能

一文字読み込みます。

■リターン値

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

(11) int streambuf ::sgetn(char* s, streamsize n)

■機能

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

設定した文字数

(12) int_type streambuf::sputbackc(char c) ;

■機能

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

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

(13) int streambuf::sungetc()

■機能

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

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

(14) int streambuf::sputc(char c)

■機能

文字 c を挿入します。

■リターン値

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

(15) int_type streambuf ::sputn(const char* s, streamsize n)

■機能

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

挿入された文字数

(16) char* streambuf ::eback() const

■機能

入力ストリームの先頭ポインタを求めます。

■リターン値

先頭ポインタ

(17) char* streambuf ::gptr() const

■機能

入力ストリームの次ポインタを求めます。

■リターン値

次ポインタ

(18) char* streambuf ::egptr() const

■機能

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

最後尾ポインタ

(19) void streambuf ::gbump(int n)

■機能

入力ストリームの次ポインタを n 進めます。

■リターン値

なし

(20)　void streambuf ::setg(char* gbeg, char* gnext, char* gend)

■機能

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;

■リターン値

なし

(21)　char* streambuf ::pbase() const

■機能

出力ストリームの先頭ポインタを求めます。

■リターン値

先頭ポインタ

(22)　char* streambuf ::pptr() const

■機能

出力ストリームの次ポインタを求めます。

■リターン値

次ポインタ

(23)　char* streambuf ::epptr() const

■機能

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

最後尾ポインタ

(24)　void streambuf ::pbump(int n)

■機能

出力ストリームの次ポインタを n 進めます。

■リターン値

なし

(25) void streambuf ::setp(char* pbeg, char* pend)

■機能

出力列の各ポインタに、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;

■リターン値

なし

(26) virtual streambuf* streambuf ::setbuf(char* s, streamsize n)

■機能

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

*this（このクラスでは処理を定義していません）

(27) virtual pos_type streambuf ::seekoff(off_type off, ios_base::seekdir way , ios_base::openmode=(ios_base::openmode)(ios_base::in | ios_base::out))

■機能

ストリーム位置を変更します。

■リターン値

(-1) （このクラスでは処理を定義していません）

(28) virtual pos_type streambuf ::seekpos(pos_type off, ios_base::openmode=(ios_base::openmode)(ios_base::in | ios_base::out))

■機能

ストリーム位置を変更します。

■リターン値

(-1) （このクラスでは処理を定義していません）

(29) virtual int streambuf ::sync()

■機能

出力ストリームをフラッシュします。

■リターン値

0（このクラスでは処理を定義していません）

(30) virtual int streambuf ::showmanyc()

■機能

入力ストリームの有効な文字数を求めます。

■リターン値

0（このクラスでは処理を定義していません）

(31) virtual streamsize streambuf ::xsgetn(char* s, streamsize n)

■機能

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

入力された文字数

(32) virtual int_type streambuf ::underflow()

■機能

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

EOF（このクラスでは処理を定義していません）

(33) virtual int_type streambuf ::uflow()

■機能

次ポインタの一文字を読み込みます。

■リターン値

EOF（このクラスでは処理を定義していません）

(34) virtual int_type streambuf ::pbackfail(int_type c=eof)

■機能

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

EOF（このクラスでは処理を定義していません）

(35) virtual streamsize streambuf ::xsputn(const char* s, streamsize n)

■機能

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

挿入された文字数

(36) virtual int_type streambuf ::overflow(int_type c=eof)

■機能

c を出力ストリームに挿入します。

■リターン値

EOF（このクラスでは処理を定義していません）

7.2.6 istream::sentry クラス

■定義名一覧

定義名	種類	説明
ok_	データ	入力可能状態か否かを意味します。
sentry(istream& is, bool noskipws=_FALSE)	関数	コンストラクタです。
~sentry()	関数	デストラクタです。
operator bool()	関数	ok_を参照します。

(1) istream::sentry::sentry(istream&is, bool noskipws=_FALSE)

■機能

内部クラス sentry のコンストラクタです。

■リターン値

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、出力列と関連する外部 C ストリームの同期をあわせます。

(2) istream::sentry::~sentry()

■機能

内部クラス sentry のデストラクタです。

■リターン値

なし

(3) istream::sentry::operator bool()

■機能

ok_を参照します。

■リターン値

ok_

7.2.7　　istream クラス

■定義名一覧

定義名	種類	説明
chcount	データ	最後にコールされた入力関数が抽出した文字数です。
int::_ec2p_getistr(char* str, unsigned int dig, int mode)	関数	str を dig が示す基数で変換します。
istream(streambuf *sb)	関数	コンストラクタです。
~istream()	関数	デストラクタです。
istream& operator >>(bool &n)	関数	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)	関数	void を指すポインタに変換して p に格納します。
istream& operator >>(streambuf *sb)	関数	文字を抽出し、sb の指す記憶領域へ格納します。
streamsize gcount() const	関数	chcount(抽出文字数)を求めます。
int_type get()	関数	文字を抽出します。
istream& get(char &c)	関数	文字を抽出し c に格納します。
istream& get(signed char &c)		
istream& get(unsigned char &c)		

定義名	種類	説明
istream& get(char* s, streamsize n)	関数	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)	関数	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)	関数	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)	関数	文字列を抽出し、sb の指す記憶領域に格納します。途中文字'delim'を検出したら、入力を終了します。
istream& getline(char* s, streamsize n)	関数	サイズ n-1 の文字列を抽出し、sb 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)		
istream& getline(char* s, streamsize n, char delim)	関数	サイズ n-1 の文字列を抽出し、sb の指す記憶領域に格納します。 途中文字'delim'を検出したら、入力を終了します。
istream& getline(signed char* s, streamsize n, char delim)		

定義名	種類	説明
istream& getline(unsigned char* s, streamsize n, char delim)	関数	サイズ n-1 の文字列を抽出し、sb の指す記憶領域に格納します。 途中文字'delim'を検出したら、入力を終了します。
istream& ignore(unsigned streamsize n=1, int_type delim=streambuf::eof)	関数	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()	関数	次の入手可能な入力文字を求めます。
istream& read(char* s, streamsize n)	関数	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(unsgined char* s, streamsize n)		
streamsize readsome(char* s, streamsize n)	関数	n 個の文字列を抽出し、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)	関数	文字を入力ストリームに戻します。
istream& unget()	関数	入力ストリームの位置を戻します。
int sync()	関数	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()	関数	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)	関数	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)	関数	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 stream::pubseekoff(off,dir)を呼び出します。

(1) int istream::_ec2p_getistr(char* str, unsigned int dig, int mode)

■機能

str を dig が示す基数で変換します。

■リターン値

変換した基数を返します。

(2) istream::istream(streambuf *sb)

■機能

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。

■リターン値

なし

(3) virtual istream::~istream()

■機能

クラス istream のデストラクタです。

■リターン値

なし

(4) 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)

■機能

抽出した文字を n に格納します。

■リターン値

*this

(5) istream& istream::operator >> (void * &p)

■**機能**

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**

*this

(6) istream& istream::operator >> (streambuf *sb)

■**機能**

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**

*this

(7) streamsize istream::gcount() const

■**機能**

chcount(抽出文字数)を参照します。

■**リターン値**

chcount

(8) int_type istream::get()

■**機能**

文字を抽出します。

■**リターン値**

抽出可能の場合：抽出した文字
抽出不可の場合：setstate(failbat)呼び出し、EOF

(9) istream& istream::get(char &c)
istream& istream::get(signed char &c)
istream& istream::get(unsigned char &c)

■**機能**

文字を抽出し c に格納します。抽出した文字が EOF の場合は、failbit を設定します。

■**リターン値**

*this

(10) istream& istream::get(char* s, streamsize n)
istream& istream::get(signed char* s, streamsize n)
istream& istream::get(unsigned char* s, streamsize n)

■機能

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

*this

(11) 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)

■機能

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

*this

(12) istream& istream::get(streambuf&sb, char delim)

■機能

文字列を抽出し、sb の指す記憶領域に格納します。
途中文字'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

*this

(13) istream& istream::get(streambuf&sb)

■機能

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

*this

(14) istream& istream::getline(char* s, streamsize n)
istream& istream::getline(signed char* s, streamsize n)
istream& istream::getline(unsigned char* s, streamsize n)

■機能

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

*this

(15) 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)

■機能

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

*this

(16) istream& istream::ignore(streamsize n = 1, int_type delim = streambuf::eof)

■機能

n 個の文字を読み飛ばします。
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

*this

(17) int_type istream::peek()

■機能

次の入力可能な入力文字を求めます。

■リターン値

ok_==FALSE の場合、EOF
ok_!=FALSE の場合、rdbuf()->sgetc()

(18) istream& istream::read(char* s, streamsize n)
istream& istream::read(signed char* s, streamsize n)
istream& istream::read(unsigned char* s, streamsize n)

■機能

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

*this

(19) streamsize istream::readsome(char* s, streamsize n)
streamsize istream::readsome(signed char* s, streamsize n)
streamsize istream::readsome(unsigned char* s, streamsize n)

■機能

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

抽出した文字数

(20) istream& istream::putback(char c)

■機能

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 EOF の場合は、badbit を設定します。

■リターン値

*this

(21) istream& istream::unget()

■機能

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 EOF の場合、badbit を設定します。

■リターン値

*this

(22) int istream::sync()

■機能

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

入力ストリームがない場合：EOF
入力ストリームがある場合：0

(23) pos_type istream::tellg()

■機能

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

ストリームの先頭からのオフセット
入力処理にエラーが発生した場合、-1

(24)　istream& istream::seekg(pos_type & pos)

■機能

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

*this

(25)　istream& istream::seekg(off_type & off, ios_base::seekdir dir)

■機能

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

*this

7.2.8 istream クラスマニピュレータ

■定義名一覧

定義名	説明
istream& ws(istream &is)	空白文字を読み飛ばします。

(1) istream& ws(istream &is)

■機能

空白類を読み飛ばします。

■リターン値

is

7.2.9 istream メンバ外関数

■定義名一覧

定義名	種類	説明
istream& operator >> (istream& in, char* s)	関数	文字を抽出し、s で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 &s)		
istream& operator >> (istream& in, singed char &s)		
istream& operator >> (istream& in, unsigned char &s)		

(1) istream& operator >>(istream& in, char* s)

istream& operator >>(istream& in, signed char* s)

istream& operator >>(istream& in, unsigned char* s)

istream& operator>> (istream& in, char &s)

istream& operator>> (istream& in, singed char &s)

istream& operator>> (istream& in, unsigned char &s)

■機能

文字を抽出し、s で指す記憶領域に格納します。
n-1 個の文字を格納したか、または入力列に 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

in

7.2.10 ostream::sentry クラス

■定義名一覧

定義名	種類	説明
ok_	データ	出力可能状態か否かを意味します。
__ec2p_os	データ	ostream オブジェクトへのポインタです。
sentry(　ostream &os)	関数	コンストラクタです。
~sentry()	関数	デストラクタです。
operator bool()	関数	ok_を参照します。

(1) ostream::sentry::sentry(ostream &os)

■機能

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。

■リターン値

なし

(2) ostream::sentry::~sentry()

■機能

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。

■リターン値

なし

(3) ostream::sentry::operator bool()

■機能

ok_を参照します。

■リターン値

ok_

7.2.11 ostream クラス

■定義名一覧

定義名	種類	説明
ostream(streambuf *sbptr)	関数	コンストラクタです
~ostream()	関数	デストラクタです
ostream & operator << (bool n)	関数	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)	関数	sbptr の出力列を出力ストリームに挿入します。
ostream & put(char c)	関数	文字 c を出力ストリームに挿入します。
ostream & write(const char* s, streamsize n)	関数	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()	関数	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。

定義名	種類	説明
pos_type tellp()	関数	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)	関数	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)	関数	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。

(1) ostream::ostream(streambuf *sbptr)

■機能

コンストラクタです。
ios (sbptr)を呼び出します。

■リターン値

なし

(2) virtual ostream::~ostream()

■機能

デストラクタです。

■リターン値

なし

(3)　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)

■機能

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

*this

(4)　ostream & ostream::operator << (streambuf *sbptr)

■機能

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

*this

(5)　ostream & ostream::put(char c)

■機能

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

*this

(6)　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)

■機能

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

*this

(7) ostream & ostream::flush()

■機能

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

*this

(8) pos_type ostream::tellp()

■機能

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

現在のストリームの位置
処理中にエラーが発生した場合は-１

(9) ostream & ostream::seekp(pos_type pos)

■機能

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

*this

(10) ostream & ostream::seekp(off_type off, seekdir dir)

■機能

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(pos,dir)を呼び出します。

■リターン値

*this

7.2.12 ostream クラスマニピュレータ

■定義名一覧

定義名	説明
ostream & endl(ostream &os)	改行を付加し、出力ストリームをフラッシュします。
ostream & ends(ostream &os)	ヌルコードを付加します。
ostream & flush(ostream &os)	出力ストリームをフラッシュします。

(1) ostream & endl(ostream &os)

■機能

ストリームに改行文字を付加します。
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

os

(2) ostream & ends(ostream &os)

■機能

出力列にヌルコードを挿入します。

■リターン値

os

(3) ostream & flush(ostream &os)

■機能

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

os

7.2.13　ostream メンバ外関数

■定義名一覧

定義名	種類	説明
ostream& operator << (ostream& os, char s)	関数	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 signed char* s)		
ostream& operator << (ostream& os, const unsigned char* s)		

(1)　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)

■機能

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

os

7.2.14 smanip クラスマニピュレータ

■定義名一覧

定義名	種類	説明
smanip resetiosflags(ios_base::fmtflags mask)	関数	mask 値で指定されたフラグをクリアします。
smanip setiosflags(ios_base::fmtflags mask)	関数	書式フラグ(fmtfl)の設定を行います。
smanip setbase(int base)	関数	出力時に用いる基数をセットします。
smanip setfill(char c)	関数	詰め文字(fillch)の設定を行います。
smanip setprecision(int n)	関数	精度(prec)の指定を行います。
smanip setw(int n)	関数	フィールド幅(wide)の設定を行います。

(1) smanip resetiosflags(ios_base::fmtflags mask)

■機能

mask 値で指定されたフラグをクリアします。

■リターン値

入出力対象のオブジェクト

(2) smanip setiosflags(ios_base::fmtflags(0), mask)

■機能

書式フラグ(fmtfl)の設定を行います。

■リターン値

入出力対象のオブジェクト

(3) smanip setbase(int base)

■機能

出力時に用いる基数をセットします。

■リターン値

入出力対象のオブジェクト

(4) smanip setfill(char c)

■機能

詰め文字の設定を行います。

■リターン値

入出力対象のオブジェクト

(5) smanip setprecision(int n)

■機能

精度の指定を行います。

■リターン値

入出力対象のオブジェクト

(6) smanip setw(int n)

■機能

フィールド幅の設定を行います。

■リターン値

入出力対象のオブジェクト

7.2.15 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。

■定義名一覧

定義名	種類	説明
_file_Ptr	データ	ファイルポインタです。
mystrbuf()	関数	コンストラクタです。
mystrbuf(void *ptr)		streambuf バッファの初期化を行います。
~mystrbuf()	関数	デストラクタです。
void *myfptr() const	関数	FILE 型構造体へのポインタを返します。
mystrbuf *open(const char* filename, int mode)	関数	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()	関数	ファイルのクローズを行います。
virtual streambuf *setbuf(char* s, streamsize n)	関数	-
virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode=(ios_base::openmode) (ios_base::in | ios_base::out))	関数	-
virtual pos_type seekpos(pos_type sp, ios_base::openmode=(ios_base::openmode) (ios_base::in | ios_base::out))	関数	ストリームポインタの位置を変えます。
virtual int sync()	関数	ストリームをフラッシュします。
virtual int showmanyc()	関数	入力列の有効な文字数を返します。
virtual int_type underflow()	関数	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)	関数	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)	関数	c によって示される文字を挿入します。
void _Init(_f_type *fp)	関数	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;
}
```

7.3　　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。

■定義名一覧

定義名	種類	説明
new_handler	マクロ	void 型を返す関数へのポインタ型です。
_ec2p_new_handler	データ	例外処理関数へのポインタです。
void* operator new(size_t size)	関数	size 分の領域を確保します。
void* operator new[](size_t size)	関数	size 分の配列領域を確保します。
void* operator new(size_t size, void *ptr)	関数	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)	関数	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 delete(void* ptr)	関数	領域を解放します。
void operator delete[](void* ptr)	関数	配列領域を解放します。
new_handler set_new_handler(new_handler new_P)	関数	_ec2p_new_handler に例外処理関数アドレス（new_P）をセットします。

(1)　　void* operator new(size_t size)

■機能

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 がセットされていれば、new_handler を呼び出します。

■リターン値

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

(2) void* operator new[](size_t size)

■機能

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 がセットされていれば、new_handler を呼び出します。

■リターン値

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

(3) void* operator new(size_t size, void *ptr)

■機能

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

ptr

(4) void* operator new[](size_t size, void *ptr)

■機能

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

ptr

(5) void operator delete(void* ptr)

■機能

ptr が指す記憶領域を解放します。ptr が NULL のときはなにもしません。

■リターン値

なし

(6) void operator delete[](void* ptr)

■機能

ptr が指す配列領域を解放します。ptr が NULL のときはなにもしません。

■リターン値

なし

(7) set_new_handler(new_handler new_P)

■ 機能

_ec2p_new_handler に new_P をセットします

■リターン値

_ec2p_new_handler の値

7.4 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。

(1) <complex>

float_complex クラス、double_complex クラスを定義します。

また、これらのクラスの階層図は次のようになります。

float_complex　double_complex

7.4.1 float_complex クラス

■定義名一覧

定義名	種類	説明
value_type	マクロ	float 型です。
_re	データ	float 精度の実数部を定義します。
_im	データ	float 精度の虚数部を定義します。
float_complex(float re=0.0f, float im=0.0f)	関数	コンストラクタです。
float_complex(const double_complex& rhs)		
float real() const	関数	実数部(_re)を求めます。
float imag() const	関数	虚数部(_im)を求めます。
float_complex& operator=(float rhs)	関数	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)	関数	rhs を実数部に加算し、和を*this に格納します。
float_complex& operator-=(float rhs)	関数	rhs を実数部から減算し、差を*this に格納します。
float_complex& operator*=(float rhs)	関数	rhs を乗算し、積を*this に格納します。
float_complex& operator/=(float rhs)	関数	rhs で除算し、商を*this に格納します。
float_complex& operator=(const float_complex&rhs)	関数	rhs をコピーします。
float_complex& operator+=(const float_complex& rhs)	関数	rhs を加算し、和を*this に格納します。
float_complex& operator-=(const float_complex& rhs)	関数	rhs を減算し、差を*this に格納します。
float_complex& operator*=(const float_complex& rhs)	関数	rhs を乗算し、積を*this に格納します。
float_complex& operator/=(const float_complex& rhs)	関数	rhs で除算し、商を*this に格納します。

(1) float_complex::float_complex(float re=0.0f, float im=0.0f)

■機能

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;

■リターン値

なし

(2) float_complex::float_complex(const double_complex& rhs)

■機能

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();

■リターン値

なし

(3) float float_complex::real() const

■機能

実数部を求めます。

■リターン値

this->_re

(4) float float_complex::imag() const

■機能

虚数部を求めます。

■リターン値

this->_im

(5) float_complex& float_complex::operator=(float rhs)

■機能

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

*this

(6) float_complex& float_complex::operator+=(float rhs)

■機能

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

*this

(7) float_complex& float_complex::operator-=(float rhs)

■機能

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

*this

(8) float_complex& float_complex::operator*=(float rhs)

■機能

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

*this

(9) float_complex& float_complex::operator/=(float rhs)

■機能

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

*this

(10) float_complex& float_complex::operator=(const float_complex& rhs)

■機能

rhs をコピーします。

■リターン値

*this

(11) float_complex& float_complex::operator+=(const float_complex& rhs)

■機能

rhs を加算し、結果を*this に格納します。

■リターン値

*this

(12) float_complex& float_complex::operator-=(const float_complex& rhs)

■機能

rhs を減算し、結果を*this に格納します。

■リターン値

*this

(13) float_complex& float_complex::operator*=(const float_complex& rhs)

■機能

rhs と乗算し、結果を*this に格納します。

■リターン値

*this

(14) float_complex& float_complex::operator/=(const float_complex& rhs)

■機能

rhs で除算し、結果を*this に格納します。

■リターン値

*this

7.4.2　float_complex メンバ外関数

■定義名一覧

定義名	種類	説明
float_complex operator+(const float_complex& lhs)	関数	lhs の単項＋演算を行います。
float_complex operator+(const float_complex& lhs, const float_complex& rhs)	関数	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)	関数	lhs の単項－演算を行います。
float_complex operator-(const float_complex& lhs, const float_complex& rhs)	関数	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)	関数	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)	関数	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)		

定義名	種類	説明
bool operator==(const float_complex& lhs, const float_complex& rhs)	関数	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)		
bool operator!=(const float_complex& lhs, const float_complex& rhs)	関数	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)	関数	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)	関数	u,(u)または x を(u,v)（u:実数部、v:虚数部）形式で出力します。
float real(const float_complex& x)	関数	実数部を求めます。
float imag(const float_complex& x)	関数	虚数部を求めます。
float abs(const float_comlex& x)	関数	絶対値を求めます。
float arg(const float_complex& x)	関数	位相角度を求めます。
float norm(const float_complex& x)	関数	２乗の絶対値を求めます。
float_complex conj(const float_complex& x)	関数	共役複素数を求めます。
float_complex polar(const float& rho, const float& theta)	関数	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)	関数	複素余弦を求めます。
float_complex cosh(const float_complex& x)	関数	複素双曲余弦を求めます。
float_complex exp(const float_complex& x)	関数	指数関数を求めます。

定義名	種類	説明
float_complex log(const float_complex& x)	関数	自然対数を求めます。
float_complex log10(const float_complex& x)	関数	常用対数を求めます。
float_complex pow(const float_complex& x, int y)	関数	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)	関数	複素正弦を求めます。
float_complex sinh(const float_complex& x)	関数	複素双曲正弦を求めます。
float_complex sqrt(const float_complex& x)	関数	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)	関数	複素正接を求めます。
float_complex tanh(const float_complex& x)	関数	複素双曲正接を求めます。

(1) float_complex operator+(const float_complex& lhs)

■機能

lhs の単項＋演算を行います。

■リターン値

lhs

(2) 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)

■ 機能

lhs と rhs を加算し、結果を lhs に格納します。

■ リターン値

float_complex(lhs)+=rhs

(3) float_complex operatorー(const float_complex& lhs)

■機能

lhs の単項－演算を行います。

■ リターン値

float_complex(－lhs.real(),－lhs.imag())

(4) float_complex operatorー(const float_complex& lhs, const float_complex& rhs)

float_complex operatorー(const float_complex& lhs, const float& rhs)

float_complex operatorー(const float& lhs, const float_complex& rhs)

■機能

lhs から rhs を減算し、結果を lhs に格納します。

■ リターン値

float_complex(lhs)－=rhs

(5) 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)

■機能

lhs と rhs を乗算し、結果を lhs に格納します。

■ リターン値

float_complex(lhs)*=rhs

(6) 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)

■機能

lhs を rhs で除算し、結果を lhs に格納します。

■ リターン値

float_complex(lhs)/=rhs

(7) 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)

■機能

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

lhs.real()==rhs.real() && lhs.imag()==rhs.imag()

(8) 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)

■機能

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

lhs.real()!=rhs.real() || lhs.imag()!=rhs.imag()

(9) istream& operator>>(istream& is, float_complex& x)

■機能

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

is

(10) ostream& operator<<(ostream& os, const float_complex& x)

■機能

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

os

(11) float real(const float_complex& x)

■機能

実数部を求めます。

■リターン値

x.real()

(12) float imag(const float_complex& x)

■機能

虚数部を求めます。

■リターン値

x.imag()

(13) float abs(const float_complex& x)

■機能

絶対値を求めます。

■リターン値

|x.real()| + |x.imag()|

(14) float arg(const float_complex& x)

■機能

位相角度を求めます。

■リターン値

atan2f(x.imag() , x.real())

(15) float norm(const float_complex& x)

■機能

2 乗の絶対値を求めます。

■リターン値

x.real()^2+ x.imag()^2

(16) float_complex conj(const float_complex& x)

■機能

共役複素数を求めます。

■リターン値

float_complex(x.real(), (-1)*x.imag())

(17) float_complex polar(const float& rho, const float& theta)

■機能

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

float_complex(rho*cosf(theta), rho*sinf(theta))

(18) float_complex cos(const float_complex& x)

■機能

複素余弦を求めます。

■リターン値

float_complex(cosf(x.real())*coshf(x.imag()), (-1)*sinf(x.real())*sinhf(x.imag()))

(19)　float_complex cosh(const float_complex& x)

■ 機能

複素双曲余弦を求めます。

■リターン値

cos(float_complex((-1)*x.imag(), x.real()))

(20)　float_complex exp(const float_complex& x)

■機能

指数関数を求めます。

■リターン値

expf(x.real())*cosf(x.imag()),expf(x.real())*sinf(x.imag())

(21)　float_complex log(const float_complex& x)

■機能

(e を底とする)自然対数を求めます。

■リターン値

float_complex(logf(abs(x)), arg(x))

(22)　float_complex log10(const float_complex& x)

■機能

(10 を底とする)常用対数を求めます。

■リターン値

float_complex(log10f(abs(x)), arg(x)/logf(10))

(23)　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)

■機能

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

exp(y* logf(x)) (float_complex pow (const float_complex& x,const float_complex& y)のとき)
exp(y*log(x)) (上記以外)

(24) float_complex sin(const float_complex& x)

■機能

複素正弦を求めます。

■リターン値

float_complex(sinf(x.real())*coshf(x.imag()), cosf(x.real())*sinhf(x.imag()))

(25) float_complex sinh(const float_complex& x)

■機能

複素双曲正弦を求めます。

■ リターン値

float_complex(0,-1)*sin(float_complex((-1)*x.imag(),x.real()))

(26) float_complex sqrt(const float_complex& x)

■ 機能

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

float_complex(sqrtf(abs(x))*cosf(arg(x)/2), sqrtf(abs(x))*sinf(arg(x)/2))

(27) float_complex tan(const float_complex& x)

■ 機能

複素正接を求めます。

■ リターン値

sin(x) / cos(x)

(28) float_complex tanh(const float_complex& x)

■ 機能

複素双曲正接を求めます。

■ リターン値

sinh(x) / cosh(x)

7.4.3　double_complex クラス

■定義名一覧

定義名	種類	説明
value_type	マクロ	double 型です。
_re	関数	double 精度の実数部を定義します。
_im	関数	double 精度の虚数部を定義します。
double_complex(double re=0.0, double im=0.0)	関数	コンストラクタです。
double_complex(const float_complex&)		
double real() const	関数	実数部を求めます。
double imag() const	関数	虚数部を求めます。
double_complex& operator=(double rhs)	関数	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)	関数	rhs を実数部に加算し、和を*this に格納します。
double_complex& operator-=(double rhs)	関数	rhs を実数部から減算し、差を*this に格納します。
double_complex& operator*=(double rhs)	関数	rhs を乗算し、積を*this に格納します。
double_complex& operator/=(double rhs)	関数	rhs で除算し、商を*this に格納します。
double_complex& operator=(const double_complex& rhs)	関数	rhs をコピーします。
double_complex& operator+=(const double_complex& rhs)	関数	rhs を加算し、和を*this に格納します。
double_complex& operator-=(const double_complex& rhs)	関数	rhs を減算し、差を*this に格納します。
double_complex& operator*=(const double_complex& rhs)	関数	rhs を乗算し、積を*this に格納します。
double_complex& operator/=(const double_complex& rhs)	関数	rhs で除算し、商を*this に格納します。

(1)　double_complex(double re=0.0, double im=0.0)

■機能

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;
```

■リターン値

なし

(2) double_complex(const float_complex&)

■機能

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();

■リターン値

なし

(3) double double_complex::real() const

■機能

実数部を求めます。

■リターン値

this->_re

(4) double double_complex::imag() const

■機能

虚数部を求めます。

■リターン値

this->_im

(5) double_complex& double_complex::operator=(double rhs)

■機能

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

*this

(6) double_complex& double_complex::operator+=(double rhs)

■機能

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

*this

(7) double_complex& double_complex::operator-=(double rhs)

■機能

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

*this

(8) double_complex& double_complex::operator*=(double rhs)

■機能

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

*this

(9) double_complex& double_complex::operator/=(double rhs)

■機能

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

*this

(10) double_complex& double_complex::operator=(const double_complex& rhs)

■機能

rhs をコピーします。

■リターン値

*this

(11) double_complex& double_complex::operator+=(const double_complex& rhs)

■機能

rhs を加算し、結果を*this に格納します。

■リターン値

*this

(12) double_complex& double_complex::operator-=(const double_complex& rhs)

■機能

rhs を減算し、結果を*this に格納します。

■リターン値

*this

(13) double_complex& double_complex::operator*=(const double_complex& rhs)

■機能

rhs と乗算し、結果を*this に格納します。

■リターン値

*this

(14) double_complex& double_complex::operator/=(const double_complex& rhs)

■機能

rhs で除算し、結果を*this に格納します。

■リターン値

*this

7.4.4 double_complex メンバ外関数

■定義名一覧

定義名	種類	説明
double_complex operator+(const double_complex& lhs)	関数	lhs の単項＋演算を行います。
double_complex operator+(const double_complex& lhs, const double_complex& rhs)	関数	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)	関数	lhs の単項ー演算を行います。
double_complex operator-(const double_complex& lhs, const double_complex& rhs)	関数	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)	関数	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)	関数	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)		

定義名	種類	説明
bool operator==(const double_complex& lhs, const double_complex& rhs)	関数	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)		
bool operator!=(const double_complex& lhs, const double_complex& rhs)	関数	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)	関数	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)	関数	u,(u)または x を(u,v)（u:実数部、v:虚数部）形式で出力します。
double real(const double_complex& x)	関数	実数部を求めます。
double imag(const double_complex& x)	関数	虚数部を求めます。
double abs(const double_comlex& x)	関数	絶対値を求めます。
double arg(const double_complex& x)	関数	位相角度を求めます。
double norm(const double_complex& x)	関数	２乗の絶対値を求めます。
double_complex conj(const double_complex& x)	関数	共役複素数を求めます。
double_complex polar(const double& rho, const double& theta)	関数	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)	関数	複素余弦を求めます。
double_complex cosh(const double_complex& x)	関数	複素双曲余弦を求めます。
double_complex exp(const double_complex&)	関数	指数関数を求めます。

定義名	種類	説明
double_complex log(const double_complex& x)	関数	自然対数を求めます。
double_complex log10(const double_complex& x)	関数	常用対数を求めます。
double_complex pow(const double_complex& x, int y)	関数	x の y 乗を求めます。
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)		
double_complex sin(const double_complex& x)	関数	複素正弦を求めます。
double_complex sinh(const double_complex& x)	関数	複素双曲正弦を求めます。
double_complex sqrt(const double_complex& x)	関数	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)	関数	複素正接を求めます。
double_complex tanh(const double_complex& x)	関数	複素双曲正接を求めます。

(1)　　double_complex operator+(const double_complex& lhs)

■機能

lhs の単項＋演算を行います。

■リターン値

lhs

(2)　　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)

■ 機能

lhs と rhs を加算し、結果を lhs に格納します。

■ リターン値

double_complex(lhs)+=rhs

(3) double_complex operatorー(const double_complex& lhs)

■機能

lhs の単項ー演算を行います。

■ リターン値

double_complex(ーlhs.real(), ーlhs.imag())

(4) double_complex operatorー(const double_complex& lhs, const double_complex& rhs)

double_complex operatorー(const double_complex& lhs, const double& rhs)

double_complex operatorー(const double& lhs, const double_complex& rhs)

■機能

lhs から rhs を減算し、結果を lhs に格納します。

■ リターン値

double_complex(lhs)ー=rhs

(5) 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)

■機能

lhs と rhs を乗算し、結果を lhs に格納します。

■ リターン値

double_complex(lhs)*=rhs

(6) 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)

■機能

lhs を rhs で除算し、結果を lhs に格納します。

■ リターン値

double_complex(lhs)/=rhs

(7) 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)

■機能

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

lhs.real()==rhs.real() && lhs.imag()==rhs.imag()

(8) 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)

■機能

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

lhs.real()!=rhs.real() || lhs.imag()!=rhs.imag()

(9) istream& operator>>(istream& is, double_complex& x)

■機能

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

is

(10) ostream& operator<<(ostream& os, const double_complex& x)

■機能

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

os

(11) double real(const double_complex& x)

■機能

実数部を求めます。

■リターン値

x.real()

(12) double imag(const double_complex& x)

■機能

虚数部を求めます。

■リターン値

x.imag()

(13) double abs(const double_complex& x)

■機能

絶対値を求めます。

■リターン値

|x.real()| + |x.imag()|

(14) double arg(const double_complex& x)

■機能

位相角度を求めます。

■リターン値

atan2(x.imag() , x.real())

(15) double norm(const double_complex& x)

■機能

2 乗の絶対値を求めます。

■リターン値

x.real()^2+ x.imag()^2

(16) double_complex conj(const double_complex& x)

■機能

共役複素数を求めます。

■リターン値

double_complex(x.real(), (-1)*x.imag())

(17) double_complex polar(const double& rho, const double& theta)

■機能

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

double_complex(rho*cos(theta), rho*sin(theta))

(18) double_complex cos(const double_complex& x)

■機能

複素余弦を求めます。

■リターン値

double_complex(cos(x.real())*cosh(x.imag()), (-1)*sin(x.real())*sinh(x.imag()))

(19) double_complex cosh(const double_complex& x)

■ 機能

複素双曲余弦を求めます。

■リターン値

cos(double_complex((-1)*x.imag(), x.real()))

(20) double_complex exp(const double_complex& x)

■機能

指数関数を求めます。

■リターン値

exp(x.real())*cos(x.imag()),exp(x.real())*sin(x.imag())

(21) double_complex log(const double_complex& x)

■機能

(e を底とする)自然対数を求めます。

■リターン値

double_complex(log(abs(x)), arg(x)))

(22) double_complex log10(const double_complex& x)

■機能

(10 を底とする)常用対数を求めます。

■リターン値

double_complex(log10(abs(x)), arg(x)/log(10)))

(23) 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)

■機能

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

exp(y*log(x))

(24) double_complex sin(const double_complex& x)

■機能

複素正弦を求めます。

■リターン値

double_complex(sin(x.real())*cosh(x.imag()), cos(x.real())*sinh(x.imag()))

(25) double_complex sinh(const double_complex& x)

■機能

複素双曲正弦を求めます。

■ リターン値

double_complex(0,-1)*sin(double_complex((-1)*x.imag(),x.real()))

(26) double_complex sqrt(const double_complex& x)

■ 機能

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

double_complex(sqrt(abs(x))*cos(arg(x)/2), sqrt(abs(x))*sin(arg(x)/2))

(27) double_complex tan(const double_complex& x)

■ 機能

複素正接を求めます。

■ リターン値

sin(x) / cos(x)

(28) double_complex tanh(const double_complex& x)

■ 機能

複素双曲正接を求めます。

■ リターン値

sinh(x) / cosh(x)

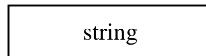
7.5 文字列操作用クラスライブラリ

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。

(1) <string>

string クラスを定義します。

また、クラスの階層図は次のようになります。



7.5.1 string クラス

定義名	種別	説明
iterator	マクロ	char*型です。
const_iterator	マクロ	const char*型です。
npos	データ	文字列の最大長(UINT_MAX 文字)です。
s_ptr	データ	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	データ	オブジェクトが格納している文字列長です。
s_res	データ	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。
string(void)	関数	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()	関数	デストラクタです。
string& operator=(const string& str)	関数	str を代入します。
string& operator=(const char* str)	関数	str を代入します。
string& operator=(char c)	関数	c を代入します。
iterator begin()	関数	文字列の先頭ポインタを求めます。
const_iterator begin() const		
iterator end()	関数	文字列の最後尾ポインタを求めます。
const_iterator end() const		

定義	種別	説明
size_t size() const	関数	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	関数	確保している領域のサイズを求めます。
void resize(size_t n, char c)	関数	格納可能な文字列の長さを n に変更します。
void resize(size_t n)	関数	格納可能な文字列の長さを n に変更します。
size_t capacity() const	関数	確保している領域のサイズを求めます。
void reserve(size_t res_arg = 0)	関数	領域の再割り当てを行います。
void clear()	関数	格納されている文字列を clear します。
bool empty() const	関数	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	関数	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)		
string& operator+=(const string& str)	関数	str の文字列を追加します。
string& operator+=(const char* str)	関数	str の文字列を追加します。
string& operator+=(char c)	関数	c の文字を追加します。
string& append(const string& str)	関数	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)	関数	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)	関数	文字列 str の n 文字分を追加します。
string& append(size_t n, char c)	関数	n 個の文字 c を追加します。

定義	種別	説明
string& assign(const string& str)	関数	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)	関数	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)	関数	文字列 str の n 文字分を代入します。
string& assign(size_t n, char c)	関数	n 個の文字 c を代入します。
string& insert(size_t pos1, const string& str)	関数	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)	関数	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* s, size_t n)	関数	pos の位置に文字列 s を n 文字分挿入します。
string& insert(size_t pos, const char* str)	関数	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)	関数	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())	関数	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)	関数	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)	関数	位置 pos から n 個分取り除きます。

定義	種別	説明
iterator erase(iterator position)	関数	position により参照された文字を取り除きます。
iterator erase(iterator first, iterator last)	関数	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)	関数	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)		
string& replace(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2)	関数	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)	関数	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)	関数	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)	関数	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)	関数	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。

定義	種別	説明
string& replace(iterator i1, iterator i2, size_t n, char c)	関数	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	関数	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)	関数	str の文字列と交換します。
const char* c_str() const	関数	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* data() const		
size_t find(const string& str, size_t pos=0) const	関数	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	関数	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。
size_t find(char c, size_t pos=0) const	関数	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	関数	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	関数	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	関数	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	関数	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)		

定義	種別	説明
size_t find_first_of(const char* str, size_t pos, size_t n) const	関数	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	関数	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	関数	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	関数	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	関数	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	関数	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)	関数	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	関数	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。
size_t find_last_not_of(const string& str, size_t pos=npos) const	関数	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	関数	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	関数	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。

定義	種別	説明
string substr(size_t pos=0, size_t n=npos) const	関数	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	関数	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	関数	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	関数	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	関数	str と比較します。
int compare(size_t pos1, size_t n1, const char* str, size_t n2=npos) const	関数	位置 pos1 から n1 文字分の文字列と str の sn2 文字分の文字列を比較します。

(1)　　string::string(void)

■機能

以下のように設定します。

s_ptr = 0;
s_len = 0;
s_res = 1;

■リターン値

なし

(2)　　string::string(const string& str, size_t pos=0, size_t n=npos)

■機能

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。

■リターン値

なし

(3) string::string(const char* str, size_t n)

■機能

以下に設定します。
s_ptr = str;
s_len = = n;
s_res = n+1;

■リターン値

なし

(4) string::string(const char* str)

■機能

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;

■リターン値

なし

(5) string::string(size_t n, char c)

■機能

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;

■リターン値

なし

(6) string::~string()

■機能

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。

■リターン値

なし

(7) string& string::operator=(const string& str)

■機能

str のデータを代入します。

■リターン値

*this

(8) string& sting::operator=(const char* str)

■機能

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

*this

(9) string& string::operator=(char c)

■機能

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

*this

(10) string::iterator string::begin()

string::const_iterator string::begin() const

■機能

文字列の先頭ポインタを求めます。

■リターン値

文字列の先頭ポインタ

(11) string::iterator sting::end()

string::const_iterator string::end() const

■機能

文字列の最後尾ポインタを求めます。

■リターン値

文字列の最後尾ポインタ

(12) size_t string::size() const

size_t string::length() const

■機能

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

格納されている文字列の文字列長

(13) size_t string::max_size() const

■機能

確保している領域のサイズを求めます。

■リターン値

確保している領域のサイズ

(14) void string::resize(size_t n, char c)

■機能

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。

■リターン値

なし

(15) void string::resize(size_t n)

■機能

オブジェクトが格納している文字列の長さを 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。

■リターン値

なし

(16) size_t string::capacity() const

■機能

確保している領域のサイズを求めます。

■リターン値

確保している領域のサイズ

(17) void string::reserve(size_t res_arg = 0)

■機能

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。

■リターン値

なし

(18) void string::clear()

■機能

格納されている文字列を clear します。

■リターン値

なし

(19) bool string::empty() const

■機能

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

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

(20) const char & 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)

■機能

s_ptr[pos]を参照します。

■リターン値

n< s_len の場合　：s_ptr [pos]
n>= s_len の場合：'¥0'

(21) string& string::operator+=(const string& str)

■機能

str が格納している文字列を追加します。

■リターン値

*this

(22) string& string::operator+=(const char* str)

■機能

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

*this

(23) string& string::operator+=(char c)

■機能

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

*this

(24) string& string::append(const string& str)

string& string::append(const char* str)

■機能

str の文字列をオブジェクトに追加します。

■リターン値

*this

(25) string& string::append(const string& str, size_t pos, size_t n);

■機能

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

*this

(26) string& string::append(const char* str, size_t n)

■機能

文字列 str の n 文字分を追加します。

■リターン値

*this

(27) string& string::append(size_t n, char c)

■機能

n 個の文字 c を追加します。

■リターン値

*this

(28) string& string::assign(const string& str)

string& string::assign(const char* str)

■機能

str の文字列を代入します。

■リターン値

*this

(29) string& string::assign(const string& str, size_t pos, size_t n)

■機能

位置 pos に文字列 str の n 文字分を代入します。

■リターン値

*this

(30) string& string::assign(const char* str, size_t n)

■機能

文字列 str の n 文字分を代入します。

■リターン値

*this

(31) string& string::assign(size_t n, char c)

■機能

n 個の文字 c を代入します。

■リターン値

*this

(32) string& string::insert(size_t pos1, const string& str)

■機能

位置 pos1 に str の文字列を挿入します。

■リターン値

*this

(33) string& string::insert(size_t pos1, const string& str, size_t pos2, size_t n)

■機能

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

*this

(34) string& string::insert(size_t pos, const char* str, size_t n)

■機能

pos の位置に文字列 str を n 文字分挿入します。

■リターン値

*this

(35) string& string::insert(size_t pos, const char* str)

■機能

pos の位置に文字列 str を挿入します。

■リターン値

*this

(36)　　string& string::insert(size_t pos, size_t n, char c)

■機能

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

*this

(37)　　string::iterator string::insert(iterator p, char c=char())

■機能

p が指す文字列の前に、文字 c を挿入します。

■リターン値

挿入された文字

(38)　　void string::insert(iterator p, size_t n, char c)

■機能

p で指す文字の前に、n 個の文字 c を挿入します。

■リターン値

なし

(39)　　string& string::erase(size_t pos=0, size_t n=npos)

■機能

位置 pos から n 個分取り除きます。

■リターン値

*this

(40)　　iterator string::erase(iterator position)

■機能

position により参照された文字を取り除きます。

■リターン値

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()

(41)　　iterator string::erase(iterator first, iterator last)

■機能

範囲[first, last]において文字を取り除きます。

■リターン値

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()

(42) string& string::replace(size_t pos1, size_t n1, const string& str)

string& string::replace(size_t pos1, size_t n1, const char* str)

■機能

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

*this

(43) string& string::replace(size_t pos, size_t n1, const string& str, size_t pos2, size_t n2)

■機能

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

*this

(44) string& string::replace(size_t pos, size_t n1, const char* str, size_t n2)

■機能

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

*this

(45) string& string::replace(size_t pos, size_t n1, size_t n2, char c)

■機能

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

*this

(46) string& string::replace(iterator i1, iterator i2, const string& str)

string& string::replace(iterator i1, iterator i2, const char* str)

■機能

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

*this

(47) string& string::replace(iterator i1, iterator i2, const char* str, size_t n)

■機能

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

*this

(48) string& string::replace(iterator i1, iterator i2, size_t n, char c)

■機能

位置 i1 から i2 までの文字を、n 個の文字 c で置き換えます。

■リターン値

*this

(49) size_t string::copy(char* str, size_t n, size_t pos=0) const

■機能

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

rlen

(50) void string::swap(string& str)

■機能

str の文字列と交換します。

■リターン値

なし

(51) const char* string::c_str() const

const char* string::data() const

■機能

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

s_ptr

(52) size_t sting::find(const string& str, size_t pos=0) const

size_t string::find(const char* str, size_t pos=0) const

■機能

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

文字列のオフセット

(53) size_t string::find(const char* str, size_t pos, size_t n) const

■機能

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

文字列のオフセット

(54) size_t string::find(char c, size_t pos=0) const

■機能

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

文字列のオフセット

(55) size_t string::rfind(const string& str, size_t pos=npos) const

size_t string::rfind(char *str, size_t pos=npos) const

■機能

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

文字列のオフセット

(56) size_t string::rfind(const char* str,size_t pos,size_t n) const

■機能

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

文字列のオフセット

(57) size_t string::rfind(char c,size_t pos=npos) const

■機能

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

文字列のオフセット

(58) 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

■機能

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

文字列のオフセット

(59) size_t string::find_first_of(const char* str, size_t pos, size_t n) const

■機能

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

文字列のオフセット

(60) size_t string::find_first_of(char c, size_t pos=0) const

■機能

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

文字列のオフセット

(61) 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

■機能

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

文字列のオフセット

(62) size_t string::find_last_of(const char* str, size_t pos, size_t n) const

■機能

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

文字列のオフセット

(63) size_t string::find_last_of(char c, size_t pos=npos) const

■機能

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

文字列のオフセット

(64) 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

■機能

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

文字列のオフセット

(65) size_t string::find_first_not_of(const char* str, size_t pos, size_t n) const

■機能

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

文字列のオフセット

(66) size_t string::find_first_not_of(char c, size_t pos=0) const

■機能

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

文字列のオフセット

(67) 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

■機能

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

文字列のオフセット

(68) size_t string::find_last_not_of(const char* str, size_t pos, size_t n) const

■機能

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

文字列のオフセット

(69) size_t string::find_last_not_of(char c, size_t pos=npos) const

■機能

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

文字列のオフセット

(70) string string::substr(size_t pos=0, size_t n=npos) const

■ 機能

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

範囲[pos,n]の文字列を持つオブジェクト

(71) int string::compare(const string& str) const

■ 機能

文字列と str の文字列を比較します。

■リターン値

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

(72)　　int string::compare(size_t pos1, size_t n1, const string& str) const

■　機能

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

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

(73)　　int string::compare(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2) const

■　機能

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

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

(74)　　int string::compare(const char* str) const

■　機能

str と比較します。

■リターン値

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

(75)　　int string::compare(size_t pos1, size_t n1, const char* str, size_t n2=npos) const

■　機能

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

文字列が同一の場合は：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

7.5.2 string クラスマニピュレータ

■定義名一覧

<table>
<tr><th>定義名</th><th>説明</th></tr>
<tr><td>string operator +(
const string &lhs,
const string &rhs)</td><td rowspan="5">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。</td></tr>
<tr><td>string operator +(
const char *lhs,
const string &rhs)</td></tr>
<tr><td>string operator +(
char lhs,
const string &rhs)</td></tr>
<tr><td>string operator +(
const string &lhs,
const char *rhs)</td></tr>
<tr><td>string operator +(
const string &lhs,
char rhs)</td></tr>
<tr><td>bool operator ==(
const string &lhs,
const string &rhs)</td><td rowspan="3">lhs の文字列と rhs の文字列を比較します。</td></tr>
<tr><td>bool operator ==(
const char *lhs,
const string &rhs)</td></tr>
<tr><td>bool operator ==(
const string &lhs,
const char *rhs)</td></tr>
<tr><td>bool operator !=(
const string &lhs,
const string &rhs)</td><td rowspan="3">lhs の文字列と rhs の文字列を比較します。</td></tr>
<tr><td>bool operator !=(
const char *lhs,
const string &rhs)</td></tr>
<tr><td>bool operator !=(
const string &lhs,
const char *rhs)</td></tr>
<tr><td>bool operator <(
const string &lhs,
const string &rhs)</td><td rowspan="3">lhs の文字列長と rhs の文字列長を比較します。</td></tr>
<tr><td>bool operator <(
const char *lhs,
const string &rhs)</td></tr>
<tr><td>bool operator <(
const string &lhs,
const char *rhs)</td></tr>
</table>

定義名	説明
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。
istream & getline(istream &is, string &str)	

(1) 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)

■機能

lhs の文字列（または文字）に rhs の文字列（または文字）を結合し、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

結合した文字列を格納するオブジェクト

(2) bool operator ==(const string &lhs, const string &rhs)

bool operator ==(const char *lhs, const string &rhs)

bool operator ==(const string &lhs, const char *rhs)

■機能

lhs の文字列と rhs の文字列を比較します。

■リターン値

文字列が同一の場合：TRUE
文字列が異なる場合：FALSE

(3) bool operator !=(const string &lhs, const string &rhs)

bool operator !=(const char *lhs, const string &rhs)

bool operator !=(const string &lhs, const char *rhs)

■機能

lhs の文字列と rhs の文字列を比較します。

■リターン値

文字列が同一の場合：FALSE
文字列が異なる場合：TRUE

(4) bool operator <(const string &lhs, const string &rhs)

bool operator <(const char *lhs, const string &rhs)

bool operator <(const string &lhs, const char *rhs)

■機能

lhs の文字列長と rhs の文字列長を比較します。

■リターン値

lhs.s_len < rhs.s_len の場合 ：TRUE
lhs.s_len >= rhs.s_len の場合 ：FALSE

(5) bool operator >(const string &lhs, const string &rhs)

bool operator >(const char *lhs, const string &rhs)

bool operator >(const string &lhs, const char *rhs)

■機能

lhs の文字列長と rhs の文字列長を比較します。

■リターン値

lhs.s_len > rhs.s_len の場合　：TRUE
lhs.s_len <= rhs.s_len の場合　：FALSE

(6) bool operator <=(const string &lhs, const string &rhs)

bool operator <=(const char *lhs, const string &rhs)

bool operator <=(const string &lhs, const char *rhs)

■機能

lhs の文字列長と rhs の文字列長を比較します。

■リターン値

lhs.s_len <= rhs.s_len の場合　：TRUE
lhs.s_len > rhs.s_len の場合　：FALSE

(7) bool operator >=(const string &lhs, const string &rhs)

bool operator >=(const char *lhs, const string &rhs)

bool operator >=(const string &lhs, const char *rhs)

■機能

lhs に格納された文字列長と rhs に格納された文字列長を比較します。

■リターン値

lhs.s_len >= rhs.s_len の場合　：TRUE
lhs.s_len < rhs.s_len の場合　：FALSE

(8) void swap(string &lhs,string &rhs)

■機能

lhs の文字列と rhs の文字列を交換します。

■リターン値

なし

(9) istream & operator >> (istream &is,string &str)

■機能

文字列を str に抽出します。

■リターン値

is

(10) ostream & operator << (ostream &os, const string &str)

■機能

文字列を挿入します。

■リターン値

os

(11) istream & getline(istream &is, string & str, char delim)

istream & getline(istream &is, string &str)

■機能

is から文字列を抽出し， str に付加します。

■リターン値

is

8. システム組み込み

8.1 システム組み込みの概要

本章では、C/C++プログラムを 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。
C/C++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。

（1） メモリの割付け
C/C++プログラムの各セクション、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。

（2） プログラム実行環境の設定
C/C++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。

第 2 節では C/C++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リンケージエディタのコマンド指定方法について実例を挙げて説明します。

第 3 節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。

第 4 節ではライブラリ関数の初期設定処理と、低水準ルーチンの作成方法を説明します。

8.2 メモリ領域の割り付け

コンパイラの出力した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、C/C++プログラムの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。
C/C++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機械語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。

8.2.1 静的領域の割り付け

(1) 静的領域の内容

オブジェクト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++初期処理データ領域、C++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。

(2) サイズの算出法

静的領域のサイズは、コンパイ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、リンケージマップリスト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。
リンクまえに静的領域のサイズを概算する場合は、コンパイルリストの統計情報にセクションごとのサイズを出力しますので、これに基づいてサイズを算出することができます。図 8.1 に統計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 8.1 統計情報例

標準ライブラリを使用しない場合は、コンパイル単位ごとに出力する統計情報のセクションごとのサイズの合計が静的領域のサイズになります。
また、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を加えなければなりません。静的領域サイズは、必ずリンケージマップリストで確認してください。
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。

また、実行時ルーチンのサイズも C ライブラリ関数や C++クラスライブラリと同じようにメモリ領域サイズに加える必要があります。

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。

■例

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
```

(3) ROM、RAM の割り付け

C/C++プログラムを ROM 化する場合は、静的な領域を以下のように 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 （(4)参照）
- 関数アドレス領域　（セクション$INDIRECT）　→ ROM
- 初期化データセクションアドレス領域(セクション C$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
- 後処理データ領域[*1](セクション C$END)　→ ROM
- 仮想関数表領域[*2](セクション C$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。

(4) 初期化データ領域の割り付け

初期化データ領域は初期値を持ったデータ領域ですが、値の変更が可能なので、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、初期化データの領域については、ROM 上と RAM 上に、二重に領域をとる必要があります。

ただし、初期値を指定した静的変数を変更しないようにプログラムを作成すれば、初期化データの領域は ROM 上に置くだけでよく、二重に割り付ける必要はありません。

(5) 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、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ごと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 8.2 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。

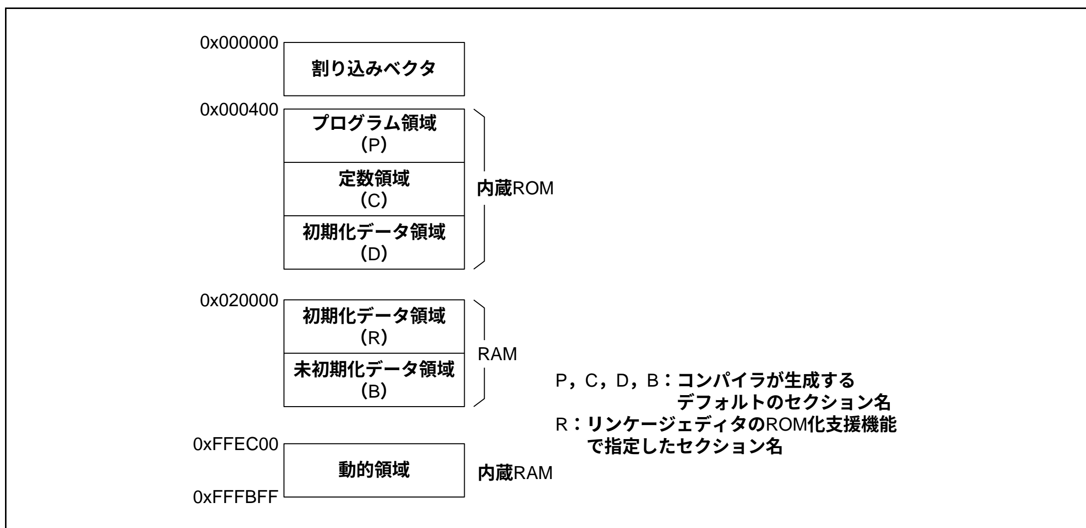


図 8.2　**静的な領域の割り付け例**

図 8.2 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。

8.2.2 動的領域の割り付け

(1) 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）

(2) サイズの算出法

(a) スタック領域

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

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。

■例

次の C プログラムに対するシンボル割り付け情報とスタック使用量の算出法を示します。ここでは、H8S/2600 用アドバンストモードの場合について説明します。

```
extern int h(char, char *, double);
int
h(char a, register char *b, double c)
{
    char   *d;

    d= &a;
    h(*d,b,c);
    {
       register int i;

       i= *d;
       return i;
     }
}

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

FILE NAME: m0280.c

Function (File m0280.c   , Line    3): h

  Parameter Allocation
   a                              0xfffffff7    saved from R0L
   b                          REG ER5           saved from ER1
   c                              0x00000008

  Level 1(File m0280.c , Line  4) Automatic/Register Variable Allocation
   d                              0xfffffff2
```

```
  Leve1 2(File m0280.c , Line  9) 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)
 Temporary Size                : 0x00000000 Byte(s)
 Register Save Area Size       : 0x00000008 Byte(s)
 Total Frame Size              : 0x0000001e Byte(s)
```

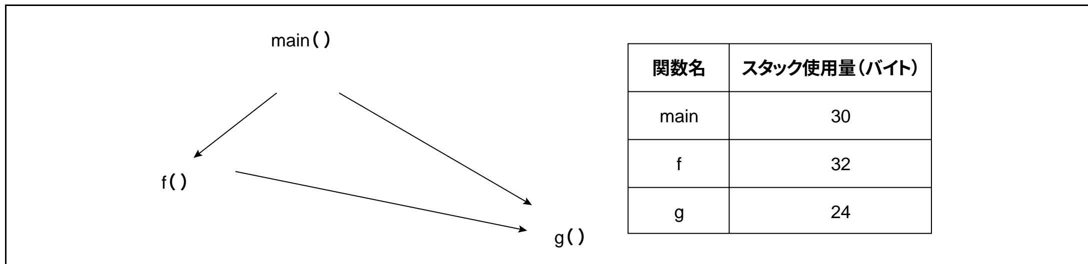
関数の使用するスタック領域サイズは、Total Frame Size の 0xle つまり 30 バイトとなります。

スタック使用量の算出法

関数の呼び出し関係から使用するスタック領域のサイズを算出します。

■例

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** 8.3 に示します。



関数名	スタック使用量（バイト）
main	30
f	32
g	24

図 8.3 **関数呼び出しの関係とスタック使用量の例**

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** 8.1 によって計算します。

表 8.1 **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（30）→f（32）→g（24）	86	**←スタック使用量（最大値）**
main（30）→g（24）	54	

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 86 バイト）のスタック領域を最低限割り付ける必要があります。

標準ライブラリのライブラリ関数を使用する場合には、ライブラリ関数を含めたスタック領域のサイズを計算する必要があります。ライブラリ関数の使用するスタック領域のサイズについては、製品添付の**「標準ライブラリのスタック使用量一覧」**を参照してください。

注意 C/C++**プログラムの中で再帰呼び出しを行っている場合は、再帰的に呼び出す回数の最大値を算出してから、その関数のスタック領域のサイズに再帰的に呼び出す回数を掛けて計算してください。**

(b)　ヒープ領域

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 6 バイト（cpu＝2600n、cpu＝2000n、cpu＝300hn、cpu＝300 指定時）または 12 バイト（cpu＝2600a、cpu＝2000a、cpu＝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**「8.4（4）C/C++ライブラリ関数の初期設定」**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＝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＝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、delete **関数で解放した領域は、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。**

(3)　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初期設定（INIT）でスタック領域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。

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。

それぞれ、**「8.3（2）初期設定（INIT）」**、**「8.4（6）低水準インタフェースルーチン」**を参照してください。

8.3 実行環境の設定

本節では、C/C++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。

ここでは、プログラムの実行環境の最も基本的な構成として、C/C++ライブラリ関数を使用しない場合について説明します。

C/C++ライブラリ関数を使用する場合は、**「8.4 ライブラリ関数使用時の実行環境の設定」**を参照してください。

図 8.4 にプログラムの構成例を示します。

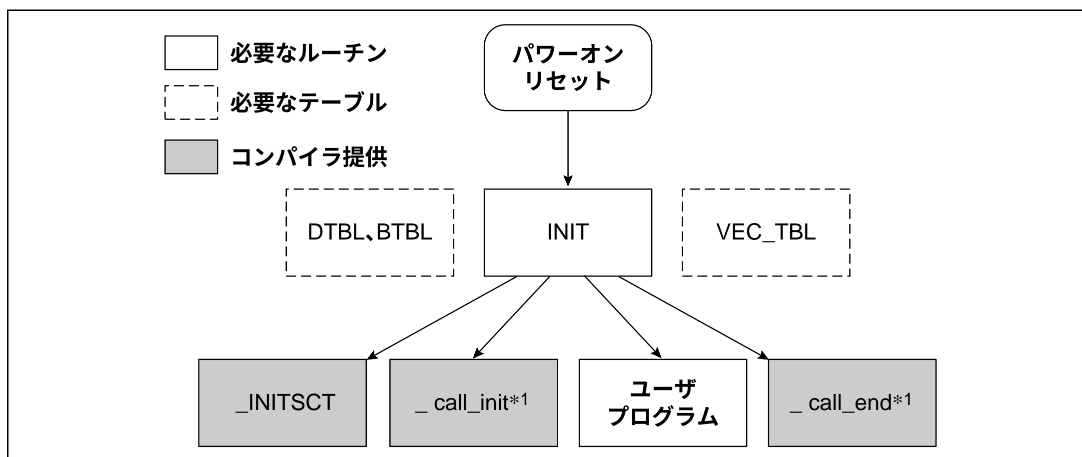


図 8.4 プログラムの構成例（C/C++ライブラリ関数を使用しない場合）

【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。

（1） ベクタテーブルの設定（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（INIT）が起動されるように、ベクタテーブルを設定します。

（2） 初期設定（INIT）
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。

（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。

（4） 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。

（5） 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。

（6） 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。

【注】 *1: **標準ライブラリとして提供しています。**

*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。**

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、レジスタの初期設定を行う関数「INIT」が呼び出されるようにするためには、ベクタテーブルの0番地に関数「INIT」の先頭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の設定もここで行います。以下にコーディング例を示します。

■例（C プログラム例）

```
#pragma interrupt IRQ0
extern void INIT(void);
extern void IRQ0(void);

#pragma section vect1          /* #pragma section 宣言により vec_table1 を*/
                               /*                Cvect1 セクションに出力します。*/
                               /* リンク時に,start オプションで Cvect1 セクション*/
                               /* を 0 番地に割り付けるよう指定します。   */

const void (*const vec_table1[])(void)={INIT};
#pragma section vect2          /* #pragma section 宣言により vec_table2 を*/
                               /*                Cvect2 セクションに出力します。*/
                               /* リンク時に,start アドレスで Cvect2 セクションを*/
                               /*       所定番地に割り付けるよう指定します。*/

const void (*const vec_table2[])(void)={IRQ0};
                               /* 空いている番地に indirect オプション   */
                               /*     #pragma   indirect 指定で       */
                               /*     作成されたアドレステーブルを割り付けます。*/
```

(2) 初期設定（INIT）

ここでは、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。初期設定（INIT）関数は、#pragma entry を用いてエントリ関数として宣言します。

C++コンパイル時、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。

_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。

以下にコーディング例を示します。

■例

```
#include <machine.h>           /* 組み込み関数を使用するために machine.h をインクルード */
#pragma stacksize 0x800        /* 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                                   */
#ifdef __cplusplus             /* C++プログラムで初期設定ルーチンを C リンケージ宣言    */
extern "C"{
#endif
```

```
void INIT(void);              /* INIT 関数の宣言                                    */
void _INITSCT(void);
#ifdef __cplusplus
}
#endif
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       */
   set_imask_ccr(0);          /*    ANDC.B #127,CCR        ;割り込みマスク設定        */
   _INITSCT();                /*    JSR    @__INITSCT   ;セクション領域の初期化     */
/*  call_init(); C++プログラム中にグローバルクラスオブジェクトが存在する場合に呼び出します */
   main();                    /*    JSR    @_main          ;main 関数呼び出し         */
/*  call_end();  C++プログラム中にグローバルクラスオブジェクトが存在する場合に呼び出します */
   sleep();                   /*    SLEEP                  ;低消費電力状態で待機      */
}
```

(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は C$BSEC、初期化データ領域は C$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 $DSEC
static const struct {
   void * rom_s;              /* 初期化データセクションの ROM 上の先頭アドレス */
   void * rom_e;              /* 初期化データセクションの ROM 上の最終アドレス */
   void * ram_s;              /* 初期化データセクションの RAM 上の先頭アドレス */
}DTBL[]={
   {__sectop (“D”), __secend (“D”), __sectop (“R”)},
   {__sectop (“$ABS8D”), __secend (“$ABS8D”), __sectop (“$ABS8R”)},
   {__sectop (“$ABS16D”), __secend (“$ABS16D”), __sectop (“$ABS16R”)}
};
#pragma section $BSEC
static const struct {
   void * b_s;                /* 未初期化データセクションの先頭アドレス */
   void * b_e;                /* 未初期化データセクションの最終アドレス */
}BTBL[]={
   {__sectop (“B”), __secend (“B”)},
   {__sectop (“$ABS8B”), __secend (“$ABS8B”)},
   {__sectop (“$ABS16B”), __secend (“$ABS16B”)}
};
```

8.4 ライブラリ関数使用時の実行環境の設定

ライブラリ関数をご使用になる場合は、プログラムの実行環境の設定としてライブラリ関数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。

本節では、C/C++ライブラリ関数使用時のプログラムの実行環境の設定方法について説明します。

図 8.5 にプログラムの構成例を示します。

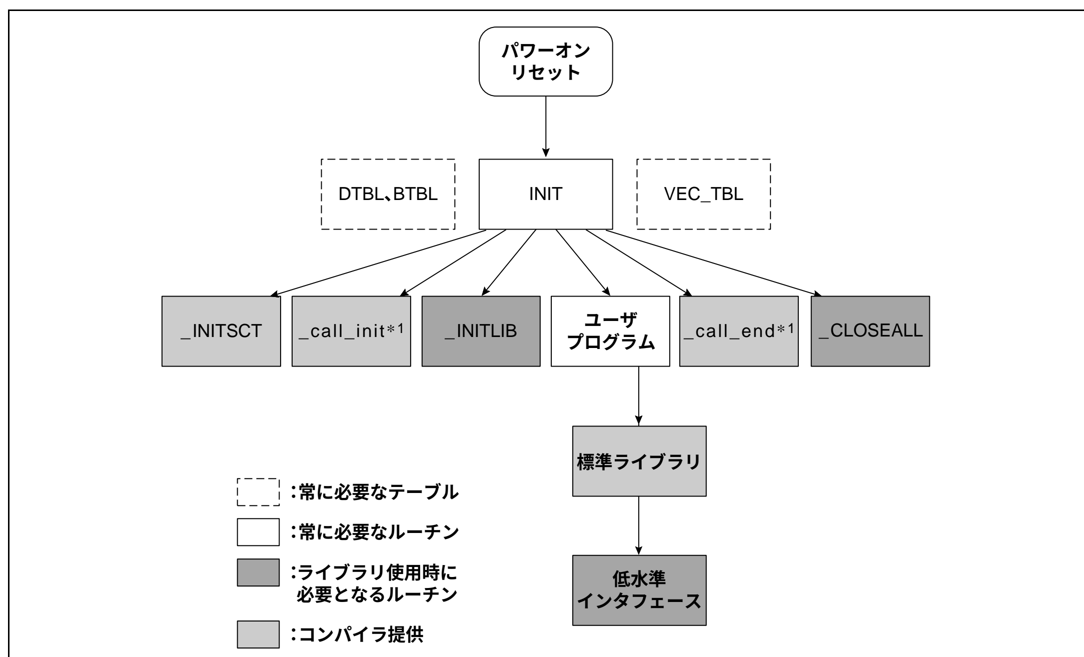


図 8.5　**プログラムの構成例**（C/C++**ライブラリ関数を使用する場合**）

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

ライブラリ使用時に必要な各構成ルーチンの内容は以下のとおりです。

（1）　ベクタテーブルの設定（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（INIT）が起動されるように、ベクタテーブルを設定します。

（2）　初期設定（INIT）
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。

（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。

（4）　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。

（5） 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。
（6） 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
（7） C/C++ライブラリ関数の初期設定（_INITLIB）
C/C++ライブラリ関数の中で、初期設定の必要なものについて、初期設定を行います。特に、標準入出力を行う準備をします。
（8） 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。
（9） 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。

【注】 *1: **標準ライブラリとして提供しています。**
*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。**

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。

注意 プログラムの終了処理を行う C ライブラリ関数 exit、onexit、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。具体的なプログラム例を「付録 D　終了処理関数の作成例」に示します。**
なお、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。**

---

(1) ベクタテーブルの設定（VEC_TBL）

ベクタテーブルは C/C++ライブラリ関数を使用しない場合と同じです。**「7.3 実行環境の設定」**を参照してください。

(2) 初期設定（INIT）

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>          /* 組み込み関数を使用するために machine.h をインクルード */
#pragma stacksize 0x800       /* 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                                         */
#ifdef __cplusplus            /* C++プログラム時、初期設定ルーチンを C リンケージ宣言    */
extern “C”{
#endif
void INIT(void);              /* INIT 関数の宣言                                         */
void _INITSCT(void);
void _INITLIB(void);          /* _INITLIB 関数の宣言                                     */
void _CLOSEALL(void);         /* _CLOSEALL 関数の宣言                                    */
#ifdef __cplusplus
}
#endif
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          */
   set_imask_ccr(0);          /*     ANDC.B #127,CCR        ;割り込みマスク設定          */
   _INITSCT();                /*     JSR    @__INITSCT    ;セクション領域の初期化        */
   _INITLIB();                /*     JSR    @__INITLIB    ;ライブラリの初期化            */
/* _call_init();     C++プログラム中に静的データが存在する場合に呼び出します             */
   main();                    /*     JSR    @_main         ;main 関数呼び出し            */
   _CLOSEALL();               /*     JSR    @__CLOSEALL   ;ファイルのクローズ            */
/* _call_end();      C++プログラム中に静的データが存在する場合に呼び出します             */
   sleep();                   /*     SLEEP                  ;低消費電力状態で待機        */
}
```

(3) セクション初期化用テーブル（DTBL、BTBL）

セクション初期化用テーブルは C/C++ライブラリ関数を使用しない場合と同じです。**「8.3 実行環境の設定」**を参照してください。

(4) C/C++ライブラリ関数の初期設定（_INITLIB）

C/C++ライブラリ関数の中には、初期設定が必要な関数があります。それらの関数を使用する場合、当然使用するまえに定められた初期設定を行わなければなりません。本項では、プログラム起動ルーチンの中の「_INITLIB」の中で初期設定を行う場合を例にとり説明します。

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。

- ライブラリのエラー状態を示す「errno」の初期設定はすべてのライブラリ関数共通に必要です。
- <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。

■例

```
#include <stdlib.h>
const size_t _sbrk_size=516 ; *1    /* ヒープ領域確保サイズの最小単位を指定 */
#ifdef __cplusplus
extern “C” {
#endif
void _INITLIB(void);
void _INIT_LOWLEVEL(void) ;
void _INIT_IOLIB(void) ;
void _INIT_OTHERLIB(void) ;
#ifdef __cplusplus
}
#endif

void _INITLIB(void)                 /* アセンブリルーチンのシンボル名から下線を一つ削除 */
 {
   errno=0;                         /* ライブラリ関数共通の初期設定 */

   _INIT_LOWLEVEL();                /* 低水準インタフェースの初期設定ルーチンの呼び出し*/
   _INIT_IOLIB();                   /* 標準入出力の初期設定ルーチンの呼び出し */
   _INIT_OTHERLIB();                /* 標準入出力以外の初期設定ルーチンの呼び出し */
}
```

以下、標準入出力の初期設定ルーチン（_INIT_IOLIB）、標準入出力以外の初期設定ルーチン（_INIT_OTHERLIB）の作成例を示します。低水準インタフェースルーチンの初期設定ルーチン（_INIT_LOWLEVEL）は、ユーザ作成の低水準インタフェースルーチンの仕様に合わせて作成してください。

【注】 *1 **コンパイラがヒープ領域として確保するメモリサイズの最小単位を指定します。指定がない場合は、**1028 **バイト（アドバンストモード時は** 1032 **バイト）とみなします。**
一回に確保するメモリサイズは、必ず偶数を指定してください。奇数を指定した場合の動作は保証しません。
確保サイズは、実際に使用するメモリサイズ＋管理領域サイズを目安にしてください。
管理領域サイズは、cpu=300，300hn，2000n，2600n **のとき** 6 **バイト、**cpu=300ha，2000a，2600a **のとき** 12 **バイトです。**

(a) 標準入出力の初期設定ルーチン（_INIT_IOLIB）の作成例

標準入出力の初期設定では、ファイルを参照するために必要な FILE 型データ（**図 8.6**）の初期設定と標準入出力ファイルのオープンを行います。FILE 型データの初期設定は、必ず標準入出力ファイルのオープンの前に行ってください。

標準入出力の初期設定を行うプログラム例を示します。

■例

```
#include <stdio.h>
#ifdef __cplusplus
#include <ios>
int ios::Init::init_cnt;
#endif

#define N 4 *4
const int _nfiles =N; *4
struct _iobuf _iob[N]; *4
unsigned char sml_buf[N]; *4

#ifdef __cplusplus
extern "C"
#endif
void _INIT_IOLIB(void)
 {
        FILE *fp;
                    /* FILE 型データの初期設定 */
        for (fp=_iob; fp<_iob+_nfiles; fp++) {
                    fp -> _bufptr=NULL ;         /* バッファポインタのクリア */
                    fp -> _bufcnt=0;             /* バッファカウンタのクリア */
                    fp -> _buflen=0;             /* バッファ長のクリア*/
                    fp -> _bufbase=NULL ;        /* ベースポインタのクリア*/
                    fp -> _ioflag1=0;            /* I/O フラグのクリア*/
                    fp -> _ioflag2=0;
                    fp -> _iofd=0;
        }
                    /* 標準入出力ファイルのオープン */
        if (freopen("stdin"*1, "r", stdin)==NULL)  /* 標準入力ファイルのオープン */
                    stdin->_ioflag1=0xff;        /* ファイルアクセスの禁止 *2 */
        stdin->_ioflag1 |= _IOUNBUF;             /* データのバッファリング無 *3 */

        if (freopen("stdout"*1,"w",stdout)==NULL)  /* 標準出力ファイルのオープン */
                    stdout->_ioflag1=0xff;
        stdout->_ioflag1 |= _IOUNBUF;
        if (freopen("stderr"*1,"w",stderr)==NULL)  /* 標準エラーファイルのオープン*/
                    stderr->_ioflag1=0xff;
        stderr->_ioflag1 |= _IOUNBUF;
}
```

【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 **ファイルのオープンが失敗した場合、ファイルアクセス禁止のフラグを立てます。**

*3 **コンソール等の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。**

*4 **入出力ファイル数を指定します。指定がない場合は、ファイル数は 20 個とみなします。**

注意 マルチリンケージ処理を使用してロードモジュールを作成するとき、すでにライブラリをリンク済のリロケータブルロードモジュールに対して、上記オブジェクトファイルのリンクを指定すると、二重定義エラーになります。ライブラリをリンクする前に、必ず上記オブジェクトファイルをリンクしてください。

```
/* 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 8.6 FILE 型データ

(b) 標準入出力以外の初期設定ルーチン（_INIT_OTHERLIB）の作成例

標準入出力以外で初期設定が必要な C/C++ライブラリ関数の初期設定用プログラム例を以下に示します。

■例

```
#include <stddef.h>

extern char *_s1ptr;
#ifdef __cplusplus
extern "C" {
#endif
void srand(unsigned int) ;
void _INIT_OTHERLIB(void);
#ifdef __cplusplus
}
#endif
void _INIT_OTHERLIB(void)
{
   srand(1);                 /* rand 関数を使用する場合の初期値の設定 */
   _s1ptr=NULL;              /* strtok 関数で使用するポインタの初期設定 */
}
```

(5) 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。

■例

```
#include <stdio.h>
#ifdef __cplusplus
extern “C”
#endif
void _CLOSEALL( )    /* アセンブリルーチンのシンボル名から下線を一つ削除 */
{
         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のチェック */
                    if(_iob[i]._ioflag1 & ( _IOREAD|_IOWRITE|_IORW))

                    /* オープンしているファイルのクローズ */
                              fclose(&_iob[i]);
}
```

(6) 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** 8.2 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 8.2　低水準インタフェースルーチンの一覧

No.	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

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、**「8.4（4）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。また、H8S，H8/300 シリーズシミュレータ・デバッガ上で実行する低水準インタフェースルーチン例を**「付録 E.　低水準インタフェースルーチンの作成例」**に示しますので、合わせて参照してください。

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。

■凡例

<table>
<tr><td colspan="5">（ルーチン名）</td></tr>
<tr><td>機能</td><td colspan="4">（ルーチンの機能概要を示します。）</td></tr>
<tr><td>インタフェース</td><td colspan="4">（ルーチンの C プログラムの関数としての宣言方法を示します。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。）</td></tr>
<tr><td rowspan="3">引数</td><td>No.</td><td>名前</td><td>型</td><td>意味</td></tr>
<tr><td>1</td><td>（インタフェースに示す引数名です。）</td><td>（引数の型を示します。）</td><td>（引数として渡される値の意味を示します。）</td></tr>
<tr><td></td><td></td><td></td><td>・</td></tr>
<tr><td rowspan="3">リターン値</td><td>型</td><td colspan="3">（リターン値の型を示します。）</td></tr>
<tr><td>正常</td><td colspan="3">（正常に終了した場合のリターン値の意味を示します。）</td></tr>
<tr><td>異常</td><td colspan="3">（エラーが生じた場合のリターン値を示します。）</td></tr>
</table>

<table>
<tr><td colspan="5">(a)　open ルーチン</td></tr>
<tr><td>機能</td><td colspan="4">ファイルをオープンします。</td></tr>
<tr><td>インタフェース</td><td colspan="4"><code>int open (char *name,
　　　　　int mode,
　　　　　int flg);</code></td></tr>
<tr><td rowspan="4">引数</td><td>No.</td><td>名前</td><td>型</td><td>意味</td></tr>
<tr><td>1</td><td>name</td><td>char 型を指すポインタ</td><td>ファイルのファイル名を指す文字</td></tr>
<tr><td>2</td><td>mode</td><td>int</td><td>ファイルをオープンするときの処理の指定</td></tr>
<tr><td>3</td><td>flg</td><td>int</td><td>ファイルをオープンするときの処理の指定（常に 0777）</td></tr>
<tr><td rowspan="3">リターン値</td><td>型</td><td colspan="3">int</td></tr>
<tr><td>正常</td><td colspan="3">正常オープンしたファイルのファイル番号</td></tr>
<tr><td>異常</td><td colspan="3">−1</td></tr>
</table>

■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参照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。

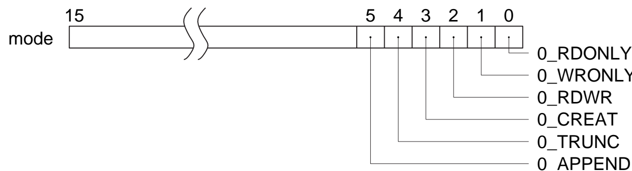


表 7.3　open ルーチン mode ビット説明

O_RDONLY（0 ビット）	このビットが 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）	このビットが 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）	このビットが 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）	このビットが 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）	このビットが 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。

<table>
<tr><td colspan="5">（b）　close ルーチン</td></tr>
<tr><td>機能</td><td colspan="4">ファイルをクローズします。</td></tr>
<tr><td>インタフェース</td><td colspan="4"><code>int close (int fileno);</code></td></tr>
<tr><td rowspan="2">引数</td><td>No.</td><td>名前</td><td>型</td><td>意味</td></tr>
<tr><td>1</td><td>fileno</td><td>int</td><td>クローズするファイル番号</td></tr>
<tr><td rowspan="3">リターン値</td><td>型</td><td colspan="3">int</td></tr>
<tr><td>正常</td><td colspan="3">0</td></tr>
<tr><td>異常</td><td colspan="3">－1</td></tr>
</table>

■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。

<table>
<tr><td colspan="5">（c） read **ルーチン**</td></tr>
<tr><td>**機能**</td><td colspan="4">**ファイルからのデータの読み込みを行います。**</td></tr>
<tr><td>**インタフェース**</td><td colspan="4">int read (int fileno,
          char *buf,
          unsigned int count);</td></tr>
<tr><td rowspan="4">**引数**</td><td>No.</td><td>**名前**</td><td>**型**</td><td>**意味**</td></tr>
<tr><td>1</td><td>fileno</td><td>int</td><td>**読み込みの対象となるファイル番号**</td></tr>
<tr><td>2</td><td>buf</td><td>char **型を指すポインタ**</td><td>**読み込んだデータを設定する領域**</td></tr>
<tr><td>3</td><td>count</td><td>unsigned int</td><td>**読み込むバイト数**</td></tr>
<tr><td rowspan="3">**リターン値**</td><td>**型**</td><td colspan="3">int</td></tr>
<tr><td>**正常**</td><td colspan="3">**実際に読み込まれたバイト数**</td></tr>
<tr><td>**異常**</td><td colspan="3">－1</td></tr>
</table>

■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。

<table>
<tr><td colspan="5">（d） write ルーチン</td></tr>
<tr><td>機能</td><td colspan="4">ファイルへのデータの書き出しを行います。</td></tr>
<tr><td>インタフェース</td><td colspan="4"><code>int write (int fileno,</code>
<code>char *buf,</code>
<code>unsigned int count);</code></td></tr>
<tr><td rowspan="4">引数</td><td>No.</td><td>名前</td><td>型</td><td>意味</td></tr>
<tr><td>1</td><td>fileno</td><td>int</td><td>書き出しの対象となるファイル番号</td></tr>
<tr><td>2</td><td>buf</td><td>char 型を指すポインタ</td><td>書き出すデータの領域</td></tr>
<tr><td>3</td><td>count</td><td>unsigned int</td><td>書き出すバイト数</td></tr>
<tr><td rowspan="3">リターン値</td><td>型</td><td colspan="3">int</td></tr>
<tr><td>正常</td><td colspan="3">実際に書き出されたバイト数</td></tr>
<tr><td>異常</td><td colspan="3">－1</td></tr>
</table>

■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）が満杯の時は、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。

<table>
<tr><td colspan="5">（e） lseek ルーチン</td></tr>
<tr><td>機能</td><td colspan="4">ファイルの読み込み／書き出しの位置を設定します</td></tr>
<tr><td>インタフェース</td><td colspan="4">int lseek (int fileno,
long offset,
int base);</td></tr>
<tr><td rowspan="4">引数</td><td>No.</td><td>名前</td><td>型</td><td>意味</td></tr>
<tr><td>1</td><td>fileno</td><td>int</td><td>対象となるファイル番号</td></tr>
<tr><td>2</td><td>offset</td><td>long</td><td>読み込み／書き出しの位置を示すオフセット（バイト単位）</td></tr>
<tr><td>3</td><td>base</td><td>int</td><td>オフセットの起点</td></tr>
<tr><td rowspan="3">リターン値</td><td>型</td><td colspan="3">long</td></tr>
<tr><td>正常</td><td colspan="3">新しいファイルの読み込み／書き出しの位置のファイルの先頭からのオフセット（バイト単位）</td></tr>
<tr><td>異常</td><td colspan="3">－1</td></tr>
</table>

■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。

9. モジュール間最適化のオプション・環境変数

9.1 コマンドラインの形式

モジュール間最適化ツールを起動するコマンドラインの形式は次のとおりです。

```
optlnk38［△<オプション>...］
    <オプション>:<オプション>［= <パラメタ>［, <パラメタ>. . . ］］
```

9.2 サブコマンドファイルの書式

サブコマンドファイルの書式は次のとおりです。

```
<サブコマンド>△［<パラメタ>［,<パラメタ>. . . ］］［△;<コメント>]
```

注意 1. **パラメタがファイル名の場合は、パラメタの区切りに空白を指定できます。**
2. **サブコマンドを１行に記述できない場合は、&を用いて継続指定できます。**
3. **コメントの開始を示すセミコロン(;)と、サブコマンドまたはパラメタの間は必ず１個以上の空白が必要です。**

ユニット名として、ソースファイル名称を指定します。アセンブリソースファイルの場合は、アセンブル時に任意のユニット名を指定することができます。

9.3 最適化機能オプション／サブコマンド

オプション／サブコマンドと短縮形および省略時解釈の一覧を**表 9.1** に示します。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。

また日立統合化開発環境の対応するダイアログメニューをタブ名［項目］で示します。ダイアログメニューの詳細は、日立統合化開発環境のユーザーズマニュアルを参照してください。

表 9.1 最適化機能オプション／サブコマンド一覧

<table>
<tr><th>No</th><th>項目</th><th>オプション／
サブコマンド名</th><th>パラメタ</th><th>ダイアログメニュー</th><th>指定内容</th></tr>
<tr><td rowspan="2">1</td><td rowspan="3">最適化
内容の
指定</td><td>OPtimize</td><td>
STring_unify
| SYmbol_delete
| Variable_access

| Register
| SAMe_code
| Function_call

| Branch
| SPeed

| SAFe</td><td>Optimize[Optimize]
[Unify Strings]
[Eliminate Dead Code]
[Use Short Addressing]

[Reallocate Registers]
[Eliminate Same Code]
[Use Indirect Call/Jump]

[Optimize Branches]
[Speed]

[Safe]</td><td>
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の活用
レジスタの再割付
共通コードの統合
間接アドレッシングモードの活用
分岐命令の最適化
スピード重視の最適化(op=st,sy,v,r,b)
安全な最適化(op=st,r,b)</td></tr>
<tr><td>NOOPtimize</td><td>—</td><td>Optimize[Optimize]</td><td>最適化の抑止指定</td></tr>
<tr><td>2</td><td>SAMESize</td><td><size> (省略時：1E)
size:16 進数</td><td>Optimize
[Eliminated Size]</td><td>共通コード統合の対象となるサイズの指定</td></tr>
<tr><td>3</td><td rowspan="5">最適化
抑止</td><td>SYmbol_forbid</td><td><シンボル名>
[,<シンボル名>...]</td><td>Optimize[Forbid Item]
[Forbid Elimination of Dead Code to]</td><td>未参照シンボル削除の最適化を抑止する変数／関数名を指定</td></tr>
<tr><td>4</td><td>SAMECode_forbid</td><td><関数名>
[,<関数名>...]</td><td>Optimize[Forbid Item]
[Forbid Elimination of Same Code to]</td><td>共通コード統合の最適化を抑止する関数名を指定</td></tr>
<tr><td>5</td><td>Variable_forbid</td><td><変数名>
[,<変数名>...]</td><td>Optimize[Forbid Item]
[Forbid Use of Short Addressing to]</td><td>短絶対アドレッシングモード活用の最適化を抑止する変数名を指定</td></tr>
<tr><td>6</td><td>Function_forbid</td><td><関数名>
[,<関数名>...]</td><td>Optimize[Forbid Item]
[Forbid Use of Indirect Call/Jump to]</td><td>間接アドレッシングモード活用の最適化を抑止する関数名を指定</td></tr>
<tr><td>7</td><td>ABSolute_forbid</td><td><addr>[+<size>]
[,<addr>[+<size>]...]
addr,size:16 進数</td><td>Optimize[Forbid Item]
[Forbid Memory Allocation in]</td><td>アドレス割付の対象外となるアドレス領域を指定</td></tr>
<tr><td>8</td><td rowspan="3">最適化
情報</td><td>Mlist</td><td><ファイル名></td><td>List [Optimization File]</td><td>最適化情報リストの出力</td></tr>
<tr><td>9</td><td>SHow</td><td>
Symbol
| Reference</td><td>List [Optimization File]
[Symbol]
[Reference]</td><td>リスト出力内容指定
シンボル最適化情報出力
シンボル参照回数出力</td></tr>
<tr><td>10</td><td>INFormation</td><td>—</td><td>Optimize
[Output Information]</td><td>最適化された関数名を画面表示</td></tr>
<tr><td>11</td><td>サブコマンドファイル</td><td>SUbcommand</td><td><ファイル名></td><td>—
[Use External Subcommand File]</td><td>サブコマンドファイルの指定</td></tr>
</table>

(1) 最適化の指定

■オプション／サブコマンド形式

オプション　：OPtimize[=<パラメタ>[,<パラメタ>...]]
　　　　　　　NOOPtimize
サブコマンド：OPtimize[△<パラメタ>[,<パラメタ>...]]
　　　　　　　NOOPtimize
パラメタ　　：STring_unify | SYmbol_delete | Variable_access |
　　　　　　　Register |SAMe_code | Function_call | Branch | SPeed | SAFe

■ダイアログメニュー

Optimize[Optimize]

■説明

optimize 指定時、モジュール間最適化を実行します。また、パラメタを指定することにより、最適化内容を指定することができます。各パラメタの内容を**表 9.2** に示します。
nooptimize 指定時、モジュール間最適化を実行せずに、リンケージ処理のみ行います。

注意 optimize **オプション／サブコマンドのパラメタは、指定されたパラメタの論理和が有効となります。例えば、**optimize=speed,same_code **が指定された場合、**optimize= string_unify,symbol_delete,variable_access, register,branch,same_code **が有効になります。すなわち、**optimize=function_call **以外の全ての最適化を実施します。**

表 9.2　optimize **オプション／サブコマンドのパラメター覧**

パラメタ	説明	最適化対象
パラメタなし	全ての最適化を実行します。optimize=string_unify, symbol_delete, variable_access, register, same_code, function_call, branch を指定した時と同じです。	ー
string_unify	const 属性を持つ定数／文字列に対し、同一値定数および同一文字列の統合を、モジュール間にわたって実施します。 const 属性を持つ定数／文字列には、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	コンパイラ出力オブジェクトのみ対象
symbol_delete	一度も参照のない変数／関数を削除します。この最適化を指定する場合は、必ず entry サブコマンドを指定してください。	コンパイラ出力オブジェクトのみ対象
variable_access	8 ビットおよび 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空きがあれば、アクセス回数の多い変数を割り当て、当該変数のアクセスコード最適化を行います。	変数：C/C++定義変数 対象アクセスコード： 全てのオブジェクトプログラム
register	関数の呼出関係を解析し、冗長なレジスタ退避・回復コードを削除します。また、呼出前後のレジスタ使用状況により、使用レジスタ番号を変更することもあります。	コンパイラ出力オブジェクトのみ対象
same_code	複数の同一命令列をサブルーチン化して、コードサイズを削減します。	コンパイラ出力オブジェクトのみ対象
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る最適化を行います。	全てのオブジェクトプログラムを対象
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。また、他の最適化項目をひとつでも実行すると、本最適化は指定の有無に関わらず、必ず実行します。	全てのオブジェクトプログラムを対象

パラメタ	説明	最適化対象
speed	最適化項目のうち、同一命令列のサブルーチン化のような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施します。optimize= string_unify, symbol_delete, variable_access, register,branch を指定したときと同じ効果になります。	―
safe	メモリ割り付け位置が固定でなければならない変数や、スピードを優先したい関数など、部分的に最適化を抑止したい場合があります。optimize=safe は、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施します。 optimize=safe は、optimize=string_unify, register, branch を指定したときと同じ効果になります。	―

(2) 共通コード最適化サイズの指定

■オプション／サブコマンド形式

オプション　：　SAMESize=<パラメタ>

サブコマンド　：　SAMESize△<パラメタ>

パラメタ　：　<数値>

■ダイアログメニュー

Optimize[Eliminated Size]

■説明

optimize=same_code で、最適化の対象となる共通コードのサイズを指定します。このオプションで指定するサイズは、オブジェクトプログラムの実際のバイト数を指します。optimize=same_code オプションが有効でない場合には、本オプション／サブコマンドは無視されます。

<数値>：

16進数で指定します。指定したサイズ以上の命令列について、共通コードのサブルーチン化を実施します。本オプション／サブコマンド省略時は、samesize=1Eを仮定します。指定できる範囲は8≦ 数値 ≦ 7FFF です。

(3) 未参照シンボル削除を抑止するシンボルの指定

■オプション／サブコマンド形式

オプション　：　SYmbol_forbid=<パラメタ>[,<パラメタ>...]

サブコマンド　：　SYmbol_forbid△<パラメタ>[,<パラメタ>...]

パラメタ　：　<シンボル名>

■ダイアログメニュー

Optimize[Forbid Item->Forbid Elimination of Dead Code to]

■説明

未参照シンボル削除（optimize=symbol_delete）の最適化を抑止する変数名／関数名を指定します。optimize=symbol_delete オプションが有効でない場合には、本オプション／サブコマンドは無視されます。

<シンボル名>：

変数名、C関数名はC/C++プログラム中での定義名の先頭に_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と同じ名前をダブルクォーテーシ

ョンで囲んで指定します。（例：“ f(int) ”）
最適化によって削除してはならない変数名、関数名を指定してください。

(4) 共通コード統合を抑止する関数の指定

■オプション／サブコマンド形式

オプション　：　SAMECode_forbid=<パラメタ>[,<パラメタ>...]

サブコマンド　：　SAMECode_forbid△<パラメタ>[,<パラメタ>...]

パラメタ　：　<関数名>

■ダイアログメニュー

Optimize[Forbid Item->Forbid Elimination of Same Code to]

■説明

共通コード統合(optimize=same_code)の最適化を抑止する関数名を指定します。optimize=same_code オプションが有効でない場合は、本オプション／サブコマンドは無視されます。

<関数名>：

C関数の場合は、プログラム中での定義名の先頭に_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と同じ名前をダブルクォーテーションで囲んで指定します。（例：“ f(int) ”）
共通コード統合の最適化は、コードサイズ削減には効果がありますが、実行速度が低下する可能性があります。スピードを重視する関数等、共通コード統合化を抑止したい関数名を指定してください。

(5) 短絶対アドレス領域割り付けを抑止する変数の指定

■オプション／サブコマンド形式

オプション　：　Variable_forbid=<パラメタ>[,<パラメタ>...]

サブコマンド　：　Variable_forbid△<パラメタ>[,<パラメタ>...]

パラメタ　：　<変数名>

■ダイアログメニュー

Optimize[Forbid Item->Forbid Use of Short Addressing to]

■説明

短絶対アドレッシングモード活用（optimize=variable_access）の最適化を抑止する変数名を指定します。 optimize=variable_access オプションが有効でない場合は、本オプション／サブコマンドは無視されます。

<シンボル名>：

変数名はC/C++プログラム中での定義名の先頭に_を付加します。
C/C++プログラム内で定義し、アセンブリプログラム内で間接参照している等、アドレスを変更してはならない変数名を指定してください。

(6) 間接アドレス呼び出しを抑止する関数の指定

■オプション／サブコマンド形式

オプション　：　Function_forbid=<パラメタ>[,<パラメタ>...]

サブコマンド　：　Function_forbid△<パラメタ>[,<パラメタ>...]

パラメタ　：　<関数名>

■ダイアログメニュー

Optimize[Forbid Item->Forbid Use of Indirect Call/Jump to]

■説明

間接アドレッシングモード活用（optimize=function_call）の最適化を抑止する関数名を指定します。optimize=function_call オプションが有効でない場合には、本オプション／サブコマンドは無視されます。

<関数名>：

C関数の場合は、プログラム中での定義名の先頭に_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と同じ名前をダブルクォーテーションで囲んで指定します。（例：“ f(int) ”）
間接アドレッシングモードの活用は、コードサイズ削減には効果がありますが、実行速度が低下する可能性があります。スピードを重視する関数等、間接アドレッシングモードの使用を抑止したい関数名を指定してください。

(7) 最適化による再割り付け使用を禁止する領域の指定

■オプション／サブコマンド形式

オプション　：　ABSolute_forbid=<パラメタ>[,<パラメタ>...]

サブコマンド　：　ABSolute_forbid△<パラメタ>[,<パラメタ>...]

パラメタ　：　<アドレス>[+<size>]

■ダイアログメニュー

Optimize[Forbid Item->Forbid Memory Allocation in]

■説明

最適化時に使用できないアドレス領域を指定します。

<アドレス>[+<サイズ>]：

アドレス、サイズとも16進数で指定します。また、アドレス値がA～Fの値
で始まる場合は、先頭に0を付加してください。optimize=function_callオプションやoptimize=variable_accessオプションでは、オブジェクト性能を向上するために、積極的にメモリ割り付けの変更を行います。I/O領域やエミュレータ予約領域など、最適化で使用できないアドレス領域を指定してください。

(8) 最適化情報リストの出力指定

■オプション／サブコマンド形式

オプション　：　Mlist[=<ファイル名>]

サブコマンド　：　Mlist△<ファイル名>

■ダイアログメニュー

List[Optimization File]

■説明

シンボルの参照回数および最適化情報をファイルに出力します。
ファイル名が省略された場合は、出力ファイル.lop になります。ファイル名の拡張子が省略された場合は、lop になります。
nooptimize オプション／サブコマンド指定時は、シンボルの参照回数のみ出力します。

(9) 最適化情報リスト内容の指定

■オプション／サブコマンド形式

オプション　　：SHow =<パラメタ>[,<パラメタ>]
サブコマンド　：SHow△<パラメタ>[,<パラメタ>]
パラメタ　　　：Symbol | Reference

■ダイアログメニュー

List[Optimization File]

■説明

最適化情報リストの出力内容を指定します。
symbol パラメタ指定時には、シンボルの最適化情報を出力します。
reference パラメタ指定時には、シンボルの参照回数情報を出力します。
mlist オプション／サブコマンドが指定されていない場合は、本オプション／サブコマンドは無効です。また、nooptimize オプション／サブコマンド指定時は、symbol パラメタは無効です。

(10) 最適化対象関数名の表示

■オプション／サブコマンド形式

オプション　　：INFormation
サブコマンド　：INFormation
パラメタ　　　：なし

■ダイアログメニュー

Optimize[Output Information]

■説明

最適化された関数名の表示を指定します。

(11) サブコマンドファイルの指定

■オプション／サブコマンド形式

オプション　　：SUbcommand=<パラメタ>
パラメタ　　　：<ファイル名>

■ダイアログメニュー

― [Output Information]

■説明

サブコマンドファイルを指定します。リンケージ機能は、サブコマンドのみサポートしているため、本オプションを省略することはできません。

<ファイル名>：

リンケージエディタ用サブコマンドファイル名称を指定します。
サブコマンドファイル名には、-を含めることはできません。

注意

1. **リンケージエディタ用サブコマンドファイルは、最適化ツール用サブコマンドファイルとして指定することができます。但し、最適化ツール用サブコマンドを指定したサブコマンドファイルを、H シリーズリンケージエディタのサブコマンドファイルとして指定した場合、エラーになります。**
2. print **サブコマンドを指定した場合、マップリストには最適化ツール出力のテンポラリサブコマンドファイル名が出力されます。このテンポラリサブコマンドファイルは、最適化ツール終了後削除されます。**

9.4 リンケージ機能サブコマンド

Hシリーズリンケージエディタでサポートしている表9.3のサブコマンドを指定できます。サブコマンドの大文字は短縮形指定時の文字を示します。また、下線は省略時解釈を示します。

また、日立統合化開発環境の対応するダイアログメニューを、タブ名[項目]で示します。ダイアログメニューの詳細は、日立統合化開発環境のユーザーズマニュアルを参照してください。

表9.3 リンケージ機能サブコマンド一覧

項番	分類	サブコマンド	ダイアログメニュー	機能
1	ファイル制御	Input	Input [Input Files]	入力ファイルの指定
2		LIBrary NOLIBrary	Input [Input Files]	ライブラリファイルの指定
3		ENTry	Input [Use Entry Point]	実行開始アドレスの指定
4		DEFine	Input [Defines]	外部参照シンボルの強制定義
5		Form = A | R	Output [Load Module Type]	アブソリュート形式で出力 リロケータブル形式で出力
6		DEBug NODEBug	Output [Debug Information]	デバッグ情報の出力
		SDebug		デバッグ情報をファイル出力
7		ELf	Output [Object Format]	ELF フォーマット出力
		SYSROF		sysrof フォーマット出力
		SYSROFPlus		sysrof フォーマット出力で、 dwarf デバッグ情報出力可能
8		Output NOOutput	Output [Load Module Path]	出力ファイルの指定
9		ROm	Output [ROM to RAM Mapped Sections]	ROM 化支援機能の指定
10		EXCLude NOEXCLude	Output [Exclude Unreferenced External Symbol]	未参照ライブラリの結合抑止
11		FSymbol	Output [Symbol Address File]	シンボルアドレスのファイル出力指定
12		Print NOPrint	List [Generate Map File]	リンケージリストファイルの出力指定
13		DIrectory	-	ディレクトリ名置き換え指定
14		STart	Section	セクションのアドレス指定
15		ALign_section	Section [Align section]	境界調整が異なるセクションの結合指定
16		CHeck_section	Verify [Check for Unlinked Section]	アドレス未指定セクションのチェック
17		Cpu	Verify [Use CPU Information File]	CPU 情報ファイルによるメモリ割り付けチェック
18		CPUCheck	Verify [Stop Linkage on CPU Information warning]	CPU 情報ファイルによるメモリ割り付けチェック時のエラー出力
19	エラーチェック	Udf NOUdf	-	未定義シンボルの表示指定

項番	分類	サブコマンド	ダイアログメニュー	機能
20	ｴﾗｰﾁｪｯｸ	UDFCheck	Verify->Check for Undefined Symbol	**未定義シンボル存在時のエラー出力**
21	その他	EXCHange	-	**ユニットの強制置換**
22		Rename	-	**シンボル名の変更**
23		DELete	-	**シンボルの削除**
24		EXIt	-	**リンケージ処理終了指定**
25		ECho NOECho	-	**サブコマンドのエコーバック**

(1) 入力ファイルの指定

■サブコマンド形式

```
Input△<ファイル名>[,<ファイル名>...]
```

■ダイアログメニュー

```
Input[Input Files]
```

■説明

入力ファイル名を指定します。複数ファイル名を指定する場合は、空白で区切ることもできます。
入力ファイル名に拡張子の指定がない場合は、.obj を仮定します。
入力ファイルとして、ライブラリファイル内のモジュールを"<ライブラリ名>(<モジュール名>)"の形式で指定することもできます。ライブラリ名に拡張子の指定がない場合は、.lib を仮定します。

■使用例

(a) オブジェクトモジュールファイルを入力します。

```
INPUT△main                    ; main.objを入力
```

(b) 複数のファイルを入力する場合はカンマまたは空白で区切ります。

```
INPUT△main△tan.obj           ; main.obj、tan.objを入力
```

(c) ライブラリファイル内のモジュールを入力します。

```
INPUT△funclib(sin,cos),tan.o  ; funclib.lib内のモジュールsin、cosを入力
```

(2) ライブラリファイルの指定

■サブコマンド形式

```
LIBrary△<ファイル名>[,<ファイル名>...]
NOLIBrary
```

■ダイアログメニュー

```
Input[Input Files]
```

■説明

ライブラリファイルの入力有無を指定します。

library サブコマンドはライブラリの入力およびライブラリ名を指定します。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、未解決の外部参照シンボルをライブラリからサーチします。

複数ファイル名を指定する場合は、空白で区切ることもできます。

ライブラリ名に拡張子の指定がない場合は、.lib を仮定します。

nolibrary サブコマンド指定時は、ライブラリを入力しません。

■使用例

(a) ライブラリファイルを指定します。

```
LIBRARY△syslib             ; syslib.lib を入力
```

(b) 複数のファイルを入力する場合はカンマまたは空白で区切ります。

```
LIBRARY△system△debug      ; system.lib と debug.lib を入力
```

(3) 実行開始アドレスの指定

■サブコマンド形式

```
ENTry△<外部定義シンボル名>
```

■ダイアログメニュー

```
Input[Use Entry Point]
```

■説明

実行開始アドレスを指定します。指定できるシンボル名は、外部定義シンボルだけです。

#pragma entry 指定と ENTRY を同時に指定した場合、ENTRY 指定を有効とします。

entry サブコマンドが指定されていない場合、optimize=symbol_delete, register が無効になります。

■使用例

```
ENTRY△_INIT
```

INIT 関数をエントリとして指定シンボル名はアセンブリプログラムで参照するシンボル名を指定します。 C プログラムの場合は先頭に下線(_)を付加します。 C++プログラムの場合は引数列を含めたプログラム中の定義名と同じ名前をダブルクォーテーションで囲んで指定します。（例：“ init(void) ”）

(4) 外部参照シンボルの強制定義

■サブコマンド形式

```
DEFine△<外部定義シンボル名>(<パラメタ>)
<パラメタ>:  <数値> | <アドレス> | <外部定義シンボル名>
```

■ダイアログメニュー

```
Input[Defines]
```

■説明

外部参照シンボルを、指定された値、アドレスまたは外部定義シンボルの値で強制的に定義します。
数値、アドレスは先頭が数字の 16 進数で指定します。先頭が A～F の場合は 0 を付加してください。また、外部定義シンボルは、既に定義済みのシンボルを指定してください。
リロケータブルロードモジュール（form△r）指定時には、本サブコマンドは無効になります。

■使用例

(a) 未定義シンボルを強制定義します。

```
DEFINE△PORT10(0FFE8)      ; 未定義シンボル"PORT10"を"FFE8"で定義
```

(b) 未定義シンボルを外部定義シンボルと同値として定義します。

```
DEFINE△MAIN_RTN(PRG_EXIT) ; MAIN_RTN を PRG_EXIT と同じ値を持つシンボルとして定義
```

(5) 出力ファイルの指定

■サブコマンド形式

```
Output△[<ファイル名>]
NOOutput
```

■ダイアログメニュー

```
Output[Load Module Path]
```

■説明

ロードモジュール出力有無を指定します。
output サブコマンドはロードモジュールの出力および出力ファイル名を指定します。ファイル名の拡張子が省略された場合は、form=a のときには.abs、form=r のときには.rel になります。
またファイル名が省略された場合は、<先頭に入力されたファイル名>に.abs または.rel の拡張子を付加したファイル名になります。
nooutput サブコマンド指定時はロードモジュールを出力しません。

■使用例

```
output△sample.abs
```

出力ファイル名を sample.abs に指定します。

(6)　ROM 化支援機能の指定

■サブコマンド形式

```
ROm△(<セクション名 1>,<セクション名 2>)[,(<セクション名 1>,<セクション名 2>)…]
<セクション名 1>：　ROM 割り付け用セクション名（コンパイラ出力セクション名）
<セクション名 2>：　RAM 割り付け用セクション名
```

■ダイアログメニュー

```
Output[Rom to Ram Mapped Sections]
```

■説明

初期化データ領域を ROM、RAM 上に二重に確保します。

コンパイラ出力の初期化データ領域セクションを ROM 上に割り付けるセクション（セクション名１）として指定します。セクション名２にサイズ０以外の存在するセクションを指定した場合はエラーになります。

■使用例

```
ROM△(D,R)
START△P,C,D(200),R,B(8000)
```

D セクションと同サイズの R セクションを確保し、D セクション内シンボル参照個所の R セクション上のアドレスでリロケーションします。

start サブコマンドを用いて、D セクションを ROM 上、R セクションを RAM 上アドレスに割り付けてください。

(7)　オブジェクトフォーマットの指定

■サブコマンド形式

```
ELf
SYSROF
SYSROFPlus
```

■ダイアログメニュー

```
Output[Object Format]
```

■説明

オブジェクトフォーマットを指定します。本サブコマンドの指定を省略した場合、SYSROF（従来フォーマット）のオブジェクトファイルを出力します。

デバッグ情報出力サブコマンド(DEBUG/SDEBUG)との組み合せで、5 種類のオブジェクトフォーマットを選択できます。使用するデバッガに合わせて選択してください。

表 9-4　使用可能なデバッガとオプション／サブコマンドの関係

使用可能なデバッガ	オプション／サブコマンド	
	ｵﾌﾞｼﾞｪｸﾄﾌｫｰﾏｯﾄ	ﾃﾞﾊﾞｯｸﾞ情報出力
3rd party 製 ELF/DWARF ｻﾎﾟｰﾄのﾃﾞﾊﾞｯｶﾞ	ELF	DEBUG
日立統合化ﾏﾈｰｼﾞｬ(Ver.4) + E7000	SYSROFPLUS	SDEBUG
日立統合化ﾏﾈｰｼﾞｬ(Ver.3) + E7000	SYSROF	SDEBUG
日立ﾃﾞﾊﾞｯｷﾝｸﾞｲﾝﾀﾌｪｰｽ(Ver.2) + E6000	SYSROF	DEBUG
日立ﾃﾞﾊﾞｯｷﾝｸﾞｲﾝﾀﾌｪｰｽ(Ver.3) + E6000	ELF	SDEBUG

注意　リンク時に ELF または SYSROFPLUS オプションを指定した場合、旧バージョンのコンパイラ、リンケージエディタ、ライブラリアンで生成したオブジェクトプログラム、ライブラリのデバッグ情報は削除されます。

■使用例

(a) ELFフォーマットのロードモジュールファイルsample.absとデバッグ情報（DWARFフォーマット）ファイルsample.dwfを出力します。

```
ELF
SDEBUG
output sample.abs
   :
```

(b) ELFフォーマットのロードモジュールファイルsample.absを出力します。デバッグ情報（DWARFフォーマット）はロードモジュール内に出力します。

```
ELF
DEBUG
output sample.abs
   :
```

(c) SYSROFフォーマットのロードモジュールファイルsample.absとデバッグ情報（SYSROFフォーマット）ファイルsample.dbgを出力します。

```
SYSROF
SDEBUG
output sample.abs
   :
```

(d) SYSROFフォーマットのロードモジュールファイルsample.absを出力します。デバッグ情報（SYSROFフォーマット）はロードモジュール内に出力します。

```
SYSROF
DEBUG
output sample.abs
   :
```

(e) SYSROFPLUSサブコマンドとDEBUGサブコマンドは同時に指定できません。指定した場合はエラーになります。

```
SYSROFPLUS
DEBUG                            ; エラー 3010 を出力
   :
```

(8) ロードモジュール形式の指定

■サブコマンド形式

Form△A｜ R

■ダイアログメニュー

Output[Load Module Type]

■説明

出力ロードモジュール形式を指定します。
form△a 指定時は、アブソリュート形式ロードモジュールを出力します。
form△r 指定時は、リロケータブル形式ロードモジュールを出力します。
rom または start サブコマンド指定時は、form△r は無効です。
本サブコマンドの省略時解釈は、form△a です。

(9) デバッグ情報出力の指定

■サブコマンド形式

DEBug

SDebug

NODEBug

■ダイアログメニュー

Output[Debug Information]

■説明

デバッグ情報の出力有無を指定します。
debug 指定時は、ロードモジュール中にデバッグ情報を出力します。
sdebug 指定時は、ロードモジュールと別ファイルとしてデバッグ情報ファイルを出力します。
nodebug 指定時は、デバッグ情報を出力しません。
debug または sdebug 指定時に、全ての入力ファイルにデバッグ情報が含まれていない場合はウォーニングメッセージを表示します。
debug、sdebug 指定は、オブジェクトフォーマット指定との組み合わせでデバッグ情報フォーマットが以下のように規定されます。

オブジェクトフォーマット指定	デバッグ情報指定	デバッグ情報フォーマット
elf	debug	dwarf
	sdebug	dwarf
sysrof	debug	sysrof（従来フォーマット）
	sdebug	sysrof（従来フォーマット）
sysrofplus	debug	指定不可
	sdebug	dwarf

注意 1. C++プログラムのソースレベルデバッグを行う場合、デバッグ情報フォーマットが dwarf になるように、オブジェクトフォーマット、デバッグ情報指定をしてください。
2. 使用例は「(7)オブジェクトフォーマットの指定」を参照してください。

(10) 未参照ライブラリの結合抑止

■サブコマンド形式

```
EXCLude
NOEXCLude
```

■ダイアログメニュー

```
Output[Exclude Unreferenced External Symbol]
```

■説明

アセンブリプログラム内で外部参照（.import）宣言した未参照シンボルの結合有無を指定します。
exclude サブコマンド指定時は、未参照シンボルの結合を抑止します。
noexclude サブコマンド指定時は、未参照シンボルを結合します。
form△r 指定時、noexclude 指定は無効になります。
本サブコマンドの省略時解釈は、noexclude です。

■使用例

```
    exclude
      :
```

アセンブリプログラム内で.import 宣言した未参照シンボルの結合抑止を指定します。

注意　exclude サブコマンドは input サブコマンドより前に指定してください。

(11) シンボルアドレスのファイル出力指定

■サブコマンド形式

```
FSymbol  <セクション名>[,<セクション名>...]
```

■ダイアログメニュー

```
Output[Exclude Unreferenced External Symbol]
```

■説明

リンケージ機能で解決した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。
出力するファイル名は、ロードモジュール名に拡張子".fsy"を付加したファイルとなります。
指定したセクション名が存在しない場合、ウォーニングメッセージを表示し、処理を継続します。
指定したセクションが全て存在しない場合、ファイルは出力されません。
form△r 指定時、本サブコマンドは無効です。

■使用例

```
    fsymbol△sct2,sct3
    output△sample.abs
```

セクション sct2 と sct3 の外部定義シンボルを sample.fsy ファイルに出力します。
【ファイルsample.fsyの出力例】

```
;H SERIES LINKAGE EDITOR GENERATED FILE   1998.10.10
;fsymbol = sct2, sct3
;SECTION NAME = sct2
        .export sym1
sym1:   .equ     h'00FF0080     ;sct2 内外部定義シンボル、アドレスを.equ 形式で出力
             :
```

sample.fsy をアセンブルし、リンク時に入力ファイルとして指定すると、sct2、sct3 セクションのオブジェクトファイルをリンクすることなく、外部参照シンボルのアドレスを解決できます。
本機能は、複数のロードモジュールで共通 ROM を参照する場合にご使用ください。

(12)　リンケージリストファイルの出力指定

■サブコマンド形式

```
Print  <ファイル名>
NOPrint
```

■ダイアログメニュー

```
Output[Generate Map File]
```

■説明

リンケージリストファイル出力の有無を指定します。
print サブコマンドはリンケージ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。
ファイル名の拡張子が省略された場合は、.map を付加したファイル名になります。ファイル名は省略できません。
noprint サブコマンド指定時は、リンケージリストファイルを出力しません。

(13)　ディレクトリ名置き換え指定

■サブコマンド形式

```
DIrectory  <シンボル名>（<ディレクトリ名>）
```

■ダイアログメニュー

なし

■説明

ディレクトリ名をシンボル名に置き換えて指定することができます。
登録したディレクトリ名の参照は、シンボル名を$と/(ＰＣ版では$と¥)で囲んで指定します。
ディレクトリ名は 16 個まで指定できます。

■使用例

```
DIRECTORY △ symbol(dir1/dir2)
INPUT △ $symbol/file1.obj
```

ディレクトリ名 dir1/dir2 をシンボル名 symbol として登録します。
$symbol/を dir1/dir2/に置き換え、ファイル名を dir1/dir2/file1.obj とします。

(14)　セクションのアドレス指定

■サブコマンド形式

```
STart  <セクション名>[[[,<セクション名>...]
             [:<セクション名>[,<セクション名>...]...](<先頭アドレス>)]...]
```

■ダイアログメニュー

```
Section[Section Start Address]
```

■説明

セクションの開始アドレスを指定します。開始アドレスの指定がないセクションは、最終割り付けアドレスに続いて割り付けます。

またセクション群をコロン（：）で区切って、同一アドレスへの複数セクションの割り付けを指定できます。

アドレスは、先頭が数字で始まる 16 進数で指定します。

■使用例

```
            :
    ROM    (D1,R1),(D2,R2)
    START  P,C,D1,D2(8000)
    START  R1,B:R2(0D0000)
```

P、C、D1、D2 セクションを指定した順に結合し、8000 番地から割り付けます。

R1、B セクションを結合し、D0000 から割り付けます。また、R2 セクションも D0000 から割り付けます。

(15) 境界調整が異なるセクションの結合指定

■サブコマンド形式

```
ALign_section
```

■ダイアログメニュー

```
Section[Align Section]
```

■説明

同一名で境界調整数が異なるセクションでも、同一のアドレスに割り付けます。本サブコマンド省略時には、同一名で境界調整数が異なるセクションは別セクションとして扱い、後から入力されたセクションを、開始アドレス未指定セクションとして扱います。

(16) アドレス未指定セクションのチェック

■サブコマンド形式

```
CHeck_section
```

■ダイアログメニュー

```
Verify[Check for Unlinked Section]
```

■説明

start オプションで指定しなかったセクションが存在するとき、ウォーニングメッセージを出力します。

(17) CPU 情報ファイルによるメモリ割り付けチェック

■サブコマンド形式

```
Cpu  <ファイル名>
```

■ダイアログメニュー

```
Verify[Use CPU Information File]
```

■説明

CPU 情報ファイルに基づき、割り付けメモリ範囲が使用可能であるかどうかをチェックします。
本サブコマンドが指定されない場合、optimize=variable_access, function_call が無効になります。
CPU 情報ファイルは、CPU 情報作成ツール(cia38)を用いて作成します。
「**付録 J CPU 情報作成ツールの使用方法**」を参照してください。

(18) CPU 情報ファイルによるメモリ割り付けチェック時のエラー出力

■サブコマンド形式

```
CPUCheck
```

■ダイアログメニュー

```
Verify[Stop Linkage on CPU Information Warning]
```

■説明

cpu サブコマンド指定時、CPU 情報ファイルで指定したメモリレイアウトと矛盾したメモリ割り付けを行った場合、エラーを出力してリンケージ処理を終了します。
cpu サブコマンドを指定していない場合は、本サブコマンドは無効です。

(19) 未定義シンボルの表示指定

■サブコマンド形式

<u>Udf</u>

NOUdf

■ダイアログメニュー

なし（常に udf が有効になります）

■説明

未定義シンボルが存在する場合、メッセージを出力します。リロケータブル形式(form=r)指定時、本サブコマンドは無効です。

(20) 未定義シンボル存在時のエラー出力

■サブコマンド形式

```
UDFCheck
```

■ダイアログメニュー

```
Verify[Check for Undefined Symbol]
```

■説明

udf サブコマンド指定時、未定義シンボルが存在する場合、エラーを出力してリンケージ処理を終了します。udf サブコマンドが指定されていない場合、本サブコマンドは無効です。

(21) ユニットの強制置換

■サブコマンド形式

```
EXCHange  <ファイル名>[(<ユニット名>[,<ユニット名>…])]
```

■ダイアログメニュー

なし

■説明

指定ファイル中の指定ユニットを、リンケージ処理中のロードモジュール内同名ユニットと入れ替えます。ファイル名として指定できるのは、コンパイラ、アセンブラ出力のオブジェクトファイルかリロケータブルロードモジュールです。

ファイル名に拡張子の指定がない場合は、.obj を仮定します。

(22)　シンボル名の変更

■サブコマンド形式

```
REname        <パラメタ>[,<パラメタ>…]
<パラメタ>:   un=<ユニット名>(<ユニット名>) |
              er=<ユニット名>.<外部参照シンボル名>(<外部参照シンボル名>) |
              ed=<ユニット名>.<外部定義シンボル名>(<外部定義シンボル名>)
```

■ダイアログメニュー

なし

■説明

ユニット名、外部定義シンボル名、外部参照シンボル名を括弧内で指定された名前に変更します。
本サブコマンド以降に指定した最初の input サブコマンドの入力ファイルのみが対象になります。
また rename サブコマンドの直後に指定できるのは、以下の５つのサブコマンドのみです。

- input
- exchange
- rename
- delete
- abort

(23)　シンボルの削除

■サブコマンド形式

```
DELete        <パラメタ>[,<パラメタ>…]
<パラメタ>:   un=<ユニット名> | ed=<ユニット名>.<外部定義シンボル名>
```

■ダイアログメニュー

なし

■説明

ユニット名、外部定義シンボル名、外部参照シンボル名を削除します。
本サブコマンド以降に指定した最初の input サブコマンドの入力ファイルのみが対象になります。
また delete サブコマンドの直後に指定できるのは、以下の５つのサブコマンドのみです。

- input
- exchange
- rename

- delete
- abort

(24) リンケージ処理終了指定

■サブコマンド形式

`EXIt`

■ダイアログメニュー

なし

■説明

リンケージ処理を終了します。サブコマンドファイルの最後に必ず指定してください。

(25) サブコマンドのエコーバック

■サブコマンド形式

`ECho`

`NOECho`

■ダイアログメニュー

なし

■説明

echo 指定時、サブコマンドをコンソール表示します。
noecho 指定時、サブコマンドをコンソール表示しません。
本サブコマンドの省略時解釈は、noecho です。

9.5 モジュール間最適化の環境変数

モジュール間最適化ツールで使用する環境変数の使用方法を**表** 9.5 に示します。

表 9.5 **モジュール間最適化の環境変数**

No.	**環境変数**	**説明**
1	path	**モジュール間最適化ツール本体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= <**モジュール間最適化ツール本体パス名**>;%path% unix **版** C **シェル** %set path =(<**モジュール間最適化ツール本体パス名**> $path) **ボーン** %PATH=:<**モジュール間最適化ツール本体パス名**>[:<**既存パス名**>...] **シェル** %export PATH
2	HLNK_LIBRARY1 HLNK_LIBRARY2 HLNK_LIBRARY3	**デフォルトライブラリ名を指定します。** **ライブラリサブコマンド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、デフォルトライブラリを** 1,2,3 **の順に入力します。** **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** %HLNK_LIBRARY1=<**ライブラリ名** 1> **シェル** %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
3	HLNK_TMP	**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

(f) sbrk ルーチン				
機能	メモリ領域を割り付けます。			
インタフェース	`char *sbrk (int size);`			
引数	No.	名前	型	意味
	1	size	int	割り付けるデータのサイズ
リターン値	型	char 型を指すポインタ		
	正常	割り付けた領域の先頭アドレス		
	異常	（char *）−1		

■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アドレスを、失敗した場合は「（char *）−1」を返してください。

10. コンパイラのエラーメッセージ

10.1 コンパイラのエラーメッセージ

本章では、以下の形式でコンパイラの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種に分類されます。

表 10.1 コンパイラのエラーレベル

(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	処理を中断します。
(-)	インターナル	処理を中断します。

表 10.2 コンパイラのエラーメッセージ

0001	(I) Character combination "/*" in comment
	注釈の中に、文字列"/*"があります。
0002	(I) No declarator
	宣言子のない宣言があります。
0003	(I) Unreachable statement
	実行されることのない文があります。
0004	(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。
0005	(I) Precision lost
	代入において、右辺の式の値を左辺の型に変換するときに、精度が失われる可能性があります。
0006	(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。
0008	(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。
0010	(I) Elimination of needless expression
	不要な式があります。
0011	(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。
0012	(I) Symbol name too long
	シンボル名長が 32 文字を超えています。シミュレータ・デバッガ(sd38)を単独で起動したとき、正常にロードできない場合があります。

番号	メッセージ
0015	`(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。
0100	`(I) Function not optimized : "関数名"`
	“関数名”のサイズが大きすぎるため、最適化できません。
0200	`(I) No prototype function`
	関数の原型宣言がありません。
0300	`(I) #pragma interrupt has no effect`
	#pragma interrupt で指定された関数が存在しません。
0301	`(I) #pragma abs8 has no effect`
	#pragma abs8 で指定された変数が存在しません。
0302	`(I) #pragma abs16 has no effect`
	#pragma abs16 で指定された変数が存在しません。
0303	`(I) #pragma indirect has no effect`
	#pragma indirect で指定された変数が存在しません。
0304	`(I) #pragma regsave/noregsave has no effect`
	#pragma regsave/noregsave で指定された変数が存在しません。
0305	`(I) #pragma inline has no effect`
	#pragma inline で指定された変数が存在しません。
0306	`(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。
0307	`(I) #pragma entry has no effect`
	#pragma entry で指定された宣言が存在しません。
1000	`(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。
1001	`(W) Illegal comparison`
	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。
1002	`(W) Illegal pointer required`
	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。
1005	`(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（逆スラッシュとそれに続く文字）を用いています。
1007	`(W) Long character constant`
	文字定数の長さが 2 文字以上になっています。
1008	`(W) Identifier too long`
	識別子の長さが 250 文字を超えています。
1010	`(W) Character constant too long`
	文字定数の長さが 2 文字を超えています。
1012	`(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。
1013	`(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。

1014	(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。
1015	(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。
1016	(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。関数呼び出しの引数のポインタの内部表現をそのまま設定します。
1017	(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。
1018	(W) Type mismatch
	extern 記憶クラスを持つ同じ名前の変数あるいは関数を、異なる有効範囲で宣言していますが、型が一致していません。
1019	(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。
1020	(W) Illegal constant expression
	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。
1021	(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。
1022	(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。
1023	(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。
1024	(W) First operand of "演算子" is not lvalue
	演算子の第一オペレランドに左辺値以外を指定しています。
1200	(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。
1201	(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。
1300	(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。
1302	(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。
1303	(W) Completed file name too long
	ルートディレクトリを起点としたパス名を含むファイル名の長さが 251 文字を超えています。コマンドライン上で指定したファイル名を debug 情報に出力します。
1305	(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。

1306	(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。
1307	(W) Section name too long
	セクション名称の長さが 32 文字を超えています。32 文字までを有効とし、それ以降の文字を無視します。
1308	(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。
1310	(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。
1311	(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。
1400	(W) #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。
1401	(W) #pragma abs16 ignored
	CPU/動作モードが 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。
1403	(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。
1404	(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。switch 文をテーブル方式に展開するためには、switch 文の case ラベル値を以下の範囲内にしてください。 （case ラベル値の範囲）＜（case ラベル数×3） case ラベル値の範囲：（case ラベル値の最大値 - case ラベル値の最小値＋1）
1405	(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。
1500	(W) EC++ does not support "クラス名"
	EC++ではサポートしていないクラスです。
2000	(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。
2001	(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。
2002	(E) "," Not found
	引数のある#define 文で引数の並びを区切るコンマ（,）が抜けています。
2003	(E) ")" Not found
	名前が#define 文で定義されているかどうかを判定する defined 式で名前の次の右括弧「)」が抜けています。
2004	(E) ">" Not found
	#include 文のファイル名の指定でファイル名の次の > がありません。
2005	(E) Cannot open include
	#include 文で指定したファイル名のファイルがオープンできません。
2006	(E) Multiple #define's
	#define 文で同じマクロ名を再定義しています。

2007	(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。
2008	(E) #elif mismatches
	#elif 文に対応する#if 文、#ifdef 文、#ifndef 文、#elif 文がありません。
2009	(E) #else mismatches
	#else 文に対応する#if 文、#ifdef 文、#ifndef 文がありません。
2010	(E) Macro parameters mismatch
	マクロ呼び出しの引数の数がマクロ定義の引数の数と異なっています。
2011	(E) Line too long
	マクロ展開後のソースプログラムの行が 8192 文字を超えています。
2012	(E) Keyword as a macro name
	プリプロセッサで規定しているキーワードを#define 文または、#undef 文のマクロ名として定義しています。
2013	(E) #endif mismatches
	#endif 文に対応する#if 文、#ifdef 文、#ifndef 文がありません。
2014	(E) #endif expected
	#if 文、#ifdef 文、#ifndef 文に対応する#endif 文がないままでファイルが終了しました。
2016	(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。
2017	(E) Missing "
	#include 文のファイル名の指定で、ファイル名の次に " がありません。
2018	(E) Illegal #line
	#line 文で指定した行数が 32767 行を超えています。
2019	(E) File name too long
	ファイル名の長さが 128 文字を超えています。
2020	(E) System identifier redefined : "名前"
	組み込み関数と同名のシンボルを定義しています。
2021	(E) System identifier mismatch : "名前"
	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。
2100	(E) Multiple strage 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。
2101	(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。
2102	(E) Illegal type combination
	型指定子の組み合わせが誤っています。
2103	(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。
2104	(E) Illegal bit filed 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。
2105	(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。
2106	(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。

2107	(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。
2108	(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。
2109	(E) Illegal function declaration
	複文内の関数型の変数の宣言において、extern 以外の記憶クラスを指定しています。
2110	(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。
2111	(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。
2112	(E) Illegal bit field type
	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 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。
2113	(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。
2114	(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。
2115	(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。
2117	(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。
2118	(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。
2119	(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。
2120	(E) Illegal parameter strage 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。
2121	(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。
2122	(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。
2123	(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。
2124	(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。
2125	(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。
2126	(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 2600n,2000n,300hn,300 のとき 65535、 2600a:20,2000a:20,300ha:20 のとき 1048575、 2600a:24,2000a:24,300ha:24 のとき 16777215、 2600a:28,2000a:28 のとき 268435455、 2600a:32,2000a:32 のとき 4294967295 です。

2127	(E) Missing array size
	配列の要素数の指定がありません。
2128	(E) Illegal pointer declaration
	ポインタ型の宣言を示す * の直後に const、volatile 以外の型指定子を指定しています。
2129	(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。
2130	(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。
2131	(E) No type nor strage 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。
2132	(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。
2133	(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。
2134	(E) Initializer for parameter
	引数の宣言において初期値を指定しています。
2135	(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。
2136	(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。
2137	(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。
2138	(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。
2139	(E) No parameter type
	関数宣言の引数宣言に型指定がありません。
2140	(E) Illegal bit field
	共用体にビットフィールドを指定しています。
2141	(E) Illegal bit field
	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。
2142	(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。
2143	(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。
2200	(E) Index not integer
	配列の添字の式が整数型ではありません。
2201	(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。

2202	(E) Number of parameter 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。
2203	(E) Illegal member reference
	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。
2204	(E) Illegal member reference
	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。
2205	(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。
2206	(E) Modifiable lvalue required
	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。
2207	(E) Scalar required
	単項演算子 ! をスカラ型でない式に使用しています。
2208	(E) Pointer required
	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。
2209	(E) Arithmetic type required
	単項演算子 + または - を算術型でない式に使用しています。
2210	(E) Integer required
	単項演算子 ~ を汎整数型でない式に使用しています。
2211	(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。
2212	(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。
2213	(E) Arithmetic type required
	二項演算子 *、/、*= または /= を算術型でない式に適用しています。
2214	(E) Integer required
	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。
2215	(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。
2216	(E) Illegal type required for parameter
	関数呼び出しの引数の型に void 型を指定しています。
2217	(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。
2218	(E) Scalar required
	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。

<table>
<tr><td rowspan="2">2219</td><td>(E) Type not compatible</td></tr>
<tr><td>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、もう一方が値 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、もう一方が 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。</td></tr>
<tr><td rowspan="2">2220</td><td>(E) Modifieble lvalue required</td></tr>
<tr><td>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。</td></tr>
<tr><td rowspan="2">2221</td><td>(E) Illegal type for postincremet or postdecrement</td></tr>
<tr><td>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。</td></tr>
<tr><td rowspan="2">2222</td><td>(E) Type not compatible</td></tr>
<tr><td>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、もう一方が 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。</td></tr>
<tr><td rowspan="2">2223</td><td>(E) Incomplete tag used in expression</td></tr>
<tr><td>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。</td></tr>
<tr><td rowspan="2">2224</td><td>(E) Illegal type for assign</td></tr>
<tr><td>代入演算子 += または -= の両辺の型が正しくありません。</td></tr>
<tr><td rowspan="2">2225</td><td>(E) Undeclared name : "名前"</td></tr>
<tr><td>宣言していない名前を式の中で用いています。</td></tr>
<tr><td rowspan="2">2226</td><td>(E) Scalar required</td></tr>
<tr><td>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。</td></tr>
<tr><td rowspan="2">2227</td><td>(E) Illegal type for equality</td></tr>
<tr><td>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もう一方が値 0 の整定数または void 型へのポインタ型である場合。</td></tr>
<tr><td rowspan="2">2228</td><td>(E) Illegal type for comparison</td></tr>
<tr><td>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。</td></tr>
<tr><td rowspan="2">2230</td><td>(E) Illegal function call</td></tr>
<tr><td>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。</td></tr>
</table>

2231	(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。
2232	(E) Illegal type for preincrement or predecrement 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。
2233	(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。
2234	(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。
2235	(E) Illegal cast ポインタを浮動小数点型にキャストしています。
2236	(E) Illegal cast in constant CPU/動作モードが 300,300hn,2600n,2000n の場合、定数式でポインタ型を char 型または long 型にキャストしています。 CPU/動作モードが 300ha,2600a,2000a の場合、定数式でポインタ型を char 型、short 型または int 型にキャストしています。
2237	(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。
2238	(E) Lvalue or function type required 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。
2239	(E) Illegal section name セクション名に使用できない文字があります。
2300	(E) Case not in switch case ラベルを switch 文以外に指定しています。
2301	(E) Default not in switch default ラベルを switch 文以外に指定しています。
2302	(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。
2303	(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。
2304	(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。
2305	(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。
2306	(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。
2307	(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。
2308	(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。
2309	(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。
2310	(E) Scalar required 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。
2311	(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。

2312	(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。
2313	(E) Missing ;
	do 文の最後のセミコロン(;)がありません。
2314	(E) Scalar required
	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。
2316	(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。
2320	(E) Illegal asm position
	#pragma asm の指定位置が適切ではありません。
2330	(E) Illegal interrupt function declaration
	割り込み関数宣言に誤りがあります。
2331	(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。
2332	(E) Interrupt function already declaration
	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。
2333	(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。
2334	(E) Illegal parameter in 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。
2335	(E) Missing parameter declaration in 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。
2336	(E) Parameter out of range in 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。
2337	(E) Illegal #pragma interrupt function type
	割り込み関数宣言に誤りがあります。
2340	(E) Illegal abs8 declaration
	短絶対アドレス変数宣言に誤りがあります。
2341	(E) 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。
2342	(E) Illegal abs8 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。
2345	(E) Illegal abs16 declaration
	短絶対アドレス変数宣言に誤りがあります。
2346	(E) 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。
2347	(E) Illegal abs16 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。
2350	(E) Illegal section name declaration
	#pragma section の指定に誤りがあります。
2352	(E) Section name table overflow
	通常のセクション、8 ビット／16 ビット絶対アドレスセクションおよび、メモリ間接呼び出しセクションの各セクション数が 64 個を超えたか、または全体で 256 個を超えました。

2360	(E) Illegal indirect function declaration メモリ間接関数宣言に誤りがあります。
2361	(E) Indirect function 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。
2362	(E) Illegal indirect 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。
2363	(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。
2370	(E) Illegal regsave/noregsaeve declaration #pragma regsave または#pragma noregsave 宣言に誤りがあります。
2371	(E) Regsave/noregsave function 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。
2372	(E) Illegal regsave/noregsave 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。
2380	(E) Illegal inline/inline_asm declaration #pragma inline または#pragma inline_asm 宣言に誤りがあります。
2381	(E) Inline/inline_asm function 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。
2382	(E) Illegal inlne/inline_asm 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。
2383	(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。
2390	(E) Illegal global_register declaration #pragma global_register 宣言に誤りがあります。
2391	(E) 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。
2392	(E) Illegal global_register 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。
2393	(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。
2400	(E) Illegal character : "文字" 不正な印字文字があります。
2401	(E) Incomplete character constant 文字定数の途中に改行があります。
2402	(E) Incomplete string 文字列の途中に改行があります。
2403	(E) EOF in comment コメントの途中でファイルが終了しました。
2404	(E) Illegal character code : "文字コード" 不正な文字コードがあります。
2405	(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。

2406	(E) 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。
2407	(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。
2408	(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。
2410	(E) Illegal #pragma entry declaration #pragma entry 宣言に構文エラーがあります。
2411	(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。
2412	(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。
2413	(E) Multiple #pragma entry declaration #pragma entry 宣言が複数存在しています。
2420	(E) Illegal #pragma pack/unpack declaration #pragma pack, #pragma unpack 宣言に構文エラーがあります。
2440	(E) #pragma stacksize declaration #pragma stacksize 宣言に構文エラーがあります。
2441	(E) Multiple #pragma stacksize declaration #pragma stacksize 宣言が複数存在しています
2500	(E) Illegal token : "語句" 語句の並びが文法に合っていません。
2501	(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。
2600	(E) #error diagnostic message : "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。
2801	(E) Illegal parameter type in inline function 組み込み関数で引数の型が一致しません。
2802	(E) Parameter out of range in inline 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。
3000	(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。
3001	(F) Block nest too deep 複文のネストが 256 レベルを超えています。
3002	(F) #if nest too deep 条件コンパイル(#if、#ifdef、#ifndef、#elif、#else)のネストが 32 レベルを超えています。
3006	(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。
3007	(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。
3008	(F) Line too long マクロ展開後の 1 行の長さが 8192 文字を超えています。

3009	(F) String literal too long
	文字列の長さが 512 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。
3010	(F) #include nest too deep
	#include 文によるファイルの取り込みのネストが 30 レベルを超えています。
3011	(F) Macro expansion nest too deep
	#define 文によるマクロ名の再置換が 32 個を超えています。
3013	(F) Too many switches
	switch 文の数が 256 個を超えています。
3014	(F) For nest too deep
	for 文のネストが 128 レベルを超えています。
3015	(F) Symbol table overflow
	コンパイラが生成するシンボルの数が 32767 を超えています。
3016	(F) Internal label overflow
	コンパイラが生成する内部ラベルの数が 49152 を超えています。
3017	(F) Too many case labels
	一つの switch 文の中の case ラベルの数が 511 個を超えています。
3018	(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。
3019	(F) Cannot open source file
	ソースファイルをオープンすることができません。
3020	(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。
3021	(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。
3022	(F) Switch nest too deep
	switch 文のネストが 128 レベルを超えています。
3023	(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。
3024	(F) Array dimension too deep
	配列の次元数が 6 次元を超えています。
3025	(F) Source file not find
	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。
3026	(F) Expression too complex
	式が複雑すぎます。
3027	(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。
3028	(F) Source line number overflow
	ソース行番号が 65535 行を超えています。

3031	(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 2600n,2000n,300hn,300 のとき 65535、 2600a:20,2000a:20,300ha:20 のとき 1048575、 2600a:24,2000a:24,300ha:24 のとき 16777215、 2600a:28,2000a:28 のとき 268435455、 2600a:32,2000a:32 のとき 4294967295 です。
3033	(F) Symbol table overflow
	デバッグ情報のためのシンボルの数が、65535 個を超えています。
3034	(F) Invalid file name "ファイル名"
	ファイル名の指定が不正です。
3200	(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 2600n,2000n,300hn,300 のとき 65535、 2600a:20,2000a:20,300ha:20 のとき 1048575、 2600a:24,2000a:24,300ha:24 のとき 16777215、 2600a:28,2000a:28 のとき 268435455、 2600a:32,2000a:32 のとき 4294967295 です。
3202	(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 2600n,2000n,300hn,300 のとき 32767、 2600a:20,2000a:20,300ha:20 のとき 524287、 2600a:24,2000a:24,300ha:24 のとき 8388607、 2600a:28,2000a:28 のとき 134217727、 2600a:32,2000a:32 のとき 2147483647 です。
3204	(F)Too many source lines for debug
	デバッグのための実行文の数が制限値を超えています。
3300	(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。
3301	(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。
3302	(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。
3303	(F) Cannot output interenal 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。
3304	(F) Cannot delete interenal 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。
3305	(F) Invalid command parameter
	コンパイラオプションの指定方法が誤っています。

3306	(F) Interrupt in compilation
	コンパイル処理中に標準入力端末から (CNTL) C コマンドによる割り込みを検出しました。
3307	(F) Compiler version mismatch "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。
3320	(F) Command parameter buffer overflow
	コマンドラインの指定が 65535 文字を超えています。
3321	(F) Illigal environment variable
	環境変数 CH38 の設定でファイル名の規約に反した指定をしているか、パス名の長さが 118 文字を超えています。
3322	(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。
3323	(F) Illegal environment specified
	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。
4000 〜 4999	(ー)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。
5001	(W) Linkage specification ignored for 'static' functions and objects
	リンケージ指定の { } の中で、static と宣言された関数やオブジェクトの extern リンケージ指令は無視されます。
6001	(E) Overloaded function "関数名" cannot specify #pragma function
	多重定義関数を#pragma 指定できません。
6002	(E) Preprocessing numerical token is not floating constant value or integer constant value
	前処理数字字句が浮動小数点定数、整数定数ではありません。
6101	(E) Illegal storage class 'auto'
	auto 記憶クラスは、ブロック内か、仮引数内のオブジェクト宣言に指定しなければなりません。
6102	(E) Illegal storage class 'register'
	register 記憶クラスは、ブロック内か、仮引数内のオブジェクト宣言に指定しなければなりません。
6103	(E) 'static' keyword must be applied to objects, functions and anonymous unions
	static 記憶クラスは、オブジェクト名、関数名または名前なし共用体以外に指定できません。
6104	(E) Function declaration "関数名" cannot have 'static' storage in a block
	ブロック内で static 関数を宣言することはできません。
6105	(E) Static linkage specifications for "名前" must agree
	名前に対する全てのリンケージ指定が一致していません。
6106	(E) Illegal storage class 'extern'
	extern 記憶クラスは、オブジェクト名、関数名または名前なし共用体以外に指定できません。
6108	(E) Inline member function "関数名" must be declared before it is called
	メンバ関数を呼んだ後でその関数を inline と宣言することはできません。
6109	(E) Illegal function specifier 'virtual'
	virtual 指定子は、クラス定義の非静的クラス・メンバ関数の宣言以外に指定できません。
6110	(E) Illegal 'typedef' declaration in function definition "名前"
	typedef 指定子は、関数定義に指定できません。
6111	(E) 'typedef' "名前" cannot re-define other types in the same scope
	同一スコープで宣言された型の名前を、別の型を参照するように再定義することはできません。

6112	(E) Cannot specify 'typedef' "名前" after 'enum'
	typedef で定義された enum 型名を enum 接頭辞のあとで使用することはできません。
6114	(E) Integral type data is not assigned to an enumeration
	整数型あるいは整数型への昇格の結果を列挙型オブジェクトに代入することはできません。
6116	(E) Undefined linkage-specification "文字列"
	リンケージ指定に未定義の文字列が指定されました。
6117	(E) Multiple linkage-specification "名前"
	関数やオブジェクトが複数のリンケージ指定を持つ時のリンケージ指定文字列が一致していません。
6118	(E) 'static' function "関数名" with linkage-specification
	リンケージ指定された関数に static 指定の関数宣言があります。
6119	(E) Multiple 'C' linkage overloaded function "関数名"
	多重定義関数の集合の中で C リンケージを持つものが複数あります。
6120	(E) Illegal 'void &' type
	void 型へのリファレンスは指定できません。
6123	(E) Cannot specify a reference to &, to bit-fields, to array or to pointer type
	リファレンス型、ビットフィールド、配列またはポインタ型に対してのリファレンス型は指定できません。
6124	(E) Initial value required for declaration & "名前"
	リファレンス型宣言に初期値の指定がありません。
6128	(E) New type defined in a return type of a function or in a parameter
	戻り値や仮引数型の中で、型を定義しています。
6129	(E) Non-static members or 'auto' variables cannot be used as default parameter
	デフォルト引数に非静的メンバが使われています。
6130	(E) Default parameter re-defined in function "名前"
	デフォルト引数は、後の宣言で再定義することができません。
6131	(E) Non-default parameters found following default parameters
	デフォルト引数の後にデフォルト引数以外の引数が存在します。
6133	(E) Overloaded operators cannot have default parameters
	多重定義演算子はデフォルト引数を持てません。
6139	(E) Overloaded functions "関数名" have the same type of parameters except & type
	関数の引数の型がリファレンス型の違いだけなので多重定義できません。
6140	(E) Functions "関数名" have the same type of parameters except const/volatile type
	関数の引数の型が const/volatile 型の違いだけなので多重定義できません。
6141	(E) Functions "関数名" have the same type of parameters except return type
	関数の戻り値が型の違いだけなので多重定義できません。
6142	(E) Cannot overload function "関数名"
	メンバ関数が static であるかどうかの違いだけなので多重定義できません。
6143	(E) Operator overloaded function "関数名" does not have correct parameters
	演算子関数は、以下のいずれかでなければなりません。 (1) メンバ関数である。 (2) クラスまたは列挙の引数を 1 つ以上持つ。 (3) クラスまたは列挙へのリファレンスの引数を 1 つ以上持つ。
6144	(E) Operator overloaded function "=" is not a nonstatic member function
	operator=() が非静的メンバ関数でありません。

6145	(E) Operator overloaded function "( )" is not a nonstatic member function
	operator()() が非静的メンバ関数でありません。
6146	(E) Operator overloaded function "[ ]" is not a nonstatic member function
	operator[]() が非静的メンバ関数でありません。
6147	(E) Operator overloaded function "->" returns illegal type value
	operator->()がクラスへのポインタ、または、operator->()を定義しているクラスのオブジェクト、あるいはリファレンスを返していません。
6148	(E) Operator overloaded function "->" is not a nonstatic member function
	operator->()が非静的メンバ関数でありません。
6149	(E) The second parameter of the postfix operator function should have type int
	後置の operator++()または operator--()の第 2 引数が int 型でありません。
6150	(E) 'const' object should have an initial value
	const 型のオブジェクトは外部リンケージを持たないので初期値が必要です。
6151	(E) 'friend' should be specified in a class declaration
	friend 指定子がクラス宣言外で使われています。
6152	(E) 'friend' keyword specified twice
	friend 指定子が 2 度指定されています。
6153	(E) 'inline' keyword specified twice
	inline 指定子が 2 度指定されています。
6154	(E) 'virtual' keyword specified twice
	virtual 指定子が 2 度指定されています。
6155	(E) 'inline' must be specified for functions
	inline 指定子が関数以外に指定されています。
6156	(E) Parameters cannot have 'inline' specifier
	引数に inline 指定子が指定されています。
6157	(E) Cannot specify 'inline' and 'extern' together
	inline と extern 指定子が同時に指定されています。
6158	(E) Object "名前" initialized with { } format
	{ } 形式で初期化できるオブジェクトは、次のいずれかです。 (1) 配列 (2) 非公開/限定公開のメンバ、基底クラス、コンストラクタ、仮想関数のいずれも持たないクラス
6159	(E) 'typedef' cannot specify 'friend'
	typedef と friend が合わせて指定されています。
6162	(E) Cannot declare 'typedef' name qualified by a class
	typedef 宣言の名前をクラスで限定することはできません。
6163	(E) Missing the number of operator function parameters
	多重定義演算子の引数の数が間違っています。
6165	(E) Operator member function cannot specify 'static'
	多重定義演算子は静的メンバ関数になれません。
6166	(E) Operator function or conversion function has function type
	多重定義演算子、変換関数が関数型ではありません。
6167	(E) Conversion function should be a nonstatic member function
	変換関数が非静的メンバ関数ではありません。
6169	(E) A destructor should be a function type
	デストラクタが関数型ではありません。

6170	(E) Non-member functions cannot specify 'const' or 'volatile' メンバ関数以外では、関数自身に const または volatile 型修飾できません。
6171	(E) Declarations qualified by a class cannot specify storage class specifier クラスで限定された宣言に記憶クラスは指定できません。
6172	(E) Cannot declare parameter declarations qualified by a class 引数宣言にクラスで限定された名前を指定しています。
6203	(E) Ambiguous name "名前" 関数名、オブジェクト名、または、型名が曖昧です。
6209	(E) Array "名前" does not have dimension size 配列が非静的メンバの型として使われていますが、全ての次元数が指定されていません。
6210	(E) Member declarator be ommited クラスのメンバ宣言子を省略することはできません。
6211	(E) Non-virtual function "関数名" cannot specify "=0" 純粋指定子 =0 指定できません。純粋指定子は仮想関数の宣言でのみ使用できます。
6212	(E) Member "名前" cannot have the same name of that class 静的データメンバ、列挙子、名前なしの共用体メンバあるいは入れ子型がクラスと同じ名前を持っています。
6213	(E) Static member function cannot access "名前" 静的メンバ関数で使用できるのは、静的メンバ、列挙子、入れ子型だけです。
6219	(E) Static data member "名前" cannot exist in a local class 局所クラスでは、静的データメンバを宣言できません。
6221	(E) Union "名前" cannot have virtual functions 共用体は、仮想関数を持つことができません。
6222	(E) Union "名前" cannot have base classes 共用体は、基底クラスを持つことができません。
6223	(E) Union "名前" cannot be a base class 共用体を基底クラスとして使用することはできません。
6224	(E) Union member cannot have a constructor, destructor or class with a user defined operator =( ) コンストラクタ、デストラクタまたはユーザ定義の代入演算子を持つクラスオブジェクトが共用体のメンバに存在します。
6225	(E) Union "名前" cannot have static data members static データメンバが共用体のメンバに存在します。
6227	(E) Anonymous union must specify 'static' static 指定のない大域的な名前無し共用体宣言が存在します。
6228	(E) Anonymous union cannot have private and protected members 名前無し共用体に private や protected メンバが存在します。
6229	(E) Anonymous union cannot have member functions 名前無し共用体に、関数メンバが存在します。
6233	(E) Nested class declaration cannot access "名前" 入れ子クラスの宣言では、それを囲むクラスからのオブジェクト、静的メンバ、列挙子以外を使用できません。
6234	(E) "名前" cannot be specified in a nested class declaration 局所クラス宣言内では、型名、静的変数、extern 変数、extern 関数、列挙子以外使用できません。
6236	(E) Member function "名前" should be defined in the local class declaration 局所クラス宣言内のメンバ関数をクラス内で定義していません。

6240	(E) Member "名前" defined more than once
	クラスメンバが 2 度宣言されています。
6241	(E) Undeclared member "名前"
	クラス宣言中に未定義のメンバ関数をクラス外でメンバ関数として定義しています。
6242	(E) "名前" declared multiple
	クラス再評価時に名前が確定しません。
6243	(E) Undeclared base class "名前"
	基底クラスに指定したクラス名が定義されていません。
6244	(E) Base class "名前" defined more than once in the derived class declaration
	派生クラスの直接基底クラスが 2 度以上指定されています。
6246	(E) Virtual base class "名前" casted to the derived class
	仮想基底クラスを派生クラスへキャストすることはできません。
6248	(E) Returning type mismatch in virtual function "関数名"
	基底クラスの仮想関数と派生クラスの仮想関数の戻り型が異なります。
6250	(E) Illegal 'static' virtual function "関数名"
	仮想関数を static 指定できません。
6251	(E) Cannot create abstract class object "名前"
	抽象クラスのオブジェクトを作成することはできません。
6254	(E) Implicit pure virtual function "関数名" call
	純粋仮想関数は明示的に限定して呼び出さなければなりません。
6256	(E) "名前" is not a member of the class
	指定されたクラスのメンバが定義されていません。
6263	(E) "名前" should be defined as a class or a struct
	class または、struct 指定子で宣言した名前が、異なる指定子で定義されています。
6267	(E) Illegal declaration "名前" in type-specifier
	クラス、列挙、typedef 名は、型指定並びの中で宣言できません。
6268	(E) "関数名" cannot have parameters and return value
	変換関数に引数型や戻り型を指定できません。
6269	(E) Ambiguous user defined conversion
	利用者定義変換が曖昧です。
6270	(E) Destructor "関数名" cannot have parameters and return value
	デストラクタに引数や戻り型を指定できません。
6271	(E) Constructor and destructor cannot have 'const', 'volatile' or 'static' keywords
	コンストラクタ、デストラクタに const、volatile、static を指定できません。
6272	(E) Constructor and destructor "関数名" do not have addresses
	コンストラクタ、デストラクタのアドレスを求めることはできません。
6274	(E) The first parameter type of operator new() should be 'size_t'
	クラスの operator new() 関数の第 1 引数は、整数型の size_t でなければなりません。
6275	(E) The return type of operator new() should be 'void *'
	クラスの operator new() 関数の戻り型は、void * 型でなければなりません。
6276	(E) The first parameter type of operator delete() should be 'void *'
	クラスまたは大域の operator delete() 関数の第 1 引数は、void *でなければなりません。
6277	(E) The return type of operator delete() should be 'void'
	クラスの ::operator delete() の戻り型は void 型でなければなりません。

6278	(E) Operator delete() function cannot be overloaded
	operator delete() は多重定義できません。
6280	(E) The second parameter type of operator delete() should be 'size_t'
	クラスの operator delete() 関数の第 2 引数は、整数型の size_t でなければなりません。
6281	(E) Illegal virtual operator new() or delete() function
	operator new()、operator delete() は、仮想関数指定できません。
6282	(E) Default constructor required
	初期設定子ならびに対応する初期値がそろっていないか、初期値がない場合のデフォルトコンストラクタが定義されていません。
6283	(E) Class "名前" requires a constructor
	仮想基底クラスの初期設定に誤りがあります。最派生クラスのコンストラクタが仮想基底クラスのメンバ初期設定子を指定していなければ、その仮想基底クラスはデフォルトコンストラクタを持つか、コンストラクタを持たないかのどちらかでなければなりません。
6284	(E) Undefined class member "名前"
	クラスメンバの初期設定に誤りがあります。クラスメンバのオブジェクトは、コンストラクタを持たないか、デフォルトコンストラクタを持つか、あるいは、クラスメンバを宣言しているクラスのコンストラクタの中でクラスメンバを初期化しなければなりません。
6285	(E) Multiple implicit conversion
	式の値に暗黙のうちに適用される利用者定義変換が複数存在します。
6286	(E) 'public' copy constructor "関数名" required
	コピーコンストラクタを private メンバとして宣言しているのでアクセスできません。
6287	(E) Illegal nonstatic 'const' array initialization
	非静的 const 配列のメンバをコンストラクタで初期設定することはできません。
6288	(E) Constructors cannot initialize indirect base classes or derived members
	間接基底クラスや派生メンバをコンストラクタで初期設定することはできません。
6289	(E) Cannot generate default assignment operator
	デフォルトの代入演算子は生成されません。クラスが const メンバ、リファレンスメンバ、非公開の operator=() を持つクラスのメンバ、あるいは、非公開の operator=() を持つ基底クラスのメンバを持っています。
6290	(E) Cannot generate default copy constructor "名前"
	デフォルトのコピーコンストラクタは生成されません。クラスが名前なしクラスか、すでにクラスのメンバまたは基底クラスのメンバが非公開コピーコンストラクタを持っています。 デフォルトのコピーコンストラクタは生成されません。クラスのメンバまたは基底クラスのメンバが非公開コピーコンストラクタを持っています。
6291	(E) Cannot assign "名前"
	基底クラスのオブジェクトを派生クラスのオブジェクトに代入することはできません。
6292	(E) Cannot access private member "名前"
	非公開メンバを参照することはできません。
6293	(E) Cannot access protected member "名前"
	限定公開メンバは参照することはできません。
6294	(E) Illegal access declaration "名前"
	アクセス宣言において、基底クラスのメンバで宣言したアクセス指定を変更することはできません。
6296	(E) Cannot change the access control of overloaded function "関数名"
	アクセス指定が同じでない多重定義関数をアクセス宣言で調整することはできません。
6297	(E) Cannot change the access control of redefined member "名前"
	派生クラスにおいてメンバが再定義された場合、基底クラスのメンバアクセスを派生クラスで調整することはできません。

6298	(E) 'friend' declaration syntax error フレンド宣言の中で、クラスを定義しようとしています。
6303	(E) Cannot access base class "クラス名" その基底クラスにアクセスできません。
6304	(E) Cannot define 'friend' data members データメンバを friend 宣言しています。
6305	(E) Illegal pure virtual function specifier 純粋仮想関数指定に =0 以外を指定しています。
6306	(E) Cannot access class "名前" member アクセスできないクラスメンバを指定しています。
6307	(E) Cannot access base class member "名前" 基底クラスメンバをアクセスできません。
6308	(E) Constructors cannot have return type コンストラクタにリターン型を指定しています。
6310	(E) Cannot define class member "名前" 別クラスの中で別クラスのメンバを宣言しています。
6311	(E) Illegal constructor/destructor declaration コンストラクタ／デストラクタの宣言に誤りがあります。
6313	(E) Cannot declare nonstatic data member "名前" 非静的データメンバをクラス外部で再宣言することはできません。
6314	(E) Illegal data member initializer データメンバの初期値指定の方法が正しくありません。
6401	(E) 'this' should be referred to in a nonstatic member function キーワード this は、非静的メンバ関数以外で参照することはできません。
6402	(E) "名前" does not exist in this file scope :: 演算子に続く識別子がファイルスコープに存在しません。
6404	(E) "名前" is not a member :: 演算子に続く名前がそのクラスのメンバではありません。
6407	(E) The type of the second operand of '? :' is different from the third operand ? : 演算子の第 2 オペランドと第 3 オペランドの型が異なるクラスの場合は、共通の基底クラスを持たなければなりません。
6408	(E) Base class objects cannot be assigned to derived class objects 基底クラスのオブジェクトは、派生クラスのオブジェクトに代入できません。
6409	(E) '? :' operands should have integral types クラス・オブジェクト、リファレンスは使用できません。
6417	(E) Illegal type conversion 関数呼び出し形式の明示的型変換において、対応するコンストラクタが宣言されていません。
6421	(E) Invalid 'new' operand new 演算子のオペランドが正しくありません。
6425	(E) An array cannot be initialized in a 'new' operator new 演算子のオペランドで指定された配列を、初期設定子を使って初期化することはできません
6428	(E) Invalid 'delete' operand delete 演算子のオペランドが正しくありません。
6430	(E) Cannot convert an object or data to a class object コンスラクタや変換演算子が正しく宣言されていないため、オブジェクトや値をクラスオブジェクトに変換できません。

6431	(E) Cannot convert a virtual base class to a derived class
	仮想基底クラスを派生クラスにキャストすることはできません。
6432	(E) Illegal explicit type conversion
	メンバへのポインタを別のメンバへのポインタ型へ変換することはできません。
6433	(E) '->*' operands type mismatch
	->* 演算子のオペランドの型が合致しません。
6434	(E) The second operand of '->*' and '.*' operator should be a pointer to member of a class
	->* 演算子、.* 演算子の第 2 オペランドはクラスのメンバへのポインタ型でなければなりません。
6435	(E) '.*' operands type mismatch
	.* 演算子のオペランドの型が合致しません。
6437	(E) Unary '&' operand require a qualified name
	単項演算子 & のオペランドで限定した名前がクラスメンバ名でありません。
6441	(E) 'typedef' "名前" used as a constructor or a destructor
	typedef 宣言されたクラスの typedef 名をコンストラクタやデストラクタの名前として使用できません。
6442	(E) Cannot refer to undefined class
	未定義のクラスは式中などで使用できません。
6446	(E) Ambiguous overloaded function call
	多重定義関数呼び出しの関数が曖昧です。
6449	(E) Ambiguous function name referred to
	多重定義関数のアドレスを示す関数名の式が曖昧です。
6450	(E) Illegal function pointer conversion
	関数へのポインタ型を他の型に標準変換できません。
6451	(E) Undeclared function call
	該当する関数が宣言されていません。
6452	(E) Pointer-to-Member cannot be used as an operand of a function call
	メンバ関数へのポインタ型が関数呼び出し演算子以外のオペランドに使用できません。
6453	(E) No return value
	リターン値がありません。
6503	(E) Statements is required in selection statements or iteration statements
	'if'、'switch' 文の中には 1 つ以上の文を含まなくてはなりません。または、反復文の文は宣言であってはなりません。
6504	(E) Declaration with initialization is not executed
	明示的あるいは暗黙の初期設定子を持った宣言を飛び越えています。
6508	(E) Illegal conditional expression
	'while'、'do'、'for' 文の条件式は、算術型、ポインタ型または、算術型かポインタ型への型変換が存在するクラス型でなければなりません。
6513	(E) Illegal jump
	不当なジャンプです。
6514	(E) "xxxx" is not supported
	本バージョンでサポートしていない機能を使用しています。
6515	(E) EC++ useless keyword "キーワード"
	Embedded C++では無効なキーワードです。
7001	(E) Too many identifiers
	識別子の数が制限値を超えています。

10.2 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。

(2)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 */
}
```

(2)説明

(2) 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。

(2) 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 10.3 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 512 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

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

11. モジュール間最適化のエラーメッセージ

11.1 モジュール間最適化のエラーメッセージ

本章では、以下の形式でモジュール間最適化ツールの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、４種に分類されます。

表 11.1 モジュール間最適化ツールのエラーレベル

0 ～ 999	インフォメーション	処理を継続し、ロードモジュールを出力します。
1000 ～ 1999	ウォーニング	処理を継続し、ロードモジュールを出力します。
2000 ～ 2999	エラー	処理を継続します。ロードモジュールは出力しません。
3000 ～ 3999	フェータル	処理を中断します。

表 11.2 モジュール間最適化ツールのエラーメッセージ

0010	<ユニット名 1> IS REPLACED WITH <ユニット名 2> (<ファイル名>)
	<ユニット名 1>を(ファイル名)中の<ユニット名 2>に置き換えました。
0020	<外部名 1> IS RENAMED TO <外部名 2>
	<外部名 1>を<外部名 2>に変更しました。
0030	<外部名> IS DELETED
	<外部名>を削除しました。
0040	DUPLICATE UNIT - （<ユニット名>） IN （<ファイル名>） IS DELETED
	<ユニット名>のユニットを複数見つけたため、<ファイル名>中のユニット名を削除しました。
0050	<外部参照シンボル名> CANNOT BE DEFINED
	<外部参照シンボル名>が見つからないため、強制定義できません。
0060	<外部名> CANNOT BE RENAMED
	<外部名>が見つからないため、変更できません。
0070	<外部名> CANNOT BE DELETED
	<外部名>が見つからないため、削除できません。
0080	<ユニット名> CANNOT BE REPLACED
	<ユニット名>が見つからないため、置き換えができません。
0200	<最適化種別> OPTIMIZE:<セクション名> SECTION IS CREATED
	<最適化種別>の最適化によって<セクション名>を作成しました。
0210	<最適化種別> OPTIMIZE:<ユニット名>.<シンボル名> MOVED <セクション名> SECTION
	<最適化種別>の最適化によって<ユニット名>.<シンボル名>を<セクション名>に移動しました。
0220	<最適化種別> OPTIMIZE:<ユニット名>.<シンボル名> IS CREATED
	<最適化種別>の最適化によって<ユニット名>.<シンボル名>を作成しました。
0230	<最適化種別> OPTIMIZE:<ユニット名>.<シンボル名> IS DELETED
	<最適化種別>の最適化によって<ユニット名>.<シンボル名>を削除しました。

0240	<ユニット名>.<シンボル名> IS OPTIMIZED <ユニット名>.<シンボル名>を最適化しました。
1010	DUPLICATE OPTION/SUBCOMMAND(<オプション／サブコマンド名>) 同じオプションまたはサブコマンドを重複して指定しています。後に指定したオプションまたはサブコマンドが有効になります。
1020	IDENTIFIER CHARACTER EXCEEDS 251(<名前>) 251 文字を超える名前（ユニット名、セクション名、シンボル名）を指定しています。251 文字までが有効になります。
1040	DUPLICATE SYMBOL(<シンボル名>) 外部定義シンボルが重複しています。先に現われた外部定義シンボルが有効になります。
1050	UNDEFINED EXTERNAL SYMBOL(<ユニット名>.<シンボル名>) 未定義の外部シンボルを参照しています。外部参照は無効になり、0 を仮定します。
1060	REDEFINED SYMBOL(<シンボル名>) 定義済みのシンボルを DEFINE オプション／サブコマンドで定義しています。DEFINE オプション／サブコマンドの指定を無視します。
1070	SECTION ATTRIBUTE MISMATCH(<セクション名>) 属性または境界調整数の異なる同名セクションを入力しました。別セクションとして扱います。
1080	RELOCATION SIZE OVERFLOW(<ユニット名>.<セクション名> - <オフセット値>) リロケーションの結果がリロケーションサイズを超えました。
1090	ENTRY POINT MULTIPLY DEFINED 実行開始アドレスの指定があるオブジェクトモジュールを複数指定しています。先に現われた実効開始アドレスの指定が有効になります。
1110	DUPLICATE SECTION NAME(<セクション名>) オプション／サブコマンドで同一セクション名を指定しています。最初に指定したセクション名を有効にします。
1120	ILLEGAL CPU INFORMATION FILE FORMAT CPU 情報ファイルのファイル形式が正しくありません。
1130	CONFLICTING DEVICE TYPE 入力オブジェクトモジュールの対象 CPU と異なる CPU 情報ファイルを指定しています。
1140	SECTION IS NOT IN SAME MEMORY AREA(<セクション名>:xxxx-yyyy) セクションが一つのメモリ領域に入りきらず、xxxx 番地から yyyy 番地が異なるメモリ領域に割り付けられています。
1150	INACCESSIBLE ADDRESS RANGE(<セクション名>) セクションが使用できない領域に割り付けられています。
1160	INVALID CPU OPTION/SUBCOMMAND ロードモジュールファイルをリロケータブル形式に指定して、CPU オプション／サブコマンドを指定しています。
1170	ADDRESS SPACE DUPLICATE セクションが重複しています。
1180	INVALID UDF OPTION/SUBCOMMAND 出力ロードモジュール形式がアブソリュート指定に対し、NOUDF オプション／サブコマンドを指定しています。NOUDF オプション／サブコマンドを無視します。
1190	RELOCATION VALUE IS ODD(<ユニット名>.<セクション名> - <オフセット値>) ディスプレースメントに対するリロケーション結果が奇数になりました。最下位ビットを切り捨てます。
1200	START ADDRESS NOT SPECIFIED FOR SECTION(<セクション名>) START オプション／サブコマンドで指定していないセクションが存在します。

1210	CANNOT FIND SECTION(<セクション名>)
	指定したセクションが見つかりません。
1220	TOO LONG SUBCOMMAND LINE
	ディレクトリ名の置き換えで文字数が 511 文字を超えました。511 文字までを有効とします。
1230	TOO MANY DIRECTORY COMMANDS
	DIRECTORY サブコマンドで 16 個を超えたディレクトリを指定しています。16 個までを有効とします。
1240	NO DEBUG INFORMATION
	デバッグ情報の全くないファイルに対して DEBUG、SDEBUG サブコマンドを指定しています。コンパイル、アセンブル時にデバッグオプションを指定してください。
1250	CANNOT SET ENTRY POINT
	出力ロードモジュールがリロケータブル形式のとき、実行開始アドレスに定数の外部参照シンボルを指定しています。出力ロードモジュールをアブソリュート形式にするか、実行開始アドレスの指定を削除してください。
1260	TOO LONG CHARACTER NUMBER(FSYMBOL)
	FSYMBOL サブコマンドで指定したセクション内のシンボルの文字数が 238 文字を超えています。
1270	EXTERNAL SYMBOL 0（<セクション名>）
	FSYMBOL サブコマンドで指定したセクション内に外部定義シンボルが存在しません。
1280	ILLEGAL SYMBOL REFERENCE （<シンボル名>）
	START サブコマンドで同一アドレスに割り付けたセクション間でシンボルを参照しています。
1600	INVALID SYMBOL_FORBID OPTION
	SYMBOL_FORBID の指定が無効です。
1610	INVALID SAMECODE_FORBID OPTION
	SAMECODE_FORBID の指定が無効です。
1620	INVALID VARIABLE_FORBID OPTION
	VARIABLE_FORBID の指定が無効です。
1630	INVALID FUNCTION_FORBID OPTION
	FUNCTION_FORBID の指定が無効です。
1640	INVALID ABSOLUTE_FORBID OPTION
	ABSOLUTE_FORBID の指定が無効です。
1700	CANNOT FIND SYMBOL SPECIFIED SYMBOL_FORBID(<シンボル名>)
	SYMBOL_FORBID で指定したシンボル名が見つかりません。
1710	CANNOT FIND SYMBOL SPECIFIED SAMECODE_FORBID(<シンボル名>)
	SAMECODE_FORBID で指定したシンボル名が見つかりません。
1720	CANNOT FIND SYMBOL SPECIFIED VARIABLE_FORBID(<シンボル名>)
	VARIABLE_FORBID で指定したシンボル名が見つかりません。
1730	CANNOT FIND SYMBOL SPECIFIED FUNCTION_FORBID(<シンボル名>)
	FUNCTION_FORBID で指定したシンボル名が見つかりません。
1800	<最適化種別> OPTIMIZE:SECTION OVERLAP
	<最適化種別>の最適化でサイズ増加により隣接するセクションと重複しました。<最適化種別>の最適化指定を無効にします。
1810	DIFFERENT SYMBOL ASSIGNED TO A GLOBAL REGISTER AMONG FILES (<シンボル名>:<レジスタ番号>)
	グローバルレジスタに割り付けるシンボル名、レジスタ番号がファイル間で異なります。
1820	STACK ACCESS SIZE OVERFLOW
	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。

1840	RELOCATION VALUE EXISTS(<ユニット名>.<セクション名> - <オフセット値>)
	定数-ラベルの式があります。<ユニット名>を最適化対象外にします。
1850	CANNOT OPTIMIZE REGISTER ALLOCATION(FUNCTION CALL NEST TOO DEEP)
	関数、呼び出しのネストが深すぎるため、レジスタ最適化指定を無視します。
1860	CANNOT OPTIMIZE REGISTER ALLOCATION(REGPARAM 2/3 MIXED)
	コンパイラオプション regparam=2 を指定したオブジェクトと regparam=3 を指定したオブジェクトが混在しています。レジスタ最適化を無視します。
2010	ILLEGAL SUBCOMMAND/OPTION
	不正なサブコマンド名（またはオプション名）を指定しています。
2020	SYNTAX ERROR
	指定されたサブコマンド（またはオプション）に構文上の不正があります。
2030	TOO LONG SUBCOMMAND LINE
	サブコマンドの長さが 511 文字を超えています。
2040	ILLEGAL SUBCOMMAND SEQUENCE
	サブコマンドの指定順序が不正です。
2070	ILLEGAL SECTION NAME(<セクション名>)
	不正なセクション名を指定しています。
2080	ILLEGAL SYMBOL NAME(<シンボル名>)
	不正なシンボル名を指定しています。
2100	TOO MANY INPUT FILES
	入力ファイル数が 65535 個を超えています。
2110	CANNOT FIND FILE(<ファイル名>)
	指定したファイルが見つかりません。
2120	CANNOT FIND UNIT(<ユニット名>)
	指定したユニットが見つかりません。
2130	CANNOT FIND MODULE(<モジュール名>)
	指定したモジュールが見つかりません。
2140	DUPLICATE START ADDRESS SPECIFIED
	同じ先頭アドレスを重複して指定しています。
2170	SUBCOMMAND COMMAND IN SUBCOMMAND FILE
	サブコマンドファイル中に SUBCOMMAND サブコマンドを指定しています。
2190	INVALID ADDRESS(<アドレス>)
	指定したアドレスが CPU のアドレス範囲を超えています。
2200	TOO MANY ROM COMMANDS
	ROM サブコマンドで 64 組を超えたセクションを指定しています。
2210	CANNOT CREATE ABSOLUTE MODULE(<モジュール名>)
	未定義の外部参照シンボルが存在しています。
2220	DIVISION BY ZERO IN RELOCATION VALUE (<ユニット名>.<セクション名> - <オフセット値>)
	0 除算を含むオブジェクトファイルを入力しました。
2600	COMPILER SUPPLEMENTARY INFORMATION FILE MISMATCH(<ファイル名>)
	コンパイラ付加情報ファイルの作成日付がオブジェクトと一致しません。
2610	ILLEGAL DUPLICATE SYMBOL(<シンボル名>)
	外部定義シンボルが重複しています。
2700	ILLEGAL ADDRESS SPECIFIED ABSOLUTE_FORBID OPTION(<アドレス>+<サイズ>)
	絶対アドレスの最適化抑止指定範囲が CPU のアドレス範囲を超えました。

2730	ILLEGAL SAMESIZE SPECIFIED
	共通コードサイズ指定が正しくありません。
2750	NOT SPECIFIED ENTRY SUBCOMMAND
	optimize=symbol_delete を指定していますが、entry サブコマンド指定がありません。
3010	ILLEGAL COMMAND PARAMETER
	不正なコマンドパラメータを指定しています。
3020	CANNOT OPEN FILE(<ファイル名>)
	ファイルをオープンできません。
3030	CANNOT READ INPUT FILE(<ファイル名>)
	ファイルを読み込むことができません。
3040	CANNOT WRITE OUTPUT FILE(<ファイル名>)
	ファイルに書き込むことができません。
3050	CANNOT CLOSE FILE(<ファイル名>)
	ファイルをクローズできません。
3060	ILLEGAL FILE FORMAT(<ファイル名>)
	指定したファイルのフォーマットが不正です。または、RENAME サブコマンドで指定した外部シンボル名が既に存在します。
3070	ILLEGAL RECORD FORMAT(<ファイル名>)
	指定したファイル中に不正なレコードがあります。または、除数が 0 の除算があります。
3080	SECTION ADDRESS OVERFLOW(<セクション名>)
	セクションの割り付けアドレスが CPU で許されるアドレス範囲を超えています。
3090	ADDRESS OVERFLOW
	指定したアドレスが CPU で許されるアドレス範囲を超えています。
3100	MEMORY OVERFLOW
	最適化ツール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。
3110	PROGRAM ERROR(<nnn>)
	最適化ツールの内部処理で何らかの障害が発生しました。プログラムエラー番号(nnn)を確認の上、当社営業担当までご連絡ください。
3120	ILLEGAL START ADDRESS ALIGNMENT(<アドレス>)
	オブジェクトモジュールの境界調整数と矛盾するアドレスを指定しています。
3140	CANNOT FIND SECTION(<セクション名>)
	指定したセクションが見つかりません。
3190	AUTOPAGE SPECIFIED AT NON-PAGE TYPE
	非ページタイプの入力ファイルに対して AUTOPAGE オプション／サブコマンドを指定しています。
3220	PAGE ADDRESS SPECIFIED AT NON-PAGE TYPE
	非ページタイプの入力ファイルに対してページアドレスを指定しています。
3230	SECTION SPECIFIED AT ROM OPTION/SUBCOMMAND DOES NOT EXIST(<セクション名>)
	ROM コマンドで指定したセクションが存在しません。
3250	ILLEGAL START SECTION(<セクション名>)
	START コマンドで指定したセクションの属性が不正です。
3260	CANNOT READ
	指定したファイル（標準入力を含む）から入力できません。
3270	SYMBOL ADDRESS OVERFLOW(<シンボル名>)
	シンボルの割り付けアドレスが CPU のアドレス範囲を超えています。

3280	ILLEGAL ROM SECTION(<セクション名>)
	ROM オプション／サブコマンドの指定で、転送先セクションにサイズ 0 以外のセクションあるいは絶対番地セクションを指定しています。または、転送元と転送先のセクションの属性が異なっています。
3290	INVALID MEMORY MAP
	CPU 情報ファイルと矛盾したメモリに割り付けています。または、異なるメモリ種別にまたがって割り付けています。
3300	ILLEGAL FILE FORMAT (INPUT ABSOLUTE FILE)
	アブソリュートロードモジュールを入力ファイルに指定しています。
3310	ILLEGAL FILE FORMAT (MISMATCH OBJECT FORMAT VERSION)
	オブジェクト形式の異なるファイルを入力しました。
3320	ILLEGAL FILE FORMAT (INPUT MISMATCH CPU TYPE)
	H シリーズ、SH シリーズ以外のファイルを入力しました。
3330	CANNOT OPEN INTERNAL FILE
	中間ファイルをオープンできません。ディスク容量に空きがないか、またはディスクにハード的なエラーがある場合があります。確認の上、再実行してください。
3340	CANNOT CLOSE INTERNAL 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か、またはディスクにハード的なエラーがある場合があります。確認の上、再実行してください
3350	CANNOT DELETE INTERNAL 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か、またはディスクにハード的なエラーがある場合があります。確認の上、再実行してください
3360	CANNOT OUTPUT INTERNAL FILE
	中間ファイルに書き込みできません。ディスク容量に空きがないか、またはディスクにハード的なエラーがある場合があります。確認の上、再実行してください
3370	CANNOT READ INTERNAL FILE
	中間ファイルから入力できません。ディスク容量に空きがないか、またはディスクにハード的なエラーがある場合があります。確認の上、再実行してください
3390	TOO MANY UNITS
	指定したユニット数が 65,535 を超えました。
3400	TOO MANY SECTIONS
	指定したセクション数が 65,535 を超えました。
3700	CANNOT OPEN CPU INFORMATION FILE
	CPU 情報ファイルがオープンできません。
3710	CANNOT OPEN INTERNAL FILE
	中間ファイルがオープンできません。
3720	CANNOT WRITE INTERNAL FILE
	中間ファイルに書き込むことができません。
3730	CANNOT CLOSE INTERNAL FILE
	中間ファイルをクローズできません。
3740	CANNOT EXECUTE(<ロードモジュール名>)
	opt38 または lnk を起動できません。正しくインストールされているか確認してください。
3750	CANNOT CREATE INTERNAL FILE
	中間ファイルを作成することができません。
3760	INTERRUPT BY USER
	処理中に標準入力端末から「(CNTL)C」コマンドによる割り込みを検出しました。

3770	CANNOT ANALYZE OBJECT(<ユニット名>)
	オブジェクトコードを解析することができません。プログラムセクション内の.DATA 制御命令を削除するか、または当該ユニットのアセンブル時の GOPTIMIZE オプション指定を外してください。
3800	TOO MANY EXTERNAL DEFINE SYMBOLS(<ユニット名>)
	ユニット内の定義シンボル数が 65535 を超えました。ファイルを分割するか、または当該ユニットのコンパイル、アセンブル時の GOPTIMIZE オプション指定を外してください。
3810	TOO MANY EXTERNAL REFERENCE SYMBOLS(<ユニット名>)
	ユニット内の参照シンボル数が 65535 を超えました。ファイルを分割するか、または当該ユニットのコンパイル、アセンブル時の GOPTIMIZE オプション指定を外してください。
3820	TOO MANY SECTIONS(<ユニット名>)
	ユニット内のセクション数が 65535 を超えました。ファイルを分割するか、または当該ユニットのコンパイル、アセンブル時の GOPTIMIZE オプション指定を外してください。
3830	<最適化種別> OPTIMIZE:SECTION OVERLAP
	<最適化種別>の最適化でサイズ増加により隣接するセクションと重複しました。

付録

A.　コンパイラが規定する言語仕様

A.1　言語仕様

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)　翻訳

表 A.1　翻訳の仕様

No.	項目	コンパイラの仕様
1	エラー検出時のエラー情報	「10.　コンパイラのエラーメッセージ」を参照。

(2)　環境

表 A.2　環境の仕様

No.	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。

(3)　識別子

表 A.3　識別子の仕様

No.	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	先頭から 250 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	先頭から 250 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。

■注意

250 文字目までが同じで、251 文字目以降が異なる二つの識別子は、同じ識別子とみなします。

■例

（a）　longabcde…ab;(250 文字が a、251 文字目が b)

（b）　longabcde…ac;(250 文字が a、251 文字目が c)

（a）と（b）の二つの識別子は、250 文字目までが一致しているので、同じ識別子とみなされます。

(4) 文字

表 A.4 文字の仕様

No.	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	単なる char 型が signed char 型と同じ値の範囲を持つか、unsigned char 型と同じ値の範囲を持つか	signed char 型と同じ値の範囲を持ちます。

(5) 整数

表 A.5 整数の仕様

No.	項目	コンパイラの仕様
1	整数型の表現方法とその値	表 A.6 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 A.6 整数型とその値の範囲

No.	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

(6) 浮動小数点

表 A.7 浮動小数点の仕様

No.	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「A.3 浮動小数点数の仕様」で説明します。表 A.8 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 A.8 浮動小数点数の限界値

No.	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。

(7) 配列とポインタ

表 A.9 配列とポインタの仕様

No.	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（ノーマルモード） unsigned long 型（アドバンストモード）
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（ノーマルモード） long 型（アドバンストモード）

(8) **レジスタ**

表 A.10 レジスタの仕様

No.	項目	コンパイラの仕様
1	レジスタ変数を割り付けることができるレジスタ	最適化あり（ER3）、ER4、ER5、ER6 最適化なし（ER3）、ER4、ER5 *1
2	レジスタに割り付けることができるレジスタ変数の型	char、unsigned char、short、unsigned short、int、unsigned int、long、unsigned long、float、ポインタリファレンス、データメンバへのポインタ

【注】 *1 （）内のレジスタは noregexpansion オプションを指定した時は、レジスタ変数を割り付けません。

(9) **クラス、構造体、共用体、列挙型、ビットフィールド**

表 A.11 クラス、構造体、共用体、列挙型、ビットフィールドの仕様

No.	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「5.3.2 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。
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。
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

ビットフィールドの割り付け方の詳細については、「5.3.3 ビットフィールド」を参照してください。

(10) **修飾子**

表 A.12 修飾子の仕様

No.	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。

(11) **宣言**

表 A.13 宣言の仕様

No.	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。

(a) 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 a；a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。

(12) **文**

表 A.14 文の仕様

No.	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。

(13) **プリプロセッサ**

表 A.15 プリプロセッサの仕様

No.	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 No.2 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	#pragma abs8、#pragma abs16、 #pragma asm、#pragma endasm、 #pragma entry、#pragma stacksize #pragma indirect、 #pragma inline、#pragma inline_asm、 #pragma pack、#pragma unpack #pragma section、#pragma indirect section、 #pragma abs8 section、#pragma abs16 section、 #pragma global_register、 #pragma regsave、#pragma noregsave をサポートしています。
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。

A.2 サポートしていないライブラリ

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 A.16 に示します。

表 A.16 サポートしていないライブラリ

No.	ヘッダファイル	ライブラリ名
1	signal.h	signal.h、signal、raise
2	stdio.h	remove、rename、tmpfile、tmpnam
3	stdlib.h	abort、exit、getenv、onexit、system
4	time.h	time.h、clock、difftime、time、asctime、ctime、gmtime、localtime

A.3　浮動小数点の仕様

(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** A.1 に示します。

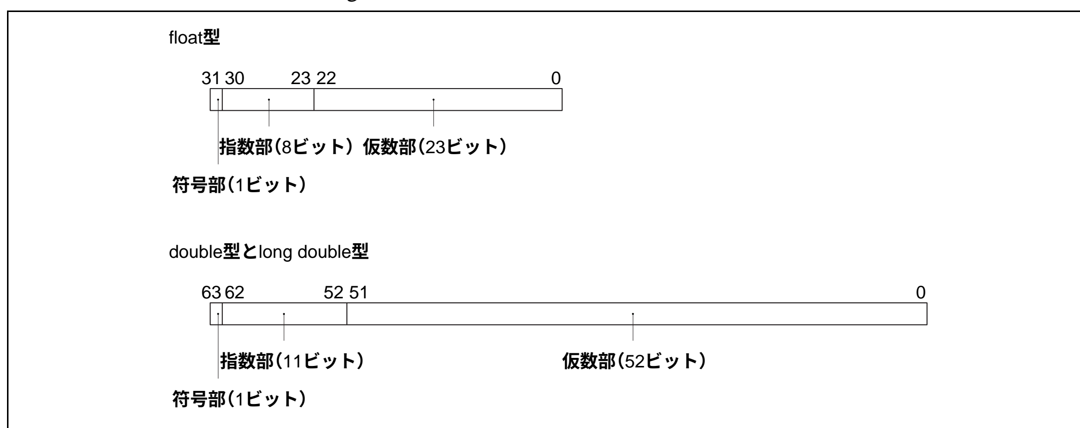


図 A.1　**浮動小数点数の内部表現の構成**

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** A.17 に示します。

■注意

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ので注意してください。

表 A.17　**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**

(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、

$$(-1)^{\text{<符号部>}} \times 2^{\text{<指数部>}-127} \times (1+\text{<仮数部>} \times 2^{-23})$$

となります。

■例

31	30　　23	22　　0
1	10000000	11000000000000000000000

符号：　－

指数：　10000000(2)－127＝1　　　(2)**は2進数を意味します。**

仮数：　1.11(2)＝1.75

値　：　$-1.75 \times 2^1 = -3.5$

（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。

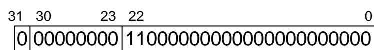
仮数部は、1～2^{23}－1で、実際の仮数は、2^{23}のビットを0と仮定し、その直後に小数点があるものとして解釈します。

非正規化数の表現する値は、

$$(-1)^{\text{<符号部>}} \times 2^{-126} \times (\text{<仮数部>} \times 2^{-23})$$

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**「A.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。

注意　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、

$$(-1)^{<\text{符号部}>}\times2^{<\text{指数部}>-1023}\times(1+<\text{仮数部}>\times2^{-52})$$

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、$1 \sim 2^{52}-1$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

63	62　　52	51　　0
1	00000000000	1110000000000000000000000000000000000000000000000000

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**「A.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。

注意　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++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ないます。

[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、以下の処理を行ないます。

[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、**「10.2　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。

■注意

定数式に関しては、コンパイル時に演算を行ないます。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) 特殊値の演算に関する注意事項

以下、特殊な値（ゼロ、無限大、非数）の演算に関する注意事項を述べます。

[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、「$\pm M \times 10^{\pm N}$」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。
（イ）　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：　$0 \leqq M \leqq 10^{17}-1$、$0 \leqq N \leqq 27$
（ロ）　誤差の限界値
（イ）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、（イ）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。

B. 引数割り付けの具体例

B.1 H8S/2600 用、H8S/2000 用、H8/300H 用
（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);
    :
```

レジスタ	値
R0L	1
R0H	2
R1L	3

[2]
```
int f(int,int,int);
    :
  f(1,2,3);
    :
```

レジスタ	値
R0	1
E0	2
R1	3

[3]
```
int f(long,long);
    :
  f(1,2);
    :
```

レジスタ	値
ER0	1
ER1	2

[4]
```
int f(char,int,int,char);
    :
  f(1,2,3,4);
    :
```

レジスタ	値
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);
    :
```

レジスタ	値
R0	1
ER1	2

引数領域（スタック）

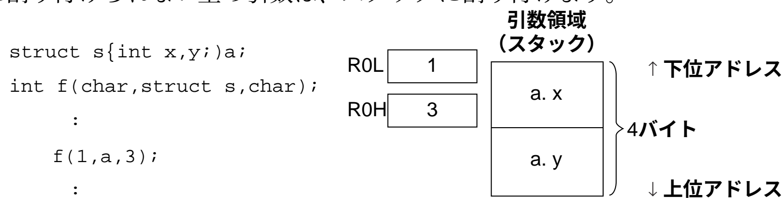
引数領域（スタック）	
無効バイト	2バイト
3	

↑ 下位アドレス

↓ 上位アドレス

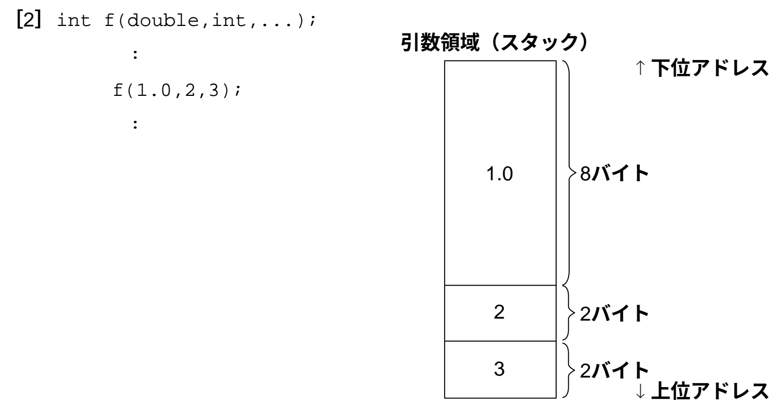
■例 3

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。



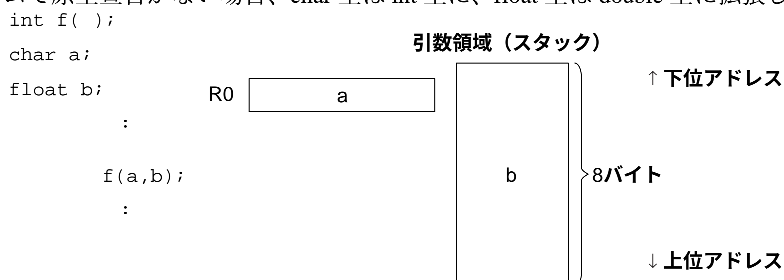
■例 4

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、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。



■例 5

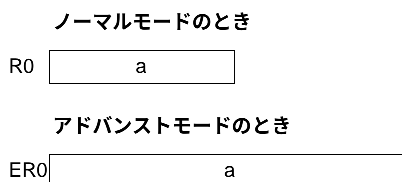
C プログラムで原型宣言がない場合、char 型は int 型に、float 型は double 型に拡張して渡します。



■例 6

ポインタ型は、ノーマルモードでは 2 バイト、アドバンストモードでは 4 バイトの領域に割り付けられます。

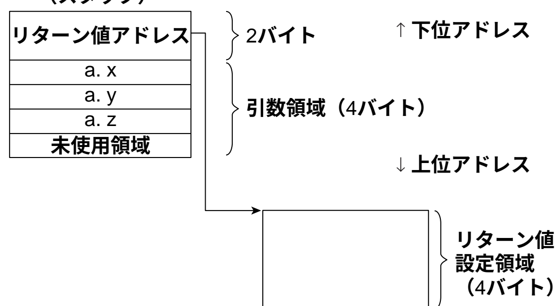
```
int f(int *);
        :
      f(a);
        :
        :
```

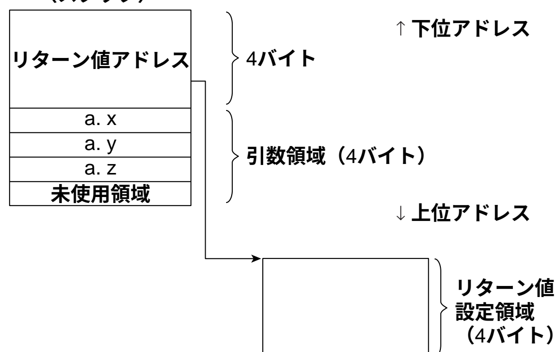


■例 7

関数の返す型が 4 バイトをこえる場合または構造体の場合、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);
          :
          :
```





B.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);
    :
```
R0L: 1
R0H: 2

[2]
```
int f(char,int,char);
    :
  f(1,2,3);
    :
```
R0L: 1
R1: 2
R0H: 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);
    :
```
R0L: 1
R1: 2
R0H: 4

引数領域
（スタック）
3 ｝2**バイト**

■例 3

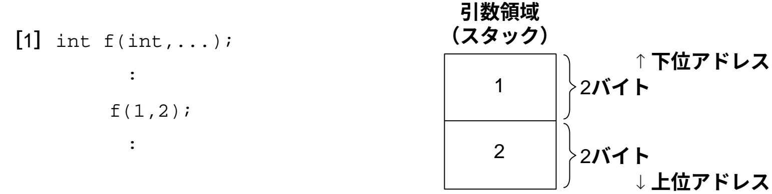
レジスタに割り付けられない型の引数は、スタックに割り付けます。

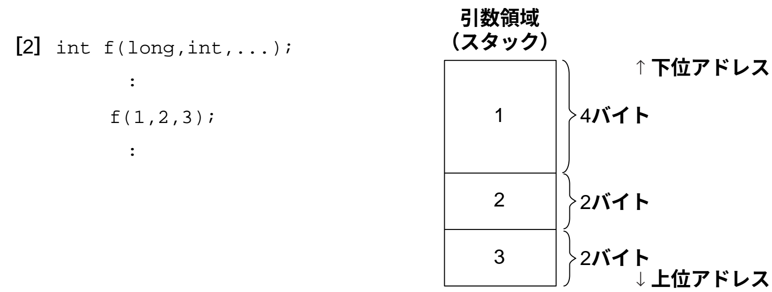
```
int f(char,long,char);
    :
  f(1,2,3);
    :
```
R0L: 1
R0H: 3

引数領域
（スタック）
2 ｝4**バイト**

■例 4

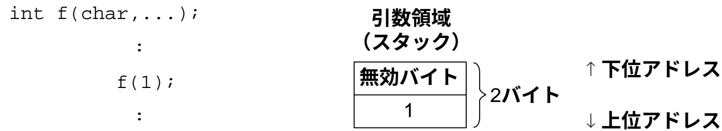
原型宣言により可変個の引数を持つ関数として宣言している場合、対応する型のない引数およびその直前の引数は、宣言順にスタックに割り付けます。





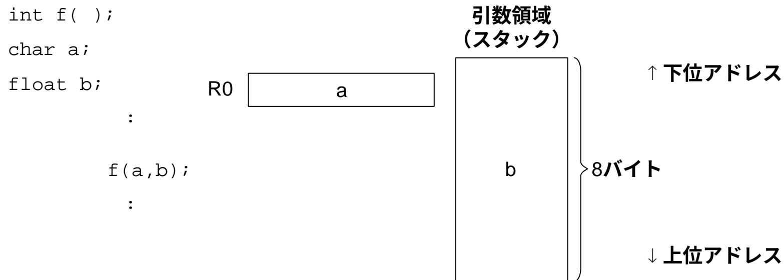
■例 5

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。



■例 6

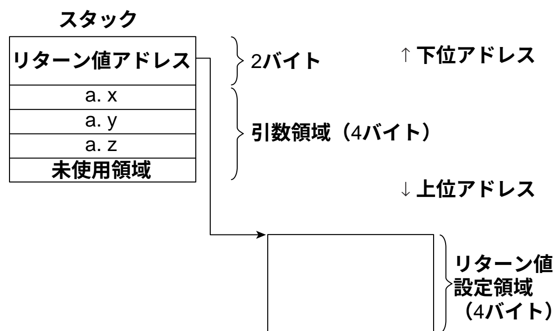
C プログラムで原型宣言がない場合、char 型は int 型に、float 型は double 型に拡張して渡します。



■例 7

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);
          :
          :
```



C. レジスタとスタック領域の使用法

C.1 H8S/2600 用、H8S/2000 用、H8/300H 用アドバンストモード（cpu＝2600a、cpu＝2000a、cpu＝300ha）

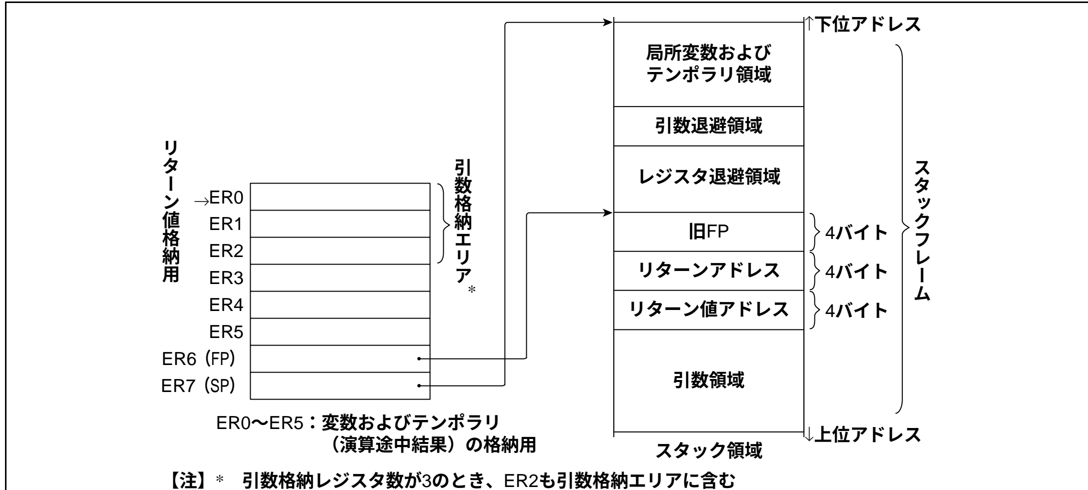


図 C.1 非最適化時のレジスタとスタック領域の使用法
（H8S/2600 用、H8S/2000 用、H8/300H 用アドバンストモード）

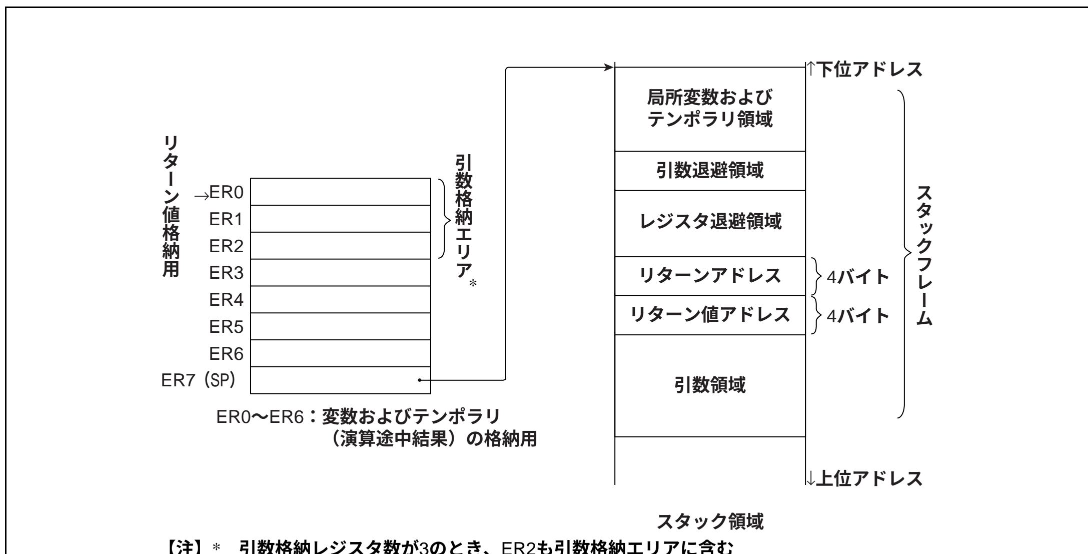


図 C.2 最適化時のレジスタとスタック領域の使用法
（H8S/2600 用、H8S/2000 用、H8/300H 用アドバンストモード）

C.2　H8S/2600 用、H8S/2000 用、H8/300H 用ノーマルモード（cpu＝2600n、cpu＝2000n、cpu＝300hn）

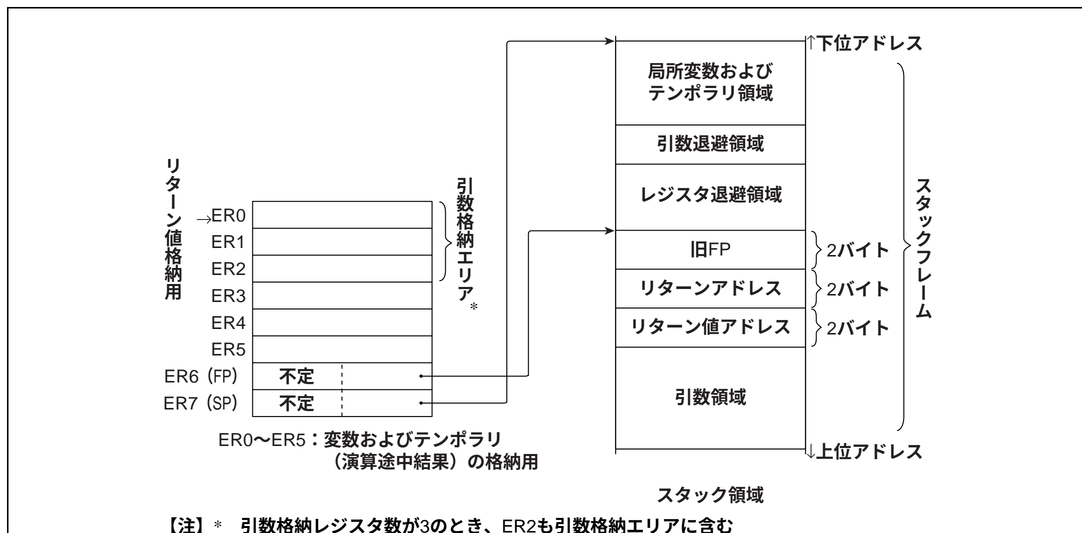


図 C.3　非最適化時のレジスタとスタック領域の使用法
（H8S/2600 用、H8S/2000 用、H8/300H 用ノーマルモード）

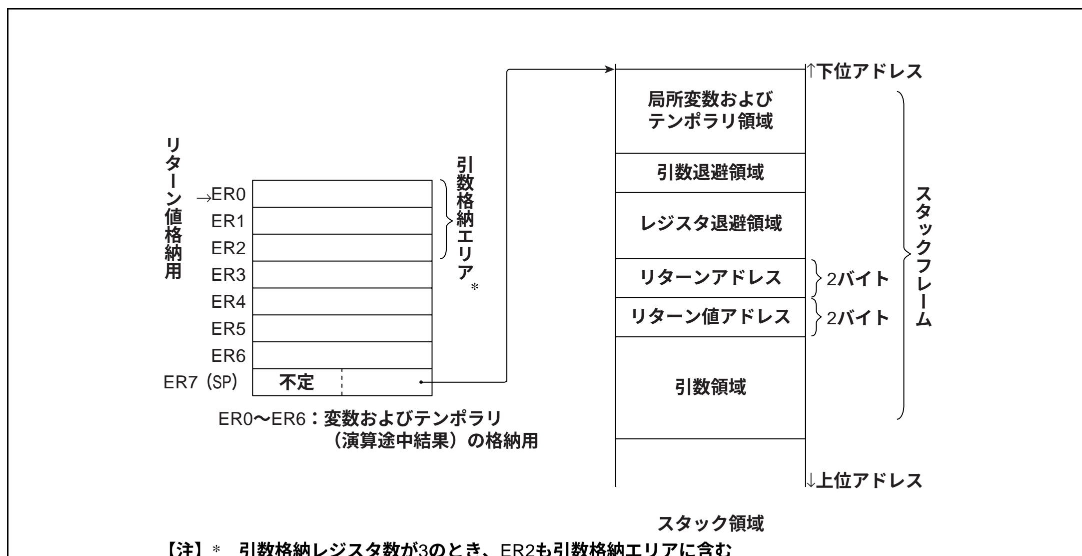


図 C.4　最適化時のレジスタとスタック領域の使用法
（H8S/2600 用、H8S/2000 用、H8/300H 用ノーマルモード）

C.3　H8/300 用（cpu＝300）

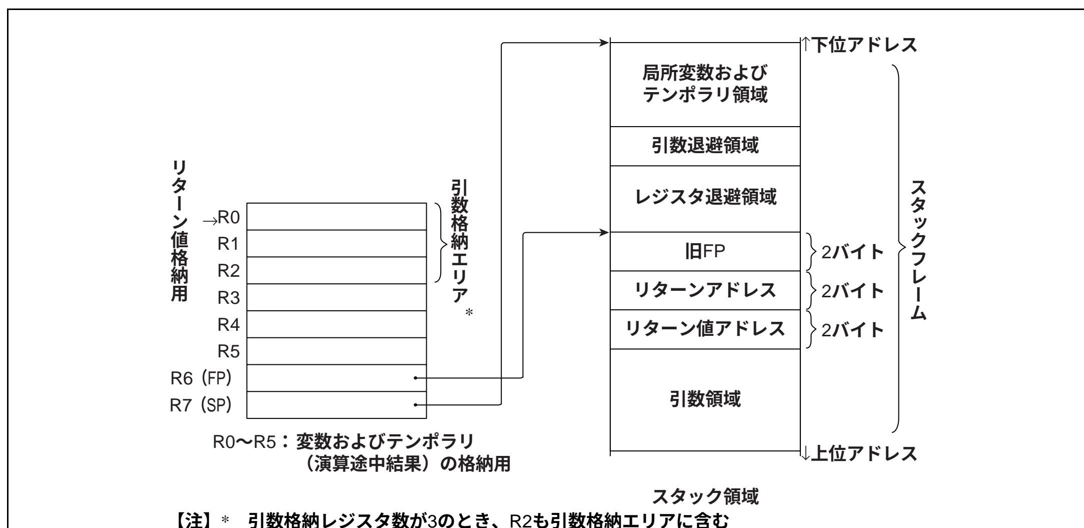


図 C.5　非最適化時のレジスタとスタック領域の使用法（H8/300 用）

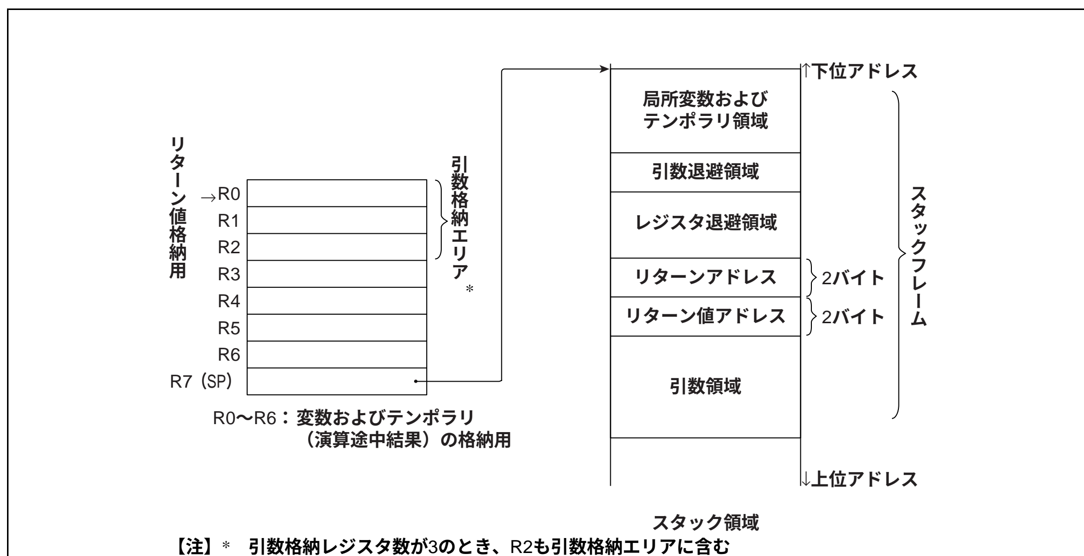


図 C.6　最適化時のレジスタとスタック領域の使用法（H8/300 用）

D. 終了処理関数の作成例

D.1 終了処理の登録と実行（onexit）ルーチンの作成例

終了処理の登録を行うライブラリ onexit 関数の作成法を示します。

onexit 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 *onexit_t ;

int _onexit_count=0 ;
onexit_t (*_onexit_buf[32])(void) ;

extern  onexit_t onexit(onexit_t (*)(void)) ;

onexit_t onexit(onexit_t (*f)(void))
{
     int i;

     for(i=0; i<_onexit_count ; i++)          /* 既に登録されていないかチェック */
          if(_onexit_buf[i]==f)
               return NULL ;
     if (_onexit_count==32)                   /* 登録数の限界値チェック          */
          return NULL ;
     else {
          _onexit_buf[_onexit_count++]=f;     /* 関数のアドレスを登録            */
          return f;
     }
}
```

D.2 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。

以下にプログラム例を示します。

■例

```
#include <setjmp.h>
#include <stddef.h>
typedef void * onexit_t ;
extern int _onexit_count ;

extern onexit_t (*_onexit_buf[32])(void) ;
void _CLOSEALL(void);
void exit(int);
extern jmp_buf _init_env ;
extern int _exit_code ;

void exit(int code)
{
     int i;
     _exit_code=code ;                          /* _exit_code にリターンコードを設定 */
     for(i=_onexit_count-1; i>=0; i--)          /* onexit 関数で登録した関数を順次実行*/
          (*_onexit_buf[i])();
     _CLOSEALL();                               /* オープンした関数を全てクローズ      */
     longjmp(_init_env, 1) ;                    /* setjmp で退避した環境にリターン    */
}
```

注意　上記関数で、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、初期化ルーチン「init」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。

```
#include <setjmp.h>
jmp_buf _init_env;
int     _exit_code;
int main(void);
extern void callmain(void);
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。        */
    /*  exit 関数からのリターン時には処理を終了します。*/
    if(!setjmp(_init_env))
          _exit_code=main();

}
```

D.3　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。

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>
void abort(void);
void _CLOSEALL(void);
void abort(void)
{
      printf(“program is abort !!¥n”);        /* メッセージ出力      */
      _CLOSEALL();                            /* ファイルのクローズ  */
      while(1)   ;                            /* 無限ループ          */
}
```

E. 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 */
                                        /* __CPU__=3:300ha,=5:2600a,=7:2000a*/
#if __CPU__==3| __CPU__==5| __CPU__==7
#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;

      /* ファイル名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;

      /* ファイル名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―                             |
;―――――――――――――――――――――――――――――――――――――――――――――
; H8S/2600、H8S/2000、H8/300H ノーマルモード用(cpu=2600n,cpu=2000n,cpu=300hn)   |
;―――――――――――――――――――――――――――――――――――――――――――――
            .CPU            2600N               ; または 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         |
;                       ――文字入出力を行います―                             |
;―――――――――――――――――――――――――――――――――――――――――――
; H8S/2600、H8S/2000、H8/300H アドバンストモード用(cpu=2600a,cpu=2000a,cpu=300ha)  |
;―――――――――――――――――――――――――――――――――――――――――――
            .CPU            2600A               ; または 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.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       |
;                  ――文字入出力を行います―                                |
;―――――――――――――――――――――――――――――――――――――――――――――――
;                  H8/300用(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
```

F. 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** F.1 に示します。

表 F.1　**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**
2600a:32 2000a:32	0xFFFFFF00～0xFFFFFFFF	0x0～0x7FFF、 0xFFFF8000～0xFFFFFFFF
2600a:28 2000a:28	0xFFFFF00～0xFFFFFFF	0x0～0x7FFF、 0xFFF8000～0xFFFFFFF
2600a[:24] 2000a[:24]	0xFFFF00～0xFFFFFF	0x0～0x7FFF、 0xFF8000～0xFFFFFF
2600a:20 2000a:20	0xFFF00～0xFFFFF	0x0～0x7FFF、 0xF8000～0xFFFFF
2600n 2000n	0xFF00～0xFFFF	—

G. エンコード規則

本コンパイラでは、C++言語仕様の関数、演算子の多重定義機能を実現するため、シンボル名のエンコード処理を行っています。そのエンコード規則を以下に示します。

エンコードの対象になる識別子は、C リンケージ指定されていない C++プログラム中の関数および、静的データメンバです。

- 関数エンコード名=__(関数名情報)__(クラス名情報)(修飾子情報)F(引数型情報)
- 静的メンバエンコード名=__(静的データメンバ名)__(クラス名情報)

(1) 関数名情報

関数名にはソースプログラム上の関数名がエンコード名に埋め込まれます。また、多重定義演算子関数の場合は下記に示す表の文字列がエンコード名に埋め込まれます。

表 G-1 演算子のエンコード

演算子	エンコード	演算子	エンコード
operator () (関数呼び出し)	__cl	operator [] (添字付け)	__vc
operator new	__nw	operator delete	__dl
operator new[]	__nwvc	operator delete[]	__dlvc
operator T (変換関数)	__op<T の型情報>	operator *	__ml
operator /	__dv	operator %	__md
operator +	__pl	operator -	__mi
operator <<	__ls	operator >>	__rs
operator ==	__eq	operator !=	__ne
operator <	__lt	operator >	__gt
operator <=	__le	operator >=	__ge
operator &	__ad	operator |	__or
operator ^	__er	operator &&	__aa
operator ||	__oo	operator !	__nt
operator ~	__co	operator ++	__pp
operator --	__mm	operator =	__as
operator ->	__rf	operator +=	__apl
operator -=	__ami	operator *=	__amu
operator %=	__amd	operator <<=	__als
operator >>=	__ars	operator &=	__aad
operator |=	__aor	operator ^=	__aer
operator ,	__cm	operator ->*	__rm

(2) クラス情報

クラス情報には、クラス名文字数とクラス名文字列がエンコード名に埋め込まれます。

また、入れ子クラスの場合には、Q(入れ子レベル)__(最外側クラス情報)…(最内側クラス情報)がエンコード名に埋め込まれます。

(3) 修飾子情報

修飾子情報には、const, volatile, signed, unsigned の情報が下記に示す表の文字列がエンコードとして埋め込まれます。

表 G-2 修飾子のエンコード

修飾子	エンコード
const	C
volatile	V
unsigned	U
signed（sign char 型以外は無視します）	S

(4) 引数型情報

引数型情報には、基本型、クラス型、派生型がエンコード文字列に埋め込まれます。

表 G-3 引数型のエンコード

型	エンコード
void	v
char	c
short	s
int	i
long	l
float	f
double	d
long double	r
...	e
クラス名	(クラス名文字列数)(クラス名文字列)
ポインタ	P
リファレンス	R
配列	A(要素数)_
メンバへのポインタ	M(クラス名文字列数)(クラス名文字列)
n 番目引数と同一型のとき	Tn(n は引数の n 番目を意味)

H.　リエントラントライブラリ

以下にリエントラントライブラリ一覧表を掲載します。 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。

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 H-1　リエントラントライブラリ一覧

No.	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	△
4	math.h	35	fabs	△
		36	floor	△
		37	fmod	△
5	setjmp.h	38	setjmp	○
		39	longjmp	○
6	stdarg.h	40	va_start	○
		41	va_arg	○
		42	va_end	○
7	stdio.h	43	fclose	×
		44	fflush	×
		45	fopen	×
		46	freopen	×
		47	setbuf	×
		48	setvbuf	×
		49	fprintf	×
		50	fscanf	×
		51	printf	×
		52	scanf	×
		53	sprintf	△
		54	sscanf	△
		55	vfprintf	×
		56	vprintf	×
		57	vsprintf	△
		58	fgetc	×
		59	fgets	×
		60	fputc	×
		61	fputs	×
		62	getc	×
		63	getchar	×
		64	gets	×
		65	putc	×
		66	putchar	×
		67	puts	×
		68	ungetc	×

No.	標準 ｲﾝｸﾙｰﾄﾞﾌｧｲﾙ		関数名	ﾘｴﾝﾄﾗﾝﾄ
7	stdio.h	69	fread	×
		70	fwrite	×
		71	fseek	×
		72	ftell	×
		73	rewind	×
		74	clearerr	×
		75	feof	×
		76	ferror	×
		77	perror	×
8	stdlib.h	78	atof	△
		79	atoi	△
		80	atol	△
		81	strtod	△
		82	strtol	△
		83	rand	×
		84	srand	×
		85	calloc	×
		86	free	×
		87	malloc	×
		88	realloc	×
		89	bsearch	○
		90	qsort	○
		91	abs	○

No.	標準 ｲﾝｸﾙｰﾄﾞﾌｧｲﾙ		関数名	ﾘｴﾝﾄﾗﾝﾄ
8	stdlib.h	92	div	△
		93	labs	○
		94	ldiv	△
9	string.h	95	memcpy	○
		96	strcpy	○
		97	strncpy	○
		98	strcat	○
		99	strncat	○
		100	memcmp	○
		102	strcmp	○
		103	strncmp	○
		104	memchr	○
		105	strchr	○
		106	strcspn	○
		107	strpbrk	○
		108	strrchr	○
		109	strspn	○
		110	strstr	○
		111	strtok	×
		112	memset	○
		113	strerror	○
		114	strlen	○
		115	memmove	○

I. 旧バージョンとの相違点

新バージョン（H8S, H8/300 シリーズ　C/C++コンパイラ Ver.3.0）と旧バージョン（H8S, H8/300 シリーズ C コンパイラ Ver.2.0）の相違点を示します。

I.1 組み込み用拡張機能の追加

(1) entry 関数機能

#pragma entry によりエントリ関数を指定できるようになりました。エントリ関数は、パワーオンリセット時に最初に実行される関数です。

entry 関数機能を使用することにより、SP(スタックポインタ)の初期設定もアセンブラ埋め込み機能を使用せずに C/C++プログラムで記述できます。

(2) セクションアドレス演算子

セクションの先頭アドレス、最終アドレスを参照する演算子(_ _sectop、_ _secend)を追加しました。それに伴い、セクション切り替え機能を使用している場合でもセクションの初期設定を行うライブラリ関数(_INITSCT)を使用できるようになりました。

(3) packed 構造体

pack オプションおよび#pragma pack により、構造体メンバの境界調整を指定できるようになりました。

I.2 追加・改善機能

(1) C++言語機能

C++プログラムをコンパイルできるようになりました。コンパイラは、C プログラムと C++プログラムを lang オプションまたはファイルの拡張子で区別します。

(2) ライブラリ

数学関数（double 型、float 型）ライブラリを標準サポートしました。

また、組み込み向けクラスライブラリ(ios、istream、ostream、iostream、string、complex、new)をサポートしました。

(3) レジスタパラメタ可変化

regparam オプションにより、パラメタ渡し用レジスタの本数を選択できるようになりました。

(4) speed オプションの強化

speed オプションのサブオプションとして、speed=expression を追加しました。

speed=expression 指定時は、ほとんどの演算を実行時ルーチン呼び出しではなく、インラインに展開します。

(5) long 型ビットフィールドのサポート

long 型データのビットフィールドを追加サポートしました。

(6) 制限値拡張

以下の制限値を拡張しました。

- シンボル長（31 文字→250 文字）
- 複文のネスト（32 レベル→256 レベル）
- 繰り返し文（while 文、do 文、for 文）のネスト（32 レベル→256 レベル）
- 選択文（if 文、switch 文）の組み合わせによる文のネスト（32 レベル→256 レベル）
- switch 文のネスト（16 レベル→128 レベル）
- for 文のネスト（16 レベル→128 レベル）
- １行の文字数（4096 文字→8192 文字）
- malloc 確保サイズ（アドバンストモード時：INT_MAX→size_t サイズ）

(7) 最適化リストの出力

モジュール間最適化ツール実行時、シンボルの参照回数や最適化情報リスト出力機能を追加しました。

(8) コマンドラインのリスト出力機能

コマンドライン指定文字列をリスト出力します。

(9) message オプション強化

インフォメーションメッセージレベルの任意の番号を抑止できるようになりました。

(10) 文字コード変換

Latin1 オプションにより、Latin1 コードをソース上に記述できるようになりました。

I.3 言語仕様上の変更

(1) *((int*)p)++のウォーニング化

V2.0 台では、エラーを出力していましたが、V3.0 以降ではウォーニングレベルに変更になりました。

(2) プロトタイプチェック

引数型指定なしのプロトタイプ宣言と引数型指定ありのプロトタイプ宣言を同時指定をした場合、V2.0 台ではエラーを出力していましたが、V3.0 以降では正常にコンパイルできるようになりました。

■例

`void f();` `void f(int);`	**→エラー 2118 を出力**	`void f();` `void f(int);`	**← 正常コンパイル**
V2.0		V3.0	

(3) 構造体先頭の無名ビットフィールド

構造体の先頭に無名ビットフィールドを記述できるようになりました。

■例

```
struct S {
   int   :1;
   int a:1; →エラー 2141 出力
};
```
V2.0

```
struct S {
   int   :1;
   int a:1; →正常コンパイル
};
```
V3.0

(4) 構造体初期値時のエラー抑止

構造体の代入と宣言を同時にできるようになりました。

■例

```
struct S {
   int a,b;
}s1;
 void test()
{
   struct S s2 = s1; ←エラー 2130 出力
}
```
V2.0

```
struct S {
   int a,b;
}s1;
 void test()
{
   struct S s2 = s1; ←正常コンパイル
}
```
V3.0

(5) static 関数の未定義エラー出力条件変更

宣言のみで定義のない static 関数に対して、未参照時のエラー出力を抑止します。

■例

```
static void func();←定義がないので無条件
void test()         にエラー 2143 出力
{
}
```
V2.0

```
static void func();← 参照もないので、
void test()           エラー出力なし
{
}
```
V3.0

(6) //コメントのサポート

C プログラムで//コメントを記述できるようになりました。このため、旧バージョンと解釈が異なる場合があります。

■例

```
int b = a //* コメント */4;
   -a;
    ↓
int b = a/4; -a; と解釈
```
V2.0

```
int b = a //* コメント */4;
   -a;
    ↓
int b = a -a; と解釈
```
V3.0

J. CPU 情報ファイルの作成

モジュール間最適化ツールは、H8S,H8/300 シリーズの CPU ごとのメモリマップに合わせて最適化および再割り付けを行うため、CPU 情報ファイルを使用します。CPU 情報ファイルは、CIA（CPU Information Analyzer）を用いて作成します。本章では、CPU 情報ファイルの作成方法について説明します。

J.1 CIA の機能

CIA は次の３つの機能を持ちます。

（a） CPU情報ファイルの作成
使用するデバイスのメモリマップ情報をファイルに作成します。

（b） CPU情報ファイルの内容表示
作成済みのCPU情報ファイルの内容を確認することができます。

（c） CPU情報の編集（削除／追加）
作成済みのCPU情報ファイルの内容を削除・追加機能を用いて変更することができます。

J.2 CIA の起動方法

CIA を起動するためのコマンドフォーマットは次のとおりです。

```
cia38△<CPU 情報ファイル名>[[△]-work=<テンポラリファイル用ドライブ名およびディレクトリ名>]
```

<CPU 情報ファイル名>として新規または既存の CPU 情報ファイル名を指定します。既存の CPU 情報ファイルを指定した場合は、出力用の CPU 情報ファイルを指定するように要求します。<CPU 情報ファイル名>のファイル形式を省略した場合は、".cpu"を仮定します。

<テンポラリファイル用ドライブ名およびディレクトリ名>は、CIA のテンポラリファイルを作成するドライブ名およびディレクトリ名を指定します。オプション省略時は、CIA を起動したディレクトリにテンポラリファイルを作成します。

J.3　CIA の使用手順と選択メニュー

CIA の使用手順を**図 J.1** に示します。

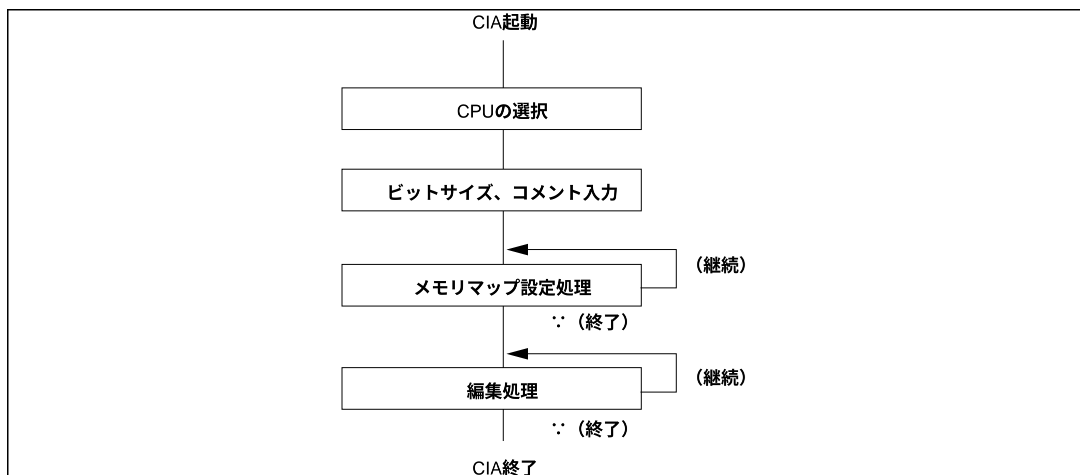


図 J.1　CIA の使用手順

(1)　CPU の選択

CPU 情報のメニューとして次の種類があります。

1:H8/300H-ADVANCED　2:H8/300H-NORMAL　3:H8/300　4:H8/300L
5:H8S/2600-ADVANCED　6:H8S/2600-NORMAL
7:H8S/2000-ADVANCED　8:H8S/2000-NORMAL

(2)　ビットサイズ、コメント入力

ビットサイズは、メモリマップの設定可能範囲なビット幅を 10 進数で指定します。例えば、16 と指定すると H'0〜H'FFFF 番地が設定可能になり、24 と指定すると H'0〜H'FFFFFF 番地が設定可能となります。 H8S シリーズでは、さらにプログラム領域とデータ領域それぞれの範囲のビット幅を指定することができます。プログラム領域にデータ領域より小さいビット幅を指定した場合、メモリマップの設定可能範囲はデータ領域に指定したビット幅になります。プログラム領域にはデータ領域より大きいビット幅を指定できません。

コメントは、CPU 情報の識別用として、文字列を指定することができます。コメントは、127 文字まで設定できます。

ビットサイズ、コメント入力は、新規ファイルの場合のみです。既存ファイルの場合は、(4)の編集処理から始まります。

(3)　メモリマップ設定

CPU 情報の入力のメニューとして次の種類があり、"."(END)が指定されるまでメモリマップの設定処理を繰り返します。

0:ROM　1:EXTERNAL　2:RAM　3:I/O　4:EEPROM　.:END

0〜4 はメモリの種別を指定するためのもので、これらを選択した場合には、メモリの開始アドレス、終了アドレス、ステート数、およびデータバス幅を設定します。

アドレスは 16 進数で指定します。指定可能なアドレス範囲は(2)で指定したビットサイズに依存します。

ステート数、データバス幅は 10 進数で指定します。ステート数の指定可能範囲は 1〜65535 です。また、データバス幅の指定可能範囲は 8〜65528 の 8 の倍数です。

"." (END)を選択した場合は、入力メニュー処理を終了します。

(4) 編集処理

CPU 情報の編集用のメニューとして、次の種類があります。

1:ADD　2:DELETE　3:COMMENT　4:CIA ABORT　.:CIA END

(a) "1"(ADD)を選択した場合は、(3)のメモリマップ設定処理を行います。
(b) "2" (DELETE)を指定した場合は、消去したいアドレスの範囲を番号で入力します。
(c) "3" (COMMENT)を選択した場合は、新規のコメントを入力します。
(d) "4" (CIA ABORT)を選択した場合は、CPU情報ファイルに出力せずに、CIAの処理を終了します。
(e) "." (CIA END)を選択した場合は、メモリマップ情報をCPU情報ファイルに出力し、正常にCIAの処理を終了します。

J.4 CIA の使用例

CIA の使用例を次に示します。下線部はユーザの入力部分です。

(1) H8/310 用 CPU 情報ファイルの作成例

```
>cia38 c310.cpu(RET)  *1

*** NEW FILE ***
   *** CPU MENU ***
   1:H8/300H-ADVANCED   2:H8/300H-NORMAL   3:H8/300   4:H8/300L
   5:H8/2600-ADVANCED   6:H8/2600-NORMAL
   7:H8/2000-ADVANCED   8:H8/2000-NORMAL
   ?   3(RET)  *2
BIT SIZE 16 ? :  16(RET) *3
COMMENT?      :  '97.03.09 H8/310 SAMPLE(RET)  *4
             *** MAP MENU ***
             0:ROM   1:EXTERNAL   2:RAM   3:I/O   4:EEPROM   .:END
             ?  0(RET)  *5
                 *  ROM AREA  START  ADDRESS?  000000(RET)  *6
                                END    ADDRESS?  0027FF(RET)  *7
                    STATE COUNT              ?  2(RET) *8
                    DATA BUS  SIZE           ?  16(RET) *9
                 *  ROM AREA  START  ADDRESS?  .(RET) *10
             *** MAP MENU ***
             0:ROM   1:EXTERNAL   2:RAM   3:I/O   4:EEPROM   .:END
             ?  4(RET)
                 *  EEPROM  START  ADDRESS  ?  006000(RET)
                            END    ADDRESS  ?  007FFF(RET)
                    STATE COUNT              ?  2(RET)
                    DATA BUS  SIZE           ?  16(RET)
                 *  EEPROM  START  ADDRESS  ?  .(RET)
```

```
            *** MAP MENU ***
            0:ROM   1:EXTERNAL   2:RAM   3:I/O   4:EEPROM   .:END
            ?  2(RET)
                *   RAM AREA   START   ADDRESS ?   00FEC0(RET)
                               END     ADDRESS ?   00FFBF(RET)
                    STATE COUNT                ?   2(RET)
                    DATA BUS   SIZE            ?   16(RET)
                *   RAM AREA   START   ADDRESS ?   .(RET)
            *** MAP MENU ***
            0:ROM   1:EXTERNAL   2:RAM   3:I/O   4:EEPROM   .:END
            ?  3(RET)
                *   I/O AREA   START   ADDRESS ?   00FFF8(RET)
                               END     ADDRESS ?   00FFFF(RET)
                    STATE COUNT                ?   3(RET)
                    DATA BUS   SIZE            ?   8(RET)
                *   I/O AREA   START   ADDRESS ?   .(RET)
            *** MAP MENU ***
            0:ROM   1:EXTERNAL   2:RAM   3:I/O   4:EEPROM   .:END
            ?  .(RET)  *11
                   ***** CPU INFORMATION *****
                   CPU : H8/300
                   '91.03.09 H8/310 SAMPLE
                   PROGRAM      AREA       BITSIZE : 16
                   DATA         AREA       BITSIZE  : 16
                     No         Device     StartEndStateBus
                      1 :       ROM AREA   :000000 - 0027FF      2      16
                      2 :       EEPROM     :006000 - 007FFF      2      16
                      3 :       RAM AREA   :00FEC0 - 00FFBF      2      16
                      4 :       I/O AREA   :00FFF8 - 00FFFF      3       8
       ** EDIT MENU **
       1:ADD    2:DELETE   3:COMMENT   4:CIA ABORT   .:CIA END
       ?  .(RET)
  *** CIA COMPLETED ***
```

【注】 *1 CIA の起動時に新規作成ファイルを指定します。
*2 CPU の種類を指定します。
*3 ビットサイズを 10 進数で指定します。省略した場合は、表示したデフォルトを設定します。
*4 コメントを入力します。省略した場合は、空白を表示します。128 文字以上入力した場合は、ウォーニングメッセージを出力し、128 文字目以降を無視します。
*5 入力メニューに対し、メモリ種別を番号で入力します。
*6 当該メモリの先頭アドレスを 16 進数で入力します。
*7 当該メモリの終了アドレスを 16 進数で入力します。
*8 当該メモリのステート数を 10 進数で入力します。
*9 当該メモリのデータバス幅を 10 進数で入力します。
*10 当該メモリ種別入力の入力を終了したい場合は"."を指定します。
*11 入力メニューを終了すると自動的に編集メニューを表示します。

J.5　制限事項一覧

表 J.1 に CIA の制限事項一覧を示します。CIA ではこれらの制限値を超える処理はできません。

表 J.1　CIA の制限事項

No	項目	制限値	備考
1	ビットサイズ	10 進数による指定 指定範囲は 16～32	－
2	アドレス指定	16 進数による指定 指定範囲はビットサイズによる	ビットサイズ 16 の場合 H'0～H'FFFF
3	ステート数	10 進数による指定 指定範囲は 1～65535	ウェイトステートを挿入する場合は、ウェイトステート数を含めた値を設定してください。
4	データバス幅	10 進数による指定 指定範囲は 8～65528 の 8 の倍数	－
5	コメントの長さ	127 文字まで	－
6	MAP 情報数	最大 65535 個	ただし、CIA が稼動するシステムのメモリ容量によって制限されます。無効領域も情報数に含みます。

J.6 CIA のエラーメッセージ

エラー番号	エラーメッセージ エラー内容

表 J.2 CIA のエラーメッセージ

エラー番号	エラーメッセージ / エラー内容
7001	CAN NOT GET MEMORY SPACE
	CIA で使うメモリを確保できません。
7002	CAN NOT OPEN INPUT CPU INFORMATION FILE
	指定された既存の CPU 情報ファイルがオープンできません。
7003	CAN NOT OPEN OUTPUT CPU INFORMATION FILE
	指定された出力用の CPU 情報ファイルがオープンできません。
7004	CAN NOT READ
	ファイルの読み出しができません。
7005	CAN NOT WRITE
	ファイルの書き込みができません。
7006	CAN NOT CLOSE
	出力する CPU 情報ファイルがクローズできません。
7007	INVALID CPU INFORMATION
	CPU 情報ファイルの内容に誤りがあります。
7008	SYNTAX ERROR
	コマンドラインの指定内容に誤りがあります。
8001	COMMENT LINE TOO LONG
	指定したコメントの文字数が 127 文字を超えています。
8002	ADDRESS RE-USE
	アドレス範囲が重複しています。
8003	ADDRESS SIZE OVERFLOW
	ビットサイズを超えたアドレスを指定しています。
8004	INVALID VALUE
	許容範囲外の数値を指定しています。
8005	INVALID CHARACTER
	使用できない文字を指定しています。
8006	INVALID END ADDRESS
	終了アドレスが先頭アドレスより小さい値になっています。

K. ASCII コード一覧表

表 K.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	ˆ	n	～
F	SI	US	／	?	O	_	o	DEL

索引

日本語索引

あ行

アセンブラ埋め込み......86

アセンブラ埋め込みインライン展開......85

アセンブリプログラムとの結合......66

– 外部名の相互参照方法......66
– 関数呼び出しのインタフェース......68
– スタックポインタに関する規則......68
– 引数とリターン値の設定、参照に関する規則......71
– レジスタに関する規則......69
– スタックフレームの割り付け、解放に関する規則......68

後処理データ領域......56

位置指示子......117

インクルードファイル

– CH38（環境変数）......52
– include（コンパイラオプション）......33
– インクルードファイル基点ディレクトリ......34
– インクルードファイルの検索方法......443
– 標準インクルードファイル......111

インターナルレベル（エラーメッセージレベル）......405

インフォメーションレベル（エラーメッセージレベル）......405

インライン展開

– #pragma inline......84
– #pragma inline_asm......85
– inline（コンパイラサブオプション）......43

ウォーニングレベル（エラーメッセージレベル）......405,431

エラー指示子......117

エラー情報（コンパイルリスト）......16

エラーメッセージ

– CIA のエラーメッセージ......482
– コンパイラのエラーメッセージ......405
– C ライブラリ関数のエラーメッセージ......428
– エラーレベル（エラーメッセージレベル）......405,431
– モジュール間最適化のエラーメッセージ......431

エンコード規則......470

エントリ関数の作成......79

オブジェクトフォーマット ... 395
オブジェクト情報（コンパイルリスト） ... 19
オブジェクトプログラムの構造 ... 55
オプション（コンパイラ） ... 29
オプション（モジュール間最適化） ... 383,391

か行

外部定義シンボルリスト ... 27
外部名 ... 66
拡張機能 ... 75
– #pragma 拡張子 ... 75
– セクションアドレス演算子 ... 90
– 組み込み関数 ... 91
仮数部 ... 444
仮想関数表 ... 62
仮想関数表領域 ... 56
可変個の実引数アクセス用ライブラリ ... 168
環境変数 ... 52,404
関数アドレス領域 ... 55
関数のインライン展開 ... 84
関数メンバへのポインタ ... 59
関数呼び出しのインタフェース ... 68
基数 ... 116
起動方法（コンパイラ） ... 7
起動方法（モジュール間最適化） ... 10
旧バージョンとの相違点 ... 474
– 組み込み用拡張機能の追加 ... 474
– 追加・改善機能 ... 474
– 言語仕様上の変更 ... 475
境界調整数 ... 58
境界調整数の指定 ... 40,83
強制定義シンボルリスト ... 28
共用体型 ... 60
組み込み関数 ... 91
クラス型 ... 60
グローバルオブジェクトの後処理 ... 364
グローバルオブジェクトの初期処理 ... 364
グローバル変数のレジスタ割り付け ... 88
限界値 ... 53
言語仕様 ... 439

– 翻訳の仕様 439
– 環境の仕様 439
– 識別子の仕様 439
– 文字の仕様 440
– 整数の仕様 440
– 整数型とその値の範囲 440
– 浮動小数点数の仕様 441
– 浮動小数点数の限界値 441
– 配列とポインタの仕様 441
– レジスタの仕様 442
– クラス、構造体、共用体、列挙型、ビットフィールドの仕様 442
– 修飾子の仕様 442
– 宣言の仕様 442
– 文の仕様 443
– プリプロセッサの仕様 443
– サポートしていないライブラリ 443
– 浮動小数点数演算の仕様 448

構造体型 60

コーディング上の注意事項 105

コマンドラインの形式（コンパイラ） 29

コマンドラインの形式（モジュール間最適化） 383

コンパイラの概要 2

コンパイラの実行 7

コンパイルリスト 14

さ行

サイズの算出法 358,361
– スタック領域サイズの算出法 362
– 静的領域サイズの算出法 358
– ヒープ領域サイズの算出法 363

最適化 43,383
– モジュール間最適化 383
– コンパイラの最適化 43
– スピード最適化 43

最適化情報リスト 22

サブコマンドファイル 51,383

指数部 444

システム組み込み 357
– メモリ領域の割り付け 358
– ライブラリ関数使用時の実行環境の設定 367
– 実行環境の設定 364

実行時ルーチン 2,359

終了処理関数の作成例 459
– abort 関数 461

– exit 関数 460
– onexit 関数 459
初期化データセクションのアドレス領域 55
初期化データ領域 55
初期処理データ領域 56
初期設定ルーチン 365,369
処理系定義 116
シンボル割り付け情報（コンパイルリスト） 17
シンボル最適化情報（最適化情報リスト） 23
シンボル参照回数情報（最適化情報リスト） 24
数値計算用ライブラリ 133,149
スカラ型 58
スタックセクションの作成 79
スタックフレーム 68
スタックポインタ 68
スタック領域 56, 79
– 下位アドレス 68
– 上位アドレス 68
スタック領域サイズの算出法 362
ストリーム入出力 114
ストリーム入出力用クラスライブラリ 256
正規化 116
静的領域サイズの算出法 358
静的領域の割り付け 358
セクション 55
– セクション初期化用テーブル 366
– 初期化データ領域 55
– 定数領域 55
– プログラム領域 55
– 未初期化データ領域 55
– 関数アドレス領域 55
– 初期化データアドレス領域 55
– 未初期化データアドレス領域 56
セクションアドレス演算子 90
セクション切り替え 80
ソースリスト情報（コンパイルリスト） 14

た行

短絶対アドレス 47, 81
短絶対アドレスのアクセス範囲 469
注意事項 105

– コーディング上の注意事項 105
– トラブル発生時の対処方法 110
– ビット操作命令に関する注意事項 107

チュートリアル 5

低水準インタフェースルーチン 374
– lseek ルーチン 380
– close ルーチン 377
– open ルーチン 376
– read ルーチン 378
– sbrk ルーチン 381
– write ルーチン 379
– 低水準インタフェースルーチンの作成例 462

定数領域 55

テキストファイル 115

データの内部表現 58

データメンバへのポインタ 59

デバッグ情報 39,397

動的領域 361
– スタック領域 361
– 動的領域の割り付け方 363
– ヒープ領域 363

統計情報（コンパイルリスト） 21

トラブル発生時の対処方法 110

な行

内部表現 58

入出力 374
– 入出力の考え方 374
– 標準入出力の初期設定 371
– 低水準インタフェースルーチン 374

入出力用ライブラリ 172

ヌル文字 115

は行

バイナリファイル 115

配列型 60

引数 71
– 型変換の規則 71
– スタック上の引数領域 73
– 引数格納用レジスタ 73
– 引数の渡し方 71
– 引数の割り付け領域 72
– 引数割り付けの具体例 450

引数格納用レジスタ数の指定 40

非数 445
非正規化数 444
ビットフィールド 64
ヒープ領域 56, 363
標準インクルードファイル 1, 112
標準入出力ファイル 115
標準ライブラリの概要 2
標準 C ライブラリ 111
ファイルアクセスモード 116
ファイル拡張子 13
ファイル終了指示子 117
ファイルの入出力 374
– close ルーチン 377
– open ルーチン 376
– read ルーチン 378
– sbrk ルーチン 381
– write ルーチン 379
– lseek ルーチン 380
ファイルポインタ 114
ファイル名の付け方 13
フェータルレベル（エラーメッセージレベル） 405,431
複素数計算用クラスライブラリ 307
複合型 60
符号拡張 64
浮動小数点数 441
浮動小数点数演算の仕様 448
– 丸め 448
– オーバフロー 448
– アンダフロー 448
– 無効演算 448
– 特殊値の演算に関する注意事項 448
– 10 進演算と内部表現 448
浮動小数点数の仕様 444
– 浮動小数点数の限界値 441
– 浮動小数点数の内部表現 444
– 符号部 444
– 仮数部 444
– 指数部 444
– 浮動小数点数の表現する値の種類 444
– 正規化数 444
– 非正規化数 444
– 無限大 444

– 非数 445
ブラウザ 1
プログラム作成上の注意事項 105
プログラム診断用ライブラリ 119
プログラムの開発手順 1
プログラムの構成例 364,367
プログラムの制御移動用ライブラリ 165
プログラム領域 55
ベクタテーブル 365
ポインタ 58

ま行

マクロ名の定義 34
丸め 116
未初期化データ領域 55
未初期化データセクションのアドレス領域 56
無限大 444
無効演算 448
メモリ管理用ライブラリ 305
メモリ間接の関数呼び出し 47, 82
メモリ領域の割り付け 358
– ROM、RAM の割り付け 359
– サイズの算出法 358,361
– 初期化データ領域の割り付け 360
– 動的領域の割り付け 361
– 静的領域の割り付け 358
– メモリ割り付け例 360
モジュール間最適化 3,44,383
モジュール間最適化のエラーメッセージ 431
モジュール間最適化のオプション・環境変数 383
文字列出力領域 37
文字列操作用クラスライブラリ 331
文字列操作用ライブラリ 233

や行

予約語 374

ら行

ライブラリ 2,111,255
– C ライブラリ関数 2,111
– C ライブラリ関数の初期設定 369
– サポートしていないライブラリ 443
– EC++クラスライブラリ 255

– 実行時ルーチン 2, 359
– 低水準インタフェースルーチン 374
– ライブラリの初期化 369
– ライブラリのエラーメッセージ 428
ライブラリ関数の説明で使用する用語 114
ライブラリ使用時の注意事項 117
ライブラリの種類 2
リエントラントライブラリ 472
リスト 14,22,25
– コンパイルリスト 14
– 最適化情報リスト 22
– リンケージリスト 25
リターンコード 115
リターン値に対する一般的な規則 71
– リターン値アドレス 74
– リターン値格納用レジスタ 74
– リターン値の設定場所 74
リファレンス型 59
リンケージ機能 391
リンケージリスト 25
– 入力情報 25
– リンケージマップリスト 26
– 外部定義シンボルリスト 27
– 未定義シンボルリスト 27
– 変更／削除シンボルリスト 28
– 強制定義シンボルリスト 28
レジスタ退避／回復コード制御 88
レジスタの仕様 442
– 引数格納用レジスタ 73
– リターン値格納用レジスタ 74
– レジスタとスタック領域の使用法 456
– レジスタ保証規則 69
– レジスタ変数の拡張 45
– グローバル変数のレジスタ割り付け 88

わ行

割り込み関数の作成 76

英数字索引

2000a（コンパイラサブオプション） 50
2000n（コンパイラサブオプション） 50
2600a（コンパイラサブオプション） 50
2600n（コンパイラサブオプション） 50
300（コンパイラサブオプション） 50
300ha（コンパイラサブオプション） 50
300hn（コンパイラサブオプション） 50
300l（コンパイラサブオプション） 50
300reg（コンパイラサブオプション） 50
$ABS8B（セクション名） 55
$ABS8C（セクション名） 55
$ABS8D（セクション名） 55
$ABS16B（セクション名） 55
$ABS16C（セクション名） 55
$ABS16D（セクション名） 55
$INDIRECT（セクション名） 55
#pragma abs8 81
#pragma abs8 section 80
#pragma abs16 81
#pragma abs16 section 80
#pragma asm 86
#pragma endasm 86
#pragma entry 79
#pragma global_register 88
#pragma indirect 82
#pragma indirect section 80
#pragma inline 84
#pragma inline_asm 85
#pragma interrupt 76
#pragma noregsave 88
#pragma pack 83
#pragma regsave 88
#pragma section 80
#pragma stacksize 79
#pragma unpack 83
_sbrk_size 363
__secend 90

__sectop 90
_call_end 364
_call_init 364
_CLOSEALL 373
INIT 365,369
_INITLIB 369
_INITSCT 364
_INIT_IOLIB 371
_INIT_LOWLEVEL 370
_INIT_OTHERLIB 372
_IOFBF 172
_IOLBF 172
_IONBF 172

A

abort 関数 461
abs 関数 231,316,328
abs8（コンパイラオプション） 47
abs16（コンパイラオプション） 47
absolute_fobid（モジュール間最適化オプション／サブコマンド） 388
acos 関数 135
acosf 関数 151
align_section（リンケージサブコマンド） 400
allocation（コンパイラサブオプション） 41
and_ccr（組み込み関数） 94
and_exr（組み込み関数） 96
arg 316,328
ASCII コード 483
asin 関数 135
asinf 関数 152
asmcode（コンパイラサブオプション） 36
assert.h 119
assert マクロ 119
atan 関数 136
atanf 関数 152
atan2 関数 137
atan2f 関数 153
atof 関数 219
atoi 関数 220

atol 関数 221
auto（コンパイラサブオプション） 46

B

B（セクション名） 55
boolalpha（ios クラスマニピュレータ） 268
branch（モジュール間最適化オプションパラメタ） 385
browser（コンパイラオプション） 39
bsearch 関数 229
bss（コンパイラサブオプション） 42
BUFSIZ 172
byteenum（コンパイラオプション） 45

C

c（コンパイラサブオプション） 32
C（セクション名） 55
C$BSEC（セクション名） 56
C$DSEC（セクション名） 55
C$END（セクション名） 56
C$INIT（セクション名） 56
C$VTBL（セクション名） 56
C ライブラリ関数のエラーメッセージ 428
calloc 関数 225
case（コンパイラオプション） 46
ceil 関数 146
ceilf 関数 162
CH38 52
CH38TMP 52
char 型 58
CHAR_BIT 131
CHAR_MIN 131
CHAR_MAX 131
check_section（リンケージサブコマンド） 400
chgincpath（コンパイラオプション） 34
cia38 477
clearerr 関数 213
close ルーチン（低水準インタフェースルーチン） 377,464
cmncode（コンパイラオプション） 46
code（コンパイラオプション） 36
comment（コンパイラオプション） 36

conj 316,328
const（コンパイラサブオプション） 37,42
const 型 107
cpp（コンパイラサブオプション） 32
cpu（コンパイラオプション） 50
cpu（リンケージサブコマンド） 400
cpucheck（リンケージサブコマンド） 401
cpuexpand（コンパイラオプション） 48
CPU 情報ファイル 477
CPU／動作モード 50
－ H8/300 50
－ H8/300H 用アドバンストモード 50
－ H8/300H 用ノーマルモード 50
－ H8S/2000 用アドバンストモード 50
－ H8S/2000 用ノーマルモード 50
－ H8S/2600 用アドバンストモード 50
－ H8S/2600 用ノーマルモード 50
cos 関数 138,316,328
cosf 関数 154
cosh 関数 139,317,329
coshf 関数 155
ctype.h 120

D

D（セクション名） 55
dadd（組み込み関数） 103
data（コンパイラサブオプション） 37,42
DBL_DIG 130
DBL_EXP_DIG 130
DBL_MANT_DIG 130
DBL_MAX 129
DBL_MAX_EXP 129
DBL_MAX_10_EXP 129
DBL_MIN 129
DBL_MIN_EXP 129
DBL_MIN_10_EXP 130
DBL_NEG_EPS 130
DBL_NEG_EPS_EXP 130
DBL_POS_EPS 130
DBL_POS_EPS_EXP 130

debug（コンパイラオプション） 39
debug（リンケージサブコマンド） 397
dec（ios クラスマニピュレータ） 270
define（コンパイラオプション） 34
define（リンケージサブコマンド） 394
delete（リンケージサブコマンド） 402
directory（リンケージサブコマンド） 399
div 関数 231
div_t 218
double 型 58
double_complex クラス 319
double_complex::real 320
double_complex::imag 320
double_complex::operator= 320,321
double_complex::operator+= 320,321
double_complex::operator-= 321
double_complex::operator*= 321,322
double_complex::operator/= 321,322
dsub（組み込み関数） 104

E

EC++クラスライブラリ 255
echo（リンケージサブコマンド） 403
ecpp（コンパイラオプション） 33
EDOM 132,133
eepmov（組み込み関数） 100
eepmov（コンパイラオプション） 49
elf（リンケージサブコマンド） 395
endl（ostream クラスマニピュレータ） 299
ends（ostream クラスマニピュレータ） 299
entry（リンケージサブコマンド） 393
enum 型 58
EOF 115,172
ERANGE 132,133
errno 118,132
errno.h 132
euc（コンパイラオプション） 35
euc（コンパイラサブオプション） 37
exchange（リンケージサブコマンド） 401

exclude（リンケージサブコマンド） 398
exit 関数 460
exit（リンケージサブコマンド） 403
exp 関数 141,317,329
expansion（コンパイラサブオプション） 41
expf 関数 157
expression（コンパイラサブオプション） 43

F

fabs 関数 146
fabsf 関数 162
fclose 関数 176
feof 関数 214
ferror 関数 215
fflush 関数 177
fgetc 関数 197
fgets 関数 198
FILE 構造体 114
fixed（ios クラスマニピュレータ） 271
float 型 58
float_complex クラス 307
float_complex::real 308
float_complex::imag 308
float_complex::operator= 308,309
float_complex::operator+= 309
float_complex::operator-= 309
float_complex::operator*= 309,310
float_complex::operator/= 309,310
float.h 129
floor 関数 147
floorf 数 163
FLT_DIG 130
FLT _EXP_DIG 130
FLT_GUARD 129
FLT _MANT_DIG 130
FLT _MAX 129
FLT _MAX_EXP 129
FLT _MAX_10_EXP 129
FLT _MIN 129

FLT _MIN_EXP 129
FLT _MIN_10_EXP 130
FLT _NEG_EPS 130
FLT _NEG_EPS_EXP 130
FLT_NORMALIZE 129
FLT _POS_EPS 130
FLT _POS_EPS_EXP 130
FLT_RADIX 129
FLT_ROUNDS 129
flush（ostream クラスマニピュレータ） 299
fmod 関数 148
fmodf 数 164
fopen 関数 178
form（リンケージサブコマンド） 397
fprintf 関数 182
fputc 関数 199
fputs 関数 200
fread 関数 208
free 関数 226
freopen 関数 179
frexp 関数 142
frexpf 数 158
fsymbol（リンケージサブコマンド） 398
fscanf 関数 187
fseek 関数 210
ftell 関数 211
function_call（モジュール間最適化オプションパラメタ） 385
function_forbid（モジュール間最適化オプション／サブコマンド） 388
fwrite 関数 209

G

getc 関数 201
getchar 関数 202
gets 関数 203
get_ccr（組み込み関数） 94
get_exr（組み込み関数） 96
get_imask_ccr（組み込み関数） 93
get_imask_exr（組み込み関数） 95
getline（string クラスマニピュレータ） 355

goptimize（コンパイラオプション） 44

H

H38CPU（コンパイラ環境変数） 52
H8S/2000 シリーズ 2
H8S/2600 シリーズ 2
H8/300H シリーズ 2
H8/300L シリーズ 2
H8/300 シリーズ 2
hex（ios クラスマニピュレータ） 270
HLNK_LIBRARY1 404
HLNK_LIBRARY2 404
HLNK_TMP 404
HUGE_VAL 133

I

ifthen（コンパイラサブオプション） 46
imag 315,327
include（コンパイラオプション） 33
indirect（コンパイラオプション） 47
indirect.h 82
information（モジュール間最適化オプション／サブコマンド） 389
inline（コンパイラサブオプション） 43
input（リンケージサブコマンド） 392
int 型 58
internal（ios クラスマニピュレータ） 270
INT_MAX 131
INT_MIN 131
int_type 256
ios クラス 263
ios クラスマニピュレータ 267
ios::operator void* 264
ios::operator ! 264
ios::rdstate 264
ios::clear 265
ios::setstate 265
ios::good 265
ios::eof 265
ios::bad 265
ios::fail 266

ios::tie 266
ios::rdbuf 266
ios::copyfmt 266
ios_base クラス 258
ios_base::Init クラス 257
ios_base::Init::init_cnt 257
ios_base::fmtflags 259
ios_base::iostate 259
ios_base::openmode 260
ios_base::seekdir 260
ios_base::_ec2p_init_base 260
ios_base::_ex2p_copy_base 260
ios_base::flags 261
ios_base::setf 261
ios_base::unsetf 261
ios_base::fill 262
ios_base::precision 262
ios_base::width 262
isalnum 関数 122
isalpha 関数 122
iscntrl 関数 123
isdigit 関数 123
isgraph 関数 124
islower 関数 124
isprint 関数 125
ispunct 関数 125
isspace 関数 126
istream クラス 283
istream マニピュレータ 292
istream::sentry クラス 282
istream::_ec2p_getistr 286
istream::sentry::operator bool 282
istream::operator >> 286
istream::gcount 287
istream::get 287
istream::getline 289
istream::ignore 289
istream::peek 289

istream::read ... 289
istream::readsome ... 290
istream::putback ... 290
istream::unget ... 290
istream::sync ... 290
istream::tellg ... 290
istream::seekg ... 291
isupper 関数 ... 126
isxdigit 関数 ... 127

J

jmp_buf ... 165

L

labs 関数 ... 232
lang（コンパイラオプション） ... 32
latin1（コンパイラオプション） ... 35
LDBL_DIG ... 130
LDBL_EXP_DIG ... 130
LDBL_MANT_DIG ... 130
LDBL_MAX ... 129
LDBL_MAX_EXP ... 129
LDBL_MAX_10_EXP ... 129
LDBL_MIN ... 129
LDBL_MIN_EXP ... 129
LDBL_MIN_10_EXP ... 130
LDBL_NEG_EPS ... 130
LDBL_NEG_EPS_EXP ... 130
LDBL_POS_EPS ... 130
LDBL_POS_EPS_EXP ... 130
ldexp 関数 ... 143
ldexpf 関数 ... 159
ldiv 関数 ... 232
ldiv_t ... 218
left（ios クラスマニピュレータ） ... 270
length（コンパイラサブオプション） ... 41
library（リンケージサブコマンド） ... 393
limits.h ... 131
list（コンパイラオプション） ... 41
log 関数 ... 143,317,329

logf 関数 159
log10 関数 144,317,329
log10f 関数 160
longjmp 関数 167
LONG_MAX 131
LONG_MIN 131
long double 型 58
long 型 58
loop（コンパイラサブオプション） 43
lseek ルーチン 380,465
L_tmpnam 172

M

mac（組み込み関数） 97
machinecode（コンパイラサブオプション） 36
machine.h 91
macsave（コンパイラオプション） 39
macl（組み込み関数） 97
malloc 関数 227
math.h 133
mathf.h 149
memchr 関数 242
memcmp 関数 239
memcpy 関数 235
memmove 関数（C ライブラリ関数） 253
memset 関数 251
message（コンパイラオプション） 36
mlist（リンケージサブコマンド） 388
modf 関数 144
modff 関数 160
movfpe（組み込み関数） 99
movtpe（組み込み関数） 99
mystrbuf クラス 303

N

new 305
new_handler 305
_ec2p_new_handler 305
noboolalpha（ios クラスマニピュレータ） 268
nop（組み込み関数） 100

norm 316,328
noshowbase（ios クラスマニピュレータ） 268
noshowpoint（ios クラスマニピュレータ） 268
noshowpos 269
noskipws（ios クラスマニピュレータ） 269
nouppercase（ios クラスマニピュレータ） 269
no_float.h 217
NULL 118

O

object（コンパイラオプション） 38
object（コンパイラサブオプション） 41
oct（ios クラスマニピュレータ） 270
off_type 256
onexit_t 218
onexit 関数（終了処理関数） 459
open ルーチン（低水準インタフェースルーチン） 376,463
operator> 354
operator< 353
operator>= 354
operator<= 354
operator>> 293,315,327,354
operator<< 300,315,327,355
operator+ 313,325,353
operator- 314,326
operator* 314,326
operator/ 314,326
operator== 314,326,353
operator!= 315,327,353
operator new 305
operator new[] 306
operator delete 306
operator delete[] 306
optimize（コンパイラオプション） 43
optimize（モジュール間最適化オプション／サブコマンド） 385
optlnk38（モジュール間最適化コマンド） 383
or_ccr（組み込み関数） 94
or_exr（組み込み関数） 97
ostream クラス 295

ostream::sentry クラス.....294
ostream::sentry::operator bool.....294
ostream::operator <<.....297
ostream::put.....297
ostream::write.....297
ostream::flush.....298
ostream::tellp.....298
ostream::seekp.....298
outcode（コンパイラオプション）.....37
ovfaddc（組み込み関数）.....101
ovfaddl（組み込み関数）.....101
ovfaddw（組み込み関数）.....101
ovfnegc（組み込み関数）.....103
ovfnegl（組み込み関数）.....103
ovfnegw（組み込み関数）.....103
ovfshalc（組み込み関数）.....102
ovfshall（組み込み関数）.....102
ovfshalw（組み込み関数）.....102
ovfsubc（組み込み関数）.....102
ovfsubl（組み込み関数）.....102
ovfsubw（組み込み関数）.....102
output（リンケージサブコマンド）.....394

P

P（セクション名）.....55
pack（コンパイラオプション）.....40
path.....52,404
perror 関数.....216
polar.....316,328
pos_type.....256
pow 関数.....145,317,329
powf 関数.....161
print（リンケージサブコマンド）.....399
printf 関数.....190
preinclude（コンパイラオプション）.....33
program（コンパイラサブオプション）.....42
preprocessor（コンパイラオプション）.....38
ptrdiff_t.....118
putchar 関数.....205

putc 関数 204
puts 関数 206

Q

qsort 関数 230

R

rand 関数 224
RAND_MAX 218
read ルーチン（低水準インタフェースルーチン） 378,464
real 関数 315,327
realloc 関数 228
regexpansion（コンパイラオプション） 45
register（コンパイラサブオプション） 43
register（モジュール間最適化オプションパラメタ） 385
regparam（コンパイラオプション） 40
rename（リンケージサブコマンド） 402
resetiosflags（smanip クラスマニピュレータ） 301
rewind 関数 212
right（ios クラスマニピュレータ） 270
rom（リンケージサブコマンド） 395
ROM、RAM の割り付け 359
ROM 化 359
rotlc（組み込み関数） 100
rotll（組み込み関数） 100
rotlw（組み込み関数） 100
rotrc（組み込み関数） 101
rotrl（組み込み関数） 101
rotrw（組み込み関数） 101

S

safe（モジュール間最適化オプションパラメタ） 386
samecode_forbid（モジュール間最適化オプション／サブコマンド） 387
samesize（モジュール間最適化オプション／サブコマンド） 386
same_code（モジュール間最適化オプションパラメタ） 385
sbrk ルーチン（低水準インタフェースルーチン） 381,465
scanf 関数 191
SCHAR_MAX 131
SCHAR_MIN 131
scientific（ios クラスマニピュレータ） 271
sdebug（リンケージサブコマンド） 397

section（コンパイラオプション） ... 42
SEEK_CUR ... 172
SEEK_END ... 172
SEEK_SET ... 172
setbuf 関数 ... 180
setbase（smanip クラスマニピュレータ） ... 301
setfill（smanip クラスマニピュレータ） ... 301
setiosflags（smanip クラスマニピュレータ） ... 301
set_ ccr（組み込み関数） ... 93
set_exr（組み込み関数） ... 96
set_imask_ccr（組み込み関数） ... 93
set_imask_exr（組み込み関数） ... 95
set_new_handler ... 306
setjmp.h ... 165
setjmp 関数 ... 166
setprecision（smanip クラスマニピュレータ） ... 302
setvbuf 関数 ... 181
setw（smanip クラスマニピュレータ） ... 302
short 型 ... 58
show（コンパイラオプション） ... 41
show（モジュール間最適化オプション／サブコマンド） ... 389
showbase（ios クラスマニピュレータ） ... 268
showpoint（ios クラスマニピュレータ） ... 268
showpos（ios クラスマニピュレータ） ... 269
SHRT_MAX ... 131
SHRT_MIN ... 131
shift（コンパイラサブオプション） ... 43
signed ... 58
sin 関数 ... 138,318,330
sinf 関数 ... 154
sinh 関数 ... 140,318,330
sinhf 関数 ... 156
size_t ... 118
sjis（コンパイラオプション） ... 35
sjis（コンパイラサブオプション） ... 37
skipws（ios クラスマニピュレータ） ... 269
sleep（組み込み関数） ... 98
smanip クラス ... 301

source（コンパイラサブオプション） 41
speed（コンパイラオプション） 43
speed（モジュール間最適化オプションパラメタ） 386
sprintf 関数 192
sqrt 関数 145,318,330
sqrtf 関数 161
srand 関数 224
sscanf 関数 193
start（リンケージサブコマンド） 399
statistics（コンパイラサブオプション） 41
stdarg.h 168
stddef.h 118
stderr 115,172
stdin 115,172
stdout 115,172
stdio.h 172
stdlib.h 218
strcat 関数 237
strchr 関数 243
strcmp 関数 240
strcpy 関数 235
strcspn 関数 244
streambuf クラス 272
streamoff 256
streamsize 256
streambuf::pubsetbuf 274
streambuf::pubseekoff 275
streambuf::pubseekpos 275
streambuf::pubsync 275
streambuf::in_avail 275
streambuf::snextc 275
streambuf::sbumpc 276
streambuf::sgetc 276
streambuf::sgetn 276
streambuf::sputbackc 276
streambuf::sungetc 276
streambuf::sputc 276
streambuf::sputn 277

streambuf::eback 277
streambuf::gptr 277
streambuf::egptr 277
streambuf::gbump 277
streambuf::setg 278
streambuf::pbase 278
streambuf::pptr 288
streambuf::epptr 288
streambuf::pbump 288
streambuf::setp 279
streambuf::setbuf 279
streambuf::seekoff 279
streambuf::seekpos 279
streambuf::sync 279
streambuf::showmanyc 280
streambuf::xsgetn 280
streambuf::underflow 280
streambuf::uflow 280
streambuf::pbackfail 280
streambuf::xsputn 280
streambuf::overflow 281
strerror 関数 252
string.h 233
string（コンパイラオプション） 37
string クラス 331
string クラスマニピュレータ 351
string::iterator 331
string::const_iterator 331
string::operator= 338,339
string::begin 339
string::end 339
string::size 339
string::length 339
string::max_size 339
string::resize 340
string::capacity 340
string::reserve 340
string::clear 340

string::empty 341
string::operator[] 341
string::at 341
string::operator+= 341
string::append 342
string::assign 342,343
string::insert 343,344
string::erase 344
string::replace 345,346
string::copy 346
string::swap 346
string::c_str 346
string::data 346
string::find 346,347
string::rfind 347
string::find_first_of 347,348
string::find_last_of 348
string::find_first_not_of 348,349
string::find_last_not_of 349
string::substr 349
string::compare 349,350
string_unify（モジュール間最適化オプションパラメタ） 385
strlen 関数 253
strncat 関数 238
strncmp 関数 241
strncpy 関数 236
strpbrk 関数 245
strrchr 関数 246
strspn 関数 247
strstr 関数 248
strtod 関数 222
strtok 関数 249
strtol 関数 223
struct（コンパイラサブオプション） 43
subcommand（コンパイラオプション） 51
subcommand（モジュール間最適化オプション） 389
swap（string クラスマニピュレータ） 354
switch（コンパイラサブオプション） 43

sysrof（リンケージサブコマンド） 395
sysrofplus 395
SYS_OPEN 172
symbol_delete（モジュール間最適化オプションパラメタ） 385
symbol_forbid（モジュール間最適化オプション／サブコマンド） 386

T

table（コンパイラサブオプション） 46
tan 関数 139,318,330
tanf 関数 155
tanh 関数 140,318,330
tanhf 関数 156
tas（組み込み関数） 99
TMP_MAX 172
tolower 関数 127
toupper 関数 128
trapa（組み込み関数） 98

U

UCHAR_MAX 131
udf（リンケージサブコマンド） 401
udfcheck（リンケージサブコマンド） 401
UINT_MAX 131
ULONG_MAX 131
USHRT_MAX 131
ungetc 関数 207
unsigned 58
uppercase（ios クラスマニピュレータ） 269

V

variable_access（モジュール間最適化オプションパラメタ） 385
variable_forbid（モジュール間最適化オプション／サブコマンド） 387
va_arg マクロ 170
va_end マクロ 171
va_list 168
va_start マクロ 169
vfprintf 関数 194
volatile（コンパイラオプション） 44
vprintf 関数 195
vsprintf 関数 196

W

width（コンパイラサブオプション）...41
write ルーチン（低水準インタフェースルーチン）...379,465
ws（streambuf クラスマニピュレータ）...292

X

xor_ccr（組み込み関数）..95
xor_exr（組み込み関数）..97

H8S, H8/300 シリーズ C/C++ コンパイラ
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

ADJ-702-137D