お客様各位

カタログ等資料中の旧社名の扱いについて

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

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

2010年4月1日
ルネサスエレクトロニクス株式会社

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

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

ご注意書き

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

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

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

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

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

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

User's Manual

M32Rファミリ用
C/C++コンパイラパッケージ V.5.00
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
www.renesas.com

Rev.1.00　2005.07

- Microsoft、MS-DOS、Windows および Windows NT は、米国 Microsoft Corporation の米国およびその他の国における商標または登録商標です。
- HP-UX は、米国 Hewlett-Packard Company のオペレーティングシステムの名称です。
- Sun、Solaris、Java およびすべての Java 関連の商標およびロゴは、米国およびその他の国における米国 Sun Microsystems, Inc.の商標または登録商標です。
- UNIX は、The Open Group の米国ならびにその他の国における登録商標です。
- Linux は、Linus Torvalds 氏の米国およびその他の国における登録商標あるいは商標です。
- Turbolinux の名称およびロゴは、Turbolinux, Inc.の登録商標です。
- IBM および AT は、米国 International Business Machines Corporation の登録商標です。
- HP 9000 は、米国 Hewlett-Packard Company の商品名称です。
- SPARC および SPARCstation は、米国 SPARC International, Inc.の登録商標です。
- Intel, Pentium は、米国 Intel Corporation の登録商標です。
- Adobe および Acrobat は、Adobe Systems Incorporated（アドビシステムズ社）の登録商標です。
- Netscape および Netscape Navigator は、米国およびその他の諸国の Netscape Communications Corporation 社の登録商標です。

その他すべてのブランド名および製品名は個々の所有者の登録商標もしくは商標です。

安全設計に関するお願い

- 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

- 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株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズ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。
- 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、株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズは責任を負いません。
- 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、株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズは、予告なしに、本資料に記載した製品又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、事前に株式会社ルネサス テクノロジ、株式会社ルネサス ソリューションズ、株式会社ルネサス販売又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。
- 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、株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズはその責任を負いません。
- 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。株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズは、適用可否に対する責任は負いません。
- 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、株式会社ルネサス テクノロジ、株式会社ルネサス ソリューションズ、株式会社ルネサス販売又は特約店へご照会ください。
- 本資料の転載、複製については、文書による株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズの事前の承諾が必要です。
- 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株式会社ルネサス テクノロジ、株式会社ルネサス ソリューションズ、株式会社ルネサス販売又は特約店までご照会ください。

製品の内容及び本書についてのお問い合わせ先

インストーラが生成する以下のテキストファイルに必要事項を記入の上、ツール技術サポート窓口support_tool@renesas.comまで送信ください。

¥SUPPORT¥製品名¥SUPPORT.TXT

株式会社ルネサス ソリューションズ

ツール技術サポート窓口	support_tool@renesas.com
ユーザ登録窓口	regist_tool@renesas.com
ホームページ	http://www.renesas.com/jp/tools

目次

付録 A　拡張機能リファレンス　1

付録 B C 標準ライブラリ関数一覧 1

付録 C　制限事項　1

付録 D　コンパイル　注意事項　1

はじめに

M3T-CC32R（以下 CC32R と略す）は、ルネサス 32 ビット RISC マイクロコンピュータ M32R ファミリ用のソフトウェア開発支援ツール一式です。CC32R は組み込み制御システムの開発に適した豊富な機能を提供します。CC32R のマニュアルセットでは、CC32R を使って M32R システムをターゲットとしたプログラミングを行うための情報を提供します。

対象読者

CC32R マニュアルセットは、M32R システム上で動作するプログラムを C/C++ またはアセンブリ言語で開発するプログラマーを対象としています。したがって、本マニュアルの読者は、プログラミング言語（C またはアセンブリ言語）、ターゲットとなる M32R ファミリ、開発環境（使用するホストマシンや OS 等）についての基礎知識があることを前提としています。

関連マニュアル

M32R ファミリ用の開発に関連するマニュアルを以下に示します。

■ M32R ファミリ ユーザーズマニュアル
■ M32R ファミリソフトウェアマニュアル

詳細は、「ルネサスマイコン」のサイトを参照してください。
URL は、

http://www.renesas.com/jpn/

参考文献

ANSI-C/ISO C++ 言語仕様の詳細については以下の文献を参照してください。

■ANSI/ISO 9899-1990 American National Standard for Programming Languages - C

■ISO/IEC 14882:2003 INTERNATIONAL STANDARD Programming Languages - C++
(American National Standards Institute, Inc. 発行)

本マニュアルは、以下の URL の内容を参考にしました。

■日本女子大学　Japan Women's University (http://momi.jwu.ac.jp/ccenter/cpp/cpp2.htm)

マニュアルの表記規則

CC32R マニュアルセットでは、とくに説明がない限り、以下のような記述規則に従っています。

■ 記号

表記	意味
斜体文字	実際に指定する場合、斜体文字部分を適する値や文字に置き換える。
a |*b*	| は複数の項目を区切り、どれか一つを選択することを示す。*a* |*b* ならば、*a*か*b*のいずれかを選択する。
[ ]	[ ] 内の項目は省略可能。
...	... の前に記述された項目が複数指定可能。
	記述省略。
<RET>	リターンキー入力。

■ 用語（1/2）

表記	意味
M32R M32Rx	ルネサス32ビットRISCマイクロコンピュータ。
M32Rシステム	M32Rファミリを搭載したシステム。
CC32R	CC32Rクロスツールキット。
C/C++コンパイラ（cc32R）	CC32Rに含まれるC/C++コンパイラ。
アセンブラ（as32R）	CC32Rに含まれるアセンブラ。
リンカ（lnk32R）	CC32Rに含まれるリンカ。
ライブラリアン（lib32R）	CC32Rに含まれるライブラリアン。
マップジェネレータ（map32R）	CC32Rに含まれるマップジェネレータ。
ロードモジュールコンバータ（lmc32R）	CC32Rに含まれるロードモジュールコンバータ。
リリースノート	CC32Rの製品パッケージに含まれているリリースに関する資料（はじめにお読みください）。
Cプログラム	C言語で記述されたプログラム。
C++プログラム	C++言語で記述されたプログラム。
アセンブリプログラム	アセンブリ言語で記述されたプログラム。
ソースファイル	C/C++やアセンブリ言語などのプログラミング言語のソースコードが書かれたテキストファイル。
オブジェクトモジュール	C/C++およびアセンブリ言語のソースコードを、M32Rに対応したマシンコードに翻訳した結果のオブジェクトファイル。Cコンパイラおよびアセンブラによって生成される。
ロードモジュール	リンク済みのオブジェクトモジュールで、M32Rシステム上で実行可能な形式のファイル。リンカやロードモジュールコンバータによって生成される。

■ 用語（2/2）

表記	意味
リンクマップ	オブジェクトモジュールやロードモジュールの、セクション配置情報と外部定義シンボル情報をリスト出力したもの。マップジェネレータによって生成される。
ライブラリ	M32Rに対応したライブラリファイル。ライブラリアンによって生成される。
C標準ライブラリ	CC32Rに含まれる、ANSI-C準拠のライブラリファイル。
ユーザーライブラリ	ユーザーがライブラリアンによって作成したライブラリ。
リターン値	呼び出された関数が処理結果として呼び出し側に返す値。関数値。
ローカル変数	関数内だけで有効な変数。局所変数。
Rx	任意のM32R汎用レジスタ。
CRx	任意のM32R制御レジスタ。
ＥＷＳ	エンジニアリングワークステーションの略称。
ＯＳ	オペレーティングシステムの略称。
デフォルト	ユーザーが値を指定しなかった場合に有効となる規定値。または、ユーザーが動作指定しなかった場合の処理。
空白文字	タブやスペースキーによって入力される空白文字。

第１章
CC32Rの概要

1.1 CC32Rの特長

CC32Rは、M32Rファミリ用のアプリケーションプログラムを開発するためのクロスツール一式です。とくに、組み込み制御システムの開発に適した豊富な機能を提供します。CC32Rのおもな特長を以下に示します。

■ C/C++言語およびアセンブリ言語で記述したソースファイルから、M32R上で実行可能なロードモジュールファイルを生成します。

■ ロードモジュールをROM化可能なモトローラSフォーマットに変換できます。

■ 実行速度が速く、コード効率の良いオブジェクトコードを生成するための最適化機能があります。

1.2 CC32Rの構成

CC32Rは次の各種クロスツールによって構成されています。

■ Cコンパイラ[コンパイルドライバ](cc32R)
■ アセンブラ（as32R）
■ プレリンカ（plink32R）
■ リンカ（lnk32R）
■ ライブラリアン（lib32R）
■ マップジェネレータ（map32R）
■ ロードモジュールコンバータ（lmc32R）
■ デバッグ情報除去ユーティリティ（strip32R）

1.3 各クロスツールの概要

各クロスツールの概要を以下に示します。

■ C/C++ コンパイラ

● コンパイルドライバ（cc32R）

ANSI/ISO 9899-1990 および ISO/IEC 14882:2003 に準拠した C/C++ コンパイラ（コンパイルドライバ）です。C/C++ 言語ソースファイルをコンパイルして、アセンブリ言語ソースファイルを生成します。また、アセンブラやリンカを起動も行います。

cc32R の起動によって、ソースファイルからロードモジュールファイルを生成するまでの一連の処理が実行できます。

● プレリンカ（plink32R）

プレリンカは、テンプレート関数やインライン化できないインライン関数の定義を、すべてのオブジェクトに一つだけ定義を生成するために使用するユーティリティです。

C/C++ コンパイラ cc32R に組込まれています。必要に応じて cc32R の内部で実行されます。

例)

a.cpp	b.cpp

template <class T>
void func(T a) { T b = a }
void afunc(void)
{
int a;
func(a);
}
	template <class T>
	void func(T a) { T b = a }
	void bfunc(void)
	{
	int a;
	func(a);
	}

上記の a.cpp と b.cpp に同じテンプレート関数が定義されていて、そのテンプレート関数の呼び出しが双方でされている場合、func テンプレート関数の定義をすべてのオブジェクト中に一つだけ生成するようにすることができます。

■ アセンブラ（as32R）

アセンブリ言語ソースファイルをアセンブルし、オブジェクトモジュールを生成します。アセンブリ言語ソースファイルには擬似命令およびマクロ命令が記述できます。また、オプション指定によりアセンブルリストを出力することができます。

■ リンカ（lnk32R）

オブジェクトモジュール、リロケータブルロードモジュール、および、ライブラリを結合（リンク）して、ロードモジュールファイルを生成します。ロードモジュールの形式は、リロケータブル形式またはアブソリュート形式のどちらかを選択できます。リロケータブル形式のロードモジュールは、リンカに再入力できます。

リンク時に、リンカ lnk32R を直接起動しないでください。必ず、コンパイルドライバ cc32R でリンクを行ってください。(例： cc32R *.mo)

（C 言語、アセンブリ言語で記述されたオブジェクトのみをリンクする場合は、lnk32R を直接起動してリンクしても問題ありません。）

■ ライブラリアン（lib32R）

オブジェクトモジュールまたはリロケータブルロードモジュールを読み込んで、ライブラリファイルを生成します。

ライブラリファイル作成時に、ライブラリアン lib32R を直接起動しないでください。ライブラリファイルを作成する場合にも、コンパイルドライバ”cc32R”を通して(-mklib オプションを使用)ライブラリアンを起動してください。（C 言語、アセンブリ言語で記述されたオブジェクトのみでライブラリファイルを作成する場合は、lib32R を直接起動も問題ありません。）

■ マップジェネレータ（map32R）

オブジェクトモジュール、リロケータブルロードモジュール、および、アブソリュートロードモジュールを読み込んで、リンクマップ（マップリストおよび外部定義シンボルリスト）を出力します。

■ ロードモジュールコンバータ（lmc32R）

リンカが生成したロードモジュールをモトローラＳフォーマットに変換します。プログラムを ROM 化するために必要なツールです。

■ デバッグ情報除去ユーティリティ（strip32R）

デバッグ情報除去ユーティリティ（strip32R）を提供しています。
strip32R は、リンカの出力するロードモジュールファイルおよび、コンパイラやアセンブラが出力するオブジェクトモジュールファイルからデバッグ情報を削除します。
strip32R は、本コンパイラの正規製品ではありません。取り扱いについては、「開発支援ユーティリティの取り扱い説明」のファイルをご覧ください。使用方法は、「デバッグ情報除去ユーティリティの説明」（strip32R_j.txt）のファイルをご覧ください。

1.4 CC32Rによる処理フロー

1.4.1 CC32Rによるアプリケーション開発フロー全体

CC32R を使った開発フローを、図 1.1 に示します。

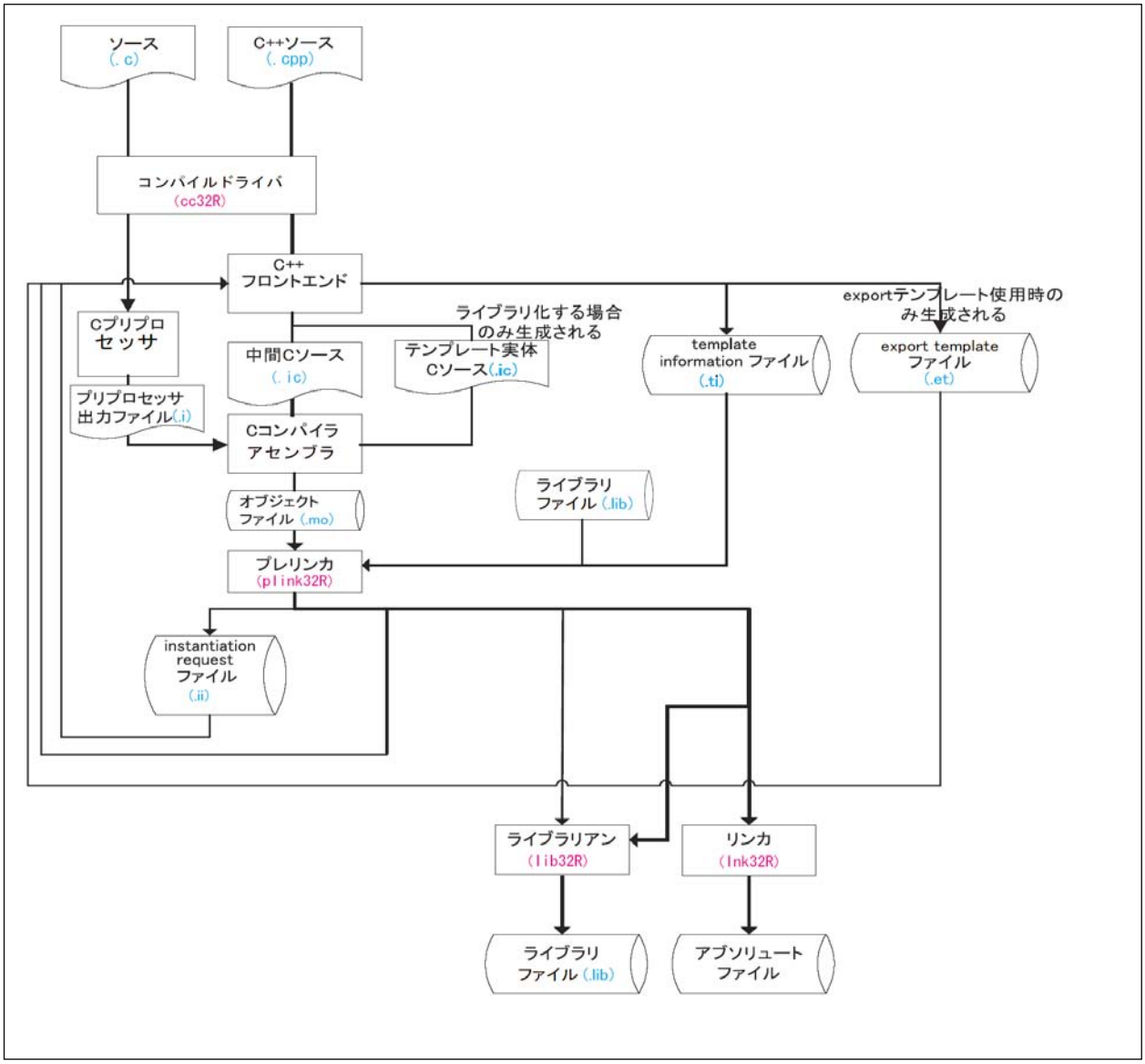


図 1.1　CC32R によるアプリケーション開発フロー

1.4.2　オブジェクトの生成フロー

オブジェクトファイル生成のフローと生成されるファイルについて、図1.2に示します。

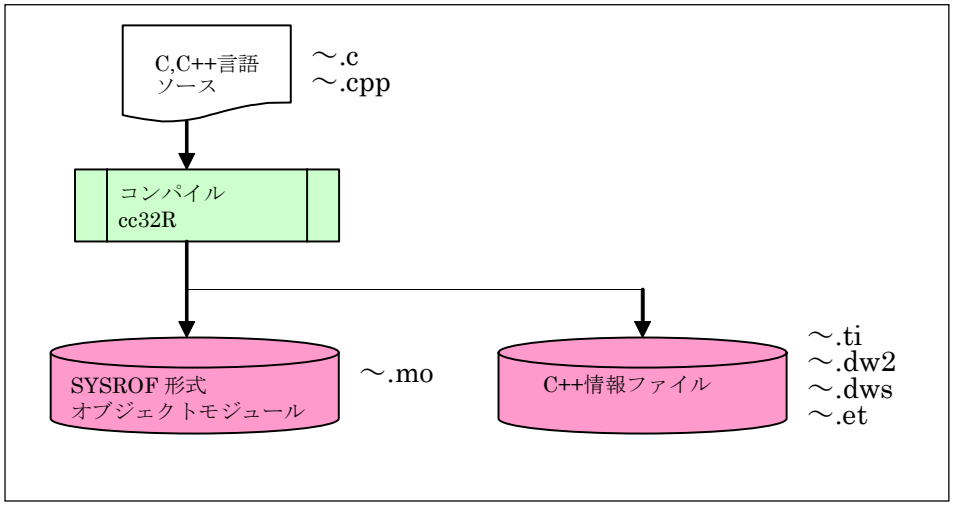


図1.2　CC32Rによるオブジェクトの生成フロー

1.4.3 リンクの仕方

オブジェクトファイルとライブラリファイルのリンクとロードモジュールファイルの生成ついて、図1.3に示します。

オブジェクトファイルのリンクだけを行う場合であっても、コンパイルドライバcc32Rを通してリンカを起動します。これはユーザがリンカを直接呼び出すと、プレリンカが起動されないため正しくリンクできるオブジェクトが作成されない可能性があるからです。

Makefile 中であっても、リンカだけを呼び出す記述はしないでください。

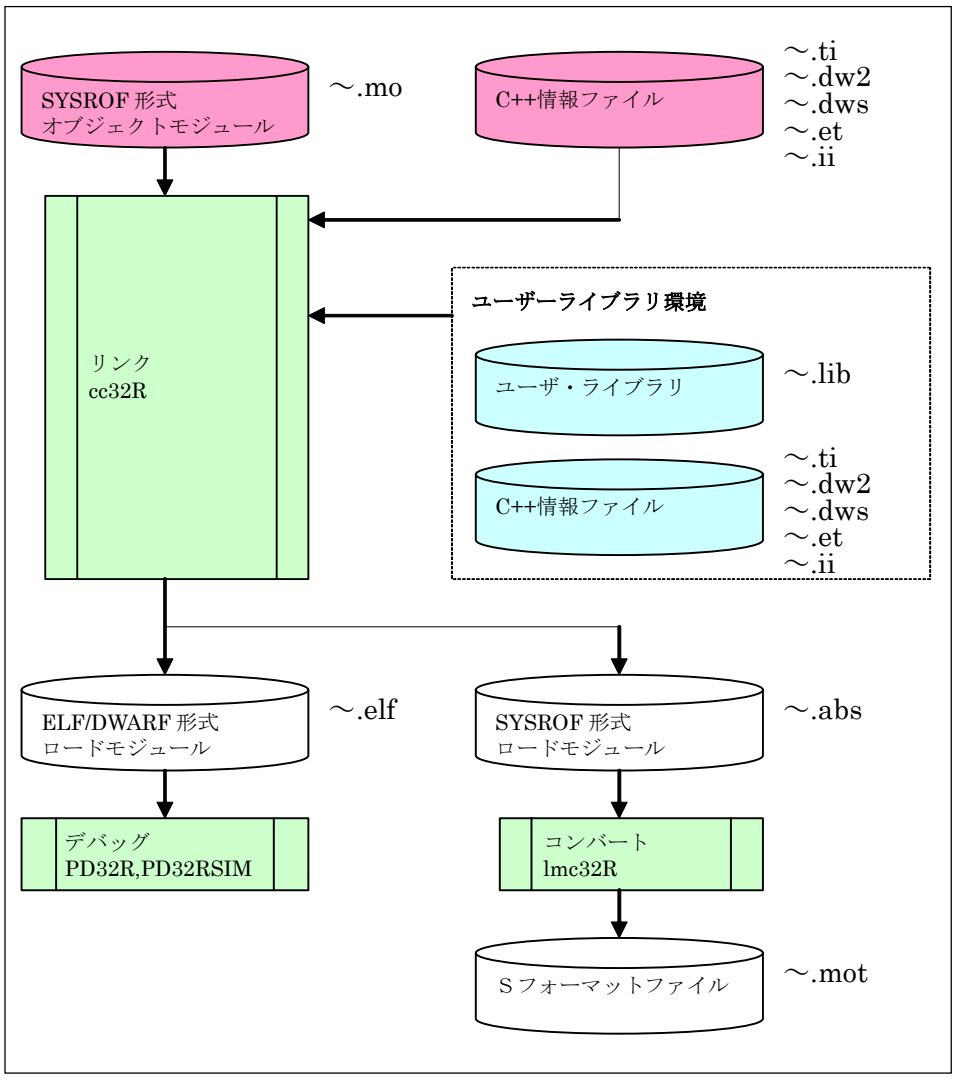


図1.3 リンクのフロー

1.4.4 ライブラリファイルの生成フロー

オブジェクトファイルとライブラリファイルのリンクとロードモジュールファイルの生成ついて、図1.4に示します。

ライブラリファイルを作成する場合、プレリンカを起動する必要があります。
このためライブラリファイルを作成する場合にも、コンパイルドライバ”cc32R”を通して(-mklibオプションを使用)ライブラリアンを起動してください。
Makefile 中であっても、ライブラリアンだけを呼び出す記述はしないでください。

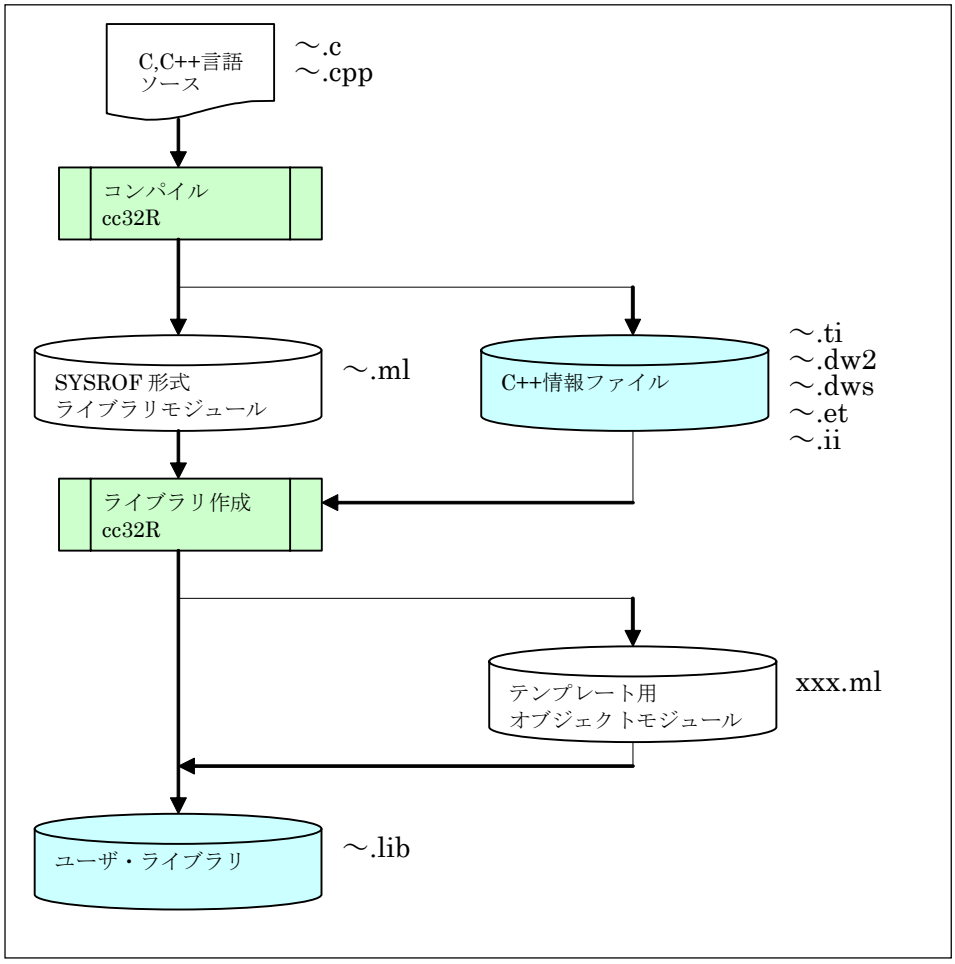


図1.4 リンクのフロー

1.5 CC32Rの取り扱う入出力ファイル名

CC32R は、ファイル名の拡張子によって入力ファイルの種類を判別したり、出力ファイルに自動的にファイル名またはファイル名の拡張子を付けたりする場合があります。表1.1 に、CC32R が取り扱う入出力ファイル名を示します。*file* は任意のファイル名です。

表 1.1　CC32R の取り扱うファイル

ファイル	取り扱い
file.c	C言語ソースファイル
file.cpp	C++言語ソースファイル
file.mi	プリプロセッサ出力ファイル（Cプリプロセッサによる展開後のC言語ソースファイル）
file.ms	アセンブリ言語ソースファイル
file.mo	オブジェクトモジュールファイル （C++の場合はロードモジュール作成専用）
file.ml	オブジェクトモジュールファイル （C++の場合はライブラリ作成専用）
file.lib	ライブラリファイル
file.abs am.out（デフォルトのファイル名）	ロードモジュールファイル
file.mot	ロードモジュールファイル（モトローラSフォーマット）
file.ic *file*.ti *file*.et *file*.ii *file*.dw2 *file*.dw3	C++言語の各機能を実現するための情報を格納したファイル群 （ロードモジュールファイル（flie.abs または am.out）作成に必要）

第２章
C/C++コンパイラcc32Rの概要

2.1 cc32Rの概要

2.1.1 機能

cc32R は CC32R クロスツールキットに含まれている C/C++ コンパイラで、以下のような機能があります。

■ C/C++ 言語ソースファイルをコンパイルし、アセンブリ言語ソースファイルを生成する（-S オプション指定時）。

■ C/C++ およびアセンブリ言語ソースファイルから、オブジェクトモジュールファイルを生成する（デフォルトおよび -c オプション指定時）。

■ C/C++ およびアセンブリ言語ソースファイルから、ロードモジュールファイルを生成する（デフォルトの動作。cc32R はデフォルトで、コンパイル→アセンブル→リンクまでの一連の処理を実行します。ソースファイルの内容（使用言語）はファイル名の拡張子で判別されます）。

2.1.2 特長

■ ANSI 仕様に準拠

C/C++ コンパイラおよび C/C++ 標準ライブラリは、ANSI/ISO 9899-1990 および ISO/IEC 14882:2003 に準拠しています。

■ 64bit 整数演算関数群をサポート

C/C++ 言語の整数演算を、64bit のダイナミックレンジで行う関数群を標準ライブラリに追加しました。C/C++ 言語の整数型と同様に、加減乗除、ビット演算、シフト、比較などの演算を 64bit の範囲で行うことが可能です。

■ 浮動小数点演算機能をサポート

浮動小数点の内部演算形式は、IEEE（The Institute of Electrical and Electronics Engineers）-754 規格に準拠しています。

■ ROM 化機能をサポート

ROM 化可能なオブジェクトモジュールの生成ができます（リンカに依存する機能です）。オブジェクトモジュールの各セクションをリンクして配置する際、初期値データをもつセクションについては、セクション領域を RAM 領域に配置し、初期値データは ROM 領域に配置することができます。

■ 最適化機能をサポート

最適化により効率の良いオブジェクトコードを生成します。最適化のレベルには以下のものがあります。

・ アセンブリ言語レベルでの最適化
アセンブリ言語レベルにおいて不要なコードの削除、最適なコードへの置換、命令のスケジューリングなどを実施します。

・ ローカルな最適化
C/C++ 言語構文の局所的な分析により、定数およびコピーの伝播、不要コードの削除、共通部分式の削除などを実施します。

・ グローバルな最適化
C/C++ 言語関数内の大域的な分析により、生存変数の解析によるコードの置き換え、制御の流れの最適化などを実施します。

また、具体的には以下の項目について最適化を行います。

・コントロールフローの最適化
・共通部分式の削除
・定数およびコピーの伝搬
・生きている変数の解析
・デッドコードの削除
・レジスタ割り付けの最適化

各レベルの最適化は単独でも実施できますが、組み合わせて実施することにより、より効率のよいコードを生成することができます。

また、各レベルの最適化のほとんどは、プログラムの実行を高速にする効果とコードサイズを小さくする効果の両方の効果がありますが、一部の最適化には、その一方だけの効果をもつもの、あるいは、一方を犠牲にして他方の効果を良くするというものもあります。このような最適化機能は、最適化の優先度を指定することによって選択できます。

■ 出力ファイルが選択可能

コンパイル結果の出力は、アセンブリ言語ソースファイル、オブジェクトモジュールファイル、ロードモジュールファイル（リンク済みオブジェクトモジュール）のうちから選べます。アセンブリ言語ソースファイルは、C/C++ 言語ソースプログラムをアセンブリ言語レベルで確認したい場合に有効です。

■ C/C++ 言語ソース行デバッグ情報が出力可能

C/C++ 言語ソースファイルのソース行デバッグ情報を、リンカが生成するロードモジュールに付加しています。このロードモジュールを使用すれば、デバッガによる C/C++ 言語ソース行レベルでのデバッグが可能です。

■ 開発支援ユーティリティの提供

本コンパイラと共に使用すると便利なユーティリティを提供しています。

○アブソリュートリスティングユーティリティ（abslist）

アブソリュートリスティングユーティリティ（abslist）を提供しています。
abslist は、アセンブラ（as32R）が出力したアセンブルリストファイル中の LOCATION 値を、リンク後の実アドレスに変換したアブソリュートリストファイルを生成します。 abslist は、本コンパイラの正規製品ではありません。取り扱いについては、「開発支援ユーティリティの取り扱い説明」のファイルをご覧ください。使用方法は、「アブソリュートリスティングユーティリティの説明」（abslist_j.txt）のファイルをご覧ください。

○スタックサイズ算出ユーティリティ（stk32R）

スタックサイズ算出ユーティリティ（stk32R）を提供しています。
stk32R は、コンパイラが出力したスタック使用量表示ファイル（-stack オプション付き）を処理してプログラムの動作に必要なスタックサイズを求めます。
また、C 標準ライブラリ（m32Rc.lib、m32RcR.lib、etc...）に登録されている関数のスタック使用量表示ファイル（m32Rc.stk、m32RcR.stk、etc...）も合わせて提供しています。stk32R にライブラリ関数を呼び出しているプログラムのスタック使用量表示ファイルを入力する場合、-I オプションで、使用しているライブラリファイルにあったスタック使用量表示ファイルを指定してください。なお、pow 関数、setlocale 関数は、再帰呼び出し関数ですので、正しい値は求められません。ファイル中の値は参考値です。
stk32R は、本コンパイラの正規製品ではありません。取り扱いについては、「開発支援ユーティリティの取り扱い説明」のファイルをご覧ください。使用方法は、「スタックサイズ算出ユーティリティの説明」（stk32R_j.txt）のファイルをご覧ください。

2.2 前バージョンとの互換性

■ CC32R V.2.10 Release 1 以前のオブジェクトをリンクする場合（リンカ lnk32R）

V.2.10 Release 1 以前の CC32R で生成したオブジェクトファイル（ライブラリファイル含む）を V.3.00 Release 1 以降（本バージョン含む）のリンカに入力すると、以下のようなワーニングを表示します。
lnk32R: "ファイル名": warning: old interface module: "revision: 01" この場合は、該当するオブジェクトファイルを最新の CC32R で生成し直してください。

■ CC32R V.1.00 Release 3 以前のオブジェクトをリンクする場合（リンカ lnk32R）

V.1.00 Release 3 以前の CC32R で生成したオブジェクトファイル（ライブラリファイル含む）を、V.1.00 Release 4 以降（本バージョン含む）のリンカに入力すると、”relocation out of range” のエラーが発生する場合があります。この場合は、該当するオブジェクトファイルを最新の CC32R で生成し直してください。

■ -MAP オプションのエラー処理の緩和

cc32R のリンクマップファイル出力オプション、-M オプション（または、lnk32R の -M オプション）を指定していた場合、リンク処理中にエラーが発生してもリンクマップを出力するようにしました。
エラー発生時のリンクマップには不完全な情報を含まれていますが、モジュールごとのセクション配置やシンボルアドレスなどがわかりますので、リンクエラーの原因の確認に利用することができます。

第３章
C/C++コンパイラの起動方法

3.1 C/C++コンパイラを起動するには

3.1.1 C/C++コンパイラの起動手順

C/C++ コンパイラを起動するためには、環境変数を設定した後、規則に従ってコマンド「cc32R」を入力し、それを実行します（3.1.2、3.1.3 参照)。

3.1.2 環境変数の設定

環境変数 M32RBIN、M32RINC、M32RLIB、M32RTMP に正しいディレクトリを設定します(通常はインストール時に設定します)。設定方法は「CC32R インストレーションガイド」を参照してください。環境変数の設定を省略すると、デフォルトの値を設定したことになります。

表 3.1 環境変数のデフォルト

環境変数	デフォルトの値			
M32RKIN	MS-Windows版の場合	”sjis”	EWS版の場合	”euc
M32RKOUT	MS-Windows版の場合	”sjis”	EWS版の場合	”euc

3.1.3 コマンド行の記述規則

C/C++ コンパイラ起動コマンド「cc32R」の、コマンド行の入力書式および規則を以下に示します。起動オプションの詳細は、次節 3.2 を参照してください。

```
cc32R   [-access=access_control_file] [-c] [-C] [-cs]
        [-CS] [-constr] [-D name[=def]] [-e entrypoint]
        [-ecpp] [-E] [-exception] [-noexception]
        [-float_only] [-fminst] [-g] [-I path] [-L dir]
        [-l lib] [-lang={cpp|c] [-M] [-MAP map_filename]
        [-MEM addr1,addr2] [-mklib] [-mklib={c|a}]
        [-noinline] [-o output_filename] [-Opriority]
        [-O[level]] [-m32re5] [-P] [-r] [-R old=new]
        [-rtti={on|off}] [-S] [-strict_standard]
        [-SEC name[=addr][,name[=addr]...]]
        [-switch_by_offset] [-U name] [-v] [-V] [-w]
        [-warn_suppressed_nested_comment] [-rel16]
        [-XX[=symbol_num]]
        [-small [-memlarge]] [-medium] [-large]
        [-stack] [-zdiv][-@] [-elf_endian={big|little}]
        [input_filenames] <RET>

※[ ]で囲まれていない項目    ：必ず入力しなければならない項目
※[ ]で囲まれた項目          ：必要に応じて入力する項目
※ －の付いた記号            ：起動オプション（詳細は3.2節参照）
※ <RET>                     ：リターンキー入力
```

図 3.1 cc32R コマンドの入力書式

■ コマンド行は、上記（図 3.1）の書式に従って記述します。各項目（コマンド名、オプション、入力ファイル名）の間は、1 文字以上の空白文字で区切ります。最後にリターンキーを入力すると、C/C++ コンパイラが実行を開始します。

■ オプションとそのパラメータとの間には任意で空白文字が入力できます。

■ *input_filenames* には、1 つまたは複数の入力ファイル名を指定します。ファイル名とファイル名の間は 1 文字以上の空白で区切ります。ファイル数の制限はありません。

■ 入力ファイルは、ファイル名の拡張子によって以下のように判断されます。

表3.2 C/C++コンパイラの取り扱う入力ファイル名

拡張子	C/C++コンパイラにおける取り扱い
.c	C言語ソースファイル
.cpp	C++言語ソースファイル
.ms	アセンブリ言語ソースファイル
.mo	オブジェクトモジュールファイル （C++の場合はロードモジュール作成専用）
.ml	オブジェクトモジュールファイル （C++の場合はライブラリ作成専用）
上記以外	オブジェクトモジュールファイル

3.1.4 コマンドファイル

本コンパイラは、複数のコマンドオプションを記述したファイル（コマンドファイル）を読み込んで起動することができます。

```
cc32R @file_name [-@]

※ file_name ：コマンドファイル名
※[ ]で囲まれた項目          ：必要に応じて入力する項目
※<RET>                      ：リターンキー入力
```

図 3.2 cc32R コマンドファイル入力書式

■ パラメータにはコマンドファイル名の前に @ を付けたもののみを指定します。パラメータとして、一つの「@ コマンドファイル名」以外が指定されている場合（-@ オプション指定を除く）、@ から始まる名前のパラメータであってもコマンドファイルとは見なしません。

（例 1） @sample.cmd がコマンドファイル指定と解釈される場合

```
>cc32R @sample.cmd
>cc32R -@ @sample.cmd
>cc32R @sample.cmd -@
```

（例 2） @sample.cmd がコマンドファイル指定と解釈され ない場合

```
>cc32R -v @sample.cmd              （コマンドファイル指定以外の引数がある）
>cc32R @sample.cmd -v              （コマンドファイル指定以外の引数がある）
>cc32R @sample.cmd @sample.cmd（コマンドファイルを2回以上指定）
```

■ コマンドファイルの記述方法は以下のようになります。

- ○ 各パラメータ（オプション指定、入出力ファイル名など）の記述形式はコマンド行でパラメータ指定を行う場合の記述形式に従います。
- ○ パラメータ間は一つ以上の空白文字または改行（リターンキー入力）で区切ります。
- ○ 一つのパラメータの文字列の途中で改行して、複数行に渡って記述することはできません。
- ○ 先頭に〝@〟を記述した行は、コメントとみなし無視されます。
- ○ コマンドファイル内からは、コマンドファイルを呼べません。

コマンドファイルを使用した場合と、通常のコマンドラインの書式を使用した場合を図 3.3 に示します。

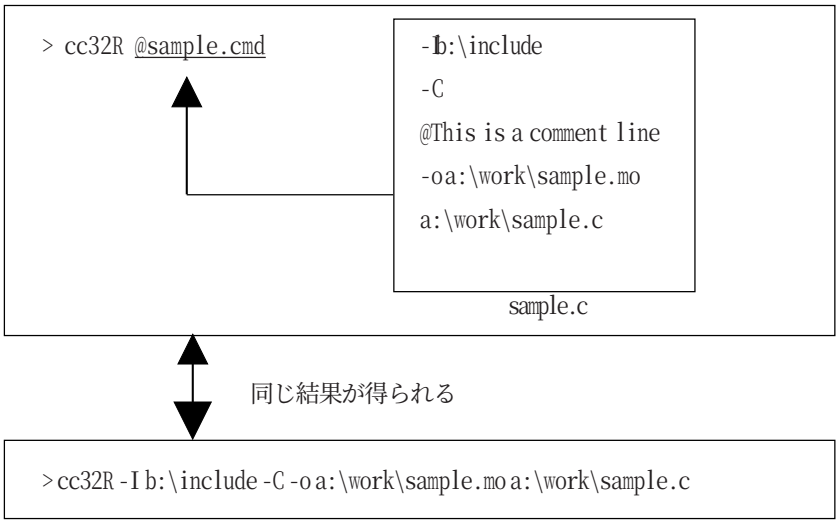


図 3.3　コマンドファイルを使用した場合とコマンドラインで記述した場合

||||| 注意 |||||

lnk32R（リンカ）と lib32R（ライブラリアン）では、コマンドファイル名の前に @ を付けなくても、コマンドファイルとして処理します。ただし、その場合、コマンドファイル中の @ はコメントとして処理しません。

||||| 注意 |||||

@から始まるファイル名は、コマンドファイル以外に使用しないでください。コマンド行が正しく処理されません（動作が保証されません）。

3.1.5 入力ファイルの条件

C/C++ コンパイラで処理できる入力ファイルの条件を以下に示します。これらの条件を満たさないファイルは入力しないでください。

表 3.3 入力ファイルの条件

条件項目	条件内容
入力可能なファイル	ソースファイル（C言語およびアセンブリ言語） オブジェクトモジュールファイル ロードモジュールファイル ライブラリ
使用できる名前の長さ	識別子（関数名・変数名等）　：240文字まで ※　注意 C/C++コンパイラは、1文字目から240文字目ま　でが同じで241文字目以降が異なる2つの名前を、同一の識別子として扱います。
使用できる名前の数	セクション名　：1ファイルにつき65535個まで シンボル名　：1ファイルにつき65535個まで モジュール名　：1ファイルにつき65535個まで ※ 注意 開発環境のシステムのメモリ容量によっては制限を受けます。

3.1.6 入力ファイルの自動判別機能

C/C++ コンパイラは、自動的に入力ファイルの内容をファイル名の拡張子から判断し、そのファイルごとに必要な処理工程から処理を開始します。例えば、C 言語ソースファイルに対してはコンパイル処理から、オブジェクトモジュールに対してはリンク処理から開始します。以下に、ファイルごとの開始工程を示します（図 1.1「CC32R によるアプリケーション開発フロー」参照）。

表 3.4 入力ファイルと開始処理

拡張子	C/C++コンパイラにおける取り扱い	開始処理
.c	C言語ソースファイル	コンパイル(C)
.cpp	C++言語ソースファイル	コンパイル（C++）
.ms	アセンブリ言語ソースファイル	アセンブル
.mo	オブジェクトモジュールファイル （C++の場合はロードモジュール作成専用）	リンク
.ml	オブジェクトモジュールファイル （C++の場合はライブラリ作成専用）	ライブラリ作成 （-mklib指定時のみ）
上記以外	オブジェクトモジュールファイル	リンク

3.1.7 出力ファイル名の命名

出力ファイル名は -o オプションで指定した名前となります。指定しなかった場合、C/C++ コンパイラによって自動的に以下のように命名されます（表 3.5）。

ただし、入力ファイルを複数指定し、かつ、出力ファイルがロードモジュールでない場合、-o（小文字）オプション指定は無視され、-o オプションを指定しなかった場合と同様、表3.5の規則によって命名された出力ファイルが生成されます。

表3.5 出力ファイル名（デフォルト）

ファイル名	内容
file.mi	プリプロセッシングの結果出力される、C言語ソースファイル。 ファイル名は、C言語ソースファイル名の拡張子が.miに置き換わったもの。
file.ms	コンパイルの結果出力される、アセンブリ言語ソースファイル。 ファイル名は、C言語ソースファイル名の拡張子が.msに置き換わったもの。
file.mo *file*.ml	コンパイルまたはアセンブルの結果出力される、オブジェクトモジュールファイル。ファイル名は、ソースファイル名の拡張子が.moに置き換わったもの。 C++言語を-mklibオプション付きでコンパイルする場合にはfile.mlを選択する。それ以外ではfile.moである。
am.out	リンクの結果出力される、ロードモジュールファイル。
file.elf	ロードモジュールファイルと同時に生成される、ELF/DWARF形式ロードモジュールファイル。C++言語のソースデバッグを行う場合は、この名前のロードモジュールをダウンロードしてください。

3.1.8 コマンド実行中の表示

複数の入力ファイルを指定した場合、C/C++ コンパイラは実行中に以下のような実行状況を知らせる表示を行います。

■ コンパイル開始時　　コンパイルを開始したファイル名が出力されます（入力ファイルにソースファイルがある場合）。

■ リンク開始時　　「Linking」というメッセージが出力されます。

入力ファイル名の指定がなければ C/C++ コンパイラは何もせずに終了します。この場

合、開始および終了に関するメッセージ表示はありません。

3.1.9 プログラムをリンクするまでの手順

C++ プログラムをリンクするまでの標準的な手順を、コマンドライン指定例により説明します。(% はプロンプトを表します。)

(1) C++ ソースファイルのコンパイル

例)

```
% cc32R -c -Otime file1.cpp
% cc32R -c -Otime file2.cpp
% cc32R -c -Otime file3.cpp
```

C++ ソースファイル（～ .cpp）をコンパイルし、オブジェクトモジュールを作成します。

上記例では、C++ ソースファイルをコンパイルして、file1.mo, file2.mo, file3.mo の３つのオブジェクトモジュールファイルと、C++ 用の特殊なファイルを生成します。

(2) スタートアップファイルのアセンブル

例)

```
% cc32R -c start.ms
```

スタートアップファイル（アセンブリ言語）をアセンブルし、オブジェクトモジュールファイルを作成します。

スタートアップファイルでは、プログラムの動作に必要な、CPU の各種設定やセクションの初期化、main 関数の呼び出しなどの各種設定を行います。
詳しくは、第７章「組み込み用アプリケーションの作成」を参照ください。

(3) リンク

例)

```
%  cc32R start.mo file1.mo file2.mo file3.mo -o loadmodule.abs
   -SEC P=400,C,D,CTOR,VTBL,@D=804000,B,COMMON -l m32RcR.lib
```

上記(1),(2)で作成したオブジェクトモジュールファイルをリンクし、ロードモジュールファイル（上記例では loadmodule.abs）を生成します。

その際、ライブラリファイル(-l オプション)や、セクションの配置順序やアドレス(-SEC オプション)の指定を行うこともできます。

上記例にある、-SEC P=400,C,D,CTOR,VTBL,@D=804000,B,COMMON は、cc32R で C++ のコンパイルおよびリンクの際に作成される、P, C, D, B, CTOR, VTBL, COMMON という７つのセクションの配置順序やアドレスを決定する指定です。
具体的には 400 番地(16 進数)以降に P,C,D(初期値),CTOR および VTBL セクションの順に配置し、804000 番地(同)以降に D(実行時領域),B および COMMON セク

ションの順に配置することを意味します。

リンク時に、リンカ lnk32R を直接起動しないでください。必ず、コンパイルドライバ cc32R でリンクを行ってください。(例： cc32R *.mo)
(C 言語、アセンブリ言語で記述されたオブジェクトのみをリンクする場合は、lnk32R を直接起動してリンクしても問題ありません。)

3.1.10 ライブラリの作成

オブジェクトモジュールファイルを使ってライブラリを作成するには、次のように cc32R に -mklib オプションを指定する方法を採ってください。

```
% cc32R -mklib -o mylib.lib lib1.mo lib2.mo lib3.mo
```

これにより、lib1.mo, lib2.mo および lib3.mo から mylib.lib というライブラリを作成します。

ライブラリファイル作成時に、ライブラリアン lib32R を直接起動しないでください。ライブラリファイルを作成する場合にも、コンパイルドライバ”cc32R”を通して(-mklib オプションを使用)ライブラリアンを起動してください。
(C 言語、アセンブリ言語で記述されたオブジェクトのみでライブラリファイルを作成する場合は、lib32R を直接起動も問題ありません。)

3.2 C/C++コンパイラの起動オプション

3.2.1 C/C++コンパイラ起動オプション

以下に、C/C++ コンパイラの各起動オプションの機能について示します。

表 3.6 C/C++ コンパイラ起動オプション(1/11)

オプション	機能
-access=アクセス制御ファイル	ベースレジスタ機能を使用する際に指定します。 アクセス制御ファイル(Access Control File)に記述された内容に基づき、 (1) デフォルトでDあるいはBセクションに割り付けられるオブジェクト(変数、構造体など)に対するアクセス (2) 固定アドレス上のオブジェクトに対するアクセスのビット命令と２つのアクセス（読み／書き）について、16レジスタ相対間接のアドレッシングを用いたコードを生成します。
-c（小文字）	コンパイルのみ行い、オブジェクトモジュールファイル（*file*.mo）を生成します。-mklibオプションと同時に使用した場合には、オブジェクトモジュールファイル（*file*.ml）を生成します。
-C（大文字）	Cプリプロセッサの処理において、コメントの削除を行いません。（C++ソースファイルでは無効です。）
-cs（小文字）	コンパイルのみ行い、オブジェクトモジュールファイル（*file*.mo）を生成します。また、Cソース行、C++ソース行を含むアセンブリ言語ソースファイル（拡張子は “．cs”）も生成します。
-CS（大文字）	コンパイルのみを行い、Cソース行、C++ソース行を含むアセンブリ言語ソースファイル（拡張子は “．ms”）を生成します。 オブジェクトファイルは生成しません。
-constr	文字列定数をCセクションに割り付け、ROM領域への配置を可能にします。
-D *name* -D *name=def*	defで指定する名前または定数を、*name*というマクロに定義します。defを省略した場合は、*name*=1となります。
-e *entrypoint*	ロードモジュールのエントリポイントを、*entrypoint*（シンボル）に設定します。リンク時に有効な指定です。

表3.6 C/C++ コンパイラ起動オプション（2/11）

オプション	機能
-ecpp	Embedded C++言語仕様に基づいてコンパイルします。 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、エラーメッセージを出力します。 Embedded C++言語仕様では、多重継承、仮想基底クラスをサポートしていません。多重継承、仮想基底クラスを記述した場合は、エラーメッセージ"Embedded C++ does not support multiple or virtual inheritance"を出力します。 本オプションは、exception オプションと同時に指定することはできません。
-E	C++フロントエンド、あるいはCプリプロセッサのみを起動し、結果を標準出力に出力します。 本コンパイラは、ソースファイルの拡張子により、C++フロントエンド、Cプリプロセッサを切り替えて起動します。
-exception	例外処理機能を有効にします。 C++例外処理機能(try,catch,throw)を有効にします。 -exception オプションを指定した場合、コード性能が低下する可能性があります。 本オプション省略時解釈は、-noexception です。 ファイル間で例外処理機能を有効にするには以下を行ってください。 ・ -rtti=on を指定する。 -exception オプションと -ecpp オプションを同時に指定することはできません。
-noexception	例外処理機能を無効にします。 C++例外処理機能(try,catch,throw)を無効にします。 本コンパイラは、デフォルトでは、-noexception オプションが指定されています。
-float_only	double型を、強制的にfloat型として扱います。 -m32re5 オプションとの併用で、浮動小数点での演算全てを FPU命令の対象にすることができます。本オプションの詳細は、A.5節を参照してください。
-fminst	FMADD(積和演算命令), FMSUB(積差演算命令)を使ってコード生成します。-m32re5オプションが無効な場合は無視します。本オプションの詳細は、A.5節を参照してください。

表 3.6 C/C++ コンパイラ起動オプション（3/11）

オプション	機能
-g	デバッグ時に必要な情報（デバッグ情報）をオブジェクトモジュールファイルまたはロードモジュールファイルに出力します。 本オプションは、デフォルトで有効になっています
-I *path*	ヘッダファイルを検索するディレクトリのリストに*path*を追加します。 ヘッダファイルの検索順序は以下のとおりです。 ① ソースファイルの存在するディレクトリ ② 本オプションで指定されたディレクトリ ③ 環境変数M32RINCに設定されているディレクトリ ただし、#include<*file*.h>の形式で指定されたヘッダファイルの検索時は、上記①の検索がスキップされます。
-l *lib*	*lib*という名前のライブラリを指定します。ライブラリの検索順序は以下のとおりです。 ① -Lオプションで指定したディレクトリ ② 環境変数M32RLIBに設定されているディレクトリ
-L *dir*	ライブラリの検索ディレクトリを指定します。
-lang={cpp|c}	ソースプログラムの言語を指定します。 lang=c を指定した場合、C プログラムとしてコンパイルします。 lang=cpp を指定した場合、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。 例： cc32r test.c Cプログラムとしてコンパイル。 cc32r test.cpp C++プログラムとしてコンパイル。 cc32r -lang=cpp test.c test.cをC++プログラムとしてコン パイル。 lang=c を指定した場合、-ecpp オプションは無効になります。
-M	makefileの依存関係を生成するために、Cプリプロセッサのみを起動し、結果を標準出力に出力します。
-MAP *map_filename*	*map_filename*という名前のマップファイルを出力します。リンク時に有効な指定です。

表3.6 C/C++ コンパイラ起動オプション（4/11）

オプション	機能

-MEM *addr1*,*addr2*	C/C++プログラムをROM化するために、各セクションを適するメモリ領域に配置するように制御するオプションです。このオプションは、-SECオプションの簡易版で、セクションがP、D、B、Cの4種類から構成される場合に有効です（ユーザーが独自に用意したセクションがある場合は使えません）。
	
	アドレスは 16進数で指定します。ただし、英文字で始まる16進数の場合は先頭に0を付加してください。
	
	*addr1*には、RAM領域（D,Bセクション配置領域）の先頭アドレスを指定します。セクションのリンク順序はD→Bの順になります。*addr1*番地からのRAM領域には、領域が確保されるだけで、その領域に初期値データは出力されません。
	
	*addr2*には、ROM領域（P,CセクションおよびDセクションの初期値データの配置領域）の先頭アドレスを指定します。セクションのリンク順序はP→C→Dの順になります。ここでのDセクション（初期値データ）は、セクション名 ROM_D として出力されます。
	
	この-MEMオプションは、-SECオプションおよび-rオプションとは併用できません。
	
	以下の指定は、どちらも同じ処理を意味します。
	`-MEM 1000,8000`
	`-SEC @D=1000,B,P=8000,C,D`

-mklib={c|a}	ライブラリファイル(拡張子 .lib)を作成するためのオプションです。 ライブラリファイル(拡張子 .lib)の新規作成(-mklib=c)およびオブジェクトモジュールファイル(拡張子 .ml)のライブラリファイルへの追加(-mklib=a)を行います。 必ず -o オプションでライブラリファイル名を明示的に指定してください。 既存のライブラリファイルにオブジェクトモジュールを追加する場合は、必ず -mklib=a を指定してください。 -mklib=c を指定した時、-o オプションで指定したライブラリファイルが既に存在する場合、そのライブラリファイルは、いったん削除され、新規作成となります。 -mklib=a を指定した時、指定したライブラリファイルが存在しない場合、そのライブラリファイルを新規作成（-mklib=c と同じ動作）します。 -C,-E,-P,-S,-M,-CS は同時に使用できません。 例： 1. cc32R -mklib=c -o xx.lib aaa.ml bbb.cpp ccc.c ライブラリファイル xxx.lib を aaa.ml と bbb.cpp,ccc.c をコンパイルしたオブジェク

表 3.6 C/C++ コンパイラ起動オプション（5/11）

オプション	機能
	トモジュールファイル から作成します。 2. cc32R -mklib=a -o xxx.lib ddd.ml eee.cpp fff.c ライブラリファイル xxx.lib に ddd.ml と eee.cpp および fff.c をコンパイルしたオ ブジェクトモジュールファイル を追加します。 ライブラリファイルを作成するには、-mklibオプションで生成したオブジェクトモジュールファイル(拡張子 .ml)を使用するか、本オプションを使用して作成してください。
-mklib	ライブラリファイル(拡張子 .lib)を構成するオブジェクトモジュールファイル(拡張子 .ml)を作成するためのオプションです。 必ず -c（小文字）オプションを同時に指定してください。対象となるファイルは、拡張子 .cpp, .c, .ms のファイルです。 -C,-E,-P,-S,-M,-CS は同時に使用できません。 例： 1. cc32R -mklib -c xxx.cpp yyy.c オブジェクトモジュールファイル xxx.ml と yyy.ml を作成します。 2. cc32R -mklib -o combine.ml -c d.ms d.ms のファイルからオジェクトモジュールファイル combine.ml を作成します。 ライブラリファイルを作成するには、-mklibオプションで、生成したオブジェクトモジュールファイル(拡張子 .ml)を使用してください。
-.ml=.mo	-mklibオプション指定の際、オブジェクトモジュール名の拡張子（.ml）を.moに変更して処理します。
-noinline	インライン展開機能のキーワード”inline”で、関数に指定されたインライン記憶クラス指定を、無効にします。 本オプションを指定した場合、関数に対するインライン記憶クラス指定は、無効となります。インライン展開機能については、A.4 節を参照してください。
-o*output_filename*	出力ファイル名を*output_filename*にします。このオプションを省略した場合、出力ファイルがロードモジュールファイルならば、ファイル名はam.outとなります。ここで指定するoutput_filenameは、-P、-S、-c、-CS 、-cs オプションにも有効です。

表3.6 C/C++ コンパイラ起動オプション（6/11）

オプション	機能
-O*priority*	最適化を速度重視にするかコードサイズ重視にするかを*priority*で指定します。*priority*にはtimeまたはspaceのいずれかを指定します。-Oと*priority*の間には空白文字は入れません。 -Otime : 速度を重視した最適化 -Ospace : サイズ縮小を重視した最適化 -Otimeと-Ospaceは同時には指定できません。 -O*level*を指定せずに本オプションのみを指定すると、自動的に、-O*level*として「-O7」を指定したことになります。 -O*level*と本オプションの両方を省略した場合、最適化は行われません。 -Oのみを指定した場合、-O*priority*として「-Otime」、-O*level*として「-O7」を指定したことになります。 例： -Otime : 速度重視、-07レベルの最適化 -Ospace -O4 : サイズ重視、-04レベルの最適化 -Ospace -O : サイズ重視、-07レベルの最適化 -O : 速度重視、-07レベルの最適化 本本オプションを指定すると、デバッグ機能に影響があります。詳しくは3.2.3節を参照してください。
-O*level*	最適化のレベルを指定します。-Oと*level*の間にはスペースは入れません。*level*には以下のレベルを指定します。 0: 最適化しない 1: アセンブリ言語レベルの最適化 2 : ローカルな最適化 4 : グローバルな最適化 1、2、4を加算した値を指定すると、各レベルを組み合わせた最適化となります。 -O*priority*を指定せずに本オプションのみを指定した場合,-O*priority*として「-Otime」を指定したことになります。-O*priority*と本オプションの両方を省略した場合、最適化は行われません。 -Oのみを指定した場合、-O*priority*として「-Otime」、-O*level*として「-O7」を指定したことになります 例： -O1 : 1レベルの最適化、速度重視 -O6 : 2,4レベル両方の最適化、速度重視 -O -Ospace : 1,2,4レベルの最適化、サイズ重視 -O : 1,2,4レベルの最適化、速度重視 本オプションを指定すると、デバッグ機能に影響があります。詳しくは3.2.3節を参照してください。 また、レベル4以上の最適化が有効な場合、インライン展開機能を有効にします。インライン展開機能の詳細は、A.4節を参照してください。

表3.6 C/C++ コンパイラ起動オプション（7/11）

オプション	機能
-m32re5	M32R/ECU#5拡張命令(M32R-FPUコアのFPU命令)を使ってコード生成を行います。また、浮動小数点定数で非正規化数になるものは0.0に切り詰めます。本オプションの詳細は、A.5 節を参照してください。
-P（大文字）	C++フロントエンド、あるいはCプリプロセッサのみを起動します。拡張子 .mi のファイル(*file*.mi)を生成し、出力結果をこのファイルに出力します。 なお、生成した .mi ファイルには行番号情報は含まれません。 本コンパイラは、ソースファイルの拡張子により、C++フロントエンド、Cプリプロセッサを切り替えて起動します。
-Ql<リンカオプション>	C言語で記述されたオブジェクトのリンク時にlnk32Rに直接渡すオプションを指定します。 なお、リンカオプションとして指定できるオプションは次のいずれかに限られます。これら以外のオプションを指定した場合は、正常に動作しない場合がありますのでご注意ください。 ・-Werrsec コンパイラの -SECオプションに、存在しないセクションを指定するとエラーとする。 ・-Wreloc relocation size overflow エラーの詳細を表示。 ・-Wlimit=message_max リンカlnk32Rが表示するメッセージ数を設定する。 ・-Wnolimit リンカのメッセージを全て表示する。 ・-Wmangle メッセージのデマングル表記の後にマングル名を表示します。 lnk32Rのオプションの詳細は、CC32Rマニュアルの<<アセンブラ編>>のリンカの項目を参照ください。 指定例）-Wreloc, -Wnolimit をリンカに渡す cc32R -Ql-Wreloc -Ql-Wnolimit -o test.abs -SEC P=1000,C,D,B file1.c file
-r	ロードモジュールファイルをリロケータブル形式で作成します。このオプションを省略した場合、ロードモジュールファイルはアブソリュート形式となります。リンク時に有効な指定です。 本オプションは、-MEMオプションおよび-SECオプションとは併用できません。

表3.6 C/C++ コンパイラ起動オプション（8/11）

オプション	機能
-R *old=new*	C/C++コンパイラが生成するセクションの名前を変更します。 C/C++コンパイラは、C言語プログラムをその機能に応じて以下の4セクションに分類し、セクション名を割り当てます（デフォルト処理）。 セクション名 内容 P プログラム C 定数値 D 初期値データを持つ大域変数 B 初期値データを持たない大域変数 これらの名前を変更したい場合、本オプションで *old*に元のセクション名（P、C、D、Bのいずれか）、 *new*に新しいセクション名を指定します。
-rtti={ on | off }	C++で、実行時型情報の有効/無効を指定します。 rtti=on を指定した場合、dynamic_cast、typeid を有効にします。 rtti=off を指定した場合、dynamic_cast、typeid を無効にします。 本オプション省略時解釈は、-rtti=off です。
-S	コンパイルのみを行い、アセンブリ言語ソースファイル（拡張子は“.ms”）を生成します。 オブジェクトファイルは。生成しません。
-strict_standard	標準C++仕様に対して、拡張された機能を使用する場合に、警告を出します。本オプションを指定した場合、 -exception と -rtti=on が有効になります。

表3.6 C/C++コンパイラ起動オプション（9/11）

オプション	機能
-SEC *name* -SEC *name=addr* -SEC *name=addr,name=addr...*	セクションのリンク順序と各セクションの先頭アドレスを指定します。*name*にはセクション名、*addr*にはそのセクションを配置するアドレスを指定します。 先頭アドレスは = の後に16進数で指定します。ただし、英文字で始まる16進数の場合は先頭に0（ゼロ）を付加してください。先頭アドレスの指定がない場合は、直前のセクションの後に続けて配置されます。 @をセクション名の頭に付けると、そのセクションについてはメモリ領域確保の情報だけを出力し、データは出力しないように制御できます（リンカの初期値データ出力抑止機能）。 @によって出力しないようにした初期値データは、別の領域に出力できます（リンカの初期値データ抽出機能）。コマンド行で、セクション名を再度@なしで指定してください。出力されるセクション名は ROM_*name* となります。 例： -SEC @D=1000,B,P=0c000,C,D 上記例では、Dセクションの領域を1000_{16}番地以降に配置し、Dセクションの初期値データをセクションCの後に配置します。初期値データはROM_D というセクション名で、ロードモジュールファイルに出力されます。 本オプションは、-MEMオプションおよび-rオプションとは併用できません。
-switch_by_offset	コンパイラがswitch文に対してテーブルジャンプを使ってコード生成する際、ROM効率の良いオフセットテーブルを使うようにします。本オプションの詳細は、3.2.4 節を参照してください。
-U *name*	*name*で示すマクロ定義（シンボル）を未定義にします。
-v	C/C++コンパイラの各フェーズの起動を確認しながら実行します。
-V	起動メッセージを標準エラー出力に出力します。他のオプション指定はすべて無視され、実際の処理は行われません。
-w	ワーニングメッセージおよびインフォメーションメッセージを抑止します。
-warn_suppress_nested_comment	コメントのネストに関する警告を抑止します。 (C言語でのコンパイル時のみ)

表 3.6 C/C++ コンパイラ起動オプション（10/11）

オプション	機能
-rel16	RAMに配置されるD及びBセクションのシンボルに対するアクセスに、レジスタ相対間接による、ロード／ストア命令を出力します。シンボルへのアクセスは、R12固定のレジスタ相対間接のアドレッシングモードとなります。 なお、本オプションを使用する際に以下の事項に注意してください。 (1) R12レジスタを設定と、”__REL_BASE”シンボルの定義を行ってください (2) D及びBセクションの合計サイズが、64Kバイトを超える場合、本オプションは指定しないでください (3) const修飾の変数（Cセクションに配置される）を別ソースで参照する場合には、必ず const の付いたプロトタイプ宣言が必要です。 (4) R12レジスタの内容を壊さないでください。 本オプションについては3.2.2節も参照してください。
-XX=*symbol_num*	コンパイル時に必要となるシンボル用のメモリを確保します。必要とするメモリのサイズは、シンボルの個数で、”*symbol_num*”に指定します。”*symbol_num*”の値は、デフォルトで、40,000 となっています。 本オプションは、ミドルウェア等の40,000を超えるシンボルを持つソースファイルをコンパイルする時に、指定します。
-small -small -memlarge -medium -large	どのメモリモデル用にコンパイルするのかを指定します。どのオプションも指定しなかった場合には、-small （スモールモデル）でコンパイルされます。 -small は、スモールモデル（コード・データ共に、0x00000000～0x00FFFFFFの範囲内に収まる）用にコンパイルすることを指定します。 -small -memlarge は、スモールモデル（-memlarge付き）（データのみ32ビットメモリ空間フルサポート）用にコンパイルすることを指定します。 -medium は、ミディアムモデル（コードは任意のアドレスA～A+0x00FFFFFFの範囲内、データは、32ビットメモリ空間フルサポート）用にコンパイルすることを指定します。 -large は、 ラージモデル（コード・データ共に、32ビットメモリ空間フルサポート）用にコンパイルすることを指定します。 本オプションについては付録A.2節も参照してください。

表 3.6 C/C++ コンパイラ起動オプション（11/11）

オプション	機能
-stack	本オプションを指定すると、スタック使用量表示ファイル（入力ファイル名の拡張子を” stk”に変換したテキストファイル）を生成します（オブジェクトファイルも同時に生成されます）。 スタック使用量表示ファイルは、C言語ソースファイル毎に生成されます。そのファイルには、関数毎のスタックサイズと、その関数が呼び出している関数名一覧が出力されます。 スタック使用量表示ファイルは、スタックサイズ算出ユーティリティ（stk32R）の入力ファイルとして使用できます -E 、-M、-P オプションの何れかと同時に指定した場合、スタック使用量表示ファイルは生成されません。 インラインアセンブル機能で使用されるスタックの使用状況は、出力されません。 インラインアセンブル機能で定義される関数は、出力されません。 インラインアセンブル機能で呼び出される関数は、出力されません。 アセンブラ関数（アセンブリ言語ソースファイル）にたいして、スタック使用量表示ファイルは生成されません。アセンブラ関数のスタック使用量表示ファイルが必要な場合は、別途テキストファイルを作成してください。 スタック使用量表示ファイルに関する説明、およびスタックサイズ算出ユーティリティ（stk32R）の機能、使用方法に関する説明は、ユーティリティマニュアルのファイルを参照してください。
-zdiv	M32R/ECUシリーズでの整数剰除算の問題を回避するため、コード生成でDIV系命令を出力するときは、その直後にNOP命令を挿入します。 また、asm関数内にDIV系命令がある場合もその直後にNOP命令を挿入します。cc32Rにアセンブリソース(拡張子.ms)を入力する場合は、as32Rでアセンブルする場合と同様です。
-@	本オプションを指定すると、標準エラー出力に出力するメッセージを標準出力に出力します。
-elf_endian={big|little}	オブジェクトファイル中のデバッグ情報をビッグエンディアンにするか、リトルエンディアンにするか、指定します。デフォルトでは、-elf_endian=big ビッグエンディアンです。

3.2.2 -rel16オプションに関するプログラミング時の注意

-rel16 オプションを指定すると、RAM に配置される**DおよびBセクション**のシンボルのアクセスに、レジスタ相対間接によるロード／ストア命令を出力します。シンボルへのアクセスは、R12 固定のレジスタ相対間接のアドレッシングモードとなります。

例:-rel16 オプション指定の有無による出力コードの違い

-rel16 オプション未使用時		-rel16 オプション指定時
LD24	R1, #_symbol	
LDUB	R1, @R1	LDUB R1, @(_symbol-__REL_BASE, R12)

-rel16オプションを使用する場合、プログラミング時に以下の点に注意してください。

■ R12 レジスタの設定および ”__REL_BASE ”シンボル の定義が必要です

本オプションを使用する場合には、プログラム開始時に R12 レジスタの設定、および、”__REL_BASE ”シンボルの定義が必要となります。一般にこれらは、スタートアッププログラム start.ms（7.3 スタートアッププログラムの作成を参照）中に設定します。

R12 レジスタおよび ”__REL_BASE ”シンボルに設定する値は、D および B セクションの先頭アドレスから 32K バイトを加算した値を設定します。例えば、D,B セクションの順で連結され、合計領域が 64K バイト、D セクションの先頭アドレスが、h’20000 番地の場合、h’28000 を設定します。（図3.4、図3.5 参照）

■ D および B セクションのデータの合計サイズは 64K バイト以内に収めてください

本オプションを使用する場合には、D および B セクションのデータの合計サイズが、64K バイト以内で、かつ、連続していなければなりません。

D および B セクションの合計サイズが、64K バイトを超える場合、本オプションは指定しないでください。16 ビットディスプレースメントを超えるため、リンク時に ”relocation size overflow ”等のエラーが発生します。（図3.4、図3.5 参照）

■ const修飾子付き変数のプロトタイプ宣言

const 修飾の変数（C セクションに配置される）を別ソースで参照する場合には、必ず const の付いたプロトタイプ宣言が必要です。

（例）プログラム b で宣言された変数をプログラム a で参照する場合

プログラム a

```
extern          int  aa;
extern   const  int  bb;  /* const */
void foo(void)
{
    if ( aa > bb )
          .....
    else
          .....
}
```

プログラム b

```
        int  aa;
const   int  bb;
void test(void)
{
    aa = 0;
    bb = 1;
    foo();
}
```

■ R12 レジスタの内容を壊さないでください

R12 レジスタは、プログラムの開始から終了まで、レジスタ相対間接の Rsrc レジスタとして使用されますので、R12 の内容を壊さないでください。

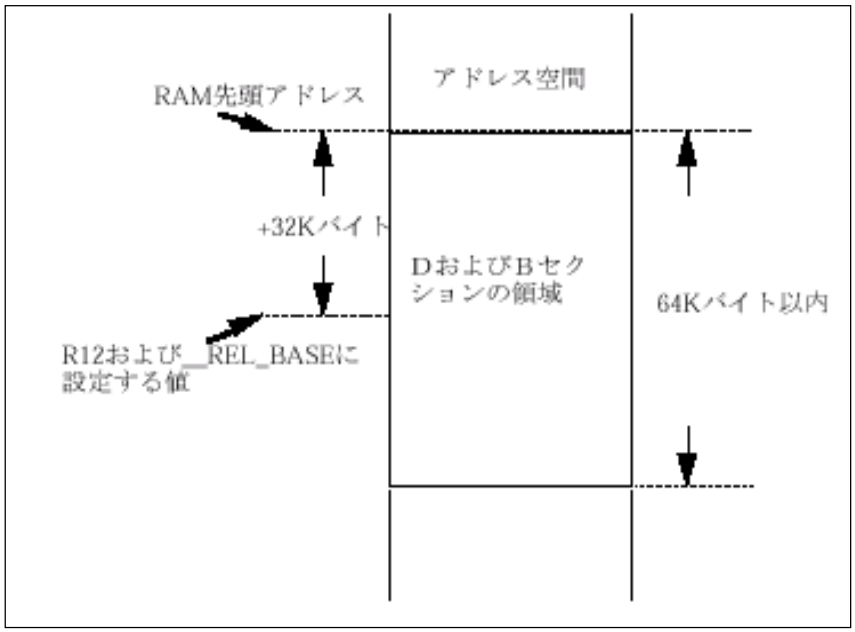


図 3.4 R12 および __REL_BASE に設定する値

```
LOOP1;          STB     R0, @R1
                ADDI    R1, #1
                ADDI    R2, #-1
                BGTZ    R2, LOOP1
loop_cntl:
                .export __REL_BASE              ;新規追加
__REL_BASE      .EQU    h'28000                 ;新規追加
                LD24    R12, #__REL_BASE        ;新規追加
                BL      $_c_main
                .END
```

図 3.5 R12 および __REL_BASE の設定例（start.ms）

3.2.3 最適化オプション指定時のデバッグの制限について

本コンパイラは、デバッグ機能は常に有効となっています。このため最適化オプション(-Opriority,-Olevel)を指定した場合には、最適化時のソースレベルデバッグが可能となります。しかし、最適化によって、不要な行や変数を削除したり、評価順序が変化することがあります。そのため、変数の値をデバッガで確認するタイミングは、以下の条件の場合に限られます。
なお、最適化を指定しなければ、下記の制限はありません。

■ 関数入口のブレークポイント（関数内のブレークポイントが設定できる最初のステートメント）においての条件

（1）グローバル変数であること。
（2）static 宣言されたグローバル変数であること（ただし、関数内で使用されていること）。
（3）static 宣言されたローカル変数であること（ただし、関数内で使用されていること）。
（4）関数の引数であること。

■ 関数出口のブレークポイント（関数内のブレークポイントが設定できる最後のステートメント）においての条件

（1） グローバル変数であること。
（2） static 宣言されたグローバル変数であること。
（3） static 宣言されたローカル変数であること（ただし、関数内で使用されていること）。

■ 任意のブレークポイントにおいての条件

（1） グローバル変数であること（ただし、構造体、共用体、配列のいずれかであること）。
（2） ローカル変数であること（ただし、構造体、共用体、配列のいずれかであり、かつ、それらは関数内で使用されていること）。

3.2.4 -switch_by_offset オプションの機能

コンパイラがswitch文に対してテーブルジャンプを使ってコード生成する際、ROM効率の良いオフセットテーブルを使うようにします。

【注意】

本オプション指定時に生成されるオフセットテーブルは、switch文の最初のcaseを起点にして32Kバイトの範囲までは対応できますが、32Kバイトを越えるような非常に大きなswitch文は記述できません。(リンカlnk32Rでのリンクの際にオーバーフローが発生します)。
この場合は "-switch_by_offset" オプションの指定をはずして再度コンパイルおよびリンクを行ってください。

[コンパイルオプション]

-switch_by_offset

コンパイル時に、cc32Rのコマンドラインオプションに追加指定してください。

[効果]

従来のCC32RでswitchがテーブルジャンプでROMサイズの縮小が図れます。

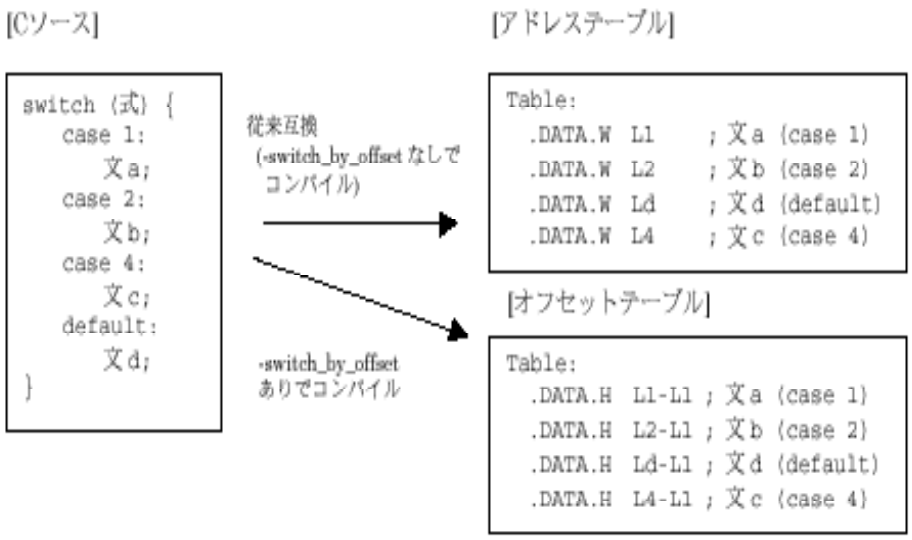
[生成コード]

従来と-switch_by_offsetをつけた場合とで、それぞれに対してCC32Rが生成するswitch文の分岐テーブルを示します。

※ 説明のため、分岐テーブル(アドレステーブルとジャンプテーブル)のみ記載しています。

※ 文a,b,c,dの先頭ラベルが、それぞれL1,L2,L3,Ldであるとします。

※ 線形サーチ(比較と分岐の連続)よりもROMサイズが大きくなる場合は、アドレステーブルやオフセットテーブルを使わずに線形サーチを用いますが、ここでは常にこれらのテーブルを使用する前提で説明しています。



[解説]

従来互換で生成されるアドレステーブルは、分岐アドレスを格納する配列です。
配列の各要素は32bit(4バイト)です。上記のswitch文は式の値が1～4かをチェックした後、Table[式 -1]が示すアドレスに分岐します。

コンパイル時に"-switch_by_offset"を指定すると、アドレステーブルはオフセット(L1 ラベルからの距離を持つ配列)になり、配列の各要素は16bit(2 バイト)と小さくなります。コンパイルはswitch 文に対し、Table[式 -1]が示すオフセット値に原点(L1)を加算したアドレスへ分岐するコードを生成します。

3.3 C/C++コンパイラの起動例

以下にC/C++コンパイラの起動例を示します（%はプロンプト、<RET>はリターンキー入力を示します）。

■ % cc32R -c -v test0.ms test1.c test2.cpp<RET>

-cオプション指定により、test0.ms、 test1.c 、test2.cpp（～.msはアセンブリ言語ソースファイル、～.cはC言語ソースファイル、～.cppはC++言語ソースファイル）についてそれぞれのオブジェクトモジュールファイルを生成します。オブジェクトモジュールファイル名はそれぞれソースファイル名の拡張子が.moに変換されたものとなります（test0.mo、test1.mo、test2.moが生成されます）。

-vオプション指定により、C/C++コンパイラの各フェーズの起動を画面表示で確認できます。

3.4 その他の注意事項

■ スタックフレーム容量の制限（auto 変数のサイズ制限）

関数一つ当たりで確保可能なスタックフレーム容量は、32,764バイトまでです。それを超えるような記述をした場合、エラーが発生し、コードは生成されません。32,764 バイトを超えるデータは、スタティックあるいはグローバルデータとして確保してください。

（例: エラーとなる記述）

```
void foo(void)
{
      char x[32765];  /* 32,764byte 以上の auto 変数 */
      x[0] = 0;
}
```

■ 構造体の代入および戻り値におけるランタイムライブラリの使用

構造体の代入や、戻り値として構造体を返す関数を記述した場合、リンク時にエラーとなることがあります（$_100_builtin_memcopy 関数が存在しないことを示すリンカのエラーメッセージ「error: external symbol not defined: $_100_builtin_memcopy」が表示されます）。これは、代入や戻り値の設定を、ランタイムライブラリ関数 $_100_builtin_memcopy により実施するために起こります。

ランタイムライブラリ関数 $_100_builtin_memcopy は、ANSI-C 標準ライブラリ（m32RcR.lib、etc...）に含まれますので、リンク時にいずれかのライブラリを指定してください。

（例: リンク時にエラーとなる記述）

```
struct s o{
      char xx[100];
}x,y;

void foo(void)
{
      x = y; /* 構造体の代入 */
}
```

■ データアクセス時の注意

C/C++ コンパイラは、アクセスするデータ型のサイズに適した命令を選択しコードを生成します。したがってポインタを介してアライメントがとれていないデータにアクセスするような記述をした場合プログラムの実行時にアドレス例外（AE）が発生します。これは、以下の例に示すようなキャストを行った場合にも、発生することがありますので、十分注意してください。

（例: リンク時にエラーとなる記述）

```
long *p;
char array[10]; /* 4バイト境界のアドレスに配置されている */

void foo(void)
{
```

```
    p = (long *)&array[1]; /* 4バイト境界にないアドレスを
                              long *として代入する */

    *p = 1; /* ST 命令でアクセスするが、アドレスが4バイト境界にない
               ため、アドレス例外(AE)が発生する */
}
```

■ スタックポインタ設定時の注意

C コンパイラは、スタック上のデータのロード／ストア処理に対して LD 命令、ST 命令、LDH 命令、STH 命令などのコードを出力します。したがって、スタートアッププログラムやプログラム中において、スタックポインタに値を設定する場合には、必ず 4 バイト境界にあるアドレスを設定してください（本マニュアル 「7.3 スタートアッププログラムの作成」を参照)。4 バイト境界にないアドレスを設定した場合、プログラムの実行時にアドレス例外（AE）が発生することがあります。

■ 浮動小数点演算関数の呼び出し

浮動小数点演算を行う C 言語プログラムをコンパイルした場合、浮動小数点演算関数（$100_F ～）を呼び出すコードを、内部で生成する場合があります。浮動小数点演算関数は、ライブラリファイル（m32RcR.lib、etc...）に含まれています。浮動小数点演算関数を呼び出すコードが生成された場合、リンク時にライブラリファイルを指定していないとエラーになりますので、C 標準ライブラリ関数を使用していないプログラムでも、リンク時にライブラリファイルを指定してください。

■ 数字で始まるモジュール名の注意

数字で始まる名前のオブジェクトファイルは、生成しないでください（仕様外です)。モジュール名が "ASM32R_MPRO" となり、ライブラリ作成時にモジュール名が重複して、ライブラリファイルを生成できなくなります。また、map32R で生成したマップファイル内のモジュール名が“ASM32R_MPRO”になります。

（例:　生成してはいけないオブジェクトファイル）

```
(1) cc32R -o 1234.mo file.c
(2) as32R -o 1234.mo file.ms
```

■ PC 版使用時の注意点

(1) 浮動小数点数の取り扱い

浮動小数点数値「-0.0」に対しては「＋0.0」に相当するコードを出力します。

(2) パスの区切り記号

パスの区切り記号は「\」です。 ただし、ソースファイル中にあるインクルードファイルのパス指定は、「\」および「/」の両方ともパスの区切りとして認識します。この場合、「/」は入力時のみ認識し、内部では「\」に置き換えて処理します。したがって、ワーニング／エラーメッセージ、リスティングファイル、およびデバッグ情報に出力されるパスの区切り記号はすべて「\」となります。

(3) ファイル名の大文字／小文字

ファイル名の大文字／小文字は同一視します。例えば、「file.c」「FILE.C」「FiLe.C」は、全て同じファイルとして扱います。ただし、生成ファイル名への大文字／小文字の指定は、ロングファイル名に反映されます。

(4) パス指定

ドライブレター付きの相対パスは使用できません。

第４章
C/C++言語仕様

本章では、C/C++ コンパイラが処理する ANSI-C/ISO C++ 言語の、基本的な仕様について説明します。

4.1 トークン（字句要素）

トークン（字句要素）とは、C/C++ コンパイラが C 言語を処理するときのテキストの基本単位です。C/C++ コンパイラでは ANSI 規格に従って以下の要素をトークンとして扱います。

- キーワード　（4.1.1 参照）
- 識別子　（4.1.2 参照）
- 定数　（4.1.3 参照）
- 文字列リテラル　（4.1.4 参照）
- 演算子　（4.1.5 参照）
- 句切り文字　（4.1.6 参照）
- コメント（注釈）文　（4.1.7 参照）

4.1.1 キーワード

キーワードとは、C/C++ 言語において予約語として使用される語です。C/C++ コンパイラの構文解析処理では以下の字句をキーワードと解釈します。

and*	and_eq*	asm	auto
bitand*	bitor*	bool*	break
case	catch*	char	class*
compl*	const	const_cast*	continue
default	delete*	do	double
dynamic_cast*	else	enum	explicit*
export*	extern	false*	float
for	friend*	goto	if
inline	int	long	mutable*
namespace*	new*	not*	not_eq*
operator*	or*	or_eq*	private*
protected*	public*	register	reinterpret_cast*
return	short	signed	sizeof
static	static_cast*	struct	switch
template*	this*	throw*	true*
try*	typedef	typeid*	typename*
union	unsigned	using*	virtual*
void	volatile	wchar_t*	while
xor*	xor_eq*		

図4.1 キーワード

||||| 注意 |||||

“ * ”が、付いているのは、C++ 言語のみのキーワードです。

4.1.2 識別子(C言語のみ)

識別子とは、以下のものを指します。

- 関数名、変数名
- ラベル名
- 構造体、共用体、列挙型の各タグ名
- 構造体、共用体、列挙型の各メンバ名
- マクロ名
- オブジェクト名
- typedef 定義による型名

識別子の文字列は、英字またはアンダースコア（_）で始まります。2 文字目以降には英数字またはアンダースコア（_）が使用できます。1 つの識別子は最大 240 文字まで記述可能です。

表 4.1 英字、数字

分類	文字
英字	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 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
数字	0 1 2 3 4 5 6 7 8 9
英数字	英字と数字

識別子の有効範囲（スコープともいう）は表 4.2 ようになります。識別子はそれぞれの有効範囲内でのみ使用できます。

表 4.2 識別子の有効範囲

識別子	有効範囲
関数外で宣言された変数 関数のプロトタイプ宣言 関数の本体定義	宣言・定義した時点からファイルの終了まで有効。 このような範囲を「ファイルの有効範囲」といいます。
関数本体定義の引数	関数の開始ブロック { から、終了ブロック } まで有効。このような範囲を「ブロックの有効範囲」といいます。
関数のプロトタイプ宣言の引数	宣言した関数プロトタイプ内でのみ有効。このような範囲を「関数プロトタイプの有効範囲」といいます。
ラベル	参照または定義した時点から関数終了まで有効。このような範囲を「関数の有効範囲」といいます。
ブロック内変数	宣言したブロック、すなわち、{ から }までの間で有効（「ブロックの有効範囲」）。ただし、現在ブロックの外の識別子と同じ識別子を宣言した場合、外の識別子は現在ブロック内では無効となり、現在のブロックが終了した時点で再び有効となります。

識別子の名前空間には次の４つがあります（表4.3)。

表4.3　名前空間と識別子(C言語のみ)

名前空間	C/C++コンパイラの解釈
関数名、変数名	ラベル、タグ、メンバ名でなければ関数名または変数名と判断する。
ラベル名	最後にコロン（：）がついた識別子やgoto 文の中の識別子はラベル名と判断する。
タグ名	struct 、union、enumの後にある識別子はタグ名と判断する。
メンバ名	演算子 （->）やピリオド（ .） で参照している識別子はメンバと判断する。

名前空間が異なる識別子どうしは、同じ名前を使用してもかまいません。同名でもCコンパイラが上記のように判断して区別します。逆に、同一空間にある識別子どうしで同じ名前は使用できません。例えば、関数名と同じ名前のラベル名があってもかまいませんが、構造体タグ名と同じ名前の共用体タグ名を記述することはできません。

||||| 注意 |||||

識別子にキーワード（4.1.1参照）は使用できません。

4.1.3　定数

定数とは一定の数値を表す語で、以下の４タイプに分類されます。

- ■ 浮動小数定数　（4.1.3.1参照）
- ■ 整数型定数　（4.1.3.2参照）
- ■ 列挙型定数　（4.1.3.3参照）
- ■ 文字定数　（4.1.3.4参照）
- ■ 真理値定数　（4.1.3.5参照）　[C++言語のみ]

これらの詳細を以下に示します。

4.1.3.1 浮動小数定数

浮動小数定数とは浮動小数を表す定数で、以下のように、仮数部（fractional constant）、指数部、および、添え字（suffix）によって構成されます。

例：

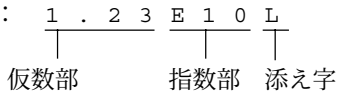


表 4.4 浮動小数定数の構成要素

部分	書式	解説
仮数部	*小数定数*	小数点を含む数字列。以下の3つのうちのいずれか。
	整数部数字列.*小数部数字列*	例： 3.12
	整数部数字列.	例： 123.
	.*小数部数字列*	例： .12
	数字列	小数点を含まない。 例： 123
指数部	e［+|-］*指数* または E［+|-］*指数*	定数を、底を10とした指数で表現するための記述。 符号（+または-）は省略可能。指数は10進数。
添え字	L、l	定数が long double 型であることを示す。
	F、f	定数が float 型であることを示す。
	なし	定数が double 型であることを示す。

※［ ］は省略可能であることを示します。 | はどちらかを選択することを示します。

仮数部が小数定数（小数点を含む数字列）の場合、浮動小数定数だと明らかにわかるので、指数部を省略することができます。これに対し、仮数部が数字列の場合、浮動小数定数かそれ以外の定数かの見分けがつかないため、指数部は省略できません。

■ 仮数部が小数定数の場合の表現形式　*小数定数*［*指数部*］［*添え字*］
■ 仮数部が数字列の場合の表現形式　*数字列* *指数部*［*添え字*］

※［ ］は省略可能であることを示します。

以下に浮動小数定数の記述例を示します。

表 4.5 浮動小数定数の記述例

記述例	解説
1.0	仮数部が小数定数（小数点を含む）の場合 、指数部 は省略可能。
10E3	仮数部が数字列（小数点を含まない）の場合、指数部は省略不可。
1.0E1L	L または l を最後に付けた場合、 long double 型のデータとなる。
1.0E1f	F または f を最後に付けた場合、 float 型のデータとなる。
1.0E1	添え字 （F,f,L,l）を付けない場合、double 型のデータとなる。
.1E1	小数点から始めてもかまわない。

4.1.3.2 整数型定数

整数型定数とは整数を表す定数で、数字で始まり、指数部および小数部を持ちません。表4.6で示す3通りの基数表現ができます。

表4.6 整数型定数の表現

表記法	整数型定数の構成
10 進数	0 以外の数字で始まり、数字で構成されます。 数字：0 1 2 3 4 5 6 7 8 9
16 進数	0x か 0X で始まり、16 進数字で構成されます。 16 進数字： 0 1 2 3 4 5 6 7 8 9A B C D E F a b c d e f
8 進数	0 で始まり、8 進数字で構成されます。 8 進数字：0 1 2 3 4 5 6 7

整数型定数の後ろに u または U を付加すると符号なし定数として処理されます。この指定がなければ符号付き定数として処理されます。また、整数定数値の後ろに l または L を付加すると、long 型の定数として処理されます。この指定がなければ int 型で処理されます。

表4.7 整数型定数のデータタイプ

定数の後ろに	C/C++コンパイラによる取り扱い（データタイプ）
u または U がある場合	符号なし定数
u または U がない場合	符号付き定数
l または Lがある場合	long型の定数
l または Lがない場合	int型の定数

表4.8に、整数型定数の記述例を示します。

表4.8 整数型定数の記述例

記述例	解説
`123`	符号付き int 型 10 進数の123
`123u`	符号なし int 型 10 進数の123
`123l`	符号付き long 型の 10 進数の123
`123ul`	符号なし long 型の 10 進数の123
`0123`	符号付き8 進数の 123
`0x123`	符号付き16 進数の 123

4.1.3.3 列挙型定数

列挙型定数は列挙型のメンバのことで、整数型（int 型）の属性を持った定数です。

例： `enum rgb{ red, green, blue }`

`red`、`green`、`blue` はそれぞれ列挙型定数

4.1.3.4 文字定数

文字定数は、文字またはエスケープシーケンスを表す定数で、文字列をシングルクォーテーション（ ' ）で囲んで記述します。' 自身を文字定数内に含める場合は、エスケープシーケンス \' で表します。文字定数内では、以下のエスケープシーケンスが使用できます。

```
\'    \"    \\    \?    \a    \b    \f    \n    \r    \t    \v
```

図 4.2 エスケープシーケンス（文字定数）

\x の後に 16 進数を続けると 16 進数となり、\ の後に 8 進数を続けると 8 進数となります（16 進数、8 進数は 3 桁まで有効となります）。

以下に文字定数の記述例を示します。

表 4.9 文字定数の記述例

記述例	解説
'\' '	エスケープシーケンス。シングルクォートを表します。
'\n'	エスケープシーケンス。改行を表します。
'\0'	8 進数。文字コード00 (hex)を表します。
'\x7'	16 進数。文字コード 07 (hex) を表します。
'\xFF'	16 進数。文字コード FF (hex)を表します。

4.1.3.5 真理値定数

真理値定数は、true と false の 2つのステータスを持つ定数です。

4.1.4 文字列リテラル

文字列リテラルは、文字列をダブルクォーテーション（ " ）で囲んで記述します。文字列リテラル内では、以下のエスケープシーケンスが使用できます。

```
\'    \"    \\    \?    \a    \b    \f    \n    \r    \t    \v
```

図4.3 エスケープシーケンス（文字列リテラル）

\x の後に 16 進数を続けると 16 進数、\ の後に 8 進数を続けると 8 進数となります。

4.1.5 演算子

演算子は演算を実行するもので、以下のものがあります。

C/C++ 言語での演算子

```
[     ]     (     )     .     ->
++    --    &     *     +     -     ~      !    sizeof
/     %     <<    >>    <     >     <=     >=   ==    !=
^     |     &&    ||    ?:
=     *=    /=    %=    +=    -=    <<=    >>=  &=    ^=    |=    ,
```

C++ 言語のみの演算子

```
::                .*                ->*
typeid
dybamic_cast      static_castreinterpret_cast       const_cast
                                                    new
      delete            delete[]                    throw
                                                    and
      and_eq            biand             bitor     compl
      not               not_eq            or        or_eq
      xor               xor_eq
```

図4.4 演算子

[と] 、（ と ）、および ? と : は、それぞれ一対で使用します（間に式が入る場合もあります）。

各演算子とその意味を以下に示します（表4.10）。

次ページ →

表 4.10　演算子の意味（1/3）

演算子	使い方	意味
::	クラス::メンバ	スコープ解決(C++言語のみ)
::	名前空間::メンバ	スコープ解決(C++言語のみ)
::	::名前	大域的(C++言語のみ)
::	::制限付き名前	大域的(C++言語のみ)
[]	ポインタ[式]	配列の添え字
()	式(式のリスト)	式の構築
()	式(式のリスト)	関数呼び出し
.	オブジェクト.メンバ	構造体、共用体のメンバ選択演算子
->	ポインタ->メンバ	構造体、共用体のポインタからのオフセット
++	左辺値++	後置インクリメント
--	左辺値--	後置デクリメント
typeid	typeid(型)	型識別(C++言語のみ)
typeid	typeid(式)	実行時型識別(C++言語のみ)
dynamic_cast	dynamic_cast<型>(式)	実行時チェック付き変換(C++言語のみ)
static_cast	static_cast<型>(式)	コンパイル時チェック付き変換(C++言語のみ)
reinterpret_cast	reinterpret_cast<型>(式)	チェックなし変換(C++言語のみ)
const_cast	const_cast<型>(式)	const変換(C++言語のみ)
sizeof	sizeof 式	オブジェクトのサイズ
sizeof	sizeof(型)	型のサイズ
++	++左辺値	前置インクリメント
--	--左辺値	前置デクリメント
~	~式	1の補数（C++言語では ”compl ”と記述も可）
!	!式	否定（C++言語では ”not ”と記述も可）
-	-式	単項のマイナス(符号反転)
+	+式	単項のプラス
&	&左辺値	アドレス取得
*	*式	間接参照
new	new 型	構築(領域確保)(C++言語のみ)
new	new 型 (式のリスト)	構築(領域確保と初期設定)(C++言語のみ)
new	new (式のリスト) 型	構築(配置)(C++言語のみ)
new	new (式のリスト) 型 (式のリスト)	構築(配置と初期設定)(C++言語のみ)
delete	delete ポインタ	解体(領域開放)(C++言語のみ)
delete[]	delete [] ポインタ	配列の解体(C++言語のみ)
()	(型)式	キャスト(型変換)

表4.10　演算子の意味（2/3）

演算子	使い方	意味
.*	オブジェクト.*メンバへのポインタ	メンバ選択(C++言語のみ)
->*	ポインタ->*メンバへのポインタ	メンバ選択(C++言語のみ)
*	式*式	2つの算術型オペランドの積(乗算)
/	式/式	2つの算術型オペランドの商(除算)
%	式%式	2つの算術型オペランドの剰余
+	式+式	2つの算術型オペランドの和(加算)
-	式-式	2つの算術型オペランドの差(減算)
<<	式<<式	ビット単位の左シフト
>>	式>>式	ビット単位の右シフト
<	式<式	より小さい(未満)
<=	式<=式	等しい、または、より小さい(以下)
>	式>式	より大きい
>=	式>=式	等しい、または、より大きい(以上)
==	式==式	等価演算子（等しい）
!=	式!=式	2つの算術型のオペランドが等しくないかを比較（C++言語では”not_eq”と記述も可）
&	式&式	ビット単位のAND演算（C++言語では”bitand”と記述も可）
^	式^式	ビット単位のXOR演算（C++言語では”xor”と記述も可）
|	式|式	ビット単位のOR演算（C++言語では”bitor”と記述も可）
&&	式&&式	2つの式の論理的AND演算（C++言語では”and”と記述も可）
||	式||式	2つの式の論理的OR演算（C++言語では”or”と記述も可）
?:	式?式:式	条件選択演算(三項演算子)
=	左辺値=式	代入演算子
*=	左辺値*=式	2つの算術型オペランドの積を求め、結果を代入
/=	左辺値/=式	2つの算術型オペランドの商を求め、結果を代入
%=	左辺値%=式	2つの算術型オペランドの剰余を求め、結果を代入
+=	左辺値+=式	2つの算術型オペランドの和を求め、結果を代入
-=	左辺値-=式	2つの算術型オペランドの差を求め、結果を代入
<<=	左辺値<<=式	ビット単位で左シフト後、結果を代入
>>=	左辺値>>=式	ビット単位で右シフト後、結果を代入
&=	左辺値&=式	ビット単位でAND演算後、結果を代入（C++言語では”and_eq”と記述も可）
|=	左辺値|=式	ビット単位でOR演算後、結果を代入（C++言語では”or_eq”と記述も可）
^=	左辺値^=式	ビット単位でXOR演算後、結果を代入（C++言語では”and_eq”と記述も可）

表 4.10　演算子の意味（3/3）

演算子	使い方	意味
throw	throw式	例外を送出
,	式,式	コンマの左から右へ順に評価し、 右側の値をとる

4.1.6 句切り文字

句切り文字は、他のトークンに対して指示をしたり範囲を指定したりするための文字で、以下のものがあります。

表4.11 句切り文字

句切り文字	用途
[ ]	配列の宣言。
( )	関数の宣言および変数の宣言。
{ }	構造体、共用体のメンバ宣言。初期値の句切り。
*	ポインタの宣言。
,	初期値の句切り。関数の引数の句切り。
:	ビットフィールドの宣言。ラベルの句切り。
=	初期値の開始。
;	宣言の終了。文の終了。
...	関数原型宣言（引数の数が可変である場合に使用）。

[と] 、(と) 、および { と } は、それぞれ一対で使用します（間に式が入る場合もあります)。

句切り文字は、記述のしかたによっては演算子とみなされる場合があります。

4.1.7 コメント（注釈）文

コメント（注釈）は、プログラム中に書いても、コンパイラによって処理されないテキストです。/* で始まり */ で終了します。
また、1行のコメントを記述する場合 "//" でも記述できます。コメントを区切り記号 "//" から、その行末までの間に記述することができます。

コメント（/* で始まり */）は、ネストできません。

コメント文に日本語を記述する場合には、環境変数 "M32RKIN" で指定した文字コードで、記述してください。環境変数 "M32RKIN" が未定義の場合には、EWS版CC32Rの場合EUC形式（拡張UNIXコード）で、PC版CC32Rの場合シフトJIS形式で記述されているものと見なします。

(例)

```
void foo(void)
{
   char    x[3];
   /* コメントが書けます。*/
   x[0] = 0;     // このようにもコメントが書けます
}
```

4.2 データ型

4.2.1 型と型指定子

型はオブジェクトの値や関数の値の意味を決定します。以下に、C/C++ コンパイラがサポートしている型と、宣言時に型を指定するための型指定子（C/C++ 言語で使用する型名）を示します。

[C++ 言語での型]

■ 基本型	void 型	void
	bool 型	bool
	ワイド文字型	wchar_t
	文字型	char, signed char, unsigned char
	符号付き整数型	char, signed char, int (=signed int) , short (=signed short、signed short int) , long (=signed long、signed long int)
	符号なし整数型	unsigned char, unsigned int , unsigned short (=unsigned short int) , unsigned long (=unsigned long int)
	浮動小数点型	float, double, long double
	クラス型	
■ その他	配列型	
	構造体型	struct
	共用体型	union
	列挙型	enum
	ポインタ型	
	関数型	
	ワイド文字型	wchar_t

次ページ →

[C言語での型]

■ 基本型	文字型	char, signed char, unsigned char
	符号付き整数型	char (signed char) ,
		int (=signed int) ,
		short (=signed short、signed short int) ,
		long (=signed long、signed long int)
	符号なし整数型	unsigned char, unsigned int ,
		unsigned short (=unsigned short int) ,
		unsigned long (=unsigned long int)
	浮動小数点型	float, double, long double
■ その他	配列型	
	構造体型	struct
	共用体型	union
	列挙型	enum
	void 型	void
	ポインタ型	
	関数型	
	ワイド文字型	wchar_t

signed、unsignedは符号の有無を示す型指定子です。型指定子が明記されていない、配列型、ポインタ型、および関数型は、決められた形式で宣言することによって、オブジェクトや関数がその型であることを指定します。

(=signed int) のように (= *別名*) で別表現が示されている型指定子は、その別表現でもC/C++コンパイラは同じ意味と解釈します。また、ANSI規格では、符号付きまたは符号なし整数型の定義に文字型を含んでいるため（表4.12参照)、上記リストにおいてintとsigned intは重複して記載されています。

||||| 注意 |||||

int 型は signed/unsigned の指定が無い場合、signed int とみなされます。
long int型は signed/unsigned の指定が無い場合、signed long とみなされます。
short int型は signed/unsigned の指定が無い場合、signed short とみなされます。
enum 型はsigned int 型です。
wchar_t 型はunsigned short 型です。

4.2.2 型の分類

型は、その性質と観点によってさまざまな表現で分類されます。例えば、整数型（文字型含む）、浮動小数点型、そしてポインタ型を総称してスカラ型と呼びます。このような型の分類を表 4.12 および図 4.5 に、また、その他の型表現を表 4.13 に示します。本書では以降、これらの用語を用いて解説することがあります。

表 4.12 型の分類

分類名	該当する型
文字型	char、 unsigned char、wchar_t (C++のみ)
符号付き整数型	signed char（文字型）、 int、short、long
符号なし整数型	unsigned char（文字型）、 unsigned int、unsigned short、unsigned long
浮動小数点型	float、double、long double
基本型	符号付き整数型、符号なし整数型、浮動小数点型
整数型	符号付き整数型、符号なし整数型、列挙型（enum）
算術型	整数型、浮動小数点型
スカラ型	整数型、浮動小数点型、ポインタ型
集合体型	構造体型（struct）、配列型、クラス型
導出宣言子型	配列型、ポインタ型、関数型、クラス型
導出型	配列型、ポインタ型、関数型、構造体型、共用体型、クラス型
bool型	bool

表 4.13 その他の型表現

型表現	解説
混成型	同じ型をもつ二つの型から作られた型。
不完全型	C/C++コンパイラが必要とする情報が足りないオブジェクトへの型（例：サイズ指定のない配列型）。
オブジェクト型	オブジェクト（関数でない）への型。
関数型	関数形式（戻り値の型、引数の型と数を持つ）の型。 例：char *(*func1)(int,int) charへのポインタを返し、2つのint型の引数をもつ関数へのポインタfunc1。
修飾あり型	型修飾子（constまたはvolatile）が付いている型。
修飾なし型	型修飾子（constまたはvolatile）が付いていない型。
先頭型（top types）	型宣言より、型指定子と型修飾子の意味を取り除いたときの型。基本型の先頭型はその型自身。 例：const int *i; ならば、"pointer to qualified int " でポインタ型（修飾あり型やint型ではない）。

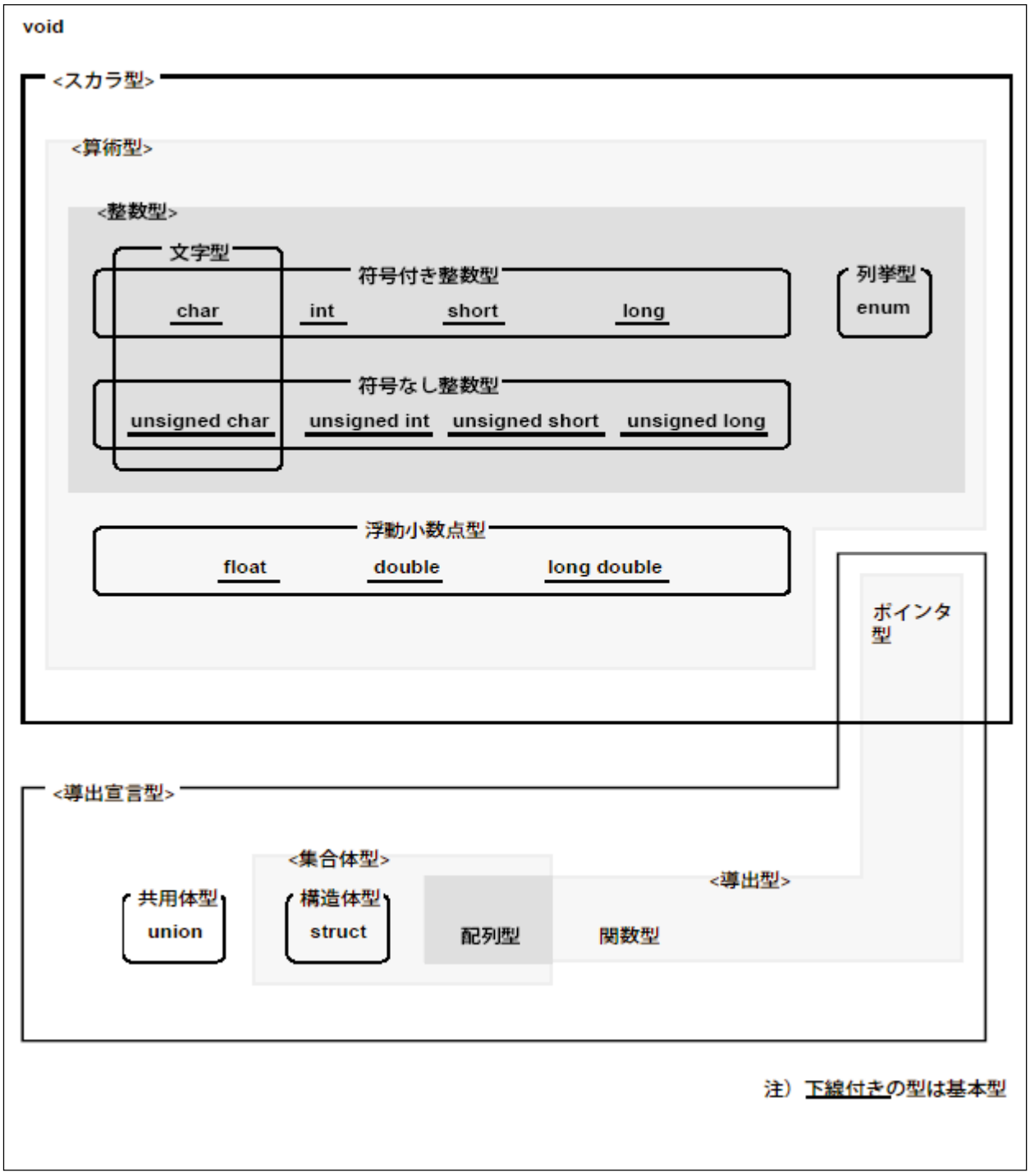


図 4.5　関数の分類（C 言語のみ）

4.2.3 基本型のサイズと極限値

基本型の各データ型に対応するサイズおよび極限値（表現できる数値範囲）は以下のとおりです。詳しい内部表現については第５章 「データの内部表現」を参照してください。

表 4.14 データ型のサイズ

データ型	サイズ	最小値	最大値
bool	1バイト（8ビット）	false	true
char	1バイト（8ビット）	-128	127
unsigned char	1バイト（8ビット）	0	255
short	2バイト（16ビット）	-32768	32767
unsigned short	2バイト（16ビット）	0	65535
int	4バイト（32ビット）	-2147483648	2147483647
unsigned int	4バイト（32ビット）	0	4294967295
long	4バイト（32ビット）	-2147483648	2147483647
unsigned long	4バイト（32ビット）	0	4294967295
float	4バイト（32ビット）	1.17549435e-38F	3.40282347e+38F
double	8バイト（64ビット）	2.2250738585072014e-308	1.7976931348623157e+308
long double	8バイト（64ビット）	2.2250738585072014e-308	1.7976931348623157e+308

4.2.4 浮動小数点型のデータフォーマット

浮動小数点型のデータは、IEEE-754 フォーマットとします。IEEE-754 とは、米国電気電子技術者協会(IEEE)で規定された実数の内部表現形式です。浮動小数点型データの内部表現形式については、第５章 「データの内部表現」を参照してください。

4.2.5 型修飾子

C/C++ コンパイラがサポートするデータ型には型修飾子（const、volatile）を付けることができます。

■ const　オブジェクトの値が変更できないことを示す型修飾子。const で修飾された型で宣言されたオブジェクトには、それ以降、値を代入できません。

例：
```
const char c=1;  ―――(a)
const char *ptr; ―――(b)
```

(a)変数 c は 1 に固定され、値は代入できません。
(b)ポインタ ptr が指す内容は固定され、変更できません。ただし、ptr は変更可能です。

■ volatile　オブジェクトが変更される可能性があることを示す型修飾子。volatile を指定すると、C/C++ コンパイラはそのオブジェクトに対する最適化を行わずに、書き込み／読み出しの処理をそのままオブジェクトコードに出力します。

例えば、割り込み等で書き換えられる領域やメモリマップ I/O 領域に対して volatile を指定し、その領域の内容が変更される可能性があることを示します。

例：　次のようなコードの場合、C/C++ コンパイラは最適化を行わず、すべてそのままオブジェクトコードに落とします。

```
volatile  int i;
volatile  int j;
          j = i;
          j = i;
```

4.2.6　記憶クラス指示子

C/C++ コンパイラがサポートするデータ型には、以下の５つの型記憶クラス指示子を付けることができます(C++ 言語では記憶域種別指定子)。なお、関数内において記憶クラス指示子を指定しなかった変数は、auto を指定したものとみなされます。

■ auto　そのオブジェクトが自動記憶クラスであることを宣言します。関数内のオブジェクトに対してのみ宣言でき、そのオブジェクトはブロック内（{と}の間）でのみ有効となり、ブロックを終了すると記憶域は解放されます。

例：　`auto int a=1;`
この宣言があるブロック内でのみ a が使用可能。

■ extern　そのオブジェクトまたは関数が外部記憶クラスであることを宣言します。外部定義された変数や関数を、当該ファイルで使用するための宣言です。

例：　`extern int val;`
別ソースファイルで宣言されている変数 val を、
当該ソースファイルで使用するための宣言。

■ static　そのオブジェクトまたは関数が静的記憶クラスであることを宣言します。それらを当該ファイル内でのみ使用するための宣言です。

例：　`static void f(int);`
当該ソースファイルだけで使用する関数 f のプロトタイプ宣言。
当該ソースファイルで使用するための宣言。

■ register　そのオブジェクトがレジスタ記憶クラスであることを宣言します。頻繁にアクセスされるオブジェクトに対し、高速アクセスができるよう要求するための宣言です。関数内のオブジェクトに対してのみ宣言でき、そのオブジェクトはブロック内（{と} の間）でのみ有効となります。本指示子で宣言された変数をとくに「レジスタ変数」と呼びます。

例：
```
register reg;
```
reg は、頻繁に使用される変数。

■ typedef　型に別名を定義します（typedef は ANSI 規格により記憶クラス指示子に分類されていますが、実際には記憶クラスを指定するものではありません）。
C++ では、typedef は記憶クラス指示子ではなく、宣言指定子に分類されます。

例：
```
typedef char * STR;
```
今後、型指定子 STR で宣言したオブジェクトは char へのポインタ型として扱われる（`STR str;` は、`char * str;` と同義）。

■ リンケージ指定 （extern “C”）

C/C++ コンパイラは、関数名を符号化して、多重定義を可能にしています。このため、C++ 言語から C 言語で記述されたの関数を呼び出すには、「extern "C" 」宣言を使用して、この符号化が行われないようにする必要があります。
また、C 言語から C++ 言語で記述された関数を呼び出す場合にも、C++ 言語で記述された関数に対して、「extern "C"」 宣言を使用する必要があります。
「extern “C”」宣言された関数は、戻り値型、および関数または関数へのポインタを持つすべての引数が、C 言語の規則に従います。

例：
```
// C/C++間の関数の呼び出し
extern  "C" {
        void foo(void);
}
int foo(void) {
        return i;
}
```

C++ 言語から、C++ 言語の関数を呼び出す際に、「extern “C++”」 と明示的に宣言することも可能です。

■ mutable　mutableは、非const かつ　非static データメンバに適用することの出来る記憶クラス指定子です。mutable指定されたメンバは、constオブジェクトの一部であっても、constにはなりません。

4.3 型変換

4.3.1 明示的型変換（キャスト）

オブジェクトまたは関数の型は、キャスト演算子を使って一時的に変換（キャスト）できます。

型変換の指定形式を図 4.6 に示します。なお、左辺値に対してはキャストできません。

形式　（*新しい型指定子*）*識別子*；

解説　*新しい型指定子*　：スカラ型または void
識別子　：スカラ型を持つ識別子

例

```
int a = 10;
double x = (double)a / 20;
```

int 型の a の演算値を double 型の x に代入するために、
double 型に変換する。

図 4.6　明示的な型変換（キャスト）の書式

元の型と適合する型に変換した場合、オブジェクトおよび関数の値は変わりません。元の型と異なる型に変換した場合は 4.3.2「暗黙の型変換」の表 4.15、表 4.16 のように、変換されます。

4.3.2 明示的型変換（C++ のみのキャスト演算子）

4.3.2.1 dynamic_cast

dynamic_cast は、実行時に型を判別して安全にダウンキャストするためのキャスト演算子です。ポインタの変換の場合、ダウンキャストが可能ならばダウンキャストしたポインタが、不可能ならば null が返されます。

形式　dynamic_cast<型>(式)

次ページ →

4.3.2.2 static_cast

static_cast は float から int への変換などのビットの変換が必要なキャストや、アップキャスト、型のテストを行わない場合のダウンキャストなどに用います。他のキャストがどれもふさわしくない場合にもこれを使います

形式　　static_cast<型>(式)

4.3.2.3 reinterpret_cast

reinterpret_cast はコンパイラに型の解釈を指示するだけで、実行時にはデータをなにも変換しないような場合に使います。解釈を指示するだけなので、ビットは変更されずただ単にコピーされます。

形式　　reinterpret_cast<型>(式)

4.3.2.4 const_cast

const_cast は const を取り除くのに使います。明らかに値が変更されない関数の引数に const がついていない場合に使います。

形式　　const_cast<型>(式)

4.3.3 暗黙の型変換

C/C++ コンパイラは、明示的に型変換指定をしていなくても、オブジェクトの型を変換することがあります。これを暗黙の型変換といい、以下のような変換があります。

■ enum 型の変換

enum 型定数は int 型に変換されます。

■ bool 型の変換

bool 型と int 型は状況に応じて相互に変換されます。

bool 型		int 型
true	--->	1
true	<---	0 以外
false	<-->	0

■ 整数型拡張（integral promotion）

オブジェクトの値が int で表現できる場合、そのオブジェクトは int に型変換されます。そうでない場合、unsigned int に変換されます。これを「整数型拡張」と呼びます。

■ 符号付き／符号なし整数型の変換

符号付き整数から符号なし整数へ、または、その逆の変換結果は、表 4.15 のようになります。

表 4.15 符号付き／符号なし整数型の変換

元の型（と値）	変換後の型	型変換後の値
正の値を持つ符号付き整数型	元の型と同じ、または、元の型より大きいサイズの符号なし整数型	変わらない。
負の値を持つ符号付き整数型	元の型と同じサイズの符号なし整数型	元の値に、（符号なし型の最大値+1）を足した値。
	元の型より大きいサイズの符号なし整数型	元の値を「元の型より大きいサイズの符号付き整数型」に符号拡張したものに、「符号なし型の最大値+1」を足し、符号なし整数型とした値。
符号なし整数型	元の型と同じサイズの符号付き整数型	元の値の下位ビット値がそのまま符号拡張される（最上位ビットは符号ビット）。
符号付き／なし整数型	元の型より小さいサイズの符号付き整数型	元の値の下位ビット値がそのまま符号拡張される（最上位ビットは符号ビット）。
符号付き／なし整数型	元の型より小さいサイズの符号なし整数型	元の値÷（1+変換後の型で表現できる最大値）で算出される正の剰余。

■ 浮動小数点型の変換

浮動小数点型から整数型、整数型から浮動小数点型、または、浮動小数点型から別の浮動小数点型への変換結果は、表4.16のようになります。

表4.16　浮動小数点型の変換

元の型（と値）	変換後の型	型変換後の値
整数型	浮動小数点型	元の値が表現できる場合、変更なし。正確に表現できない場合、変換後の型で表現できる範囲で、元の値に最も近い値に丸められる。
浮動小数点型	整数型	小数部が切り捨てられる。
浮動小数点型	元の型より小さいサイズの浮動小数点型	元の値が表現できる場合、変更なし。正確に表現できない場合、変換後の型で表現できる範囲で、元の値に最も近い値に丸められる。
float	doubleまたはlong double	変わらない。
double	long double	変わらない。

■ 通常の算術変換

二項演算時には、両オペランドが共通の型になるよう、暗黙の型変換が適用されます。演算結果もその共通の型となります。これを「通常の算術変換」と呼びます。以下に、通常の算術変換のケースを示します。

表4.17　通常の算術変換

一方のオペランド	他方のオペランド	暗黙の型変換
long double	long double以外	他方のオペランドがlong doubleに変換される。
double	double 以外	他方のオペランドがdoubleに変換される。
float	float 以外	他方のオペランドがfloatに変換される。
unsigned long	unsigned long以外	他方のオペランドがunsigned longに変換される。
long	unsigned int	他方のオペランドの値がlongで表せる場合、それがlong型に変換される。そうでない場合、両者ともunsigned longに変換される。
long	long以外	他方のオペランドがlongに変換される。
unsigned int	unsigned int以外	他方のオペランドがunsigned intに変換される。

上記以外の場合は、両オペランドとも int となります。

||||| 注意 |||||

プログラミングの際、暗黙の型変換が起こることに注意してください。

例えば、以下のコーディング例のように、unsigned int 型整数（0 以上の値をとりうる）と、-1 という値持つ int 型データを比較した場合、論理上は -1 の方が小さいことになりますが、実際にはその逆と判定されます。

```
unsigned int u;

if( u>(-1) ){           /* 論理上では常に真と判定されることが期待される */
    printf("True");     /* 真ならばこちらの printf が実行される */
}else{
    printf("false");    /* 実際には偽となり、こちらの printf が実行される */
}
```

これは、次のような暗黙の型変換が実行されるためです。

① 通常の算術変換によって、int 型の -1 が unsigned int 型に変換される。
② ①の際、符号付き／符号なし整数型の暗黙の変換によって、-1 が
「元の値に、（符号なし型の最大値 +1）を足した値」（表 4.15 参照）すなわち
「-1+0xffffffff（int の最大値）+1」で 0xffffffff に変換される。

4.4 前処理命令（Preprocessing Directives）

前処理命令（Preprocessing Directives）とは#で始まる命令で、C/C++コンパイラ内のCプリプロセッサというフェーズで処理されるものです。前処理命令には以下のものがあります。

表4.18 前処理命令

前処理命令	意味
#include	ファイルを挿入する。
#define	マクロを定義する。
#undef	マクロを未定義にする。
#if	条件コンパイルを行う。
#else	条件コンパイルを行う。
#endif	条件コンパイルを終了する。
#elif	条件コンパイルを行う。
#ifdef	条件コンパイルを行う。
#ifndef	条件コンパイルを行う。
#line	行指示を行う。
#error	メッセージを出力し処理を中止する。

予約マクロ（あらかじめ定義されているマクロ）には以下のものがあります。

表4.19 予約マクロ

マクロ名	意味
_ _LINE_ _	現在のコンパイル中ファイルの行番号。
_ _FILE_ _	現在のコンパイル中ファイルのファイル名。
_ _STDC_ _	ANSI 準拠であるときは1。
_ _DATE_ _	現在のコンパイル日付。
_ _TIME_ _	現在のコンパイル時間。
_ _M32R_ _	ターゲットのマイクロプロセッサがM32Rである。
__cplusplus	C言語記述のプログラムを、C++コンパイラでコンパイルする。
c_plusplus	予約マクロ”__cplusplus”と同じ。
__embedded_cplusplus	Embedded C++ としてコンパイルする。（オプション-ecpp指定時のみ）
_EXCEPTIONS	オプション -exception が指定された場合に定義される。

定数式で使用可能な演算子を以下に示します。

```
(       )       &       *       +       -       ~       !
sizeof  /       %       <<      >>      <       >       <=
>=      ==      !=      ^       |       &&      ||      ?
:
```

図 4.7　定数式で使用可能な演算子

4.5 システム予約名

以下の形式の名前は、M32R ファミリ開発環境の中でシステム予約名として規定されており、C プログラム中で使用できます。

形式　　*_番号_名前*

解説　　*番号*　：3 桁の 10 進数字
名前　：任意の英数字およびアンダースコア（4.1.2 参照）からなる文字列

例　　_900_main、_900_INITLIB

図 4.8　システム予約名

システム予約名は番号によって以下のように割り振られています（通常のプログラムでは、000 ～ 099、901 ～ 999 の番号のシステム予約名が使用できます）。

■ オペレーティングシステム予約名(000 ～ 099)

オペレーティングシステム(OS)によって提供される変数、関数のシステム予約名です。その内容は OS ごとに異なります。OS とのインタフェースをとるために使用します。

■ 言語、ライブラリシステム予約名(100 ～ 900)

言語、ライブラリ等の開発支援ツールが用いる予約名です。この名前を使用すると、言語処理系、ライブラリの使用する予約名と衝突する可能性がありますので、プログラム内では使用しないでください。C 標準ライブラリの内部では、この予約名を使用しています。

■ ユーザーシステム予約名(901～999)

ユーザープログラムの中で自由に使用できるシンボル名です。ユーザーシステムの中に階層を設け、シンボル名の衝突を避けるために使用することができます。

なお、Cプログラムの外部定義/参照シンボルに対応するオブジェクトプログラム内での名前は、Cプログラム内での名前の先頭にアンダースコア（ _ ）またはドルマーク($)を付加したものになります。したがって上記（図4.8）のシステム予約名の例でも、オブジェクトファイル内ではそれぞれ _ _900_main、_ _900_INITLIB 、あるいは、$_900_main、$_900_INITLIBとなります。

4.6 言語仕様に関する注意事項

4.6.1 C++言語におけるconst の仕様

const 指定されたオブジェクトが ROM に配置されるとは限りません。
動的初期化を伴うものは ROM に配置できません。

例 1) const int a = func();

例 2) const struct S { int a; S() {} }b;

第５章
データの内部表現

本章では、C/C++ プログラム内の各型のデータが、実際にどのような形でメモリに割り当てられるのかについて説明します。

5.1 メモリ上におけるデータ表現要素

C/C++ 言語の各種データの、メモリ上における表現（内部表現）は、以下の要素によって決まります。

■ データのサイズ　　データが占有するメモリサイズ。

■ データの意味　　データのメモリ上の内容（ビットパターン）と、C/C++ 言語でのデータの値の対応関係。スカラ型データのとる値の範囲や集合型データの要素の割り付け方を規定します。

■ データのアライメント（境界整合）

データの種類によっては、整数2か 4の倍数のアドレスに割り付けると高速に参照できる場合があります。この整数をデータのアライメントといいます。C/C++ コンパイラはデータをその型に応じたアライメントに従って割り付けます。

アライメントが整数nのデータに対して、そのデータはnバイト整合であるといいます。その場合、nの倍数のアドレスを「アライメント境界」または「nバイト境界」と呼びます。nバイト整合のデータは、そのアライメント境界（nバイト境界）をまたいで割り付けられることはありません。

5.2 整数型の内部表現

C/C++ 言語における整数型データの内部表現を以下に示します（表5.1）。

表5.1 整数型の内部表現

型	サイズ	アライメント	符号	最小値	最大値
char	1バイト	1バイト	あり	-2^7（-128）	2^7-1（127）
unsignedchar	1バイト	1バイト	なし	0	2^8-1（255）
short	2バイト	2バイト	あり	-2^{15}（-32768）	2^{15}-1（32767）
unsignedshort	2バイト	2バイト	なし	0	2^{16}-1（65535）
int	4バイト	4バイト	あり	-2^{31}（-2147483648）	2^{31}-1（2147483647）
unsignedint	4バイト	4バイト	なし	0	2^{32}-1（4294967295）
long	4バイト	4バイト	あり	-2^{31}（-2147483648）	2^{31}-1（2147483647）
unsigned long	4バイト	4バイト	なし	0	2^{32}-1（4294967295）
bool	1バイト	1バイト	あり	-	-
リファレンス	4バイト	4バイト	なし	0	2^{32}-1（4294967295）

unsigned 指定のない型は符号付きデータであることを示し、最上位の１ビットが符号を意味します（0 ならば正または0、1 ならば負のデータとなります）。なお、負のデータは 2 の補数の形式で表現されます。

unsigned 指定された型は符号なしデータであることを示し、その値は常に正または0となります。

以下に、各文字型および整数型データの内部表現例を示します（表5.2）。

表 5.2　整数型データの内部表現例

型	内部表現	値
char	0000 0001	1
	1111 1111	-1
unsignedchar	0000 0001	1
	1111 1111	255
short	0000 0000 0000 0001	1
	1111 1111 1111 1111	-1
unsignedshort	0000 0000 0000 0001	1
	1111 1111 1111 1111	2^{16}-1 (65535)
int	0000 0000 0000 0000 0000 0000 0000 0001	1
	1111 1111 1111 1111 1111 1111 1111 1111	-1
unsignedint	0000 0000 0000 0000 0000 0000 0000 0001	1
	1111 1111 1111 1111 1111 1111 1111 1111	2^{32}-1 (4294967295)
long	0000 0000 0000 0000 0000 0000 0000 0001	1
	1111 1111 1111 1111 1111 1111 1111 1111	-1
unsignedlong	0000 0000 0000 0000 0000 0000 0000 0001	1
	1111 1111 1111 1111 1111 1111 1111 1111	2^{32}-1 (4294967295)

5.3 浮動小数点型の内部表現

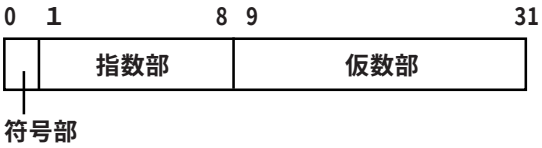
C/C++ 言語における浮動小数点型データの内部表現を以下に示します（表 5.3）。

表 5.3　浮動小数点型の内部表現

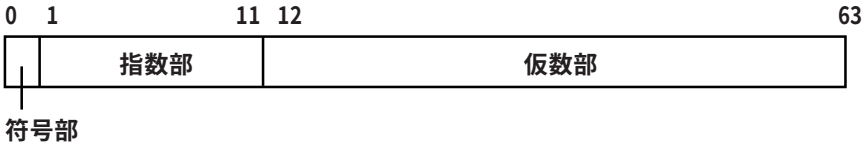
型	サイズ	アライメント	符号	最小値	最大値
float	4バイト	4バイト	あり	1.17549435e-38F	3.40282347e+38F
double	8バイト	4バイト	あり	2.2250738585072014e-308	1.7976931348623157e+308
longdouble	8バイト	4バイト	あり	2.2250738585072014e-308	1.7976931348623157e+308

C/C++ 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。float 型は IEEE の単精度形式（32 ビット）、double 型および long double 型は IEEE の倍精度形式（64 ビット）で表現します。以下にその内部表現の構成を示します。

■ float 型



■ double 型、long double 型



符号部　：浮動小数点数の符号を示す。0 のとき正、1 のとき負。
指数部　：浮動小数点数の指数を 2 の累乗で示す。
仮数部　：浮動小数点数の有効数字（2 進数）を示す。

図 5.1　浮動小数点型の内部表現

以下に、浮動小数点型データの内部表現例を示します（表5.4）。

表5.4　浮動小数点型データの内部表現例

型	内部表現	値
float	0011 1111 1000 0000 0000 0000 0000 0000	1.0
	0011 1111 0100 0000 0000 0000 0000 0000	0.75
double、longdouble		
	0011 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	1.0
	0011 1111 1110 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	0.75

5.4 配列型の内部表現

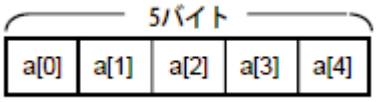
配列型は、配列の要素となるデータを、要素の数だけ連続した領域に割り付けるデータ型です。配列型データのサイズは、「配列要素のサイズ × 要素数」の値になります。配列型データのアライメントは、配列の要素のデータ型のアライメントと同じになります。以下に、配列型データの内部表現例とそのアライメントを示します。

■ 例 1

C/C++ 言語記述

```
char a[5];
```

内部表現



アライメント

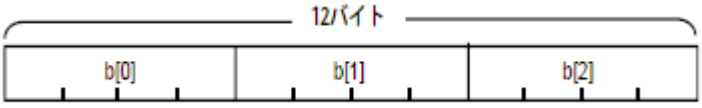
a のサイズは 5 バイト（1 バイト × 5）です。配列要素のデータ型は char なので、char 型と同じアライメント（1）となります。

■ 例 2

C/C++ 言語記述

```
int *b[3];
```

内部表現



アライメント

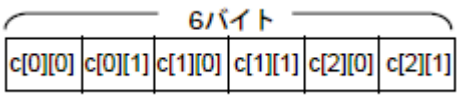
b のサイズは 12 バイト（4 バイト × 3）です。配列要素のデータ型はポインタなので、ポインタ型と同じアライメント（4）となります。

■ 例 3

C/C++ 言語記述

```
char c[3][2];
```

内部表現



アライメント

c のサイズは 6 バイト（1 バイト × 2 × 3）です。配列要素のデータ型は char なので、char 型と同じアライメント（1）となります。

5.5 構造体型（struct）の内部表現

構造体型は、メンバのデータを、メンバの宣言順に連続した領域に割り付けるデータ型です。ただし、ビットフィールドのメンバに対してはビットフィールド特有の割り付け規則が適用されます（5.9「ビットフィールドの内部表現」参照）。

構造体型データのアライメントは、そのメンバのデータ型のアライメントによって決まり、メンバの中で最大のアライメントを持つものと同じアライメントになります。構造体のサイズは、原則としてそのメンバのサイズの和となります。ただし、構造体の各メンバを割り付けるとき、各メンバのアライメントを合わせるために、直前のメンバの領域との間に１～３バイトの隙間が生じる場合があります。

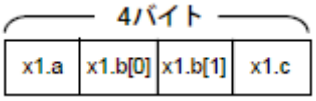
また、構造体型のアライメントはそのメンバの最大のアライメントと一致するため、最後のメンバの領域がアライメントに合ったアドレスで終了していない場合、アライメントに合ったアドレスまでを構造体型の領域として扱います。したがって、構造体型のサイズは、必ずそのアライメントの倍数になります。以下に、構造体型データの内部表現例とそのアライメントを示します（例１～4）。

■ 例１

C/C++言語記述

```
Struct s1{
        char a;
        char b[2];
        char c;
}x1;
```

内部表現



アライメント

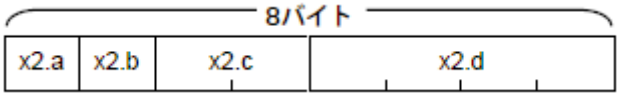
各メンバのアライメントは１バイトなので、全体のアライメントも１バイトになります。構造体のサイズは４バイトになります。

■ 例 2

C/C++ 言語記述

```
struct s2{
        char  a, b;
        short c;
        char *d;
}x2;
```

内部表現



アライメント

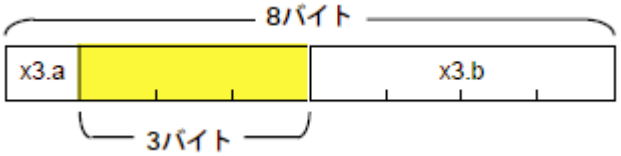
メンバの中に 4 バイトのアライメントを持つポインタ型のものがあるので、構造体のアライメントは 4 バイトになります。

■ 例 3

C/C++ 言語記述

```
struct s3{
        char a;
        int  b;
}x3;
```

内部表現



アライメント

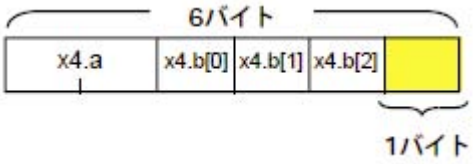
メンバの中に 4 バイトのアライメントを持つ int 型のものがあるので、構造体のアライメントは 4 バイトとなります。メンバ b を割り付けるときアライメントを合わせるために 3 バイトの隙間が生じるため、全体のサイズは 8 バイトになります。

■ 例４

C/C++言語記述

```
struct s4{
        short a;
        char  b[3];
}x4;
```

内部表現



アライメント

メンバの中に2バイトのアライメントを持つ short 型のものがあるので、構造体のアライメントは2バイトとなります。最後のメンバbの領域が奇数バイト目で終っているので、最後に1バイトの領域が付加され、サイズは6バイトになります。

5.6 共用体型（union）の内部表現

共用体型は、そのメンバのデータを同一のアドレスに割り付けるデータ型です。共用体型データのアライメントは、そのメンバのデータ型のアライメントの最大値となります。メンバの中で最大のアライメントを持つものによって、共用体全体のアライメントも４バイト、２バイト、１バイトのいずれかになります。

共用体型のサイズは、原則としてそのメンバのサイズの最大値となります。ただし、共用体型のアライメントが２バイトまたは４バイトで、最大値がアライメントの倍数でないときは、アライメントの倍数に切り上げて共用体の領域として扱います。したがって、共用体型のサイズは、必ずそのアライメントの倍数になります。

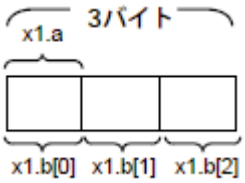
以下に、共用体型データの内部表現の例を示します（例１～３）。

■ 例１

C/C++言語記述

```
union u1{
        char a;
        char b[3];
}x1;
```

内部表現



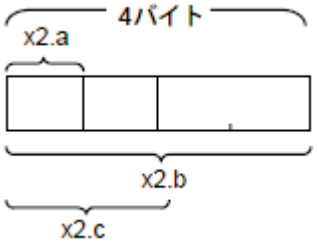
アライメント　各メンバのアライメントは１バイトなので、共用体全体のアライ メントも1バイトになります。サイズは3バイトです。

■ 例 2

C/C++ 言語記述

```
union u2{
        char  a;
        int   b;
        short c;
}x2;
```

内部表現



アライメント

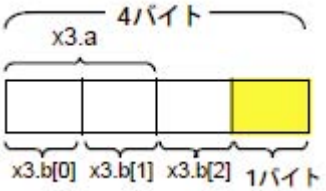
4 バイトのアライメントのメンバがあるので、共用体全体のアライメントは 4 バイトになります。サイズは 4 バイトです。

■ 例 3

C/C++ 言語記述

```
union u3{
        short a;
        char  b[3];
}x3;
```

内部表現



アライメント

2 バイトのアライメントのメンバがあるので、共用体全体のアライメントは 2 バイトになります。最大の領域を持つメンバのサイズが奇数バイトなので、最後に 1 バイトの領域が付加され、サイズは 4 バイトになります。

5.7 列挙型（enum）の内部表現

列挙型は、int 型と同じ内部表現を持ちます。

表5.5 列挙型（enum）の内部表現

型	サイズ	アライメント	符号	最小値	最大値
enum	4バイト	4バイト	あり	-2^{31}（-2147483648）	$2^{31}-1$（2147483647）

列挙型の各メンバ名の値は、特に指定がなければ、0からの整数値をメンバの出現順に与えます。

■ 例

```
enum{ a, b, c }
```

このときの各メンバ名の値は a=0、b=1、c=2 となります。

5.8 ポインタ型の内部表現

ポインタ型データの値は、データや関数を格納するアドレスとなります。

表5.6　ポインタ型の内部表現

型	サイズ	アライメント	符号	最小値	最大値
ポインタ	4バイト	4バイト	なし	0	2^{32}-1（4294967295）
データメンバへのポインタ	4バイト	4バイト	あり	0	2^{32}-1（4294967295）
関数メンバ・仮想関数メンバへのポインタ	8バイト	4バイト	-	-	-

■ ポインタ

C/C++ コンパイラは、ポインタ型データを符号なしデータとして扱います。ポインタ型データを、符号付きの論理アドレス値として使用したいときは、int 型に型変換してから使用してください。

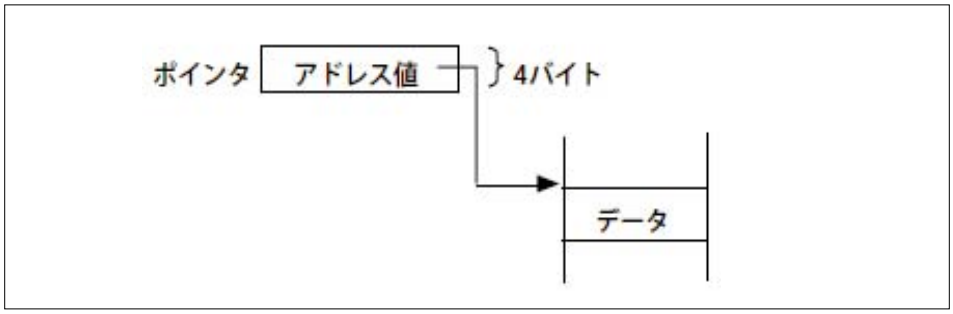


図5.2　ポインタ型データ

■ 関数メンバ・仮想関数メンバへのポインタ

関数メンバ・仮想関数メンバへのポインタは、以下のデータ構造で表現しています。

```
class PMF {
   public:
      short d;    /* オブジェクトのオフセット値 */
      short i;    /* 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 {
            void (*f)();      /* 対象メンバ関数が非仮想関数のときの
                                 関数アドレス */
            long offset;      /* 対象メンバ関数が仮想関数のときの仮
                                 想関数 のオブジェクト中のオフセット */
            };
   } obj;
```

5.9 ビットフィールドの内部表現

ビットフィールドとはビットの集まりで、構造体や共用体を構成する各メンバのデータ形式です。宣言時にはフィールド幅（ビット数）を指定します。

ビットフィールドデータは、可能な限り、バイト（8 ビット）、ハーフワード（16 ビット）、または、ワード（32 ビット）サイズのメモリ領域内に、宣言した順番に詰め込まれます。

5.9.1 ビットフィールドのデータ型

ビットフィールドの宣言に有効な型は signed または unsigned の、char、short、int、および long 型と、bool 型です。(bool 型は C++ 言語のみ。)

||||| 注意 |||||

ANSI-C/ISO C++ 言語仕様では、ビットフィールドに対して許されている型は、int（signed int または unsigned int）のみです。したがって、char や short を用いたプログラムは他のコンパイラでは処理できない場合があります。

実際にビットフィールドを使用する場合、ビットフィールドは、それが宣言された型のサイズを持つデータに拡張されます。指定したフィールド幅は、拡張データの下位ビットから割り当てられます。拡張部分である上位ビットは、ビットフィールドが符号なし(unsigned 型）か符号付きかによって以下のように設定されます。

- ■ 符号なしビットフィールド　　拡張時、上位ビットは 0 に設定される（ゼロ拡張）。
- ■ 符号付きビットフィールド　　拡張時、上位ビットは、ビットフィールドの最上位ビットの値に設定される（符号拡張）。

例えば（図 5.3 参照)、「`char member:2` 」のような、char 型でフィールド幅を 2 ビットと宣言したビットフィールドを使用する場合、上位 6 ビットを符号拡張して 1 バイト（8 ビット）として扱います。`member` を -1 で初期化すると、このビットフィールドの割り付けられたバイトの値は「`1100 0000`」となります。このうちの 2 ビット「11」が `member` の割り付けられたビットです。

また、ビットフィールドは符号付き／なしによって、同じビットパターンでも、表現しうる値の範囲が異なります（下記の例参照)。注意してください。

例：
```
char  a:4              ――― a の値は範囲は -8 ～ 7。1111 の場合、-1。
unsigned char b:4;     ――― b の値は範囲は 0 ～ 15。1111 の場合、15。
```

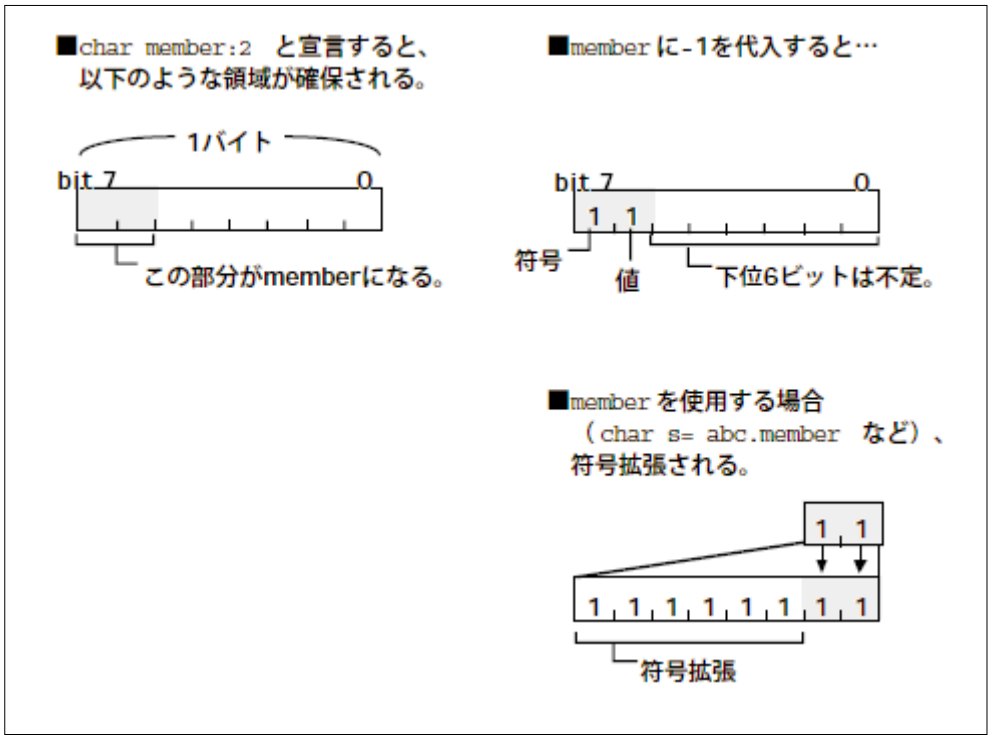


図 5.3　ビットフィールドの内部表現と使用例

||||| 注意 |||||

ビットフィールドは、インプリメンテーションによって取り扱いが異なる場合があります。プログラムの移植等の際、注意してください。

■ int

ANSI-C/ISO C++ 言語仕様により、「signed int」で宣言したビットフィールドは符号拡張されます。しかし、「int」で宣言したビットフィールドはゼロ拡張と符号拡張のどちらが行われるか不定です。そのため、コンパイラによっては、ビットフィールドが符号付きデータであっても、「int」で宣言した場合はゼロ拡張される可能性があります。

```
struct b1 {
    signed int     a:4;
           int     b:4;
}x;
x.a = -1;          /* この代入の結果、x.a の値は -1 となる。*/
x.b = -1;          /* この代入の結果、x.b の値は不定（符号拡張された場合は
                     正しく -1 となるが、ゼロ拡張された場合は 15 となる） */
```

■ signed int

ANSI-C/ISO C++ 言語仕様により、「signed int」で宣言したビットフィールドは符号拡張されます。したがって、フィールド幅として 1 を指定した場合、とりうる値の範囲は -1 ～ 0 となります。0 ～ 1 ではありません。

```
signed int c:1;          /* c のとりうる値は -1 ～ 0 */
```

5.9.2 ビットフィールドの内部表現

ビットフィールドに対して指定できる型指定子のサイズは、1バイト（8ビット、char）、2バイト（16ビット、short）、4バイト（32ビット、int、longおよびbool）です。

ビットフィールドがメモリに配置される場合の規則および例（例1～4）を以下に示します。

■ 配置領域のアライメント境界は型指定子のサイズ単位。

メモリ領域は、型指定子で示すサイズ（1、2、4バイト）を単位とした、全メンバを配置するために十分なサイズが確保されます。例えばint型のビットフィールドだけからなる構造体がある場合、4バイト単位で領域が確保されます。

■ 型指定子の示すサイズの領域に詰め込まれる。

連続した複数のビットフィールドは、1単位（1、2、4バイト）の領域内に、可能な限り詰め込まれます（例1参照）。例えば、int型でフィールド幅4のメンバを3つ含む構造体がある場合、4バイト（32ビット）の領域に全メンバ（合計12ビット）が詰め込まれます。

■ 上位ビット側から下位ビット側へ配置される。

連続したビットフィールドは宣言順に、型指定子で示すサイズの領域の左側（上位ビット側）から右側（下位ビット側）へ配置されます。

■ アライメント境界をまたぐ配置はない。

1つのビットフィールドが、アライメント境界をまたいで配置されることはありません。連続に配置するとアライメント境界をまたいでしまう場合は、境界を越えて次の領域の先頭から配置されます。その結果、ビットフィールドが配置されないビットが発生し、メモリ上でデータが不連続に並ぶ場合もあります。例えば、次に配置されるビットフィールドが4ビット長の場合、アライメント境界まで4ビット以上空きがなければ、境界を越えて次の領域の先頭から配置されます（例2参照）。

■ 0ビット長のビットフィールドは、次の領域に配置される。

次に配置されるビットフィールドが0ビット長の場合、そのビットフィールドは境界を越えて次の領域の先頭から配置されます（例3参照）。

■ 型が混在する場合、各ビットフィールドごとにアライメントが決まる。

構造体や共用体に異なる型のビットフィールド（メンバ）が混在している場合、各ビットフィールドが配置されるごとに、そのビットフィールドの型指定子のサイズが配置領域のアライメントとみなされます。構造体や共用体全体のアライメントは、メンバに使用されている最大サイズの型のサイズとなります。例えば、int型とshort型の2種類のビットフィールドからなる構造体がある場合、全体のアライメントは4バイト（32ビット）

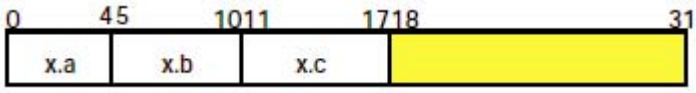
となります。また、int 型データ配置時は配置領域のアライメントが 4 バイト、char 型データ配置時は配置領域全体のアライメントが 2 バイト（16 ビット）であるとみなされます（例 4 参照）。

■ 例 1

C/C++ 言語記述

```
struct b1 {
        unsigned int a:5;
        unsigned int b:6;
        unsigned int c:7;
} x;
```

内部表現



解説

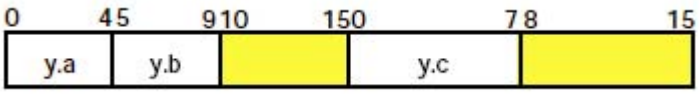
構造体 b1 のメンバは、4 バイト（unsigned int のサイズ）の領域の、左端から配置されます。

■ 例 2

C/C++ 言語記述

```
struct b2 {
        short a:5;
        short b:5;
        short c:8;
} y;
```

内部表現



解説

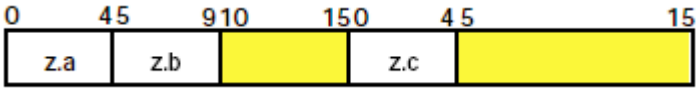
構造体 b2 では、c は領域に残っているビット数（上記の例では 6）では入りきらないので、新しい次の領域に割り当てられます。

■ 例 3

C/C++ 言語記述

```
struct b3 {
        short a:5;
        short b:5;
        short:0;
        unsigned short c:5;
} z;
```

内部表現



解説

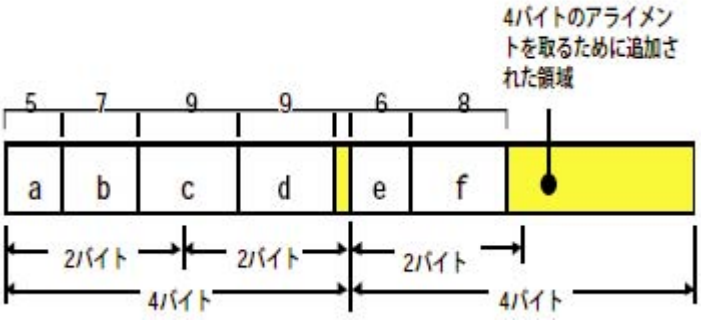
構造体 b3 では、長さが 0 のビットフィールド（short:0）を c の前に指定したため、c は新しい領域に割り当てられます。

■ 例 4

C/C++ 言語記述

```
struct b4 {
        short a:5;
        short b:7;
        int c:9;
        int d:9;
        short e:6;
        short f:8;
 } z;
```

内部表現

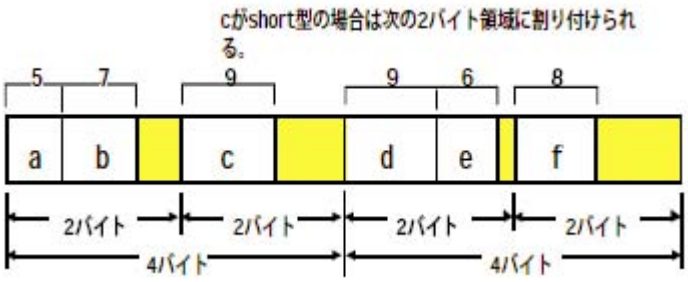


解説

構造体 b4 は、ビットフィールドの型指定子が混合しています。最大の型指定子は int なので、全体のアライメントは 4 バイト（32 ビット）となります。各ビットフィールドの型指定子によって、そのビットフィールドが割り付けられる領域を何バイト領域とみなすかが決まります。c は int 型なのでアライメントが 4 バイトとみなされ、9 ビットを b の次に続けて配置できます。

次ページ →

もし、c が short 型であったとすると、アライメントは 2 バイトとみなされます。b の後ろには、2 バイト境界まで 4 ビットしかないため 9 ビット長の c を続けて配置することはできません。その場合、c は次の 2 バイト領域から配置されます（下図参照）。



5.10 クラス型の内部表現

クラスは、Ｃ言語の構造体を拡張したものです。構造体にはメンバとしてデータしか記述できませんが、クラスにはデータとそのデータを操作する関数を一緒に記述できます。メンバ変数、メンバ関数といい、これらを合わせてクラスメンバといいます。

C++ のクラスでは、クラスメンバに対するアクセスを制限できます。クラスの定義内で「private」、「public」というアクセス指定子で、クラスメンバに対するアクセスを指定します。

```
private:   の後に書かれたメンバをプライベートメンバ
public:    の後に書かれたメンバをパブリックメンバ
```

プライベートメンバは、外部に対し非公開で外の関数からの参照により変更したり出来ません。
プライベートメンバにアクセスできるのは、原則として同じクラスのメンバ関数だけです。
パブリックメンバは、外部に対し公開で、外部のどの関数からもアクセスできます。

5.10.1 クラス型の内部表現

5.10.1.1 境界調整数

クラス型の境界調整数は、仮想関数がある場合は常に 4、そうでない場合はデータメンバの境界調整数のうちの最大値となります。

5.10.1.2 クラス型の割付け方

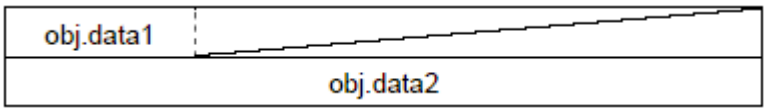
■ 基底クラス、仮想関数がないクラス

構造体データの割付け規則に従ってデータメンバを割付けます。

C++言語記述

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

内部表現



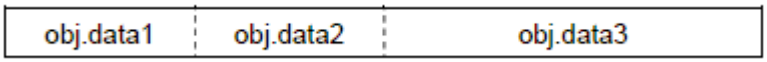
■ 境界調整数が１の基底クラスから派生したクラスの先頭メンバが 1byte データの場合

空き領域を作らないようにデータメンバを割付けます。

C++言語記述

```
class A {
        char data1;
};
class B : public A {
        char data2;
        short data3;
} obj;
```

内部表現



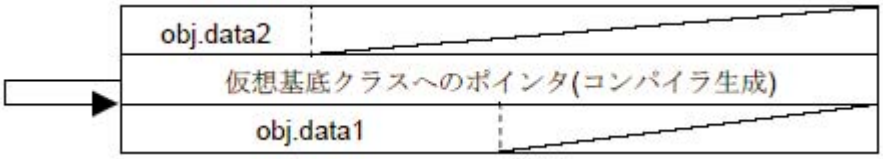
■ クラスに仮想基底クラスがある場合

仮想基底クラスへのポインタを割付けます。

C++ 言語記述

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

内部表現



■ クラスに仮想関数がある場合

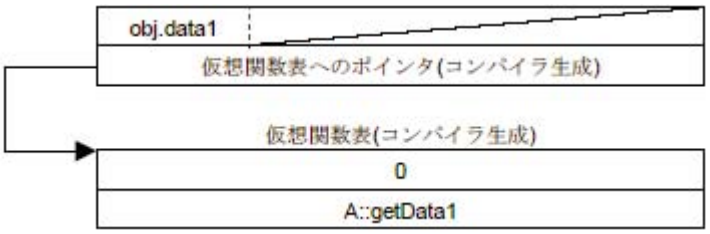
コンパイラは仮想関数表を作成し、仮想関数表へのポインタを割付けます。

仮想関数表は「5.8 ポインタ型の内部表現」の「関数メンバ・仮想関数メンバへのポインタ」で説明したデータ構造の配列です。この配列の 2 番め以降に仮想関数へのポインタが格納されます。

C++ 言語記述

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

内部表現



■ 仮想基底クラス、基底クラス、仮想関数があるクラス

C++言語記述

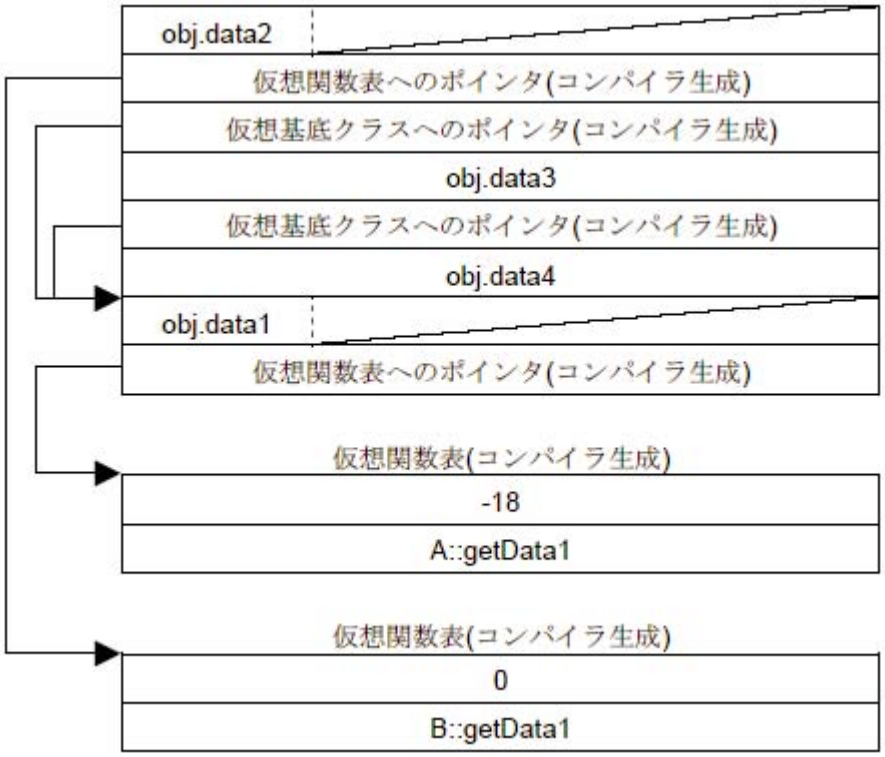
```
class A {
        short data1;
        virtual char getData1();
};

class B : virtual public A {
        char data2;
        char getData2();
        char getData1();
};

class C : virtual protected A {
        int data3;
};

class D : virtual public A, public B, public C {
   public:
        int data4;
        char getData1();
} obj;
```

内部表現



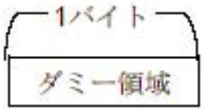
■ 空クラスの場合、または空クラスを規定クラスに持つ空クラスの場合

１バイトのダミー領域を割付けます。

C++ 言語記述

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

内部表現



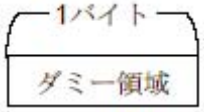
■ 空クラスを基底クラスに持つ空クラスの場合

１バイトのダミー領域を割付けます。

C++ 言語記述

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

内部表現



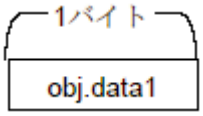
■ 空クラスを基底クラスに持つ空でないクラス

ダミー領域は割付けません。

C++ 言語記述

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

内部表現



第６章
C/C++呼び出し規則

本章では、C/C++ 言語で書かれたプログラム（以下 C/C++ プログラム）呼び出し処理について、C/C++ コンパイラではどのように取り扱うかについて示します。また、アセンブリ言語で書かれたプログラム（以下アセンブリプログラム）とのインタフェースについて示します。

6.1 レジスタの使用規則

6.1.1 汎用レジスタ（R0～R15）の使用規則

M32R には 16 本の 32 ビット汎用レジスタ（R0 ～ R15）があります（図 6.1）。

C/C++ コンパイラは、R14 をリンクレジスタ、R15 をスタックポインタ（SP）として使用します。

また、R0 ～ R13 はワークレジスタとして関数演算の途中結果や変数の一時的に格納するために使用します。このうち、R4 ～ R7 は関数引数用に使用されます。

また、R11 ～ R13 はベースレジスタ機能を使用した場合、ベースレジスタとして使用されます（その場合、ベースレジスタに指定されたレジスタは常にベースアドレスを保持するため、ワークレジスタとして使用できません）。

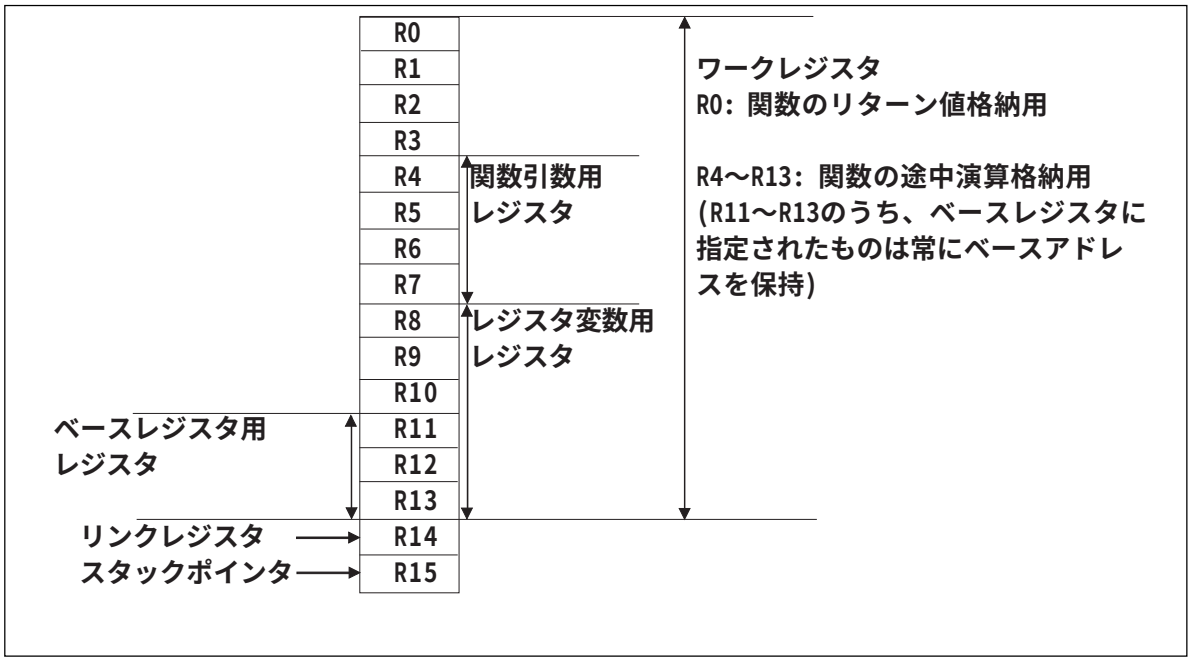


図 6.1　汎用レジスタの用途

以下に、C/C++ コンパイラによる、汎用レジスタの各用途別の使用方法について説明します。

■ ワークレジスタ（R0 ～ R13）

関数演算の途中結果や変数を一時的に格納します。なお、関数からのリターン値は R0 に格納します。

■ 関数引数用レジスタ（R4 ～ R7）

関数の引数を格納します。なお、関数の引数に、浮動小数点型(double, long double)、構造体型、共用体型等を使用した場合、「スタック渡し」となります。

■ ベースレジスタ用レジスタ（R11 ～ R13）

ベースレジスタ機能を使用した場合に、ベースレジスタ用レジスタとして用います。ベースレジスタ用レジスタとして使用されているときは、レジスタ変数用レジスタとしてもワークレジスタとしても使用しません。

■ リンクレジスタ（R14）

R14 はリンクレジスタと呼び、関数呼び出し時にリターンアドレス（戻り番地）を格納します。なお、リンクレジスタは、C/C++ コンパイラにより、ワークレジスタまたはレジスタ変数用レジスタとして使用される場合があります。

■ スタックポインタ（R15）

R15 はスタックポインタ（SP）と呼び、スタック領域の最下位アドレスを格納します。SP によってスタックを管理します。

6.1.2 レジスタの保証規則

C/C++ プログラムの関数を実行した場合、関数呼び出し前の各レジスタ値が、関数実行後においても保証される（関数実行前と変らない）かどうかを以下に示します。（表6.1。○は呼び出し前の値が保証されること、×は呼び出し前の値が保証されないことをそれぞれ示します)。

表6.1 レジスタの保証規則

レジスタ	保証（○）、未保証（×）	備考
R0～R3	×	
R4～R7	×	関数引数用
R8～R10	○	
R11～R13	○	ベースレジスタの場合あり
R14（リンクレジスタ）	×	
R15（SP）	○	
PSW（CR0）注1	○	
CBR（CR1）注2	○	
アキュムレータ	○	

注1) PSW（CR0） プロセッサ状態後レジスタ。M32R の制御レジスタの一つ。スタックモード、割り込み許可、演算結果の条件ビットやそれらの退避値が設定されています。

注2) CBR（CR1) 条件ビットレジスタ。M32R の制御レジスタの一つ。読み出し専用で、直前の演算結果（キャリー、ボロー、オーバーフローなど）が格納されています。

6.2 スタックフレームの基本構成

スタックフレームとは、関数が呼び出されるたびにスタック上に割り付けられる領域です。基本的なスタックフレームは以下に示す構成となっています（領域(0)～(2)が順に、上位アドレスから下位アドレス方向に向かって積まれます)。なお、領域(0)の引数領域は、「6.4.2 スタック渡しとなる場合」に記載されている場合にのみ、スタック上に積まれます。

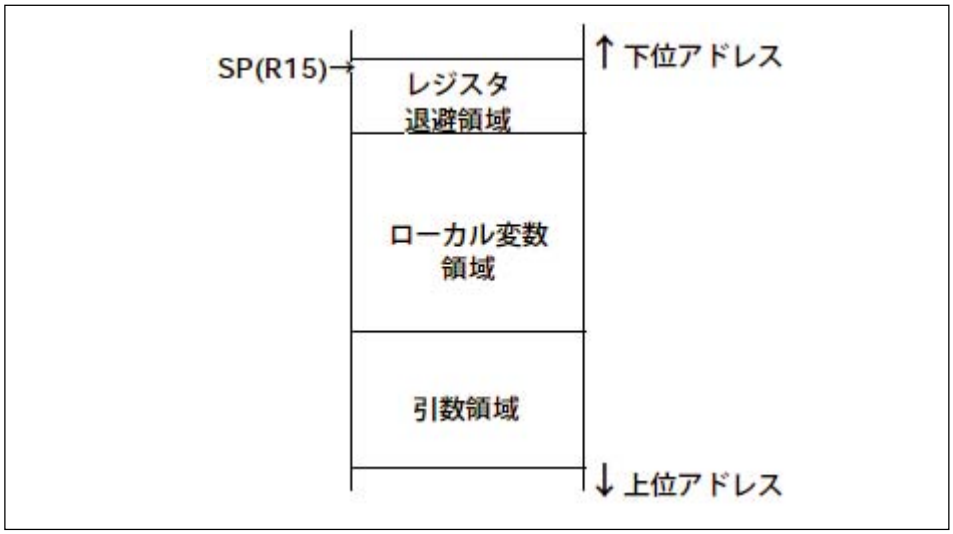


図 6.2 スタックフレームの構成

(0) 引数領域　「スタック渡し」となる場合に、呼び出される関数に渡す引数が設定される領域。
引数は、第１引数が下位アドレスになるように設定されます。

(1) ローカル変数領域　呼び出される関数で宣言されたローカル変数が設定される領域。

(2) レジスタ退避領域　呼び出される関数で使う汎用レジスタを退避（現在値を保存）する領域。

6.3 関数呼び出しとリターンの基本手順

C/C++ プログラムの関数の呼び出しおよびリターンは、基本的に以下の(1)～(8)の手順で行われます。なお、C/C++ コンパイラ起動時に -O オプションを指定した場合は、以下の構成と一致しない場合があります。

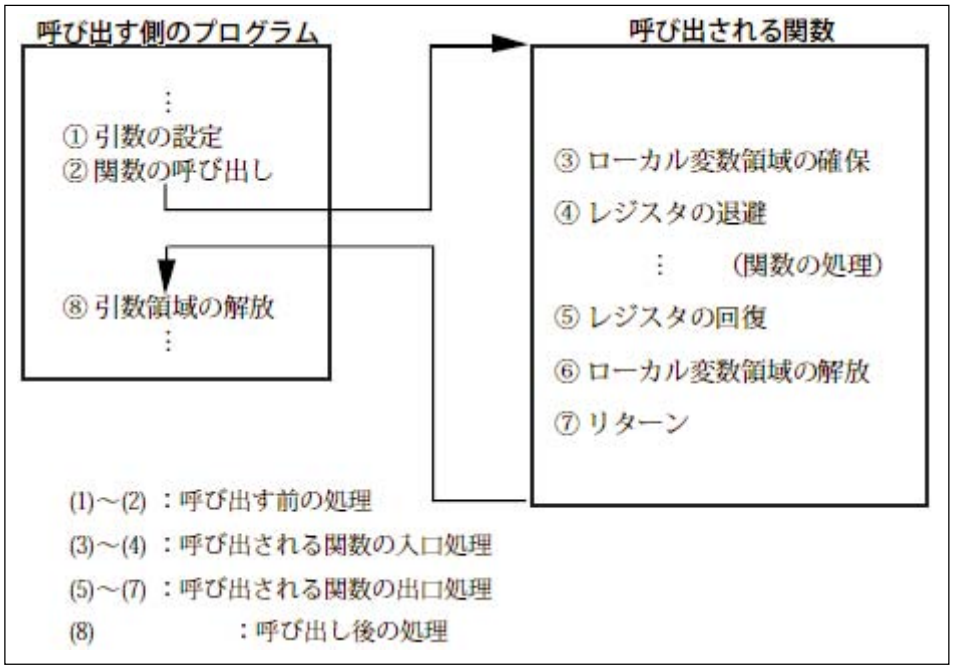


図 6.3 関数呼び出しとリターン（C/C++ プログラム）

(1)引数の設定

C/C++ コンパイラは、関数を呼び出す際の引数渡しの方法として、「レジスタ渡し」を原則としています。なお、引数の格納の仕方については、型変換やアライメントに関する規則があります（6.4 「引数の設定規則」参照)。

(1-1) 「レジスタ渡し」の場合

最初の 4 つの引数が順にレジスタ R4 ～ R7 に格納されます。関数の引数が 4 つに満たないときは、関数の引数の数だけのレジスタが使用されます（例えば、引数が 2 つなら R4 と R5 だけが使用されます)。また、関数の引数が 5 つ以上ある場合には 5 つ目以降の引数は、スタック渡しとなります。

(1-2) 「スタック渡し」の場合

(「6.4.2 スタック渡しとなる場合」に記載されている条件を満たす場合)スタック上に引数が積まれ、引数領域が設定されます。このとき、最終引数から順に積まれ、第 1 引数が引数領域の最下位アドレスに設定されます（図 6.4)。

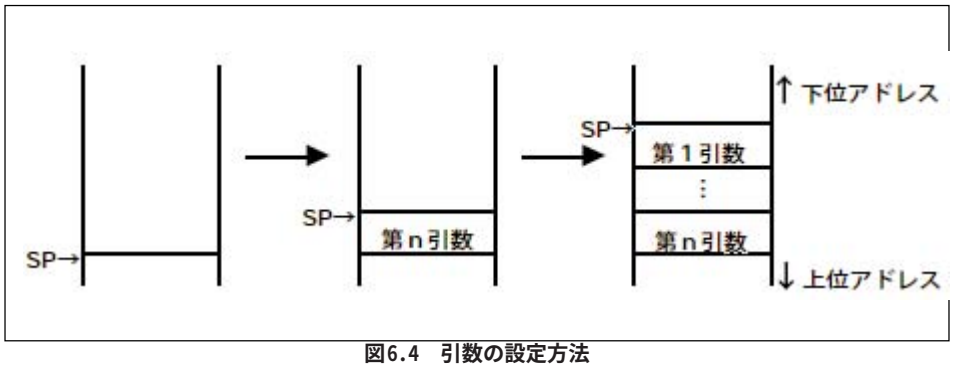


図6.4　引数の設定方法

構造体型または共用体型のリターン値を持つ関数を呼び出す場合は、第１引数が積まれた後、さらにリターン値を設定する領域の先頭アドレスが積まれます。リターン値を設定する領域は、呼び出す側で関数呼び出し前に確保されます（通常スタック領域中に確保されます）。構造体型または共用体型のリターン値を持つ関数の引数領域の構成を以下に示します（図6.5）。

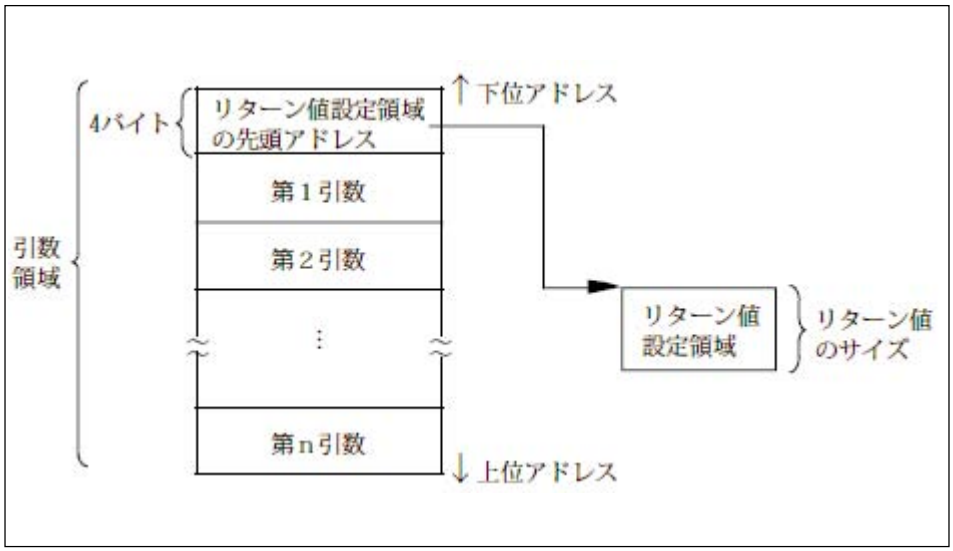


図６.５　構造体／共用体型の値を返す関数の引数領域

(2) 関数の呼び出し

BL または JL 命令によって関数が呼び出されます。このとき、関数呼び出しを行う命令の次の命令のアドレス（リターンアドレス）がリンクレジスタ R14 に設定されます。

(3) ローカル変数領域の確保

呼び出された関数内で使用するローカル変数領域がスタック上に確保されます。すなわち、確保される領域のサイズを SP 値から減算します。確保される領域のサイズは必ず 4 の倍数となります。

(4) レジスタの退避

R8～R13のうち、呼び出された関数内で使用するものをスタック上に退避します。退避領域のサイズは、「退避するレジスタ数 ×４バイト」です。リンクレジスタR14の退避が必要な場合はリンクレジスタも退避されます。

(5) レジスタの回復

呼び出された関数内で、関数の入口で退避したレジスタ（「(4)レジスタの退避」参照）を回復させます。

(6) ローカル変数領域の解放

呼び出された関数内で、関数の入口で確保したローカル変数領域を解放します（「(3)ローカル変数領域の確保」参照）すなわち、SPにローカル変数領域のサイズを加算します。

(7) リターン

リンクレジスタに格納されているリターンアドレスがPCに再設定されることによって、呼び出す側のプログラムに処理が復帰します。

(8) 引数領域の解放

関数を呼び出した側に制御が戻ると、SPにパラメータ領域のサイズを加えることによってパラメータ領域が解放されます。

6.4 引数の設定規則

C/C++ コンパイラは、関数を呼び出す際の引数渡しの方法として、「レジスタ渡し」を原則としています（関数引数が５つ以上ある場合、関数引数に浮動小数点型(double, long double)、構造体型、または共用体型等を使用した場合、「スタック渡し」となります)。以下に、引数の設定方法を示します。

6.4.1 引数の設定規則

関数引数は、R4 ～ R7 のレジスタを使用します。各引数は、必ず４の倍数バイトを単位としてレジスタに格納されます。関数宣言で最初の４つの引数が順に R4 ～ R7 に格納されます。関数の引数が４つに満たないときは、関数の引数の数だけのレジスタが使用されます（例えば、引数が２つなら R4 と R5 だけが使用されます)。また、関数の引数が５つ以上ある場合には５つ目以降の引数は、スタック渡しとなります。なお、関数呼び出しの前後での R0 ～ R7 の値は保証されません。

レジスタ渡しで渡せる引数は、文字型、整数型、ポインタ型、float 型です。

例： int func(int a, int b, int c);

この関数呼び出しを実行すると、引数は

R4	a
R5	b
R6	c

の順にレジスタに格納されます。

■ 型変換の規則

関数のプロトタイプ宣言によって引数の型が宣言されている場合、引数はその型に変換されます。プロトタイプ宣言で引数の型が宣言されていない場合は、以下の規則に従います。

・char、unsigned char、short、unsigned short 型の引数は int に変換します。
・float型の引数はdouble型に変換します。[注]
・その他の型の引数は変換しません。

■レジスタへの格納

引数は、上記の型変換後の型によって、以下の規則に従ってレジスタに格納されます。

・ 型変換後の型が char、unsigned char、short、unsigned short の場合、int に変換してから格納します。

注） float型の引数は、関数のプロトタイプ宣言によって型が宣言されていない場合、double型に変換されて、スタック渡しとなります。
float 型をレジスタ渡しにする場合には、必ず関数のプロトタイプ宣言で引数の型を宣言して下さい。

6.4.2 スタック渡しとなる場合

関数引数の数やリターン値や引数の型によっては、一部または全部の引数がスタック渡しとなる場合があります。以下にその規則を示します。

- 関数引数が５つ以上ある場合、５つ目以降の引数は、スタック渡しとなります。
- 最初の4つの引数の中に、浮動小数点型（double，long double）、構造体型、または共用体型のいずれかの引数が混在している場合、その引数の１つ前の引数までがレジスタ渡しとなります。その引数とそれ以降の引数はすべてスタック渡しとなります。
- リターン値の型が、浮動小数点型（double，long double、構造体型、または共用体型のいずれかである関数の場合、全引数スタック渡しとなります。
- 可変引数を持つ関数(printf関数のように引数リストの最後に「...」が指定されている関数）の場合は、可変引数とその１つ前の引数はスタック渡しとなります。

6.4.2.1 スタックへの積み方

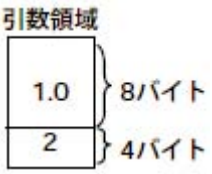
スタック渡しとなる場合、以下の規則に従ってスタック上に積まれます。

- 型変換後の型がchar、unsigned char、short、unsigned shortの場合、intに変換してから積みます。それら以外のスカラ型の場合、変換せずに積みます。
- 型変換後の型が構造体型または共用体型の場合、その型の境界整合が４バイトならばそのまま積みます。そうでない場合、その型のサイズを４バイトの倍数に切り上げたサイズの領域をスタック上にとり、その領域の最下位アドレス（４の倍数）から始まる領域に引数を設定します。

例１～例５にスタック渡しの例を示します（図はスタック上の引数領域のイメージ）。

- **例1**

```
int f();
      .
      .
f(1.0f, 2);
```

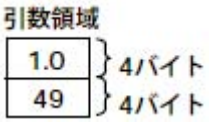


第１引数は floatですが、プロトタイプ宣言がないためdoubleに変換されてからスタックに積まれます。

■ 例2

```
int f(float, char);

    f(1, 49);
```



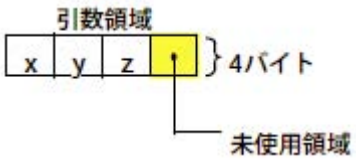
第１引数は float の 1.0 に変換されます。第２引数はプロトタイプ宣言に従って char に変換され、スタックに積まれる際、再び int に変換されます。

■ 例3

```
struct s{
   char x,y,z;
}a;

int f();

    f(a);
```

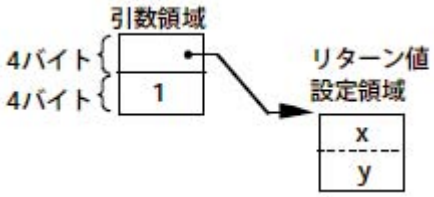


引数のサイズは３バイト、境界整合は１バイトですが、実際にスタックに積まれる際には最後に１バイトの未使用領域を補って４バイトとします。

■ 例4

```
struct s{
   int x, y;
}f();

 f(1);
```



リターン値が構造体の場合、スタックの引数領域の最下位アドレスに、リターン値設定領域（呼び出し側で確保）のアドレスが設定されます。

■ 例5

```
int f(float, ...);

    f(1.0, 1.0f);
```

第１引数はプロトタイプ宣言で float と指定されているため、float に変換されます。第２引数は対応する引数の型宣言がないので、double に変換されます。

6.4.3 コンパイル後の関数名

オブジェクトモジュール内での関数名は、C/C++ プログラムにおける関数名の前にドルマーク（$）をつけた名前となります。ただし、浮動小数点型（double, long double)、構造体、共用体のいずれかをリターン値として返す関数、および、可変引数を持つ関数については例外で、それぞれ以下のようになります。

■ 浮動小数点型（double, long double)、構造体、共用体のいずれかをリターン値として返す関数

・ 関数名の前にアンダーライン（ _ ）をつけた名前となります。

■ 可変引数を持つ関数

実際にレジスタによって渡された引数の数をｎとして、

・ n=0 の場合

C/C++ 言語プログラムにおける関数名の前にアンダーライン
（ _ ）をつけた名前となります。
例： `int foo(char *, ...);` は、
`_foo` という関数名になります。

・ n=4 の場合

C/C++ 言語プログラムにおける関数名の前にドルマーク（$）をつけた名前となります。
例： `int foo(int, int, int, int, int, ...);` は、
`$foo` という関数名になります。

・ n= １以上３以下の場合

C/C++ 言語プログラムにおける関数名の前にドルマーク（$）とレジスタによって渡される引数の数をつけた名前となります。
例： `int foo(int, int, ...);` は、
`$1foo` という関数名になります。

6.4.4 設定した引数を参照するには

呼び出された関数内で引数を参照するには、引数が設定されているレジスタを参照します。なお、スタック渡しで設定される引数（構造体など）については、スタックフレームを参照します。

6.5 リターン値の設定規則

リターン値（関数値）は、まず関数の返す型に変換されてから設定されます。設定規則はリターン値のデータ型によって以下のようになっています。

■ 整数型、ポインタ型　　呼び出される側で、リターン値を R0 に設定します。リターン後、呼び出し側ではリターン値として R0 を参照します。

■ 浮動小数点数型　　構造体型、共用体型の場合と同じ。

■ 構造体型、共用体型　　呼び出す側が用意したリターン値設定領域のアドレスは、呼び出される側で引数領域の先頭に格納されます（図6.5、図6.6）。このアドレスを使ってリターン値をリターン設定領域に設定します。リターン後、呼び出し側ではリターン値としてリターン値設定領域を参照します。

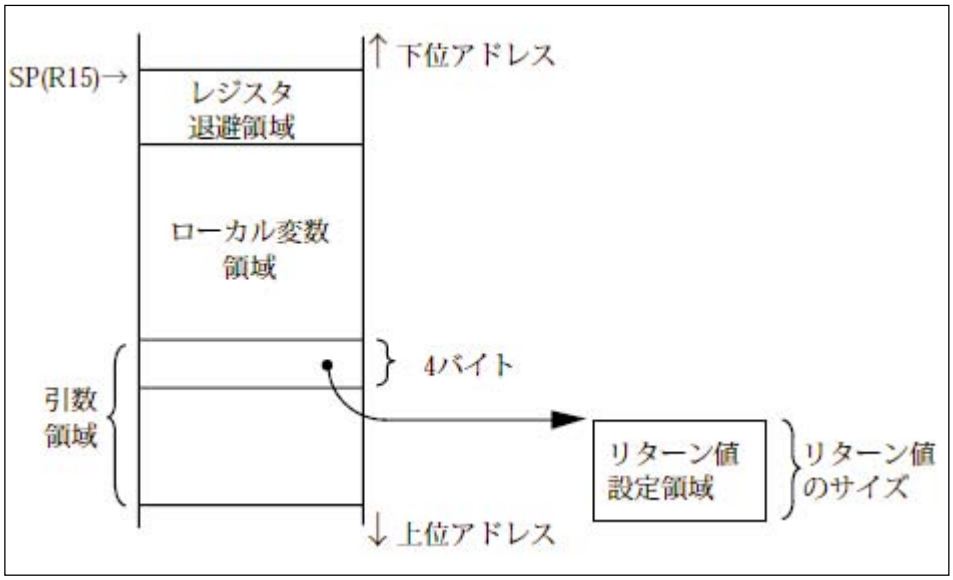


図 6.6　構造体型、共用体型のリターン値設定

6.6 アセンブリプログラムとのインタフェース

複数のプログラム（モジュール）を結合して１つのアプリケーションを開発する場合、各プログラムが相互に別プログラムのデータを参照したり関数を呼び出したりすることがあります。ここでは、C/C++ プログラムからアセンブリプログラムを参照する方法、および、アセンブリプログラムから C/C++ プログラムを参照する方法について、例を使って説明します。

C/C++ プログラム間における関数呼び出しについては「6.3 関数呼び出しとリターンの基本手順」、データ参照については「6.4 引数の設定規則」をそれぞれ参考にしてください。

アセンブリプログラムにおいて、C/C++ プログラムのデータを参照する場合、または、C/C++ プログラムの関数を呼び出す場合、C/C++ 呼び出し規則に従って記述する必要があります。

6.6.1 C/C++プログラムから アセンブリプログラムのデータを参照するには

C/C++ プログラムにおいて、アセンブリプログラム中のデータを参照するには、以下の例のように各プログラムを記述します。

C/C++ プログラム

```
extern int i, j;

void func()
{
    i = j;
}
```

アセンブリプログラム

```
      .EXPORT  _i, _j
      .SECTION D,DATA,ALIGN=4
_i:   .DATA.W  1
_j:   .DATA.W  1
      .END
```

図 6.7 アセンブリプログラムのデータ参照例

■ アセンブリプログラム側

C/C++ プログラムから参照したいデータは、擬似命令 .EXPORT または .GLOBAL で宣言して外部参照を可能にしておきます。図 6.7 の場合、ラベル _i と _j を外部から参照できるように、.EXPORT で宣言しています。

■ C/C++ プログラム側

参照したいアセンブリプログラムのデータを extern 宣言します。
extern 宣言するときは、アセンブリプログラムの対応するラベルの先頭から__（アンダースコア）を除いた名前で宣言します。

6.6.2 アセンブリプログラムから C/C++プログラムのデータを参照するには

アセンブリプログラムにおいて、C/C++ プログラム中のデータを参照するには、以下の例のようにアセンブリプログラムを記述します。

C/C++ プログラム

```
char a ,b;
```

アセンブリプログラム

```
.GLOBAL  _a
.GLOBAL  _b
.SECTION P,CODE,ALIGN=4
LD24 R1,#_a
LDB  R1,@R1
LD24 R0,#_b
STB  R1,@R0
JMP  R14
.END
```

図 6.8 C/C++ プログラムのデータ参照例

■アセンブリプログラム側

C/C++ プログラム中で宣言した char 型変数 a、b をアセンブリプログラムで参照するには、ラベル _a、_b を擬似命令 .IMPORT または .GLOBAL で指定します。

C/C++ コンパイラ cc32R では、C/C++ プログラムの外部参照／定義シンボルをオブジェクトモジュールに出力する場合、そのシンボル名の先頭に__（アンダースコア）を付加します。したがって図 6.8 の場合、C/C++ プログラム中の変数 a、b は、それぞれアセンブリプログラム中のラベル _a、_b に対応します。

6.6.3 アセンブリプログラムの関数を C/C++プログラムから呼び出すには

C/C++ プログラムにおいて、アセンブリプログラム中の関数を呼び出すには、以下の例のように各プログラムを記述します。

呼び出す側（C プログラム）

```
extern int func(int,int);

void foo()
{
      func(1,2);
}
```

呼び出す側（C++ プログラム）

```
extern "C" {
extern int func(int,int);
}
void foo()
{
      func(1,2);
}
```

呼び出される側（アセンブリプログラム）

```
.EXPORT    $func
           .SECTION P,CODE,ALIGN=4
$func:
           MV      R0,R5    ; 第 2 引数の内容を R0 に転送
           ADD     R0,R4    ; 第 2 引数 + 第 1 引数の結果を R0 に設定
           JMP     R14
           .END
```

図 6.9 アセンブリプログラムの関数を呼び出す例

■ C++ プログラムからアセンブリプログラムを呼び出す場合

呼び出したい関数を「extern "C"」を用いてプロトタイプ宣言してください。

■ 引数渡しを行う場合

関数名に対応するアセンブリプログラムのシンボル名は$（ドルマーク）で始まります。

・ 呼び出す側（C/C++ プログラム）
引数はレジスタに設定されます。

・ 呼び出される側（アセンブリプログラム）
図 6.9 の場合、第 1 引数は R4、第 2 引数は R5 で参照します。戻り値は R0 へ設定します。

6.6.4 アセンブリプログラムから C/C++プログラムの関数を呼び出すには

アセンブリプログラムにおいて、C/C++ プログラム中の関数を呼び出すには、以下の例のようにアセンブリプログラムを記述します。

呼び出す側（アセンブリプログラム）

```
.GLOBAL    $func
           .SECTION    P,CODE,ALIGN=4
           LDI  R5,#2              ;y
           LDI  R4,#1              ;x
           BL   $func
           .END
```

呼び出される側（C プログラム）

```
int func(int x, int y)
{

}
```

呼び出される側（C++ プログラム）

```
extern “C” {
     int func(int x, int y)
{

}
}
```

図 6.10 C/C++ プログラムの関数を呼び出す例

■ C++ プログラムを呼び出す場合

呼び出される関数を「extern "C"」を用いて定義してください。

■ 引数渡しを行う場合

関数名に対応するアセンブリプログラムのシンボル名は＄（ドルマーク）で始まります。

呼び出す側（アセンブリプログラム）では、引数をレジスタ（R4 ～ R7）に設定します。図 6.10 の場合、引数は 2 つなので R4 と R5 を使用し、第 1 引数を R4、第 2 引数を R5 に設定します。

6.7 CプログラムとC++プログラムとのインタフェース

複数のプログラム（モジュール）を結合して１つのアプリケーションを開発する場合、各プログラムが相互に別プログラムのデータを参照したり関数を呼び出したりすることがあります。ここでは、C プログラムから C++ プログラムを参照する方法、および、C++ から C プログラムを参照する方法について、例を使って説明します。

6.7.1 extern "C"と外部シンボル名について

関数名、静的データメンバから生成する外部シンボル名は、C++ 言語のコンパイルのときには、C 言語とは異なる規則で変換を行います。コンパイラが生成した外部シンボル名を知るには、次の２つの方法があります。

(1)コンパイル時にアセンブリ言語ファイルを生成するオプション(-S,-CS,-cs)を指定し、出力された外部シンボル名を確認する。

(2)生成されたオブジェクトモジュールまたはロードモジュールから、map32R にオプション -mangle を付けてマップ情報を生成する。

C++ の関数を 「extern "C"」 を付けて関数定義を行えば、外部シンボル名は C 言語の関数と同様の生成規則になります。

ただし、その関数は、多重定義できなくなります。

6.7.2 Cプログラムから C++プログラムのデータを参照するには

C プログラムにおいて、C++ プログラム中のデータを参照する場合、C++ プログラム上では、C プログラムと同一の型を用いて定義してください。

C プログラム

```
extern struct { int a; int b; } s;
extern int i,j;

void func(void)
{
    s.a = s.b;
    i = j;
}
```

C++ プログラム

```
int i,j;
struct { int a; int b; } s;
```

図 6.11 C プログラムから C++ プログラムのデータ参照例

||||| 注意 |||||

C++ 特有の型(class など)のデータは C プログラムでは使えません。

6.7.3 C++プログラムから Cプログラムのデータを参照するには

C++ プログラムにおいて、C プログラム中のデータを参照する場合、C プログラムで定義したのと同じ型と名前で extern 宣言してください。

C プログラム

```
int i,j;
}s;
struct { int a; int b; } s;
```

C++ プログラム

```
extern struct { int a; int b;

extern int i,j;

void func(void){
{
   s.a = s.b;
   i = j;
}
```

図 6.12 C++ プログラムから C プログラムのデータ参照例

||||| 注意 |||||

C++ 特有の型(class など)のデータは C プログラムでは使えません。

6.7.4 C++プログラムから Cプログラムの関数を呼び出すには

C++ プログラムにおいて、C プログラム中の関数を呼び出すには、以下の例のように「extern "C"」を用いて呼び出したい関数をプロトタイプ宣言します。

呼び出す側（C++ プログラム）

```
extern "C" int func(int,int);

void foo()
{
    func(1,2);
}
```

呼び出される側（C プログラム）

```
int func(int x, int y)
{
    return x + y;
}
```

図 6.13 C++ プログラムから C プログラムの関数を呼び出す例

6.7.5 Cプログラムから C++プログラムの関数を呼び出すには

C プログラムにおいて、C++ プログラム中の関数を呼び出すには、以下の例のように C++ プログラム側で「extern "C"」を用いて関数定義またはプロトタイプ宣言してください。

呼び出す側（C プログラム）

```
extern int func(int, int);

void foo(void)
{
    func(1,2);
}
```

呼び出される側（C++ プログラム）

```
extern "C" int func(int x,int y)
{
    return x + y;
}
```

図 6.14 C++ プログラムから C プログラムの関数を呼び出す例

第７章
組み込み用アプリケーションの作成

本章では、M32R システム組み込み用アプリケーションプログラム（以下、組み込み用アプリケーション）のプログラミング時に必要な情報を提供します。

7.1 C/C++コンパイラによるセクション構成

7.1.1 セクション構成について

C/C++ コンパイラによって C/C++ ソースファイルからオブジェクトモジュールファイルを生成すると、プログラムに含まれるすべてのコードやデータが、自動的に以下のセクション（領域）のいずれかに定義されます。

- ■ P セクション　　プログラムコードの領域
- ■ C セクション　　定数データ領域（const 宣言された変数領域）
- ■ D セクション　　初期値ありデータ領域（初期値を持つ大域変数領域）
- ■ B セクション　　初期値なしデータ領域（初期値を持たない大域変数領域）

C++ ソースファイルの場合は、さらに次の 3 つのセクションが生成されます。

- ■ CTOR セクション　　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とデストラクタのアドレスを格納する領域
- ■ VTBL セクション　　クラス宣言中に仮想関数があるときに仮想関数を呼び出すためのデータを格納する領域
- ■ COMMON セクション　　RTTI の情報やインライン関数内の static 変数を格納する領域

次ページ →

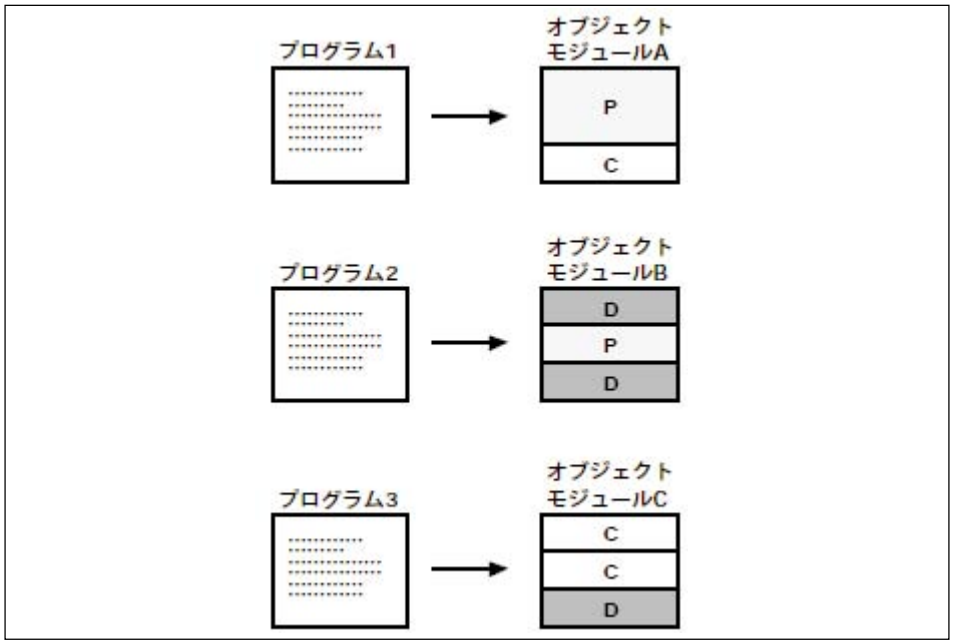


図 7.1　C/C++ コンパイラはプログラムの内容をセクション定義する

セクションとは、リンカにおけるプログラムの処理単位となるものです。各セクションには以下のように、セクション名、セクション属性、および配置属性がそれぞれ定義されています[注]。

表 7.1　C/C++ コンパイラが出力するセクション

セクション名	セクション属性	配置属性	概要
P	CODE	ALIGN=4	プログラムコードの領域
C	DATA	ALIGN=4	定数データ（const宣言された変数）の領域
D	DATA	ALIGN=4	初期値ありデータ領域（初期値を持つ大域変数領域。const宣言された変数を除く）
B	DATA	ALIGN=4	初期値なしデータ領域（初期値を持たない大域変数領域）
CTOR	DATA	ALIGN=4	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とデストラクタのアドレスを格納する領域
VTBL	DATA	ALIGN=4	クラス宣言中に仮想関数があるときに仮想関数を呼び出すためのデータを格納する領域
COMMON	DATA	ALIGN=4	RTTIの情報やインライン関数内のstatic変数を格納する領域

注） セクション名、セクション属性、および配置属性は、アセンブラ疑似命令 .SECTION によって定義でき、表 7.1 で示す以外の設定も可能です。命令の記述規則は「CC32R ユーザーズマニュアル＜アセンブラ編＞」で、セクション名や各属性の種類や意味については「CC32R ユーザーズマニュアル＜アセンブラ編＞」のリンカの章で、それぞれ説明しています。

リンカは、これらの定義がすべて一致するセクションは同一セクションとみなします。同一セクションは結合（リンク）され[注]、ロードモジュール内で連続した配置となります。例えば、起動オプション -SEC D,P,C と指定した場合、図7.2のようになります。

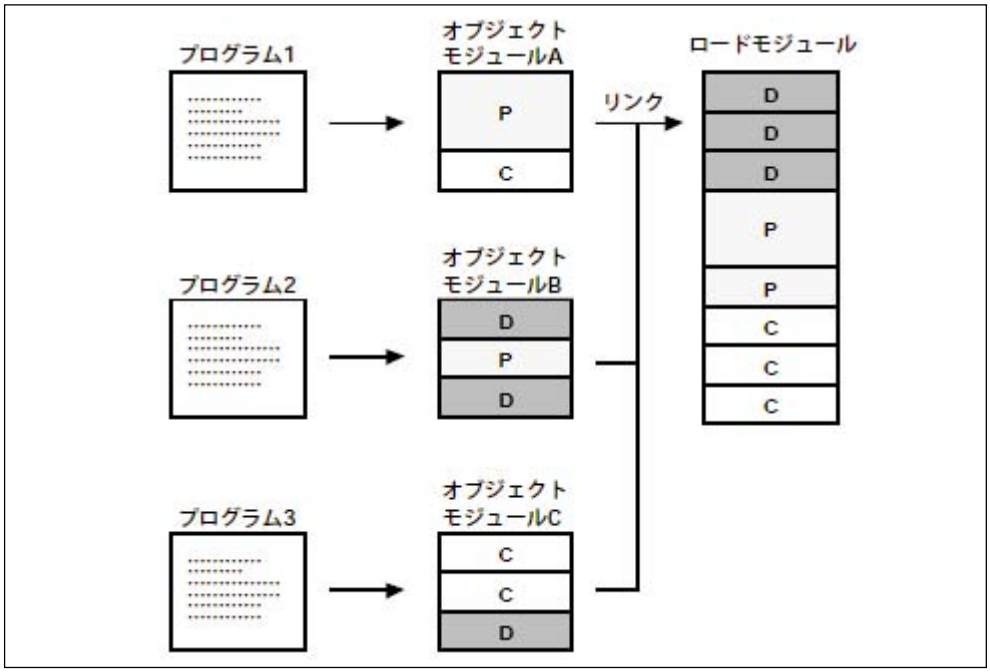


図7.2　セクション単位でリンクされる（イメージ）

C/C++ コンパイラでは、起動オプション -SEC あるいは -MEM によって、セクションの結合順序や各セクションの開始アドレスが指定できます。C/C++ コンパイラはその指定に従って各セクションのアドレスを決定します。

こうした機能は、組み込み用アプリケーションを作成する場合、すなわち、プログラムを ROM 化したい場合に有効です（詳しくは「M3T-CC32R ユーザーズマニュアル＜アセンブラ編＞」の「第２部　リンカ　lnk32R」を参照）。

||||| 注意 |||||

-SEC オプションと -MEM オプションを同時に指定することはできません。
-MEM オプションは CTOR,VTBL,COMMON の各セクションの制御はサポートしていません。

次ページ →

注）結合方法はセクション属性によって、結合順序はオプション指定内容によって異なります。

実際の結合順序や各セクションの開始アドレスは、以下の規則によって決まります。オプションの機能や指定方法については第３章「C/C++ コンパイラの起動方法」を参照してください。

■ セクション結合順序

結合順序の指定があればそれが優先されます。指定のないセクションについては、指定のあるセクションを結合した後、それに続いて結合されます。

① -SECオプション指定時

-SECオプションでの結合順序指定（記述順）に応じた順序。

-MEM（リンカの場合は-LOC）オプション指定時

・RAM領域　D→B

・ROM領域　P→C→D

② 指定がない場合（デフォルト）

コマンド行でのファイル指定順（記述順）、および、
1ファイル内に複数セクションがある場合はセクションの出現順（記述順）による順序。

■ 開始アドレス

アドレス指定があればそれが優先されます。指定のないセクションについては、結合形式に応じて相対的に決まります。アライメントはデフォルトで４バイトです（アセンブラ擬似命令 .SECTION によって任意指定も可能）。ロードモジュール全体の先頭アドレスは、指定がなければ０番地となります。

① -SECオプション指定時

各セクションに対して指定されたアドレス。
指定のないセクションの開始アドレスは、結合順序によって自動的に決定。

-MEM（リンカの場合は-LOC）オプション指定時

・ Dセクション　RAM領域に対して指定されたアドレス。
・ Pセクション　ROM領域に対して指定されたアドレス。
・ B、C、Dセクション　前方（下位アドレス方向）に結合されているセクションの最終アドレスの後続アドレス。

② 指定がない場合（デフォルト）

・ 各セクション　前方（下位アドレス方向）に結合されているセクションの最終アドレスの後続アドレス。
・ ロードモジュール全体　0番地

7.1.2 CTOR セクション記述

C++ フロントエンドは、グローバルコンストラクタへの関数ポインタを CTOR セクションに配置します。そのため、スタートアッププログラムにグローバルコンストラクタへの関数ポインタを格納する CTOR セクションを記述する必要があります。

スタートアップファイル中に以下の記述を追加します。

```
.section CTOR,data,align=4
; ここにグローバルコンストラクタへの関数ポインタが格納される
```

CC32R では、セクションの先頭アドレスとセクションの最後のアドレス+1 へのラベルが生成されます。

__TOP_CTOR (セクションの先頭アドレス)、
__END_CTOR （ セクションの最後のアドレス+1）

7.1.3 VTBLセクション記述

C++ フロントエンドは、仮想関数を持つクラスに対して仮想関数テーブルを生成し VTBL セクションに配置します。そのため、スタートアッププログラムに仮想関数テーブルを格納する VTBL セクションを記述する必要があります。

スタートアップファイル中に以下の記述を追加します。

```
.section VTBL,data,align=4
; ここに仮想関数テーブルが格納される
```

7.1.4 COMMONセクションについて

インライン関数内の static 変数や、実行時型情報機能を使用した場合に、COMMON セクションにデータが出力されます。
COMMON セクションはリンカ（lnk32R）が必ず生成しますので、コンパイル時に "-SEC" オプションで COMMON セクションを必ず指定してください。

- COMMON セクションは、ＲＡＭに配置されます。
- 同名のセクションを設けるとリンク時にエラーになります。
- COMMON 属性と混同しないでください。

7.2 組み込み用アプリケーションの作成手順

組み込み用アプリケーションを作成する場合、機能を実現するユーザープログラム以外に、スタートアッププログラムや低水準ライブラリの作成が必要です。リンク時には、組み込み用としての適切な配置指定、エントリポイント指定、および、ROM 化するためのアドレス指定等が必要です。さらに、ロードモジュールファイルは ROM 化するためにＳフォーマットファイルに変換する必要があります。こうした作業を含めた、組み込み用アプリケーション作成の基本手順（①～⑥）を以下に示します。

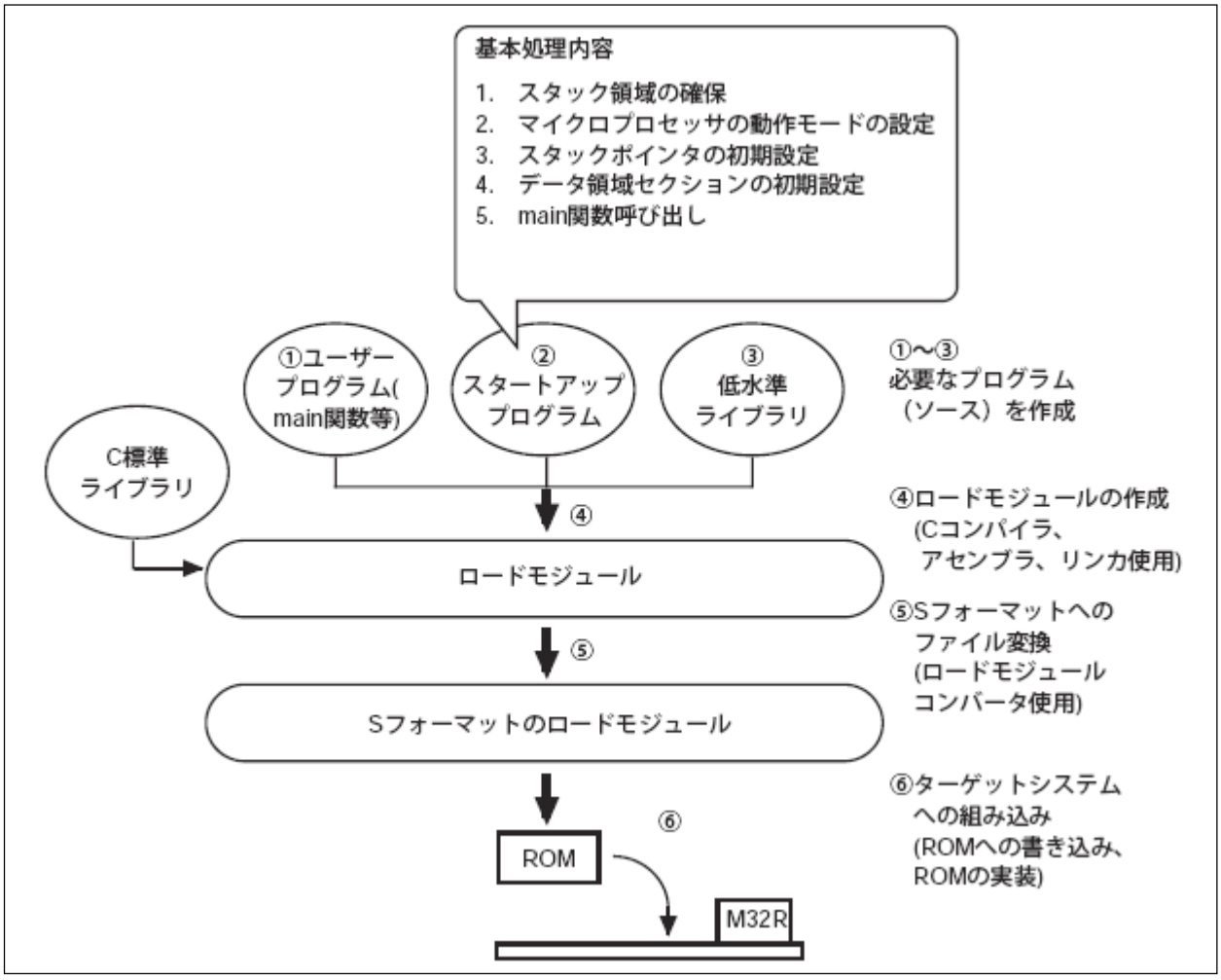


図 7.3　組み込み用アプリケーション作成の基本手順

① ユーザープログラムの作成

main 関数ほか、必要な処理を行うプログラムを作成します。C/C++ 言語で記述する場合は C/C++ 言語仕様（第 4 章「C/C++ 言語仕様」参照）に従います。アセンブリ言語で記述する場合は M32R 用のアセンブリ言語仕様（「CC32R ユーザーズマニュアル＜アセンブラ編＞」参照）に従います。処理内容によっては、C 言語ソースプログラム中にアセンブリ命令を記述することもできます（インラインアセンブル機能）。

② スタートアッププログラムの作成

ターゲットシステムに応じたスタートアッププログラムを作成します。スタートアッププログラムとは、アプリケーションがターゲットシステム上で動作するための初期設定を行うプログラムです。

通常スタートアッププログラムによって行う処理を以下に示します。

1. スタック領域の確保
2. マイクロプロセッサの動作モードの設定
3. スタックポインタの初期設定
4. データ領域セクションの初期設定
5. _cpp_main 関数の呼び出し
6. グローバルコンストラクタの呼び出し
   (ランタイムライブラリの _ctor を呼び出す)
7. _call_main 関数の呼び出し
8. グローバルデストラクタの呼び出し
   (ランタイムライブラリの _dtor 関数を呼び出す)
9. main 関数の呼び出し

スタートアッププログラムの作成方法については7.3「スタートアッププログラムの作成」を参照してください。

③ 低水準ライブラリの作成

アプリケーション中でＣ標準ライブラリを使用して以下のような処理を実現している場合、低水準ライブラリを用意する必要があります。

- ■ 標準入出力
- ■ メモリ管理
- ■ シグナル操作
- ■ 時間操作

実際にどのＣ標準ライブラリ関数がどの低水準ライブラリを必要としているかという情報や、低水準ライブラリの仕様については第11章「低水準ライブラリ」で説明しています。そこで示す仕様に従って作成し、作成したプログラムは、ライブラリ化しておきます。

④ ロードモジュールの作成

cc32Rコマンドによって、作成したユーザープログラム、スタートアッププログラムからロードモジュールを作成します。

リンク時には、コマンド行で指定した順にファイルが参照され、その順にセクション結合が行われます。したがって、コマンド行に入力ファイルを記述する際は必ずスタートアッププログラムを一番最初に指定します。

また、以下のリンク指定を行います。

■ -l および -L オプション
必要な低水準ライブラリおよびC標準ライブラリをリンク指定。

■ -e オプション
エントリポイントとしてスタートアッププログラムの開始アドレスを指定。

■ -SEC（または -MEM）オプション
各セクションの配置順序、および、配置アドレスを指定。組み込み用アプリケーションの場合、一般に、D、Bセクションを RAM 領域に、Dセクションの初期化用データと P、Cセクションを ROM 領域に配置します。Dセクションを初期化するためのデータを ROM 領域に配置するには、リンカの「初期値データ抽出機能」を使用します（セクション名は ROM_D となります）。

cc32R コマンドを実行した結果コンパイルエラーがあれば、ソースファイルの誤りを修正します。コンパイルエラーがすべて取り除かれたオブジェクトモジュールが生成できたら、自動的にリンカが起動し、起動オプションによるリンク指定に従って、ロードモジュールが生成されます。

以下に、cc32R コマンドのコマンド行の指定例を示します（% はプロンプト）。

例：
```
% cc32R  -l m32RcR.lib  -e startup
-SEC @D=1000,B,SPI,SPU,P=8000,C,D
start.ms cpp_main.cpp call_main.c user1.c user2.c
```

この例では、各入力ファイルを以下のように想定しています。

■ start.ms cpp_main.cpp call_main.c
ターゲットシステムに対応した初期設定処理を行うスタートアッププログラム（7.3.9「スタートアッププログラムサンプル」参照）。

■ user1.c user2.c　　main 関数を含むユーザープログラム。

上記例のコマンド行を実行すると、以下の指定が反映されたロードモジュール am.out が生成されます。

■ -l m32RcR.lib
C標準ライブラリ m32RcR.lib をリンクします。

■ -e startup
エントリポイントをスタートアッププログラムの開始アドレスとします（スタートアッププログラム内に、開始アドレスをシンボル

startup で定義した場合)。これによって、アプリケーションはスタートアッププログラムから実行されます。

■ -SEC @D=1000,B,SPI,SP0,P=8000,C,D

セクションの結合順序を、D→B→SPI→SPU→P→C→ROM_D とします（SPI、SPU セクションは、アセンブリプログラム内でユーザーがスタック用に用意したセクション。ROM_D セクションは、D セクションを初期化するためのデータ領域)。また、各セクションを以下のように配置します。

- D　　領域のみを 1000_{16} 番地から配置
- P　　8000_{16} 番地から配置
- B、SPI、SPU、C、ROM_D　　前方のセクションに続いて配置

■ start.ms、cpp_main.cpp、call_main.c、user1.c、user2.c

それぞれコンパイルまたはアセンブルを行った後、リンクします。

⑤ S フォーマットファイルへの変換

ロードモジュールを、ROM に書き込むためのモトローラ S フォーマットに変換します。ロードモジュールコンバータ lmc32R を用います。

例：

```
% lmc32R  -d 4 -o file am.out
```

上記例を実行した場合、ロードモジュール am.out は、モトローラ S フォーマットに変換され、file.m40、file.m41、file.m42、 および file.m43 の 4 ファイルに分割されます。ロードモジュールコンバータの使用方法は「CC32R ユーザーズマニュアル 3< アセンブラ編 >」のロードモジュールコンバータの章で説明しています。

⑥ ターゲットシステムへの組み込み

モトローラ S フォーマットに変換したロードモジュールを、ROM ライタを使用してプログラムを ROM に書き込みます。その ROM をターゲットシステムに実装します。

7.3 スタートアッププログラムの作成

7.3.1 スタートアッププログラムの構成

スタートアップファイルは、start.ms（アセンブリ言語）と cpp_main.cpp（C++ 言語）と call_main.c（C 言語）の 3 つから構成されています。

スタートアップの設定は、start.ms を書き変えることで更新することができます。一部、cpp_main.cpp で行っている設定もありますが、通常は start.ms のみの更新で対応できます。

7.3.2 スタートアッププログラムの基本処理

組み込み用アプリケーションをターゲットシステム上で動作させるには、ユーザープログラム（main 関数で始まる）の呼び出し前後に実行する「スタートアッププログラム」が必要です。スタートアッププログラムでは、基本的に、以下の①～⑤のような初期設定または終了処理を行います。

[startup.ms]

① スタック領域の確保
② マイクロプロセッサの動作モードの設定
③ スタックポインタの初期設定
④ データ領域セクションの初期設定
⑤ _cpp_main 関数呼び出し

[cpp_main.cpp]

⑥ グローバルコンストラクタを呼び出す(ランタイムライブラリの _ctor を呼び出す)
⑦ _call_main 関数を呼び出す
⑧ グローバルデストラクタを呼び出す(ランタイムライブラリの _dtor 関数を呼び出す)

[call_main.c]

⑨ main 関数を呼び出す

なお、実際には、ユーザープログラムの内容によって、これら以外にも追加すべき処理がある場合、または、この中から削除すべき処理がある場合があります。

以下より、各処理の詳細を説明し、7.3.10 節にスタートアッププログラムのサンプルを示します。

7.3.3 スタック領域の確保

アプリケーションの動作に必要なスタック領域を確保します。実際には、アセンブリプログラムにおいて、擬似命令 .SECTION でスタック領域セクションを宣言し、擬似命令 .RES でスタック領域を確保します。

C/C++ プログラムでは、関数を呼び出すたびに、スタック上にスタックフレームが形成されます（6.2 節参照）。スタックフレームは、呼び出された関数から呼び出したプログラムへ処理がリターンするまで保存されます。したがって、関数 A →関数 B →関数 C というように、呼び出し元の関数にリターンせずにさらに関数呼び出しが行われた場合、関数 C が呼び出された時点では、関数 A、B、C すべてのスタックフレームが保存されています。

アプリケーションの動作に必要なスタックサイズは、このような関数の呼び出し関係や、各関数が使用するスタックサイズを考慮して求めます。ただし、厳密に算出することは困難で、 データの内部表現（第 5 章参照）や C 呼び出し規則（第 6 章参照）などによって、およそのスタックサイズを予測して指定します。一般には、デバッグ工程や評価工程で実際にプログラムを実行してスタックの使用量を調べる方法（初期段階で十分なスタック領域を確保しておき、デバッガを用いてスタックの使用量を調べる、など）が用いられます。割り込み用のスタック領域も同様に確保します。

7.3.4 マイクロプロセッサの動作モードの設定

ターゲットのマイクロプロセッサに対して、使用スタックの指定や割り込みレベルの指定を行います（レジスタ PSW の設定）。設定するための情報（PSW の構成など）についてはマイクロプロセッサのマニュアルを参照してください。

7.3.5 スタックポインタの初期設定

確保したスタック領域の最上位アドレスをスタックポインタに設定します。

7.3.6 データ領域セクションの初期設定

組み込み用 アプリケーションでは、データ領域（D、B セクション）の初期設定が必要です。そのためには、アプリケーションのリンク時、および、スタートアッププログラム実行時に以下の作業が必要です。

■ 組み込み用アプリケーションのリンク時の作業

- D、B セクション領域を RAM 領域に配置する（データは出力しない）。
- D セクション用の初期値データ領域（ROM_D）を ROM 領域に配置する。

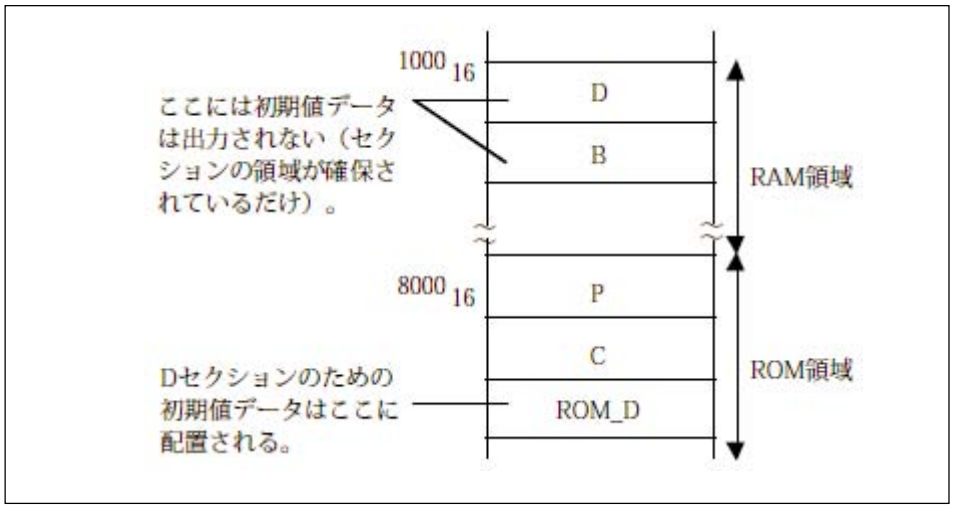


図７.４　データ領域セクションの配置

リンク時における、各セクションの配置指定やデータ抽出は、-SEC または -MEM オプションで指定できます。また、その結果生成されるロードモジュールには、自動的に各セクションの先頭アドレスおよび最終アドレスを表すラベルが生成されます。ラベル名のつき方は以下のとおりです。

表７.２　各セクションの予約ラベル名

位置	ラベル名	例（Dセクションの場合）	例（ROM_D セクションの場合）
セクションの先頭アドレス	_ _TOP_	_ _TOP_D	_ _TOP_ROM_D
セクションの最終アドレス	_ _END_	_ _END_D	_ _END_ROM_D

||||| 注意 |||||

組み込みアプリケーション（ROM 化するプログラム）のためのセクション配置やセクションの初期化について詳しくは、「CC32R ユーザーズマニュアル＜アセンブラ編＞」のリンカの章で説明しています。なお、リンク指定は、C/C++ コンパイラ起動時に、起動オプションによって指定できます。

■ スタートアッププログラムでの初期化処理時

- ROM 領域内の ROM_D セクションにあるデータを、RAM 領域内の D セクションの領域に転送することによって初期化する。
- RAM 領域内の B セクションの領域を、ゼロクリア（0 データで埋める）によって初期化する。

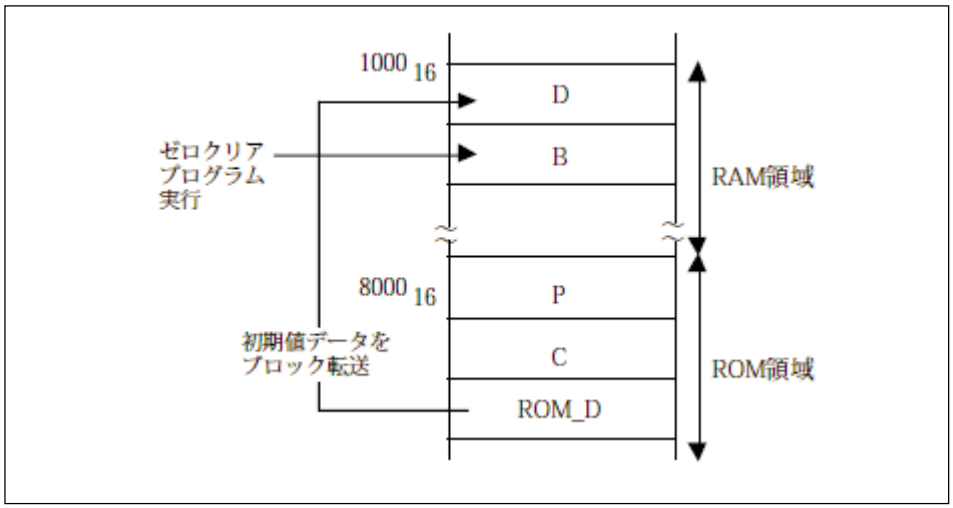


図７．５　データ領域の初期設定

上記処理をプログラミングする際、各セクションの開始アドレスを参照するためには、リンクによって生成される予約ラベル（表7.2参照）やセクション名を利用します。

7.3.7　_cpp_main 関数の呼び出し

C++言語の初期設定と標準ライブラリを使用するために、 $_cpp_main関数 を呼び出します。

7.3.8　_cpp_main 関数の処理

C++言語の初期設定と標準ライブラリ使用のための初期設定を行い、
_call_main 関数を呼び出す

7.3.9　_call_main 関数の処理

C++言語では、スタートアップ処理からユーザープログラムのmain関数を呼び出すことが出来ないため、_call_main 関数を介して、main関数を呼び出します。

7.3.10 スタートアッププログラムサンプル

以下にスタートアッププログラムの例を示します。

■ スタートアッププログラム　　start.ms

このサンプルプログラムでは基本処理①～⑤を行うプログラムを、STARTUP で始まるアセンブリプログラムとして記述しています。

```
;***********************************************************************
; COPYRIGHT(C) 2004 RENESAS TECHNOLOGY CORPORATION AND RENESAS SOLUTIONS
; CORPORATION ALL RIGHTS RESERVED
;
; start.ms : cc32R startup and low-level-sample program
;
;    [ Contents ]
;        (1) Sample startup routine.
;        (2) Stack memory area
;        (3) Sample low-level routine.
;        (4) Reset vector area
;
;    [ NOTES ]
;      * This startup sample never set addresses for all sections,
;        expect RI (Reset Vector Table) section, then you have to
;        set address by -SEC option by lnk32R (M32R linker) .
;
;***********************************************************************

;-----------------------------------------------------------------------
;  HEAPSIZE definition
;-----------------------------------------------------------------------
HEAPSIZE        .EQU            H'4000

;-----------------------------------------------------------------------
;  STACKSIZE definition
;-----------------------------------------------------------------------
USTACKSIZE      .EQU            H'1000
ISTACKSIZE      .EQU            H'1000

;-----------------------------------------------------------------------
;  (1) Startup routine
;-----------------------------------------------------------------------
                .export         STARTUP
                .export         HALT
                .import         $_cpp_main
                .import         __TOP_B, __END_B
                .import         __TOP_D, __END_D
                .import         __TOP_COMMON, __END_COMMON
                .import         __TOP_ROM_D,__END_ROM_D
                .section        P,code,align=4
                ;
                ; Initialize PSW control-register.
                ;
```

```
STARTUP:        LDI             R0, #128
                MVTC            R0, PSW
                ;
                ; Setting the user and interrupt stack.
                ;
                SETH            R0, #HIGH(USTACK_BOTTOM)
                OR3             R0, R0, #LOW(USTACK_BOTTOM)
                MVTC            R0, SPU
                SETH            R0, #HIGH(ISTACK_BOTTOM)
                OR3             R0, R0, #LOW(ISTACK_BOTTOM)
                MVTC            R0, SPI
                ;
                ; Clear the B section to zero.
                ;
                SETH            R0,#HIGH(__TOP_B)
                OR3             R0, R0, #LOW(__TOP_B)
                SETH            R1,#HIGH(__END_B)
                OR3             R1, R1, #LOW(__END_B)
                LDI             R2, #0
LOOP1:          CMP             R0, R1
                BNC             SKIP1
                STB             R2, @R0
                ADDI            R0, #1
                BRA             LOOP1
SKIP1:
                ;
                ; Clear the COMMON section to zero.
                ;
                SETH            R0,#HIGH(__TOP_COMMON)
                OR3             R0, R0, #LOW(__TOP_COMMON)
                SETH            R1,#HIGH(__END_COMMON)
                OR3             R1, R1, #LOW(__END_COMMON)
                LDI             R2, #0
LOOP2:          CMP             R0, R1
                BNC             SKIP2
                STB             R2, @R0
                ADDI            R0, #1
                BRA             LOOP2
SKIP2:
                ;
                ; Transfer the data in the ROM_D section in ROM area
                ; to the RAM area.
                ;
                SETH            R1, #HIGH(__TOP_ROM_D)
                OR3             R1, R1, #LOW(__TOP_ROM_D)
                SETH            R2, #HIGH(__TOP_D)
                OR3             R2, R2, #LOW(__TOP_D)
                SETH            R3, #HIGH(__END_ROM_D)
                OR3             R3, R3, #LOW(__END_ROM_D)
```

```
LOOP3:          CMP             R1, R3
                BNC             SKIP3
                LDB             R0, @R1
                STB             R0, @R2
                ADDI            R1, #1
                ADDI            R2, #1
                BRA             LOOP3
SKIP3:
                ;
                ; Jump to the C standard initialize routine.
                ;
                SETH            R0, #HIGH($_cpp_main)
                OR3             R0, R0, #LOW($_cpp_main)
                JL              R0
                ;
                ; End of program ( infinity loop )
                ;
HALT:           BRA             HALT

                .section        C,data,align=4
                .section        ROM_D,data,align=4
                .section        CTOR,data,align=4
                .section        VTBL,data,align=4

;-------------------------------------------------------------------------
;  (2) Sample low-level routine.
;-------------------------------------------------------------------------
                .export         $_exit
                .export         $_get_core
                .export         $write
                .export         $_rel_core
                .export         $_strerror
                .export         $close
                .export         $getuniqnum
                .export         $lseek
                .export         $open
                .export         $read
                .export         $getenv
                .export         $raise
                .export         $remove
                .section        P,code,align=4
                ;
                ; Terminate the program run.
                ;
$_exit:         BRA             HALT
                ;
                ; Get the heap memory routine.
                ;
```

```
$_get_core:     MV              R1, R4
                SETH            R2, #HIGH(HEAP_POINTER)
                OR3             R2, R2, #LOW(HEAP_POINTER)
                LD              R0, @R2
                ADD             R1, R0
                ST              R1, @R2
                JMP             R14
                ;
                ; ( Not implemented routine )
                ;
$write:
$_rel_core:
$_strerror:
$close:
$getuniqnum:
$lseek:
$open:
$read:
$getenv:
$raise:
$remove:        LDI             R0, #0
                JMP             R14

;-----------------------------------------------------------------------
;  (3) Heap and stack memory area
;-----------------------------------------------------------------------
                .section        D,data,align=4
HEAP_POINTER:   .DATA.W          HEAP_TOP

                .section        B,data,align=4
HEAP_TOP:
                .RES.B          HEAPSIZE

                .RES.B          USTACKSIZE
USTACK_BOTTOM:
                .RES.B          ISTACKSIZE
ISTACK_BOTTOM:

;-----------------------------------------------------------------------
;  (4) Reset vector area
;-----------------------------------------------------------------------
                .section        RI,code,locate=h'00000000
                SETH            R0, #HIGH(STARTUP)
                OR3             R0, R0, #LOW(STARTUP)
                JMP             R0

                .end            STARTUP

;***********************************************************************
; COPYRIGHT(C) 2004 RENESAS TECHNOLOGY CORPORATION AND RENESAS SOLUTIONS
; CORPORATION ALL RIGHTS RESERVED
;***********************************************************************
```

注） このスタートアッププログラムは、サンプルです。実際には、ご使用になられる環境に合わせた、スタートアッププログラムをご使用ください。

■ スタートアッププログラム　cpp_main.cpp

このサンプルプログラムでは基本処理⑥～⑧を行うプログラムを、C++ 言語プログラムとして記述しています。

```
#if defined(__M32R__)
/* COPYRIGHT(C) 1993 (2003) RENESAS TECHNOLOGY CORPORATION */
/* AND RENESAS SOLUTIONS CORPORATION */

#include <setjmp.h>

extern "C" {
    externjmp_buf _900_init_env;
    void_ctor( void );
    void_dtor( void );
    int_call_main( void );
    int_100_C_get_exit_code( void );
    void_100_C_action_atexit( void );
    void__call_terminate( void );
    void_exit( int );
    extern void(*_900_exit_stdio_func_ptr)(void);
    extern void(*_900_exit_mem_func_ptr)(void);
}

extern "C" void _cpp_main( void )
{
    int exit_code;

#ifdef __EXCEPTIONS
    try {
#endif // __EXCEPTIONS

        _ctor(); // call global constructor

        if ( ! setjmp( _900_init_env ) ) {
            exit_code = _call_main();
        } else {
            exit_code=_100_C_get_exit_code();
        }
        if ( ! setjmp( _900_init_env ) ) {
            _100_C_action_atexit();
        }

        _dtor(); // call global destructor
```

```
#ifdef __EXCEPTIONS
    } catch(...) {
        if ( ! setjmp( _900_init_env ) ) {
            __call_terminate();
        } else {
            exit_code = _100_C_get_exit_code();
        }
         }
#endif // EXCEPTIONS

    if (_900_exit_stdio_func_ptr)
        (*_900_exit_stdio_func_ptr)(); // terminating stdio library
    if (_900_exit_mem_func_ptr)
        (*_900_exit_mem_func_ptr)(); // terminate memory library

    _exit( exit_code );
}
#endif /* defined(__M32R__) */
```

■ スタートアッププログラム　　call_main.c

このサンプルプログラムでは基本処理⑨を行うプログラムを、C 言語プログラムとして記述しています。

```
extern int main();
extern int _call_main(void);

int _call_main(void)
{
    return main();
}
```

7.4 HEWにおけるスタートアップファイル start.ms について

HEW でプロジェクトを新規作成する際に作成する、start.ms は、TM およびユーザーズマニュアルで使用している start.ms を、HEW 用に変更したものです。
内容は基本的に同じですが、HEW からアセンブラ as32R の -D オプションに次のシンボルを指定することで、start.ms のパラメータを操作するようになっています。
start.ms を編集される場合は、この点にご注意ください。

表 7.3 HEW におけるスタートアップファイル start.ms のシンボルの意味

シンボル名	HEWのプロジェクト作成ダイアログでの項目名	意味	初期値
__STANDARD_IO__	Use Standard I/O Library	標準ライブラリの初期化を行ってからmain( ) を実行するかどうか	0 (標準ライブラリを初期化しない)
__HEAPSIZE__	Heap Size	ヒープサイズ	H'4000
__USTACKSIZE__	User Stack Pointer Stack Size	ユーザ用スタック(SPU)サイズ	H'1000
__ISTACKSIZE__	Interrupt Stack Pointer Stack Size	割り込み用スタック(SPI)サイズ	H'1000

7.5 インラインアセンブル機能

7.5.1 インラインアセンブル機能概要

組み込み用アプリケーションでは、 ハードウエアやオペレーティングシステム（OS）の制御など低レベルな部分の処理が必要な場合があります。また、機能上、高速で効率のよい処理が要求されます。それに対応すべく、C/C++ コンパイラでは C/C++ 言語ソースプログラム中にアセンブリ言語命令が記述できる「インラインアセンブル機能」をサポートしています。

実際にアセンブリ言語記述を C/C++ 言語ソースプログラム内に挿入するには、asm 関数という特別な関数を使用します。asm 関数は、C/C++ 言語ソースプログラム中にアセンブリ言語のコードを埋め込む機能を提供します。

||||| 注意 |||||

C/C++ コンパイラの最適化では、不要コードの削除、命令の置換、および、命令の入れ換えなどが強力に実施されます。したがって、インラインアセンブル機能は、C/C++ コンパイラの最適化の影響を十分に考慮したうえで使用してください。

7.5.2 asm関数の記述方法

asm 関数を使用するためには、asm 関数を呼び出す前に以下のように定義します。

```
#pragma keyword asm on
```

これは、「asm」を通常の識別子でなく、予約語として認識させるための設定です。再び asm を識別子として使用する場合は、次のように keyword asm off と指定してください。

```
#pragma keyword asm off
```

asm 関数は図 7.6 に示す書式で記述します。

```
#pragma keyword asm on
asm （ "アセンブリコード"[, パラメータ1] [, パラメータ2] );
```

アセンブリコード	埋め込みたいアセンブリ言語のコード
パラメータ1	レジスタ R0 にセットする式
パラメータ2	レジスタ R1 にセットする式

※パラメータには、整数型の定数あるいは識別子を記述できます

図 7.6 asm 関数の書式

*アセンブリコード*の部分には埋め込みたいアセンブリ言語のコードを記述します。記述方法はアセンブラの表記規則に従ってください。文字列リテラル（4.1.4参照）と同様、

エスケープシーケンスも記述できます。なお、*アセンブリコード*の先頭はラベルとして認識されるので、ニーモニックは空白文字の後の２文字目以降に記述してください。

例：
```
asm("  ldi R0, #h'10");
      ↑1文字以上の空白
```

asm 関数では、アセンブリコード以外に、最大 2 個のパラメータ（式）を指定できます。各パラメータは asm 関数のアセンブリコード実行前に評価され、*パラメータ 1* の値はレジスタ R0 に、*パラメータ 2* の値はレジスタ R1 にそれぞれセットされます。パラメータ値は整数型のみ有効です（signed および unsigned の char、short 型に対しては、整数型拡張（integral promotion）の規則が適用されます。4.3.2「暗黙の型変換」参照）。それ以外の型の式を記述した場合、C/C++ コンパイラの動作は保証されません。

例：
```
int  i, j;
asm ( "  add R0, R1", i, j );
```
変数 i の値が R0 に、変数 j が R1 にあらかじめセットされます。

パラメータを指定しなかった場合、R0 および R1 の内容は不定です。

7.5.3 asm関数の制限事項

asm 関数に関して以下のような制限事項および注意事項があります。記述時に注意してください。

■ レジスタの使用における制限事項

asm 関数内で使用することが認識されているレジスタは R0 ～ R3 の 4 つです。これら以外のレジスタおよびアキュムレータを asm 関数で使用したい場合は、そのレジスタ値が asm 関数の呼び出し前と後で変わらないようにプログラミングする必要があります（ユーザーの作業）。それには、asm 関数内でレジスタを使用する前にレジスタ値を保存しておきます。そしてレジスタを使用し終わったら、保存しておいた値をレジスタに戻します。

例： asm 関数内でレジスタ R4 を退避・回復する例

```
asm("  ST  R4,@-sp\n"      /*R4 の退避 */
    " ..................... \n"    /*R4 を使用した処理 */
    "  LD  R4,@sp+\n");    /*R4 の回復 */
```

■ コンパイル時の制限事項（最適化指定）

コンパイラの最適化では、不要コードの削除、命令の置換、命令の入れ換えなどが強力に実施されます。したがって、asm 関数を使用しているソースファイルを最適化を指定してコンパイルする場合、C/C++ コンパイラの最適化の影響を十分考慮してください。

■ エラーチェックに関する注意

C/C++ コンパイラは、asm 関数内のアセンブリコードの内容をチェックしません。asm 関数中に不正なアセンブリコードを記述した場合、アセンブラによってエラーが検出されます。この場合、エラーメッセージは C/C++ 言語ソースファイルに対してではなく、アセンブリ言語ソースファイルに対して出力されます。C/C++ コンパイラを起動した結果、アセンブラのエラーメッセージが出力された場合、まず、asm 関数内の内容を確認することをおすすめします。

■ パラメータ指定における制限事項

asm 関数のパラメータには式が記述可能ですが、定数式や識別子以外を記述することは推奨されません。以下のような式を記述した場合、C/C++ コンパイラが正しく動作しない可能性があります。

・ 副作用のある式（++、- -、=、+ - などの演算子。関数呼び出しが含まれている式。等）
・ 複雑な式

■ asm 関数の長さについての制限事項

asm 関数には、1,000 文字程度まで記述可能です。複数行のアセンブリコードを記述する場合は分割して複数の asm 関数に記述することをお勧めします。

■ ラベルについての制限事項

ラベルはコンパイラが内部生成するラベルと重複しないもの（”_”アンダースコアで始まるラベル名）を使用することが推奨されます。その際、ラベル名のアンダースコア以外の部分が C/C++ 言語で定義されているシンボルや関数名と重複しないようにしてください。

■asm 関数に記述できる命令に関する制限事項

疑似命令およびマクロ処理機能は記述できません。

■ asm 関数を記述したプログラムに対する最適化に関する制限事項

asm 関数を記述したプログラムに最適化を指定してコンパイルすると、以下に示すワーニングメッセージが表示され、最適化の一部が抑制されることがあります。インラインアセンブル機能を使用している関数は、別のモジュールで定義していただくことを推奨します。

○ワーニングメッセージ

<command line>: warning: xxx.c: unable to optimize -- skipped phase

■ その他制限事項

asm 関数中に以下のような記述を行うことは推奨されません。これらの記述を行う場合、ユーザーの責任において十分な注意を払ってください。

・ 分岐命令
・ アセンブラの擬似命令およびマクロ命令
・ スタックの内容の書き換え
・ ラベルの定義
・ C/C++ コンパイラが内部生成するラベルの参照

7.5.4 使用例

asm 関数を用いたプログラミング例を以下に示します（図 7.7）。

```
/* 配列 X[cnt],Y[cnt]の各要素を掛けて、総和を求める例 */
#pragma keyword asm on
void sumXY(short *X, short *Y, int cnt, int *output)
{
    asm ("  mvtachi     r0\n"
        "   mvtaclo     r0", 0);
    for ( ; cnt-- > 0; ++X, ++Y )
         asm("    macwlo   r0, r1", *X, *Y);
    asm ("    mvfachi   r3\n"
          "    st         r3,@r0\n"
           "    mvfaclo   r3\n"
          "    st          r3,@+r0\n", (int)output);
}
```

図 7.7 asm 関数使用例

上記図 7.7 に示すソースプログラムのコンパイル結果の例を図 7.8 に示します。

```
    .IMPORT $_100_builtin_memcopy
    .SECTION    P,CODE,ALIGN=4
    .EXPORT $sumXY
$sumXY:
    LDI     R0,#0
    mvtachi r0
    mvtaclo r0
    BRA     L5
L4:
    LDH     R0,@R4
    LDH     R1,@R5
    macwlo  r0, r1
L2:
    ADDI    R4,#2
    ADDI    R5,#2
L5:
    MV      R1,R6
    ADD3    R6,R1,#-1
    BGTZ    R1,L4
L3:
    MV      R0,R7
    mvfachi r3
    st      r3,@r0
    mvfaclo r3
    st      r3,@+r0

L1:
    JMP     R14
    .END
```

図7.8　asm関数コンパイル結果（例）

第８章
標準ヘッダファイル

8.1 標準ヘッダファイルの概要

標準ヘッダファイルとは、C標準ライブラリ関数を使用するために必要なプロトタイプ宣言、マクロ定義、および、データ型宣言が記述されているファイルです。標準ヘッダファイルは15種類あります（表8.1）。C標準ライブラリ関数を使用する場合、ライブラリ関数の実行に必要な定義や宣言を行っているヘッダファイルを、処理単位ごとにインクルードする必要があります。

以下に標準ヘッダファイル、および、それぞれのヘッダファイルに対応するライブラリ関数（分類名）を示します。

表8.1　標準ヘッダファイル一覧

ヘッダファイル	概要	対応するライブラリ関数
assert.h	プログラムの診断情報の出力を行うマクロ定義	プログラム診断関数
ctype.h	文字操作関数や文字チェック関数のマクロ定義	文字操作関数
errno.h	エラー番号に関するマクロ定義	全関数（必要に応じて）
float.h	浮動小数点数の内部表現に関する各種制限値マクロの定義	数学関数など（float.hのマクロ使用時のみ）
limits.h	コンパイラの内部処理に関する各種制限値マクロの定義	全関数（必要に応じて）
locale.h	ロケール（地域化）操作関数の宣言	ロケール（地域化）操作関数
math.h	倍精度および単精度の数学関数の宣言、マクロ定義	数学関数
mathf.h	単精度数学関数の宣言、マクロ定義	数学関数
setjmp.h	分岐関数の宣言、データ型宣言	プログラムの制御移動関数
signal.h	シグナル（割り込み）処理関数の宣言	シグナル処理関数
stdarg.h	可変個の実引数を持つ関数のマクロ定義、データ型宣言	可変個の実引数アクセス関数
stddef.h	各標準ヘッダで共通に使用する定義、データ型宣言	全関数（必要に応じて）
stdio.h	入出力関数の宣言、データ型宣言、マクロ定義	入出力関数
stdlib.h	メモリ管理等のCプログラムでの標準的処理関数の宣言、データ型宣言、マクロ定義	標準処理関数
string.h	文字列操作関数やメモリ操作関数の宣言	文字列操作関数
time.h	日付および時刻操作関数の宣言	日付・時刻操作関数

8.2 標準ヘッダファイルの内容

ここでは、各標準ヘッダファイルにおいて、宣言または定義している標準ライブラリ関数を示します。これらの関数を使用する場合は、必ず対応するヘッダファイルのインクルードが必要です。また、制限値を定義しているマクロがある場合はマクロ名の一覧を示します。ヘッダファイルはアルファベット順に掲載してあります（8.2.1～8.2.15）。

8.2.1 assert.h

プログラム診断用関数assert（リエントラント性　×）を定義しています（引数を持つマクロ定義）。

8.2.2 ctype.h

文字操作関数のプロトタイプ宣言およびマクロ定義を行っています。宣言している関数は以下のとおりです。

表8.2　ctype.hで宣言している関数

関数	概要	リエントラント性
isalnum	英字または10進数字かどうかを判定します。	○
isalpha	英字かどうかを判定します。	○
iscntrl	制御文字かどうかを判定します。	○
isdigit	10進数字かどうかを判定します。	○
isgraph	空白を除く印字文字かどうかを判定します。	○
islower	英小文字かどうかを判定します。	○
isprint	空白を含む印字文字かどうかを判定します。	○
ispunct	特殊文字かどうかを判定します。	○
isspace	空白類文字かどうかを判定します。	○
isupper	英大文字かどうかを判定します。	○
isxdigit	16進数字かどうかを判定します。	○
tolower	英大文字を英小文字に変換します。	○
toupper	英小文字を英大文字に変換します。	○

8.2.3 errno.h

エラーが発生したときに、エラー番号を保持する外部変数errnoと、エラー番号を表すマクロ定義を行っています。定義しているマクロ名は以下のとおりです（表8.3）。

表8.3 errno.hで定義しているマクロ

マクロ名	説明
EDOM	入力引数の値が関数内で定義されている値の範囲外であるとき、 errno に設定する値を示しています。
ERANGE	関数の計算結果がdouble型の値として表せないとき 、あるいはオーバーフロー／アンダフローとなったとき、errno に設定する値を示しています。

8.2.4 float.h

浮動小数点数の内部表現に関する各種制限値のマクロを定義しています。定義しているマクロ名および定義値は以下のとおりです。

表8.4 float.hで定義しているマクロ(1/2)

マクロ名	説明
FLT_RADIX	指数部表現における基数を示します。
FLT_ROUNDS	加算演算結果の丸めのモードを示します。本マクロの定義の意味は以下のとおりです。 ・最も近い値（最近値）に丸める ：1 ・+∞方向に丸める ：2 ・-∞方向に丸める ：3 ・0方向に丸める ：0 ・特に規定しない ：-1 丸め、切り捨ての方法は処理系定義です。
FLT_MAX	float 型の浮動小数点数値として表現できる正の最大値を示します。
DBL_MAX	double 型の浮動小数点数値として表現できる正の最大値を示します。
LDBL_MAX	long double 型の浮動小数点数値として表現できる正の最大値を示します。
FLT_MIN	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	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	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	float 型の浮動小数点数値として表現できる基数の累乗の最大値を示します。
DBL_MAX_EXP	double 型の浮動小数点数値として表現できる基数の累乗の最大値を示します。
LDBL_MAX_EXP	long double 型の浮動小数点数値として表現できる基数の累乗の最大値を示します。
FLT_MIN_EXP	float 型として表現できる浮動小数点数値の基数の累乗の最小値を示します。
DBL_MIN_EXP	double 型として表現できる浮動小数点数値の基数の累乗の最小値を示します。
LDBL_MIN_EXP	long double 型として表現できる浮動小数点数値の基数の累乗の最小値を示します。

表 8.4　float.h で定義しているマクロ（2/2）

マクロ名	説明
FLT_MAX_10_EXP	float 型の浮動小数点数値として表現できる10の累乗の最大値を示します。
DBL_MAX_10_EXP	double 型の浮動小数点数値として表現できる10の累乗の最大値を示します。
LDBL_MAX_10_EXP	long double 型の浮動小数点数値として表現できる10の累乗の最大値を示します。
FLT_MIN_10_EXP	float 型として表現できる浮動小数点数値の10の累乗の最小値を示します。
DBL_MIN_10_EXP	double 型として表現できる浮動小数点数値の10の累乗の最小値を示します。
LDBL_MIN_10_EXP	long double 型として表現できる浮動小数点数値の10の累乗の最小値を示します。
FLT_DIG	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	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	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	float 型の浮動小数点数値を基数に合わせて表現したときの仮数部の最大桁数を示します。
DBL_MANT_DIG	double 型の浮動小数点数値を基数に合わせて表現したときの仮数部の最大桁数を示します。
LDBL_MANT_DIG	long double 型の浮動小数点数値を基数に合わせて表現したときの仮数部の最大桁数を示します。
FLT_EPSILON	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_EPSILON	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_EPSILON	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。

浮動小数点数の内部表現に関する各種制限値を定義します。
以下はすべて処理系定義です。

種別	定義名	定義値	説明
	FLT_GUARD	1	乗算演算結果においてガードビット を用いるかどうを示します。本マクロの定義の意味は以下の通りです。 ・ガードビットを用いる場合 ：1 ・ガードビットを用いない場合：0
	FLT_NORMALIZE	1	浮動小数点数の値を正規化するかどうかを示します。本マクロの定義の意味は以下のとおりです。 ・正規化する場合 ：1 ・正規化しない場合 ：0
	FLT_EXP_DIG	8	float型の浮動小数点数値を基数に合わせて表現した時の指数部の最大桁数です。
	DBL_EXP_DIG	11	double型の浮動小数点数値を基数に合わせて表現した時の指数部の最大桁数です。
	LDBL_EXP_DIG	11	long double型の浮動小数点数値を基数に合わせて表現した時の 指数部の最大桁数です。
	FLT_POS_EPS	5.9604648328104304e-8F	float型において、1.0＋x≠1.0である最小の浮動小数点数値xを示します。
	DBL_POS_EPS	1.1102230246251567e-16	double型において、1.0＋x≠1.0で ある最小の浮動小数点xを示します。
	LDBL_POS_EPS	1.1102230246251567e-16	long double型において、1.0＋x≠1.0である最小の浮動小数点数値xを示します。
	FLT_NEG_EPS	5.9604642999033786e-8F	float型において、1.0－x≠1.0である最小の浮動小数点数値xを示します。
	DBL_NEG_EPS	1.1102230246251565e-16	double型において、1.0－x≠1.0である最小の浮動小数点数値xを示します。
	LDBL_NEG_EPS	1.1102230246251565e-16	long double型において、1.0－x≠1.0である最小の浮動小数点数値xを示します。
	FLT_POS_EPS_EXP	-23	float型において、 1.0＋(基数)n≠1.0となる最小の整数n を示します
	DBL_POS_EPS_EXP	-52	double型において、1.0＋(基数)n≠1.0となる最小の整数nを示します。
	LDBL_POS_EPS_EXP	-52	long double型において、1.0＋(基数)n≠1.0となる最小の整数nを示します。
マクロ	FLT_NEG_EPS_EXP	-24	float型において、1.0－(基数)n≠1.0となる最小の整数nを示します。
マクロ	DBL_NEG_EPS_EXP	-53	double型において、1.0－(基数)n≠1.0となる最小の整数nを示します。
マクロ	LDBL_NEG_EPS_EXP	-53	long double型において、1.0－(基数)n≠1.0となる最小の整数nを示します。

8.2.5 limits.h

各型の数値上の制限値に関するマクロ定義を行っています。定義しているマクロは以下のとおりです。

表8.5 limits.hで定義しているマクロ

マクロ名	説明
CHAR_BIT	char 型が何ビットから構成されるかを示します。
CHAR_MAX	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	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	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	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	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	short 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。
SHRT_MIN	short 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。
USHRT_MAX	unsigned short 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_MAX	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	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	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	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	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	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。

8.2.6 locale.h

プログラムの地域化を操作する関数（ロケール操作関数）のプロトタイプ宣言およびマクロ定義を行っています。宣言している関数および定義しているマクロ名は以下のとおりです（表8.6、表8.7）。

表8.6 locale.hで宣言している関数

関数	概要	リエントラント性
localeconv	lconv型構造体を初期化します。	×
setlocale	ロケール情報の設定、検索をします。	×

表8.7 locale.hで定義しているマクロ

マクロ名	説明
LC_ALL	すべてのロケール情報を設定、検索します。
LC_COLLATE	strcoll関数, strxfrm関数に影響を与える情報を設定、検索します。
LC_CTYPE	isdigit関数とisxdigit関数を除くすべての文字操作関数とマルチバイトを扱う関数すべてに影響を与える情報を設定、検索します。
LC_MONETARY	localeconv関数が返す通貨情報に影響を与える情報を設定、検索します。
LC_NUMERIC	入出力関数と文字列操作関数で使用される小数点、およびlocaleconv関数が返す通貨情報以外の情報に影響を与える情報を設定、検索します。
LC_TIME	strftime関数に影響を与える情報を設定、検索します。

8.2.7 math.h

数学関数のプロトタイプ宣言およびマクロ定義を行っています。宣言している関数および定義しているマクロは以下のとおりです（表8.8、表8.9）。

表 8.8 math.h で宣言している関数

関数	概要	リエントラント性
acos	浮動小数点数の逆余弦を計算します。	×
asin	浮動小数点数の逆正弦を計算します。	×
atan	浮動小数点数の逆正接を計算します。	×
atan2	浮動小数点数どうしを除算した結果の値の逆正接を計算します。	×
ceil	浮動小数点数の小数点以下を切り上げた整数値を求めます。	×
cos	浮動小数点数のラディアン値の余弦を計算します。	×
cosh	浮動小数点数の双曲線余弦を計算します。	×
exp	浮動小数点数の指数関数を計算します。	×
fabs	浮動小数点数の絶対値を計算します。	×
floor	浮動小数点数の小数点以下を切り捨てた整数値を求めます。	×
fmod	浮動小数点数どうしを除算した結果の余りを計算します。	×
frexp	浮動小数点数を（0.5，1.0）の値と２の累乗の積に分解します。	×
ldexp	浮動小数点数と2の累乗の乗算を計算します。	×
log	浮動小数点数の自然対数を計算します。	×
log10	浮動小数点数の10を底をする対数を計算します。	×
modf	浮動小数点数を整数部分と小数部分に分解します。	×
pow	浮動小数点数の累乗を計算します。	×
sin	浮動小数点数のラディアン値の正弦を計算します。	×
sinh	浮動小数点数の双曲線正弦を計算します。	×
sqrt	浮動小数点数の正の平方根を計算します。	×
tan	浮動小数点数のラディアン値の正接を計算します。	×
tanh	浮動小数点数の双曲線正接を計算します。	×

表 8.9 math.h で定義しているマクロ

マクロ名	説明
HUGE_VAL	関数の計算結果がオーバーフローしたときに、関数のリターン値として返す値を示しています。

8.2.8 setjmp.h

分岐関数のプロトタイプ宣言および必要なデータ型宣言を行っています。宣言している関数およびデータ型は以下のとおりです（表8.10、表8.11）。

表8.10 setjmp.hで宣言している関数

関数	概要	リエントラント性
longjmp	setjmp関数で退避していた関数の実行環境を回復し、setjmp関数を呼び出したプログラムの位置に制御を移動します。	○
setjmp	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。	○

表8.11 setjmp.hで宣言しているデータ型

データ型	概要
jmp_buf	関数間の制御の移動を可能とする情報を保存しておくための記憶域に対応する型名を示しています。

8.2.9 signal.h

シグナル処理関数のプロトタイプ宣言およびマクロ定義を行っています。宣言している関数は以下のとおりです（表8.12）。

表8.12 signal.hで宣言している関数

関数	概要	リエントラント性
signal	シグナルに応答する関数を設定します。	ユーザー記述に依存
raise	プログラムに対してシグナルを送ります。	ユーザー記述に依存

8.2.10 stdarg.h

可変個の実引数を持つ関数に必要なマクロ定義およびデータ型宣言を行っています。定義しているマクロおよび宣言しているデータ型は以下のとおりです（表 8.13、8.14）。なお、ここに示すマクロについては、9.2「標準ライブラリ関数詳細」で詳しく説明しています。

表 8.13　stdarg.h で定義しているマクロ

マクロ名	説明	リエントラント性
va_start	可変個の引数の参照を行うため、初期設定処理を行います。	○
va_arg	可変個の引数から順に引数を取り出します。	○
va_end	可変個の引数を持つ関数の引数への参照を終了します。	○

表 8.14　stdarg.h で宣言しているデータ型

データ型	概要
va_list	可変個の引数を参照するために、va_start，va_arg，va_end マクロで共通に使用される変数の型を示しています。

8.2.11 stddef.h

各標準ヘッダファイルで共通に使用するマクロの定義、共通で使用するデータ型の宣言を行っています。定義しているマクロおよび宣言しているデータ型は以下のとおりです(表8.15、表8.16)。

表8.15　stddef.hで定義しているマクロ

マクロ名	説明
errno	ライブラリ関数の処理中にエラーが発生した場合、そのライブラリごとに定義されたエラーコードがこのerrno に設定されます。ライブラリ関数を呼び出す前にerrno に0を設定しておき、ライブラリ関数の処理終了後に errno に設定されているコードを調べることによってライブラリ関数の処理中に発生したエラーをチェックすることができます。
NULL	ポインタが何も指していないときの値を示します。

表8.16　stddef.hで宣言しているデータ型

データ型	概要
ptrdiff_t	二つのポインタを減算した結果の型を示します。
size_t	sizeof演算子による演算結果の型を示します。
wchar_t	ワイド文字(L'..')の型を示します。

8.2.12 stdio.h

入出力関数のプロトタイプ宣言、および、入出力関数に必要なマクロ定義とデータ型の宣言を行っています。宣言している関数、定義しているマクロ、および、宣言しているデータ型は以下のとおりです（表8.17、表8.18、表8.19）。

表8.17　stdio.hで宣言している関数（1/2）

関数	概要	リエントラント性
clearerr	ストリーム入出力用ファイルのエラー状態をクリアします。	×
fclose	ストリーム入出力用ファイルをクローズします。	×
feof	ストリーム入出力用ファイルが終わりであるかどうかを判定します。	×
ferror	ストリーム入出力用ファイルがエラー状態であるかどうかを判定します。	×
fflush	ストリーム入出力用ファイルの内容をファイルへ出力します。	×
fgetc	ストリーム入出力用ファイルから１文字入力します。	×
fgetpos	現在のファイルストリームの位置を求めます。	×
fgets	ストリーム入出力用ファイルから文字列を入力します。	×
fopen	ストリーム入出力用ファイルを指定したファイル名によってオープンします。	×
fprintf	書式に従ってストリーム入出力用ファイルへデータを出力します。	×
fputc	ストリーム入出力用ファイルへ１文字出力します。	×
fputs	ストリーム入出力用ファイルへ文字列を出力します。	×
fread	ストリーム入出力用ファイルから指定した記憶域にデータを入力します。	×
freopen	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て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。	×
fscanf	ストリーム入出力用ファイルからデータを入力し、書式に従って変換します。	×
fseek	ストリーム入出力用ファイルの現在の読み書き位置を移動します。	×
fsetpos	ファイルストリーム上の位置を変更します。	×
ftell	ストリーム入出力用ファイルの現在の読み書き位置を求めます。	×
fwrite	記憶域からストリーム入出力用ファイルへデータを出力します。	×
getc	ストリーム入出力用ファイルから1文字入力します。	×
getchar	標準入力（stdin）から1文字入力します。	×
gets	標準入力（stdin）から文字列を入力します。	×
perror	標準エラー出力（ 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。	×
printf	データを書式に従って変換し、標準出力(stdout)へ出力します。	×
putc	ストリーム入出力ファイルへ1文字出力します。	×

表 8.17　stdio.h で宣言している関数（2/2）

関数	概要	リエントラント性
putchar	標準出力（stdout）へ1文字出力します。	×
puts	標準出力（stdout）へ文字列を出力します。	×
remove	ファイルを削除します。	ユーザー記述に依存
rename	ファイル名を変更します。	ユーザー記述に依存
rewind	ストリーム入出力用ファイルの現在の読み書き位置をファイルの先頭に移動します。	×
scanf	標準入力(stdin)からデータを入力し、書式に従って変換します。	×
setbuf	ストリーム入出力用のバッファをユーザープログラム側で定義して設定します。	×
setvbuf	ストリーム入出力用のバッファをユーザープログラム側で定義し、さらにバッファの使用方法を設定します。	×
sprintf	データを書式に従って変換し、指定した領域へ出力します。	×
sscanf	指定した記憶域からデータを入力し、書式に従って変換します。	×
tmpfile	テンポラリファイルを生成します。	×
tmpnam	既存しないテンポラリファイル名を生成します。	×
ungetc	ストリーム入出力用ファイルへ１文字を戻します。	×
vfprintf	可変個のパラメー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へ出力します。	×
vprintf	可変個のパラメータリストを書式に従って、標準出力へ出力します。	×
vsprintf	可変個のパラメータリストを書式に従って、指定した記憶域へ出力します。	×

表 8.18　stdio.h で定義しているマクロ

マクロ名	説明
_IOFBF	バッファ領域の使用方法として、入出力処理はすべてバッファ領域を使用することを示しています。
_IOLBF	バッファ領域の使用方法として、入出力処理は行単位でバッファ領域を使用することを示しています。
_IONBF	バッファ領域の使用方法として、入出力処理はバッファ領域を使用しないことを示しています。
BUFSIZ	入出力処理において必要とするバッファの大きさを示しています。
EOF	ファイルの終わり（end of file）すなわちファイルからそれ以上の入力が無いことを示す文字です。
L_tmpnam	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	ファイルの現在の読み書き位置を現在の位置からのオフセットに移すことを示しています。
SEEK_END	ファイルの現在の読み書き位置をファイルの終了位置からのオフセットに移すことを示しています。
SEEK_SET	ファイルの現在の読み書き位置をファイルの先頭位置からのオフセットに移すことを示しています。
TMP_MAX	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大値を示します。
FOPEN_MAX	fopen関数が一度にオープンできるファイルの最大数を示します。ただし、その最大数おstdio、stdout、stderrを含みます。
FILENAME_MAX	オープンできるファイル名の最大長を示します。
stderr	標準エラー出力ファイルに対応のファイルポインタを示します。
stdin	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	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。

表8.19　stdio.hで宣言しているデータ型

データ型	概要
FILE	ストリーム入出力処理で必要とするバッファへのポインタやエラー指示子、終了指示子などの各種制御情報を保存しておく構造体の型を示しています。
fpos_t	fsetpos、fgetpos関数でファイルの位置を表すのに用います。

8.2.13 stdlib.h

標準的処理（メモリ管理、終了処理）関数のプロトタイプ宣言、および、標準的処理関数に必要なマクロ定義とデータ型の宣言を行っています。宣言している関数、定義しているマクロ、および、宣言しているデータ型は以下のとおりです（表8.20、表8.21、
表8.22)。

表8.20　stdlib.hで宣言している関数

関数	概要	リエントラント性
abort	プログラムの実行を強制終了します。	×
abs	int 型整数の絶対値を計算します。	○
atexit	プログラムが正常終了した時に呼び出される関数を登録します。	×
atof	数を表現する文字列をdouble型の浮動小数点数値に変換します。	×
atoi	10進数を表現する文字列を int 型の整数値に変換します。	×
atol	10進数を表現する文字列をlong 型の整数値に変換します。	×
bsearch	２分割検索を行います。	○(比較関数に依存)
calloc	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。	×
div	int 型整数の除算の商と余りを計算します。	○
exit	プログラムを終了します。	×
free	指定された記憶域を開放します。	×
getenv	環境変数の内容を取得します。	ユーザー記述に依存
labs	long 型整数の絶対値を計算します。	○
ldiv	long 型整数の除算の商と余りを計算します。	○
malloc	記憶域を割り当てます。	×
mblen	マルチバイト文字のバイト数を求めます。	○
mbstowcs	マルチバイト文字列をワイド文字列に変換します。	○
mbtowc	マルチバイト文字をワイド文字に変換します。	○
qsort	ソートを行います。	○(比較関数に依存)
rand	0からRAND_MAXの間の擬似乱数整数を生成します。	×
realloc	記憶域の大きさを指定された大きさに変更します。	×
srand	rand 関数で生成する擬似乱数数列の初期値を設定します。	×
strtod	数を表現する文字列を double 型の浮動小数点数値に変換します。	×
strtol	数を表現する文字列をlong 型の整数値に変換します。	×
strtoul	数を表現する文字列をunsigned long型の整数値に変換します。	×
system	コマンド文字列をコマンドプロセッサによって実行されるホスト環境に渡します。	ユーザー記述に依存
wcstombs	ワイド文字列をマルチバイト文字列に変換します。	○
wctomb	ワイド文字をマルチバイト文字に変換します。	○

表 8.21　stdlib.h で定義しているマクロ

マクロ名	説明
RAND_MAX	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。

表 8.22　stdlib.h で宣言しているデータ型

データ型	概要
div_t	div 関数のリターン値の構造体の型を示しています。
ldiv_t	ldiv 関数のリターン値の構造体の型を示しています。

8.2.14 string.h

文字列操作関数とメモリ操作関数のプロトタイプ宣言を行っています。宣言している関数以下のとおりです。

表8.23　string.hで宣言している関数

関数	概要	リエントラント性
memchr	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。	○
memcmp	二つの記憶域を比較します。	○
memcpy	複写元の記憶域の内容を指定した大きさ分、複写先の記憶域に複写します。	○
memmove	複写元の記憶域の内容を指定した大きさ分、複写先の記憶域に移動します。	○
memset	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。	○
strcat	文字列の後に、文字列を連結します。	○
strchr	文字列において、指定された1文字が最初に現われる位置を検索します。	○
strcmp	二つの文字列を比較します。	○
strcoll	現在のロケールに基づいて指定された２つの文字列を比較します。	○
strcpy	複写元の文字列の内容を、複写先の記憶域にnull文字も含めて複写します。	○
strcspn	文字列において、指定外の文字が先頭から何文字続くかを求めます。	○
strerror	エラーメッセージを返します。	○(_strerrorに依存)
strlen	文字列の長さを計算します。	○
strncat	文字列の後に、文字列を指定した文字数分連結します。	○
strncmp	二つの文字列を指定された文字数分まで比較します。	○
strncpy	複写元の文字列を指定された文字数分、複写先の記憶域に複写します。	○
strpbrk	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。	○
strrchr	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	文字列において、指定された文字が先頭から何文字続くかを求めます。	○
strstr	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	文字列をいくつかのトークン（字句）に切り分けます。	×
strxfrm	現在のロケールに基づいて文字列を変換します。	○

8.2.15 time.h

日付・時刻操作関数のプロトタイプ宣言とマクロ定義を行っています。宣言している関数以下のとおりです。

表 8.24　time.h で宣言している関数

関数	概要	リエントラント性
asctime	tm型構造体で示される日時をテキスト文字列の形式に変換します。	×
clock	プログラムの実行時間を求めます。	ユーザー記述に依存
ctime	time_t型の暦時間をテキスト文字列の形式に変換します。	×
difftime	指定された２つの時間差を求めます。	○
gmtime	暦時間を世界標準時に変換します。	×
localtime	暦時間を現在の現地時間に変換します。	×
mktime	tm型構造体で示される日時を暦時間に変換します。	×
strftime	tm型構造体で示される日時を書式に従って変換します。	×
time	現在の暦時間を求めます。	ユーザー記述に依存

第９章
C標準ライブラリ

本章では、CC32R に含まれる C 標準ライブラリについて説明します。

9.1 C標準ライブラリの概要

9.1.1 C標準ライブラリの種類

C 標準ライブラリには、メモリモデル（「A.2 メモリモデル」参照）ごとに、m32RcR.lib (スモールモデル用),m32RcRM.lib(ミディアムモデル用),m32RcRL.lib(ラージモデル用) があります。含有している関数はどちらも同じです。

■ m32RcR.lib　(スモールモデル用)
　m32RcRM.lib (ミディアムモデル用)
　m32RcRL.lib (ラージモデル用)

実際にどちらのライブラリを使用するかは、C コンパイラまたはリンカ起動時に -l オプションで指定します。

注意

1 つのアプリケーション内で、スタック渡し対応ライブラリとレジスタ渡し対応ライブラリの両方を混合して使用しないでください。

9.1.2 C標準ライブラリ関数の分類

C標準ライブラリはANSI仕様（ANSI/ISO 9899-1990）に準拠しています。含まれる関数は、機能別に以下の11種類に分類されます（表9.1）。各種関数は、対応する標準ヘッダファイルを各処理単位ごとにインクルードすることによって使用可能となります。

表9.1 C標準ライブラリの関数分類

分類	機能概要	対応するヘッダファイル
プログラム診断関数	プログラムの診断情報の出力	assert.h
文字操作関数	文字の操作およびチェック	ctype.h
数学関数	三角関数等の数値計算	math.h
プログラムの制御移動関数	関数間の制御の移動	setjmp.h
可変個の実引数アクセス関数	可変個の実引数を持つ関数における実引数へのアクセス	stdarg.h
入出力関数	入出力操作	stdio.h
標準処理関数	メモリ管理等のCプログラムでの標準的処理	stdlib.h
文字列操作関数	文字列の比較、複写等	string.h
ロケール操作関数	ロケール（地域化）の設定、操作	locale.h
日付・時刻操作関数	日付、時刻の設定、変更等	time.h
シグナル処理関数	シグナル（割り込み）の送受信	signal.h

この他に、上記関数を使用するための諸設定を行う「初期設定関数」および「終了処理関数」が用意されています。これらは非ANSI仕様で、関数名がアンダースコア（_）で始まります。通常、スタートアッププログラム内で使用します（7.3.6～7.3.8参照）。

9.1.3 C標準ライブラリ使用時の注意

C標準ライブラリを使用するにあたって、以下の事項に注意してください。

■ 標準ヘッダファイルをインクルードしてください。

C標準ライブラリを使用する場合、必要な標準ヘッダファイルをインクルードしてください。必要なヘッダファイルを知るには、第8章「標準ヘッダファイル」を参照してください。また、9.2「C標準ライブラリ関数詳細」では、各関数ごとに必要なヘッダファイルを示しています（#includeでインクルードしています）。

■ C標準ライブラリ関数と同名の関数を作らないでください。

ユーザープログラム中では、C標準ライブラリの関数と同名の関数を作らないようにしてください。ユーザーがANSI-C仕様と同名の関数を作ることは推奨されません。

9.1.4 C標準ライブラリ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、標準ヘッダファイルerrno.hで定義しているerrnoにエラー番号を設定する関数があります。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、そのエラーメッセージを出力することができます。errnoをチェックするプログラム例を以下に示します。

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <string.h>
void main(void)
{
  double x,a=2.0;

  x = asin(a);
  if( errno == EDOM )
   printf( "%s\n", strerror(errno) ); /* print error message */
}
```

図9.1 errnoチェックプログラムの例

上記の例では、asin関数を使ってaの逆正弦値を求めています。引数aがasin関数の定義域[－1, 1]を超えているとき、errnoに値EDOMが設定されるので、printf関数でエラーメッセージを出力しています。strerror関数は、エラー番号を実引数として渡すと、対応するエラーメッセージの文字列ポインタを返す関数です。

9.2 標準ライブラリの再構築方法

標準ライブラリのソース環境を用いて、任意のコード生成オプションでライブラリを作成できます。ここでは、この環境を用いてライブラリを作成する手順を説明します。

9.2.1 ライブラリ作成手順

(1) 環境変数の設定が完了している、コマンドプロンプトを用意します。

(2) 任意の作業用ディレクトリを用意します。

(3) (2) で用意したディレクトリに移動します。

(4) 次の例のように、ライブラリ名とコンパイルオプションを指定し、libg32R を起動して下さい。

【起動例】

ライブラリ名　m32RcR.lib, small モデル , 速度重視最適化に対応したライブラリを作成する場合。

```
libg32R m32RcR.lib -small -Otime
ライブラリ名 ｺｰﾄﾞ生成ｵﾌﾟｼｮﾝ
```

ライブラリ作成が正常であれば、次のファイルが (2) のディレクトリに作成されます。

m32RcR.lib	ライブラリ
m32RcR.stk	スタック使用量表示ファイル

【注意】

・libg32R を実行したディレクトリ上に、ライブラリまたはスタック使用量表示ファイルと同じ名前のファイルがある場合は、上書きされます。

・コード生成オプションとして指定できるのは、次のオプションの組み合わせに限られます。他のオプションは指定できません。

-access= アクセス制御ファイル

-constr

-D <name>

-D <name>=<def>

-exception

-float_only

-large

-medium

-noexception

-noinline

-O <priority> (-Ospace, -Otime)

-O <level> (-O0, -O1, ..., -O7)

-R <old>=<new>

-rel16

-rtti=off

-rtti=on

-small

-strict_standard

-switch_by_offset

-U <name>

-zdiv

《参考》

標準添付しているライブラリに対応する libg32R のコマンドライン指定の一覧

ライブラリ名	コマンドライン指定
m32RcR.lib	libg32R m32RcR.lib -small -Otime -zdiv
m32RcRM.lib	libg32R m32RcRM.lib -medium -Otime -zdiv
m32RcRL.lib	libg32R m32RcRL.lib -large -Otime -zdiv

9.3 C標準ライブラリ関数詳細

本節ではC標準ライブラリの各関数の詳細を、アルファベット順に示します。各関数詳細の読み方は以下のとおりです。

関数名 **分類**[注)]

機能概要

書式 関数の呼び出し形式を示します。#include ＜ヘッダファイル＞ で示すヘッダファイルは、この関数の実行に必要な標準ヘッダファイルです。必ずインクルードしてください。

戻り値 関数の戻り値（リターン値）を示します。戻り値の後に「：」を付けて記載されているコメントは、その戻り値についての説明（リターン条件等）です。

解説 関数の仕様について説明します。

注意 注意事項があればここで示します。

図9.2 C標準ライブラリ関数詳細の見方

注） ここで「non-ANSI」とある場合、C標準ライブラリには含まれるがANSIでは定義されていない関数であることを意味します(7.3.6、7.3.7、7.3.8節参照)。

_100_C_action_atexit

終了処理関数（non-ANSI）

ユーザー終了処理関数（atexit関数で登録された関数）を実行します。

書式　　void_100_C_action_atexit (void);

戻り値　　なし

解説　　_100_C_action_atexit関数は、atexit関数によって登録された、ユーザーの終了処理関数を実行します。登録関数をすべて実行終了した後、登録関数のカウンタを初期化します。

注意　　本関数は通常、スタートアッププログラム内でmain関数実行後に呼び出します（7.2、7.3節参照）。

_100_C_get_exit_code

終了処理関数（non-ANSI）

exit関数の終了コードを取り出します。

書式　　int_100_C_get_exit_code (void);

戻り値　　exit関数が返した終了コード

解説　　_100_C_get_exit_code関数は、exit関数が返す終了コードを取り出します。本関数は通常、スタートアッププログラム内で、exit関数による終了コードを知るために呼び出します（7.3.9「スタートアッププログラムサンプル」参照）。

abort

標準処理関数

プログラムの実行を強制終了します。

書式

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

戻り値　なし

解説　abort 関数は、プログラムの実行を強制終了します。また、その際にシグナル SIGABRT を発生します。abort 関数は、バッファのフラッシュ、ファイルのクローズおよびテンポラリファイルの削除はしません。

abs

標準処理関数

int 型整数の絶対値を計算します。

書式

```
#include  < stdlib.h >
int  abs ( int j );
   j;                                  /* 絶対値を求める整数 */
```

戻り値

演算結果の絶対値

解説

abs 関数は int 型の絶対値を計算します。演算結果が int 型で表現できないときの動作は保証されません。

acos

数学関数

浮動小数点数の逆余弦を計算します。

書式

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

戻り値　引数の計算値

解説　acos 関数は引数 x の逆余弦を計算し、0 からπまでの範囲の値を返します。x の値は -1 以上 1 以下の範囲でなければなりません。x が -1 より小さい、もしくは 1 より大きいときは errno に EDOM の値を設定します。

asctime

日付・時刻操作関数

tm型構造体で示される日時をテキスト文字列の形式に変換します。

書式

```
#include < time.h >
char *asctime ( const struct tm *timeptr );
   timeptr;                          /* tm型構造体へのポインタ */
```

戻り値

変換した文字列へのポインタ

解説

asctime 関数は、timeptr で示された日時を以下の形式に変換します。

```
Thu May 12 16:00:00 1995\n\0
```

asin

数学関数

浮動小数点数の逆正弦を計算します。

書式

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

戻り値

引数の計算値

解説

asin関数は引数xの逆正弦を計算し、-π/2からπ/2までの範囲の値を返します。 xの値は-1以上1以下の範囲でなければなりません。xが-1より小さい、もしくは1より大きいときはerrnoにEDOMの値を設定します。

assert

プログラム診断関数

プログラム中に診断機能を付け加えます。

書式

```
#include  < assert.h >
#include  < stdio.h >
void  assert ( int expression );
   expression;                  /* 評価する式 */
```

戻り値　なし

解説

assert 関数はマクロとして定義されており、expression が偽(0)のとき、診断メッセージを標準エラーファイルに出力し、その後 abort 関数を起動します。また、 expression が真のときは何も行いません。

assert 関数は通常、プログラムの論理的なエラーの検出に用い、expression にはプログラムが正常に動作した場合だけ真になるような条件式を指定します。

プログラムのデバッグ後 assert の記述をプログラムから削除する場合、<assert.h> を取り込む前にマクロ NDEBUG を #define 文で定義してください（#define　NDEBUG)。

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 関数の呼び出し効果は保証されません。

atan

数学関数

浮動小数点数の逆正接を計算します。

書式

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

戻り値

引数の計算値

解説

atan 関数は x の逆正接を計算します。atan 関数の値は 0 からπまでの範囲の値を返します。

atan2

数学関数

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。

書式

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

戻り値

引数の計算値

解説

atan2 関数は y/x の逆正接を計算します。atan2 関数は - πからπまでの範囲の値を返します。なお、atan2 関数の結果は点(x, y)と原点(0, 0)を通る直線と x 軸がなす角を示しています。

atan2 関数の両引数 x、y が共に 0 のとき、errno に EDOM の値を設定します。

atexit

標準処理関数

プログラムが正常終了した時に呼び出される関数を登録します。

書式

```
#include < stdlib.h >
int atexit ( void (*func)(void) );
    func;                          /* 登録する関数へのポインタ */
```

戻り値

0	：登録できた
0以外	：登録できなかった

解説

atexit関数は、プログラムが正常終了時あるいはexit関数を呼び出した時において呼び出される関数を登録します。関数の最大登録数は32個です。プログラムが正常終了した時に実行される関数の順序は登録順序の逆順です。

atof

標準処理関数

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。

書式

```
#include  < stdlib.h >
double  atof ( const char *nptr );
   nptr;                          /* 変換対象文字列へのポインタ */
```

戻り値

変換された値

解説

atof関数は、数を表現する文字列nptrをそれぞれdouble型の値に変換し、変換した値を返します。変換結果がオーバーフローあるいはアンダーフローの場合、errnoにERANGEの値を設定し、戻り値にHUGE_VAL（負の場合は-HUGE_VAL）あるいは0を返します。

atoi

標準処理関数

10進数を表現する文字列を int 型の整数値に変換します。

書式

```
#include  < stdlib.h >
int  atoi ( const char *nptr );
   nptr;                          /* 変換対象文字列へのポインタ */
```

戻り値

変換された値

解説

atoi 関数は数を表現する文字列 nptr を int 型の値に変換し、変換した値を返します。変換結果がオーバーフローの場合、errno に ERANGE の値を設定し、戻り値に INT_MAX（負の場合は INT_MIN）を返します。

atol

標準処理関数

10進数を表現する文字列を long 型の整数値に変換します。

書式

```
#include  < stdlib.h >
long  atol ( const char *nptr );
   nptr;                          /* 変換対象文字列へのポインタ */
```

戻り値

変換された値

解説

atol 関数は数を表現する文字列 nptr を long 型の値に変換し、変換した値を返します。変換結果がオーバーフローの場合、errno に ERANGE の値を設定し、戻り値に LONG_MAX（負の場合は LONG_MIN）を返します。

bsearch

標準処理関数

２分割検索を行います。

書式

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

    key;                        /* 検索するデータへのポインタ */
    base;                       /* 検索対象となるテーブルへのポインタ */
    nmemb;                      /* 検索対象のメンバの数 */
    size;                       /* 検索対象のメンバのバイト数 */
    compar ;                    /* 比較を行う関数へのポインタ */
```

戻り値

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た
NULL	：検索できなかった

解説

bsearch 関数は、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検索します。比較を行う関数は、比較する２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 のとき、負の値を返します。
・*p1 == *p2 のとき、0 の値を返します。
・*p1 > *p2 のとき、正の値を返します。

なお、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。

calloc

標準処理関数

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。

書式

```
#include  < stdlib.h >
void  *calloc ( size_t nelem, size_t elsize );
   nelem;                        /* 要素の数 */
   elsize;                       /* ひとつの要素の占めるバイト数 */
```

戻り値

割り当てられた記憶域の先頭のアドレス	：正常終了
NULL	：エラー（記憶域の割り当てができなかった。引数のいずれかが0）

解説

calloc関数は、elsizeバイト単位の記憶域をnelem個割り当てます。割り当てられた記憶域をすべて0によって初期化します。

注意

calloc関数では、記憶領域が割り当てられる毎に、その記憶領域にプラスして 12 バイトの領域が確保されます。この領域には、記憶領域の割り当て情報が格納されます。

ceil

数学関数

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。

書式

```
#include  < math.h >
double  ceil ( double x );
   x;                          /* 浮動小数点数 */
```

戻り値　計算結果

解説　ceil 関数は、x の値より大きいまたは等しい、最小の整数値を double 型の値として返します。

clearerr

入出力関数

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。

書式

```
#include  < stdio.h >
void  clearerr ( FILE *fp );
   fp;                          /*  FILE構造体へのポインタ  */
```

戻り値　なし

解説　clearerr 関数は、ファイルポインタ fp が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。エラー指示子、ファイル終了指示子とは、ストリームファイル毎に保持しているデータです。これらのデータはそれぞれ、エラーが発生したか否か、ファイルが終了したか否かを示し、それぞれ ferror、feof 関数によって参照できます。ストリームファイルを扱う関数のうち、その戻り値だけ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場合、これらのデータを用いてファイルの状態を調べます。

clock

日付・時刻操作関数

プログラムの実行時間を求めます。

書式

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

戻り値　未サポート

解説　clock 関数は、使用するシステムの環境に依存します。そのため現在 clock 関数はサポートされていません。

注意　clock 関数を使用するためには、clock 関数自身を作成する必要があります（第 11 章の表 11.2 参照）。

cos

数学関数

浮動小数点数のラディアン値の余弦を計算します。

書式

```
#include  < math.h >
double  cos ( double x );
   x;                          /* 浮動小数点数（ラディアン単位）*/
```

戻り値 引数の計算値

解説 cos 関数は x の余弦を計算します。

cosh

数学関数

浮動小数点数の双曲線余弦を計算します。

書式

```
#include  < math.h >
double  cosh ( double x );
   x;                          /* 浮動小数点数 */
```

戻り値 引数の計算値

解説 cosh 関数は x の双曲線余弦を計算します。関数の計算結果が double 型の値として表せないとき、errno に ERANGE の値を設定します。また、計算結果がオーバーフローした場合は、戻り値に HUGE_VAL を返します。

ctime

日付・時刻操作関数

time_t型の暦時間をテキスト文字列の形式に変換します。

書式

```
#include  < time.h >
char  *ctime ( const time_t *timer );
   timer;                    /* 暦時間(calendar time) */
```

戻り値　変換した文字列へのポインタ

解説　ctime関数は、timerで示される暦時間を以下のような形式に変換します。

```
Thu May 12 16:00:00 1995\n\0
```

difftime

日付・時刻操作関数

指定された２つの時間差を求めます。

書式

```
#include  < time.h >
double  difftime ( time_t time1, time_t time2 );
   time1;                    /* 時間１ */
   time2;                    /* 時間２ */
```

戻り値　time1からtime2を引いた値（秒単位）

解説　difftime関数は、time1からtime2を引いた値を求めます。

div

標準処理関数

int 型整数の除算の商と余りを計算します。

書式

```
#include  < stdlib.h >
div_t  div ( int number, int denom );
   number;                    /* 被除数 */
   denom;                     /* 除数 */
```

戻り値　演算結果の商と余り

解説　div 関数は、number を denom で除算した結果の商と余りを計算します。戻り値は div_t 構造体で、そのメンバの quot に商、rem に余りが入ります。

exit

標準処理関数

プログラムを終了します。

書式

```
#include  < stdlib.h >
void  exit ( int status );
   status;                    /* 終了ステータス */
```

戻り値　なし

解説　exit 関数は、プログラムを終了します。その際の後処理として以下の処理を実行します。

①オープンされているストリームのバッファをすべてフラッシュします。
②オープンされているファイルをすべてクローズします。
③ tmpfile で生成したファイルをすべて削除します。
④ atexit 関数で登録された関数を登録したときの逆順ですべて実行します。
⑤ setjmp 関数で保存しておいた環境へ、制御が復帰します（本関数を呼び出した側には制御は戻りま　せん)。

復帰先の環境において status の値を知るには _100_C_get_exit_code 関数を使用します。

exp

数学関数

浮動小数点数の指数関数を計算します。

書式

```
#include  < math.h >
double  exp ( double x );
   x;                          /* 浮動小数点数 */
```

戻り値 指数関数(e^x)の値

解説 exp 関数は x の指数関数(e^x)を計算し、その結果を返します。変換結果がオーバーフローの場合、errno に ERANGE の値を設定し、戻り値として HUGE_VAL を返します。

fabs

数学関数

浮動小数点数の絶対値を計算します。

書式

```
#include  < math.h >
double  fabs ( double x );
   x;                          /* 浮動小数点数 */
```

戻り値 x の絶対値

解説 fabs 関数は、x の絶対値を返します。

fclose

入出力関数

ストリーム入出力用ファイルをクローズします。

書式

```
#include  < stdio.h >
int  fclose ( FILE *fp );
   fp;                          /* FILE構造体へのポインタ */
```

戻り値

0	：正常終了
EOF	：エラー

解説

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。なお、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場合、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。さらに、ストリームがtmpfile関数でオープンされたものであれば対応するファイルを削除します。

feof

入出力関数

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。

書式

```
#include  < stdio.h >
int  feof ( FILE *fp );
   fp;                          /* FILE構造体へのポインタ */
```

戻り値

0	：ファイルが終わりでないとき
0以外	：ファイルが終わりのとき

解説

feof関数はファイルポインタfpが示すストリーム入出力用ファイルが終了したか否かを判定します。feof関数はファイル終了指示子が設定されていればファイルが終わりと判断します。

ferror

入出力関数

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。

書式

```
#include  < stdio.h >
int  ferror ( FILE *fp );
   fp;                                  FILE構造体へのポインタ
```

戻り値

0	：ファイルがエラー状態でない
0以外	：ファイルがエラー状態

解説

ferror関数は、ファイルポインタfpが示すストリーム入出力用ファイルがエラー状態か否かを判定します。ferror関数は、エラー指示子が設定されているときエラー状態であると判断します。

fflush

入出力関数

ストリーム入出力用ファイルの内容をファイルへ出力します。

書式

```
#include  < stdio.h >
int  fflush ( FILE *fp );
   fp;                                  FILE構造体へのポインタ
```

戻り値

0	：正常終了
EOF	：エラー

解説

fflush関数は、fpで指定したストリーム入出力用ファイルが出力用にオープンされているとき、ストリームのバッファの未出力内容をファイルに出力します。また、入力用にオープンされているとき、ungetc関数の指定を無効にします。

fgetc

入出力関数

ストリーム入出力用ファイルから１文字入力します。

書式

```
#include  < stdio.h >
int  fgetc ( FILE *fp );
   fp;                          /* FILE構造体へのポインタ */
```

戻り値

入力した１文字	：正常終了
EOF	：ファイルの終了時およびエラー発生時

解説

fgetc 関数は、ファイルポインタ fp が示すストリーム入出力用ファイルから１文字入力し、その文字を返します。

関数の戻り値だけでは、入出力エラー発生の有無やファイルの終了の情報が得られない場合、そのファイルの持っているエラー指示子（エラーが発生したか否かを示す）やファイル終了指示子（ファイルが終了したか否かを示す）を参照すれば知ることができます。ストリームファイルはそれぞれ、エラー指示子、ファイル終了指示子というデータを保持しています。エラー指示子は ferror 関数、ファイル終了指示子は feof 関数によって参照できます。

fgetpos

入出力関数

現在のファイルストリームの位置を求めます。

書式

```
#include  < stdio.h >
int  fgetpos ( FILE *stream, fpos_t *pos );
   stream;                    /* FILE構造体へのポインタ */
   pos;                       /* 現在のファイルストリームの位置 */
```

戻り値

0	：正常終了
0以外	：エラー（errnoにエラー番号を返します）

解説

fgetpos関数は、streamで指定されたファイルストリームの現在の位置をposが指す記憶域に返します。

fgets

入出力関数

ストリーム入出力用ファイルから文字列を入力します。

書式

```
#include  < stdio.h >
char  *fgets ( char *s, int n, FILE *fp );
   s;                          /* データの格納領域へのポインタ */
   n;                          /* 格納する文字数 */
   fp;                         /* FILE構造体へのポインタ */
```

戻り値

文字列を格納する記憶域へのポインタｓ　：正常終了

NULL　：ファイルの終了時およびエラー発生時

※ｓが指すデータ格納域の内容は、入力ファイルの終了時には変化しませんが、エラー発生時にはその内容は保証されません。

解説

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文字列を入力し、ポインタ s の指す記憶域に格納します。n-1 文字または改行文字まで、あるいは、ファイルの終わりまでの文字を入力し、その文字列の最後に null 文字を付加します。入力した改行文字は、文字列に含めて格納します。

関数の戻り値だけでは、入出力エラー発生の有無やファイルの終了の情報が得られない場合、そのファイルの持っているエラー指示子（エラーが発生したか否かを示す）やファイル終了指示子（ファイルが終了したか否かを示す）を参照すれば知ることができます。ストリームファイルはそれぞれ、エラー指示子、ファイル終了指示子というデータを保持しています。エラー指示子は ferror 関数、ファイル終了指示子は feof 関数によって参照できます。

floor

数学関数

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。

書式

```
#include  < math.h >
double  floor ( double x );
   x;                          /* 浮動小数点数 */
```

戻り値　計算結果

解説　floor 関数は、x の値を超えない範囲の整数の最大値を double 型の値として返します。

fmod

数学関数

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。

書式

```
#include  < math.h >
double  fmod ( double x, double y );
   x, y;                       /* 浮動小数点数 */
```

戻り値　x/y の余り

解説　fmod 関数は、x/y の余りを算出し、それを double 型で返します。戻り値（f とする）は被除数（x）と同符号で、f の絶対値は除数（y）の絶対値より小さく、次の式が成り立ちます。

```
x = y * i + f          ※ i はある整数値
```

y=0 の場合など、x/y の商を表現できない場合、処理結果は保証されません。

fopen

入出力関数

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。

書式

```
#include  < stdio.h >
FILE  *fopen ( const char *fname, const char *mode );
    fname;                          /* ファイル名 */
    mode;                           /* ファイルアクセスモード */
```

戻り値

オープンしたストリームへのポインタ	：正常終了
NULL	：エラー

解説

fopen 関数は、引数 fname に指定したファイルをオープンします。引数 mode には、ファイルのアクセスモードを下表より選んで設定してください。

アクセスモード	意味
"r"	ファイルを読み出し用にオープンします。
"w"	ファイルを書き込み用にオープンします。
"a"	ファイルを追加用にオープンします。
"r+"	ファイルを読み出し用かつ更新用にオープンします。
"w+"	ファイルを書き込み用かつ更新用にオープンします。
"a+"	ファイルを追加用かつ更新用にオープンします。

ファイルはテキストモードでオープンされますが、各アクセスモードの後に b を付けることによってバイナリモードでオープンできます(例 ： r+b)。

書き込みモードおよび追加モードで、存在しないファイルをオープンしようとした場合は、可能な限り新しいファイルを作成します。

既存のファイルを書き込みモードでオープンした場合、ファイルの先頭から書き込みが行われ、以前の内容は消去されます。

追加モードでオープンした場合、そのファイルの終わりの位置から書き込み処理が実行されます。

更新モードでオープンした場合、そのファイルに対して入出力処理が可能になります。ただし、出力処理後に 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、同様に入力処理後、上記関数を実行せずに出力処理を続けることはできません。

fprintf

入出力関数

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。

書式

```
#include <stdio.h>
int fprintf(FILE *fp, const char *control, ...);
    fp;                         /* FILE構造体へのポインタ */
    control;                    /* 書式を示す文字列へのポインタ */
    ...;                        /* 出力データを示す可変個引数列 */
```

戻り値

出力した文字数　：正常終了
負の値　：エラー

解説

fprintf関数は、書式controlに従って、可変個引数列で示す出力データを変換および編集し、ファイルへのポインタfpの指すストリーム入出力用ファイルへ出力します。

以下より、書式の詳細について示します(printf、sprintf、vfprintf、vprintf、vsprintf関数で扱う書式もこれと同様です)。

＜書式を示す文字列の構成＞

書式を示す文字列（control で指定する）は、通常文字列と変換仕様文字列から構成されます。変換仕様文字列に含まれる変換指定子の数と順番は、可変個引数列で列挙する引数の数と順番に対応しています。

■通常文字列

%で始まる文字列（変換仕様文字列）でないもの。そのまま出力されます。

例：　`fprintf(fp,"data=%02d", a);` ……下線部分が通常文字

■変換仕様文字列

%で始まる文字列。可変個引数列の各引数（ 以下、引数）の変換方法を指定します。文字列は%を先頭に、以下の変換指定要素から必要なものを組み合わせて構成します。

・フラグ
・フィールド幅
・精度
・引数のサイズ指定
・変換指定子

例：　`fprintf(fp,"data=%02d", a);` …下線部分が変換仕様。aを変換。

＜変換仕様文字列の形式＞

変換仕様文字列は次の形式で各要素を指定します（[]:省略可能）。

```
%[フラグ...][フィールド幅][.[精度]][引数のサイズ指定]変換指定子
```

例： `%02d` ――― %+フラグ（0）+フィールド幅（2）+変換指定子（d）

各要素は連続して記述します（スペースで区切らない）。変換仕様文字列に対する引数がない場合（可変個引数列がない、または、引数の数が足りない）の動作は保証されません。変換仕様文字列より引数の数が多い場合、余分な引数はすべて無視されます。

＜変換仕様文字列の各要素の詳細＞

変換仕様文字列の各要素について、その機能と指定方法を以下に示します。

■フラグ

符号を付けるなど、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その意味を以下に示します。

フラグ	意味
－	左詰め。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とき、データをフィールド内で左詰めにして出力する。このフラグがないときは右詰めとなる。
＋	符号付加。符号付きのデータに変換するとき、データの符号に従って、変換したデータの先頭に '+' あるいは '-' 符号を付ける。
空白	符号なし。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とき、そのデータの先頭に空白を付ける。上記の「+」と共に指定すると、本フラグは無視される。
#	修飾。変換指定子の種類に従って、以下のように変換後のデータに修飾を行う。
0	変換指定子がd、i、o、u、x、X、e、E、f、g、Gの場合、左詰めの空白がなくなるよう、数値の左側を複数の0で埋める。フラグ'－'と同時指定した場合、あるいは、変換指定子がd、i、o、u、x、Xの場合、精度が指定されると本フラグは無視される。

変換指定子	修飾
c,d,i,s,u	本フラグは無視される。
o	変換したデータが0でない場合、先頭に0を付ける。
x,X	変換したデータが0でない場合、先頭に0x（X変換の場合は0X）を付ける。
e,E,f,g,G	変換したデータに小数点以下がないときでも小数点を出力する。また、g,Gの場合、変換後のデータの後に付く0は取り除かない。

■フィールド幅

変換したデータの出力文字数を、10進数または*（アスタリスク）で指定します。

変換したデータの出力文字数がフィールド幅より少ないときは、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'－'を指定した場合は、データの後に空白が付けられます。

変換したデータの出力文字数がフィールド幅より大きいときは、フィールド幅が変換結果を出力できる幅に拡張されます。

フラグ'0'を指定した場合は、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。

■精度

変換指定子の種類に従って変換したデータの精度を指定します。

指定は、ピリオド(.)の後に10進数または*（アスタリスク）を続ける形式で行います。ピリオドの後に数字がない場合、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。

変換指定子の種類とそのときの精度指定の意味を以下に示します。

変換指定子	精度指定
d,i,o,u,x,X	変換したデータの最小の桁数を示す。
e,E,f	変換したデータの小数点以下の桁数を示す。
g,G	変換したデータの最大有効桁数を示す。
s	印字される最大文字数を示す。

■フィールド幅あるいは精度に対する * 指定

フィールド幅と精度指定の値には、*（アスタリスク）が指定できます。

*を指定すると、その変換仕様に対応する可変個引数の値が、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。フィールド幅に対する可変個引数の値が負の場合、正のフィールド幅に -（マイナス）フラグが指定されたと解釈します。また、精度に対する可変個引数の値が負の場合、精度が省略されたものと解釈します。

■引数のサイズ指定

変換指示子がd、i、o、u、x、X、e、E、f、g、G、nの場合、対応する引数のサイズを、修飾子h、l、Lで指定できます。修飾子は変換指示子の直前に書きます。サイズ指定の種類とその意味を下表に示します。

サイズ指定	該当変換指定子	対応する引数に対して指定するサイズ
h	d,i,o,u,x,X	shortintあるいはunsignedshortint
	n	shortintへのポインタ
l（エル）	d,i,o,u,x,X,n	longintあるいはunsigned longint
	n	long intへのポインタ
L	e,E,f,g,G	longdouble

変換指子がd、i、o、u、x、X、e、E、f、g、G、n以外の場合、本指定は無視されます。

■変換指定子

変換指定子は引数をどのような形式に変換するかを指定します。

変換する引数が、構造体や配列型のときや、構造体や配列型を指すポインタのときは、動作は保証されません（ただし、s変換で文字の配列を変換するとき、p変換でポインタを変換するときを除きます）。

以下に、変換指定子と変換方式を示します。ここに挙げていない文字を変換指定子として指定した場合、動作は保証されません。なお、「変換対象となる引数の型」は、h、l、Lによるサイズ指定がないときのものです。

（fprintf:変換指定子 1/2）

変換指定子	変換の方式	変換対象となる引数の型	精度に対する注意事項
d i	int型データを符号付き10進数の文字列に変換する。d変換とi変換は同仕様。	int	精度の指定は最低何文字出力されるかを示す。 変換後の文字数が精度の値より少ない場合、文字列の先頭に0が付く。 精度を省略すると1が仮定される。 0の値を持つデータを、精度0を指定して変換し出力しようとしても、何も出力されない。
o	unsignedint型データを符号なし8進数の文字列に変換する。	unsignedint	
u	unsignedint型データを符号なし10進数の文字列に変換する。	unsignedint	
x	unsignedint型データを符号なし16進数に変換する。 16進文字にはa、b、c、d、e、fを用いる。	unsignedint	
X	unsignedint型データを符号なし16進数に変換する。 16進文字にはA、B、C、D、E、Fを用いる。	unsignedint	
f	double型データを[-]ddd.dddの形式の10進数の文字列に変換する。	double	精度の指定は小数点以降の桁数を表す。小数点以降の文字が存在するときには、必ず小数点の前に一桁の数字が出力される。 精度を省略すると６が仮定される。 精度に０を指定すると、小数点と小数点以降の文字は出力されない。出力データは丸められる。
e	double型データを[-]d.ddde±dddの形式の10進数の文字列に変換する。 指数は最低2桁は出力される。	double	精度の指定は小数点以降の桁数を表す。変換した文字は、小数点の前に一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。精度を省略すると6が仮定される。 精度に0を指定すると、小数点以降の文字は出力されない。出力データは丸められる。
E	double型データを[-]d.dddE±dddの形式の10進数の文字列に変換する。 指数は最低2桁は出力される。	double	

（fprintf:変換指定子 2/2）

変換指定子	変換の方式	変換対象となる引数の型	精度に対する注意事項
g G	変換する値と有効桁数を指定する精度の値から、f変換形式で出力するか、e変換（あるいはE変換）形式で出力するかを決めdouble型データを出力する。 変換されたデータの指数が、-4より小さい、または、有効桁数を指定する精度以上の場合、e変換（あるいはE変換）形式に変換する。 変換後、小数点以下の末尾の0は除去される。 また、小数点以下に文字がなければ小数点も除去される。	double	精度の指定は、変換されたデータの最大有効桁数を示す。
c	int型データをunsigned char型データとし、そのデータに対応する文字に変換する。	int	精度の指定は無効となる。
s	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null文字まで、あるいは、精度で指定された文字数分、出力する（ただし、null文字は出力されない）。	char へのポインタ	精度の指定は、出力する文字数を示す。精度を省略すると、データが指す文字列のnull文字までの文字が出力される（ただし、null文字は出力されない）。
p	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する。	void へのポインタ	精度の指定は無効となる。
n（変換なし）	データはint型へのポインタ型とみなされ、データが指すメモリ領域にいままで出力したデータの文字数を設定する。	intへのポインタ	精度の指定は無効となる。
%（変換なし）	%を出力する。	なし	精度の指定は無効となる。

fputc

入出力関数

ストリーム入出力用ファイルへ１文字出力します。

書式

```
#include <stdio.h>
int fputc ( int c, FILE *fp );
    c;                          /* 出力する文字 */
    fp;                         /* FILE構造体へのポインタ */
```

戻り値

出力した文字c　：正常終了

EOF　：エラー

解説

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用ファイルに出力します。

関数の戻り値だけでは、入出力エラー発生の有無やファイルの終了の情報が得られない場合、そのファイルの持っているエラー指示子（エラーが発生したか否かを示す）やファイル終了指示子（ファイルが終了したか否かを示す）を参照すれば知ることができます。ストリームファイルはそれぞれ、エラー指示子、ファイル終了指示子というデータを保持しています。エラー指示子はferror関数、ファイル終了指示子はfeof関数によって参照できます。

fputs

入出力関数

ストリーム入出力用ファイルへ文字列を出力します。

書式

```
#include <stdio.h>
int fputs(char *s, FILE *fp);
    s;                          /* 出力する文字列へのポインタ */
    fp;                         /* FILE構造体へのポインタ */
```

戻り値

0	：正常終了
0以外	：エラー

解説

fputs関数は、sが指す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に出力します。ただし文字列の最後のnull文字は出力されません。

関数の戻り値だけでは、入出力エラー発生の有無やファイルの終了の情報が得られない場合、そのファイルの持っているエラー指示子（エラーが発生したか否かを示す）やファイル終了指示子（ファイルが終了したか否かを示す）を参照すれば知ることができます。ストリームファイルはそれぞれ、エラー指示子、ファイル終了指示子というデータを保持しています。エラー指示子はferror関数、ファイル終了指示子はfeof関数によって参照できます。

fread

入出力関数

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。

書式

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t n, FILE *fp);
    ptr;                    /* データ格納領域へのポインタ */
    size;                   /* 1メンバのバイト数 */
    n;                      /* 読み出すメンバの数 */
    fp;                     /* FILE構造体へのポインタ */
```

戻り値

読み出しに成功したメンバ数（通常 n と同じ値）：正常終了

n より小さい値　　：ファイルの終了時やエラー発生時（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）

解説

fread 関数は、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、ファイルポインタ fpが示すストリーム入出力用ファイルから読み込んでptrが指す記憶域に格納します。

size もしくはnが0のとき、戻り値として0を返し、ptr の指す記憶域の内容は変化しません。また、エラーが発生した場合、またはメンバの途中までしか読み出せなかった場合、そのファイルの位置指示子の値は保証されません。

free

標準処理関数

指定された記憶域を開放します。

書式

```
#include <stdlib.h>
void free (void *ptr);
    ptr;                          /* 解放する記憶域の先頭アドレス */
```

戻り値

なし

解説

free 関数は、ptr が指す記憶域を解放し、再度割り当てて使用できるようにします。ptr が NULL であるとき、何もしません。

free関数で解放しようとした記憶域、またはrealloc関数のptrが指す記憶域が、calloc、malloc、realloc関数で割り当てられた記憶域でないとき、または、既に free、realloc関数によって解放されていたときの動作は保証されません。

freopen

入出力関数

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。

書式

```
#include <stdio.h>
FILE *freopen ( const char *fname, const char *mode, FILE *fp );
    fname;                  /* ファイル名 */
    mode;                   /* ファイルアクセスモード */
    fp;                     /* FILE構造体へのポインタ */
```

戻り値

オープンしたストリームへのポインタ	：正常終了
NULL ポインタ	：エラー

解説

freopen関数は、現在オープンされているストリーム入出力用ファイルfpをクローズし、同じfpの指すFILE構造体を再使用して新しいファイルfnameをストリーム入出力用にオープンします。modeにはファイルのアクセスモードを、fopen関数で使用するアクセスモードの中から選んで設定してください。

frexp

数学関数

浮動小数点数を（0.5，1.0）の値と２の累乗の積に分解します。

書式

```
#include <math.h>
double frexp(double x, int *e);
    x;                          /* 浮動小数点数 */
    e;                          /* 指数部(整数値)を格納する記憶域へのポインタ */
```

戻り値

係数部ｍの値

解説

frexp関数は、x=ｍ＊2^nが成り立つような2の累乗部(n)とその係数部(ｍ)を求めます。ここで、係数部ｍの絶対値は1.0より小さく、0.5以上の値となります。累乗値ｎは、引数ｅが指す記憶域に格納され、係数部ｍは戻り値として返されます。なお、ｘが0.0のときｍとｎは共に0.0になります。

fscanf

入出力関数

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。

書式

```
#include <stdio.h>
int fscanf(FILE *fp, const char *control, ...);
    fp;                         /* FILE構造体へのポインタ（入力データへのポインタ） */
    control;                    /* 書式を示す文字列へのポインタ */
    ...;                        /* 各入力データの格納先へのポインタ（可変個引数列） */
```

戻り値

変換し代入を行った引数の数	：正常終了（この数に、%n 変換、* による代入抑止は含まれません）
EOF	：最初の変換処理（%% は除く）が成功する前に入力データが終了した。または、エラーが発生した。

解説

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テキストデータを入力し、書式controlに従って交換および編集し、その結果を可変個引数列で示す記憶域へ格納します。

入力データ（fscanf関数の場合、ファイル内のテキスト）は、1つ以上の空白、水平タブ（'\t'）、または改行文字（'\n'）によって複数のトークンに区切られます。また、空白類文字自体は読み飛ばされます。例えば、入力データ"　ABCE　abcd"　は、"ABCD"と"abcd"　の2トークンと認識され、それぞれ変換後に可変個引数列で示す記憶域に格納されます（この場合、可変個引数列には2つの引数が必要です）。

以下より、書式の詳細について示します（scanf関数、sscanf関数で扱う書式もこれと同様です）。

＜書式を示す文字列の構成＞

書式を示す文字列（control で指定する）は、通常文字列、変換仕様列から構成されます。変換仕様文字列に含まれる変換指定子の数と順番は、各入力データの格納先へのポインタを示している、可変個引数列で列挙する引数（ 以下、引数）の数と順番に対応しています。

■通常文字列

空白類文字でなく、かつ、%で始まる文字列（変換仕様文字列）でないもの。通常文字を指定すると、入力データ（fscanf 関数の場合、ファイル内のテキスト）から、それと一致する文字が入力されます。実際に入力されたデータ内に一致しない文字があれば、その文字は入力ストリームに残されます。

■変換仕様文字列

%で始まる文字列。入力データの変換方法を指定します。文字列は % を先頭に、以下の変換指定要素から必要なものを組み合わせて指定します。

・*

・フィールド幅

・引数のサイズ指定

・変換指定子

＜変換仕様文字列の形式＞

変換仕様文字列は % を先頭に次の形式で指定します（[ ]:省略可能）。

```
%[*][ フィールド幅 ][ 引数のサイズ指定 ] 変換指定子
```

例： `%2d%f` ――― %+ フィールド幅（2）+ 変換指定子（d）+%+ 変換指定子（f）

各要素は連続して記述します（スペースで区切らない）。変換仕様文字列に対して、可変個引数列の引数が足りない場合、動作は保証されません。変換仕様文字列より引数の数が多い場合、余分な引数はすべて無視されます。

＜変換仕様文字列の各要素の詳細＞

変換仕様文字列の各要素について、その機能と指定方法を以下に示します。

■ *（アスタリスク）

変換指定子の前に * を付けると、入力データ中の対応トークンは読み込まれますが、対応する引数への書き込みが抑制されます。

■フィールド幅

入力データの最大読み込み幅（文字数）を 10 進数で指定します。

■引数のサイズ指定

変換指示子がd、i、o、u、x、X、e、E、f、g、G、nの場合、対応する引数のサイズを、修飾子h、l、Lで指定できます。修飾子は変換指示子の直前に書きます。サイズ指定の種類とその意味を下表に示します。

サイズ指定	該当変換指定子	対応する引数に対して指定するサイズ
h	d,i,o,u,x,X	shortintあるいはunsignedshortint
	n	shortintへのポインタ
l（エル）	d,i,o,u,x,X,n	longintあるいはunsigned longint
	n	long intへのポインタ
L	e,E,f,g,G	longdouble

変換指子がd、i、o、u、x、X、e、E、f、g、G、n以外の場合、本指定は無視されます。

■変換指定子

入力データをどのような形式に変換するかを指定します。

fscanf関数は、変換指定子がc、[、n、%でない限り、変換前に空白類文字でない文字（この文字はフィールド幅の対象にはなりません）まで読み飛ばします。

変換中に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見つけると、その文字を読まずに残して処理を終了します。変換中に、指定されたフィールド幅に達した場合は処理を終了します。

以下に、変換指定子と変換方式を示します。ここに挙げていない文字を変換指定子として指定した場合、動作は保証されません。なお、「対応する引数の型」は、h、l、Lによるサイズ指定がないときのものです。

（fscanf:変換指定子　1/2）

変換指定子	変換の方式	対応する引数の型
d	10進数字の文字列を整数型データに変換する。	intへのポインタ
i	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する。 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、文字列を整数型データに変換する。 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、文字列を整数型データに変換する。	intへのポインタ
o	8進数字の文字列を整数型データに変換する。	unsignedintへのポインタ
u	符号なしの10進数字の文字列を整数型データに変換する。	unsignedintへのポインタ
x X	16進数字の文字列を整数型データに変換する。 xとXは同じ仕様。	unsignedintへのポインタ
s	空白、水平タブ、改行文字を読み出すまでをひとつの文字列として変換する。文字列の最後にはnull文字を付加する（変換されたデータの格納域には、null文字を付加した結果の文字列が格納できるサイズが必要)。	charへのポインタ
c	1文字を入力する。このとき、入力する文字がchar型空白類文字であっても読み飛ばされない。 フィールド幅が指定されている場合、その指定分の文字が読み込まれる。したがって、このとき、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。	charへのポインタ
e E f g G	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する。 入力引数の形式はstrtod関数で表現できる浮動小数点数。	floatへのポインタ
p	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する。	voidへのポインタ
n	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る。	intへのポインタ

（fscanf:変換指定子　2/2）

変換指定子	変換の方式	対応する引数の型
[	[　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 ^ でないときは、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 ^ のときは､^ 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 null 文字を付加する(変換したデータを設定する文字列は､null 文字を含めて格納できるサイズが必要である)。	char へのポインタ
%	% を読み出す（変換も引数への書き込みもなし)。	引数なし

fseek

入出力関数

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します。

書式

```
#include <stdio.h>
int fseek(FILE *fp, long offset, int type);
    fp;                         /* FILE構造体へのポインタ */
    offset;                     /* typeによる指定位置からのオフセット */
    type;                       /* オフセットの種類 */
```

戻り値

0	：正常終了
0以外	：エラー

解説

fseek関数は、ファイルポインタfpが示す、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。オフセットの種類は以下のとおりです。

type引数	意味	offset引数
SEEK_SET	ファイルの先頭	0、正の値
SEEK_CUR	現在の読み書き位置	負の値、0、正の値
SEEK_END	ファイルの最後	0、負の値

テキストファイルの場合、type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。

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。

fsetpos

入出力関数

ファイルストリーム上の位置を変更します。

書式

```
#include <stdio.h>
int fsetpos(FILE*stream, const fpos_t*pos);
    stream;                     /* FILE構造体へのポインタ */
    pos;                        /* 変更するファイルストリームの位置 */
```

戻り値

0	：正常終了
0以外	：エラー（errnoにエラー番号を返します）

解説

fsetpos関数は、streamで指定されたファイルストリームの位置をposが指す記憶域で指定した位置に変更します。posは、fgetposで返された値を指定します。

ftell

入出力関数

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。

書式

```
#include <stdio.h>
long ftell(FILE *fp);
    fp;                         /* FILE構造体へのポインタ */
```

戻り値

ファイルの現在の読み書き位置

解説

ftell関数は、ファイルポインタfpが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、戻り値として返します。

fwrite

入出力関数

記憶域からストリーム入出力用ファイルへデータを出力します。

書式

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *fp);
    ptr;                    /* データ格納領域へのポインタ */
    size;                   /* 1メンバのバイト数 */
    n;                      /* 書き込むメンバの数 */
    fp;                     /* FILE構造体へのポインタ */
```

戻り値

実際に出力に成功したメンバの数（通常 n と同じ値）	：正常終了
n より小さい値	：エラー発生時

解説

fwrite関数は、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、ptrの指す記憶域から読み出してファイルポインタ fpが示すストリーム入出力用ファイルに出力します。

getc

入出力関数

ストリーム入出力用ファイルから１文字入力します。

書式

```
#include <stdio.h>
int getc(FILE *fp);          /* ファイルからの1文字入力 */
    fp;                      /* FILE構造体へのポインタ */
```

戻り値

入力した１文字	：正常終了
EOF	：ファイルの終了時およびエラー発生時

解説

getc関数は、ファイルポインタfpが示すストリーム入出力用ファイルから１文字入力し、その１文字を返します。

関数の戻り値だけでは、入出力エラー発生の有無やファイルの終了の情報が得られない場合、そのファイルの持っているエラー指示子（エラーが発生したか否かを示す）やファイル終了指示子（ファイルが終了したか否かを示す）を参照すれば知ることができます。ストリームファイルはそれぞれ、エラー指示子、ファイル終了指示子というデータを保持しています。エラー指示子はferror関数、ファイル終了指示子はfeof関数によって参照できます。

getchar

入出力関数

標準入力（stdin）から１文字入力します。

書式

```
#include <stdio.h>
int getchar(void);                    /* 標準入力からの1文字入力 */
```

戻り値

入力した１文字	：正常終了
EOF	：ファイルの終了時およびエラー発生時

解説

getchar関数は、getc(stdin)関数と同じものです。

関数の戻り値だけでは、入出力エラー発生の有無やファイルの終了の情報が得られない場合、そのファイルの持っているエラー指示子（エラーが発生したか否かを示す）やファイル終了指示子（ファイルが終了したか否かを示す）を参照すれば知ることができます。ストリームファイルはそれぞれ、エラー指示子、ファイル終了指示子というデータを保持しています。エラー指示子はferror関数、ファイル終了指示子はfeof関数によって参照できます。

getenv

標準処理関数

環境変数の内容を取得します。

書式

```
#include <stdlib.h>
char *getenv ( const char *name );
    name;                          /* 環境変数名 */
```

戻り値

変数の定義する文字列へのポインタ	：name で指定した変数が見つかった
NULL	：name で指定した変数が見つからなかった

解説

getenv 関数は、環境リストを検索し name で指定された変数を探します。

注意

getenv関数を使用するためには、getenv関数自身を作成する必要があります（第11章の表11.2参照）。

gets

入出力関数

標準入力（stdin）から文字列を入力します。

書式

```
#include <stdio.h>
char *gets ( char *s );          /* 標準入力からの文字列入力 */
    s;                           /* データの格納領域へのポインタ */
```

戻り値

文字列を格納する記憶域へのポインタs　：正常終了

NULL　：ファイル終了時およびエラー発生時

※またsが指すデータ格納域の内容は、標準入力ファイルの終了時には変化しませんが、エラー発生時にはその内容は保証されません。

解説

gets関数は、標準入力(stdin)から1行分のデータを入力し、sが指す記憶域に格納します。1行分のデータとは、入力開始から改行文字(またはファイルの終了)までの文字列をいいます。入力した文字列の改行文字は捨てられ、最後にnull文字が付加されます。

関数の戻り値だけでは、入出力エラー発生の有無やファイルの終了の情報が得られない場合、そのファイルの持っているエラー指示子（エラーが発生したか否かを示す）やファイル終了指示子（ファイルが終了したか否かを示す）を参照すれば知ることができます。ストリームファイルはそれぞれ、エラー指示子、ファイル終了指示子というデータを保持しています。エラー指示子はferror関数、ファイル終了指示子はfeof関数によって参照できます。

gmtime

日付・時刻操作関数

暦時間を世界標準時に変換します。

書式

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

戻り値

変換した結果（tm型構造体へのポインタ）：正常終了
NULL　：timerを世界標準時に変換できなかった

解説

gmtime関数は、timerで示す暦時間を世界標準時（UTC）に変換し、tm型(broken-down time)に変換します。

isalnum

文字操作関数

英字または10進数字かどうかを判定します。

書式

```
#include  < ctype.h >
int  isalnum ( int c );
   c;                              /* 判定する文字 */
```

戻り値

0以外の値　：cが英数字（'A'-'Z','a'-'z','0'-'9'）
0　：cが英数字以外

解説

cが英数字ならば0以外の値を返し、英数字でなければ0を返します。英数字とは以下の文字を指します。

・英大文字　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
・英小文字　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
・10進数字　0 1 2 3 4 5 6 7 8 9

注意

cの値がunsigned char型で表現できる範囲に含まれず、かつEOFでない場合、この関数の動作は保証されません。

isalpha

文字操作関数

英字かどうかを判定します。

書式

```
#include  < ctype.h >
int  isalpha ( int c );
   c;                          /* 判定する文字 */
```

戻り値

0以外の値	：cが英字('A'-'Z','a'-'z')
0	：cが英字以外

解説

cが英字ならば0以外の値を返し、英字でなければ0を返します。英字とは以下の文字を指します。

・英大文字　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

・英小文字　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

注意

cの値がunsigned char型で表現できる範囲に含まれず、かつEOFでない場合、この関数の動作は保証されません。

iscntrl

文字操作関数

制御文字かどうかを判定します。

書式

```
#include  < ctype.h >
int  iscntrl ( int c );
   c;                          /* 判定する文字 */
```

戻り値

0以外の値	：cが制御文字
0	：cが制御文字以外

解説

cが制御文字ならば0以外の値を返し、制御文字でなければ0を返します。制御文字とは印字文字以外の文字を指します。印字文字とは、ディスプレイに表示可能な文字で、ASCIIコード0x20～0x7Eに対応するものです。

注意

cの値がunsigned char型で表現できる範囲に含まれず、かつEOFでない場合、この関数の動作は保証されません。

isdigit

文字操作関数

10 進数字かどうかを判定します。

書式

```
#include <ctype.h>
int isdigit(int c);
    c;                          /* 判定する文字 */
```

戻り値

0 以外の値	: c が 10 進数字
0	: c が 10 進数字以外

解説

c が 10 進数字ならば 0 以外の値を返し、10 進数字でなければ 0 を返します。10 進数字とは以下の文字を指します。

・10 進数字　0 1 2 3 4 5 6 7 8 9

注意

c の値が unsigned char 型で表現できる範囲に含まれず、かつ EOF でない場合、この関数の動作は保証されません。

isgraph

文字操作関数

空白を除く印字文字かどうかを判定します。

書式

```
#include <ctype.h>
int isgraph(int c);
    c;                          /* 判定する文字 */
```

戻り値

0 以外の値	: c が空白を除く印字文字
0	: c が空白を除く印字文字以外

解説

c が空白を除く印字文字ならば 0 以外の値を返し、空白を除く印字文字でなければ 0 を返します。空白を除く印字文字とは、ディスプレイに表示可能な文字で、ASCII コード 0x21 ～ 0x7E に対応するものです。

注意

c の値が unsigned char 型で表現できる範囲に含まれず、かつ EOF でない場合、この関数の動作は保証されません。

islower

文字操作関数

英小文字かどうかを判定します。

書式

```
#include <ctype.h>
int islower(int c);
    c;                          /* 判定する文字 */
```

戻り値

0以外の値	：cが英小文字
0	：cが英小文字以外

解説 cが英小文字ならば0以外の値を返し、英小文字でなければ0を返します。英小文字とは以下の文字を指します。

・英小文字　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

注意 cの値がunsigned char型で表現できる範囲に含まれず、かつEOFでない場合、この関数の動作は保証されません。

isprint

文字操作関数

空白を含む印字文字かどうかを判定します。

書式

```
#include <ctype.h>
int isprint(int c);
    c;                          /* 判定する文字 */
```

戻り値

0以外の値	：cが空白を含む印字文字
0	：cが空白を含む印字文字以外

解説 cが空白を含む印字文字ならば0以外の値を返し、空白を含む印字文字でなければ0を返します。印字文字とは、ディスプレイに表示可能な文字で、ASCIIコード0x20～0x7Eに対応するものです。

注意 cの値がunsigned char型で表現できる範囲に含まれず、かつEOFでない場合、この関数の動作は保証されません。

ispunct

文字操作関数

特殊文字かどうかを判定します。

書式

```
#include <ctype.h>
int ispunct(int c);
    c;                          /* 判定する文字 */
```

戻り値

0以外の値	：cが特殊文字
0	：cが特殊文字以外

解説 cが特殊文字ならば0以外の値を返し、特殊文字でなければ0を返します。特殊文字とは、空白、英大文字、英小文字、10進数字をそれぞれ除いた任意の印字文字です。

注意 cの値がunsigned char型で表現できる範囲に含まれず、かつEOFでない場合、この関数の動作は保証されません。

isspace

文字操作関数

空白類文字かどうかを判定します。

書式

```
#include <ctype.h>
int isspace(int c);
    c;                          /* 判定する文字 */
```

戻り値

0以外の値	：cが空白類文字
0	：cが空白類文字以外

解説 cが空白類文字ならば0以外の値を返し、空白類文字でなければ0を返します。空白類文字とは以下の文字を指します。

・空白類文字 空白( )、書式送り(\f)、改行(\n)、復帰(\r)、水平タブ(\t)、垂直タブ(\v)

注意 cの値がunsigned char型で表現できる範囲に含まれず、かつEOFでない場合、この関数の動作は保証されません。

isupper

文字操作関数

英大文字かどうかを判定します。

書式

```
#include <ctype.h>
int isupper(int c);
    c;                          /* 判定する文字 */
```

戻り値

0以外の値	：cが英大文字
0	：cが英大文字以外

解説

cが英大文字ならば0以外の値を返し、英大文字でなければ0を返します。英大文字とは以下の文字を指します。

・英大文字　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

注意

cの値がunsigned char型で表現できる範囲に含まれず、かつEOFでない場合、この関数の動作は保証されません。

isxdigit

文字操作関数

16進数字かどうかを判定します。

書式

```
#include <ctype.h>
int isxdigit(int c);
    c;                          /* 判定する文字 */
```

戻り値

0以外の値	：cが16進数字
0	：cが16進数字以外

解説

cが16進数字ならば0以外の値を返し、16進数字でなければ0を返します。16進数字とは以下の文字を指します。

・16進数字　0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

注意

cの値がunsigned char型で表現できる範囲に含まれず、かつEOFでない場合、この関数の動作は保証されません。

labs

標準処理関数

long 型整数の絶対値を計算します。

書式

```
#include <stdlib.h>
    long labs ( long j );
    j;                                  /* 絶対値を求める整数 */
```

戻り値

演算結果の絶対値

解説

labs関数はlong型整数の絶対値を計算します。演算結果がlong型で表現できないときの動作は保証されません。

ldexp

数学関数

浮動小数点数と 2 の累乗の乗算を計算します。

書式

```
#include <math.h>
double ldexp ( double x, int f );
    x;                                  /* 浮動小数点数 */
    f;                                  /* 指数(整数型) */
```

戻り値

x＊2^fの演算結果

解説

ldexp関数は、x＊2^fの値を計算し、結果を返します。

演算結果がオーバーフローのときは、errno に ERANGE の値を設定し、戻り値に HUGE_VAL（負の場合は -HUGE_VAL）返します。

ldiv

標準処理関数

long 型整数の除算の商と余りを計算します。

書式

```
#include <stdlib.h>
ldiv_t ldiv ( long number, long denom );
    number;                    /* 被除数 */
    denom;                     /* 除数 */
```

戻り値

演算結果の結果の商と余り

解説

ldiv 関数は、number を denom で除算した結果の商と余りを計算します。戻り値は ldiv_t 構造体で、そのメンバ quot に商、rem に余りが入ります。

localeconv

ロケール操作関数

lconv型構造体を初期化します。

書式

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

戻り値

初期化したlconv型構造体へのポインタ

解説

localeconv関数は、lconv型構造体を初期化します。

注意

ロケールは "C" 環境のみサポートしています。
localeconv関数を使用するためには、この関数が(関数内部で)呼び出している_get_core,_rel_core関数を作成する必要があります

localtime

日付・時刻操作関数

暦時間を現在の現地時間に変換します。

書式

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

戻り値 変換した結果（tm 型構造体へのポインタ）

解説 localtime関数は、timerで示す暦時間を現地時間（local time）に変換して、tm型に変換します。

注意 ロケールは "C" 環境のみサポートしています。
localtime関数を使用するためには、この関数が（関数内部で）呼び出しているgetenv関数を作成する必要があります。

log

数学関数

浮動小数点数の自然対数を計算します。

書式

```
#include <math.h>
double log(double x);
    x;                              /* 浮動小数点数 */
```

戻り値 x の自然対数

解説 log 関数は x の自然対数を計算し、その結果を返します。x の値が負のとき errno に EDOM の値を設定します。x の値が 0.0 のとき errno に ERANGE の値を設定します。

log10

数学関数

浮動小数点数の 10 を底をする対数を計算します。

書式

```
#include <math.h>
double log10(double x);
    x;                          /* 浮動小数点数 */
```

戻り値

x の常用対数

解説

log10 関数は x の常用対数を計算し、その結果を返します。x の値が負のとき errno に EDOM の値を設定します。x の値が 0.0 のとき errno に ERANGE の値を設定します。

longjmp

プログラムの制御移動関数

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。

書式

```
#include <setjmp.h>
void longjmp(jmp_buf env, int ret);
    env;                        /* 実行環境が記憶されている変数 */
    ret;                        /* setjmp関数への戻り値 */
```

戻り値

なし

解説

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。このときlongjmp関数のretがsetjmp関数の戻り値として返ります。longjmp関数はリターンしません。

longjmp関数は、ジャンプ先のsetjmp関数を実行する前に実行された場合、あるいはsetjmp関数を呼び出した関数が既にreturn文を実行している場合、その動作は保証されません。

注意

longjmp 関数では、汎用レジスタの退避／復帰が行われます。アキュムレータ（ACC）および制御レジスタの退避／復帰は行われません。

malloc

標準処理関数

記憶域を割り当てます。

書式

```
#include <stdlib.h>
void *malloc(size_t size);
    size;                          /* 割り当てる記憶域のバイト数 */
```

戻り値

割り当てられた記憶域の先頭のアドレス	：正常終了
NULL	：エラー（記憶域の割り当てができなかった。引数のいずれかが0）

解説

malloc関数は、sizeで示されるバイト分の記憶域を割り当てます。

注意

malloc関数では、記憶領域が割り当てられる毎に、その記憶領域にプラスして 12 バイトの領域が確保されます。この領域には、記憶領域の割り当て情報が格納されます。

mblen

標準処理関数

マルチバイト文字のバイト数を求めます。

書式

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
    s;                          /* マルチバイト文字へのポインタ */
    n;                          /* 検査バイト数 */
```

戻り値

＜s がNULL の場合＞

0 以外	：マルチバイト文字セットが状態に依存する。
0	：マルチバイト文字セットが状態に依存しない。

＜s がNULL でない場合＞

バイト数	：sが指すマルチバイト文字列のバイト数。
0	：sがnull 文字を指している。
-1	：sが指す文字列がマルチバイト文字列でない。

解説

mblen 関数は、s が示すマルチバイト文字列のバイト数を返します。なお、mbtowc 関数が、シフト状態に影響を受けなければ mblen は以下の例と同値です。

例: mbtowc((wchar_t *)0, s, n);

注意

現在、マルチバイト文字に対する最大値は 3 バイトです。
ロケールは"C"の他、”euc ”、”sjis ”、”utf8 ”の環境をサポートしています。

mbstowcs

標準処理関数

マルチバイト文字列をワイド文字列に変換します。

書式

```
#include <stdlib.h>
size_t mbstowcs ( wchar_t *pwcs, const char *s, size_t n);
    pwcs;                       /* ワイド文字列の格納バッファ */
    s;                          /* マルチバイト文字列へのポインタ */
    n;                          /* 変換するワイド文字数 */
```

戻り値

変換した文字数	:正常終了（ただし、終端null文字は文字数に含まない）
-1	:変換中にマルチバイト文字でない文字を検出したときは-1をsize_tにキャストして返します

解説

mbstowcs関数は、sで示されるマルチバイト文字列をそれに対応するワイド文字の文字列にnで示される文字数分変換し、結果をpwcsの示すバッファに返します。

注意

現在、マルチバイト文字に対する最大値は3バイトです。
ロケールは"C"の他、”euc”、”sjis”、”utf8”の環境をサポートしています。

mbtowc

標準処理関数

マルチバイト文字をワイド文字に変換します。

書式

```
#include <stdlib.h>
int mbtowc ( wchar_t *pwc, const char *s, size_t n );
    pwc;                    /* ワイド文字の格納バッファ */
    s;                      /* マルチバイト文字へのポインタ */
    n;                      /* 検査バイト数 */
```

戻り値

変換マルチバイト文字数	：pwc、ｓが共にNULLでない
0	：sがnull文字を指している
-1	：ｓがマルチバイト文字を正しく指していない

※ｓがNULLのとき、または、pwcとｓが共にNULLのときは、その時点でのマルチバイト文字セットが状態に依存すれば0以外を返し、そうでなければ0を返します。

解説

mbtowc関数は、ｓが示すマルチバイト文字に対して以下のような処理を行います。

・pwcおよびｓがNULL以外のとき：
　ｓが指し示すところからｎバイト分のマルチバイト文字をワイド文字に変換します。

・pwcがNULLでｓがNULL以外のとき：
　mblen関数と同じ動作をします。

・ｓがNULL、またはpwcとｓが共にNULLのとき：
　その時点でのマルチバイト文字セットが状態に依存するかどうかをチェックします。

注意

現在、マルチバイト文字に対する最大値は3バイトです。
ロケールは"C"の他、”euc”、”sjis”、”utf8”の環境をサポートしています。

memchr

文字列操作関数

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。

書式

```
#include <string.h>
void *memchr ( const void *s, int c, size_t n );
    s;                          /* 検索を行う記憶域へのポインタ */
    c;                          /* 検索する文字 */
    n;                          /* 検索を行う文字数 */
```

戻り値

文字位置へのポインタ	：検索の結果文字が見つかった
NULL	：検索の結果文字が見つからなかった

解説

memchr 関数は、指定された記憶域の先頭から n 文字の中で最初に現れた文字 c と同一文字の位置へのポインタを戻り値として返します。

memcmp

文字列操作関数

二つの記憶域を比較します。

書式

```
#include <string.h>
int memcmp ( const void *s1,const void *s2, size_t n );
    s1;                         /* 比較される記憶域へのポインタ */
    s2;                         /* 比較する記憶域へのポインタ */
    n;                          /* 比較する記憶域の文字数 */
```

戻り値

正の値	:s1の内容 > s2の内容
0	:s1の内容 == s2の内容
負の値	:s1の内容 < s2の内容

解説

memcmp 関数は、s1 で指定された記憶域と s2 で指定された記憶域の最初の n 文字分の内容を比較します。

memcpy

文字列操作関数

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。

書式

```
#include <string.h>
void *memcpy ( void *s1, const void *s2, size_t n );
    s1;                         /* 複写先の記憶域へのポインタ */
    s2;                         /* 複写元の記憶域へのポインタ */
    n;                          /* 複写する文字数 */
```

戻り値　複写先の記憶域へのポインタ

解説　memcpy 関数は、複写元の記憶域の内容を指定された大きさ分、記憶域に複写します。

memmove

文字列操作関数

複写元の記憶域の内容を指定した大きさ分、複写先の記憶域に移動します。

書式

```
#include<string.h>
void *memmove ( void *s1, const void *s2, size_t n );
    s1;                         /* 複写先（格納バッファ） */
    s2;                         /* 複写元 */
    n;                          /* 複写するバイト数 */
```

戻り値　s1

解説　memmove 関数は、s2 の内容の先頭から n バイト分 s1 に複写します。memmove 関数は、s1 と s2 のオブジェクト（メモリ領域）が重複しても複写は正常に行われます（ただし、重複部分の、複写元のデータは失われます）。複写した結果、s1 のバッファ領域を超えるような場合の動作は保証されません。

memset

文字列操作関数

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。

書式

```
#include <string.h>
void *memset ( void *s, int c, size_t n );
    s;                          /* 文字が設定される記憶域へのポインタ */
    c;                          /* 設定する文字 */
    n;                          /* 設定する文字数 */
```

戻り値　文字を設定した記憶域へのポインタs

解説　memset 関数は、記憶域 s に n 文字分、文字 c を設定します。

mktime

日付・時刻操作関数

tm 型構造体で示される日時を暦時間に変換します。

書式

```
#include < time.h >
time_t mktime ( struct tm *timeptr );
    timeptr;                    /* tm型構造体へのポインタ */
```

戻り値

変換結果	：正常に暦時間に変換終了
-1	：変換ができなかった（-1をtime_t型にキャストして返します）

解説　mktime関数は、timeptrで示される tm型構造体の要素別の時間（broken-down time）を暦時間に変換します。なお、mktime関数ではtimeptrで指定された tm 型構造体を以下のように取り扱います。

・tm 型構造体の tm_wday と tm_yday のオリジナルの値（読み込んだときの値）は無視します。
・timeptrの tm型構造体の内容は、暦時間に合うように更新されます。

modf

数学関数

浮動小数点数を整数部分と小数部分に分解します。

書式

```
#include <math.h>
double modf ( double x, double *i );
    x;                          /* 浮動小数点数 */
    i;                          /* 整数部分を格納する記憶域を指すポインタ */
```

戻り値

x の小数部分　　：正常終了

解説

modf 関数は、x の値を小数部と整数部に分け、小数部を返します。整数部は i が指す記憶域に格納します。

perror

入出力関数

標準エラー出力（ 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。

書式

```
#include <stdio.h>
void perror ( const char *s );
    s;                                  /* エラーメッセージへのポインタ */
```

戻り値

なし

解説

perror関数は、標準エラー出力(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。

pow

数学関数

浮動小数点数の累乗を計算します。

書式

```
#include <math.h>
double pow(double x, double y);
    x,y;                        /* 浮動小数点数 */
```

戻り値

x^y の値	：正常終了
HUGE_VAL	：計算結果がオーバーフロー

解説

pow 関数は、x の y 乗の値を返します。x の値が 0.0 で、かつ y の値が 0.0 以下のとき、あるいは x の値が負で y の値が整数値でないとき、errno に EDOM を設定します。また、計算結果がオーバフローした場合、errno に ERANGE を設定します。

printf

入出力関数

データを書式に従って変換し、標準出力(stdout)へ出力します。

書式

```
#include <stdio.h>
intprintf(constchar*control,...);
    control;                    /* 書式を示す文字列へのポインタ */
    ...;                        /* 出力データを示す可変個引数列 */
```

戻り値

出力した文字数	：正常終了
負の値	：エラー

解説

printf 関数は、書式 control に従って、可変引数列を変換および編集し、標準出力ファイル（stdout）へ出力します。書式 control の詳細は fprintf 関数の解説を参照してください。

putc

入出力関数

ストリーム入出力ファイルへ１文字出力します。

書式

```
#include <stdio.h>
int putc ( int c, FILE *fp );
    c;                          /* 出力する文字 */
    fp;                         /* FILE構造体へのポインタ */
```

戻り値

出力した文字c　：正常終了

EOF　：エラー

解説

putcマクロは、文字cをファイルポインタfpの示すストリーム入出力用ファイルに出力します。

関数の戻り値だけでは、入出力エラー発生の有無やファイルの終了の情報が得られない場合、そのファイルの持っているエラー指示子（エラーが発生したか否かを示す）やファイル終了指示子（ファイルが終了したか否かを示す）を参照すれば知ることができます。ストリームファイルはそれぞれ、エラー指示子、ファイル終了指示子というデータを保持しています。エラー指示子はferror関数、ファイル終了指示子はfeof関数によって参照できます。

putchar

入出力関数

標準出力（stdout）へ１文字出力します。

書式

```
#include <stdio.h>
int putchar(int c);
    c;                          /* 出力する文字 */
```

戻り値

出力した文字ｃ	：正常終了
ＥＯＦ	：エラー

解説

putcharマクロは､putc(c,stdout)関数と同じものです。

関数の戻り値だけでは、入出力エラー発生の有無やファイルの終了の情報が得られない場合、そのファイルの持っているエラー指示子（エラーが発生したか否かを示す）やファイル終了指示子（ファイルが終了したか否かを示す）を参照すれば知ることができます。ストリームファイルはそれぞれ、エラー指示子、ファイル終了指示子というデータを保持しています。エラー指示子はferror関数、ファイル終了指示子はfeof関数によって参照できます。

puts

入出力関数

標準出力（stdout）へ文字列を出力します。

書式

```
#include <stdio.h>
int puts(char *s);
    s;                          /* 出力する文字列へのポインタ */
```

戻り値

0	：正常終了
0以外	：エラー

解説

puts関数は、sが指す文字列を標準出力(stdout)に出力します。このとき、文字列の最後のnull文字は、改行文字に置き換えて出力します。

関数の戻り値だけでは、入出力エラー発生の有無やファイルの終了の情報が得られない場合、そのファイルの持っているエラー指示子（エラーが発生したか否かを示す）やファイル終了指示子（ファイルが終了したか否かを示す）を参照すれば知ることができます。ストリームファイルはそれぞれ、エラー指示子、ファイル終了指示子というデータを保持しています。エラー指示子はferror関数、ファイル終了指示子はfeof関数によって参照できます。

qsort

標準処理関数

ソートを行います。

書式

```
#include <stdlib.h>
void qsort ( const void *base, size_t nmemb, size_t size,
        int ( *compar )( const void *, const void *) );
    base;                       /* ソート対象となるテーブルへのポインタ */
    nmemb;                      /* ソート対象のメンバの数 */
    size;                       /* ソート対象のメンバのバイト数 */
    compar ;                    /* 比較を行う関数へのポインタ */
```

戻り値　なし

解説　qsort 関数は、base の指すテーブルのデータをソートします。データを並べる順序は、比較を行う関数へのポインタによって指定します。この関数の仕様は、bsearch関数の比較関数と同じです。

raise

シグナル処理関数

プログラムに対してシグナルを送ります。

書式

```
#include <signal.h>
int raise( int sig);
    sig;                        /* 発生させるシグナル */
```

戻り値

0	：正常終了
0 以外	：エラー

解説　raise関数は、プログラムに対してsigで示すシグナルを送ります。

注意　raise関数を使用するためには、raise関数自身を作成する必要があります（第11章の表11.2参照）。

rand

標準処理関数

0 から RAND_MAX の間の擬似乱数整数を生成します。

書式

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

戻り値

生成した擬似乱数整数　　：正常終了

解説

rand 関数は、0 から RAND_MAX の間の擬似乱数整数を生成します。

realloc

標準処理関数

記憶域の大きさを指定された大きさに変更します。

書式

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
    ptr;                        /* 変更する記憶域の先頭アドレス */
    size;                       /* 変更後の記憶域のバイト数 */
```

戻り値

割り当てられた記憶域の先頭のアドレス	：記憶域の割り当てができた
NULL	：エラー（記憶域の割り当てができなかった。 または引数のいずれかが 0。）

解説

realloc関数は、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に変更します。変更後の記憶域が変更前より小さい場合、変更後の記憶域の大きさまでの内容は変化しません。

free関数で解放しようとした記憶域、またはrealloc関数のptrが指す記憶域が、calloc、malloc、realloc関数で割り当てられた記憶域でないとき、または、既に free、realloc関数によって解放されていたときの動作は保証されません。

注意

realloc関数では、記憶領域が割り当てられる毎に、その記憶領域にプラスして 12 バイトの領域が確保されます。この領域には、記憶領域の割り当て情報が格納されます。

remove

入出力関数

ファイルを削除します。

書式

```
#include <stdio.h>
int remove ( const char *FILE filename );
    FILEfilename;                    /* 削除するファイル名 */
```

戻り値

0	：削除に成功した
0以外	：エラー

解説 remove関数は、filenameで指定したファイルを削除します。filenameで指定したファイルがオープンされていても削除します。

注意 オープンされているファイルを変更したり、削除できるかどうかは動作環境に依存します。remove関数を使用するためには、remove関数自身を作成する必要があります（第11章の表11.2参照）。

rename

入出力関数

ファイル名を変更します。

書式

```
#include <stdio.h>
int rename ( const char *old, const char *new );
    old;                             /* 元のファイル名 */
    new;                             /* 新しいファイル名 */
```

戻り値

0	：ファイル名が変更できた
0以外	：エラー

解説 rename関数は、oldで指定したファイル名をnewで指定したファイル名に変更します。oldで指定したファイルがオープンされてもファイル名を変更します。newで指定したファイルが既に存在してもファイル名を変更します。

注意 オープンされているファイルを変更したり、削除できるかどうかは動作環境に依存します。rename関数を使用するためには、rename関数自身を作成する必要があります（第11章の表11.2参照）。

rewind

入出力関数

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。

書式

```
#include <stdio.h>
void rewind ( FILE *fp );
    fp;                              /* FILE構造体へのポインタ */
```

戻り値

なし

解説

rewind関数は、ファイルポインタfpが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。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。

scanf

入出力関数

標準入力(stdin)からデータを入力し、書式に従って変換します。

書式

```
#include <stdio.h>
int scanf(const char *control, ...);
    control;                    /* 書式を示す文字列へのポインタ */
    ...;                        /* 各入力データの格納先へのポインタ（可変個引数列）*/
```

戻り値

変換し代入を行った引数の数	：正常終了（この数には、%n 変換、および * による代入抑止は含まれません）
EOF	：最初の変換処理（%% は除く）が成功する前に入力データが終了した。またはエラーが発生した。

解説

scanf関数は、標準入力（stdin）からデータを入力し、書式controlに従って交換および編集し、その結果を可変個引数列の指す記憶域へ格納します。書式controlの詳細はfscanf関数の解説を参照してください。

setbuf

入出力関数

ストリーム入出力用のバッファをユーザープログラム側で定義して設定します。

書式

```
#include <stdio.h>
void setbuf ( FILE *fp, char *buf );
    fp;                         /* FILE構造体へのポインタ */
    buf;                        /* バッファへのポインタ */
```

戻り値

なし

解説

setbuf関数は、fpの示すストリーム入出力用ファイルのバッファを、デフォルトのバッファ[注)]から、bufの指す記憶域に変更します。バッファサイズは変わりません。バッファサイズも変更する場合はsetvbuf関数を使用してください。

bufにNULLを設定すると、バッファは使用されません（バッファリングされない）。これはsetvbuf関数の第3引数type（バッファの管理方式）に_IONBFを指定した場合と同じです。

注）fopenまたはfreopen関数によるファイルオープン時に自動的に割り当てられるバッファ。そのバッファサイズ（BUFSIZ）はstdio.hで定義されています。

setjmp

プログラムの制御移動関数

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。

書式

```
#include <setjmp.h>
int setjmp(jmp_buf env);
    env;                          /* 実行環境が記憶されている変数 */
```

戻り値

0	：setjmp関数が実行された。envには実行環境が記憶される。
longjmp関数の第2引数ret	：longjmp関数より制御が移された。ただし、longjmp関数の第2引数retが0の場合はこの戻り値は1となる。

解説

setjmp関数は、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。setjmp関数はlongjmp関数と組み合わせて用い、関数外へのグローバルなジャンプを実現することができます。通常の関数呼び出しやリターンの規約を用いずに、以前に呼び出したルーチンへエラー処理の実行制御を渡す場合等に使用します。

setjmp関数を複雑な式から呼び出す場合、式の評価の途中結果等の現在の実行環境の一部が失われる可能性があります。setjmp関数はsetjmp関数の結果と定数式の比較という形態だけで使用し、複雑な式の中では呼び出さないようにしてください。

注意

setjmp関数では、汎用レジスタの退避／復帰が行われます。アキュムレータ（ACC）および制御レジスタの退避／復帰は行われません。

setlocale

ロケール操作関数

ロケール情報の設定、検索をします。

書式

```
#include<locale.h>
char*setlocale(intcategory,constchar*locale);
    category;                   /* 設定するロケール情報部門 */
    locale;                     /* 設定するロケール */
```

戻り値

localeに設定可能な文字列へのポインタ　：正常終了（locale引数が返されるわけではありません）
NULL　：category、localeのどちらも当てはまるものがない

解説

ロケールは "C" 環境のみサポートしています。

setlocale関数は、categoryで示されるロケール情報部門をlocaleで指定したロケールに設定します。localeがNULLのときは、現在のロケール情報を求めます。そのときのプログラムのロケール情報は変化しません。以下にcategoryに指定できるマクロを示します。

category	意味
LC_ALL	すべてのロケール情報を設定、検索します。
LC_COLLATE	strcoll関数,strxfrm関数に影響を与える情報を設定、検索します。
LC_CTYPE	isdigit関数とisxdigit関数を除くすべての文字操作関数とマルチバイトを扱う関数すべてに影響を与える情報を設定、検索します。
LC_MONETARY	localeconv関数が返す通貨情報に影響を与える情報を設定、検索します。
LC_NUMERIC	入出力関数と文字列操作関数で使用される小数点、およびlocaleconv関数が返す通貨情報以外の情報に影響を与える情報を設定、検索します。
LC_TIME	strftime関数に影響を与える情報を設定、検索します。

setvbuf

入出力関数

ストリーム入出力用のバッファをユーザープログラム側で定義し、さらにバッファの使用方法を設定します。

書式

```
#include <stdio.h>
int setvbuf(FILE *fp, char *buf, int type, size_t size);
    fp;                                 /* FILE構造体へのポインタ */
    buf;                                /* バッファへのポインタ */
    type;                               /* バッファの管理方式 */
    size;                               /* バッファサイズ */
```

戻り値

0	：新しいバッファが割り当てられた
0以外	：エラー（引数が不適切）

解説

setvbuf関数は、fpの示すストリーム入出力用ファイルのバッファを、デフォルトのバッファ[注)]から、bufの指す記憶域に変更します。また、バッファサイズをsizeに、バッファの管理方式をtypeに変更します。管理方式は以下に示す3通りに設定できます。

type	意味
_IOFBF	フルバッファリング。入出力処理はすべてバッファを使用して行います。バッファがいっぱいになった場合かフラッシュ処理された場合にデータがバッファから取り出されます。
_IOLBF	行バッファリング。入出力処理は行単位でバッファを使用して行います。バッファがいっぱいになった場合、改行文字が書き込まれた場合、またはフラッシュ処理された場合にデータがバッファから取り出されます。また、バッファが空のとき、データがバッファへ格納されます。
_IONBF	バッファリングなし。入出力処理はバッファを使用せず、ストリームに読み書きする単位で行います。

注意

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処理が行われるまでの間で使用してください。また、ファイルをクローズする前にバッファを解放してはいけません。

注）fopenまたはfreopen関数によるファイルオープン時に自動的に割り当てられるバッファ。そのバッファサイズ（BUFSIZ）はstdio.hで定義されています。

signal

シグナル処理関数

シグナルに応答する関数を設定します。

書式

```
#include <signal.h>
void (*signal(int sig,void(*func)(int)))(int);
    sig;                        /* シグナルの値 */
    func;                       /* シグナルを検出したときに実行される関数 */
```

戻り値

funcの以前（最近）の値	：正常終了
SIG_ERR	：エラー（errnoに適切な値を返します）

解説

signal関数は、sigで指定したシグナルを検出したときにfuncで示される処理ハンドラを呼出します。

funcで指定するハンドラには、ユーザーで定義する以外に以下の特殊な2つのマクロがあります。

- SIG_DFL デフォルトのシグナル処理を実行します。
- SIG_IGN シグナルを無視します。

注意

signal関数を使用する場合は、signal関数自身を作成する必要があります（第11章の表11.2参照）。

sin

数学関数

浮動小数点数のラディアン値の正弦を計算します。

書式

```
#include <math.h>
double sin(double x);
    x;                          /* 浮動小数点数（ラディアン単位） */
```

戻り値 引数の計算値

解説 sin関数はxの正弦を計算します。

sinh

数学関数

浮動小数点数の双曲線正弦を計算します。

書式

```
#include <math.h>
double sinh(double x);
    x;                          /* 浮動小数点数 */
```

戻り値 引数の計算値

解説 sinh関数はxの双曲線正弦を計算します。関数の計算結果がdouble型の値として表せないとき、errnoにERANGEの値を設定します。また、計算結果がオーバフローのときは、戻り値にHUGE_VAL（負の場合は-HUGE_VAL）を返します。

sprintf

入出力関数

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。

書式

```
#include <stdio.h>
int sprintf(char *s,const char *control,...);
    s;                          /* データを出力する記憶域へのポインタ */
    control;                    /* 書式を示す文字列へのポインタ */
    ...;                        /* 出力データを示す可変個引数列 */
```

戻り値

出力した文字数

※エラーを表す戻り値はありません。

解説

sprintf 関数は、書式 control に従って、可変個引数列で示す各データを変換および編集し、s の指す記憶域へ出力します。書式 control の詳細は、fprintf 関数の解説を参照してください。

sqrt

数学関数

浮動小数点数の正の平方根を計算します。

書式

```
#include <math.h>
double sqrt(double x);
    x;                          /* 浮動小数点数 */
```

戻り値

x の正の平方根

解説

sqrt 関数は、x の正の平方根を計算します。x の値が負のとき errno に EDOM の値を設定します。

srand

標準処理関数

rand 関数で生成する擬似乱数数列の初期値を設定します。

書式

```
#include <stdlib.h>
void srand(unsigned int seed);
    seed;                                   /* 擬似乱数列生成の初期値 */
```

戻り値

なし

解説

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ときに、再度srand関数で同じ値の初期値を設定すると、擬似乱数列は繰り返し生成されることになります。

srand関数よりもrand関数が先に呼ばれた場合、擬似乱数列生成の初期値として1が設定されます。

sscanf

入出力関数

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。

書式

```
#include <stdio.h>
int sscanf(char s, const char *control, ...);
    s;                          /* 入力データを格納した記憶域へのポインタ */
    control;                    /* 書式を示す文字列へのポインタ */
    ...;                        /* 各入力データの格納先へのポインタ（可変個引数列） */
```

戻り値

変換し代入を行った引数の数	：正常終了（この数には、%n 変換 ,* による代入抑止は含まれません）
EOF	：最初の変換処理（%% は除く）が成功する前に入力データが終了した。またはエラーが発生した。

解説

sscanf関数は、sの指す記憶域からデータを入力し、書式controlに従って交換および編集し、その結果を可変個引数列で示す記憶域へ格納します。書式controlの詳細はfscanf関数の解説を参照してください。

strcat

文字列操作関数

文字列の後に、文字列を連結します。

書式

```
#include <string.h>
char *strcat ( char *s1, const char *s2 );
    s1;                                  /* 連結される文字列へのポインタ */
    s2;                                  /* 連結する文字列へのポインタ */
```

戻り値　連結後の文字列へのポインタs1

解説　strcat関数は、文字列s1の後に文字列s2を連結します。このとき、文字列s1の最後のnull文字はs2の先頭文字で置き換えられ、連結後の文字列s2の最後には必ずnull文字が付きます。なおstrncat関数では、連結する文字数を引数nで指定できます。

strchr

文字列操作関数

文字列において、指定された１文字が最初に現われる位置を検索します。

書式

```
#include <string.h>
char *strchr ( const char *s, int c );
    s;                                   /* 検索を行う文字列へのポインタ */
    c;                                   /* 検索する文字 */
```

戻り値

検索の結果見つかった文字へのポインタ	：文字が見つかった
NULL	：文字が見つからなかった

解説　strchr関数は、文字列s中から文字cを検索し、それが最初に現れた位置へのポインタを戻り値として返します。

strcmp

文字列操作関数

二つの文字列を比較します。

書式

```
#include <string.h>
int strcmp(const void *s1, const void *s2);
    s1;                         /* 比較される文字列へのポインタ */
    s2;                         /* 比較する文字列へのポインタ */
```

戻り値

正の値	:s1の内容 > s2の内容
0	:s1の内容 == s2の内容
負の値	:s1の内容 < s2の内容

解説

strcmp関数は、s1で指定された文字列と、s2で指定された文字列の内容を比較し、その結果を戻り値として設定します。

strcoll

文字列操作関数

現在のロケールに基づいて指定された２つの文字列を比較します。

書式

```
#include <string.h>
int strcoll(const char *s1,const char *s2);
    s1;                         /* 比較文字列１ */
    s2;                         /* 比較文字列２ */
```

戻り値

正の値	:s1の内容 > s2の内容
0	:s1の内容 == s2の内容
負の値	:s1の内容 < s2の内容

解説

ロケールは"C"環境のみサポートしています。

strcoll関数は、現在のロケールが示すLC_COLLATE（locale.hに定義）に基づいて（変換して）s1とs2を比較します。

strcpy

文字列操作関数

複写元の文字列の内容を、複写先の記憶域にnull文字も含めて複写します。

書式

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
    s1;                                 /* 複写先の記憶域へのポインタ */
    s2;                                 /* 複写元の文字列へのポインタ */
```

戻り値　複写先の記憶域へのポインタ

解説　strcpy関数は、複写元の文字列を指定された記憶域にnull文字も含めて複写します。

strcspn

文字列操作関数

文字列において、指定外の文字が先頭から何文字続くかを求めます。

書式

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
    s1;                                 /* 調べられる文字列へのポインタ */
    s2;                                 /* 指定文字から構成される文字列へのポインタ */
```

戻り値　検索結果の文字列長

解説　strcspn関数は、文字列s1の先頭から、文字列s2を構成する文字のいずれも出現しない部分が何文字分続くかを調べ、その長さを戻り値として返します。

strerror

文字列操作関数

エラーメッセージを返します。

書式

```
#include <string.h>
char *strerror(int e);
    e;                              /* エラー番号 */
```

戻り値　エラー番号 e に対応するエラーメッセージへのポインタ

解説　strerror関数は、エラー番号eに対応するエラーメッセージへのポインタを戻り値として返します。

strftime

日付・時刻操作関数

tm 型構造体で示される日時を書式に従って変換します。

書式

```
#include<time.h>
size_t strftime(char *s, size_t maxsize, const char *format, const struct tm *timeptr);
    s;                          /* 変換文字列格納バッファ */
    maxsize;                    /* 変換文字数の最大値 */
    format;                     /* 書式文字列 */
    timeptr;                    /* tm型構造体へのポインタ */
```

戻り値

書き込んだ文字数	：sに書き込んだ文字数がmaxsize以下
0	：書き込んだ文字数がmaxsizeを超えた（このときsの内容は不定）

解説

ロケールは"C"環境のみサポートしています。

strftime関数は、timerptrで示されるtm型構造体のbroken-down timeをformatで指定された書式に従って変換しsに変換結果を格納します。そのときの変換結果は、最大maxsize数までsに格納されます。

以下にformat中で使用できる各変換指定子を示します。これら以外の変換指定子を指定した場合の動作は不定です。

（続く）

変換指定子	意味
%a	ロケールでの曜日の略語
%A	ロケールでの曜日の名前
%b	ロケールでの月の略語
%B	ロケールでの月の名前
%c	ロケールでの適切な日付と時刻の表現
%d	月の中での日付を整数で表したもの（01から31）
%H	時間（24時間制）を整数で表したもの（00から23）
%I	時間（12時間制）を整数で表したもの（01から12）
%j	年初からの日数を整数で表したもの（001から366）
%m	月を整数で表したもの（01から12）
%M	分を整数で表したもの（00から59）
%p	12時間を表すそのときのロケールの午前(AM）と午後（PM）
%S	秒を整数で表したもの（00から59）
%U	年初からの週数を整数で表したもの（ただし、日曜を週の最初の日とします(00から53)）
%w	日曜を0（最初の日）としたときの週の中での日を整数で表したもの（0から6）
%W	年初からの週数を整数で表したもの（ただし、月曜を週の最初の日とします(00から53)）
%x	ロケールでの適切な日付の表現
%X	ロケールでの適切な時刻の表現
%y	世紀中での年数を整数で表したもの（00から99）
%Y	西暦を整数で表したもの
%Z	時間帯またはその略語を表したもの（時間帯が特定できないときはNULL文字で表します）
%%	%自身を表します

strlen

文字列操作関数

文字列の長さを計算します。

書式

```
#include <string.h>
size_t strlen(const char *s);
    s;                              /* 長さを求める文字列へのポインタ */
```

戻り値

計算した文字列の文字数

解説

strlen関数は、文字列sの長さを計算します。ただし、文字列の終端を示すnull文字は数えません。

strncat

文字列操作関数

文字列の後に、文字列を指定した文字数分連結します。

書式

```
#include <string.h>
char *strncat ( char *s1, const char *s2, size_t n );
    s1;                         /* 連結される文字列へのポインタ */
    s2;                         /* 連結する文字列へのポインタ */
    n;                          /* 連結する文字数 */
```

戻り値　連結後の文字列へのポインタs1

解説　strncat関数は、文字列s1の後に文字列s2を連結します。このとき、文字列s1の最後のnull文字はs2の先頭文字で置き換えられ、連結後の文字列s2の最後には必ずnull文字が付きます。連結する文字数は引数nで指定できます。

strncmp

文字列操作関数

二つの文字列を指定された文字数分まで比較します。

書式

```
#include <string.h>
int strncmp ( const void *s1, const void *s2, size_t n );
    s1;                         /* 比較される文字列へのポインタ */
    s2;                         /* 比較する文字列へのポインタ */
    n;                          /* 比較する文字数の最大値 */
```

戻り値

正の値	:s1の内容 > s2の内容
0	:s1の内容 == s2の内容
負の値	:s1の内容 < s2の内容

解説　strncmp関数は、s1で指定された文字列と、s2で指定された文字列の内容を、引数nで指定された文字数分比較し、その結果を戻り値として設定します。

strncpy

文字列操作関数

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。

書式

```
#include <string.h>
char *strncpy ( char *s1, const char *s2, size_t n );
    s1;                                  /* 複写先の記憶域へのポインタ */
    s2;                                  /* 複写元の文字列へのポインタ */
    n;                                   /* 複写する文字数 */
```

戻り値

複写先の記憶域へのポインタ

解説

strncpy関数は、複写元の文字列s2を指定された文字数分、指定された記憶域s1に複写します。s2で指定された文字列の長さがn文字より短い場合、n文字になるまでnull文字が付加されます。逆に文字列s2がn文字以上の場合、s1に複写された文字列はnull文字で終了しないことになります。

strpbrk

文字列操作関数

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。

書式

```
#include <string.h>
char *strpbrk ( const char *s1, const char *s2 );
    s1;                              /* 検索を行う文字列へのポインタ */
    s2;                              /* 指定文字から構成される文字列へのポインタ */
```

戻り値

検索の結果見つかった文字へのポインタ　：文字が見つかった
NULL　：文字が見つからなかった

解説

strpbrk関数は、文字列s1内の先頭から、文字列s2を構成する文字のいずれかが最初に現れる位置を検索し、その位置へのポインタを戻り値として返します。

strrchr

文字列操作関数

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

書式

```
#include <string.h>
char *strrchr(const char *s, int c);
    s;                              /* 検索を行う文字列へのポインタ */
    c;                              /* 検索する文字 */
```

戻り値

検索の結果見つかった文字へのポインタ　：文字が見つかった

NULL　：文字が見つからなかった

解説

strrchr関数は、文字列s中から文字cを検索し、それぞれ最初に現れた位置、最後に現れた位置を求め、そのポインタを戻り値として返します。 文字列sの終端を示すnull文字も検索の対象として含みます。

strspn

文字列操作関数

文字列において、指定された文字が先頭から何文字続くかを求めます。

書式

```
#include <string.h>
    size_t strspn(const char *s1, const char *s2);
    s1;                             /* 調べられる文字列へのポインタ */
    s2;                             /* 指定文字から構成される文字列へのポインタ */
```

戻り値

検索結果の文字列長

解説

strspn関数は、文字列s1の先頭から、文字列s2を構成する文字のいずれかと一致する部分が何文字分続くかを調べ、その長さを戻り値として返します。

strstr

文字列操作関数

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。

書式

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
    s1;                         /* 検索を行う文字列へのポインタ */
    s2;                         /* 検索する文字列へのポインタ */
```

戻り値

検索の結果見つかった文字へのポインタ　：文字が見つかった

NULL　：文字が見つからなかった

解説

strstr関数は、文字列s1において、文字列s2が最初に現れる位置を検索し、その位置へのポインタを戻り値として返します。s2 がNULL の場合、戻り値はNULL になります。

strtod

標準処理関数

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。

書式

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
    nptr;                       /* 変換対象文字列へのポインタ */
    endptr;                     /* 読み込みの終了位置 */
```

戻り値

変換された数値

解説

strtod関数は文字列をdouble型の数値に変換します。入力文字列は、変換する型の数値として解釈できる文字の並びであり、数字の一部として解釈できない文字があると、そこで文字列の入力を中止します。endptrの指す領域には、入力文字列のうち数値として解釈できない最初の文字へのポインタを設定します。endptr がNULL の場合、この設定は行われません。

変換結果がオーバーフローあるいはアンダーフローの場合、errno にERANGE の値を設定し、戻り値にHUGE_VAL（負の場合は -HUGE_VAL）、あるいは0 を返します。

strtok

文字列操作関数

文字列をいくつかのトークン（字句）に切り分けます。

書式

```
#include <string.h>
char *strtok( char *s1, const char *s2);
    s1;                         /* 字句に切り分ける文字列へのポインタ */
    s2;                         /* 文字列を切り分けるための文字列へのポインタ */
```

戻り値

切り分けた字句の先頭へのポインタ	：字句の切り分けを行った
NULL	：字句が切り分けられなかった

解説

strtok関数は、文字列s2中の文字を区切り子として、文字列s1をいくつかのトークン（字句）に切り分けます。strtok関数は連続的に呼び出され、最初の呼び出し時は文字列s1の先頭から、2回目以降は以前に切り分けられた字句の次の文字から、文字列s2中の文字によって字句に切り分けます。

2回目以降の呼び出し時は、第1引数に NULL を指定します。第2引数s2の内容は呼び出しの度に変わってもかまいません。切り出された字句の最後にはnull文字が付きます。

strtol

標準処理関数

数を表現する文字列をlong 型の整数値に変換します。

書式

```
#include <stdlib.h>
long strtol (const char *nptr, char **endptr, int base);
    nptr;                       /* 変換対象文字列へのポインタ */
    endptr;                     /* 読み込みの終了位置 */
    base;                       /* 変換の基数 ( 0,2～36 ) */
```

戻り値

変換された値	：変換された
0	：変換できなかった

解説

strtol関数は、ntprで示す文字列を読み込み、baseで指定した基数の数値（long型）に変換します。

strtol関数は、変換対象文字列を先頭から順に読み込み、基数baseの数値を表す数字として解釈できない文字かnull文字を検出した時点で読み込みを中止します。その時指している文字へのポインタをendptrに設定します（endptrがNULLでない場合のみ）。なお、変換対象文字列の始めの空白文字（先頭から空白文字以外を検出するまでの文字）は無視され、変換の対象になりません。

baseが0の場合、nptrの指す文字列の先頭の文字から基数を判断します（4.1.3.2「整数型定数」参照）。baseの値が2～36の間の場合、base値が変換するときの基数となります。

変換対象文字列中のaからz（およびAからZ）の文字はそれぞれ10から35の値に対応付けられ、baseの値以上の文字は解釈できない文字と認識されます。符号の後の0、および、baseが16のときの0x(0X)は無視されます。

変換できなかった場合、戻り値として0を返し、endptrにnptrがセットされます（endptrがNULLでない場合のみ）。

変換後の値がオーバーフローを起こす（longで表現できない）場合は、戻り値にLONG_MAX（負の場合はLONG_MIN）を返し、errnoに ERANGEが設定されます。

strtoul

標準処理関数

数を表現する文字列をunsigned long型の整数値に変換します。

書式

```
#include <stdlib.h>
unsigned long int strtoul (const char *nptr, char **endptr, int base);
    nptr;                       /* 変換対象文字列へのポインタ */
    endptr;                     /* 読み込みの終了位置 */
    base;                       /* 変換の基数 ( 0,2～36 ) */
```

戻り値

変換された値	：変換された
0	：変換できなかった

解説

strtoul関数は、nptrで示す文字列をbaseで指定した基数の数値(unsigned long型)に変換します。

strtoul関数は、変換対象文字列を先頭から順に読み込み、基数baseの数値を表す数字として解釈できない文字かnull文字を検出した時点で読み込みを中止します。その時指している文字へのポインタをendptrに設定します（endptrがNULLでない場合のみ)。なお、変換対象文字列の始めの空白文字（先頭から空白文字以外を検出するまでの文字）は無視され、変換の対象になりません。

baseが0の場合、nptrの指す文字列の先頭の文字から基数を判断します。baseの値が2～36の間の場合、base値が変換するときの基数となります。

変換対象文字列中のａからｚ（およびＡからＺ）の文字はそれぞれ10から35の値に対応付けられ、baseの値以上の文字は解釈できない文字と認識されます。符号の後の0、および、baseが16のときの0x(0X)は無視されます。

変換できなかった場合、戻り値として0を返し、endptrにnptrがセットされます（endptrがNULLでない場合のみ)。

変換後の値がオーバーフローを起こす（unsigned longで表現できない）場合は、戻り値にLONG_MAX（負の場合はLONG_MIN）を返し、errnoに ERANGEが設定されます。

strxfrm

文字列操作関数

現在のロケールに基づいて文字列を変換します。

書式

```
#include <string.h>
size_t strxfrm(char *s1,const char *s2,size_t n);
    s1;                         /* 変換結果文字列格納バッファ */
    s2;                         /* 変換文字列 */
    n;                          /* 変換文字数 */
```

戻り値

変換後の文字列のバイト数（文字列の最後のnull文字は含みません）

※戻り値がnより大きいときs1の内容は不定です。

解説

ロケールは"C"環境のみサポートしています。

strxfrm関数は、現在のロケールが示すLC_COLLATEに基づいてs2の文字列を最初の文字からnバイト分変換しs1に変換結果を格納します。

system

標準処理関数

コマンド文字列をコマンドプロセッサによって実行されるホスト環境に渡します。

書式

```
#include <stdlib.h>
int system(const char *string);
    string;                     /* コマンド文字列 */
```

戻り値

0以外　　：stringがNULLで、コマンドプロセッサが存在する

0　　：stringがNULLで、コマンドプロセッサが存在しない

※stringがNULLでないときの戻り値は、処理系に依存します。

解説

system関数は、stringで指定された文字列をコマンドプロセッサによって実行されるホスト環境に渡します。また、stringにNULLを指定した場合は、コマンドプロセッサが存在するかどうかを調べます。

注意

system関数を使用するには、system関数自身を作成する必要があります（第11章の表11.2参照）

tan

数学関数

浮動小数点数のラディアン値の正接を計算します。

書式

```
#include <math.h>
double tan(double x);
    x;                          /*  浮動小数点数（ラディアン単位） */
```

戻り値　引数の計算値

解説　tan関数はxの正接を計算します。関数の計算結果がdouble型の値として表せないとき、errnoにERANGEの値を設定します。また、計算結果がオーバーフローの場合、戻り値にHUGE_VAL（負の場合 -HUGE_VAL）を返します。

tanh

数学関数

浮動小数点数の双曲線正接を計算します。

書式

```
#include <math.h>
double tanh(double x);
    x;                          /* 浮動小数点数 */
```

戻り値　引数の計算値

解説　tanh関数はｘの双曲線正接を計算します。また、計算結果がオーバーフローの場合、戻り値にHUGE_VAL（負の場合 -HUGE_VAL）を返します。

time

日付・時刻操作関数

現在の暦時間を求めます。

書式

```
#include <time.h>
time_t time(time_t *timer);
    timer;                              /* 暦時間 */
```

戻り値

現在の暦時間	：timerがNULLでない
-1	：現在の暦時間が得られないとき（-1をtime_t型にキャストして返します）

解説 time関数は、現在の暦時間をtimerの指し示すバッファに結果を書き込みます。

注意 time関数を使用するには、time関数自身を作成する必要があります（第11章の表11.2参照）。

tmpfile

入出力関数

テンポラリファイルを生成します。

書式

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

戻り値

生成したファイルのストリームへのポインタ	：ファイルが生成できた
NULL	：ファイルが生成できなかった

解説 tmpfile関数は、テンポラリファイルを生成します。生成されたファイルは、バイナリファイルの更新モード(wb+)でオープンされ、ファイルをクローズしたりプログラムが終了したときに自動的に削除されます。

tmpnam

入出力関数

既存しないテンポラリファイル名を生成します。

書式

```
#include <stdio.h>
char *tmpnam ( char *s );
    s;                              /* tmpnamが生成したファイル名を格納するバッファ */
```

戻り値

生成したファイル名 ：ファイルが生成できた
NULL ：tmpnam 関数呼び出しが TMP_MAX 回を超えた

解説

tmpnam 関数は、既存しないファイル名でテンポラリファイル名を生成し、s で示すバッファに生成したファイル名を格納します。s に NULL を指定した場合は、tmpnam 内部のバッファに結果が書き込まれます。なお、このバッファは、tmpnam を呼ぶ度に書き換わります。

tmpnam 関数は TMP_MAX(stdio.h に定義）回まで呼び出せ、TMP_MAX 回を超えた場合は NULL を返します。

s は、最低でも L_tmpnam（stdio.h に定義）サイズのバッファを確保していなければなりません。

tolower

文字操作関数

英大文字を英小文字に変換します。

書式

```
#include <ctype.h>
int tolower(int   c);
    c;                              /* 変換する文字 */
```

戻り値

変換された文字 ：引数として指定した文字 c が条件を満たす場合
文字 c（そのまま） ：引数として指定した文字 c が条件を満たさない場合

解説

tolower 関数は、文字 c が英大文字のとき小文字に変換し、変換した文字を返します。そうでないとき、c を変換せずそのまま返します。

toupper

文字操作関数

英小文字を英大文字に変換します。

書式

```
#include <ctype.h>
int toupper(int   c);
    c;                              /* 変換する文字 */
```

戻り値

変換された文字　　：引数として指定した文字ｃが条件を満たす場合
文字ｃ（そのまま）：引数として指定した文字がｃ条件を満たさない場合

解説

toupper関数は、文字ｃが英小文字のとき大文字に変換し、変換した文字を返します。そうでないとき、ｃを変換せずそのまま返します。

ungetc

入出力関数

ストリーム入出力用ファイルへ１文字を戻します。

書式

```
#include <stdio.h>
int ungetc ( int c, FILE *fp );
    c;                          /* 戻す文字 */
    fp;                         /* FILE構造体へのポインタ */
```

戻り値

戻した文字c	：正常終了（通常）
EOF	：エラー発生時

解説

ungetc関数は、文字cをファイルポインタfpが示すストリーム入出力用ファイルに戻します。また、ここで戻された文字は、fflush、fseek、rewind関数を呼び出さなければ次の入力データとなります。もしこれらの関数を呼び出す事なくungetc関数を2回以上呼び出した場合、その動作は保証されません。なお、ungetc関数が実行されるとファイルに対する現在の位置指示子がひとつ戻されますが、この位置指示子が既にファイルの先頭に位置している場合、位置指示子は保証されません。

関数の戻り値だけでは、入出力エラー発生の有無やファイルの終了の情報が得られない場合、そのファイルの持っているエラー指示子（エラーが発生したか否かを示す）やファイル終了指示子（ファイルが終了したか否かを示す）を参照すれば知ることができます。ストリームファイルはそれぞれ、エラー指示子、ファイル終了指示子というデータを保持しています。エラー指示子はferror関数、ファイル終了指示子はfeof関数によって参照できます。

va_arg（マクロ）

可変個の実引数アクセス関数

可変個の引数から順に引数を取り出します。

書式

```
#include <stdarg.h>
type va_arg(va_list ap, type);
    ap;                         /* 引数リストを指すポインタ */
    type;                       /* 戻り値の型 */
```

戻り値

現在の引数

解説

va_start、va_arg、va_endマクロによって、可変個の引数を持つ関数に対してその引数の参照を可能にします。

第1引数にva_startマクロで初期化したva_list型の変数apを指定し、第2引数に参照する引数の型を指定します。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引数が順次本マクロの戻り値として返されます。

注意

typeに、型変換によってサイズが変わるような型を指定した場合、正しく引数を参照することができません。このような型を指定すると動作は保証されません。

va_end（マクロ）

可変個の実引数アクセス関数

可変個の引数を持つ関数の引数への参照を終了します。

書式

```
#include <stdarg.h>
void va_end (va_list ap);
    ap;                          /* 引数リストを指すポインタ */
```

戻り値　なし

解説　va_start、va_arg、va_endマクロによって、可変個の引数を持つ関数に対してその引数の参照を可能にします。

va_endマクロは引数への参照を終了させます。apはva_startマクロで初期化された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ときは、その関数の動作は保証されません。

va_start(マクロ)

可変個の実引数アクセス関数

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。

書式

```
#include <stdarg.h>
void va_start (va_list ap, parmN);
    ap;                          /* 引数リストを指すポインタ */
    parmN;                       /* 引数の並びの最右端の引数の識別子 */
```

戻り値　なし

解説　va_start、va_arg、va_endマクロによって、可変個の引数を持つ関数に対してその引数の参照を可能にします。

va_startマクロはva_arg、va_endマクロで使用されるapの初期化を行います。 parmNには、外部関数定義における引数の並びの最右端の引数の識別子、即ち「, ...」の直前の識別子を指定します。可変個の名前のない引数を参照するためには、最初にva_startマクロを呼び出す必要があります。

vfprintf

入出力関数

可変個のパラメー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へ出力します。

書式

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE *fp, const char *control, va_list arg);
    fp;                         /* FILE構造体へのポインタ */
    control;                    /* 書式を示す文字列へのポインタ */
    arg;                        /* 引数並びを指すポインタ */
```

戻り値

出力した文字数　：正常終了

負の値　：出力エラーが発生時

解説

vfprintf関数はストリーム入出力用ファイルfpに可変個の引数をフォーマットを指定して出力します。この関数はfprintf関数と同じですが、vfprintf関数の場合、「引数の並び（可変個引数列）」ではなく「引数の並びへのポインタ」を受け取ります。

vfprintf関数は出力した文字列の最後にnull文字を追加しますが、このnull文字は戻り値として数えません。controlで指定する書式仕様はfprintf関数で取り扱う書式仕様と同じです。詳細はfprintf関数の仕様を参照してください。

引数リストを示すargは、va_startおよびva_argマクロによって初期化されていなければなりません。なお、本関数はva_endマクロを呼び出しません。

vprintf

入出力関数

可変個のパラメータリストを書式に従って、標準出力へ出力します。

書式

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *control, va_list arg);
    control;                    /* 書式を示す文字列へのポインタ */
    arg;                        /* 引数並びを指すポインタ */
```

戻り値

出力した文字数	：正常終了
負の値	：出力エラーが発生時

解説

vprintf関数は標準出力（stdout）に可変個の引数をフォーマットを指定して出力します。この関数はprintfと同じですが、vprintf関数の場合、「引数の並び（可変個引数列）」ではなく「引数の並びへのポインタ」を受け取ります。

vprintf関数は出力した文字列の最後にnull文字を追加しますが、このnull文字は戻り値として数えません。controlで指定する書式仕様はfprintf関数で取り扱う書式仕様と同じです。詳細はfprintf関数の仕様を参照してください。

引数リストを示すargは、va_startおよびva_argマクロによって初期化されていなければなりません。なお、本関数はva_endマクロを呼び出しません。

vsprintf

入出力関数

可変個のパラメータリストを書式に従って、指定した記憶域へ出力します。

書式

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *s, const char *control, va_list arg);
    s;                          /* データを出力する記憶域へのポインタ */
    control;                    /* 書式を示す文字列へのポインタ */
    arg;                        /* 引数並びを指すポインタ */
```

戻り値

出力した文字数　：正常終了

負の値　：出力エラーが発生時

解説

vsprintf関数は記憶域sに可変個の引数をフォーマットを指定して出力します。この関数はsprintfと同じですが、vsprintf関数関数の場合、「引数の並び（可変個引数列）」ではなく「引数の並びへのポインタ」を受け取ります。

vsprintf関数は出力した文字列の最後にnull文字を追加しますが、このnull文字は戻り値として数えません。controlで指定する書式仕様は、fprintf関数で取り扱う書式仕様と同じです。詳細はfprintf関数の仕様を参照してください。

引数リストを示すargは、va_startおよびva_argマクロによって初期化されていなければなりません。なお、本関数はva_endマクロを呼び出しません。

wcstombs

標準処理関数

ワイド文字列をマルチバイト文字列に変換します。

書式

```
#include <stdlib.h>
size_t wcstombs ( char *s, const wchar_t *pwcs, size_t n );
    s;                          /* マルチバイト文字列格納バッファ */
    pwcs;                       /* ワイド文字列 */
    n;                          /* 書き込みバイト数 */
```

戻り値

変換した文字数	：正常に変換された（ただし、終端null文字は文字数に含みません）
-1	：変換中に正しくないワイド文字が検出された（-1をsize_tにキャストして返します）

解説

現在、マルチバイト文字に対する最大値は３バイトです。
ロケールは"C"の他、”euc”、”sjis”、”utf8”の環境をサポートしています。

wcstombs関数は、pwcsで示されるワイド文字列をnで示されるバイト数分マルチバイト文字に変換して、その結果をsの指し示すバッファに格納します。

wctomb

標準処理関数

ワイド文字をマルチバイト文字に変換します。

書式

```
#include <stdlib.h>
int wctomb ( char *s, wchar_t wchar );
    s;                          /* マルチバイト文字格納バッファ */
    wchar;                      /* ワイド文字 */
```

戻り値

＜s がNULL の場合＞

0　　：その時点でのマルチバイト文字セットが状態に依存しない。

0 以外　　：その時点でのマルチバイト文字セットが状態に依存する。

＜s がNULL でない場合＞

バイト数　　：変換されたマルチバイト文字のバイト数。

0　　：wchar が0。

-1　　：wchar に対応するマルチバイト文字が存在しない。

解説

現在、マルチバイト文字に対する最大値は3バイトです。

ロケールは"C"の他、”euc ”、”sjis ”、”utf8 ”の環境をサポートしています。

wctomb 関数は、wchar で示されるワイド文字をそれに対応するマルチバイト文字に変換して結果をs が指し示すバッファに格納します。s がNULL の場合、その時点でのマルチバイト文字セットが状態に依存するかどうかをチェックします。

第10章
cc32RのC言語の振舞い

本章では、ANSI 規格が示すところの「未定義の振舞い（undefined behavior)」、「インプリメンテーション依存の振舞い（implementation-defined behavior)」、および「ロケール特有の振舞い（locale-specific behavior)」に対する C/C++ コンパイラ cc32R の C 言語での振舞いについて説明します。各説明は ANSI 規格「ANSI/ISO 9899-1990」の節に対応付けて行います。

ANSI 規格では、「未定義の振舞い」、「インプリメンテーション依存の振舞い」および「ロケール特有の振舞い」を次のように定義しています。

■ 未定義の振舞い

移植性を考えていないプログラムやエラーのあるプログラム構造、エラーのあるデータ、または不定値を含んだオブジェクトなどを使用した場合、ANSI 規格の必要条件として課されていない振舞い。

■ インプリメンテーション依存の振舞い

正しいプログラム構成と正しいデータに対する動作で、そのインプリメンテーションの特性に依存する動作（各種コンパイラごとに規定される動作)。

■ ロケール特有の振舞い

国籍、文化、言語といった地域の習慣に依存した動作。

10.1 ANSI規格未定義の振舞い

ANSI 規格で「未定義の振舞い」扱いとなっている動作は、C/C++ コンパイラでの動作が保証されません。ほとんどの場合、無視する、エラーなどの警告を出す、実行時にエラーが発生する、のいずれかとなる可能性があります。したがって、「未定義の振舞い」に該当しないようなコーディングを行うことが推奨されます。

以下より、「未定義の振舞い」扱いの動作に対する、C/C++ コンパイラ cc32R で予測される振舞い（保証されるものではありません）を示します。「■ ANSI 規格」に続く番号と見出しは、対応する ANSI 規格「ANSI/ISO 9899-1990」の節番号と節タイトルです。

■ ANSI 規格 5.1.1.2　翻訳段階（ソースファイルの終わり）

ソースファイルの最後に改行文字が無い場合、改行文字は自動的に付加されます（ファイルの最後の行は改行文字で終わる必要はありません）。

ソースファイルが前にバックスラッシュのついた改行文字で終わっている場合、バックスラッシュと改行文字が削除されます。" ソースファイルが前処理トークン[注1)] やコメント文の途中で終わっている場合、エラーとなります。

注1) 前処理トークン（詳細は ANSI 規格 6.1 参照）は C ソースファイル内テキストの基本処理単位で次のものがあります：ヘッダファイル名、識別子、前処理数、文字定数、文字列リテラル、演算子、句切り文字、および、左記以外の単一文字（空白を除く）。

■ ANSI 規格 5.2.1　文字セット（文字セット以外の文字）

ソースファイル中に使用できる文字セット以外の文字が出現した場合（ただし、トークンに変換されない前処理トークン、文字定数、文字列リテラル、ヘッダ名、およびコメント文は除く）、ワーニングとなり、該当文字は無視されます。

■ ANSI 規格 5.2.1.2　マルチバイト文字

コメント、文字定数、文字列リテラル以外でマルチバイト文字を使用した場合、本コンパイラにより生成されたコードの動作は、保証されません。また、コメントの終わり「*/」の直前が、シフト状態（初期シフト状態以外）の場合、コメントの終わりを認識できない場合があります。

■ ANSI 規格 6.1　字句要素（引用符の対）

ソース中に対になっていない ' または " が出現した場合、エラーとなります。

■ANSI 規格 6.1.2.1　識別子のスコープ

同一関数内で同じ識別子を複数回ラベルとして使用するとエラーとなります。

カレントスコープ（有効範囲）にない識別子を使用するとエラーとなります。

■ANSI 規格 6.1.2　識別子

同じものを指す識別子において、有意文字以降の文字が異なる場合、ワーニングとなります。

■ANSI 規格 6.1.2.2　識別子のリンケージ

関数を表す同じ識別子を内部識別子と外部識別子の両方で宣言した場合、外部識別子とみなします。ただし、関数定義においてstatic宣言されていれば内部識別子とみなします。関数以外の識別子を内部識別子と外部識別子の両方で定義した場合、内部識別子とみなします。

■ANSI 規格 6.1.2.4　オブジェクトの記憶持続期間

自動記憶持続期間をもつオブジェクト用に予約されていた記憶域がもはや保証されなくなった場合、そのオブジェクトを参照するポインタ値を使用すると、コンパイルエラーにはなりませんがプログラムの動作は保証されません。

■ANSI 規格 6.1.2.6　型の適合、混成型

同じオブジェクトまたは関数に対して2つの宣言があり、それらが適合した型（compatible type）でない場合、エラーとなります。

■ANSI 規格 6.1.3.4　文字定数

文字定数または文字列リテラル中にサポートされていないエスケープシーケンスが出現した場合、ワーニングとなり、バックスラッシュが無視されます（例：'\c' は 'c' と解釈されます）。

■ANSI 規格 6.1.7　ヘッダ名

ヘッダ名中に \ , " , または /* が出現しても、それはファイル名を構成する文字として認識されます（特殊文字として処理されません）。

■ANSI 規格 6.2.1　算術演算

算術型変換の結果、与えられたスペースで表せない結果（精度不足）となった場合近似値をとります。ただし、整数へ変換の場合、小数点以下の桁と、納まりきらなかった上位桁のビットパターンは捨てられます。

■ANSI規格 6.2.2.1　左辺値（lvalue）

初期化式で配列を初期化する以外、不完全な型を左辺値に使用するとエラーとなります。

■ANSI規格 6.2.2.2　void

void型の値を使用してアクセスしたり、void式に暗黙的型変換（voidへの変換を除く）を適応したりすると、エラーとなります。

■ANSI規格 6.3　式

副作用

式のシーケンスポイント間における副作用は不定です。副作用によって演算結果が異なる可能性のあるコードは記述しないようにしてください。

例えば、「*p++=*p+5」というコードは、*p+5がp++よりも前に評価される場合と後に評価される場合があり、*p+5の代入先が不定となります。この場合、目的の処理に応じて、以下のどちらかでコーディングしてください。

```
p=*p+5;
++p;
```

または、

```
*(p+1)=*p+5;
p++;
```

無効演算、定義域エラー

無効な演算（0除算など）の場合、エラーとなります。演算結果が定義域エラー（オーバーフロー、アンダフローなど）の場合、ワーニングとなります。ただし、いずれもコンパイル時点で検出可能な場合にのみ警告されます。また、いずれの場合も演算による動作は保証されません。

■ANSI規格 6.3.2.2　関数呼び出し

関数への引数がvoid式の場合

実引数として、空引数以外のvoid式を指定するとエラーになります。また、空引数（void式）指定時、呼び出した関数に仮引数が1つ以上定義されている場合、関数に渡される値は不定です。

実引数と仮引数の型の不一致

関数プロトタイプ宣言がない場合の関数呼び出しにおいて、その関数が関数宣言の見えないところで定義されており、かつ、拡張後（暗黙の型

変換実行後）の実引数と仮引数の型と一致しない場合、実引数の値は保証されません。例えば、short 宣言された実引数（関数プロトタイプなしのとき int に暗黙的に変換される）が、unsigned int の仮引数をもつ関数に渡される場合はエラーとなり、int の仮引数の関数に渡される場合はエラーにはなりません。

関数プロトタイプと関数定義の型の不一致

関数プロトタイプ宣言が見えている場合の関数呼び出しで、かつ、その関数が宣言と適合する型で定義されていない場合、エラーとなります。

可変個引数のプロトタイプ宣言

可変個引数列を受け付ける関数が、「...」で終わる関数プロトタイプが見えないところで呼び出された場合、可変個引数列部分の実引数は正しく渡されないことがあります。

■ANSI 規格 6.3.3.2　単項演算子（&、＊）

アドレス演算子&および間接参照演算子＊によって、以下のような参照を行った場合、動作は保証されません。

・ 無効な配列を参照した場合

・ null ポインタを参照した場合

・ 有効範囲が終了した自動記憶持続期間をもつオブジェクトを参照した場合

■ANSI 規格 6.3.4　キャスト演算子

関数へのポインタを異なる型の関数へのポインタにキャストし、元の型に適合しない型の関数を呼び出した場合、動作は保証されません。

関数へのポインタは、オブジェクトへのポインタにキャストできます。また、オブジェクトへのポインタは関数へのポインタにキャストできます。

ポインタを整数型（文字型を含む）およびポインタ型以外へキャストすると多くの場合エラーとなります。また、エラーとならない場合でもプログラムの動作は保証されません。

■ANSI 規格 6.3.6　加法演算子

配列へのポインタを加算／減算し、ポインタが配列要素の領域以外を指す結果となっても、ポインタ値は正常に加算／減算されます（エラーとはなりません）。また、ポインタの指す内容を＊演算子で参照した場合、そこに格納されているデータが参照できます。ただし、そのデータは配

列の要素ではないため、プログラムの動作は保証されません。

■ANSI 規格 6.3.7　シフト演算子

シフト演算におけるシフト量の指定が、負数あるいはシフトされる式のビット幅以上の場合、動作は保証されません（動作例：シフト量が負数の場合 、シフト方向が逆転することがある。シフト量がシフトされる式のビット幅以上の場合、型のサイズで表現できる限り、正常にシフトされることがある)。

■ANSI 規格 6.3.8　マクロ置き換え

関係演算子（<,<=,>,>=）で比較されるポインタが、同じ集合体または共用体を指していない場合でも、エラーとはなりませんが動作は保証されません。

■ANSI 規格 6.3.16.1　代入演算子（単純代入＝）

あるオブジェクトを重複するオブジェクトに代入する場合、重複部分のデータは保証されません。

■ANSI 規格 6.5　宣言

リンケージなしで宣言されたオブジェクトが、その宣言が終わっても、あるいは、(そのオブジェクトが初期値をもっているならば）その初期宣言が終わっても、不完全な型である場合、エラーとなります。

■ANSI 規格 6.5.1　記憶クラス指示子

関数が、ブロックスコープにおいて extern 以外の記憶クラス指定子で宣言される場合、動作は保証されません。

■ANSI 規格 6.5.2.1　構造体／共用体の指定子

名前のないメンバ

名前のないメンバだけからなる構造体あるいは共用体を定義した場合、動作は保証されません。

構造体のビットフィールドの型

構造体のビットフィールドの宣言に有効な型は、singed または unsigned の、char、short、int、long です。それ以外の型を宣言した場合、動作は保証されません。

■ANSI 規格 6.5.3　型修飾子

const 以外の左辺値によって const 宣言されたオブジェクトを変更しようとした場合、すなわち、const 宣言された領域をキャストなどで const で

ないかのように処理しようとした場合、動作は保証されません（エラーとならない場合もあります)。

volatile以外の左辺値によってvolatile宣言されたオブジェクトを変更しようとした場合、すなわち、volatile宣言された領域をキャストなどでvolatileでないかのように処理しようとした場合、動作は保証されません（エラーとならない場合もあります)。

■ANSI規格 6.5.7　初期化

初期化されていない自動記憶持続時間をもつオブジェクトを、値を代入する前に使用した場合、エラー等の警告が出ない場合もありますが、その値は不定（indeterminate）です。

次のような場合、エラー等の警告が出ない場合もありますが、動作は保証されません。

・ 静的記憶持続時間をもつ集合体または共用体型オブジェクトが、中カッコ {} で囲まれていない初期値をもつ場合。

・ 自動記憶持続時間をもつ集合体または共用体型オブジェクトが、そのオブジェクトの型の初期化式、あるいは、カッコ {} で囲まれていない初期値のいずれかをもつ場合。

■ANSI規格 6.6.6.4　return文

関数値を参照したが、関数側で値をreturnしなかった場合、参照した関数値は不定値です。

■ANSI規格 6.7　外部定義

外部リンケージをもつ、2つ以上の同一識別子を定義する場合、それらが同一ソース中にあるならばコンパイル時にエラーとなり、複数のソースに散在するならばリンク時にエラーとなります。

■ANSI規格 6.7.1　関数定義

可変個引数を受け付ける関数が、その関数定義の仮引数リストが「...」で終わっていない場合、仮引数リストで宣言した個数以上の引数を渡そうとするとエラーとなります。

■ANSI規格 6.7.2　外部オブジェクト定義

内部リンケージもつ不完全型オブジェクトの識別子が、あいまいな定義で宣言された場合、動作は保証されません（ワーニングとなる場合もあります)。

■ANSI 規格 6.8.1　条件付き取り込み

#if あるいは #elif 前処理指令の展開中に生成されるトークン defined は演算子として扱われます。

■ANSI 規格 6.8.2　ソースファイル取り込み

#include 前処理指令が、二つあるヘッダファイルの形式のどちらにも合わない場合、エラーとなります。

■ANSI 規格 6.8.3　マクロ置換

引数のない関数列マクロ呼び出しはエラーとなります。

マクロコールの実引数の並びの中に、# で始まる行、すなわち前処理命令があれば、それは前処理命令とみなされます。

■ANSI 規格 6.8.3.2　# 演算子（文字列化）

前処理用演算子 # による文字列化の結果、有効な文字列定数にならない場合、動作は保証されません。展開時にエラーとなる場合もあります。

■ANSI 規格 6.8.3.3　## 演算子（トークン結合）

前処理用演算子 ## でトークンを結合した結果、有効な前処理トークンにならない場合、動作は保証されません。例えば、func#1 が展開されると func1 となりますが、func1 が意味のないトークンである場合、コンパイル時にワーニングとなり、リンク時にエラーとなる可能性があります。

■ANSI 規格 6.8.4　#line

展開後の #line 前処理指令が文法に合わなければエラーとなります。このとき、行情報は更新されません。

■ANSI 規格 6.8.8　予約マクロ名

_ _LINE_ _、_ _FILE_ _、_ _DATE_ _、_ _TIME_ _、_ _STDC_ _は、すでに定義済みのマクロで、#define や #undef による定義や定義の取消しを行うとエラーとなります。

■ANSI 規格 7　ライブラリ

memmove 以外のライブラリ関数の使用によって、重複オブジェクトにオブジェクトをコピーしようとした場合、重複部分のデータは保証されません。

■ANSI 規格 7.1.2　標準ヘッダ

外部定義内でのインクルード

C 標準ライブラリで提供される、関数宣言、オブジェクト宣言、型定義、マクロ定義、およびキーワードと同じ名前のマクロ定義は、初回参照前に、対応する標準ヘッダファイルをインクルードする必要があります。参照後にインクルードした場合、正しく動作しません。

予約外部名の再定義

プログラム予約外部名（ヘッダ内の外部名など）を定義した場合の処理はリンカに依存します。

■ANSI 規格 7.1.4　エラー <errno.h>

errno は、マクロおよび外部変数により実現されています。マクロ定義を無効にした場合でも errno へのアクセスは可能です。

■ANSI 規格 7.1.6　共通定義 <stddef.h>

offsetof のマクロの第 2 パラメータに構造体のビットフィールドメンバを指定するとエラーとなります。

■ANSI 規格 7.1.7　ライブラリ関数の使用方法

ライブラリ関数の実引数が無効な値の場合、プログラムの動作は保証されません。

可変個引数を受け付けるライブラリ関数が、ヘッダのインクルードなどによって宣言されていない場合、該当関数は正しく動作しないことがあります。

■ANSI 規格 7.2　診断関数ヘッダ <assert.h>

assert はマクロにより実現されています。関数をアクセスするためにマクロ呼び出しを無効にして assert を呼び出すと、コンパイル時にはワーニングとなり、リンク時には外部シンボルが存在しないためエラーとなります。

■ANSI 規格 7.3　文字操作関数ヘッダ <ctype.h>

文字操作関数への引数が unsigned char または EOF 以外の場合、動作は保証されません。

■ANSI 規格 7.6　大域ジャンプ関数ヘッダ <setjmp.h>

setjmp はマクロ定義を無効にした場合でもエラーとはなりません。

■ANSI 規格 7.6.1.1　setjmp マクロ

setjmp マクロは以下に示す用途で利用が推奨されます。これら以外の用途で利用してもエラーとはなりませんが、複雑な式の中で使用した場合、現在の実行環境の一部（式の評価の途中結果など）が失われる可能性があります。

・ 選択文、繰り返し文、および整数定数式の比較における、オペランドの制御（単項演算子！による暗黙処理など）

・ 選択文や繰り返し文のオペランドの制御

・ 式文（void へのキャストなど）

■ANSI 規格 7.6.2.1　longjmp 関数

setjmp 実行から longjmp 呼び出しまでの間に、volatile 指定されていない自動記憶クラスのオブジェクトが変更された場合、そのオブジェクトの値は保証されません。

longjmp 関数がネストされたシグナルのルーチンから起動した場合、setjmp 関数に戻りますが、それ以降の動作はユーザーが作成する signal 関数（低水準関数）の仕様に依存します。

■ANSI 規格 7.7.1.1　signal 関数

C 標準ライブラリには signal 関数は実装されていません。シグナルに関する処理はユーザーが作成する signal 関数（低水準関数）の仕様に依存します。したがって、次のような場合もユーザーが作成する signal 関数（低水準関数）の仕様に依存します。

・ シグナルが、abort または raise 関数呼び出しの結果として発生する場合。

・ シグナルハンドラが、signal 関数以外の標準ライブラリ関数を呼び出す場合。

・ シグナルハンドラが、volatile sig_atomic_t 型以外の任意の静的オブジェクトを参照する場合。

・ シグナル発生後(abort、raise関数呼び出しの結果を除く)、 errno の値が参照され、対応するシグナルハンドラが値 SIG_ERR を返す signal 関数を呼ぶ場合。

■ANSI 規格 7.8.1　可変個の実引数アクセス関数ヘッダ <stdarg.h >

ある関数（関数 A とする）において、va_arg マクロの引数 ap（可変個引数列）を実引数として呼び出された関数（関数 B とする）の中で、その ap を使って va_arg マクロを呼び出した場合、次のようになります。

・ 関数B（ある関数Aから呼び出された側）では、呼び出された時点でのapが指す可変個引数から参照できます。

・ 関数A（関数Bを呼び出した側）では、関数Bが可変個引数を参照する／しないにかかわらず、関数Bを呼び出した時点のapが指す可変個引数から参照できます。

ただし、apのアドレスを引数にして渡したり、集合体（apが集合体であるとき）を引数として渡した場合、関数Bから戻ってきた後の関数Aのapは関数Bの終了時点からの続きになります。

va_start、va_arg、va_endはマクロにより実現されています。関数をアクセスするためにマクロ呼び出しを無効にしてこれらを呼び出すと、コンパイル時にはワーニングとなり、リンク時には外部シンボルが存在しないためエラーとなります。

■ANSI規格 7.8.1.1　va_startマクロ

va_startマクロの第2引数parmNの型宣言が、レジスタクラス変数、関数型、配列型であるとき、または、規定実引数拡張後の型（引数に対する暗黙の型変換後の型）と合わないとき、動作は保証されません。

■ANSI規格 7.8.1.2　va_argマクロ

va_argを呼び出したとき、処理指定の引数が実際には存在しない場合、動作は保証されません。

va_argを呼び出したとき、処理指定の引数が指定された型でない場合、動作は保証されません。

■ANSI規格 7.8.1.3　va_endマクロ

va_startマクロを呼び出す前にva_endを呼び出してもエラーとはならず正常に動作します。

va_endマクロを呼び出す前に、va_start マクロによって初期化された可変引数リストをもつ関数がreturnしてもエラーにはなりませんが、プログラムの動作は保証されません。

■ANSI規格 7.9.5.2　fflush関数

入力ストリームに対するfflushは無視されます（エラーも返りません）。

■ANSI規格 7.9.5.3　fopen関数

同じストリームに対する入力要求と出力要求の間に、fflush関数またはファイル位置指定関数[注2)]がない場合、入出力動作は保証されません。

注2) ファイル位置指定関数　fgetpos,fseek,fsetpos,ftell,rewind

■ANSI規格 7.9.6　書式指定入出力関数

printf系／scanf系

関数仕様の型と引数リストの対応する数が一致しない場合、および、変換指示子の数より引数の数が少ない場合、エラーとはなりませんが動作は保証されません。変換指示子の数より引数が多い場合、余分な引数は無視されます。

printf系、scanf系関数において無効な変換仕様に対する入出力結果は不定です。ほとんどの場合、エラーメッセージは出力されません。期待どおりに入出力が得られない場合、変換仕様のコーディングが正しい書式かどうか確認してください。

printf系／scanf系　%%変換

printf系またはfscanf系関数の変換仕様%%で、%と%の間に数字以外の文字含む場合、多くの場合、%と%の間の文字が入出力対象となります。例えば、"%abcdefg%" は "abcdefg" に変換されます。

■ANSI規格 7.9.6.1　printf系

修飾子

printf系の変換仕様において、修飾子（サイズ指定文字h、l）が該当変換指定子（o,x,X,e,E,f,g,G）以外の変換指定子の前のｈまたはｌの前に指定された場合、その修飾子は無視されます。

フラグ

printff系の変換仕様において、フラグ#が該当変換指定子
(o,x,X,e,E,f,g,
G ）以外の前に指定された場合、そのフラグは無視されます。

printff系の変換仕様において、フラグ0が該当変換指定子
(d,i,o,u,x,X,e,E,f,g, G ）以外の前に指定された場合、そのフラグは無視されます。

変換結果

集合体、共用体、または集合体か共用体へのポインタが 、printf系の%pおよび%s以外に指定された場合でも正常に動作します。

printf 関数による単一の%変換の変換結果が509文字を超える文字出力となる場合、動作は保証されません。

■ANSI規格 7.9.6.2　scanf系

修飾子

scanf系の変換仕様において、修飾子（サイズ指定文字h、l、L）が以下のように該当変換指定子以外の前に指定された場合、その修飾子は無視されます。

・ d,i,n,o,u,x以外の変換指定子の前のｈまたはｌ

・ e,f, g以外の変換指定子の前のＬ

printf系の%pとの互換

printf系の%p変換の出力形式とscanf系の%pに代入されるアドレス形式には互換性があります。

変換結果の格納領域

scanf系関数による変換結果値を代入する領域が、容量不足あるいは適合しない型である場合、動作は保証されません。

■ANSI規格 7.10.1　文字列変換関数（文字列から数値への変換）

文字列を数値に変換する関数（atof 、atoi 、atol）の変換結果が定義域エラーで表現できない値の場合、定義域最大値（HUGE_VAL、INT_MAXなど）を返します。

■ANSI規格 7.10.3　メモリ管理関数（free関数、realloc関数）

free関数かrealloc関数によって解放された領域を参照した場合、動作は保証されません。

free関数またはrealloc関数の第1引数（解放対象領域へのポインタ）として次のような値を渡した場合、動作は保証されません。

・ calloc、malloc、またはrealloc関数の戻り値（割り当て完了領域へのポインタ）でない値。

・ 以前にfree関数またはrealloc関数によって解放された領域へのポインタ。

■ANSI規格 7.10.4.3　exit関数

プログラムが2回以上exit関数の呼び出しを実行する場合（atexit関数でexit関数登録した場合など）の動作は保証されません。

■ANSI 規格 7.10.6　整数型の算術演算関数

整数型の算術演算関数（abs、div、labs、ldiv）の結果が表現できない場合、その値は保証されません。

■ANSI 規格 7.10.7　マルチバイト文字関数（シフト状態）

マルチバイト文字のシフト状態は存在します。また、標準関数は、呼び出された時点で、常に初期シフト状態と見なして処理します。

■ANSI 規格 7.11.2,7.11.3　複製／連結関数

以下の場合、動作は保証されません。

memcpy、memmove、strcpy、strncpy 関数において、複写先のサイズが複写元のサイズより小さい場合。strcat、strncat 関数において、連結される文字列の格納領域が、連結結果を格納するために十分でない場合。

■ANSI 規格 7.12.3.5　strftime 関数

strftime 関数の時間変換書式 format 中に変換指定子でない文字がある場合、その文字をそのまま出力します。

同じ配列を指していないポインタ同士で減算する場合、全オペランド（ポインタ）の型が適合する型（compatible type）ならばエラーとはなりません。ただし、キャストによって型を適合させた場合、演算結果は保証されません。

実引数と仮引数の数の不一致

関数プロトタイプが見えないところでの関数呼び出しにおいて、実引数と仮引数の数が一致しない場合、不足分の仮引数の値は保証されません。

ワイド文字列リテラルの連結

ワイド文字列リテラル指定時、連結文字列は無効となります。

左辺値によるアクセス

適合しない型をもつ左辺値をオブジェクトに代入した場合、動作は保証されません。

■ANSI 規格 6.1.4　文字リテラル

ワイド文字列リテラルとの混合指定時

通常の文字列リテラル指定時、ワイド文字列リテラルトークンが出現した場合、接頭文字Ｌは無視され、文字列リテラルとして連結されます。
ワイド文字リテラル指定時、文字列の連結指定はできません。

10.2 インプリメンテーション依存の振舞い

ANSI 規格で「インプリメンテーション依存の振舞い」扱いとなっている動作について、C/C++ コンパイラ cc32R における C 言語での振舞いを以下に示します。おもに、エラーメッセージの通知のしかた、識別子として有効な文字の数、整数や浮動小数点のフォーマットなどを規定しています。

「■ANSI 規格」に続く番号と見出しは、対応する ANSI 規格「ANSI/ISO 9899-1990」の節番号と節タイトルです。また、「インプリメンテーション依存の振舞い」扱いとなっている事項を＜＞内に、それに対する cc32R の動作を＜＞の後に示します。

10.2.1 変換（Translation）

■ANSI 規格 5.1.1.3　診断

＜C/C++ コンパイラのメッセージ出力形式＞

C/C++ コンパイラの C コンパイラとしての診断メッセージには、インフォメーションメッセージ、ワーニングメッセージ、エラーメッセージ、重大エラーメッセージがあります。メッセージの出力形式および詳細については、第 12 章「C コンパイラのメッセージ」を参照してください。

10.2.2 環境（Environment）

■ANSI 規格 5.1.2.2.1　プログラムの開始

＜main 関数への引数の意味＞

main 関数に渡される引数は、ユーザーが作成するスタートアッププログラムの仕様に依存します。

■ANSI 規格 5.1.2.3　プログラムの実行

＜対話型デバイスを構成するもの＞

入出力デバイスの動作は、ユーザーが作成する read 関数および write 関数（低水準関数）の仕様に依存します。

10.2.3 識別子（Identifiers）

■ANSI 規格 6.1.2　識別子

＜外部リンケージのない識別子では、先頭から（31 を越えて）何文字まで認識されるか。＞

外部リンケージのない識別子は先頭から 240 文字まで有効です。241 文字以降の文字は無視されます。

＜外部リンケージのある識別子では、先頭から（6 を越えて）何文字まで認識されるか。＞

外部識別子は先頭から 240 文字まで有効です。241 文字以降の文字は無視されます。また、外部識別子は大文字／小文字が区別されます。

＜ case 文は外部リンケージをもつ識別子において有効かどうか。＞

case 文には整数定数式のみ記述できます。

10.2.4 文字（Characters）

■ANSI 規格 5.2.1　文字セット

＜ソース用、および実行用の文字セット（本国際標準において明白に指定されたものを除く）の種類＞

ソース文字セット、実行文字セット共、JIS X 0201,0208 で定義される文字が使用できます。ただし、JIS X 0201 のラテン文字部分は ASCII と見なして処理します。実際の文字コード（エンコード）には、EUC (Expanded Unix Code)、シフト JIS、および UCS-2 (UTF-8 エンコード) を使用できます。

■ANSI 規格 5.2.1.2　マルチバイト文字

＜マルチバイト文字のシフト状態＞

マルチバイト文字のためのシフト状態（マルチバイト文字の始まりと終わりを示す文字列）はありません。

■ANSI 規格 5.2.4.2.1　整数型のサイズ

＜実行用の文字セットの 1 文字のビット数＞

実行文字セットの 1 文字は 8 ビットです。

■ANSI規格 6.1.3.4　文字定数

＜ソース用文字セットの実行用文字セットへのマッピング＞

環境変数M32RKINが utf8 の場合は、JIS X 0201,0208 で定義される文字セットに正規化されます。 それ以外の場合は、ソース文字セット中の文字と、実行文字セットとの配置対応は1対1です。

＜基本実行文字セットやワイド文字定数の拡張文字セットにない文字を含む整数文字定数の値＞

環境変数M32RKOUTが euc,sjis の場合は、最右端の2文字を Big-Endian で連結したものになります。環境変数M32RKOUTがutf8 の場合には、0xffff になります。

＜2文字以上からなる整数文字定数、または、2文字以上のマルチバイトからなるワイド文字定数の値＞

ワイド文字ではない文字定数は、最右端の文字の値となります。ワイド文字定数は、環境変数M32RKOUTにより変化します。

＜マルチバイト文字を、対応するワイド文字（コード）に変換するためのロケール＞

”euc”、”sjis”、”utf8”の各ロケールを用意しています。

■ANSI規格 6.2.1.1　文字と整数

＜charはsigned charとunsigned charのどちらに相当するか。＞

char は生成されるコード上では signed char と同様に振る舞います。ただし互換性のため、文法上は char は signed char とは異なる型として処理します。

10.2.5 整数（Integers）

■ANSI規格 6.1.2.5　型

＜整数型の表現＞

各種整数型データの内部表現および極限値については5.2「文字型・整数型データの内部表現」を参照してください。なお、C/C++ コンパイラは、int と signed int 、short と signed short、long と signed long はそれぞれ同じものと解釈します。また、char と signed char は、文法上は異なる型として処理しますが、生成されるコード上では等価な型として扱います。

■ANSI規格 6.2.1.2　符号付き／符号なし整数

＜値が表現できない場合、整数データをより短い符号付き整数型に変換した結果、または、符号なし整数データを同サイズの符号付き整数型に

次ページ →

変換した結果＞

整数を「よりサイズの小さい符号付き整数」に変換するときは、もとの整数の下位ビット値がそのまま符号付き整数に変換されます。変換後の符号付き整数の最上位ビットは符号ビットです。

符号なし整数を「同一サイズの符号付き整数」に変換するときは、もとの整数の下位ビット値がそのまま符号付き整数に変換されます。

■ANSI 規格 6.3　式

＜符号付き整数のビット演算結果＞

符号付き整数のビット演算は、符号なし整数のように扱われます。

■ANSI 規格 6.3.5　乗法演算子

＜整数の除算結果の剰余の符号＞

余りの符号は被除数の符号と同じです。

■ANSI 規格 6.3.7　ビット単位シフト演算子

＜負の値をもつ符号付き整数型の右シフト＞

符号付きの負の整数型の右シフトは算術シフトです。

10.2.6 浮動小数点（Floating Point）

■ANSI 規格 6.1.2.5　型

＜浮動小数点数型の表現＞

各種浮動小数点型データの内部表現および極限値については、5.3「浮動小数点型の内部表現」を参照してください。

■ANSI 規格 6.2.1.3　浮動小数点数と整数の演算

＜整数が浮動小数点数型に変換され、もとの数値を正確に表現できないときの丸め＞

変換後の浮動小数点数型で表現できる範囲で、もとの値に最も近い値に丸められます。

■ANSI 規格 6.2.1.4　浮動小数点数の演算

＜浮動小数点数型がより小さいサイズの浮動小数点型に変換されたときの丸め＞

変換後の浮動小数点数型で表現できる範囲で、もとの値に最も近い値に丸められます。

10.2.7 配列とポインタ（Arrays and Pointers）

■ANSI 規格 6.3.3.4　sizeof 演算子、7.1.1　ライブラリ用語の定義

＜sizeof 演算子の型 size_t ＞

sizeof 演算子の型 size_t は、unsigned long として定義されています。

■ANSI 規格 6.3.4　キャスト演算子

＜ポインタ型の整数型へのキャスト、またはその逆＞

ポインタを整数に変換、あるいは、整数をポインタに変換するときは、表記を変えずにすべてのビットを使うため、正しい変換となります。

■ANSI 規格 6.3.6　加法演算子、7.1.1　ライブラリ用語の定義

＜ptrdiff_t の型＞

二つのポインタの差を保持する整数の型 ptrdiff_t は、int として定義されています。

10.2.8 レジスタ（Registers）

■ANSI 規格 6.5.1　記憶クラス指定子

＜register 宣言できるオブジェクトの数＞

記憶クラス指定子 register は無視されます。

10.2.9 構造体、共用体、ビットフィールド（Structures, Unions, Enumerations, and Bit-fields）

■ANSI 規格 6.3.2.3　構造体／共用体のメンバ

＜union のメンバが異なる型のメンバによってアクセスされる場合＞

union のメンバに格納されているビットパターンがアクセスされ、その値は、アクセスされたメンバの型に従って解釈されます。

■ANSI規格 6.5.2.1　構造体／共用体指定子

＜構造体のメンバのパディング（データ詰め）とアライメント（整合条件）＞

ビットフィールドのパディングおよびアライメントの詳細は、5.9.2「ビットフィールドの内部表現」を参照してください。

＜「int」型ビットフィールドは「signed int」か「unsigned int」か。＞

通常のint型ビットフィールドはsigned int型ビットフィールドとして取り扱います。

＜ビットフィールドの記憶装置上への割り付け順＞

ビットフィールドは上位ビット側から下位ビット側へ割り付けられます

＜ビットフィールドは記憶領域の境界をまたぐかどうか。＞

1つのビットフィールドが、アラインメント境界をまたいで配置されることはありません。

■ANSI規格 6.5.2.2　列挙型指定子

＜列挙型の値の型＞

列挙型はint型として取り扱います。

10.2.10修飾子（Qualifiers）

■ANSI規格 6.5.3　型修飾子

＜volatile修飾子をもつオブジェクトへのアクセス方法＞

volatileのオブジェクト名を参照するたびに、そのオブジェクトをアクセスします。volatileオブジェクトの最適化は行いません。

10.2.11宣言子（Declarators）

■ANSI規格 6.5.4　宣言子

＜算術演算、構造体、共用体の型が修正可能な宣言子の最大数＞

宣言子の最大数にとくに制限はありません。

10.2.12文（Statements）

■ANSI 規格 6.6.4.2　switch 文

＜switch 文中の case 値の最大数＞

switch 文中の case 値の最大数は、使用可能なメモリ容量に依存します。

10.2.13前処理命令（Preprocessing Directives）

■ANSI 規格 6.8.1　条件付きインクルード

＜条件つきインクルードを制御する定数式中の単文字文字定数の値が、実行文字セット中の同じ文字定数の値と一致するか。また、そのような文字定数が負の値を持つか。＞

条件付きインクルードを制御する定数式中の１文字定数の値は、実行文字セット中の同じ文字定数の値と一致します。このような文字定数は負の値を持つこともできます。

■ANSI 規格 6.8.2　ソースファイルインクルード

＜インクルードファイル（ヘッダファイル）の検索方法＞

#include で指定されたヘッダファイルの検索順序は、3.2「C/C++ コンパイラの起動オプション」表3.6 の -I オプション（-I *path*）の機能欄で説明しています。

＜引用符で囲まれたインクルードファイル名＞

#include 前処理命令では、インクルードファイル名を引用符（"）で囲んで指定することもできます。

＜ソースファイル文字シーケンスの配置＞

ソースファイルの文字にはそれぞれASCII 文字の値が割り当てられます。

■ANSI 規格 6.8.6　#pragma

＜コンパイラにおける #pragma 前処理命令の振舞い＞

解釈できない #pragma 前処理指令は無視されます。

■ANSI 規格 6.8.8　予約マクロ名

＜__DATE__ と __TIME__ の定義 ＞

__DATE__ と __TIME__ は常時使用可能です。

10.2.14ライブラリ関数（Library Functions）

■ANSI 規格 7.1.6　共通定義ヘッダ<stddef.h>

＜NULL で展開される null ポインタ＞

マクロ定数NULL（null ポインタ）の定義は(　(void *) 0　)です。

■ANSI 規格 7.2　診断関数用ヘッダ<assert.h>

＜assert 関数による診断メッセージ＞

assert 関数の終了によって出力される診断メッセージは「Assertion failed: 式 , file:ファイル名 , line:行番号」です。

■ANSI 規格 7.3.1　文字判定関数

＜文字判定関数で検査される文字セット＞

isalnum、isalpha、iscntrl、islower、isupper、isprint が真を返す文字（ASCII 文字）は次のとおりです。

関数名	文字セット
isalnum	0～9、A～Z、a～z
isalpha	A～Z、a～z
iscntrl	0x00～0x1F、0x7F
islower	a～z
isupper	A～Z
isprint	0x20～0x7E

■ANSI 規格 7.5.1　エラー処理

＜定義域エラー（domain error）発生時、数学関数が返す値＞

数学関数において定義域エラーが起きた場合、errno に EDOM が設定されます。

＜アンダーフローエラー発生時、数学関数が errno を ERANGE マクロの値にセットするかどうか。>
数学関数においてアンダーフローエラーが起きた場合、errno に ERANGE が設定されます。

■ANSI 規格 7.5.6.4　fmod 関数

＜fmod 関数の第 2 引数が 0 のとき＞

fmod 関数の第 2 引数が 0 の場合、ドメインエラーとなり、errno に EDOM が設定されます（計算結果は保証されません）。

■ANSI 規格 7.7.1.1　signal 関数

＜signal 関数のシグナル設定＞

ユーザーが作成する signal 関数（低水準関数）の仕様に依存します。

＜signal 関数で処理できるシグナルは？＞

ユーザーが作成する signal 関数（低水準関数）の仕様に依存します。

＜signal 関数で処理できる各シグナルの、デフォルトハンドラおよびプログラム開始ハンドラ＞"
ユーザーが作成する signal 関数（低水準関数）の仕様に依存します。

＜signal（sig，SIG_DEL)相当のものがシグナルハンドラ呼び出しより先に実行されない場合、シグナルのブロックは実行されるか。＞

ユーザーが作成する signal 関数（低水準関数）の仕様に依存します。

＜signal 関数に指定したハンドラによってシグナル SIGILL を受け付けた場合、デフォルトハンドラがリセットされるかどうか。＞

ユーザーが作成する signal 関数（低水準関数）の仕様に依存します。

■ANSI 規格 7.9.2　ストリーム

＜テキストストリーム（テキストファイル）の最終行には改行文字が必要か。＞

ユーザーが作成する read 関数および write 関数（低水準関数）の仕様に依存します。なお、これらの関数を呼び出す C 標準ライブラリの関数は、最終行に改行コードがなくても動作するように設計されています。

＜改行文字の直前にテキストストリームへ書き出された空白文字は読み込み時に出力されるか。＞

ユーザーが作成する read 関数および write 関数（低水準関数）の仕様に依存します。

＜バイナリストリームに追加される null 文字の数＞

バイナリストリームの末尾に null 文字は付加されません。

■ANSI 規格 7.9.3　ファイル

＜追加モードストリームのファイル位置指示子の位置＞

ファイル位置指示子は、はじめにファイルの終わりに位置します。

＜テキストストリームへの書き込みがそのポイントを超えて関連ファイルを切り詰めるかどうか。＞

テキストストリームの切り詰めは起こりません。

＜ファイルバッファリングの特性＞

バッファリング方式には、フルバッファリング、行バッファリング、バッファリングなしの 3 種類があり、setbuf 関数および setvbuf 関数によって設定／変更できます。

＜長さ0のファイルが実際に存在するかどうか。＞

ユーザーが作成する低水準関数の仕様に依存します。

＜有効なファイル名を作るための規則＞

ユーザーが作成する低水準関数の仕様に依存します。

＜同一のファイルを何回もオープンできるか。＞

ユーザーが作成するopen関数（低水準関数）の仕様に依存します。

■ANSI規格 7.9.4.1　remove関数

＜オープンファイルにおけるremove関数の効果＞

ユーザーが作成するremove（低水準関数）の仕様に依存します。

■ANSI規格 7.9.4.2　rename関数

＜rename関数を呼び出す前に新しい名前を持つファイルがあった場合、そのファイルはどうなるか。＞

ユーザーが作成するrename関数（低水準関数）の仕様に依存します。

■ANSI規格 7.9.6.1　fprintf関数

＜fprintf関数における%p変換の出力＞

printf系の変換指定%pの出力は、%08Xの出力と等しくなります。

■ANSI規格 7.6.9.2　fscanf関数

＜fscanf関数における%p変換の入力＞

scanf系の変換指定%pの入力は、%Xの入力と等しくなります。

＜scanf系におけるハイフンの解釈＞

scanf系の%[変換では、－（ハイフン）で文字コードセット上の範囲を指定し、複数の文字を指定できます。以下に、文字セットがASCIIの場合の指定例を示します。

指定	指定される文字コード
%[a-f]	abcdef
%[0-9][a-f]	0123456789abcdef
%[--a]	-からaの文字まで
%[---]	－のみ"%[z-a]　なし（何にも一致しません）
%[-af]	-af（ハイフンが1文字目の場合、範囲指定と見なされません）
%[af-]	af-（ハイフンが最後の文字の場合、範囲指定と見なされません）
%[a-f-z]	abcdef-z

■ANSI 規格 7.9.9.1　fgetpos 関数、7.9.9.4　ftell 関数

＜fgetpos 関数、ftell 関数が失敗した場合＞

fgetpos 関数および ftell 関数が失敗した場合（環境がエラーを返した場合）、errno には EFSEEK が設定されます。

■ANSI 規格 7.9.10.4　　perorr 関数

＜perror 関数によって生成されるメッセージ＞

ユーザーが作成する _strerror 関数（低水準関数）の仕様に依存します。

■ANSI 規格 7.10.3　メモリ操作関数

＜0 バイトサイズのメモリが要求された場合の calloc、malloc、realloc 関数の動作＞

calloc、malloc、realloc 関数に 0 バイトのサイズのメモリを要求したときは、NULL ポインタを返します。

■ANSI 規格 7.10.4.1　abort 関数

＜オープンファイルと一時ファイルに関する abort 関数の動作＞

ユーザーが作成する abort 関数（低水準関数）の仕様に依存します。

■ANSI 規格 7.10.4.3　exit 関数

＜exit 関数が返す終了ステータス（引数の値が 0、EXIT_SUCCESS、または EXIT_FAILURE のいずれでもない場合）＞

ユーザーが作成する _exit 関数（低水準関数）の仕様に依存します。

■ANSI 規格 7.10.4.4　getenv 関数

＜getenv 関数における環境名のセットおよび環境リストの変更方法＞

ユーザーの作成する getenv 関数（低水準関数）の仕様に依存します。

■ANSI 規格 7.10.4.5　system 関数

＜system 関数による文字列の内容および実行モード＞

ユーザーの作成する system 関数（低水準関数）の仕様に依存します。

■ANSI 規格 7.11.6.2　strerror 関数

＜strerror 関数に返されるエラーメッセージ＞

ユーザーが作成する _strerror 関数（低水準関数）の仕様に依存します。

■ANSI 規格 7.12.1　時刻データの内容

＜現地時間と夏時間＞

現地時間（各ロケールにおける暦時間）の設定には環境変数 TZ を使用します。現地時間として、JST（デフォルト）、EST5EDT、CST6CDT、MST7MDT、PST8PDT、UTC をサポートしています。夏時間として、EST5EDT、CST6CDT、MST7MDT をサポートしています。

■ANSI 規格 7.12.2.1　clock 関数

＜clock 関数における経過時間＞

ユーザーが作成する clock 関数（低水準関数）の仕様に依存します。

10.3 ロケール特有の振舞い

ANSI 規格で「ロケールに特有な振舞い」扱いとなっている動作について、C/C++ コンパイラ cc32R における C 言語での振舞いを以下に示します。

「■ANSI 規格」に続く番号と見出しは、対応する ANSI 規格「ANSI/ISO 9899-1990」の節番号と節タイトルです。また、「ロケールに特有な振舞い」扱いとなっている事項を＜＞内に、それに対する cc32R の動作を＜＞の後に示します。

■ANSI 規格 5.2.1　文字セット

＜拡張実行用文字セットの内容＞

JIS X 0201（ラテン文字を除く）、JIS X 0208 で定義される文字が使用できます。

■ANSI 規格 5.2.2　文字表示の意味

＜印字方向＞

印字方向は左から右です。

■ANSI 規格 7.1.1　ライブラリ用語定義

＜小数点を表す文字＞

小数点をあらわす文字は、全てのロケールにおいて 0x2E（‘．’）です。

■ANSI 規格 7.3　文字操作関数用ヘッダ<ctype.h>

＜文字判定関数における判定文字セット＞

文字操作関数の文字判定関数における、実装に依存する判定文字セットについては、10.2「インプリメンテーション依存の振舞い」の 10.2.14「ライブラリ関数」にある「ANSI 規格 7.3.1 文字判定関数」の部分を参照してください。

全てのロケールにおいて、ctype.h に定義および宣言されたマクロや関数は、”C”環境（”C”locale)と同じ動作をします。

■ANSI 規格 7.11.4.4　strncmp 関数

＜文字セットの照合順序 ＞

全てのロケールのおいて、strncmp 関数における文字セットの照合順序は、ASCII の照合順序と同じです。

■ANSI規格 7.12.3.5　strftime関数

＜時刻と日付の形式＞

strftime関数における日付けおよび時刻は、全てのロケールにおいて以下のように表現します。

・日付　　mm1/dd/yy （mm1は月、ddは日、yyは西暦の下2桁）
・時刻　　hh:mm2:ss （hhは時間、mm2は分、ssは秒）

第11章
低水準ライブラリ

11.1 低水準ライブラリの作成

11.1.1 C標準ライブラリに必要な低水準ライブラリ

標準入出力、メモリ管理、シグナル処理、および、時間操作などのライブラリ関数の一部の機能は、ターゲットシステムに依存します。C標準ライブラリでは、これらのターゲットシステムに依存した機能を低水準関数（低水準ライブラリ）として分離し、各機能を実現する処理関数（低水準関数。表11.1、表11.2参照）の入出力仕様を定義しています。

C標準ライブラリでは、ターゲットシステムに依存する機能を、低水準関数を呼び出すことで実現しています。したがって、C/C++プログラム内でターゲットシステムに依存する機能を使用する場合、必要な低水準関数をユーザーが作成する必要があります。

各低水準関数の仕様は、11.2「低水準ライブラリの仕様」に示します。なお、どのC標準ライブラリ関数がどの低水準関数を使用しているかについては表11.3で示しています。

表11.1 C標準ライブラリが使用する低水準関数

低水準関数名	機能
open	ファイルのオープン
close	ファイルのクローズ
read	ファイルからの読み出し
write	ファイルへの書き込み
lseek	ファイルの読み出し／書き込みの位置の設定
_get_core	メモリ領域の確保
_rel_core	メモリ領域の解放
getuniqnum	プロセスごとのユニークな番号を取得
_strerror	エラー番号に対応するエラーメッセージの取得
_exit	プログラムの終了

ANSI 規格ではエントリ関数と規定されているものでも、C 標準ライブラリでは低水準関数と位置付けられているものとして次の関数があります（表 11.2）。これらの関数の仕様は第 9 章「C 標準ライブラリ」を参照してください。

表 11.2　cc32R では低水準関数扱いとなる関数

ヘッダ	ライブラリ関数名	機能
signal.h	raise	シグナルを送る
	signal	シグナルを受けた場合の処理を指定する
stdio.h	remove	ファイルを削除する
	rename	ファイル名を変更する
stdlib.h	getenv	環境変数を得る
	system	パラメータを実行されるホスト環境に渡す
time.h	clock	実行時間を取り出す
	time	現在の暦時間を得る

C 標準ライブラリ関数のうち、低水準ライブラリを必要とする関数を以下に示します。それぞれ、表11.1および表11.2で示す低水準関数のどれを必要とするかも示します。

表 11.3　C 標準ライブラリと低水準関数の対応一覧（1/2）

分類	C標準ライブラリ関数	必要な低水準関数
assert.h	assert	_get_core, _rel_core, write, lseek, raise
locale.h	localeconv	_get_core, _rel_core
	setlocale	_get_core, _rel_core
signal.h	raise	raise
	signal	signal
stdio.h	perror	write, _get_core, _rel_core, lseek, _strerror
	fclose	write, getuniqnum, _rel_core, close, remove, lseek
	fflush	write, lseek
	fopen	open, close, lseek
	freopen	write, getuniqnum, _rel_core, close, open, remove, lseek
	remove	remove
	rename	rename
	tmpfile	getuniqnum, open, close, lseek
	tmpnam	getuniqnum
	fgetpos	lseek
	fseek	lseek, write
	fsetpos	lseek, write
	ftell	lseek
	rewind	lseek, write
	fgetc	read, _get_core, _rel_core
	fgets	read, _get_core, _rel_core

表 11.3 C標準ライブラリと低水準関数の対応一覧（2/2）

分類	C標準ライブラリ関数	必要な低水準関数
stdio.h（続き）	fputc	write, _get_core, lseek, _rel_core
	fputs	write, _get_core, lseek, _rel_core
	getc	read, _get_core, _rel_core
	getchar	read, _get_core, _rel_core
	gets	read, _get_core, _rel_core
	putc	write, _get_core, _rel_core, lseek
	putchar	write, _get_core, _rel_core, lseek
	puts	write, _get_core, _rel_core, lseek
	ungetc	_get_core, _rel_core
	fread	read, _get_core, _rel_core
	fwrite	write, _get_core, _rel_core, lseek
	fprintf	write, _get_core, _rel_core, lseek
	fscanf	read, _get_core, _rel_core
	printf	write, _get_core, _rel_core, lseek
	scanf	read, _get_core, _rel_core
	sprintf	write, _get_core, _rel_core, lseek
	sscanf	read, _get_core, _rel_core
	setbuf	_rel_core
	setvbuf	_rel_core
	vfprintf	write, _get_core, _rel_core, lseek
	vprintf	write, _get_core, _rel_core, lseek
	vsprintf	write, _get_core, _rel_core, lseek
stdlib.h	abort	raise
	calloc	_get_core, _rel_core
	free	_rel_core
	getenv	getenv
	malloc	_get_core, _rel_core
	realloc	_rel_core, _get_core
	system	system
time.h	asctime	write, _get_core, _rel_core, lseek
	clock	clock
	ctime	write, _get_core, getenv, _rel_core, lseek
	localtime	getenv
	strftime	write, _get_core, getenv, _rel_core, lseek
	time	time

||||| 注意 |||||

低水準ライブラリを実際に使用するために必要な初期設定は、C/C++ プログラム起動前に実行します。スタートアッププログラム中に、必要な低水準ライブラリの初期設

次ページ →

定プログラムを記述しておいてください。

11.1.2 低水準ライブラリによる入出力について

ファイルは、標準ライブラリレベルの標準入出力関数では、FILE 型のデータを使って管理しています。これに対し、低水準ライブラリでは、ファイルに「ファイル番号」を割り当てて管理します。ファイル番号は、実際のファイルと 1 対 1 に対応する 0 以上の整数です。

ファイル番号 0、1、2 には、それぞれ標準入力、標準出力、標準エラー出力にあらかじめ割り当てておきます。これらは、0 はコンソールからの入力、1 および 2 はコンソールへの出力とするのが一般的です。ファイル番号 0 ～ 2 に対しては、実行環境において矛盾のないように低水準ライブラリを作成してください。とくに、0 ～ 2 がすでに割り当てられているときに open 関数が 0 ～ 2 の値を返さないように注意してください。

低水準ライブラリの open 関数は、与えられたファイルのパス名（ファイルシステムの中でファイルを識別するための名前）に対してファイル番号を与えます。open 関数では、このファイル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き、open 関数で判定する必要があります。

■ バッファの位置、サイズ等の情報

ファイルのバッファリングをする場合に必要です。

■ ファイルの先頭から次に読み出しまたは書き込みを行う位置までのバイトオフセット

ディスクファイルへの入出力に必要です。

open 関数で設定した情報にもとづいて、以後、そのファイルに対して入出力（read/write 関数）、読み出し／書き込み位置の設定（lseek 関数）を行います。close 関数では、ファイルのバッファリングを行っている場合、バッファの内容を読み出して実際のファイルに書き込み、open 関数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。

11.2 低水準ライブラリの仕様

以下より、低水準ライブラリを作成するために、各関数の仕様を示します（アルファベット順）。内容は、形式（関数を呼び出すためのインタフェース）、機能（動作概要）、関数値（値：意味）、および解説（動作内容や実現上の注意事項、例など）です。

_exit　　低水準関数

形式　　void _exit (int *ex_num*);

ex_num;　　/* プログラムの終了コード */

機能　　環境にプログラムの終了コードを返し、プログラムの実行を終了します。

関数値（戻り値）

なし

解説　　_exit関数は、プログラムが終了することを環境に通知します。つまり、この関数内において使用している環境の終了処理を記述してください。

_get_core

低水準関数

形式　　void *_get_core (int *size*);

size;　　/* 割り付けるデータのサイズ */

機能　　メモリ領域を割り付けます。

関数値（戻り値）

割り付けた領域の先頭アドレス　：正常終了
(void*)0　：エラー終了

解説　　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。割り付けるための空き領域がなくなった場合はエラーになります。
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「(void *)0」を返してください。_get_core関数の実現例を以下に示します。

```
#define  HEAPSIZE xxxxxx …… _get_core関数で管理する領域のサイズをバイト数で指定してください。
static union{
   int dummy;
   char heap [HEAPSIZE];
}heap_area; …… 割り付けのためのデータ領域の宣言です。4バイト境界に整合させるために、
                  int型との共用体で宣言します。

static char *brk = heap_area.heap; ……割り付ける領域への先頭アドレスを代入して初期化します。

void *_get_core(int size)
{
    char *p;

   Êif( brk + size > heap_area.heap + HEAPSIZE) ……領域が残っているかどうかチェックします。
      return((void *)0);
    p = brk;
    brk += size;   …… 割り付けた領域の最終アドレスを更新します。
    return((voidÊ*)p);  …… 割り付けた領域の先頭アドレスを返します。
}
```

_rel_core

低水準関数

形式　　void _rel_core (void *_ptr_);

ptr;　　/* 解放するメモリへのポインタ */

機能　　_get_core関数で得られたメモリを解放します。

関数値（戻り値）

なし

解説　　解放するメモリへのポインタが引数として渡されます。_get_core関数で管理しているメモリを解放する処理をします。

_strerror

低水準関数

形式　　char *_strerror (int *error_number*);

error_number;　　/* ユーザー定義のエラー番号 */

機能　　ユーザー定義のエラー番号に対応するエラーメッセージを返します。

関数値（戻り値）

エラーメッセージ（文字列）へのポインタ

解説　　エラー番号が引数として渡されます。この番号に対応するエラーメッセージ文字列へのポインタを返してください。

close

低水準関数

形式　　int close (int *file_no*) ;

file_no;　　/* クローズするファイル番号 */

機能　　ファイルをクローズします。

関数値（戻り値）

0　　：正常終了
-1　　：エラー終了

解説　　open 関数で得られたファイル番号が引数として渡されます。
open 関数で設定したファイルの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、低水準ライブラリ内で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出力してください。
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。

getuniqnum

低水準関数

形式　　int getuniqnum (void);

機能　　複数のプログラムが同時に実行するような場合、その実行単位ごとに付けられた固有の値を得ます。

関数値（戻り値）

プログラムの動作単位に固有の値を返します。プログラム動作中は、この値は変化してはなりません。

解説　　プログラムが同時に複数動作するような場合、環境がその動作単位を特定するために用いている番号を返します。この値は、例えば同時に動作している２つの動作単位が同一の実行コードである場合でも、getuniqnum関数が返す値は互いに異なる必要があります。また、同一の動作単位である以上は、getuniqnum関数は必ず同じ値を返してください。なお、プログラムが同時に複数動作しないような環境である場合は、戻り値は常に同じ値でなければなりません。

lseek

低水準関数

形式　long lseek (int *file_no*, long *offset*, int *base*);

file_no;　/* 対象となるファイル番号 */
offset;　/* 読み出し／書き込みの位置を示すオフセット（バイト単位）*/
base;　/* オフセットの起点 */

機能　ファイルの読み出し／書き込みの位置を設定します。

関数値（戻り値）

新しいファイルの読み出し／書き込みの位置の
ファイルの先頭からのオフセット（バイト単位）　：正常終了

-1　：エラー終了

解説　ファイル番号file_noのファイルにおける、読み出し／書き込みを行う位置を、バイト単位で設定します。新しい位置は、第3引数(base)によって、以下の方法で計算され設定されます。

(a) baseが0の時　ファイルの先頭からoffsetバイトの位置に設定します。
(b) baseが1の時　現在の位置にoffsetバイトを加えた位置に設定します。
(c) baseが2の時　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、新しいオフセットの値が負になった場合、また(a)、(b)でファイルのサイズを超えた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出し／書き込み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。

open

低水準関数

形式　int open (const char *name, int mode, int flag);

```
name;       /* ファイルのパス名を指す文字列 */
mode;       /* ファイルをオープンするときの処理の指定 */
flag;       /* ファイルをオープンするときの処理の指定(常に0777) */
```

機能　ファイルをオープンします。

関数値（戻り値）

オープンしたファイル番号　：正常終了
-1　：エラー終了

解説　引数として渡されたファイルのパス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関数では、後で読み出し／書き込みを行うために、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関数で返したファイル番号を用いて読み出し／書き込みを行うたびに参照されます。第2パラメータの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。

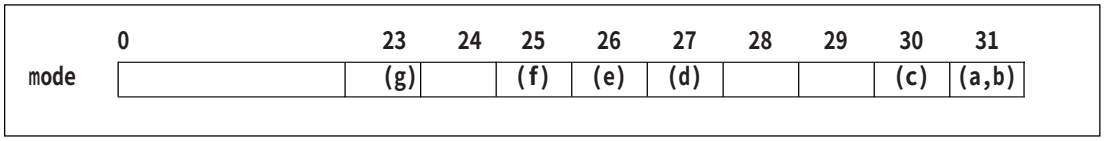


図11.1　modeによる指定

(a) O_RDONLY(第31ビット)　このビットが0のとき、ファイルを読み出し専用にオープンすることを示します。

(b) O_WRONLY(第31ビット)　このビットが1のとき、ファイルを書き込み専用にオープンすることを示します。

(c) O_RDWR(第30ビット)　このビットが1で、31ビットが0のとき、ファイルを読み出し、書き込み両用にオープンすることを示します。なお、30、31ビットが共に1となることはありません。

(d) O_APPEND(第27ビット)　このビットが1のとき、次に読み出し／書き込みを行うファイル内の位置を、ファイルの最後に設定することを示します。
このビットが0のときはファイル内の位置はファイルの先頭になります。　（続く）

(e)O_CREAT(第26ビット)　このビットが1のとき、パス名で示すファイルが存在しない場合、ファイルを新規に作成することを示します。

(f) O_TRUNC(第25ビット)　このビットが1のとき、パス名で示すファイルが存在する場合、ファイルの内容を捨て、ファイルのサイズを0にすることを示します。

(g) O_BINARY(第23ビット)　このビットが0のとき、ファイルをテキストモードでオープンすることを示します。
このビットが1のとき、ファイルをバイナリモードでオープンすることを示します。

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（0以上の整数)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、低水準ライブラリ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。

read

低水準関数

形式

```
int read ( int file_no,  char *buffer,  unsigned int count );

file_no;      /* 読み出しの対象となるファイル番号 */
buffer;       /* 読み出したデータを格納する領域へのポインタ*/
count;        /* 読み出しバイト数 */
```

機能

ファイルからのデータの読み出しを行います。

関数値（戻り値）

実際に読み出したバイト数　：正常終了
-1　　：エラー終了

解説

ファイル番号 file_no で示すファイルから count で示すバイト数以内の量のデータを読み出し、buffer の指す領域に格納します。ファイルの読み出し／書き込みの位置は、読み出したバイト数だけ先に進みます。正常に読み出しができた場合は実際に読み出したバイト数、読み出しに失敗した場合は -1 を返してください。また、読み出しデータがなくファイルの終端（EOF）に到達したときは -1 でなく 0 を返してください。

write

低水準関数

形式　int write (int *file_no*, const char *buffer, unsigned int *count*);

```
file_no;    /* 書き込みの対象となるファイル番号 */
buffer;     /* 書き込みデータが格納されている領域へのポインタ */
count;      /* 書き込みバイト数 */
```

機能　ファイルへのデータの書き込みを行います。

関数値（戻り値）

実際に書き込まれたバイト数	：正常終了
-1	：エラー終了

解説　bufferの指す領域から、countバイトのデータを、ファイル番号file_noで示すファイルへ書き込みます。ファイルの読み出し／書き込みの位置は、書き込んだバイト数だけ先に進みます。正常に書き込みができた場合は実際に書き込んだバイト数を、書き込みに失敗した場合は -1 を返してください。

第 12 章
単精度数学関数ライブラリ

C言語標準の数学関数ライブラリ(math.h ヘッダで定義)を単精度化したライブラリです。単精度化により、従来倍精度を用いていた数学関数ライブラリを用いるアプリケーションの効率向上(実行速度向上とサイズ抑制)が図れます。

単精度数学関数には、FPU 命令を使わない関数と、FPU 命令を使う関数があります。

12.1 関数構成

表12.1 に単精度数学関数ライブラリの関数リストを示します。

これら単精度数学関数の戻り値と引数の型は、表中に記載しているC言語のプロトタイプ宣言の形式で確認ください。

表 12.1 単精度数学関数ライブラリ

単精度数学関数 (C言語のプロトタイプ宣言の形式)		機能	対応する倍精度数学関数
FPU命令を使わない	FPU命令を使う		
float cosf(float)	float cosf5(float)	余弦	cos
float sinf(float)	float sinf5(float)	正弦	sin
float tanf(float)	float tanf5(float)	正接	tan
float acosf(float)	float acosf5(float)	逆余弦	acos
float asinf(float)	float asinf5(float)	逆正弦	asin
float atanf(float)	float atanf5(float)	逆正接	atan
float atan2f(float,float)	float atan2f5(float,float)	逆正接(除算)	atan2
float coshf(float)	float coshf5(float)	双曲線余弦	cosh
float sinhf(float)	float sinhf5(float)	双曲線正弦	sinh
float tanhf(float)	float tanhf5(float)	双曲線正接	tanh
float powf(float,float)	float powf5(float,float)	累乗	pow
float sqrtf(float)	float sqrtf5(float)	正の平方根	sqrt
float ceilf(float)	float ceilf5(float)	小数点以下を切り上げた整数値	ceil
float expf(float)	float expf5(float)	指数関数	exp
float fabsf(float)	float fabsf5(float)	絶対値	fabs
float floorf(float)	float floorf5(float)	小数点以下を切り捨てた整数値	floor
float fmodf(float,float)	float fmodf5(float,float)	剰余	fmod
float frexpf(float,int*)	float frexpf5(float,int*)	0.5～1.0の値と2の累乗に分解	frexp
float ldexpf(float,int)	float ldexpf5(float,int)	2の累乗の乗算	ldexp
float logf(float)	float logf5(float)	自然対数	log
float log10f(float)	float log10f5(float)	10が底の対数	log10
float modff(float,float*)	float modff5(float,float*)	整数部分と小数部分に分解	modf

これらの単精度数学関数の仕様は、C 言語標準の倍精度数学関数をベースに、次のような規則に基づいて決めています。

[1] 機能
内部演算をfloat型で行うほかは、基本機能はベースとなる倍精度数学関数と同じです。

[2] 関数名
FPU 命令を使わない関数は、ベースとなる倍精度数学関数の名前にそれぞれ'f'を付けたものです。FPU命令を使う関数は、同様に'f5'を付けたものです。

[3] 引数と戻り値の型
引数および戻り値をdouble型からfloat型にしたものです。

12.2 使用方法

12.2.1 ヘッダファイル

単精度数学関数を用いる場合、次のいずれかのヘッダファイルをインクルードする必要があります。必要に応じてどちらか一方のヘッダを選択してください。

mathf.h	単精度数学関数を用いる場合
math.h	C 標準の倍精度数学関数または単精度数学関数を用いる場合

なおmath.hはmathf.hの機能を包含していますので、math.hをインクルードしている場合は、mathf.hをインクルードする必要はありません。これらのヘッダファイルには次の内容が記述されています。

[1] 関数のプロトタイプ宣言
mathf.h では 単精度数学関数の、math.h では 倍精度数学関数と単精度数学関数両方のプロトタイプ宣言を行っています。

[2] 関数名置換
mathf.h は、コンパイル時に -m32re5 オプション((M32R-FPU コアの FPU 命令を使用する)を指定すると、FPU 命令を使わない単精度数学関数の呼び出しを、同じ機能の FPU 命令を使う単精度数学ライブラリ関数の呼び出しに変更します。

```
例1) "-m32re5" を同時指定
        #include <mathf.h>   /* <math.h>でも可 */
        ans=cosf(rd);
                ↓   -m32re5 付きでコンパイル
        ans = cosf5(rd);  と等価になる
```

この置換は例えばcosf関数の場合は次のようなマクロにより行っています。

```
#definecosfcosf5
        :
(他の単精度数学関数についても同様に定義)
        :
```

【注意】
呼び出す関数名が置換される場合、ロードモジュールには置換前の関数名ではなく置換後の関数名を格納します。このため、デバッガ(M3T-PD32R など)や TM インスペクタでは、置換前の関数名を指定したり表示させたりすることはできません(置換後の関数名の表示や指定は可能です)。

12.2.2 単精度数学関数ライブラリとのリンク

単精度数学関数ライブラリは、添付の C 標準ライブラリ(m32RcR.lib 等)に含まれています。
従来の C 標準ライブラリ関数を使用するのと同様に、C 標準ライブラリをリンクしてください。

12.3 注意事項

12.3.1 ダイナミックレンジ

単精度(float型)は、表現可能な数値の絶対値が倍精度(double型)に比べて小さくなっています。倍精度の数学関数を単精度の数学関数に置き換える場合は、入力値や出力値が単精度の範囲を越えないかどうかに注意してご使用ください。

12.3.2 エラー処理について

次のエラーを発生する条件は、C 標準の倍精度の数学ライブラリと同様に行います。これらの条件は関数によって異なりますので、対応する倍精度数学関数の関数仕様をユーザーズマニュアルの「9.3 C標準ライブラリ関数詳細」にて確認ください。

- 定義域エラー（EDOM）が発生する条件
- 範囲エラー（ERANGE）が発生する条件
- 値としてオーバーフロー値（HUGE_VAL）が返される条件
- 値としてアンダーフロー値（0）が返される条件

第 13 章

64bit整数演算関数ライブラリ

C言語の整数演算を、64bitのダイナミックレンジで行う関数群を標準ライブラリに追加しました。C言語の整数型と同様に、加減乗除、ビット演算、シフト、比較などの演算を64bitの範囲で行うことが可能です。

13.1 ヘッダファイル long64.h

64bit整数演算関数を用いる場合は、ヘッダファイルlong64.hをインクルードする必要があります。

long64.h では、必要な型、定数や関数プロトタイプなどの宣言を行っています。

(1) 型名

64bit整数を保持する型(構造体)です。全ての64bit整数演算関数は、この型を用いて64bit整数を入力および出力します。

符号つき 64 ビット整数 ... LONG64

符号なし 64 ビット整数 ... ULONG64

(2) 定数

64bit整数の最大値、最小値を表す定数です。

LONG64_MAX ... LONG64の最大値

LONG64_MIN ... LONG64の最小値

ULONG64_MAX ... ULONG64の最大値

(3) プロトタイプ宣言

64bit整数演算関数のプロトタイプ宣言を行います。

13.2 関数構成

本ライブラリの関数は、主にC言語の演算子を関数にしたものから構成されています。64bit演算は、これらの関数を使用します。

(1) 算術演算関数

64bit整数の算術演算を行い、結果を64bit整数型で返します。

- 四則(加減乗除)、剰余、単項負

(2) ビット演算関数

64bit整数のビット演算を行い、結果を64bit整数型で返します。

- ビットシフト(左シフト、右論理シフト、右算術シフト)
- ビット論理演算(論理和、論理積、排他的論理和、反転)

(3) 比較･判定関数

64bit整数の比較または0かどうかの判定を行い、結果をint型で返します。

- 比較
- 0かどうかの判定

(4) 型変換関数

64bit整数からC言語の整数型または浮動小数点型への変換、およびその逆方向の変換を行います

- 符号つき64bit整数 ←→ 符号なし64bit整数
- 符号つき64bit整数 ⟷ float, double
- 符号なし64bit整数 ⟷ float, double
- 符号つき64bit整数 ⟷ long, unsigned long
- 符号なし64bit整数 ⟷ long, unsigned long

(5) その他の関数

即値をセットしたり、10進数などの文字列を64bit整数に置き換える関数です。

- 即値のセット
- 文字列を64bit整数値に変換

表13.1～表13.6に、これら64bit整数演算関数ライブラリのリストを示します。
これら関数の戻り値と引数の型は、表中に記載しているC言語のプロトタイプ宣言の形式で確認ください。

表13.1 (1) 算術演算関数

関数 (C言語のプロトタイプ宣言の形式)	機能 (S)=符号つき (U)=符号なし	演算内容
・四則(加減乗除)、剰余、単項負		
LONG64 addl64(LONG64 n1, LONG64 n2);	(S)加算	n1 + n2
LONG64 subl64(LONG64 n1, LONG64 n2);	(S)減算	n1 - n2
LONG64 mull64(LONG64 n1, LONG64 n2);	(S)乗算	n1 * n2
LONG64 divl64(LONG64 n1, LONG64 n2);	(S)除算	n1 / n2
LONG64 modl64(LONG64 n1, LONG64 n2);	(S)剰余算	n1 % n2
LONG64 negl64(LONG64 n1);	(S)単項負	-n1
ULONG64 addul64(ULONG64 n1, ULONG64 n2);	(U)加算	n1 + n2
ULONG64 subul64(ULONG64 n1, ULONG64 n2);	(U)減算	n1 - n2
ULONG64 mulul64(ULONG64 n1, ULONG64 n2);	(U)乗算	n1 * n2
ULONG64 divul64(ULONG64 n1, ULONG64 n2);	(U)除算	n1 / n2
ULONG64 modul64(ULONG64 n1, ULONG64 n2);	(U)剰余算	n1 % n2
ULONG64 negul64(ULONG64 n1);	(U)単項負	-n1

表13.2 (2) ビット演算関数

関数 (C言語のプロトタイプ宣言の形式)	機能 (S)=符号つき (U)=符号なし	演算内容
・ビットシフト(左シフト、右論理シフト、右算術シフト)		
LONG64 shltl64(LONG64 n1, unsigned int nbit);	(S)左シフト	n1 << nbit
LONG64 shrtl64(LONG64 n1, unsigned int nbit);	(S)右算術シフト	n1 >> nbit
ULONG64 shltul64(ULONG64 n1,unsigned int nbit);	(U)左シフト	n1 << nbit
ULONG64 shrtul64(ULONG64 n1,unsigned int nbit);	(U)右論理シフト	n1 >> nbit
・ビット論理演算(論理和、論理積、排他的論理和、反転)		
LONG64 orl64(LONG64 n1, LONG64 n2);	(S)論理和	n1 | n2
LONG64 andl64(LONG64 n1, LONG64 n2);	(S)論理積	n1 & n2
LONG64 xorl64(LONG64 n1, LONG64 n2);	(S)排他的論理和	n1 ^ n2
LONG64 notl64(LONG64 n1);	(S)反転	~n1
ULONG64 orul64(ULONG64 n1, ULONG64 n2);	(U)論理和	n1 | n2
ULONG64 andul64(ULONG64 n1, ULONG64 n2);	(U)論理積	n1 & n2
ULONG64 xorul64(ULONG64 n1, ULONG64 n2);	(U)排他的論理和	n1 ^ n2
ULONG64 notul64(ULONG64 n1);	(U)反転	~n1

表13.3 (3) 比較・判定関数

関数 (C言語のプロトタイプ宣言の形式)	機能 (S)=符号つき (U)=符号なし	判定内容
・比較		
int cmpl64(LONG64 n1, LONG64 n2); int cmpul64(ULONG64 n1, ULONG64 n2);	(S)比較 (U)比較	n1 > n2 なら 正の値 n1 == n2 なら 0 n1 < n2 なら 負の値
・0かどうかの判定		
int evall64(LONG64 n1); int evalul64(ULONG64 n1);	(S)判定 (U)判定	n1 == 0 なら 0、 n1 != 0 なら 0でない値

表13.4 (4) 型変換関数

関数 (C言語のプロトタイプ宣言の形式)	変換内容
・符号つき64bit整数 ←→ 符号なし64bit整数	
ULONG64 l64_to_ul64(LONG64 input);	LONG64 → ULONG64
LONG64 ul64_to_l64(ULONG64 input);	ULONG64 → LONG64
・符号つき64bit整数 ←→ float, double	
floatl64_to_float(LONG64input);	LONG64 → float
doublel64_to_double(LONG64input);	LONG64 → double
LONG64float_to_l64(floatinput);	float → LONG64
LONG64double_to_l64(doubleinput);	double → LONG64

表13.5 (4) 型変換関数

関数 (C言語のプロトタイプ宣言の形式)	変換内容
・符号なし64bit整数 ←→ float, double	
floatul64_to_float(ULONG64input);	ULONG64 → float
doubleul64_to_double(ULONG64input);	ULONG64 → double
ULONG64float_to_ul64(floatinput);	float → ULONG64
ULONG64double_to_ul64(doubleinput);	double → ULONG64
・符号つき64bit整数 ←→ long, unsigned long	
longl64_to_long(LONG64input);	LONG64 → long
unsignedlongl64_to_ulong(LONG64input);	LONG64 → unsigned long
LONG64long_to_l64(longinput);	long → LONG64
LONG64ulong_to_l64(unsignedlonginput);	unsigned long → LONG64
・符号なし64bit整数 ←→ long, unsigned long	
longul64_to_long(ULONG64input);	ULONG64 → long
unsignedlongul64_to_ulong(ULONG64input);	ULONG64 → unsigned long
ULONG64long_to_ul64(longinput);	long → ULONG64
ULONG64ulong_to_ul64(unsignedlonginput);	unsigned long → ULONG64

表13.6 (5) その他の関数

関数 (C言語のプロトタイプ宣言の形式)	機能 (S)=符号つき (U)=符号なし	演算内容
・即値のセット		
LONG64 imml64(signed long shigh, unsigned long ulow);	(S)即値設定	shigh<<32 + ulow
ULONG64 immul64(unsigned long uhigh, unsigned long ulow);	(U)即値設定	uhigh<<32 + ulow
・整数文字列を64bit整数値に変換 (strtol, strtoulを64bit整数化したもの)		
LONG64strtol64(constchar*s,char**endptr,intbase);	(S)文字列 →整数変換	strtol64bit版
ULONG64strtoul64(constchar*s,char**endptr,intbase);	(U)文字列 →整数変換	strtoul64bit版

13.3 使用方法と例

関数を利用する場合、次のようにプログラムを作成してください。

(1) long64.h をインクルードする

(2) 値を格納する変数を LONG64, ULONG64 のいずれかで定義する。

(3) 必要な演算に対し、該当する型に対応する関数を表13.1から選択して呼び出す。

(4) 戻り値を LONG64, ULONG64 型の変数で保持する。

64bit整数演算関数群は、添付のC標準ライブラリ(m32RcR.lib等)に含まれています。従来のC標準ライブラリ関数を使用するのと同様に、C標準ライブラリをリンクしてください。

以下に、プログラム例として、a = (b * (c - 10)) >> 33 を符号つき64ビットで行い、結果をlong型に変換して返す関数を示します。

```
#include <long64.h>
LONG64 a, b, c;
long
func(void)
{
    LONG64 s1, s2, s3;
    long k;
    s1 = long_to_l64(10L);    /* s1 = (LONG64)10L */
    s2 = subl64(c,s1);        /* s2 = c - s1 */
    s3 = mull64(b,s2);        /* s3 = b * s2 */
    a = shrtl64(s3,33);       /* a = s3 >> 33 */
    k = l64_to_long(a);       /* k = (long)a */
    return k;

}
```

図13.7 64bit整数演算関数の使用例

13.4 注意事項

13.4.1 符号に関する注意

64bit演算関数では、符号つきと符号なしを混在して演算することはできません。
演算関数の利用の際は、型の変換を行うなどをして、64bit整数の入力と出力全てを符号つきか符号なしのどちらかに揃えてください。

例) 符号なし(ULONG64)変数 a に符号つき(LONG64)変数 b を加算して、
符号なし変数 c に代入する場合

誤) c = addl64(a,b); または c = addul64(a,b);

↓ b を符号なしに変換し、全てを符号なしにする

正) c = addul64(a, l64_to_ul64(b));

第14章
Cコンパイラのメッセージ

14.1 コンパイラの実行結果を知るには

コンパイラの実行結果は、メッセージおよび終了コードから判断できます。

14.1.1 メッセージの出力形式

コンパイラは動作中にエラーを検出すると、実行状況を示すメッセージを標準エラー出力（通常は画面）に表示します。メッセージの出力形式を以下に示します。

■ 基本形式

入力情報：*メッセージ種別*：*メッセージ*

※「入力情報: 」は必要時のみ出力

■ パターン

ファイル名：*メッセージ種別*：*メッセージ*
ファイル名,*行番号*：*メッセージ種別*：*メッセージ*
<コマンド行>：*メッセージ種別*：*メッセージ*
メッセージ種別：*メッセージ*
※下線部分は入力情報（下線は出力されません）。

■ 出力例

```
"smp.c", line 2: error: unterminated #ifdef conditional
```

ファイル名　行番号　メッセージ種別　メッセージ

```
<command line>: error: macro name missing after -D option
```

メッセージ種別　メッセージ

```
"a.c", line 9: warning: main: function has no return statement
```

ファイル名　行番号　メッセージ種別　メッセージ

14.1.2 メッセージ種別

メッセージは警告程度によって、表14.1のように分類されています。

表14.1 メッセージ種別

メッセージ種別（表示）	メッセージ発生時の動作
インフォメーション（information）	インフォメーションメッセージを出力し、処理を続行します。
ワーニング（warning）	ワーニングメッセージを出力し、処理を続行します。
コマンドラインエラー（<command line>: error）	コマンド入力行に対するエラーメッセージを出力し、処理を中止します。
エラー（error）	エラーメッセージを出力し、処理を中止します。
重大エラー（fatal）	エラーメッセージを出力し、処理を中止します。

メッセージの詳細は、14.2「Cコンパイラのメッセージ一覧」を参照してください。

14.1.3 終了コード

コンパイラは実行後、以下の終了コード（実行結果を示す値）を返します（表14.2）。

表14.2 終了コード

終了コード	実行結果
0	正常終了した。または、ワーニングが発生した。
1	エラーが発生した。

14.2 コンパイラのメッセージ一覧

以下より、コンパイラのメッセージリストを示します（アルファベット順）。メッセージ中の「*xxx*」は、任意の入力情報（入力ファイル名、ソースコード中の識別子、名前、数字、文字列など）を意味します。

14.2.1 インフォメーション

表 14.3 コンパイラのインフォメーションメッセージ

メッセージ	解説
constant out of range due to unportable conversion	定数が範囲外です。
less complete than prior compatible declaration -- this declaration ignored	この宣言は、以前の適合する宣言に対して宣言要素が不足しています。この宣言は無視されました。
more complete than prior compatible declaration -- prior declaration ignored	この宣言は、以前の適合する宣言よりも多くの宣言要素があります。以前の宣言は無視されます。
prior identical declaration -- ignored	以前に同一の宣言があります。宣言は無視されました。
some garbage after #*xxx* argument	#*xxx* の引数に、不適切な文字が含まれています。
unnecesarily punctuated parameter list in #define	#define による関数形式のマクロ定義において、引数リストが不完全です。
#*xxx* with no argument	#*xxx* の引数がありません。

14.2.2 ワーニング

表 14.4　コンパイラのワーニングメッセージ（1/4）

メッセージ	解説
argument type inconsistent with (imputed) declaration	引数の型が宣言と一致しません。
array subscript imply one element initialized by 0	配列の添え字が 0 で初期化された要素を含んでいます。
calling a non-function object: "*xxx*".	関数でないオブジェクト *xxx* を呼び出そうとしています。
'ch': illeagal escape sequence	エスケープシーケンスが正しくありません。
xxx: conflicts with prior option -- ignored	オプション指定 *xxx* が矛盾しています。より先に指定されたものが有効となります。
constant out of range due to unportable conversion	移植の際、定数の範囲外となります。
xxx: empty macro definition; regarded as empty	-D オプションのマクロ定義が空です（-D *xxx*= の次に定義が指定されていません）。
EOF in character constant -- constant truncated	文字定数に EOF が含まれています。定数は切り縮められました。
EOF or NUL in string literal -- string truncated	文字列リテラルに EOF または NUL が含まれています。文字列は切り縮められました。
fewer arguments specified than in (imputed) definition	引数の指定が足りません。
floating point number overflow	浮動小数点数がオーバーフローしました。
floating point number underflow	浮動小数点数がアンダーフローしました。
function called before declared	宣言以前に関数が呼ばれています。

表 14.4　コンパイラのワーニングメッセージ（2/4）

メッセージ	解説
function does not return value with defined return type	関数が定義された戻り値の型で値を返していません。
function has no return statement	関数に return 文がありません。
illegal character in input file -- ignored illegal character	入力ファイル中に不正文字がありました。不正文字は無視されました。
xxx is referenced before set	名前 *xxx* が代入されずに参照されています。
length of character constant is greater than 512 -- constant truncated	文字定数の長さが 512 文字を超えています。定数は切り縮められました。
length of identifier name is 240 characters -- name truncated	識別子の長さは 240 文字です。名前は切り縮められました。
length of identifier name is 31 characters -- name truncated	識別子の長さは 31 文字です。名前は切り縮められました。
length of string literal is greater than 512 -- string truncated	文字列リテラルの長さが 512 文字を超えています。文字列は切り縮められました。
xxx: may be referenced before set	名前 *xxx* が代入されずに参照されている可能性があります。
missing terminal '''' for string -- assumed at end of line	文字列終端の " がありません。行端と見なしました。
mixed normal and wide characters in the same string -- concatenated string omitted	同じ文字列中に普通の文字とワイド文字が混在しています。連結文字列は省かれました。
more arguments specified than in (imputed) definition	引数の指定が多すぎます。
xxx: multiple optimization options -- ignored	最適化の指定が重複しています。オプション指定 *xxx* は無視されます。
nothing declared in current declaration -- ignored	宣言行に何も宣言されていません。宣言は無視されました。

表 14.4　コンパイラのワーニングメッセージ（3/4）

メッセージ	解説
number of parameters not equal between use and definition	プロトタイプ宣言とくらべて引数の数が一致しません。
parameter incompatible with previous use	引数に互換性がありません。
#*xxx*: requires exactly 1 identifier	#*xxx* で指定しなければならないのは 1 個の識別子のみです。
shift count greater than number of bits	シフトのカウントがビット数より多くなっています。
storage specifier conflicts with prior declaration -- this declaration ignored	記憶クラス指定が以前の宣言と一致していません。この宣言は無視されました。
the characters /* are found in a comment	コメント内部に /* が見つかりました。
too many digits in floating point number; extra digits ignored	浮動小数点数の桁数が多すぎます。余分な桁は無視されました。
xxx: unable to optimize -- skipped phase	入力ファイル *xxx* の最適化ができません。最適化処理をスキップします。
xxx: undefined name in #if constant expression; regarded as 0	#if の定数式中に未定義の名前 *xxx* があります。0 とみなされます。
unknown commandline option -- ignored	このコマンドオプションはサポートされていません。オプションは無視されました。
unknown directive -- directive ignored	この指令はサポートされていません。指令は無視されました。
unknown size static global array used	静的広域配列に対してこのサイズは指定できません。
xxx: unknown suffix, passed to linker	入力ファイル *xxx* の拡張子が判断できません（.c、.ms、または .mo でない）。オブジェクトファイルとみなし、ファイル名を変更せずにリンカにそのまま渡します。
unportable character constant	文字定数の値は移植性に問題があります。

表 14.4　コンパイラのワーニングメッセージ（4/4）

メッセージ	解説
unrecognized #pragma -- directive ignored	サポートしていない #pragma 命令です。命令は無視されました。
\x is hex escape sequence	＼x の後に 16 進数字を表す文字が指定されていません（＼x は 16 進数字コードの接頭辞です）。

14.2.3 コマンドラインエラー

表 14.5　コンパイラのコマンドラインエラーメッセージ

メッセージ	解説
xxx: can't execute	子プロセス *xxx*（cpre、cfrt など）が起動できません。環境変数 M32RLIB の設定やデフォルトのディレクトリ内のファイルを確認し、子プロセスが起動されるよう環境設定してください。
xxx: invalid multicharacter option	*xxx* というオプションは存在しません。
xxx: invalid parameter	オプション *xxx* に続くパラメータ指定が不適切です。" 例：-R old=new において、old が P,D,C,B 以外、または、new 指定がない。
xxx: invalid unicharacter option	*xxx* というオプションは存在しません。
xxx: missing parameter	オプション *xxx* の後ろに必要な引数指定がありません。
-: there is no option	オプションが指定されていません。

14.2.4 エラー

表 14.6　コンパイラのエラーメッセージ（1/15）

メッセージ	解説
#*xxx* after #else	#else の後ろに #*xxx* があります。
aggregate/union type object must have constant expression initializer list	集合体および共用体は定数式によって初期化しなければなりません。
argument name not specified in function header	関数ヘッダに引数が指定されていません。
#*xxx* argument starts with a digit	前処理命令 #*xxx* の初めの引数が数字で始まっています（識別子でなければなりません）。
arguments given to macro `*xxx*'	マクロ *xxx* には引数が必要です。
array subscript requires combination of pointer to object type and integral type	配列は、ポインタと整数型の組み合わせでなければなりません。
at most one storage class specifier per declaration	宣言に複数の記憶クラス指定子があります。"（宣言に記憶クラス指定子は１つしか指定できません。）
`auto' must appear within a function	auto 変数は関数内で宣言しなければなりません。
badly punctuated parameter list in #define	#define の引数に不適切な文字があります。
bit field members must have integral types	ビットフィールドのメンバは整数型でなければなりません。
bit field size must be integral constant expression	ビットフィールドのサイズ指定は整数型定数でなければなりません。
bit field size must not be negative	ビットフィールドのサイズ指定は正でなければなりません。
bit field size too large	ビットフィールドのサイズ指定が大きすぎます。

表 14.6　コンパイラのエラーメッセージ（2/15）

メッセージ	解説
block scope initialization not allowed for external declaration	ブロック内では外部変数を初期化してはなりません。
break statement in invalid context	break はここでは使用できません。
call to non-function attempted	関数でないものに対して関数コールしようとしています。
cannot const a function type object	関数型に const 宣言はできません。
cannot initialize a typedef variable	typedef 宣言時の初期化はできません。
cannot start or end with ## operator	## 演算子は置換文字列リストの最初または最後にあってはなりません。
cannot typedef function definition	関数定義に typedf 宣言はできません。
case label must be an integral constant expression	case には整数定数式を指定しなければなりません。
case or default label in invalid context	case もしくは default はここでは使用できません。
cast type must be "void" or scalar type	キャストは void またはスカラ型を指定しなければなりません。
character constant must fit its storage range	文字定数の値は範囲内でなければなりません。
constant expression evaluated out of range	定数式の値が範囲を越えています。
constant should fit its storage range	定数の値は範囲内でなければなりません。
continue statement in invalid context	continue はここでは使用できません。

表 14.6　コンパイラのエラーメッセージ（3/15）

メッセージ	解説
controlling expression of if statement must have scalar type	if 文中の式はスカラ型でなければなりません。
controlling expression of iteration statement must have scalar type	繰り返し文中の式はスカラ型でなければなりません。
controlling expression of switch statement must have integral type	switch 文中の式は整数型でなければなりません。
xxx: `-D option' must be followed by an identifier	-D オプションには識別子を指定してください。
`defined' is followed by invalid macro name	defined の後に続くマクロ名が正しくありません（識別子でなければなりません）。
xxx: `defined' is followed by invalid macro name	defined の後に続くマクロ名が正しくありません（識別子でなければなりません）。
dereference a pointer to void	void へのポインタは間接参照できません。
directive has unexpected non-whitespace preceding newline	#line 宣言内に不適切な文字があります。
division by zero	ゼロ除算を行っています。
division by zero in #if constant expression	#if の式中にゼロ除算があります。
double quoted strings not allowed in #if constant expression	#if の式においてダブルクォーテーションで囲まれた文字列は指定できません。
duplicate case label in switch statement	switch 文に重複する case があります。
duplicate definition of enumeration-constant	enum 定数に重複する定義があります。
empty constant expression in #if	#if の式がありません。

表 14.6　コンパイラのエラーメッセージ（4/15）

メッセージ	解説
enum used as type is incomplete or undefined	不完全もしくは未定義の enum が使われています。
enumeration-constant out of range	enum 定数の値が範囲を超えています。
EOF in comment	コメントの終わりがありません。
expected parameter list missing -or- missing type for variable	引数または初期化式が見つかりません。あるいは変数の型が不明です。
fewer arguments are specified than declared	宣言に対して引数が足りません。
first operand of conditional operator must have scalar type	条件演算の第 1 項はスカラ型でなければなりません。3 項演算はスカラ型でなければなりません。
first operand of "." is not struct/union type	演算子 .（ピリオド）は構造体および共用体に使用しなければなりません。
first operand of "->" must be pointer to struct/union type	演算子 -> は構造体および共用体のポインタに使用しなければなりません。
floating point numbers are not allowed in #if constant expression	#if の式中で浮動小数点数は指定できません。
function already defined	関数はすでに定義されています。
function cannot be an array element	配列の要素として関数は作れません。
function cannot return a function	関数は関数型を返すことはできません。
function cannot return an array	関数は配列を返すことはできません。
function missing parameter declaration	関数に引数宣言がありません。

表 14.6　コンパイラのエラーメッセージ（5/15）

メッセージ	解説
function return type incompatible with previous declaration	関数の戻り値の型が、以前に宣言された型と異なっています。
function return type is not declared	関数の戻り値の型が宣言されていません。
garbage after end of constant expression in #if	#if の定数式の後ろに不適切な文字があります。
identifier already used as member name in this struct/union	メンバ名はすでに使用されています。
identifier cannot have type "void"	識別子を void 型として定義することはできません。
identifier is not member of left hand side struct/union	構造体または共用体に定義されていないメンバを使用しています。
xxx: identifier is required between '(' and ')' in `defined ( )'	defined の()内には識別子が必要です。
identifier must be defined as a typedef-name	typedef で定義されていない名前を使用しています。
identifier not member of left-hand side struct/union	構造体または共用体に定義されていないメンバを使用しています。
identifier redeclared in current declaration	重複宣言されている識別子があります。
identifier redefined in current declaration	重複定義されている識別子があります。
identifier undeclared in current declaration	宣言されていない識別子があります。
identifier undefined	未定義の識別子があります。
identifier with no linkage and incomplete object type	リンケージがなく、不完全な型を持つ識別子です。

表 14.6　コンパイラのエラーメッセージ（6/15）

メッセージ	解説
illegal combination of types in initialization	初期化における型の組み合わせが正しくありません。
illegal floating point constant	浮動小数点定数の記述が正しくありません。
illegal hexadecimal constant	16 進定数の記述が正しくありません。
illegal octal constant	8 進定数の記述が正しくありません。
implicit declaration conflicts with prior (possibly implicit) declaration	暗黙の宣言が以前の暗黙の宣言と矛盾しています。
initializer must be constant expression	初期化は定数式でなければなりません。
integer character constant requires one or more multibyte characters enclosed in single-quotes	文字定数は 1 つ以上の文字をシングルクォーテーションで囲まなければなりません。
invalid character constant in #if	#if において文字定数の指定が間違っています。
invalid combination of types in assignment	代入式において型の組み合わせが正しくありません。
invalid file name in #*xxx*	前処理命令 #*xxx* で指定しているファイル名が不適切です。
xxx: invalid file name in #include	#include で指定しているファイル名が不適切です。
invalid initializers	初期化が正しくありません。
invalid line number in #line	#line の行番号指定が不適切です。
xxx: invalid macro name	マクロ名が正しくありません。

表 14.6　コンパイラのエラーメッセージ（7/15）

メッセージ	解説
invalid macro name	マクロ名が正しくありません。
invalid parameter name in #define	#define のパラメータ名が不適切です。
invalid token in #if constant expression	#if の定数式中に不適切なトークンがあります。
#*xxx* is not within a conditional	前処理命令 #*xxx* の位置が不適切です。
`)' is required after `defined (macroname'	defined に続く識別子を囲む、カッコが足りません（閉じられていません）。
left operand of assignment operator must be modifiable lvalue	代入式の左辺値は変更可能でなければなりません。
less parameters than definition	関数原型宣言における引数の数が不足しています。関数定義における引数の数と一致しません。
line number shall not specify zero, nor a number greater than 32767	行番号は 1 から 32767 の範囲でなければなりません。
xxx: macro body/parameter-list redefined	マクロ定義、引数リストが再定義されています。
macro name `*xxx*' is reserved	*xxx* は予約マクロです。
macro name missing after -D option	-D オプションにマクロ名が指定されていません。
macro name missing after -U option	-U オプションにマクロ名が指定されていません。
member fields with bit size of zero must not be named	サイズが 0 のビットフィールドには名前をつけられません。
more arguments are specified than declared	宣言よりも引数が多すぎます。

表 14.6 コンパイラのエラーメッセージ（8/15）

メッセージ	解説
more parameters than definition	関数原型宣言における引数の数が多すぎます。関数定義における引数の数と一致しません。
new style function can not have an old style parameter declaration	古い引数の宣言方法と新しい引数の宣言方法が混在してはなりません。
newline character not allowed	文字定数に改行文字を含めることはできません。
xxx: No such file or directory	インクルードファイル *xxx* が読み込めません。
no arguments to macro `*xxx*'	マクロ *xxx* に引数がありません。
no repetition of type qualifier in declaration	型修飾子は繰り返して宣言できません。
non-integral constant-expression for enumeration-constant	enum 定数に非整数定数を定義しようとしています。
nonintegral initialization of bitfield	ビットフィールドを非整数型で初期化しようとしています。
#: not first non-whitespace character	行の先頭以外に # が記述されています。
not in any file	__FILE__ が展開できません。
object with block scope and external or internal linkage can not be initialized	外部／内部リンケージを持つオブジェクトは初期化できません。
only *xxx*a argument(s) to macro `*xxx*b' (*xxx*c argument(s) expected)	マクロの引数が不足しています（マクロ *xxx*b では *xxx*c 個の引数が要求されますが *xxx*a 個しか指定されていません）。
only leftmost dimension of array may be incomplete	配列の最左端の次元が不完全です。
only one default statement allowed per switch statement	switch 文に default 文は 1 つしか記述できません。

表 14.6　コンパイラのエラーメッセージ（9/15）

メッセージ	解説
operand of cast can not be type "void"	void 型以外へのキャストのオペランドにはスカラ型を指定しなければなりません。
operand of equality operator has invalid type	等値演算子のオペランドの型が正しくありません。
operand of "-" has invalid type	- 演算子のオペランドの型が正しくありません。
operand of "+" has invalid type	+ 演算子のオペランドの型が正しくありません。
operand of "++"/"--" must be modifiable lvalue	++ または -- 演算子の左辺値は変更可能でなければなりません。
operand of "++"/"--" must have arithmetic type or pointer to object type	++ および -- 演算子のオペランドは算術型もしくはポインタ型でなければなりません。
operand of "~" must have integral type	~ 演算子のオペランドは整数型でなければなりません。
operand of "++"/"--" must have scalar type	++ および -- 演算子のオペランドはスカラ型でなければなりません。
operand of "!" must have scalar type	!演算子のオペランドはスカラ型でなければなりません。
operand of "sizeof" must not be bitfield	sizeof 演算子のオペランドにビットフィールドを指定してはいけません。
operand of "sizeof" must not be function type	sizeof 演算子のオペランドに関数型を指定してはいけません。
operand of "sizeof" must not be incomplete type	sizeof 演算子のオペランドに不完全な型を指定してはいけません。
operand of unary "&" must be lvalue or function designator	& アドレス演算子のオペランドは、左辺値または関数指示子でなければなりません。
operand of unary "*" must be pointer type	* 間接参照演算子のオペランドは、ポインタ型でなければなりません。

表 14.6　コンパイラのエラーメッセージ（10/15）

メッセージ	解説
operand of unary "+"/"-" must have arithmetic type	+ および - 単項演算子のオペランドは算術型でなければなりません。
operand of unary "&" must not refer to bitfield	& アドレス演算子はビットフィールドには使用できません。
operand of unary "&" must not refer to register object	& アドレス演算子はレジスタ宣言されたオブジェクトには使用できません。
operands of "*"/"/" (multiply/divide) must have arithmetic type	*（乗算）および /（除算）演算子のオペランドは算術型でなければなりません。
operands of "&"/"^"/"|" must have integral type	&、^、および、|　演算子のオペランドは算術型でなければなりません。
operands of "<<"/">>" must have integral type	<< および >> 演算子のオペランドは整数型でなければなりません。
operands of "%" must have integral type	% 演算子のオペランドは整数型でなければなりません。
operands of "&&"/"||" must have scalar type	&& および || 演算子のオペランドはスカラ型でなければなりません。
operands of relational operator have invalid type	関係演算子（<,<=,>,>=）のオペランドの型が不適当です。
# operator should be followed by a macro argument name	# 演算子の次にはマクロ引数名がなければなりません。
parameter can not have any storage class other than register	引数には register 以外の記憶クラスは指示できません。
parameter cannot have initializer in function header	関数のヘッダ部では引数の初期化はできません。
parameter in this definition incompatible with prior use	定義されている引数が以前の宣言と矛盾しています。
parameter incompatible with previous declaration	引数が前の宣言と矛盾しています。

表 14.6　コンパイラのエラーメッセージ（11/15）

メッセージ	解説
parameter missing, declaration is not allowed	引数が不十分です。宣言と異なっています。
parameter name only is not allowed on function declaration	関数の宣言で引数が名前だけではいけません。
parameter name starts with a digit in #define	#define の初めの引数が数字で始まっています（識別子でなければなりません）。
parameter redeclared in current declaration	引数が再宣言されています。
#pragma keyword has unrecognized keyword/option	#pragma に不適切な指定があります。
prior errors have corrupted symbol type	先のエラーによってシンボルのタイプが崩されました。
proscribed type specifier combination in declaration	型の組み合わせで禁止されている宣言があります。
redefined statement label in function scope	関数内でラベルを再定義しています。
referenced label not declared in current function scope	関数内に参照すべきラベルが見つかりません。
`register' must appear within a function	関数外でレジスタ変数の宣言があります。
return type incompatible with function's defined return type	関数の戻り値が宣言されたものと異なります。
return with expression in function with "void" return type	void 宣言された関数から値を返そうとしました。
same argument name used in macro *xxx*	マクロで同じ引数が使われています。
second or third operand of conditional operator has invalid type	三項演算で不適切な型が指定されています。

表 14.6　コンパイラのエラーメッセージ（12/15）

メッセージ	解説
size of return type from function must be known	関数の戻り値のサイズが不明です。
sizeof object is of unknown length	sizeof 演算に長さの分からないオブジェクトを指定しています。
xxx: : storage class incompatible with subsequent file scope declaration	記憶クラス宣言が次のファイルスコープ宣言と矛盾しています。
xxx: storage class is nonvalid for formal argument	記憶クラス宣言が正しくありません。
struct or union must not contain member of function type	構造体、共用体のメンバに関数型を含めることはできません。
struct used as type is incomplete or undefined	構造体が不完全または未定義です。
struct/union improperly defined, possibly as a result of previous error(s)	構造体、共用体が不適当です。以前にエラーとなった可能性があります。
subscript must be positive, non-zero, integral value	配列のサイズの指定は、0 より大きい整数列の定数でなければなりません。
syntax error	文法エラーです。
syntax error: at or near symbol *xxx*	シンボル *xxx* の付近に文法エラーがあります。
syntax error in #if constant expression:	#if の定数式の記述に文法エラーがあります。
tag conflicts with prior struct/union appearance	タグが、前述の struct/union のものと矛盾しています。
tag conflicts with prior struct/union/enum appearance	タグが、前述の struct/union/enum のものと矛盾しています
tag redeclared in current declaration	再宣言されているタグがあります。

表 14.6　コンパイラのエラーメッセージ（13/15）

メッセージ	解説
xxx: tag use inconsistant with previous definition	タグ *xxx* の使用が以前の定義と矛盾しています。
xxx: tentative definition static object shall not have incomplete type at declaration	static オブジェクトの型宣言が不完全です。
too many arguments to macro `*xxx*' (*n* arguments)	マクロ *xxx* の引数が多すぎます（*n* 個あります）。
too many characters in a character constant	文字定数中の文字が多すぎます。
too many initializers	初期化が多すぎます。
too many parameters for 'asm'	asm 関数の引数が多すぎます。
type cannot be inherited from a typedef	この型は typedef で受け継ぐことができません。
type incompatible with subsequent definition	後の定義と互換性がありません。
xxx: type incompatible with subsequent definition	型が次の定義と矛盾しています。
type of argument does not match with prototype	引数の型が、関数原型宣言と一致しません。
type of 'asm' parameter must be integral	asm 関数の引数は整数型でなければなりません。
xxx: type of initialized entity can not be function type	初期化された構成要素の型は関数型にできません。
type of struct member is an array of unknown length	構造体のメンバに大きさの分からない配列が宣言されています。
type used for this symbol is incomplete or undefined	このシンボルの型は不完全もしくは未定義です。

表 14.6　コンパイラのエラーメッセージ（14/15）

メッセージ	解説
unable to evaluate case label (out of range?)	case で評価できません。
unbalanced #endif	#endif の対応がとれていません（無意味な #endif の指定があります）。
undefined static function	static 宣言された関数が見つかりません。
xxx: undefining not-userdefined macro	定義していないマクロを未定義にしようとしています。
unexpected string as initializer	予期せぬ文字列です。
union used as type is incomplete or undefined	共用体名が不完全または未定義です。
xxx: unknown preprocessor directive	前処理命令が不明です。
unnamed parameter	引数名がありません。
unterminated character constant	文字定数の終わりが見つかりません。
unterminated comment	コメントの終わりが見つかりません。
unterminated #*xxx* conditional	#*xxx* 命令に対する #endif 命令がありません。
unterminated macro call	マクロコールの終わりが見つかりません。
unterminated string	文字列の終わりが見つかりません。
file inclusion is too deep (up to xxx nesting level)	ファイルのネストが深すぎます。

表 14.6　コンパイラのエラーメッセージ（15/15）

メッセージ	解説
operand of cast to pointer must not be a floating	浮動少数点は、ポインタへのキャストのオペランドには使えません。
operand of cast to floating must not be a pointer	ポインタは、浮動少数点へのキャストのオペランドには使えません。
member array size too large	構造体メンバが大きすぎます。

14.2.5 重大エラー

表 14.7　コンパイラの重大エラーメッセージ

メッセージ	解説
no memory available	メモリが不足しています。
out of memory	メモリが不足しています（最適化処理時）。
prior errors have corrupted symbol scoping	以前のエラーでスコープが壊されています。

付録A
拡張機能リファレンス

CC32Rは、M32Rファミリを用いたシステムへの組み込みを容易にするために独自の拡張機能を追加しています。

付録Aでは、言語仕様以外に関する拡張機能の使用方法を説明します。

表A.1　拡張機能

拡張機能	機能の内容
ベースレジスタ機能	1．16ビットレジスタ相対間接アドレッシングの基点(ベース)となるレジスタ(ベースレジスタ)を複数(ベースレジスタ専用レジスタを設定)設けて、変数毎にどのベースレジスタの相対(16ビットレジスタ相対間接アドレッシング)であるかを指定することで、コードサイズを縮小させる機能です。 2．本機能を使用するには、 (1) アクセス制御ファイル(Access Control File)※ (2) コンパイルオプション"-access=アクセス制御ファイル"が必要です。
メモリモデル	1．コンパイル時に、コード(P,Cセクション)とデータ(D,Bセクション)が収まるアドレス空間上での大きさや位置に応じて、最適なオブジェクトを生成するために、アプリケーションのアドレス空間上での格納パターンをいくつか想定したものです。 2．本コンパイラには４種類のメモリモデルがあります。 (1) スモールモデル (2) スモールモデル(C/C++コンパイラオプション"-memlarge"付き) (3) ミディアムモデル (4) ラージモデル
#pragma 拡張機能	1．C/C++言語から、M32Rファミリハードウェア仕様を効率よく活かすための拡張機能を使用できます。
インライン展開機能	インライン展開機能は、関数を呼び出すかわりに、呼び出す関数の中身を直接展開する機能です。サブルーチンジャンプ命令(BL)などのオーバーヘッドを省略できるため、インライン展開機能によって通常の関数呼び出しより速度面で有利なコードを得ることができます。
M32R/ECU#5 (M32R-FPUコア)対応機能	M32R/ECUシリーズ32180および32182グループ(以下、M32R/ECU#5と略します)の拡張命令(FPU命令など)に対応しました。
日本語処理機能	1．プログラムの文字定数に日本語文字を記述することができます。 2．日本語文字をマルチバイト文字やワイド文字として処理することができます。 3.日本語文字対応の文字コードとして、JIS(EUC,シフトJIS)、Unicode(UTF-8)を使用できます。 4.環境変数の操作により、ソースファイルはシフトJIS、ターゲットではUTF-8といった柔軟な対応も可能です。

※ アクセス制御ファイル(Access Control File)は、
(1) 16ビットレジスタ相対間接の基点(ベース)となるアドレス
(2) 基点(ベース)となるアドレスを保持するレジスタ
(3) ベースレジスタ機能を適応するオブジェクト(変数や構造体)。
を記述したファイルです。詳しくは、「A.1.7 アクセス制御ファイルについて」を参照して下さい

A.1 ベースレジスタ機能

A.1.1 ベースレジスタ機能とは

ベースレジスタ機能とは、16 ビットレジスタ相対間接アドレッシングの基点（ベース）となるレジスタ（ベースレジスタ）を複数(ベースレジスタ専用レジスタを設定)設けて、変数毎にどのベースレジスタの相対（16 ビットレジスタ相対間接アドレッシング）であるかを指定することで、コードサイズを縮小させる機能です。

CC32R の生成するコードのうち、

(1) Dセクション（初期値ありデータ領域）あるいは、Bセクション（初期値なしデータ領域）に割り付けられるオブジェクト(変数、構造体など)に対するアクセスについてのコードを対象として、16 ビットレジスタ相対間接アドレッシングを用いたコードを生成します。

(2) 固定アドレス上のオブジェクトに対するアクセスの２つのアクセス（読み／書き）についてのコードを対象として、16 ビットレジスタ相対間接アドレッシングを用いたコードを生成します。

本機能を使用するには、

- アクセス制御ファイル(Access Control File)※1
- コンパイル時に、コンパイルオプション ”-access=アクセス制御ファイル ”

が必要です。

また、アクセス制御ファイルは、マップジェネレータ ”map32R ”※2 を使用して、生成することが出来ます。

※1.アクセス制御ファイル(Access Control File)は、
(1) 16ビットレジスタ相対間接の基点(ベース)となるアドレス
(2) 基点(ベース)となるアドレスを保持するレジスタ
(3) ベースレジスタ機能を適応するオブジェクト(変数や構造体)。
を記述したファイルです。詳しくは、「A.1.7 アクセス制御ファイルについて」を参照して下さい

※2.マップジェネレータ ”map32R ”に付いては、「CC32R ユーザーズマニュアル <<アセンブラ編 >> 第２章 マップジェネレータmap32」を参照して下さい。

A.1.2 ベースレジスタ機能の対象となるアクセスの種類とコード出力

A.1.2.1 変数へのアクセス

a. 対象となる変数

アクセス制御ファイルでオブジェクト登録行に記述されていて、「A.2.2 ベースレジスタ機能の対象となるオブジェクト」に該当している変数が、ベースレジスタ機能の対象となります。
(#pragma ADDRESS で配置アドレスを定義されたオブジェクトも、同様に扱われます。)

b. 生成コード

変数へのアクセスに対するコード出力の際、「変数のラベル -(ハイフン) ベースシンボル」という式を相対間接のオフセットに使用します。
ベースシンボルは、対応するベースアドレスを示し、__REL_BASExx の形式(xx にはレジスタ番号(11～13)が入ります)です。ただし、コンパイラの出力コード中ではこのシンボルは定義していない(.IMPORT で参照している)ので、スタートアップファイル等で値を定義する必要があります。

[コード出力例]
ベースが R12 の場合で、大域変数 variable のアクセスに対するコード出力の場合を示します。
(例ではvariableはunsigned char型)。

例1) 書き込み

[ベースレジスタ機能なし]	[ベースレジスタ機能を使用]
LD24 R0,#_variable STB R1,@R0	**STB R1,@(_variable-__REL_BASE12,R12)**

例2) 読み出し

[ベースレジスタ機能なし]	[ベースレジスタ機能を使用]
LD24 R0,#_variable LDUB R0,@R0	**LDUB R1,@(_variable-__REL_BASE12,R12)**

A.1.2.2 定数アドレスへのアクセス

a. 対象となる定数

定数を下記「A.1.3 ベースレジスタ機能の対象となるオブジェクト」へのポインタにキャストして、ポインタの指す領域をアクセスする場合、このアドレスがベースアドレスを中心に （ベースアドレス-0x8000 ～ ベースアドレス+0x7FFF）の範囲にある場合、ベースレジスタ機能の対象となります。

```
例1) *((int*)0x12345678) = 100;  /* 0x12345678番地への書き込み */
例2) return *((int*)0x12345678); /* 0x12345678番地からの呼び出し */
```

b. 生成コード

定数アドレスへのアクセスに対するコード出力の際、「固定アドレス,(カンマ)ベースアドレス」という式を相対間接のオフセットに使用します。

[コード出力例]
ベースレジスタが R13 = 12340000(16進)の場合

例1) 書き込み

[ベースレジスタ機能なし]	[ベースレジスタ機能を使用]
SETH R0,#HIGH(0x12345678) OR3 R0,R0,#LOW(0x12345678) ST R1,@R0	**ST R1,@(0x5678,R13)**

例2) 読み出し

[ベースレジスタ機能なし]	[ベースレジスタ機能を使用]
SETH R0,#HIGH(0x12345678) OR3 R0,R0,#LOW(0x12345678) LD R0,@R0	**LD R0,@(0x5678,R13)**

A.1.3 ベースレジスタ機能の対象となるオブジェクト

A.1.3.1 記憶クラス・リンケージ

ベースレジスタ機能の対象となるオブジェクトは、メモリに静的に割り付けられるオブジェクトである必要があります。

[関数外オブジェクト]
(1) 大域変数
(2) static大域変数
(3) extern大域変数
[関数内オブジェクト]
(4) 関数内static変数
(5) ブロック内static変数

A.1.3.2 型の種類

ベースレジスタ機能の対象となるオブジェクトの型は、次のとおりです。

(1) 整数型(char/short/long,signed/unsigned,enum)
(2) 浮動小数点(float/double/longdouble)
(3) 構造体(ビットフィールドメンバを持つ構造体を含む)
(4) 共用体
(5) 配列
(6) ポインタ

A.1.3.3 型修飾子の種類

ベースレジスタ機能の対象となるオブジェクトの型の修飾子は、次のとおりです。

(1) 型修飾無し

(2) volatile

A.1.4 ベースレジスタ機能の対象とならないオブジェクト

A.1.4.1 型・派生型などの種類

(1) メンバ名 (構造体として指定することは可能)

(2) 関数

(3) 定数

A.1.4.2 記憶クラス、ストレージ

(1) 関数外オブジェクトと同じ名前の、関数内およびブロック内のstatic変数

(2) 関数内オブジェクトと同じ名前の大域変数

(3) auto

(4) register

(5) typedef

A.1.4.3 修飾子

(1) const

A.1.5 ベースシンボルとベースレジスタの設定

ベースレジスタ機能では、コンパイラの出力コードで 「(1)ベースシンボルの定義」と 「(2)ベースレジスタの初期化」を行わないので、これらはスタートアッププログラム等に記述する必要があります。
以下の例は、ベースレジスタを R11～R13 の 3 本を使用している場合のスタートアップから該当部分を抜粋したものです。
なお、例では R11～R13 を 3 本全てをベースレジスタに割り当た場合ですが、割り当てないレジスタは設定を行う必要はありません。

【注意】
アクセス定義ファイルにベースアドレスを記述している場合は、ベースシンボルの値をそのアドレスと同一値で指定する必要があります。

(1) ベースシンボルの定義 (.exportは.globalでも可)

```
        .EXPORT __REL_BASE11
        .EXPORT __REL_BASE12
        .EXPORT __REL_BASE13

__REL_BASE11:   .EQU  0x10000000
__REL_BASE12:   .EQU  0x20000000
__REL_BASE13:   .EQU  0x30000000
```

(2) ベースレジスタの初期化

```
SETH    R11,#HIGH(__REL_BASE11)
OR3     R11,R11,#LOW(__REL_BASE11)

SETH    R12,#HIGH(__REL_BASE12)
OR3     R12,R12,#LOW(__REL_BASE12)

SETH    R13,#HIGH(__REL_BASE13)
OR3     R13,R13,#LOW(__REL_BASE13)
```

A.1.6 ベースレジスタ機能の制限事項

(1) オフセットが 32768 以上の場合
非常に大きな構造体や配列で、先頭から 32768 バイト以上のオフセットの位置にあるメンバや要素をアクセスする場合、ベースレジスタは適用されません。

(2) ベースアドレスの重複
ベースレジスタがカバーする範囲が重複していて、「A.1.2.2 定数アドレスへのアクセス」に該当するものが、その重複空間上の場合は、先にベースアドレスを定義していたベースレジスタが割り当てられます。

(3) ベースレジスタ設定の混在に関する制限
ベースレジスタ機能を利用する場合は、全ての C/C++ ソースプログラムに同一のアクセス制御ファイルを指定してコンパイルするようにしてください。
当社では、異なるベースレジスタの設定によりコンパイルされたオブジェクト同士をリンクすることは推奨しておりません。

A.1.7 アクセス制御ファイルについて

アクセス制御ファイル(Access Control File)は、ベースレジスタ機能を使用するために必要な情報、

(1) 16ビットレジスタ相対間接の基点(ベース)となるアドレス
(2) 基点(ベース)となるアドレスを保持するレジスタ
(3) ベースレジスタ機能を適応するオブジェクト(変数や構造体)。

を記述したファイルです。このファイルをコンパイラのオプション ”-access=アクセス制御ファイル”に指定します。
また、アクセス制御ファイルは、マップジェネレータ ”map32R ”※を使用して、生成することが出来ます。

A.1.7.1 アクセス制御ファイルの記述内容

アクセス制御ファイルには、以下の内容を記述します。

(1) ベースアドレス

ベースレジスタ機能（レジスタ相対間接）のベースとなるアドレスです。

ベースアドレスを中心に 「ベースアドレス-0x8000 ～ ベースアドレス+0x7FFF」 の空間へ固定アドレスでアクセス（読み／書き）している場合、レジスタ相対間接としてコード生成対象となります。

(2) ベースレジスタ

ベースアドレスを保持するレジスタです。

R11、R12、R13 を割り当て可能です。
ベースレジスタに割り当てたレジスタは、関数内ではワークなどの他の用途に利用されません。

(3) 対象オブジェクト

ベースレジスタ機能（レジスタ相対間接）を適用したいオブジェクトの指定です。

コンパイル時点では、オブジェクト(変数、構造体、配列など)の割付アドレスが決まっていないので、個々のオブジェクトをアクセス制御ファイルに登録する必要があります(#pragma ADDRESS の場合も含む)。

※.マップジェネレータ ”map32R ”については、「CC32R ユーザーズマニュアル <<アセンブラ編 >> 第３部 マップジェネレータmap32R」を参照して下さい。

A.1.7.2 アクセス制御ファイルの文法

アクセス制御ファイルは、行単位で記述します。

(1) コメント行

[記述形式]	[形式1]	;コメント
	[形式2]	(空行)
[機能]	コメントを記述します。コメント行は無視されます	
[例]	;This is comment 1 ⟵	[形式1]のコメント
	⟵	[形式2]のコメント(空行)
	;This is comment 2 ⟵	[形式1]のコメント

(2) ベースレジスタ定義行

[記述形式]	@ベースレジスタ名 [ベースアドレス]
[機能]	ベースアドレスと、ベースレジスタとして割り当てるレジスタを定義します (a) ベースレジスタ ・同じレジスタを、2回以上ベースレジスタとして指定することはできません。 (b) ベースアドレス ・省略可能。省略すると固定アドレスへのアクセスは非対象になります。 ・'0x' に続けて16進数(8桁以内)を記述します。 ・ベースアドレス 0xffffffff は予約されているので指定できません。
[例]	@R13 0xF78000 @R12

(3) オブジェクト登録行

[記述形式]	[形式1]	オブジェクト名
	[形式2]	オブジェクト名 | ソースファイル名 | 関数名
	[形式 3]	*
	[形式 4]	* セクション名
[機能]	ベースレジスタ定義行の後に、ひとつまたは複数記述することができます そのベースレジスタを使ってレジスタ相対間接にしたいオブジェクト名を指定します。 (a) 形式1: ・グローバル変数とファイル内staticが対象になります。 (b) 形式2: ・指定ファイルの指定関数の中にある変数が対象になります。 (c) 形式3: ・アクセス制御ファイル中の他のオブジェクト定義行に書かれていない全ての静的オブジェクトが対象になります(ワイルドカード指定)。 ・形式3は、アクセス制御ファイル中にただ1回だけ記述できます。	

	(d) 形式4: ・アクセス制御ファイル中の他のオブジェクト定義行([形式3]を除く)に書かれていない、指定セクションに属する全ての静的オブジェクトが対象になります(セクション指定つきワイルドカード指定)。 ・指定されるセクション名は、(1)#pragma SECTIONの変更セクション名、(2)規定セクション名 のいずれかです。これらに該当するセクションに割付けられる静的オブジェクトをベースレジスタの対象とします。 ・規定セクションがPまたはCであるセクション名を指定しても、その指定は無視します。 ・同じセクション名に対する形式4の指定は、アクセス制御ファイル中にただ1回だけ記述できます。		
[例]	var1	←	[形式1]の記述
	var2|sample.c|func	←	[形式2]の記述
	*	←	[形式3]の記述
	*D1	←	[形式4]の記述

A.1.7.3 アクセス制御ファイルの記述のヒント

●関数内の static変数を指定するには

変数名として「変数名 | ファイル名 | 関数名」を指定すると、関数内の static 変数がベースレジスタの対象になります。

(例)　　var2|sample.c|func

これは、ソースファイル sample.c の関数 func 中にある、var2 という変数を指定しています。なお、指定した関数に属さない var2 には無効です。

●すべての変数を一度に指定するには

変数名に「*」を記述すると、データ領域（セクション D,B）に割り付けられる変数のうち、他で指定されなかったすべての変数が対象となります（ワイルドカード指定)。

(例）グローバル変数 var1 にベースレジスタ R13 を適用し、その他のデータ領域の変数には、全てベースレジスタ R12 を適用する場合のアクセス制御ファイルの記述例。

```
(例)
@R13  0x00F78000
var1
@R12  0x00F88000
*
```

●コメント

(1)行頭に ;(セミコロン）があると、行末までコメントとみなします。

(2)変数名の後ろに空白を見つけると、それ以降行末までコメントとみなします。

（ただし、”| ”文字はコメントには含むことが出来ません。）

●空白

(1)タブは空白として処理します。

(2)行頭の連続した空白は、無視します（セミコロンの前の空白も同様です)。

●重複指定

次の記述は、重複して記述できません。

・同じベースレジスタ名

・同じ変数名

・ワイルドカード指定

A.1.8　ベースレジスタ機能の使用例

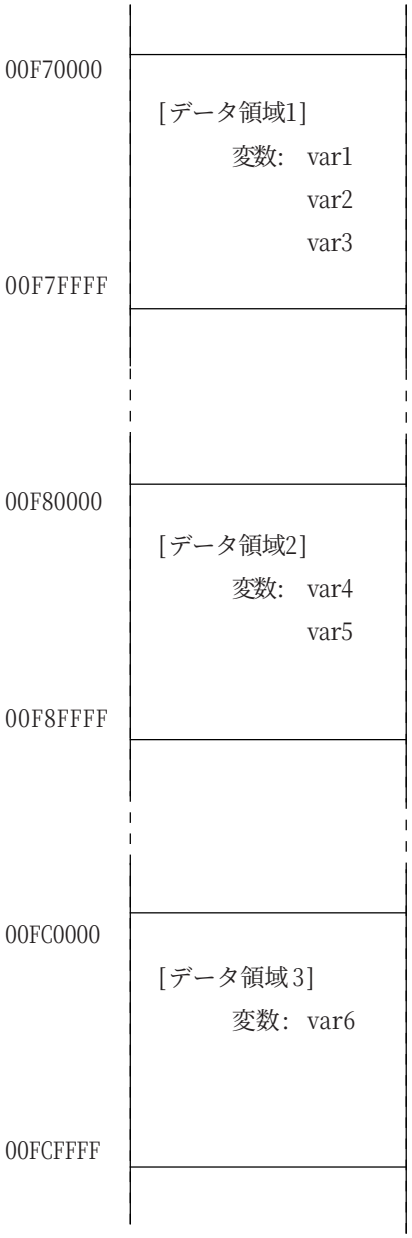
ベースレジスタ機能の使い方について説明します。

A.1.8.1　使用例

下記のように64Kバイトのデータ領域が3つあり、それぞれにベースレジスタを割り当てる場合の基本的な手順(下記[1]～[5])ついて説明します。

・データ領域1,2,3にはベースレジスタR13,R12,R11の3本をそれぞれ割り当てるものとします

・また、データ領域1にはvar1,var2,var3、データ領域2にはvar4,var5、データ領域3 にはvar6という名前のグローバル変数をそれぞれ含むものとします。



[1] ベースアドレスの決定

ベースアドレスは、ベースレジスタにセットしておく固定アドレスです。

M32R のレジスタ相対間接の仕様により、ベースレジスタは次の範囲をカバーできます。

ベースアドレス -0x8000 ～ ベースアドレス +0x7FFF

データ領域 1 ～ 3 をカバーするためのベースアドレスは、それぞれ

データ領域 1	0xF78000
データ領域 2	0xF88000
データ領域 3	0xFC8000

ですので、これらを R13,R12,R11 に割り付けるようにします。

[2] アクセス制御ファイルの作成

ベースアドレスを決定したら、次のようなベースレジスタ機能の詳細設定を行う「**アクセス制御ファイル**」を作成します。(ファイル名を sample.acc とします)。

sample.acc

```
@R13   0xF78000
var1
var2
var3

@R12   0xF88000
var4
var5

@R11   0xFC8000
var6
```

アクセス制御ファイルには、上記のように、

'@' ベースレジスタ名　'0x' ベースアドレス(16進数)

という記述によりベースレジスタを定義し、そのベースレジスタの対象としたいグローバル変数名をその行の後に並べていきます。

・ロードモジュールからアクセス制御ファイルを自動生成させることもできます。「CC32R ユーザーズマニュアル <<アセンブラ編>> 第２章 マップジェネレータ map32」を参照ください

・関数内のstatic変数を指定するには、「A.1.7.3 アクセス制御ファイルの記述のヒント」を参照ください。

[3] コンパイル

[2]で作成したアクセス制御ファイルを指定してコンパイルします。

コンパイラのコマンドライン指定に　-access=sample.acc　を加えます。

[出力コード例]

var1, var4 が int 型のグローバル変数で、sample.acc を指定してコンパイルすると、var1, var4に対するコードは次のようになります。

● C言語

```
(式1) var1 = 5;          /* var1の書き込み */
(式2) returnvar4;        /* var4の読み出し */
```

● ベースレジスタ機能適用時の出力コード

```
(式1)          LDI   R0,#5
               ST    R0,@(_var1-__REL_BASE13,R13)

(式2)          LD    R0,@(_var4-__REL_BASE12,R12)
```

[4]　　ベースシンボルの定義とベースレジスタの設定処理

次に、スタートアッププログラムに、(a)ベースシンボルの定義 と (b)ベースレジスタの設定を行うコードを追加します。

ベースシンボルは __REL_BASExx(xx はベースレジスタのレジスタ番号)という名前で、ベースアドレスを表します。コード生成時にオフセット計算のために必要となるシンボルです（前述「[3]コンパイル」の [出力コード例]を参照ください)。

以下の例はベースレジスタ R13 ～ R11 の場合の設定です。

(a) ベースシンボルの定義

```
.EXPORT __REL_BASE13
.EXPORT __REL_BASE12
.EXPORT __REL_BASE11
__REL_BASE13:  .EQU  0x00F78000
__REL_BASE12:  .EQU  0x00F88000
__REL_BASE11:  .EQU  0x00FC8000
```

(b) ベースレジスタの設定処理

```
SETH  R13,#HIGH(__REL_BASE13)
OR3   R13,R13,#LOW(__REL_BASE13)
SETH  R12,#HIGH(__REL_BASE12)
OR3   R12,R12,#LOW(__REL_BASE12)
SETH  R11,#HIGH(__REL_BASE11)
OR3   R11,R11,#LOW(__REL_BASE11)
```

【参考】

・アクセス制御ファイルの生成と同様に、map32R を使用してベースレジスタ設定プログラムを自動的に作成できます。「CC32R ユーザーズマニュアル <<アセンブラ編>> 第２章 マップジェネレータ map32」を参照ください。

[5]リンク

[4]で作成したプログラムも含め、構成プログラム全てをリンクします。

A.2 メモリモデル

A.2.1 メモリモデルとは

メモリモデルとは、コンパイル時に、コード（P,Cセクション）とデータ（D,Bセクション）が収まるアドレス空間上での大きさや位置に応じて、最適なオブジェクトを生成するために、アプリケーションのアドレス空間上での格納パターンをいくつか想定したものです。
このため、アプリケーションの規模に応じて最適なオブジェクトを生成することができます。
本コンパイラで用意された、４種類のメモリモデルを以下に示します。

メモリモデル名	コード・データで使用可能な アドレス空間	C標準ライブラリ
スモールモデル	コード　0x00000000～0x00FFFFFF データ　0x00000000～0x00FFFFFF	m32RcR.lib
スモールモデル （-memlarge付き）	コード　0x00000000～0x00FFFFFF データ　0x00000000～0xFFFFFFFF （32ビットメモリ空間全て）	m32RcRM.lib
ミディアムモデル	コード　任意のアドレスA～A+0x00FFFFFF データ　0x00000000～0xFFFFFFFF （32ビットメモリ空間全て）	m32RcRM.lib
ラージモデル	コード　0x00000000～0xFFFFFFFF （32ビットメモリ空間全て） データ　0x00000000～0xFFFFFFFF （32ビットメモリ空間全て）	m32RcRL.lib

*C標準ライブラリ を使用される場合には、各メモリモデルに対応したライブラリファイルを使用してください。

A.2.2 メモリモデルの詳細

それぞれのメモリモデルについて説明します。

■ スモールモデル

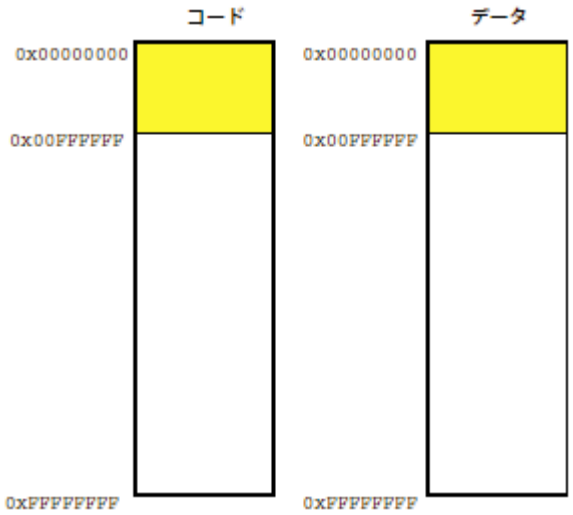
スモールモデルは、アプリケーションのコード・データ共に、
　アドレス空間上　0x00000000～0x00FFFFFF（図A.1の色付きの部分）
の範囲に収まることを前提としたメモリモデルです。
このメモリモデルでコンパイルする場合は、コンパイル時にオプション

```
-small
```

を指定します。
メモリモデルを指定しない場合は、スモールモデルが指定されたものとして、コンパイルされます。
このメモリモデル対応のC標準ライブラリは、

```
m32RcR.lib
```

となります



図A.1 スモールモデルのアドレス空間

■ スモールモデル（-memlarge付き）

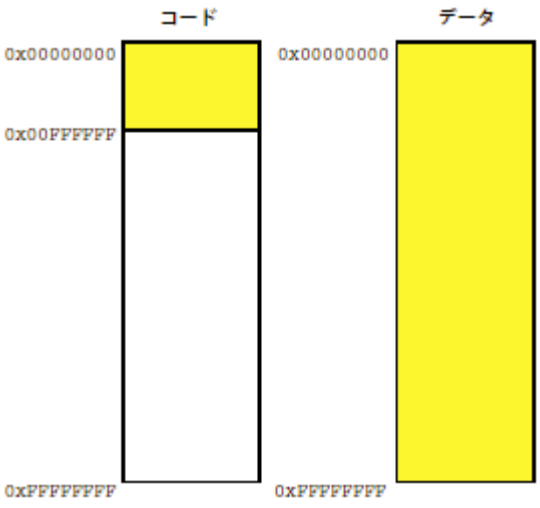
スモールモデル（-memlarge付き）は、
アプリケーションのコードが 、
　アドレス空間上 0x00000000～0x00FFFFFF（図A.2の色付きの部分）
アプリケーションのデータが、
　アドレス空間上 0x00000000～0xFFFFFFFF（図A.2の色付きの部分）
の範囲に収まることを前提としたメモリモデルです。
このメモリモデルでコンパイルする場合は、コンパイル時にオプション

```
-small -memlarge
```

を指定します。
このメモリモデル対応のC標準ライブラリは、

```
m32RcRM.lib
```

となります。



図A.2 スモールモデル （-memlarge）のアドレス空間

■ ミディアムモデル

ミディアムモデルは、アプリケーションのコードが

アドレス空間上任意のアドレス”A”～”A” + 0x00FFFFFF（図A.3の色付きの部分）

アプリケーションのデータが、

アドレス空間上 0x00000000～0xFFFFFFFF（図A.3の色付きの部分）

の範囲に収まることを前提としたメモリモデルです。

このメモリモデルでコンパイルする場合は、コンパイル時に、オプション

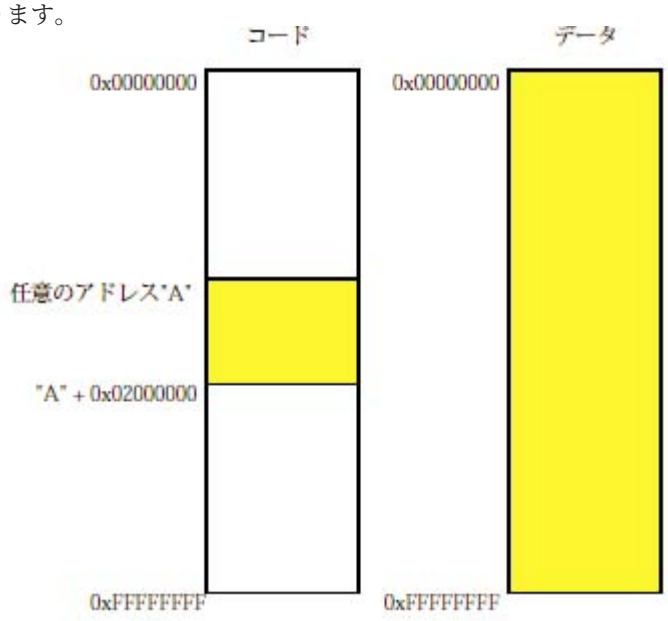
```
-medium
```

を指定します。

このメモリモデル対応のC標準ライブラリは、

```
m32RcRM.lib
```

となります。



図A.3 ミディアムモデルのアドレス空間

■ ラージモデル

ラージモデルは、アプリケーションのコード・データ共に、

アドレス空間上 0x00000000～0xFFFFFFFF（図A.4の色付きの部分）

の範囲に収まることを前提としたメモリモデルです。

このメモリモデルでコンパイルする場合は、コンパイル時に、オプション

```
-large
```

を指定します。

このメモリモデル対応のC標準ライブラリは、

```
m32RcRL.lib
```

となります。

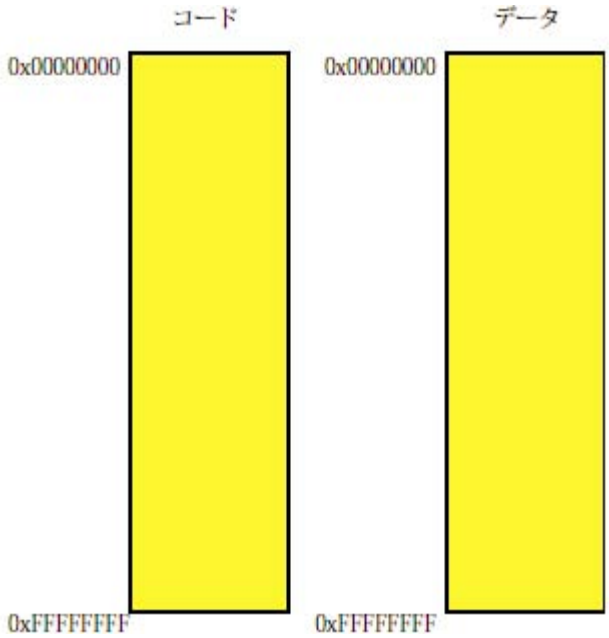


図 A.4 ラージモデルのアドレス空間

A.3 #pragma 拡張機能

A.3.1 #pragma　拡張機能の一覧

#pragmaに関する拡張機能の内容と規定を一覧表で示します。

表 A.2　#pragma 拡張機能一覧

拡張機能	機能の内容
#pragma ADDRESS	指定した変数を、指定の絶対アドレスに配置するように宣言します 記述形式)#pragma ADDRESS 変数名 絶対アドレス 記述例)　#pragma ADDRESS val_1 0x1000
#pragma SECTION	コンパイラが作成するデフォルトのセクション名を変更します。 記述形式)#pragma SECTION 既定セクション名 変更セクション名 記述例)　#pragma SECTION B USR_SEC_B
#pragma INTERRUPT (#pragma INTF)	C/C++言語で記述した割り込み処理関数を宣言します。 この宣言により、関数の出入り口で割り込み処理関数の手続きを行うコードを生成します。また、関数出口ではRTE命令でリターンします。 記述形式) #pragma INTERRUPT 割り込み処理関数名 [レジスタ名△レジスタ名△・・・][※1] 記述例)#pragma INTERRUPT int_func R6
#pragma keyword asm on #pragma keyword asm off	インラインアセンブル機能(asm関数)[※2]の使用を宣言します。 これは、「asm 」を通常の識別子でなく、予約語として認識させるための設定です。再び asm を識別子として使用する場合は、 #pragma keyword asm off と指定してください。 記述形式)#pragma keyword asm on (or off) 記述例)　#pragma keyword asm on

※1. ”△ ”は、必須のスペースを示します。

※2. インラインアセンブル機能に付いては、「7.4　インラインアセンブル機能」を参照して下さい。

#pragma ADDRESS

変数の絶対アドレスの指定機能

[機　能]　**変数を、指定の絶対アドレスに配置するように宣言します。**

[書　式]　#pragma ADDRESS 変数名 絶対アドレス

[解　説]

●#pragma ADDRESSは、指定の絶対アドレスに変数を配置するように宣言します

●コンパイラは、この宣言で指定された変数名をアセンブラの.EQU疑似命令でシンボルの値として定義するコードを生成します。したがって変数の領域は確保されません。

●絶対アドレスは、文字列として.EQU疑似命令の式としてそのまま出力されます。
したがって、アセンブラの.EQU疑似命令の記述規則に依存します。なお、絶対アドレスには、次の数値表現形式の定数のみ指定可能です。また、式は定数の式のみ記述可能です。

8進数 ： 0で始まり、0～7の数字で構成される定数
10進数： 0以外の数字で始まり、0～9の数字で構成される定数
16進数： 0xか0Xで始まり、0～9およびA(a)～F(f)で構成される定数

●変数名には、次の記憶クラスとデータ型の変数名を指定することができます。

[記憶クラス]

外部定義変数(グローバル変数)
外部参照変数(extern変数)
静的定義変数(static変数、ただし関数内定義のstatic変数を除く)

[データ型]

char, short, int, long,
unsigned char, unsigned short, unsigned int, unsigned long,
long double, float, double
struct, union
enum, ポインタ

●#pragma ADDRESSの宣言は、この宣言より前および後に定義した変数に対して有効になります。

[規　定]

●同一の変数に対して2回以上#pragma ADDRESS宣言した場合、コンパイル時にエラーとなります。

●初期化式を記述した変数を指定することはできません。指定した場合、コンパイル時にワーニングとなり、その宣言は無効とされます

●関数内で定義された変数(ローカル変数、関数内定義のstatic変数)を指定することはできません。指定した場合、コンパイル時にワーニングとなり、その宣言は無効とされます。

●構造体メンバ名、共用体メンバ名、enum型メンバ名、配列要素を変数名に指定することはできません。指定した場合、その宣言は無効とされます(ワーニング表示はありません)。

●関数引数を変数名に指定することはできません。指定した場合、その宣言は無効とされます(ワーニング表示はありません)。

#pragma ADDRESS

[注意事項]

●指定した絶対アドレスが、変数のアライメントに従っているかどうかは、チェックされません。必ずデータ内部表現のアライメントに従った絶対アドレスを指定してください。

●指定した絶対アドレスにより、変数領域がオーバーラップするかどうかは、チェックされません。アドレス指定の際、十分注意してください。

●指定した絶対アドレスにより、配列や構造体のアクセスアドレスが32ビットアドレスをオーバーフローしてもチェックされません。絶対アドレスの指定時にはオーバーフローしないよう十分注意してください。

●#pragma ADDRESSの宣言により、異なる変数名に対して同一の絶対アドレスを指定することは、コンパイラの最適化処理に影響を及ぼし正しいコードの生成を妨げることがあります。同一の絶対アドレスを指定する場合、volatile修飾子を変数に指定してください。

●以下のような記述は動作異常の原因となりますが、コンパイラではチェックされません。十分注意して下さい。

[変数portaの領域と変数portbの領域がオーバーラップするアドレス指定の記述例]

```
#pragma ADDRESS porta 0x10010
#pragma ADDRESS portb 0x10012
int porta;
int portb;
```

[配列のアクセスアドレスがオーバーフローする記述例]

```
#pragma ADDRESS x 0xffffffff
char x[2];
void func(void)
{
   x[1] = 0;
}
```

[コンパイラの最適化処理に影響を及ぼす記述]

```
#pragma ADDRESS x 0x1000
#pragma ADDRESS y 0x1000
int x, y;
int func(void)
{
   x = 0;
   y += 1;※
   if (x == 0)
      return 0;
   return 1;
}
```

※ANSI-C仕様では、上述のyに対する変更(y += 1)がxの値に影響しないと解釈することが許されます。このためコンパイラは、x == 0のif文は、常に真であると解釈し、常に0を返すコードを生成することがあります。

図A.5 #pragma ADDRESS宣言でコンパイラでチェックされない記述例

#pragma ADDRESS

[使用例]　#pragma ADDRESS は、以下のようにC/C++ソースに直接記述します。

[記述例]

```
#pragma ADDRESS var1  0x10000
#pragma ADDRESS var2  0x20000
char  var1;
static  char  var2;

void
func(void)
{
      var1 = var2;
}
```

[記述例の出力（アセンブルコード）]

```
_var1:     .EQU  0x10000
           .SECTION   P,CODE,ALIGN=4
           .EXPORT    $func
$func:
           LD24  R1,#0x20000
           LD24  R0,#0x10000
           LDB   R1,@R1
           STB   R1,@R0
           JMP   R14

           .EXPORT    _var1
           .END
```

図A.6　#pragma ADDRESS宣言の使用例

記述例では、それぞれの #pragma ADDRESS により、変数var1は 0x10000番地、変数var2は 0x20000番地にそれぞれ配置されているものとみなされます。

これにより、関数 func()中の式 var1 = var2 には、0x20000番地から1バイト読み出して、0x10000番地に書き込む処理を行なうコードが生成されます。

また、var1は外部定義変数のため、.EQUでは対応するシンボル「var1」を定義し、さらに .EXPORT で他のオブジェクトから参照可能にします。

#pragma SECTION

セクション名の変更機能

[機　能]　**コンパイラが作成するデフォルトのセクション名を変更します。**

[書　式]　#pragma SECTION 既定セクション名 変更セクション名

[解　説]

●#pragma SECTIONは、コンパイラがデフォルトで作成するセクションの名前をユーザ定義の名前に変更します。この宣言は、同じ既定セクションに対する #pragma SECTION が記述されるか、ファイルの末尾まで有効です。

●既定セクション名には、コンパイラが作成する次のセクション名のいずれかを指定します。

セクション名	用途
P	プログラムコードが格納されるセクション セクション属性：CODE、配置属性：ALIGN=4
C	定数データが格納されるセクション セクション属性：DATA、配置属性：ALIGN=4
D	初期値ありデータが格納されるセクション セクション属性：DATA、配置属性：ALIGN=4
B	初期値なしデータが格納されるセクション セクション属性：DATA、配置属性：ALIGN=4

●変更セクション名には、新しいユーザ定義のセクション名を指定します。セクション名の記述規則は、アセンブラの名前の記述規則に依存します(ユーザーズマニュアル<<アセンブラ編>>「3.5 名前の参照規則」)。変更セクション名のセクションの属性(セクション属性および配置属性)は、既定セクション名の属性と同じになります。以下は、ユーザーズマニュアル「**3.5 名前の記述規則**」よりの抜粋です。

> **名前の記述規則を以下に示します。**
> **■1文字目に使用できる文字**
> **英字、ドルマーク($)、アンダーライン(_)のいずれか。**
> **先頭文字に数字は使用できません。**
> **■2文字目以降に使用できる文字**
> **英数字、ドルマーク($)、アンダーライン(_)のいずれか。**
> **■記述できる文字数**
> **・モジュール名206 文字まで**
> **・シンボル名243文字まで**
> **・セクション名243文字まで**
> **・プリプロセッサ変数、マクロ名それぞれ32文字まで**
> **■英大文字／英小文字の区別**
> **ユーザー定義できる名前の場合は区別あり。**
> **ユーザー定義できない名前の場合は区別なし。**
>
> **ユーザーが名前を定義する場合は上記の規則に従って記述します。**
> **なお、その際以下の点に注意してください。**
> **■モジュール名以外のものに、予約語と同じ名前は使用できません。**
> **■シンボルとセクション名など、ユーザー定義の名前は重複してはいけません。**

#pragma SECTION

[解　説]

●変更セクション名を既定セクション名と同じにすることにより、コンパイラのデフォルトのセクション名に戻すことができます。

● "#pragma SECTION " は、関数定義もしくはデータ定義よりも前に　記述してください。

●Dセクションはスタートアップ時に初期値で初期化する必要があります。Dセクションの名前を変更した場合、変更セクションをスタートアップ時に初期値で初期化してください。

●Bセクションは通常スタートアップ時にゼロクリアする必要があります。Bセクションの名前を変更した場合、変更セクションをスタートアップ時にゼロクリアしてください。

[規　定]

●Pセクションの名前を変更した場合、この宣言以降に定義された関数のセクション名が変更セクション名になります。

●C、D、Bセクションの名前を変更した場合、この宣言以降に定義された変数や定数データのセクション名が変更セクション名になります。ただし、複数行にまたがる宣言の途中に "#pragma SECTION "を記述した場合には、データ定義した変数名の後の区切り文字 " , (カンマ) "、 ";(セミコロン) "、 "=(イコール) "、 "} " までが有効です。

● "#pragma SECTION "は、同じ既定セクションに対して、複数回指定することができます。

●セクション属性および配置属性を指定することはできません。

●コンパイラの "-R "コマンドラインオプション("-R = 既定セクション名=変更セクション名 " 機能:C/C++コンパイラが生成するセクションの名前の変更)との併用について

・#pragma　SECTION宣言は、-Rコマンドラインオプションの影響を受けません。併用した場合、#pragma SECTION宣言が有効な箇所は、#pragma宣言が優先されます。

・#pragma SECTION宣言の変更セクション名が既定セクション名と同じ場合（コンパイラがデフォルトで生成するセクション名に戻した場合)コマンドラインオプション "-R "の指定が有効になります。

[注意事項]

●1つの関数や1つのデータ定義を複数のセクション名に分割することはできません。

●同じ変数に対しては、同じセクション名の#pragma宣言をしてください。一時定義などで異なるセクション名を指定しないでください。

#pragma SECTION

[使用例]　#pragma SECTION は、次のように C/C++ソースに直接記述します。

```
[記述例]
     int a;
     #pragma SECTION B B1
     int b;
     #pragma SECTION B B2
     int c;
     #pragma SECTION P P1

     void func(void)
     {
        ...
     }

変数aは、セクションBに配置されます。(#pragma SECTION宣言より前にあるため)
変数bは、セクションB1に配置されます。
変数cは、セクションB2に配置されます。
関数funcは、セクションP1に配置されます。
```

図 A.7　#pragma SECTION宣言の使用例

#pragma INTERRUPT (#pragma INTF)

割り込み関数の記述機能

[機　能]　**C/C++言語で記述した割り込み処理関数を宣言します。**

[書　式]

#pragma INTERRUPT 割り込み処理関数名 [レジスタ名△レジスタ名△・・・]※
#pragma INTF 割り込み処理関数名 [レジスタ名△レジスタ名△・・・]※

[解　説]

●#pragma INTERRUPTは、指定された関数を割り込み処理関数として宣言します。

●必要に応じて関数の入口/出口で、退避/復帰したいレジスタを指定することが出来ます

●コンパイラは、割り込み処理関数として宣言された関数の入口および出口で、使用するレジスタをスタックに退避および復帰するコードを生成します。また、関数出口ではRTE命令でリターンします。

●レジスタ名を指定しなかった場合(デフォルト設定)、R0～R7レジスタ およびR14レジスタ(割り込み処理関数内で関数呼び出しがある場合)を退避および復帰するコードを生成します。

また、R8～R13は、関数呼び出し規則上(ユーザーズマニュアル「6.3 関数呼び出しとリターンの基本手順」に詳細)保存されるため、デフォルトでは退避および復帰対象にはなっていません(R11～R13のいずれかが、ベースレジスタに指定されていても、同様にデフォルト設定では退避および復帰対象にはなっていません)。以下は、ユーザーズマニュアル「6.3 関数呼び出しとリターンの基本手順」よりの抜粋です。

> (4)レジスタの退避
> R8～R13 のうち、呼び出された関数内で使用するものをスタック上に退避します。退避領域のサイズは、「退避するレジスタ数 ×4 バイト」です。
> リンクレジスタR14 の退避が必要な場合はリンクレジスタも退避されます

●レジスタ名には、次のレジスタを指定することができます。

汎用レジスタ名：R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14

制御レジスタ名：PSW(またはCR0), BPC(またはCR6)

アキュムレータ：ACC0(またはA0), ACC1(またはA1), ACC

レジスタ名を指定した場合、デフォルト設定のレジスタに追加して、指定されたレジスタを退避および復帰するコードを生成します。デフォルト設定以外のレジスタを退避/復帰するには、レジスタ名を指定しなければなりません。複数のレジスタを指定するには、レジスタ名の間を空白(スペース)で区切って指定します。また、レジスタ名の大文字、小文字は区別されません。

(注1)ACC1(またはA1)は、M32Rxコア命令セット用の予約レジスタ名であり、この指定は無視されます。

(注2)ACCが指定された場合、ACC0(またはA0)のみが指定された場合と同じ動作になります。

●#pragma INTERRUPTの宣言は、この宣言より前および後に関数の実体を定義した場合のいずれの場合にも有効になります。

●他のコンパイラとの互換性のため、INTERRUPTのかわりに#pragma INTFと記述することができます。

●-accessオプションにより指定したベースレジスタ(R11～R13)は、#pragma INTERRUPTのレジスタ名として指定がなければ、退避/復帰されません。

※ ”△”は、必須のスペースを示します

#pragma INTERRUPT

[規　定]

- ●void型以外の割り込み処理関数、および、引数を持つ割り込み処理関数を記述した場合、コンパイル時にエラーとなります。
- ●同じ関数に対して2回以上#pragma INTERRUPT宣言した場合、コンパイル時にエラーとなります。
- ●関数名以外を指定した場合、その指定は無視されます(#pragma INTERRUPTは、無効)。
- ●コンパイラは、割り込み処理関数内で割り込みを再び許可するための命令コードは生成しません。
- ●多重割り込みには未対応です(指定関数内で割り込みを再許可するコードは生成しません)。

[使用例]

#pragma INTERRUPT は、次のように C/C++ソースに直接記述します。

```
[記述例]
     #pragma INTERRUPT int_func R12
     extern int counter;
     void int_func(void)
     {
          counter ++;
     }
```

記述例では、#pragma INTERRUPT により、関数 int_func は割り込み処理関数とみなされ、デフォルトレジスタ(R0～R7)と指定レジスタ R12 を復帰および退避するコードが生成されます。また、関数の終了は RTE命令となります。

#pragma keyword asm on (#pragma keyword asm off)

インラインアセンブラ使用宣言機能

[機　能]　**インラインアセンブル機能(ａｓｍ関数)の使用を宣言します。**

[書　式]

```
#pragma keyword asm on
#pragma keyword asm off
```

[解　説]

●#pragma keywoed asm on(off)は、「asm」を通常の識別子ではなく、asm関数の予約語として認識させ、インラインアセンブル機能として使用するための設定です。再びasm を識別子として使用する場合は、#pragma keyword asm off と指定してください。
インラインアセンブル機能については、「7.4 インラインアセンブル機能」を参照してください。

A.4 インライン展開機能

A.4.1 インライン展開機能とは

インライン展開機能は、関数を呼び出すかわりに、呼び出す関数の中身を直接展開する機能です。サブルーチンジャンプ命令(BL)などのオーバーヘッドを省略できるため、インライン展開機能によって通常の関数呼び出しより速度面で有利なコードを得ることができます。

インライン展開機能を使用するためには、関数に対し、inline 記憶クラス指定子を指定して、インライン展開される関数であることを宣言します。

||||注意||||

インライン展開機能を使用した場合、展開する関数のサイズや規模が大きかったり、呼び出す箇所が多い場合に、コードサイズが増大するなどの悪影響があります。インライン展開はその効果を十分確認されたうえでご使用ください。

[形式]

inline△型指定子△関数名(); (△は1つまたはそれ以上の空白類も時を意味します)

例1) 関数宣言
inlineintfunc1(void);
例2) 関数定義
inlinestaticintfunc2(inta)
{
a *= 2;
return a;
}

[コンパイルオプション]

-0<level> (<level>=0 ～ 7)
レベル 4 以上の最適化が有効な場合、インライン展開機能を有効にします。 最適化なし、レベル 0 ～ 3 の場合はインライン展開機能は無効です。 ※ -0space, -0time 指定時は、-0<level> の指定がなくてもデフォルトで -07(<level>=7) が指定されたことになるため、インライン展開機能が有効になります。
-noinline
インライン展開機能を抑止します。このオプションを指定すると、inline キーワードが指定されていても、その関数をインライン展開しません。

[解説]

・inline キーワードを関数宣言または定義の際に指定することができます。inline キーワードは、指定した関数をインライン展開の対象とする関数(inline 関数)であることを指定します。

なお、inline キーワードは関数定義の際に指定してください。関数宣言時に指定しても、同じ関数に対して inline キーワードのついた関数定義がなければ、inline 関数とは、認識しません。

・inline キーワードを指定しても、コンパイル時に -noinline オプションを指定した場合は無視します。(エラーにはなりません。)

・インライン展開は、最適化のひとつとして行うため、-O4 レベルの最適化が有効なときだけ実行します。

・inline 関数の関数定義よりも前(ソース行の上方)にその関数の呼び出しがあっても、インライン展開は行います。

・inline 関数から inline 関数を呼び出してもインライン展開の対象になります。ただし、後述の注意事項(「2. インライン展開が抑止される場合」)に該当する inline 関数を除きます。

[使用例]

ソースファイル

```
% cat sample.c
inline int func (int a)
{
      int b = a * 2;
      return b;
}
int answer;
void foo(int a)
{
      answer = func(a);

}
```

コマンドライン

```
% cc32R -S -O7 sample.c
```

生成コード

```
% cat sample.ms

        .IMPORT $_100_builtin_memcopy
        .EXPORT _answer
          .SECTION P,CODE,ALIGN=4
        .EXPORT $func
$func:
      MV      R0,R4
     SLLI    R0,#1
      JMP R14
       .EXPORT $foo
$foo:
      MV  R0,R4       ;  --- インライン展開部分
     SLLI    R0,#1   ;       (answer = a * 2; と同等)
      LD24    R1,#_answer
      ST      R0,@R1
      JMP R14
          .SECTION B,DATA,ALIGN=4
      .ALIGN 4
_answer:
     .RES.B 4

    .END
```

[注意事項]

1. static宣言されたinline関数

static宣言されたinline関数の呼び出しが全てインライン展開されたか、またはもともとどこからも呼び出されていなかった場合は、コンパイラはその関数の本体を必要ないと判断し、削除します。

※ ただし､アドレス参照されているinline関数は削除しません。

2. インライン展開が抑止される場合

次のようなinline関数の呼び出しはインライン展開しません。

・可変個引数を含んでいる

･関数内にstatic変数を含んでいる

・呼び出し時の実引数の型が仮引数のそれと互換性がない

・呼び出しと関数定義が同じコンパイル単位上に記述されていない

・自分自身を呼び出している

3. デバッグ情報について

インライン展開された部分のデバッグ情報(C/C++ ソースと生成コードとの関連付け情報)は、展開前の inline 関数本体の内容から生成しま

す。例えば、インライン展開された部分をPD32Rなどのデバッガでステップ実行すると、展開前のinline関数本体のソース行が表示されます。 ただし、次のような場合には、該当するデバッグ情報は制限を受けます。

【インライン展開時のデバッガ上での制限事項】
※コマンド名はPD32R,PD32RSIMのものです。

(1) ステップ実行
関数の入り口または出口付近の行が、その関数の呼び出した付近の行と順番が前後する場合があります。

(2) 関数呼出し状況表示 (whereコマンド)
呼び出し経路中の関数のうち、インライン展開された関数は表示されません。

(3) 関数参照 (funcコマンド)
static宣言のあるinline関数のうち､必要なくなったオリジナルの関数は削除されます。この場合は関数を参照するコマンドが使えません。

(4) インライン展開されている関数に対して､次の操作は行わないでください。予期しない結果になる場合があります。
(動作は保証できません。)

- 変数の代入と参照-
インライン展開された関数内の変数(仮引数含む)と、展開先(呼び出し元)とに同じ名前がある場合、デバッガでは両者の区別がつきません。
このような場合、これらの変数の値の変更や参照をしないでください。

-ソース行にブレークポイントをかける-
ソース行がインライン展開されてコードが複数個作られている場合、ソース行ブレークが有効になるコードはそのうち一つだけです。他にはブレークはかかりません。

4．インライン展開の互換性

ANSI-C/ISO C++で未定義である項目を除き、インライン展開した場合とそうでない場合とで、プログラムを実行した結果は一致します。

ただし、次の例のようなANSIで未定義の項目は、インライン展開によって結果が変化する場合がありますので注意が必要です。

例)

関数内のローカルな文字列リテラルは、インライン展開される度に新たな領域が確保されます。このため、ローカルな文字列リテラルを書き変える関数をインライン展開した場合、インライン展開しなかった場合とで結果が異なることがあります。

※文字列リテラルを書き変えることは ANSI-C/ISO C++では未定義で、推奨されません。

[ソースファイル]

```
char c;
inline char * func(void)
{
        char *s = "abcde";
        c = s[2];    /* 書き変える前の s[2] を c にセット */
        s[2] = '-'; /* ここで文字列リテラルを書き変える */
        return s;    /* 現在のポインタを返す */
}

char c1, c2, c3, c4;
void foo(void)
{
        char *s;
        s = func();
        c1 = c;
        c2 = s[2];
        s = func();
        c3 = c;
        c4 = s[2];
}
```

[関数 foo()の実行結果]

```
        func()をインライン展開しなかった場合
        (同じ "abcde" に対する操作)
                c1: 'c'
                c2: '-'
                c3: 'c'
                c4: '-'
        func()をインライン展開した場合
        (異なる "abcde" に対する操作)
                c1: 'c'
                c2: '-'
                c3: '-'
                c4: '-'
```

A.5 M32R/ECU#5 (M32R-FPUコア)対応機能

M32R/ECU シリーズ 32180 および 32182 グループ(以下、M32R/ECU#5 と略します)の拡張命令(FPU 命令など)に対応しました。

・M32R/ECU#5 拡張命令を使ってコード生成ができます (C/C++ コンパイラ)
・M32R/ECU#5 拡張命令を用いたアセンブリ言語を記述できます (アセンブラ)
・アセンブリ言語において浮動小数点定数を記述できます (アセンブラ)

A.5.1 オプション指定

-m32re5	M32R/ECU#5拡張命令を使ってコード生成を行います。また、浮動小数点定数で非正規化数になるものは 0.0 に切り詰めます。
-fminst	FMADD(積和演算命令), FMSUB(積差演算命令)を使ってコード生成します。"-m32re5"オプションが無効な場合は無視します。このオプションを指定しない場合、FMADD,FMSUB命令は使用しません。このオプションを使用する場合は、A.5.3.4 も参照してください。
-float_only	double型を強制的にfloat型と見なします。 "-m32re5"オプションと併用すると、浮動小数点演算全てをFPU命令の対象にすることができます。この機能を利用の際は、「A.5.3.1 "-float_only"オプション指定時の注意」の内容も必ず参照してください。

A.5.2 効果的にFPU命令を利用する

"-m32re5" オプションは、float 型の演算に対して FPU 命令を生成します。
浮動小数点演算で M32R/ECU#5 の能力を最大限に発揮させるためには、次の方法でできるだけ多くの浮動小数点演算がfloat型になるようにしてください。

"-m32re5"オプションが有効でも、double型の演算に対しては従来通りランタイムルーチンを使用します。

[1] double型宣言をfloat型宣言に変更する
ただし、float型でも演算精度および数値の有効範囲に問題がないようにしてください。

[2] 浮動小数点定数に精度を指定する
実定数の末尾にfloat型であることを示す精度指定(f)を行ってください。

正しい記述 1.234f	float型になります。
誤った記述 1.234	精度指定がないので、double型になります。

[3] プロトタイプ宣言で引数をfloat型で宣言する
浮動小数点数値を関数の戻り値や引数に使用している場合は、必ず引数に型宣言のあるプロトタイプ宣言を行ってください。

```
例)
        extern void func1(float);
        void
        func2(float fv)
        {
                func1(fv);
        }
```

※ func1関数のプロトタイプ宣言がない場合は引数はdouble型になります。

[4] 可変個引数を使用しない
可変個引数に渡ったfloat型はdouble型に変換されるため、可変個引数でfloat型を扱わないようにしてください。

[5] 単精度数学関数を用いる
"math.h"に含まれる数学関数を用いている場合は、同じ機能の単精度数学関数を呼び出してください。詳しくは、「第8章 標準ヘッダファイル」を参照してください。

A.5.3 FPU命令利用時の注意事項

A.5.3.1 "-float_only"オプション指定時の注意

"-float_only"オプションは、ソースファイルの変更をしなくても全ての浮動小数点演算にFPU命令を適用できる方法ですが、使用すると次のような互換性の問題が発生しますので、利用の際には十分ご注意ください。

[a] ANSI-C/ISO C++仕様に違反します。
[b] math.hをインクルードしている場合に -float_onlyを指定すると、C標準の倍精度数学関数(cos等)の呼び出しは、同じ機能を持つ単精度数学関数(cosf等)の呼び出しに置き換わります。

```
例) -float_onlyを指定
        #include <math.h>
        ans = cos(rd);
                ↓  -float_only 付きでコンパイル
        ans = cosf(rd);   と等価になる
```

この置換は例えばcosf関数の場合は次のようなマクロにより行っています。

```
#define cos cosf
        :
(他の単精度数学関数についても同様に定義)
        :
```

呼び出す関数名が置換される場合、ロードモジュールには置換前の関数名ではなく置換後の関数名を格納します。このため、デバッガ(M3T-PD32Rなど)やTMインスペクタでは、置換前の関数名を指定したり表示させたりすることはできません(置換後の関数名の表示や指定は可能です)。

[c] double 型や可変個引数(浮動小数点数値が渡される可能性があるもの)を引数にとる関数のインタフェース仕様が変化します。
このため、"-float_only" オプション付きでコンパイルして生成したオブジェクトと以下のいずれかに該当するオブジェクトとの間が、double 型引数か可変個引数を持つ関数でインタフェースをとっている場合、両者をリンクすることはできません。
・"-float_only" オプションを付けずに生成したオブジェクト。
・V.3.10 以前の CC32R で作成したオブジェクト。
・標準ライブラリの "math.h" 以外の関数(printf, scanf 系関数など)。
・標準ライブラリの数学関数を "math.h" のインクルードなしで呼び出している場合。

A.5.3.2 非正規化数の扱い

[a] コンパイル時
"-m32re5" オプションが有効な場合、浮動小数点定数が非正規化数の場合は警告を出したうえで、+0.0 に切り詰めます。

[b] 実行時
FPU 命令の演算中に非正規化数になった場合は、非実装例外が発生する場合があります(FPU 命令の非正規化数の扱いについては、M32R/ECU#5 のソフトウェアマニュアルを参照ください)。

A.5.3.3 丸めモード

CC32Rでは、float演算は丸めモードがRound to nearestが前提で設計されています。
M32R/ECU#5 の丸めモードもこれに合わせるようにしてください。

A.5.3.4"-fminst"オプション

FMADD 命令は、FMUL,FADD 命令を組み合わせて演算を行った場合とで丸めの扱いが異なります(FMSUB と FMUL,FSUB の組み合わせも同様)。

このため、"-fminst"オプションを指定したときとしない時とで結果に差が出る場合がありますのでご注意ください。
(FPU 命令の丸めモードについては、M32R/ECU#5 のソフトウェアマニュアルを参照ください)

A.6 日本語処理機能

* プログラムの文字定数に日本語文字を記述することができます。
* 日本語文字をマルチバイト文字やワイド文字として処理することができます。
* 日本語文字対応の文字コードとして、JIS(EUC, シフトJIS)、Unicode(UTF-8)を使用できます。
* 環境変数の操作により、ソースファイルはシフトJIS、ターゲットではUTF-8といった柔軟な対応も可能です。

A.6.1 文字セットと文字コード

A.6.1.1 日本語文字

JIS X 0201 および JIS X 0208 で定義される文字セットを使用できます。
(ただし、JIS X 0201のラテン文字部分はASCIIとみなして処理します。)

JIS X 0201 ... 一般的に「半角文字」と呼ばれています。
JIS X 0208 ... JIS第1水準､第2水準の漢字文字を含みます。
一般的に「全角文字」と呼ばれています。

本書では、今回追加された、JIS X 0201の右半分(0x80～0xFF)とJIS X 0208を総称して、「日本語文字」と呼びます。

A.6.1.2 文字コード種類

対応する文字コード名を表A.3に示します。
本書では、これらの文字コードを「文字コード名」欄の名称で呼びます。

表A.3 対応文字コード一覧

文字コード名	一般表記	概要	使用可能な箇所		
			C/C++ソース	マルチバイト文字列	ワイド文字列
euc	euc-JP	JIS X 0201,0208のEUC (Extended Unix Code) エンコード	○	○	○
sjis	Shift-JIS	JIS X 0201,0208のシフトJISエンコード	○	○	○
utf8	UTF-8	JIS X 0201,0208をUnicode(UCS-2)に変換し、UTF-8エンコードしたもの	○	○	×
	UTF-16	JIS X 0201,0208をUnicode(UCS-2)に変換し、UTF-16エンコードしたもの	×	×	○

※ Unicode(utf8) 利用の際にはいくつかの注意事項があります。
詳細はA.6.4.7を参照してください。

A.6.1.3 文字コードの指定方法

表A.3 の文字コード名は、コンパイラに対しては環境変数で、標準ライブラリ関数に対してはsetlocale関数(LC_CTYPEカテゴリ)に対して設定します。
なお、環境変数は大文字・小文字を区別しないのに対し、LC_CTYPE に設定する場合は大文字・小文字を区別する点に注意して下さい。

a. 入力時文字コード(コンパイラ)

環境変数 [M32RKIN]

C/C++ ソースファイルを記述する文字コードを指定します。

* 表A.3 の「文字コード名」に示す名前を指定します。
* 文字コード名の大文字・小文字は区別しません。
* この環境変数が未定義の場合は、PC版ではsjis、EWS版ではeucがデフォルトになります。

b. 出力時文字コード(コンパイラ)

環境変数 [M32RKOUT]

文字定数や文字列リテラルに記述した日本語文字をどの文字コードでオブジェクトモジュールに出力するかを指定します。

* 表A.3 の「文字コード名」に示す名前を指定します。
* 文字コード名の大文字・小文字は区別しません。
* この環境変数が未定義の場合は、PC版ではsjis、EWS版ではeucがデフォルトになります。

c. 実行時の文字コード(標準ライブラリ)

ロケール [LC_CTYPE カテゴリ]

* 表A.3 の「文字コード名」に示す名前を指定します。
* 文字コード名の大文字・小文字は区別します。
* LC_ALL カテゴリに文字コード名を指定した場合も LC_CTYPE に文字コードがセットされます。
* LC_ALL カテゴリの初期値は、"C" です。
* この指定が有効な関数は、マルチバイト処理関数、printf 系、scanf 系関数のみです。string.h や ctype.h 関数群には無効ですのでご注意ください。詳細はA.6.4.3 を参照ください。

A.6.2 日本語文字の記述方法

日本語文字は文字定数や文字列リテラルに直接記述できます。
また、これらは L を付けるか付けないかでワイド文字(列)とマルチバイト文字の２種類があり、これらの組み合わせで合計４種類の表現が可能です。

* "漢" ... マルチバイト文字 (char *)
* L'漢' ... ワイド文字 (wchar_t)
* "漢字" ... マルチバイト文字列 (char *)
* L"漢字" ... ワイド文字列 (wchar_t *)

(括弧内は等価な型名を示します。)

【注意】マルチバイト文字を ' 漢 ' のように文字定数('..')で書くことはできません(ワーニングになります)。

A.6.2.1 マルチバイト文字

日本語の１文字を１以上の複数バイトで表現したものです。

* 文字列(char の配列型)で表現します。単一文字であっても " 字 " のように文字列の形式になります。
* マルチバイト文字列(マルチバイト文字の連続)は、１文字分のバイト長が可変のため、文字検索・挿入や削除を行う場合は、文字列の先頭から文字の切れ目を判定しながら行う必要があります。
* 一方で、従来のバイト列を扱う関数(printf や strcpy 等の標準関数)の多くがそのまま使えるといった利点があります。

A.6.2.2 ワイド文字

日本語の１文字を表現します。

* ワイド文字は wchar_t 型で表します。この型は標準ヘッダ stddef.h で定義されています。
* ワイド文字列(ワイド文字の配列)は、マルチバイトに比べてデータ量が多く、バイト列処理関数で扱えないといった短所があります。
* 一方で、要素一つが必ずひとつの文字になるので、文字の切れ目を意識しないで検索・挿入や削除が任意の位置から行うことができるという長所があります。

【注意】現在の CC32R の wchar_t 型は signed short 型と等価です。
ただし、これを前提にしたコーディングはしないでください。

A.6.3 日本語処理機能を使ったプログラミング

日本語処理機能を使ったプログラミングの例を示します。

```
#include <stddef.h>      /* wchar_t 型の定義 */
#include <locale.h>      /* ロケールの制御のため */
#include <stdlib.h>      /* mbstowcs 関数利用のため */
#include <stdio.h>
#include <string.h>

char    str1[] = "CC32R";
char    str2[] = "日本語?の処理"; /* (1) マルチバイト文字列で日本語を記述 */
wchar_t wstr[] = L":新機能";     /* (2) ワイド文字列で日本語を記述 */

#define BUFSIZE  256
#define WBUFSIZE 128

char    buff[BUFSIZE];
wchar_t wbuff[WBUFSIZE];


void    kanjiout(wchar_t wc)       /* ワイド文字出力用関数(ダミー) */
{
        /* wc を表示機器に出力する */
        return;
}

int
main(void)
{
        int     size_wch, i;

*/      /* (3) マルチバイト文字列は、分割などがなければ従来の文字列処理が可能
        strcpy(buff, str1);
        strcat(buff, str2);

        /* (4) mbstowcs 関数の利用に備え、文字コードを設定 */
         setlocale(LC_CTYPE, "sjis");     /* (ShiftJIS の場合) */

        /* (5) ワイド文字列に変換 */
          size_wch = mbstowcs(wbuff, buff, WBUFSIZE);

        /* (6) 変換したワイド文字を１つずつ取り出して出力用関数に出力 */
        /* (ただし、「?」と「の」だけは省く) */
         for (i = 0; i < size_wch; ++i) {
                  if (wbuff[i] != L'?' && wbuff[i] != L'の')
                          kanjiout(wbuff[i]);
        }

        /* (7) 別のワイド文字を１つずつ取り出して出力用関数に出力 */
         for (i = 0; i < sizeof(wstr); ++i) {
                  kanjiout(wstr[i]);
        }

         return 0;

}
```

図A.8　日本語処理機能を使ったプログラミング例

このプログラムは、配列str1, str2, wstr に書かれた日本語を含む文字列を処理し、kanjiout関数に１文字ずつ送り出すというものです。

この例のkanjiout関数はダミーですが、この関数で表示装置への出力を行うなどといった用途に応用することができます。

以下に、プログラムの各部分について説明します。

(1) マルチバイト文字列で日本語を記述

```
char str2[] = "日本語?の処理";
```

日本語文字が含まれている場合でも、従来の文字列リテラルと同様に記述できます。

(2) ワイド文字列で日本語を記述

```
wchar_t       wstr[] = L":新機能";
```

ワイド文字列の場合は、文字列リテラルの先頭に L を付けます。日本語文字も従来の文字も両方記述できます。

(3) マルチバイト文字列のコピー、結合

```
strcpy(buff, str1);
strcat(buff, str2);
```

マルチバイト文字列は、従来の文字列と同様にコピーや連結といった処理が可能です。

(4) 文字コードの設定

```
setlocale(LC_CTYPE, "sjis");
```

次の(5)でマルチバイト文字列からワイド文字列へ変換するために、setlocale関数で
LC_CTYPEカテゴリに文字コードを表すロケール名(表A.3の文字コード名)を設定します。
このとき、LC_CTYPE には、M32RKOUT と同じ文字コードを指定する必要があります。
※ M32RKOUTが未設定の場合は、PC版はsjis, EWS版はeucを指定して下さい。

(5) ワイド文字列へ変換

```
size_wch = mbstowcs(wbuff, buff, WBUFSIZE);
```

上記(3)で作成したマルチバイト文字列を、(6)で文字単位の処理を行うためにワイド文字列に変換します。

(6) 文字を１つずつ取り出し、特定の文字を除去して出力用関数に出力

```
for (i = 0; i < size_wch; ++i) {
   if (wbuff[i] != L'?' && wbuff[i] != L',Ì')
         kanjiout(wbuff[i]);

}
```

ワイド文字列はwchar_t型の配列です。(6)では、文字列の要素を一つずつチェックして、特定の文字(このプログラムでは'?'と'の'の２文字)以外の文字を出力用関数に送り出します。

(7) 別のワイド文字を１つずつ取り出して出力用関数に出力

```
for (i = 0; i < sizeof(wstr); ++i) {
    kanjiout(wstr[i]);

}
```

上記(6)と同様に1つずつ取り出して出力用関数に送り出します。文字チェックのない単純な場合です。

A.6.4 制限事項

A.6.4.1 メッセージ表示

コンパイラのメッセージ中に日本語文字を含む場合、もとの文字ではなく、文字コードを表す8進数を表示します。

例) '漢' に対するワーニングメッセージ(M32RKOUTがsjisの場合)
warning:\0212\0277:unportablecharacterconstant
(\0212\0277は、sjisにおける 漢(0x8aと0xbf)の8進2連表記)
※ 可視文字の違いにより、ホスト環境ごとに表示結果は異なります。

A.6.4.2 マルチバイト文字列処理上の注意

マルチバイト文字列を従来のchar型(配列)として処理する場合、文字の切れ目が一定ではないので、次の操作は文字の切れ目がはっきりしている場所に対してのみ実行して下さい。
文字が分断されたり、正常に検索ができないことがあります。

* 任意の場所で文字列を分割する。
* 任意の場所に文字や文字列を挿入する。
* 文字列の転送を途中で打ち切る。
* 任意の場所から文字を検索する。
* 任意の場所から他の文字との比較を始める。

A.6.4.3 標準関数使用上の注意

次に示す関数は、LC_CTYPE に指定されるロケールに関らず、常に "C" ロケールであるように動作します。このため、これらの関数で日本語を含む文字や文字列を処理する場合は注意が必要です。

(1) 文字列操作関数（string.h に属す関数）

マルチバイト文字の中の日本語がある場合、文字の切れ目を認識しないため、返される結果が期待しないものになる場合があります。

* memchr, strchr, strpbrk, strrchr 関数 ... 検索が正常に行われません。
* strcspn, strspn 関数 ... 正確な文字数が返されません。

(2) 文字操作関数 (ctype.h に属す関数)

日本語文字を指定した場合、正常に処理できません。

例) 次のような呼び出しはしないでください。

```
isprint(L'漢')
isupper(L'Ａ')       /* 全角のA */
```

A.6.4.4 アセンブラの対応

アセンブリ言語で日本語を記述することはできません。

A.6.4.5 関連ツールの対応

CC32R V.3.00 かそれ以前の CC32R に対応するデバッガで、日本語文字に関して特殊な処理を行っていない場合、日本語文字を表示させると文字化けを起こすことがあります。

A.6.4.6 プリプロセッサ出力(-P,-Eオプション)

-P および -E で出力されるプリプロセッサ処理後の出力では、日本語文字は euc に変換されています。

A.6.4.7 Unicodeの制限事項

utf8(Unicode)を使ったプログラミングをされる場合は、次の点に注意してください。

(1) 使用できる文字

utf8指定時においても、使用できる文字は6.1(1)に示した文字のみとなります。
JIS X 0201/0208で定義される日本語以外の文字を含むプログラムは正常に処理できません。

(2) 実装により異なる文字

Unicode対応を謳っていても、ホスト環境(エディタ)やターゲット環境の実装により、同じ文字に対して割り当てている番号が異なる場合があります。
このため、環境変数 M32RKIN,M32RKOUT に utf8 を指定する場合は、これらの環境が想定している Unicode の扱いに注意して使用して下さい。

(3) 類似文字の扱い

入力された文字は内部で一旦 euc に変換しており、Unicode 上で形状の似た文字のいくつかは同じ euc 上の文字に正規化します。この処理によって、入力・出力共に utf8 にしておいても、いくつかの文字はソースに記述したのと異なる文字になる場合がありますのでご注意ください。

A.6.5 日本語処理の補足

A.6.5.1 日本語文字の内部表現

(1) マルチバイト文字列の内部表現

C/C++ ソース上のマルチバイト文字列は、M32RKOUT に従った文字コードに変換された、8bitのテキストストリーム(テキストファイルの中身のイメージ)を並べたものとなります。

例) マルチバイト文字列 char kanji1[] = "**漢字1**"; の、
各文字コードでの等価な記述を示します。

```
/* sjis  */ char kanji1[] = {0x8a,0xbf,0x8e,0x9a,0x31,0};
/* euc   */ char kanji1[] = {0xb4,0xc1,0xbb,0xfa,0x31,0};
/* utf8  */ char kanji1[] = {0xe6,0xbc,0xa2,0xe5,0xad,0x92,0x31,0};
```

(2) ワイド文字の表現

* euc および sjis では、マルチバイト文字列上の2バイト文字をワイド文字に変換すると、最初の1バイトを上位、後の1バイトを下位に変換した値となります。
  1バイト文字の場合は、上位を0拡張した値に変換されます。
* utf8の場合は、UTF-16(UCS-2)に変換した16bit値をワイド文字とします。
* ワイド文字は、文字コードに関らずメモリ上ではBig-Endianで(High-Lowの順に)配置します。

例1) ワイド文字列 wchar_t kanji2[] = L"漢字2"; の、各文字コードでの等価な記述をします。

```
/* sjis  */ wchar_t kanji2[] = {0x8abf,0x8e9a,0x0032,0x0000};
/* euc   */ wchar_t kanji2[] = {0xb4c1,0xbbfa,0x0032,0x0000};
/* utf8  */ wchar_t kanji2[] = {0x6f22,0x5b57,0x0032,0x0000};
```

例2) ワイド文字 wchar_t kanji3 = L'漢'; の、各文字コードでの等価な記述を示します。

```
/* sjis  */ wchar_t kanji3 = 0x8abf;
/* euc   */ wchar_t kanji3 = 0xb4c1;
/* utf8  */ wchar_t kanji3 = 0x6f22;
```

※ char kanji4 = L'漢'; は、オーバーフローになります(文字コードに関らず)。
この場合、kanji4 には 下位１バイト分が(charに変換されて)入ります。

A.6.5.2 標準ライブラリ

(1) ロケール指定(locale.h)

従来の "C" ロケールの他、表９に示す文字コード名をそのままロケール名として指定できます。

(2) ワイド文字変換関数(mbtowc, mbstowcs, mblen, wctomb, wcstombs)

LC_CTYPEがeuc, sjis および utf8 ロケールの場合、それぞれの文字コードに基づき処理します。

(3) printf,scanf系関数

フォーマット指定文字列は、LC_CTYPEに指定された文字コードのマルチバイト文字列とみなして処理します。

(4) その他の標準関数

指定されるロケールに関らずCロケールと同じ動作です。

(5) シフト状態について

標準ライブラリ関数は、シフト状態を記憶していません。呼び出された時点を常に初期シフト状態にして処理を行います。

付録B
C標準ライブラリ関数一覧

本付録では、C 標準ライブラリの全関数の概要を機能別に示します。

■プログラム診断関数

assert	プログラム中に診断機能を付け加えます。

■文字操作関数

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

■数学関数

acos	浮動小数点数の逆余弦を計算します。
asin	浮動小数点数の逆正弦を計算します。

atan	浮動小数点数の逆正接を計算します。
atan2	浮動小数点数どうしを除算した結果の値の逆正接を計算します。
ceil	浮動小数点数の小数点以下を切り上げた整数値を求めます。
cos	浮動小数点数のラディアン値の余弦を計算します。
cosh	浮動小数点数の双曲線余弦を計算します。
exp	浮動小数点数の指数関数を計算します。
fabs	浮動小数点数の絶対値を計算します。
floor	浮動小数点数の小数点以下を切り捨てた整数値を求めます。
fmod	浮動小数点数どうしを除算した結果の余りを計算します。
frexp	浮動小数点数を(0.5, 1.0)の値と 2 の累乗の積に分解します。
ldexp	浮動小数点数と 2 の累乗の乗算を計算します。
log	浮動小数点数の自然対数を計算します。
log10	浮動小数点数の 10 を底をする対数を計算します。
modf	浮動小数点数を整数部分と小数部分に分解します。
pow	浮動小数点数の累乗を計算します。
sin	浮動小数点数のラディアン値の正弦を計算します。
sinh	浮動小数点数の双曲線正弦を計算します。
sqrt	浮動小数点数の正の平方根を計算します。
tan	浮動小数点数のラディアン値の正接を計算します。
tanh	浮動小数点数の双曲線正接を計算します。

■プログラムの制御移動関数

longjmp	setjmp関数で退避していた関数の実行環境を回復し、setjmp関数を呼び出したプログラムの位置に制御を移動します。
setjmp	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。

■可変個の実引数アクセス関数

va_arg（マクロ）	可変個の引数から順に引数を取り出します。
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します。
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。

■入出力関数

clearerr	ストリーム入出力用ファイルのエラー状態をクリアします。
fclose	ストリーム入出力用ファイルをクローズします。
feof	ストリーム入出力用ファイルが終わりであるかどうかを判定します。
ferror	ストリーム入出力用ファイルがエラー状態であるかどうかを判定します。
fflush	ストリーム入出力用ファイルの内容をファイルへ出力します。
fgetc	ストリーム入出力用ファイルから１文字入力します。
fgetpos	現在のファイルストリームの位置を求めます。
fgets	ストリーム入出力用ファイルから文字列を入力します。
fopen	ストリーム入出力用ファイルを指定したファイル名によってオープンします。
fprintf	書式に従ってストリーム入出力用ファイルへデータを出力します。
fputc	ストリーム入出力用ファイルへ１文字出力します。
fputs	ストリーム入出力用ファイルへ文字列を出力します。
fread	ストリーム入出力用ファイルから指定した記憶域にデータを入力します。
freopen	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。
fscanf	ストリーム入出力用ファイルからデータを入力し、書式に従って変換します。
fseek	ストリーム入出力用ファイルの現在の読み書き位置を移動します。
fsetpos	ファイルストリーム上の位置を変更します。
ftell	ストリーム入出力用ファイルの現在の読み書き位置を求めます。
fwrite	記憶域からストリーム入出力用ファイルへデータを出力します。
getc	ストリーム入出力用ファイルから１文字入力します。
getchar	標準入力（stdin）から1文字入力します。
gets	標準入力（stdin）から文字列を入力します。
perror	標準エラー出力（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。
printf	データを書式に従って変換し、標準出力(stdout)へ出力します。
putc	ストリーム入出力ファイルへ１文字出力します。
putchar	標準出力（stdout）へ1文字出力します。

puts	標準出力（stdout）へ文字列を出力します。
remove	ファイルを削除します。
rename	ファイル名を変更します。
rewind	ストリーム入出力用ファイルの現在の読み書き位置をファイルの先頭に移動します。
scanf	標準入力(stdin)からデータを入力し、書式に従って変換します。
setbuf	ストリーム入出力用のバッファをユーザープログラム側で定義して設定します。
setvbuf	ストリーム入出力用のバッファをユーザープログラム側で定義し、さらにバッファの使用方法を設定します。
sprintf	データを書式に従って変換し、指定した領域へ出力します。
sscanf	指定した記憶域からデータを入力し、書式に従って変換します。
tmpfile	テンポラリファイルを生成します。
tmpnam	既存しないテンポラリファイル名を生成します。
ungetc	ストリーム入出力用ファイルへ１文字を戻します。
vfprintf	可変個のパラメー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へ出力します。
vprintf	可変個のパラメータリストを書式に従って、標準出力へ出力します。
vsprintf	可変個のパラメータリストを書式に従って、指定した記憶域へ出力します。

■標準処理関数

abort	プログラムの実行を強制終了します。
abs	int 型整数の絶対値を計算します。
atexit	プログラムが正常終了した時に呼び出される関数を登録します。
atof	数を表現する文字列をdouble型の浮動小数点数値に変換します。
atoi	10進数を表現する文字列を int 型の整数値に変換します。
atol	10進数を表現する文字列をlong 型の整数値に変換します。
bsearch	２分割検索を行います。
calloc	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。
div	int 型整数の除算の商と余りを計算します。

exit	プログラムを終了します。
free	指定された記憶域を開放します。
getenv	環境変数の内容を取得します。
labs	long 型整数の絶対値を計算します。
ldiv	long 型整数の除算の商と余りを計算します。
malloc	記憶域を割り当てます。
mblen	マルチバイト文字のバイト数を求めます。
mbstowcs	マルチバイト文字列をワイド文字列に変換します。
mbtowc	マルチバイト文字をワイド文字に変換します。
qsort	ソートを行います。
rand	0 から RAND_MAX の間の擬似乱数整数を生成します。
realloc	記憶域の大きさを指定された大きさに変更します。
srand	rand 関数で生成する擬似乱数数列の初期値を設定します。
strtod	数を表現する文字列を double 型の浮動小数点数値に変換します。
strtol	数を表現する文字列をlong 型の整数値に変換します。
strtoul	数を表現する文字列をunsigned long型の整数値に変換します。
system	コマンド文字列をコマンドプロセッサによって実行されるホスト環境に渡します。
wcstombs	ワイド文字列をマルチバイト文字列に変換します。
wctomb	ワイド文字をマルチバイト文字に変換します。

■文字列操作関数

memchr	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。
memcmp	二つの記憶域を比較します。
memcpy	複写元の記憶域の内容を指定した大きさ分、複写先の記憶域に複写します。
memmove	複写元の記憶域の内容を指定した大きさ分、複写先の記憶域に移動します。
memset	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。
strcat	文字列の後に、文字列を連結します。
strchr	文字列において、指定された１文字が最初に現われる位置を検索します。

strcmp	二つの文字列を比較します。
strcoll	現在のロケールに基づいて指定された２つの文字列を比較します。
strcpy	複写元の文字列の内容を、複写先の記憶域にnull文字も含めて複写します。
strcspn	文字列において、指定外の文字が先頭から何文字続くかを求めます。
strerror	エラーメッセージを返します。
strlen	文字列の長さを計算します。
strncat	文字列の後に、文字列を指定した文字数分連結します。
strncmp	二つの文字列を指定された文字数分まで比較します。
strncpy	複写元の文字列を指定された文字数分、複写先の記憶域に複写します。
strpbrk	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。
strrchr	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	文字列において、指定された文字が先頭から何文字続くかを求めます。
strstr	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	文字列をいくつかのトークン（字句）に切り分けます。
strxfrm	現在のロケールに基づいて文字列を変換します。

■ロケール操作関数

localeconv	lconv型構造体を初期化します。
setlocale	ロケール情報の設定、検索をします。

■日付・時刻操作関数

asctime	tm型構造体で示される日時をテキスト文字列の形式に変換します。
clock	プログラムの実行時間を求めます。
ctime	time_t型の暦時間をテキスト文字列の形式に変換します。
difftime	指定された２つの時間差を求めます。
gmtime	暦時間を世界標準時に変換します。
localtime	暦時間を現在の現地時間に変換します。
mktime	tm型構造体で示される日時を暦時間に変換します。
strftime	tm型構造体で示される日時を書式に従って変換します。
time	現在の暦時間を求めます。

■シグナル処理関数

raise　　プログラムに対してシグナルを送ります。

signal　　シグナルに応答する関数を設定します。

■終了処理関数（non-ANSI）

_100_C_action_atexit　　ユーザー終了処理関数（atexit 関数で登録された関数）を実行します。

_100_C_get_exit_code　　exit 関数の終了コードを取り出します。

■浮動小数点特殊数値判定関数（non-ANSI）

浮動小数点型(float および double)の数値が、ゼロ(0.0)・無限大(∞)・非数(NaN)のいずれかの特殊数値に該当するかどうかを判定する関数です。

浮動小数点数値の演算（浮動小数点を用いた式や関数呼び出し）において、事前に演算に入力する値が特殊数値でないかのチェックや、演算の結果得られる数値が特殊数値になっていないかを判定するのに用いることができます。

表B.1　浮動小数点特殊数値判定関数

判定対象の特殊数値		判定関数（C言語のプロトタイプ宣言の形式）	
		double型	float型
無限大	±∞	int isinf(double)	int isinff(float)
ゼロ	±0.0	int iszero(double)	int iszerof(float)
非数(Not a Number)	NaN	int isnan(double)	int isnanf(float)

これらの関数は、入力された数値が対象となる特殊数値に該当する場合は1を、該当しない場合は0をリターン値として返します。

これらの判定関数を用いる場合は、次のいずれかのヘッダファイルをインクルードする必要があります。必要に応じてどちらか一方のヘッダを選択してください。

mathf.h　float型の判定関数を用いる場合
math.h　double型またはfloat型の判定関数を用いる場合

なおmath.hはmathf.hの機能を包含していますので、math.hをインクルードしている場合は、mathf.hをインクルードする必要はありません。

使用例） 入力が無限大や非数のときに -1.0 を返す関数

```
#include <mathf.h>   /* <math.h>でも可 */
float
func(float fin)
{
    if (isinff(fin))
        return -1.0f;  /* 無限大の場合 */
    if (isnanf(fin))
        return -1.0f;  /* NaNの場合 */
    return fin * 2.0f;  /* それ以外の正常な場合 */
}
```

付録C
制限事項

本付録では、現在確認されている CC32R の制限事項を示します。

■ デバッグ情報のないファイルを得るには

C/C++ コンパイラ cc32R、アセンブラ as32R、リンカ lnk32R において、常にデバッグ情報が有効になるように変更になっているのに伴い、cc32R，as32R，lnk32R は、常に従来の -g オプションが付いた状態で処理を行います。

なお、これらのデバッグ情報の出力は、オプション等では抑止できません。

従来と同様のデバッグ情報が無い出力を得るためには、デバッグ情報除去ツール strip32R をご利用ください。strip32R は、リンカが出力するロードモジュールファイルだけでなく、コンパイラやアセンブラが出力するオブジェクトモジュールファイルも処理することができます。すなわち、cc32R，as32R，lnk32R の実行のすぐ後に、それぞれの出力ファイルを直接 strip32R で処理するようにすれば、従来の V.4.10 以前の CC32R と等価な動作にすることができます。

[strip32R の使用例　(% はプロンプトを表します)]

通常使用

cc32R，as32R，lnk32R の出力ファイルに対して、その都度適用できます。
リンク前のオブジェクトモジュールファイルでも、リンク後のロードモジュールファイルでも同様に処理できます。

```
% cc32R -c -o sample1.mo sample1.c
% strip32R sample1.mo
% as32R sample2.ms
% strip32R sample2.mo
% lnk32R -o sample.abs sample1.mo sample2.mo
% strip32R sample.abs
```

複数指定

コンパイルとアセンブルを行った後に、まとめて処理できます。

```
% cc32R -c sample1.c sample2.c sample3.c
% cc32R -c sample4.c
% as32R -c sample5.ms
% strip32R sample1.mo sample2.mo sample3.mo sample4.mo sample5.mo
```

※ ワイルドカードも使用可能です。

```
% strip32R *.mo
```

■ ベースレジスタ機能と標準ライブラリを併用する場合の注意事項

[ベースレジスタ機能使用上の注意事項の補足]

次のいずれかのオブジェクトファイルの組み合わせでリンクを行うことは推奨しておりません。(ユーザーズマニュアル<<C コンパイラ編>>「A.1.6 ベースレジスタ機能の制限事項」参照)

(1) ベースレジスタ機能を有効にして作成したオブジェクトファイルと、無効にして作成したオブジェクトファイル

(2) それぞれ、異なるアクセス制御ファイルを用いて作成した複数のオブジェクト

[ベースレジスタ機能と標準ライブラリ併用時の注意事項]

製品添付の標準ライブラリは、ベースレジスタ機能を無効にして作成しています。このため、ベースレジスタ機能を用いたユーザプログラムと標準ライブラリは、上記(1)の組み合わせになります。

このような場合、標準ライブラリ関数処理中は、ベースレジスタがベースアドレスを保持していません。関数処理後はベースレジスタはベースアドレスに戻りますが、次のような場合はベースレジスタの値が不正になります。

(1) 割込み処理ルーチン
標準ライブラリ関数実行中に割り込みがかかることがあるため、割り込み処理ルーチン中ではベースレジスタの値は不定と考える必要があります。

(2) 特定の標準ライブラリ関数(qsort,bsearch 等)から呼び出されるユーザ関数

[対策]

ベースレジスタ機能と標準ライブラリを併用する場合は、次の(1),(2)のうちいずれかの方法を用いて対策を行ってください。

(1) ユーザプログラムと同じアクセス制御ファイルを用い、ベースレジスタ機能を有効にした標準ライブラリを再構築し、現在の標準ライブラリと差し替えてください。

(2) 割込み処理ルーチンおよび、特定の標準ライブラリ関数(qsort,bsearch等)から呼び出されるユーザ関数を、ベースレジスタ機能を無効にして再コンパイルしてください。

■ M32R/ECU シリーズでの整数剰除算の問題に対する対応方法について

M32R/ECU シリーズにおいて、整数剰除算命令（DIV,DIVU,REM,REMU の各命令。以下 DIV 系命令と略します）でゼロ除算を行った場合、その直後に実行した命令の実行結果が不正になる問題があります。詳しくは テクニカルニュース No.M32R-50-0301「M32R/ECU シリーズ 0 除算実行時の注意事項」を参照ください。
以下に、CC32R での対応方法とゼロ除算問題抑止オプション -zdiv について説明します。

[対応方法]

C/C++ 言語プログラム、アセンブリ言語プログラムの場合

(1)論理的にゼロ除算が行われないないように、プログラミングしてください
CC32R は、C/C++ 言語の整数除算(/ および /=) と 整数剰余(% および %=) に対して DIV 系命令を生成しますので、これらの演算子の除数が 0 にならないようにしてください。
アセンブリ言語では、DIV 系命令の第 2 レジスタが 0 にならないようにしてください。

(2)(1)が確実にできない場合は、-zdiv オプションを指定して再コンパイルあるいはアセンブルしてください。

標準ライブラリを用いている場合

標準ライブラリは本問題に対応済みですので、DIV 系命令で 0 除算を行っても問題は発生しません。
なお、CC32R V.4.10 Release 1で用意しておりましたゼロ除算対策版のライブラリ(m32RcRZ.lib, m32RcRZM.lib, m32RcRZL.lib)は、本バージョンでは通常版(m32RcR.lib, m32RcRM.lib, m32RcRL.lib)に統合しております。
このため、CC32R V.4.10 Release 1 を使用されていた方で、リンクする標準ライブラリにゼロ除算統合版を指定なさっていた場合は、それらは通常版に戻してご使用ください。

標準ライブラリ以外のライブラリを用いている場合

お客様で作成されたライブラリまたは、添付の標準ライブラリソースを再構築して作成したライブラリをご使用の場合は、-zdivオプションを付けてライブラリを再構築してください。

[-zdiv オプションの解説]

cc32R でコンパイルする場合

-zdiv オプション付きでコンパイルすると、コード生成で DIV 系命令を出力するときは、その直後に NOP 命令を挿入します。
また、asm 関数内に DIV 系命令がある場合もその直後に NOP 命令を挿入します。
なお、-S および -CS オプションと本オプションを同時に指定した場合、asm 関数内のコメントは除去し、記述も全て大文字に変換したうえでアセンブリソースを出力しますのでご注意ください。
cc32R にアセンブリソース(拡張子 .ms)を入力する場合は、as32R でアセンブルする場合と同様です。

as32R でアセンブルする場合

DIV系命令を持つアセンブリコードを -zdiv オプション付きでアセンブルすると、その DIV 系命令の直後に NOP 命令を挿入します。
ただし、すでに DIV 系命令の直後に NOP 命令が存在する場合を除きます。
これは、この DIV 系命令と NOP 命令のあいだに次のものが存在していない場合です。すなわち、これらのものが DIV 系命令と NOP 命令の間に存在しているときは、NOP 命令を DIV 系命令の直後に追加します。

(1) ラベル
(2) NOP以外の一般命令
(3) 領域に影響を及ぼす擬似命令(下記)

```
.ALIGN    .DATA    .DATAB    .END      .FDATA
.FDATAB   .FRES    .RES      .SDATA    .SDATAB
.SECTION
```

■ 可変引数関数をポインタ間接で呼び出す場合の問題

プロトタイプ宣言がない関数へのポインタ変数を用いて、間接的に可変引数の関数を呼び出すような場合、生成されるコードが正しい動作をしません。

【記述例】

```
#include <stdio.h>
   int (*funcptr)() = printf;
   int main(void) {
      (*funcptr)("calling printf with %d\n", 1);
   }
```

【回避方法】

関数へのポインタ変数にプロトタイプ宣言をつけてください。

```
   #include <stdio.h>
   int (*funcptr)(const char *,...) = printf;
   int main(void) {
      (*funcptr)("calling printf with %d\n", 1);

   }
```

■ コードセクション中のデータ定義について

アセンブラは、コードセクション中にデータ（もしくは空き領域）がある場合に注意を促すため、ワーニング(warning: caution! there are some data in code section)を出力します。
データは、データセクションに記述して頂くことを推奨いたします。
なおこのワーニングは、オプション(-warn_suppress_code_data)で抑止することができます。

■ マクロボディ内におけるプリプロセッサ変数の使用

マクロボディ内において、マクロコール直後の行の第１カラムからプリプロセッサ変数を記述した場合、マクロコールにおいてプリプロセッサ変数が正しく展開されない場合があります。

【記述例】

```
    .macro INST_MACRO
     MOV        #0,R0
     .endm
     .macro LABEL_MACRO label
     INST_MACRO
\label:                  ← プリプロセッサ変数を第１カラムから記述。
     .endm
     .section P,code,align=2
     LABEL_MACRO L1      ← 正しく展開されない。
     LABEL_MACRO L2      ← 正しく展開されない。
     .end
```

【回避方法】

マクロボディ内におけるプリプロセッサ変数は、第２カラム以降から記述してください。

■ 500行以上の関数を持つプログラムをコンパイルする場合

500 行以上の大きな関数を持つ C/C++ ソースプログラムをコンパイルする場合、”Out of memory ”のエラーが発生する場合があります。
このような場合は、この関数を分割して小さくしてください。

■ 関数引数の設定規則変更に伴う注意事項

V.3.00 Release 1 以降、関数の引数はレジスタ渡しになっています(従来の -RBPP オプション相当)。従って、V.2.10 Release 1 以前の CC32R 用に作成された次のようなプログラムと組み合わせて使用することはできません。それぞれ次のように対応してください。

(1)引数をスタック渡しする C/C++ 言語プログラム

V.2.10 Release 1以前のコンパイラを用いて、-RBPP オプション指定なしでコンパイルされたオブジェクト及びライブラリが対象となります。

【対応】

V.3.00 Release 1で再コンパイルしてください。

(2)引数をスタック渡しにするアセンブリプログラム

C/C++ 言語の関数を呼び出すプログラム、および、C/C++ 言語の関数から呼び出されるアセンブリ言語プログラムのうち、引数をスタック渡しするアセンブリ言語プログラムが対象になります（スタートアッププログラム、低水準ライブラリ関数を含　　む)。

【対応】

* V.3.00 Release 1の関数引数の設定規則（ユーザーズマニュアル《C/C++コンパイラ編》の「C/C++呼び出し規則」の章を参照ください）に従って、アセンブリ言語プログラムを変更してください。

* レジスタ渡しで呼び出しを行う関数名は、オブジェクトモジュール内では、C/C++プログラムにおける関数名の先頭に、アンダースコア（_）ではなく ドルマーク（$）が付加されます。アセンブリ言語中の関数名はこれに従った名前に 変更してください。

上記 (1) や (2)に該当するプログラムを、【対応】を行わずにCC32R V.3.00 Release 1用に作成されたプログラムとリンクすると、"external symbol not defined" のエラーとなります。

■ ライブラリに対してC++のデバッグをする場合の注意

ライブラリ内のオブジェクトモジュールに対してC++ソースデバッグをするには、そのオブジェクトモジュール（拡張子 .ml）を作成したのと同じディレクトリにライブラリファイルを配置する必要があります。

付録D
コンパイル　注意事項

D.1. リンクついて

オブジェクトファイルのリンクだけを行う場合であっても、コンパイルドライバ cc32R を通してリンカを起動(-mklib オプションを使用)します。これはユーザがリンカを直接呼び出すと、プレリンカが起動されないため正しくリンクできるオブジェクトが作成されないからです。

Makefile 中であっても、リンカだけを呼び出す記述はしないでください。

D.2. ライブラリ作成について

ライブラリファイルを作成する場合も、プレリンカを起動する必要があります。このためコンパイルドライバ cc32R を通してライブラリアンを起動（-mklib オプションを使用）します。

Makefile 中であっても、ライブラリアンだけを呼び出す記述はしないでください。

D.3. ディレクトリの移動について

コンパイル後にオブジェクトファイル等を別ディレクトリに移動すると、正しくリンクできません。これは、コンパイル時のパス情報をプレリンカでの処理用に保持しているためです。

D.4. オブジェクトファイル名の変更について

リロケータブルオブジェクトファイルの拡張子を変更すると、プレリンカが正しく動作しません。プレリンカは、リロケータブルオブジェクトファイルの拡張子を取り除いた部分の名前から外部情報ファイルを検索します。このため -o オプション等でリロケータブルオブジェクトファイルの名称を変更すると、最終的に正しくリンクできなくなります。

D.5. C++用の特殊なファイルに関する注意事項

C++ のコンパイルや、リンクおよびライブラリの作成の際は、C++ 言語の各種機能を実現するため、次のような拡張子を持つファイルを扱います。

～ .ic
～ .ti
～ .et
～ .ii
～ .dw2
～ .dws

これらのファイルは、～ .mo と同じディレクトリ上に配置され、cc32R コマンドにより、C++ 言語の処理のために必要に応じて作成、参照、更新および削除されます。

C++ 機能では非常に重要なファイルですので、コンパイル・リンク・デバッグの一連の手順が完了するまでは、次のいずれの操作も行わないようにご注意ください。

(1) C++ 情報ファイルのファイルの更新や削除
(2) 対応する～ .mo ファイルと異なるディレクトリに C++ 情報ファイルを移動
(3) C++ 情報ファイルと同じ名前のファイルを作成

D.6. 外部シンボル名について

関数名、静的データメンバから生成する外部シンボル名は、C++ 言語のコンパイルのときには、C 言語とは異なる規則で変換を行います。コンパイラが生成した外部シンボル名を知るには、次の 2 つの方法があります。

(1) コンパイル時にアセンブリ言語ファイルを生成するオプション(-S,-CS,-cs)を指定し、出力された外部シンボル名を確認する。

(2) 生成されたオブジェクトモジュールまたはロードモジュールから、map32R にオプション -mangle を付けてマップ情報を生成する。

C++ の関数を 「extern "C"」 を付けて関数定義を行えば、外部シンボル名は C 言語の関数と同様の生成規則になります。

ただし、その関数は、多重定義できなくなります。

D.7. 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

D.7.1 関数の原型宣言

関数を使用する前に原型宣言が必要です。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()
{
```

```
extern void func1(int);
void g()
{
```

D.7.2 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;
```

D.7.3 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
```

MEMO

M32Rファミリ用
C/C++**コンパイラパッケージ** V.5.00
C/C++**コンパイラユーザーズマニュアル**

発行年月日 2005**年**07**月**15**日** Rev.1.00

発行 **株式会社 ルネサス テクノロジ 営業企画統括部**
〒100-0004 **東京都千代田区大手町**2-6-2

編集 **株式会社 ルネサス ソリューションズ ツール開発部**

M32R ファミリ用
C/C++ コンパイラパッケージ V.5.00
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

RJJ10J0904-0100